CS 2742 (Logic in Computer Science) – Fall 2008
Lecture 30

Antonina Kolokolova

November 30, 2008

9.1 Correctness of recursive algorithms

Many of you know the Binary Search algorithm for finding an element in a sorted array. The inputs are an array A of, say, integers and an another integer x, and the output is either the index of the array where that element is located (e.g, if for some i $A[i] = x$, then return i) or an error code (e.g., 0). Because the input array is sorted, it is possible to find x in the array (or check that it is not there) much faster then by checking every element in a way similar to what we did with Max. The BinarySearch algorithm does it as follows: it splits the array into two halves, and then checks if x is greater than the middle element. If so, it is clear that if x is in the array then it must be in the right half; if not, in the left half.

In this section, we will analyze the Recursive Binary search algorithm; the iterative version is left as an exercise. The program consists of two parts: MainBinSearch is a short program making the first recursive call to RecBinSearch, which is the part where all the work is done.

$MainBinSearch(A, x)$
    return $RecBinSearch(A, 1, |A|, x)$

Let's leave the precondition and postcondition for $MainBinSearch$ as an exercise, and look at the recursive part of the program.

The following code does the main part of the work.

$RecBinSearch(A, f, l, x)$
  if $f = l$ **then**
    if $A[f] = x$ **then**
      return f

```
  else
    return 0
else
  m = (f + l)/2
  if A[m] ≥ x then
    return RecBinSearch(A, f, m, x)
  else
    return RecBinSearch(A, m + 1, l, x)
  end if
end if
```

RecBinSearch:
Precondition: $1 \leq f \leq l \leq |A|$ and $A[f..l]$ is sorted.
Postcondition: return t, $f \leq t \leq l$, such that $A[t] = x$, or, if x is not in the array between f and l, return $t = 0$.

It is easy to see that the precondition of *RecBinSearch* follows from the precondition for $MainBinSearch$ for $1 = f$ and $l = |A|$. Also, the postcondition of $MainBinSearch$ follows from the postcondition of *RecBinSearch* since the statements that x is somewhere between the first and the last element of the array and that it is somewhere in the array are equivalent.

It remains to prove correctness of *RecBinSearch*. The proof of correctness of *RecBinSearch* will proceed by strong induction on length of the array, $l - f$.

$P(k)$: if $1 \leq f \leq l \leq |A|$ and $|A[f..l] = k$ and $A[f..l]$ is sorted then the call terminates and returns t, $f \leq t \leq l$, such that $A[t] = x$, or, if x is not in the array between f and l, return $t = 0$.

Base Case: $k = 1$. Then $l = f$ so there is just one element in $A[f..l]$. RecBinSearch returns f if this element is x and 0 otherwise, and makes no further recursive calls.

Induction. Step. Strong induction:
Assume for all $1 \leq i < k$ $P(i)$ holds.
Then the algorithm makes one recursive call to *RecBinSearch*, with different parameters depending whether $A[m] \geq x$ and $A[m] < x$. The input to the algorithm is just a part of the array between f and l. As we said before, if $x > A[m]$ then it must be between $A[m+1]$ and $A[l]$; otherwise, it is in the other half. By induction hypothesis, recursive call will return us the correct answer which we just return. Here we are using strong induction because the sizes of the new arrays are half of the size of the old ones, and besides, due to rounding/odd length of the subarray it might be the case that the length of both halves is the same.

The last remaining part to prove is that the algorithm will terminate. Note that the algorithm

halves the length of the array at every iteration, so eventually the size of the array will approach 1. It cannot become a 0 because the two halves always differ at most by 1 element, and so if one of them is a 0, then another must be 1, but in this case the array we are splitting would be already of length 1.