CS 4349.003.19F Lecture 4—August 28, 2019

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

Recursion

- Recall from last Wednesday, *recursion* is a special type of reduction where both X and Y are the same problem. So just like induction proofs,
  - if the problem instance can be solved directly, solve it directly.
  - Otherwise, reduce the problem instance to one or more **smaller instances of the same problem**.
- Remember, you should imagine that the recursive calls are handled by some other entity called the Recursion Fairy. All they ask is that you give them a smaller instance of the problem. Then, they'll take care of things using methods **THAT ARE NONE OF YOUR BUSINESS**.
- You should not attempt to think about what your algorithm does in the recursive calls, because that just leads to cognitive overload.
- Most of what we do for the next month or so will be designing algorithms using recursion. We'll begin with a common algorithm design paradigm called divide-and-conquer.

Mergesort

- 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. And amazingly, it's like it was designed for teaching divide-and-conquer.
- So let's say we're given an array A[1.. n] of things we want to sort (numbers, letters, shoes, whatever). Our goal is to rearrnage the elements of A[1 .. n] so that $A[1] \le A[2] \le \ldots \le A[n]$.
  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.
- **[Go walk through the figure line by line as you go through these bullet points.]**
- The first step is easy: just find the median array index.
- The Recursion Fairy handles the recursive sorts. Again, the details of the how the left and right recursive sorts are done are not important, and it's best not to worry about it.
- The merge step requires a good merging algorithm. We can again think recursively to design the merge algorithm:

1. Identify the first element of the output array. It's the first element of from one of the two subarrays we just sorted.
2. Recursively merge the what remains of the two sorted subarrays.

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

- 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 performed after the first iteration of the for loop. The first two if/else statements are like base cases.

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.
- Proof:
  - Let A[1..n] be any array and assume A[1..m] and A[m+1..n] are both sorted.
  - We'll prove that for any k from 1 to n + 1, the last n - k + 1 iterations of the main loop correctly merge A[i .. m] and A[j .. n] into sorted array B[k .. n] (B[n + 1 .. n] is shorthand for the empty subarray.)
  - We'll proceed by induction on ell_k := n - k + 1, meaning ell_k will get smaller when we apply the induction hypothesis. In other words, we'll assume "recursive" iterations work out correctly.
  - Now, consider any k from 1 to n + 1.

    - Let k' be any integer where k < k' ≤ n + 1, (implying ell_{k'} < ell_k). Let i' and j' be the values of i and j during iteration k' of the main loop. We'll assume, inductively, that the last n - k' + 1 iterations of the main loop correctly merge A[i'..m] and A[j'..n] into B[k'..n].
    - If k > n (meaning ell_k = 0), there are no iterations left, and the algorithm correctly does nothing to merge two empty subarrays. This is the base case.
    - 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 min(A[i..m] cup A[j..n]) = A[j]
        - otherwise., if A[i] < A[j], then min(A[i..m] cup A[j..n]) = A[i]
        - otherwise., A[i] ≥ A[j] and min(A[i..m] cup A[j..n]) = 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 the last ell+{k+1} = n - k iterations correctly merge sorted subarrays A[i+1..m] and A[j..n] into B[k+1..n].
    - In the other cases, the induction hypothesis implies the last ell_{k+1} = n - k iterations correctly merge 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].
- This one's easier, and it more obviously shows off the pattern of using induction proofs to argue correctness of recursive algorithms.
- Proof:
    - Assume the algorithm sorts arrays of length k < n.
    - If n ≤ 1, the algorithm correctly does nothing; the array is already sorted.
    - Otherwise, the induction hypothesis implies the recursive calls correctly sort A[1..m] and A[m+1..n] which are shorter arrays than A[1 .. n]. 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.

Analysis

- So we now know what we're doing: sorting array A[1 .. n]. We know how: follow that pseudocode. We know why it works thanks to induction. But how fast is this algorithm? Can we bound its worst-case running time?
- Unlike on Monday, we can't just look at a couple for loops and do some multiplication. Those recursive calls make things tricky.
- The most straightforward way of analyzing a recursive algorithm is to express its running time as a *recurrence*, a function defined on smaller instances of itself.
- Let T(n) denote the worst-case running time for MergeSort(A[1 .. n]), whatever it is.

- MergeSort takes the time to do two recursive calls on arrays of size ceil{n / 2} and floor{n / 2}, so T(n) = T(ceil{n / 2}) + T(floor{n / 2}) + some other stuff.
- The other stuff is dominated by Merge(A[1 .. n], m), which is just a single for loop running in O(n) time. So
  - T(n) = T(n) = T(ceil{n / 2}) + T(floor{n / 2}) + O(n).
- When you have recurrences for divide-and-conquer algorithms where problem sizes are being divided by a constant, like in mergesort, the ceilings and floors don't matter. Just solve T(n) = 2T(n / 2) + O(n). Please look at the textbooks if you're curious why this is OK.
- I want to emphasize that we don't know what T(n) is yet, we just know it follows that recurrence. Therefore, we need to pick a big-oh bound that is correct for any T(n) that can make that equation true. Any guesses on the solution?
- It turns out T(n) = O(n log n), meaning MergeSort runs in O(n log n) time. You can verify this fact using induction. We'll discuss how to compute the solution from scratch on Wednesday.

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.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	**P**	L
Partition:	A	G	O	E	I	N	L	M	**P**	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	**P**	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	**P**	R	S	T	X

- 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. Here's one 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] (the pivot) and nothing in A[ell + 1 .. i] is less than A[n].
- Proof sketch:
  - 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 greater.
  - If A[i] ≥ A[n], then we leave A[1 .. ell] alone and we add 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 greater than A[n]. Also, we set what used to be A[ell + 1] as A[i], and we inductively know it is not less than A[n].
- Then, we need to prove QuickSort is correct.
- Proof:
  - If n = 0, 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.

Analysis

- Like in MergeSort, we need a recurrence, but now the size of the recursive calls depends upon r, the rank of the pivot element.
- Intuitively, we could write a worst-case running time recurrence of T(n) = T(r - 1) + T(n - r) + O(n), because Partition is just a for loop over n - 1 elements.
- The best thing would be if we could pick the *median* element as the pivot so r = ceil{n / 1}. Then

    - T(n) = T(ceil(n / 2) - 1) + T(floor{n / 2}) + O(n) ≤ 2T(n / 2) + O(n).
- And we already know the solution to that recurrence is O(n log n).
- In a couple lectures, we'll see a way to find this median element in O(n) time, but it's not really a practical algorithm, especially when used as a subroutine in Quicksort.
- What usually happens is programmers will do something simple like pick the first or last element as the pivot. But now you have no control over the rank of the pivot. The worst-case running time follows the recurrence
    - T(n) = max_{1 ≤ r ≤ n} (T(r - 1) + T(n - r) + O(n)).
- And using the technique for solving recurrences I'll show next Wednesday, you can show the worst case is when the two subproblems are completely unbalanced, i.e. r = 1 or r = n. Now,
    - T(n) = T(n - 1) + O(n)
- And this recurrence solves to **T(n) = O(n^2)**. That's not great.
- There are other ways to pick the pivot that tend to work better. A popular choice is to pick the median of the three elements such as one each from the beginning, middle, and end of the array. But in the worst case, you still get a really unbalanced recurrence that solves to O(n^2).
- One things that does work safely, though, is to pick your pivot uniformally at random. If you do so, you can prove that running time will be O(n log n) *with high probability* no matter what the elements of A[1 .. n] look like.

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.
- When the instance size falls below some threshold, you abandon recursion and switch to some different (usually trivial or brute force) algorithm instead.
- Proving divide-and-conquer algorithms correct always requires induction.
- Next Wednesday we'll discuss how we analyze the running time of these things.