CS 6363.003.21S Lecture 3—January 26, 2021

Main topics are #divide-and-conquer including #example/mergesort and #example/quicksort.

Mergesort

- Last week, we reviewed induction and its algorithmic counterpart, induction. This week, we'll start a section on a particular recursive strategy called *divide-and-conquer*.
- Before discussing *what* divide-and-conquer generally means, I'll show a couple examples.
- We'll start with at an algorithm called mergesort proposed by John von Neumann around 1945. You may have seen it before, because it is one of the earliest algorithms designed for general purpose computers and still gets used in practice today. We're going over it again now, because it's the perfect example of divide-and-conquer.
- Let's say we're given an array A[1.. n] of things we want to sort (numbers, letters, shoes, whatever) where we can compare any two elements. Our goal is to rearrange the elements of A[1 .. n] so that A[1] ≤ A[2] ≤ … ≤ A[n].
  1. Divide the input array into two subarrays of roughly equal size.
  2. Recursively mergesort the two subarrays.
  3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse Left:	I	N	O	R	S	T	G	E	X	A	M	P	L
Recurse Right:	I	N	O	R	S	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

- The first step is easy: just find the median array index.
- The Recursion Fairy handles the recursive sorts. It's tempting to think about how these recursive sorts are performed or describe the algorithm as repeatedly dividing and then merging on the way back up. But for the sake of designing and describing this algorithm it is sooooo much better to just trust the Recursion Fairy does their thing. You don't need anything else in your head other than those three simple steps.
- That said, the merge step requires a good merging algorithm. Surprisingly, we can think about the merge algorithm recursively as well!
  1. Identify the first element of the output array. The smallest elements from the two subarrays are at their leftmost position, so we just need to check those two positions to find the first element of the output array.
  2. Recursively merge the what remains of the two sorted subarrays.
- That said, the merge procedure is usually written iteratively:

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

- The "recursive call" in Merge is performed after the first iteration of the for loop.
- Before moving on, a bit about notation I like to use. Sometimes, I'll write an array like C[n + 1 .. n] or D[1 .. 0] where the first index is larger than the last. This is a way to denote the empty array that is useful in certain situations like inductive proofs. After all, there are no elements with index at least n + 1 and at most n.

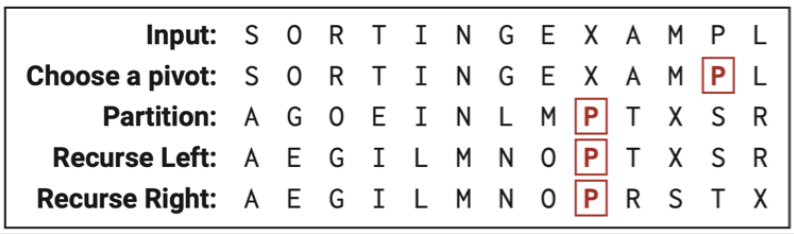
Correctness

- There's a fair amount going on here, so we need a non-trivial proof of correctness. And whenever you want to prove correctness for a recursive algorithm (or iterative algorithm doing something more complicated than a max), you should immediately think "induction".
- We'll actually need two induction proofs; one for merging and one for sorting.
- Lemma: Merge correctly merges the subarrays A[1..m] and A[m+1..n], assuming those subarrays are sorted in the input.
    - Recall how I described the merge procedure as a recursive algorithm written iteratively. That suggests how we'll do our inductive proof.
    - Fix some assignments k, i, and j immediately before an iteration of the for loop begins. We'll count k = n + 1 as the "last" trivial iteration so $1 \le k \le n + 1$. There are n - k + 1 iterations remaining, including the kth.
    - I claim the algorithm puts the elements of A[i .. m] and A[j .. n] into B[k .. n] in sorted order. In particular, the choices of k = 1, i = 1, and j = m + 1 imply the lemma.
    - We'll assume (inductively) that for any assignments $k' > k$, $i' \ge i$, $j' \ge j$ immediately before a *later* iteration that the algorithm puts the elements of A[i' .. m] and A[j' .. n] into B[k' .. n] in sorted order. We can make this assumption, because $n - k' + 1 < n - k + 1$.
    - And now we get to various cases. If k = n + 1, our zero iterations trivially leave empty array B[k .. n] = B[n + 1 .. n] sorted.
    - Otherwise:

    - If j > n, A[j..n] is empty, so min(A[i..m] cup A[j..n]) = A[i].
    - otherwise, if i > m, A[i..m] is empty, so the min is A[j]
    - otherwise., if A[i] < A[j], then the min is A[i]
    - otherwise., A[i] ≥ A[j] and the min can be A[j].
  - In all four cases, B[k] is correctly assigned the smallest element of A[i..m] and A[j..n].
  - In the two cases with B[k] ← A[i], the induction hypothesis implies iterations k + 1 and on merge what remains (A[i+1..m] and A[j..n]) into B[k+1..n].
  - Otherwise, iterations k + 1 and on merge what remains (A[i..m] and A[j+1..n]) into B[k+1..n].
- Yes, that was slightly more tedious than I would require for homework, but only slightly.
- Theorem: MergeSort correctly sorts A[1..n].
  - Assume the algorithm sorts arrays of length k < n.
  - If n ≤ 1, the algorithm correctly does nothing; the array is already sorted.
  - Otherwise, the Recursion Fairy sorts the arrays A[1..m] and A[m+1..n] which both have fewer than n elements. Merge correctly merges them by the previous lemma.
- Both of these proofs follow the usual style of proof for recursive algorithms.
- We do some stuff, the algorithm works correctly on the smaller instances by the induction hypothesis, we do some more stuff.
- Normally, we'd analyze running time at this point, but let's discuss another divide-and-conquer sorting algorithm first, before reviewing the framework we use for run time analysis.

Quicksort

- Quicksort is another recursive sorting algorithm. It was discovered by Tony Hoare in 1959. It's used extensively in practice, because it meshes nicely with the way caches work in our computers.
- Unlike mergesort, all the hard work is done *before* the recursive calls.
  1. Choose a *pivot* element from the array.
  2. Partition the array into three subarrays containing elements smaller than the pivot, the pivot itself, and the elements larger than the pivot.
  3. Recursively quick sort the first and last subarrays.



- Here's the pseudocode. Partition(A[1 .. n], p) takes the *index* of the pivot element as its second parameter and returns the new index of the pivot element after partitioning. This

new index is known as the pivot's *rank*. There are many efficient partitioning algorithms. This algorithm is by Nico Lomuto.

```
QuickSort(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1..r−1])    ⟨⟨Recurse!⟩⟩
      QuickSort(A[r+1..n])    ⟨⟨Recurse!⟩⟩
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                       ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n−1
      if A[i] < A[n]
          ℓ ← ℓ+1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ+1]
  return ℓ+1
```

Correctness

- Like MergeSort, we need two induction proofs for QuickSort.
- First, we need to prove Partition is correct. For that, we argue that after the ith iteration, everything in A[1 .. ell] is less than A[n] (which becomes the pivot after that initial swap) and nothing in A[ell + 1 .. i] is less than A[n].
  - If we haven't looped yet, then the statement is trivially true, because both subarrays are empty. We can count this as the state after iteration 0.
  - Otherwise, at the end of iteration i - 1, it's true by induction that A[1 .. ell] is less than A[n] and A[ell + 1 .. i - 1] is not less.
  - If A[i] ≥ A[n], then we leave A[1 .. ell] alone and we append a single item ≥ A[n] to the end of A[ell + 1 .. i - 1].
  - If A[i] < A[n], then we increment ell and do that swap. Afterward A[1 .. ell] still has elements less than A[n]. Also, we set what used to be A[ell + 1] as A[i] which we inductively know it is not less than A[n].
- Next, we need to prove QuickSort is correct knowing Partition is correct.
- Proof:
  - If n ≤ 1, we correctly do nothing.
  - Otherwise, we correctly partition the array by the previous claim. Now we have three blocks of elements in the correct order.
  - Finally, the induction hypothesis guarantees the two recursive calls correctly sort the blocks of elements before and after the pivot.

Divide-and-Conquer

- Mergesort and quicksort are examples of a *divide-and-conquer* algorithms. They all share this form:
  1. **Divide** the given instance into several *independent smaller* instances of the same problem.
  2. **Delegate** each smaller instance to the Recursion Fairy.

3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

- If the instance size is below some threshold, you ignore recursion and use some different (usually trivial or brute force) algorithm instead.
- Proving divide-and-conquer algorithms correct always requires induction, but usually isn't too bad, assuming you trust the Recursion Fairy.