COMP 340 - Lecture Notes - 03 - Recursion, Induction, and Summation

Spring 2014

In these notes we discuss Recursion, Induction, and Summation notation. This material is covered in Skiena Chapter 1.

Recursion

Recursion is central to our understanding of computation.

Recursive Objects

Skeina makes it clear that the mathematical structures typically involved in the models used in our algorithms can be recast as well-define *recursive structures*. We already know a lot about recursive structure from as far back as COMP160 and everyone's favorite first recursive structure, the List.

> A List has two variants. The first is the empty list ∅ containing no elements. The second is the composition of a single element, *first*, and another List called rest.

This definition exhibits all the things you need for a recursive object:

1. There is at least one variant of the structure in which a smaller version of the same structure combined with something else[1]. This is the recursive, or self-referencing, moment in the structure.[2]
2. There is at least one variant in which no recursion takes place.

[1] date or another structure

[2] This is called Structural Recursion

These requirements leave a lot of freedom in terms of where the recursion happens and how we recursively decompose a structure.

- Singleton cases
- Fixed n cases. (two items, three items, etc.)
- $ButLast \circ last$
- $left \circ right$
- $first \circ second \circ Rest$

In all the above $\circ$ is some suitable concatenation/join operation.

Recursive Algorithms

Once you have a recursive structure, you can often get at some kind of recursive algorithm[3]. The key is usually to choose a suitable recursive decomposition for your problem. Take sorting a list as an example. If the list is $\emptyset$ then your list is trivially sorted. If your list is $first \circ Rest$ then you first recursively sort $Rest$. Now you have $first \circ SortedRest$, which is typically not sorted. What we need is an operation f to replace $\circ$ such that $f(first, SortedRest)$ is a sorted list. That operation is the well know INSERT, and this algorithm ends up being INSERTIONSORT.

[3] see HtDP

Now what if we have $Left \circ Right$? What well known algorithm arises if you recursively sort on this structure?

Let's go ahead and knock the whole algorithm out using recursive functions. First INSERT[4]. Let e be some singular datum and S be a Sorted (ascending order) List.

[4] using the same recursive strategy of course

```
INSERT($e, S$)
1  if $S == \emptyset$
2      return $e \circ \emptyset$
3  elseif $e <= S.first$
4      return $e \circ S$
5  else
6      return $S.first \circ$ INSERT($e, S.rest$)
```

Now, for list L

```
INSERTION-SORT($L$)
1  if $L == \emptyset$
2      return $\emptyset$
3  else
4      return INSERT($L.first$, INSERTION-SORT($L.rest$))
```

There we have it. I'd be remiss if I didn't stop and point out that this algorithmic strategy is called *Structural Recursion* as our algorithm recurses on the recursive structure of the data. It requires two things:

1. A recursive structure.
2. An operation which combines a solution and new data to form a new solution.

The first is often easier to find than the second.

Iteration

Our INSERTION-SORT differs from Skiena's in that his is written for arrays and uses iterative loops and ours is written for lists and uses recursion. As you should know, an array is a recursive object by way of its set of index values[5]. We could easily re-tune our recursive list-based algorithm to a recursive array-based algorithm. That leaves just the iterative process itself to consider.

[5] $[0, size) \equiv [0, 0] \cup [1, size)$

Our recursive sort essentially had the following components:

1. A base case in which the input was trivially sorted

2. A recursive case where all but one of the list was recursively sorted and then the remaining item was combined with the now sorted portion using a special operation.

Most of that structure was pulled from the list structure. The key addition we made was identifying and designing the combination operation[6].

[6] Insert

Now, consider the classic iterative version presented by Skiena. We still use Insert logic to combine an arbitrary element with a sorted collection, so clearly it's all about when, where, and how we're utilizing Insert. The iterative solution no longer defers insertion until all but one of the list is sorted. Instead it *constructs the sorted portion as it proceeds*. The effect of this is subdivision of the array into two regions, sorted and unsorted. This is the classic INVARIANT of insertion sort. Let n be the size of the array. Then for all $0 \leq i < n$, the $[0, i)$ is sorted and $[i, n)$ is unsorted at the start of each loop iteration i. The key thing here is we've introduced more than just the INSERT operation logic, we've introduced explicit STATE to our algorithm.

When we iterate we choose to *accumulate* a solution as we proceed. This requires some sense of state for the procedure. With our sort, we can simply accumulate the solution in place, in the array. Often, we introduce new variables to act as accumulators.[7]. Either way, the the fundamental difference is that iteration is an inherently STATEFUL process where recursion is not. So, what are the requirements for iteration:

[7] In programming you often use another piece of state, the index variable, to control the loop itself

- Identify and design an ACCUMULATOR operation to combine singular datum with accumulated state.

- Identify the proper initial state. This usually means the accumulator IDENTITY[8] value but can also mean jump starting the process with the first $0 < m < n$ values.

[8] The value that identifies what it's combined with. $a + 0 = a$

The similarities to recursion should be evident. The key difference is that iteration does the work, insert in our case, right up front, where recursion waits until it effectively finds a trivial case to solve. A great discussion of recursion v. iteration can be found in the now out-of-print *Concrete Abstractions*[9].

[9] https://gustavus.edu/+max/concrete-abstractions.html

Summations

Summations should be viewed as notation, as declaratives. They're not imperatives. If you see a summation, don't immediately reduce it to something. Often you'll actually want to expand it in order to find some larger pattern that leads to a closed-form solution. That's a topic for another time though, right now we want to talk about the fact that summations have a RECURSIVE STRUCTURE.

$$\sum_{i=0}^{n} f(i) \equiv (f(0) + \sum_{i=1}^{n} f(i)$$

Killer Apps

Memorize these and learn how to prove their correctness. They[10] are insanely useful in algorithm analysis.

[10] and their limiting behaviors

1. ARITHMETIC SERIES

$$S(n) = \sum_{i=0}^{n} = \frac{n(n-1)}{2} \qquad (1)$$

2. GEOMETRIC SERIES

$$G(n,a) = \sum_{i=0}^{n} a^i = \frac{a^{n+1-1}}{a-1} \qquad (2)$$

Induction

Whether your algorithm is recursive or iterative[11], you need to evaluate its correctness. Once you're pretty sure it's correct, that means a mathematical proof. The good news is that the go to technique for recursion and iteration is the same: MATHEMATICAL INDUCTION. Quick review of induction. The idea for induction comes from the recursive structure of the Natural Numbers[12]. Thus, we follow two steps:

[11] or something else

[12] $n \in \mathbb{N} = \begin{cases} 0 \\ 1 \\ 1 + n' \in \mathbb{N} \end{cases}$

1. *Basis Step*Show the property holds for a basis step[13]
2. *Inductive Step* Make the INDUCTIVE HYPOTHESIS and assume the property holds for n[14] and show that under this assumption, the property follows for $n + 1$.

[13] 0,1 or some finite n

[14] the Recursive component n' of the recursive case

Sometimes we'll need to make a slightly stronger claim with the induction and show that there's some $k > 1$ such that the property holds for all i where $0 \leq i \leq k$. The basic form is roughly, "it works at the bottom, so I should be able to climb to n.". This stronger form is, "I can climb from the bottom to k, so I should be able to climb to n".

The key here is the logic you use to prove the inductive step should invoke some sense of the basis step. Strong induction makes sure your recursive has a base case to hit.

The property we're primarily concerned with is algorithm correctness, which is not obviously akin to counting. What we've seen though is that our algorithms typically have some kind of recursive structure like counting. If you wrote a recursive algorithm based on structural recursion than your work is pretty much done. We'll simply prove by induction on the structure[15]. Take Insertion Sort for example. The base case of the recursion is the base case of the induction. The recursively sorted data is what we get by the INDUCTIVE HYPOTHESIS. Thus, we're left to justify that our combination operation correctly produces a sorted list[16].

[15] structural induction

[16] Here there be dragons. Don't gloss over details in this step

Proof. **TODO: Proof** □

What about the iterative algorithm? Now we need to prove by induction on our state. The overall structure of the accumulation of problem state is essentially the same as recursive structure. The base case is the initial state we chose. The inductive hypothesis gives us sorted data up to n. Once again, this leaves us to justify the correctness of our combination logic[17].

[17] ACCUMULATOR in this case

Proof. **TODO: Proof** □

What about those summations? Sometimes we think we know a closed-form solution and being mathematics, we need to prove the correctness of our closed-form. For that we use induction. This time we do induction on the upper-bound[18] of the summation. The base case is when the upper-bound is the same as the lower bound[19]. When then get summation up to n by our hypothesis and must justify that adding the $n+1$ term results in the same thing as our closed-form. Once again, we're just working on top of recursive structures here.

[18] the n in $\sum_{i}^{n}$

[19] typically 0 or 1

Proof. **TODO: Proof** □