Introduction to the theory of computation
week 7 (chapter 2 of Course Notes)

28th June 2005

Now that you have seen two examples of proving an interative program correct, you are ready to look at another paradigm.

In the iterative programs we examined part of the trick was to come up with a loop invariant that was intimately involved with the correct operation of the program. In both cases I pointed us toward an appropriate loop invariant, although in Assignment 3 you will need to come up with the invariant(s) on your own (but you'll have 2 weeks).

Correctness of recursive programs

The motivation for writing programs recursively is that the structure of the program closely resembles our thought process in solving the problem. We take a large problem and break it into smaller instances of the same problem. If we know that (a) our solution of the smaller instances are correct, and (b) our method of combining the solutions to the smaller instances is correct, then we know our program is correct. Students usually find this somewhat more natural than iterative correctness. Termination and correctness are generally proven in one step: a program is correct and terminates if its parts are correct and terminate.

MergeSort

You have probably already encountered the mergeSort algorithm. The informal description of this program is very brief, assuming that you understand the operation "merge," where you combine two sorted lists by merging them into a single sort ed list. If not, take a few playing cards, split them into two groups, sort the two groups, and then merge the two into a single sorted group (ask your instructor or TA if you've never done this).

Given "merge," here's how to describe mergeSort:

- If your list is of length 1, it is already sorted
- Otherwise, your list has length longer than 1, so break it into two halves
  - mergeSort the first half
  - mergeSort the second half
  - merge the two sorted halves.

The power of recursive algorithms (when they are applicable) is that this simple description is nearly enough to write a working program. The "nearly" part is because (a) the merging step requires some work,[1] and (b) we need to specify the bit about breaking into "halves." Assume that merge works (or else do an iterative

proof of correctness of the program in Examples.java), and use that assumption to prove that mergeSort works correctly.

If we believe that merge is correct with respect to its precondition/postcondition pair, and add the postcondition of mergeSort to the precondition of merge, then the postcondition of merge guarantees the postcondition of mergeSort (look at Example.java on the web). Thus the correctness of mergeSort combines the correctness of merge with the induction hypothesis of the correctness of mergeSort for smaller inputs. We formalize this into a proof by induction.

CLAIM: Let A be an integer array, and let $P(k)$ be "If f, l are integers with $0 \leq f \leq l \leq A.length - 1$, and $k = l - f$, then mergeSort(A, f, l) terminates, and when it does $A[f..l]$ is sorted, contains the same elements as before, and all other elements of A are unchanged." Then $P(k)$ is true for all $k \in \mathbf{N}$.

PROOF (COMPLETE INDUCTION ON k): [2]

Notice that the structure of the proof (base case, induction step) closely parallels the structure of the program. The careful work involves checking that the recursive calls are really on smaller input sizes (from 0 to $k - 1$, inclusive).

RECURSIVE BINARY SEARCH

It is fairly easy to write binary search as a recursive program. The key change is that, if recBinSearch is to be called recursively, the sub-array to be searched must be passed as a parameter. Thus we use parameters f and l as in the iterative binSearch, but now we pass them as parameters (see Example.java).

It seems pretty clear that recBinSearch works properly on trivial arrays of length 1, and that if it works property on shorter arrays it also works properly on longer arrays. We formalize this intuition through induction on $k = l - f$, the size of the sub-array

CLAIM: Let $P(k)$ be "If f, l are integers such that $0 \leq f \leq l \leq A.length - 1$, $k = l - f$, and $A[f..l]$ is sorted when recBinSearch(A, f, l, x) is called, then this call terminates and either returns t such that $f \leq t \leq l$ and $A[t] = x$ (if such t) exists, or else returns $A.length$ (if $x \notin A[f..l]$)." Then $P(k)$ is true for all $k \in \mathbf{N}$.[3]

If you choose $k = A.length - 1$, you can see that recBinSearch$(A, 0, A.length - 1, x)$ is correct with respect to its preconditions.

Again we prove correctness and termination in one step. Termination follows from showing that the recursive call is made on a strictly smaller case. And the structure of the recursive algorithm guides the structure of the proof: the base case is where $(f == l)$, and the induction step assumes that the algorithm works for smaller cases. Since we don't want to fuss with exactly how much smaller, complete induction is the right tool.

Notes

[1](for example an iterative program that would need to be proved correct in its own right)

[2]Suppose f, l are integers with $0 \leq f \leq l \leq A.length - 1$ and $k = l - f = 0$. Then $l = f$ and the "$(f == l)$" branch of mergeSort is executed, so mergeSort(A, f, l) returns without altering A. The single-element array $A[f..f]$ is sorted, contains the same elements as before, and no other elements of A are changed, so the base case $P(0)$ holds.

Induction step: Suppose you have $0 \leq f \leq l \leq A.length - 1$, and assume that $P(\{0, \ldots, k-1\})$ are true. I wish to prove $P(k)$. If $k = 0$, there is nothing to prove, since this was covered in the base case. Otherwise, $k > 0$, which means that $f < l$, and the "$(f == l)$" else branch is executed, and $m = (f + l)/2$ is calculated. By a previous result we know that $f \leq m < l$, so $0 \leq m - f, l - (m+1) < l - f$, so by our IH we assume both $P(m - f)$ and $P(l - (m + 1))$. In other words, by our IH we assume that mergeSort(A, f, m) and mergeSort$(A, m + 1, l)$ both terminate, and when they do that $A[f..m]$ and $A[m + 1..l]$ are both sorted, and all other elements of A are unchanged (in particular, mergeSort$(A, m + 1, l)$ doesn't mess up the results of mergeSort(A, f, m)). Together, these two facts satisfy the precondition for merge(A, f, m, l), and the postcondition of merge(A, f, m, l) implies the postcondition of mergeSort(A, f, l) (that the program terminates, and when it does $A[f..l]$ is sorted, has the same elements, and the rest of A is unchanged).
Thus $P(\{0, \ldots, k-1\})$ implies $P(k)$, as wanted.
I conclude that $P(k)$ is true for all $k \in \mathbf{N}$. QED.

[3]Proof (induction on k): Suppose $0 \leq f \leq l \leq A.length - 1$ and $k = l - f = 0$. In this case $A[f..l]$ is trivially sorted when recBinSearch$A(f, l, x)$ is called. The branch "$(f == l)$" is executed, and in this case there is a single element, $A[f]$, in the subarray. If x is in the subarray, then $A[f] = x$, and the program returns f, as wanted. Otherwise, $x \notin A[f..l]$, so $A[f] \neq x$, and the program returns $A.length$, as wanted. In either case the claim $P(0)$ holds for the base case $k = 0$.

Induction step: Suppose f, l are integers with $0 \leq f \leq l \leq A.length - 1$, and assume that $P(\{0, \ldots, k-1\})$ is true. I want to show this implies $P(k)$. If $k = 0$, I've already shown in the base case that $P(k)$ is true. Otherwise, $k > 0$, which means that $l > f$, and the "$(f == l)$" else branch is executed, and $m = (f + l)/2$ is calculated. By a previous result we know that, in this case, $f \leq m < l$, and we consider two sub-cases:

Case 1: If $A[m] \geq x$, then recBinSearch(A, f, m, x) is called. Since $f \leq m < l$, you know that $0 \leq m - f < l - f = k$, so by the IH $P(m - f)$ holds. Also, $f \leq m < l$ implies that the array $A[f..m]$ is sorted (it is a subarray of a sorted array) and $0 \leq f \leq m < l \leq A.length - 1$. Thus, by $P(m - f)$ recBinSearch(A, f, m, x) terminates, and when it does, if $x \in A[f..l]$, then (since A is sorted) it must be in $A[f..m]$, so by the IH $t \in [f, m]$ is returned such that $A[t] = x$. Otherwise $x \notin A[f..m]$ and by the IH $A..length$ is returned. Thus recBinSearch(A, f, m, x), and hence recBinSearch(A, f, l, x) terminate and return the value specified in the postcondition.

Case 2: If $A[m] < x$, then recBinSearch$(A, m + 1, l)$ is called. Since $f \leq m < l$ implies that $0 \leq l - (m + 1) < l - f$, by the IH $P(l - [m + 1])$ holds. Since $A[f..l]$ is assumed to be sorted, so is its subarray $A[m + 1..l]$, and we also have $0 \leq f < m + 1 \leq l \leq A.length - 1$. Thus we can invoke the IH $P(l - [m + 1])$ to imply that recBinSearch$(A, m + 1, l)$ terminates, and when it does if there is an occurrence of x in $A[f..l]$ then there must be one in $A[m + 1..l]$ (since the array is sorted), and some $t \in [m + 1, l]$ is returned such that $A[t] = x$. Otherwise $x \notin A[m + 1..l]$ and by

the IH $A.length$ is returned. Thus recBinSearch$(A, m+1, l, x)$ and hence recBinSearch(A, f, l, x) terminate and return the value specified in the postcondition.
In both cases $P(\{0, \ldots, k-1\}) \Rightarrow P(k)$, as wanted.
I conclude that $P(k)$ holds for all $k \in \mathbf{N}$. QED.