20190913

Divide and Conquer Algorithms

The Divide and Conquer Paradigm

```
To solve a problem of size n:
    If n is "small":
        solve the problem directly
    else:
        Subdivide the problem into two or more (usually disjoint) subproblems
        Solve each of the subproblems recursively
        Combine the subproblem solutions to get the solution to the original
            problem
```

Examples of D&C algorithms are familiar to everyone who has studied computing: binary search, Quicksort, and MergeSort are classic examples.

Today we looked at MergeSort as an example of both the D&C paradigm, and also an example of the last of the four types of recursive algorithm we looked at yesterday.

To set the stage: putting a set of objects in order may seem like a pedestrian task, but it is profoundly useful – as we will see, there are many problems that are superficially unrelated to putting things in order, but for which a preliminary stage of sorting the data can greatly accelerate the algorithm that actually solves the desired problem.

The problem of sorting a set is also of great theoretical interest. It was one of the first problems for which it was possible to prove a non-trivial lower bound on the computational complexity. Here's that result in a nutshell: consider the range of all possible sorting algorithms that are based on comparing elements of the set being sorted. As you might imagine, there are infinitely many such algorithms (really? Yes, we can prove that!) and humans have only examined a very small number of them. But we can prove that there cannot exist a comparison-based sorting algorithm that belongs to any complexity class lower than O(n*log n).

But naïve sorting algorithms (such as SelectionSort, outlined just below) run in O(n^2) time.

```
SelectionSort(A):
    # A is an array containing n elements.  I don't like 0-based addressing so I
    # am going to assume that the first element is A[1] and the last is A[n]
    for i = n down to 1:
        find the largest value in the data range A[1] to A[i]
        Let A[p] be the largest value found
        Swap A[p] and A[i]
```

There is an implicit loop in the line "find the largest value ..." since we do that by looking at each value in that range. We look at n elements, then n-1, then n-2 etc. so the total number of times we look at a data element is $n + n - 1 + n - 2 + \ + 1 = \frac{(n+1)*n}{2}$ which is clearly in O(n^2)

So the question facing algorithm designers was this: "We know we can't find a sorting algorithm that runs in better than O($n * \log n$), but the sorting algorithms we have exhibit running time in O(n^2). Can we close the gap between the lower bound and the actual running time?"

(Actually, MergeSort has been known for ages – nobody even knows who first came up with the algorithm. So MergeSort probably predates the whole study of computational complexity and it is misleading of me to suggest that it was discovered as part of a quest to answer the question just posed. In fact, the chronological sequence was more like "MergeSort runs in O($n * \log n$) time – can we do better?" "Oh, O($n * \log n$) is a lower bound on all comparison-based sorting algorithms. MergeSort has optimal complexity." However, this type of question is unresolved regarding *other* problems and represents a core theme in the study of algorithms. This seemed like a good time to introduce it.)

But how do we know MergeSort has complexity in O(n*\log n)?

Here's the algorithm in very high-level pseudo-code.

```
MergeSort(A):
    # A is an array as specified above for SelectionSort
    if  A has <= 5 elements:  # see the note below regarding the magic number 5
        sort A using any method
    else:
        MergeSort(left half of A)
        MergeSort(right half of A)
        merge the two sorted halves together into a full sorted set
```

When A is small (in the code above, I have used 5 as the cut-off point for "small") we can use any sorting algorithm we like, even a relatively slow one like SelectionSort. This is ok because with <= 5 elements, the time required for the sorting process is bounded above by the time it takes to sort exactly 5 elements – which is constant. Does the cut-off point have to be 5? Absolutely not. You could make it 1 … or 1000 … without changing the complexity of the algorithm. If you are doing a production-quality implementation of MergeSort you will need to determine the best value here. With a very low value you can incur extra time through many nested recursive calls, whereas with a very high value the large subsets being sorted by SelectionSort may overwhelm the savings accomplished by the divide and conquer approach.

Now what about the final line "merge the two sorted halves together"?

We want to find the smallest value and put it into the first position in A. After the Left Half and the Right Half are sorted, the smallest value over all must either be the first one in the Left Half or the first one in the Right Half. We just compare these two and pick the smaller. Now we need to find the next smallest. Again it is either the smallest remaining value in the Left Half or the smallest remaining value in the Right Half.

We always need exactly one comparison to find the next value we need to build the sorted version of A. Since A has n elements, the merge phase is in O(n)

So the recurrence relation is this:

$$T(n) = c_1 \text{ if } n \leq 5$$

$$T(n) = c_2 + c_3 * n + 2T\left(\frac{n}{2}\right) \text{ if } n > 5$$

And Lo and Behold … this is exactly the fourth recurrence relation we looked at … so we know without any further effort that the running time of MergeSort is in O($n * \log n$)

A question was asked in class regarding dividing A into more than two parts to reduce the actual (real world) running time. For example we could divide A into Left Third, Middle Third and Right Third. We would sort each third independently and then merge them back together. Choosing the correct value at each stage of the merge would require two comparisons (in class I mistakenly said “three comparisons”) so the merging would be slower. However the number of recursive calls would be reduced because we approach the cut-off point for ending the recursion faster. The recursive part of the recurrence relation would look like this:

$$\mathrm{T}(n) = c_2 + c_3 * n + 3 * T\left(\frac{n}{3}\right)$$

which still works out to be in O($n * \log n$) so it is in the same complexity class as the original MergeSort.

The actual coefficients in the “split in half” and “split in thirds” (and by extension, the “split in quarters”, “split in eighteenths”, “split in ninety-ninths” etc) will all be different, but every one of these variations is in O($n * \log n$). To determine if “split in thirds” is actually faster or slower than “split in half” would require careful operation counting and some amount of experimentation.