Quick Sort

Quicksort

- Merge sort does split, sort, merge, but the split part is done naively
- Quicksort does the splitting part carefully, and so doesn't need the merge step
- Uses a new helper function called “partition”

Partition

- Partition takes an array and a “pivot value” (in quicksort this is one of the values in the array) as parameters
- It moves elements in the array so that all elements smaller than the pivot come before it in the array and all elements larger come after
- It returns the index of the pivot after partitioning. In general, we don't know where the pivot will end up until we finish partitioning, and the caller needs to know where it is

Quicksort

- Partition the array around a pivot (we'll see a few ways to pick pivots)
- Sort the part before the pivot
- Sort the part after the pivot

Implementing Partition

- Can be done “in place” (ie no need to allocate another array, like merge sort does)
- Several possible implementations, we'll look at a “two sided” version

Partition

- Details intentionally left vague...

```
Swap the pivot to the end of the array
left = 0, right = just before the end
while not done:
    while arr[left] < pivot: increment left index
    //left points to an element on wrong side of the pivot
    while arr[right] > pivot: decrement right index
    //right points to an element on the wrong side of the pivot
    if left is before right:
        swap left/right
swap the pivot from the end into the correct spot
```

Analyizing Complexity

- Partition is O(N) since it touches all the elements
- So cost of quicksort is O(N) + O(sort left) + O(sort right)
- A little bit tricky to analyze since we don't know how big left/right are!
- Not even obvious what best/worse cases are!

Best Case: even splits

- In the best case, our pivot is the median value and left/right sides each have N/2 elements
- In this case analysis is similar to mergesort (partition ~= merge, recursive sorts are the same in both analyses)
- Gives us O(N lg N) runtime
- If pivots are “usually” 25/75 splits or “better” we get (N lg N) complexity. Take an algorithms class for the details

Worst Case: bad splits

- In the worst case, our partition doesn't divide the array at all
- An example is when the left half is empty and the right half is N −1 elements
- In this case the cost at the “top level” is N, the next level is N - 1, then N −2, etc
- So our overall run time is N + (N −1) + (N −2) + ... 1
- Which we know is $O(N^2)$
- This is a very common input!!!

Choices of Pivot

- First element (uh oh!)
- Last element (uh oh!)
- Middle element (probably OK)
- Random element (almost certainly OK)
- Is there a choice that's guaranteed to be good enough? Stay tuned

Partition with a predicate

- Partition is a super userful even outside of its use in quicksort
- With a minor tweak we can make it even more widely useful
- As we implemented it all the stuff on the left side has `cmp(a[i], pivot) ≤ 0` and all the stuff on the right has `cmp(a[i], pivot) > 0`
- We can replace the comparison with any boolean function and then we'll order the array so all the stuff which the function returns false for is in the left part, and stuff it returns true for is in the right half
- We call this a `predicate` function: `boolean predicate(E elem)`

Implementation advice

- Quicksort is really easy: 3 lines + a base case
- Partition is tricky!!
- Good news, you can easily write tests for partition on its own!
- I strongly recommend doing that before you even write quick sort! Try partition on a variety of tricky cases such as:
  - 1 and 2 element lists
  - lists of all equal elements
  - when the pivot is the first element
  - when the pivot is the last element