Proving that Recursive Functions are Correct

Discussion

Because most algorithms have an infinite number of possible inputs, it is not possible to ever determine that such algorithms are correct through testing. The only way that the correctness of an algorithm can be definitely established is using a proof.

To prove that an algorithm is correct, we must define a precondition, which is true before the algorithm begins and a postcondition that must be true after the algorithm completes. Then we must prove that starting with the precondition the postcondition will in fact always be true after the algorithm executes.

Although it is possible to perform proofs of algorithms that involve iteration, it is a relatively complicated process that requires the using of inference rules and requires the definition of many intermediate conditions and a loop invariant for each loop.

By contrast it is much simpler to prove the correctness of recursive algorithms using mathematical induction. The base case of a recursive function will always correspond to the base case of the inductive proof and the recursive case to the inductive case of the proof.

Sample Problem

Consider the following iterative function that computes the *n*th triangular number:

```
int triangular(int n)
{
  int result = 0;
  for (int i = 1; i <= n; i ++)
    result += i;
  return result;
}
```

Rewrite the function `triangular` using recursion and add preconditions and postconditions as comments. Then prove by induction that the recursive function is correct.

Solution

```
//Precondition n >= 1

int triangular(int n)
{
  if (n == 1)
    return 1;
  return triangular(n-1) + n;
}
// Postcondition Returns n(n+1)/2
```

Proof by induction on *n*:

Base case: $n = 1$

1(1+1)/2 = 1, which is what the function returns in its base case

Inductive case: We assume that it is true for $n = k - 1$, we must show it is true for $n = k$, where $n > 1$

return value	= `triangular(k-1) + k`	From program
	= `(k-1)(k-1+1)/2 + k`	By inductive hypothesis
	= $(k^2 - k + 2k) / 2$	By algebra
	= `k(k+1)/2`	By algebra

Determining when One Algorithm Becomes More Efficient than Another

Discussion

When there are multiple algorithms that can solve the same problem and those algorithms have different efficiencies, it is often the case that which algorithm is more efficient may depend upon the size of the data set.

If we have mathematical functions that count the number of lines of code executed for each algorithm as a function of the the size of the data set, represented by n, by determining where the graphs of those functions intersect we can determine at which point(s) the most efficient algorithm changes.

Sample Problem

Suppose the number of steps required in the worst case for two algorithms are as follows:

- Algorithm 1: $f(n) = 3n^2 + 5$
- Algorithm 2: $g(n) = 53n + 9$

Determine at what point algorithm 2 becomes more efficient than algorithm 1.

Solution

We begin by setting the two functions equal to one another:

$$3n^2 + 5 = 53n + 9$$

By algebraic manipulation, we transform the above equation into the quadratic equation below:

$$3n^2 - 53n - 4 = 0$$

We then use the quadratic formula to solve the equation:

$$r_1, r_2 = \frac{53 \pm \sqrt{(-53)^2 - 4(3)(-4)}}{2(3)}$$

$$r_1, r_2 = \frac{53 \pm \sqrt{2809 + 48}}{6}$$

$$r_1, r_2 = \frac{53 \pm 53.45}{6}$$

$$r_1, r_2 = 17.74, -0.075$$

When $n \geq 18$ algorithm 2 is more efficient then algorithm 1

Calculating the Growth Rate of Algorithms by Counting Significant Operations

Discussion

Algorithms containing only assignments and `if` statements can never have more than constant growth rates. To have more than constant growth rates, an algorithm must contain either iteration or recursion. We know that single loops exhibit linear behavior.

Determining the growth rates for nested loops, or even simple loops which contain conditional statements can be a bit more complicated, however. Although a doubly nested loop in which both loops range from 1 to *n* is clearly n^2, when the number of times the inner loop iterates depends upon the loop control variable of the outer loop, it is more complicated to determine the number of times the body of the inner loop will execute.

There is a natural correspondence between mathematical summations and the `for` loop found in most programming languages. The loop-control variable of the `for` loop is akin to the variable whose lower and upper bounds are specified in a mathematical summation. By using such summations, we define a formula that expresses the number of times a loop will iterate. For nested loops, nested summations are required.

We can find a formula for any iterative algorithm using this technique, but there is one important limitation. Although closed forms are known for many simple summations, there is no known technique for generating a closed form for an arbitrary summation. The best we can do is to prove a formula correct by mathematical induction, if we are able to guess a correct formula.

Sample Problem

Using summation evaluation, determine a formula for the number of swaps in the following algorithm, which transposes a square matrix:

```
transpose(int matrix[n][n])
  for i = 1 to n
    for j = 1 to i-1
      swap matrix[i, j] with matrix[j, i]
```

Note: In the above pseudocode, we have departed from Java convention of beginning array subscripts with 0. Also, we skipped the case where `i` and `j` are equal because the values along the main diagonal do not really need to be swapped.

Solution

For this problem, we are interested in defining a formula that will count the number of swaps that are required as a function of *n*, which is the size of the square matrix. We use nested summations to express this formula as follows:

$$f(n) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} 1$$

Because we are counting swaps, notice that one swap is made each time that the innermost loop is executed, which is why the value in the innermost summation is 1. So evaluating the innermost summation is quite simple—it is just $i - 1$. Therefore, we have the following:

$$f(n) = \sum_{i=1}^{n} i - 1 = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \sum_{i=1}^{n} i - n$$

The remaining summation is a relatively simple one—one for which a closed form is known, giving the final formula:

$$f(n) = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}$$

Calculating the Efficiency of Recursive Algorithms Using Recurrence Equations

Discussion

One technique for determining the execution-time efficiency of recursive algorithms is to define a recurrence equation that expresses the amount of time an algorithm requires as a function of some *n*. A recurrence equation is a natural way to express such functions for recursive methods because a recurrence equation is defined recursively. Furthermore, in much the same way that recursive methods must have a base case, so too must a recurrence equation be accompanied by an initial condition in order to be well defined.

This technique is very general but its primary drawback is that there is no general technique for solving recurrence equations and some such equations have no known solutions that can be expressed in a closed algebraic form.

Sample Problem

Given the following recursive chip-and-conquer algorithm for finding the largest value in an array of integers:

```
int chip(int[] array, int first)
  if first is the last element
    return array[first]
  return max(array[first], chip(array, first + 1))

int max(int left, int right)
  if (left > right)
    return left;
  return right;
```

Find the initial condition and recurrence equation that expresses the execution time for this algorithm and then solve that recurrence.

Solution

First, we must observe that *n* represents the size of the subarray under examination on each activation of the recursive method `chip` and that the execution time will be proportional to the number of calls to the method `max`.

Next, we determine our initial condition from the base case of the `chip` method. The base case is when the subarray has only one element, which is when *n* is 1. In that case, no calls to `max` are required, so our initial condition is:

$$t(1) = 0$$

Finally, we can write the recurrence equation by examining the recursive case in the `chip` method. The number of calls to `max` will be the one call made in the method `chip` plus however many calls result from the recursive call to `chip`, which will be given a subarray with one less element to examine. We express that relationship with the following recurrence equation:

$$t(n) = t(n - 1) + 1, \text{ for } n \geq 2$$

Such recurrence equations have a relatively simple solution, which is the following:

$$t(n) = n - 1, \text{ for } n \geq 1$$