COT 5407:Introduction to Algorithms

Author and Copyright: Giri Narasimhan
Florida International University
Lecture 3: September 4, 2007.

1 Recurrence relations and their solutions

A recurrence relation is useful to express the time complexity of iterative and recursive algorithms. For example, $T_1(n) = T_1(n-1) + (n+2)$ expresses the time complexity of an algorithm (such as the iterative SelectionSort) in which the time it takes to solve a problem on an input of size n is equal to the time it takes to solve a problem on an input of size $n-1$ plus some extra work, which in this case is $n-2$ steps. A second example is also provided to make it more clear. $T_2(n) = 2T_2(n/2) + n$ expresses the time complexity of an algorithm (such as the recursive MergeSort) in which the time it takes to solve a problem on an input of size n is equal to the time it takes to solve two subproblems on inputs of size $n/2$ plus the extra work of n steps (in the case of MergeSort, this is the time it takes to do the Merge of the results form the two subproblems – the "conquer" part of the divide-and-conquer algorithm). Note that the left-hand side of the recurrence relation is usually simple the value of the function. The right-hand side of the recurrence relation consists of some "recurrent" terms (such as $T_1(n-1)$ and $T_2(n/2)$) and some "non-recurrent" terms (such as $(n+2)$ and n.

There are three ways to solve recurrence relations: (a) the substitution method, (b) the recursion-tree method, and (c) the master theorem method. All three methods are explained below.

A typical recurrence relation defines a function $T(n)$ (left-hand side of the relation) in terms of some non-recurrent terms and some recurrent terms (right hand-side). The recurrent terms must necessarily be on values smaller than n if we are to be able to solve them using these methods. For our discussions, we will assume that a recurrence relation has the following general form

$$T(n) = F(g(n), T(f_1(n)), T(f_2(n), \ldots, T(f_K(n)), \tag{1}$$

where K is an integer constant, $g(\cdot), f_i(\cdot), i = 1, \ldots, K$ are arbitrary non-recurrent functions, $F(\cdot)$ is an arbitrary function on $K+1$ variables, and $f_i(n) < n$ for $i = 1, \ldots, K$. Every recurrence relation also has a limiting case, which is often implicitly assumed. Whenever not specified, we will assume that $T(1) = O(1)$ is the limiting case. If T models time or space complexities of some algorithm, then this is not an unreasonable assumption to make.

1.1 The substitution method

Given a recurrence relation, the substitution method has two steps: the first step is to guess a solution and the second step is to verify it. The first step is admittedly hard. However, it does get easier as you solve more and more recurrences. The second step involves, verifying

the solution using *induction*. In the verification process, we assume that the guess is correct for values smaller than n. In other words, we assume that the guess is applicable to the right-hand side of the recurrence relation. However, for the verification process to be complete, we need to show that the left-hand side of the recurrence relation is indeed smaller than the right-hand side after applying the appropriate substitutions. It is worth pointing out that the verification process is basically an induction proof.

In class, we discussed the solution of the following recurrence relation:

$$T(n) = 2T(n/2) + n. \tag{2}$$

Exercise 3.1 *Use the substitution method to show that $T(n) = O(n)$ is not a solution for Eq. (2). Also, show that $T(n) = O(n^2)$ is a valid solution, but leaves a lot of "slack". Finally, show that $T(n) = O(n \log n)$ is a valid solution and leaves no slack.*

Exercise 3.2 *If the recurrence is changed to the following:*

$$T(n) = 2T(n/2) + 3n,$$

how will your analysis change? How do the constants change?

Suggestion 3.1 *I recommend that you read the sections on* **Subtleties, Avoiding pitfalls**, *and* **Changing variable** *on pages 65-66 to see some good examples*

1.2 The recursion-tree method

The idea is to recursively expand out all the recurrent terms and then to add them all up. To facilitate the expansion of the recurrent terms, the recursion-tree method suggests that a tree of these terms be used to represent all the terms. Assume that a recurrence relation is provided to you as shown in Eq.(1). To start with you the initialize the tree to a single node labeled $T(n)$. Since $T(n)$ has a recurrence relation describing it, we will refer to it as a recurrent node. Then replace this recurrent node by a subtree whose non-recurrent root node is labeled with the non-recurrent function $g(n)$ and whose K recurrent children are labeled $T(f_1(n)), T(f_2(n), \ldots, T(f_K(n)$ respectively. At any given iteration, the method replaces a recurrent node labeled $T(f(n)$ with a non-recurrent root node labeled with the function $g(f(n))$ and whose K children are labeled $T(f_1(f(n))), T(f_2(f(n)), \ldots, T(f_K(f(n))$ respectively. Note that the number of nodes in the tree and the number of levels in the tree are typically functions of n.

Once the tree is fully expanded out, it often provides us with creative and alternative ways to sum up all the terms. For example, one could add up the terms level by level and try to see a trend in these sums. A recursion tree for Eq. (2) will have the terms on each level add up to the same value n. Combining that with the fact that the recursion tree for Eq. (2) has $O(\log n)$ levels, tells us that its solution is exactly $n \log n$.

1.3 The master theorem method

This is the easiest of the three methods to solve recurrence relations. The master theorem gives you solutions for special cases of solving recurrence relations. It considers recurrence relations of the form

$$T(n) = aT(n/b) + f(n). \tag{3}$$

It provides solutions for the following three cases:

1. if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$;
2. if $f(n) = O((n^{\log_b a - \epsilon})$, for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
3. if $f(n) = \Omega(n^{\log_b a + \epsilon})$. for some $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

Note that the master theorem is only applicable if the recurrence relation has the form show in Eq. (3).

2 Summary

The recursion-tree method requires one to compute summations, which can be quite complicated. In such cases, a crude computation using a recursion-tree method followed by an application of the substitution method can help to quickly zero in the correct function and even nail down the constants involved. The master theorem is only applicable if the recurrence relation has a specific form that arises when analyzing divide-and-conquer algorithms. In practice, it is suggested that you try to use the master theorem first. If it is not applicable then try the recursion-tree method. If you need to nail down the precise constants, then the substitution method is the best method available to you.