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

Now, for list L

```
INSERTION-SORT(L)
1  if L == ∅
2        return ∅
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.

In general, the goal is to have the algorithm recursively follow the recursive structure of the input. Both INSERT and INSERTION-SORT from above employ this strategy.

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[5] are insanely useful in algorithm analysis.

[5] and their limiting behaviors

1. ARITHMETIC SERIES

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

2. GEOMETRIC SERIES

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

Induction

Whether your algorithm is recursive or iterative[6], 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.

[6] or something else

A quick review of induction. The idea for induction comes from the recursive structure of the Natural Numbers[7]. Thus, we follow two steps:

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

1. *Basis Step*Show the property holds for a basis step[8]
2. *Inductive Step* Make the INDUCTIVE HYPOTHESIS and assume the property holds for n[9] and show that under this assumption, the property follows for $n + 1$.

[8] 0,1 or some finite n

[9] 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.

Recursive Algorithms and Induction

The property we're primarily concerned with is algorithm correctness, which is not always 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[10]. 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[11].

[10] structural induction

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

Theorem 1. *For any Sorted List S and additional datum e,* $\textsc{Insert}(e, S)$ *is the list containing all the elements of S long with e sorted in ascending order.*

We prove theorem 1 by structural induction on the list S.

Proof. Basis Step: Let S be the empty list $\emptyset$. Then for any datum e, $\textsc{Insert}(e, S)$ is the singleton list $e \circ \emptyset \equiv (e)$. Any singleton list is trivially sorted.

Induction Step: Assume then that for non-empty list $S = S.first \circ S.Rest$ with datum e, $\textsc{Insert}(e, S.rest)$ is a list containing all the elements of $S.rest$ along with e sorted in ascending order. Now consider two cases.

1. $e \leq S.first$

   Let datum e be less than or equal to the first of $S.first$. In this case, $Insert(e, S)$ is $e \circ S$. Because S is sorted, $S.first$ is less than or equal to all elements found in $S.Rest$. It follows then that $e \circ S$ is also sorted in ascending order.

2. $e > \leq S.first$

   Let datum e be strictly greater than $S.first$. Then $Insert(e, S)$ is the list $S.first \circ \textsc{Insert}(e, S.rest)$. Because $S.first < e \leq n \in S.Rest$, and, by the inductive hypothesis, $\textsc{Insert}(e, S.rest)$ is sorted in ascending order, it follows that the list $S.first \circ \textsc{Insert}(e, S.rest)$ is also sorted in ascending order.

Thus, by the hypothesis of induction, $\textsc{Insert}(e, S)$ is a list sorted in ascending order for all datum e and sorted lists S. □

Now that we know INSERT is correct, we can turn to INSERTION-SORT.

Theorem 2. *For any list L,* INSERTION-SORT(L) *is the list containing all the elements of L sorted in ascending order.*

We prove theorem 2 by structural induction on the list L.

Proof. Basis Step: Let L be the empty list $\emptyset$. Then INSERTION-SORT(L) is also $\emptyset$ which is trivially sorted. *Induction Step*: Assume for non-empty list $L = L.first \circ L.Rest$ that INSERTION-SORT$(L.rest)$ is a list containing all the elements of $L.Rest$ sorted in ascending order. Then, INSERT(L) is the list INSERT$(L.first$,INSERTION-SORT$(L.Rest)$. By theorem 1, this list is all the elements for L sorted in ascending order.

Thus, by the hypothesis of induction, for any list L, INSERTION-SORT(L) returns a list sorted in ascending order. □

It's important to note that some assumptions were made in the course of designing and proving the correctness of our sorting algorithm. First, we assumed a well defined $\circ$ operator. This is a fair assumption for lists as concatenation of lists is well-defined. The trickier assumption was that the elements of the list and the datum argument of INSERT were well-ordered to the point that $\leq$ and $>$ could be determined. If any of these assumptions doesn't apply to our problem, then our algorithm *is not correct*.

Summations and Induction

Proving the closed form of a summation is a classic case for induction. In this case we use induction on the sequence of integers over which the summation ranges. The base case occurs when the range is empty or contains a single item. The inductive hypothesis comes from the recursive structure of the summation, which in turn comes from the recursive structure of the integers. Let's prove that repeated addition is multiplication.

Theorem 3. $\sum_{i=1}^{n} a = an$

We prove theorem 3 by induction on n.

Proof. Basis Step: Let $n = 1$. Then,

$$\sum_{i=1}^{1} a = a$$

. *Induction Step:* Assume that the closed form holds for $n - 1$ and $\sum i = 1n - 1a = a(n - 1)$. Then for n

$$\sum_{i=1}^{n} a = \sum_{i=1}^{n-1} + a$$

. By the inductive hypothesis,

$$\begin{aligned}\sum_{i=1}^{n-1} +a &= a(n-1)+a \\ &= an\end{aligned}$$

□

Notice that we invoked the recursive structure in the summation when we peeled out the $n-1$ summation. It was there that we assumed[12] that induction works.

[12] rightly

Iteration

Our INSERTION-SORT differs from Skiena's; his is written for indexed sequences[13] and uses iterative loops and ours is written for recursive lists and uses structural recursion. First off, any indexed structure is a recursive object by virtue of its set of index values[14]. We can easily re-tune our recursive list-based algorithm to a recursive indexed sequence based algorithm. We can even do so in a much more imperative, stateful manner where we modify the sequence contents in place. Let $S[0\,..\,n-1]$ be an indexed sequence of size $n \geq 0$ where for $0 \leq i < n$, $S[i]$ is the i^{th} element in the sequence.

[13] think arrays

[14] $[0, size) \equiv [0,0] \cup [1, size)$

INSERT$(S[0\,..\,n-1])$

```
1  if n ≤ 1
2        return
3  if S[n − 1] < S[n − 2]
4        swap(S[n − 1], S[n − 2])
5  INSERT(S[0 .. n − 2])
```

INSERTION-SORT$(S[0\,..\,n-1])$

Now that we know the indexing isn't the source of our differences we can turn to the iterative process itself. 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[15]. The process itself proceeds by first sorting the rest and

[15] Insert

then that's done, inserting the first. So, the last item to be sorted is the first item in the list.

Now, consider the classic iterative version presented by Skiena. He gives it in C[16]. Let's generalize to pseudocode and re-organize it a bit. Let $S[0..n-1]$ be an indexed sequence of size $n \geq 0$ where for $0 \leq i < n$, $S[i]$ is the i^{th} element in the sequence.

[16] pg 4

INSERTION-SORT$(S[0..n-1], n)$

```
1  for i = 1 to n − 1 by 1
2      for j = i to 1 by -1
3          if S[j] < S[j − 1]
4              swap(S[j], S[j − 1])
```

We still use Insert logic to combine an arbitrary element with a sorted collection[17]. 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 sorted and unsorted regions. The key thing here is we've introduced more than just the INSERT operation logic, we've introduced explicit STATE to our algorithm and in this case that state is index by i.

[17] in this case we've embedded it in the inner loop

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.[18]. 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 then:

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

- Identify and design an ACCUMULATOR operation to "properly" combine singular datum with accumulated state.
- Identify the proper initial state. This usually means the accumulator IDENTITY[19] value but can also mean jump starting the process with the first $0 < m < n$ values[20].

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

[20] as our Insertion sort does

The similarities to recursion[21] should be evident. A great discussion of recursion v. iteration can be found in the now out-of-print *Concrete Abstractions*[22].

[21] and hence our in for induction

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

Induction and Iteration

What about the iterative algorithm? The overall structure of the accumulation of problem state is essentially the same as recursive structure. This lets us proceed by using induction on the accumulation of state. When we're counting state like we do in Insertion Sort, then we

hang our induction on that. The trick is that we typically need to express the action of the loop as a LOOP INVARIANTPROPERTY THAT'S TRUE PRIOR TO THE LOOP AND AFTER EACH LOOP ITERATION property that we can prove to be true. The base case is the initial state we chose, which should satisfy our invariant. The inductive hypothesis allows us to assume the property holds up to $i-1$. This once again leaves us to justify the correctness of our combination logic[23]. Finally, we typically have to connect a few dots to show that when the loop terminates, it leaves us in the desired final state. As you can see, there's a little more to do compared to structural recursion, but the overall big picture is the same: induction. Here's my more formal retelling of Skiena's proof[24] for this iterative algorithm[25].

[23] ACCUMULATOR in this case

[24] pg 15

[25] pg 4

Let's begin with a helper statement proving the effectiveness of the inner-loop structure[26]. This induction is different as we need to work our way down the ladder rather than up it. This means our inductive hypothesis assumes things work from the base case down, and we must show the property holds for one more step down.

[26] Insert!

Lemma 1. *For any indexed sequence $S[0\,..\,n-1]$ with $n > 0$ and $S[0, n-2]$ initially sorted in ascending order, the loop*

```
1  for i = (n − 1) to 1 by -1
2      if S[i] < S[i − 1]
3          swap(S[i], S[i − 1])
```

leaves $S[0\,..\,n-1]$ sorted in ascending order.

We prove lemma 1 by stating and then proving the loop invariant using downward induction on the value of i. Correctness then follows from this invariant.

Proof. The following is an invariant property of this loop: the subsequence $S[i+1\,..\,n-1]$ is sorted in ascending order. The proof of this property can be show by downward induction on i.

Basis Step: Let $i = n-1$. The subsequence $S[n\,..\,n-1]$ is empty and trivially sorted.

Induction Step: Assume that the invariant holds for $i+1$. Now consider the cases for i.

1. $S[i] < S[i+1]$ By the hypothesis of induction, it follows that $S[i+1]$ is less than or equal to all the elements of $S[i+2\,..\,n-1]$, and if $S[i] < S[i+1]$, then so too is $S[i]$. The loop body in this cases causes $S[i]$ and $S[i+1]$ to be swapped. Thus the resultant region $S[i+1\,..\,n-1]$ is sorted in ascending order.

2. $S[i] \geq S[i+1]$ By the hypothesis of induction we know $S[i+1\,..\,n-1]$ is sorted in ascending order. Because $S[i] \geq S[i+1]$, the subsequence $S[i\,..\,n-1]$ is sorted in ascending order.

The loop clearly terminates when $i = 1$, and so by induction the invariant holds for $i = n - 1$ to 1 for any value of n. It then follows that when the loop terminates, the sequence $S[0\,..\,n-1]$ is sorted in ascending order. □

We can now leverage lemma 1 to show that insertion sort works. This will again proceed by induction on a loop invariant. This time we're looking at the outer loop of the algorithm.

Theorem 4. *For any sequence $S[0\,..\,n-1]$ of $n \geq 0$ elements,* INSERTION_SORT(s, n) *sorts s in ascending order.*

We prove theorem 4 by stating and proving by induction the loop invariant of the outer loop.

Proof. Let us proceed by proving the following invariant of the outer loop: the sequence $S[0\,..\,i-1]$ is sorted in ascending order.

Basis Step: Let $i = 1$. Then the sequence $S[0\,..\,0]$ is a singleton and trivially sorted. *Induction Step:* Assume that for $i - 1$, the subsequence $s[0\,..\,i-2]$ is sorted in ascending order. Now by lemma 1 the subsequence $S[0\,..\,i-1]$ is sorted in ascending order at the completion of the inner loop. As the inner loop ends the outer loop, it follows from the hypothesis of induction that the invariant holds and $S[0\,..\,i-1]$ is sorted in ascending order. The outer loop clearly terminates at $n - 1$ and so for any value of n, $S[0\,..\,n-1]$ will be sorted in ascending order. □

I did some mathematical hand-waving in these proofs. The term "clearly" was used with respect to loop termination values. Loop termination is often not clear, so when in doubt clearly justify the termination of the loop itself. Guess what you'd use to do that? You guessed it, induction.