CS 5050 Advanced Algorithms

Spring Semester, 2018

Solving Recurrences

Haitao Wang

In general, there are four methods for solving recurrences: expanding, recursion tree, guess-and-verification (substitution), and Master theorem.

1 Expanding Recurrences

This might be the most natural approach to solve recurrences. The idea is to expand the recurrence until we reach the basic case.

Consider the following recurrence:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + n, & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

To solve the recurrence, we can simply expand the recurrence as follows.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &= 4 \cdot T(n/4) + n + n \\ &= 8 \cdot T(n/8) + n + n + n \\ &= \cdots \\ &= 2^k \cdot T(n/2^k) + kn \end{aligned}$$

The expanding stops once $n/2^k$ is equal to 1, which means $k = \log n$. When $k = \log n$, we have $T(n) = 2^k \cdot T(1) + kn = n + n \log n = O(n \log n)$.

Therefore, we conclude that $T(n) = O(n \log n)$.

As another example, consider the following recurrence:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + 1, & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

To solve the recurrence, we can simply expand the recurrence as follows.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + 1 \\ &= 4 \cdot T(n/4) + 2 + 1 \\ &= 8 \cdot T(n/8) + 4 + 2 + 1 \\ &= \cdots \\ &= 2^k \cdot T(n/2^k) + 2^{k-1} + 2^{k-2} + \cdots + 4 + 2 + 1 \end{aligned}$$

By geometric series, we can obtain $2^{k-1} + 2^{k-2} + \cdots + 4 + 2 + 1 = 2^k - 1$.

The expanding stops once $n/2^k$ is equal to 1, or $k = \log n$. When $k = \log n$, we have $T(n) = 2^k \cdot T(1) + 2^k - 1 = n + n - 1 = O(n)$.

Therefore, we conclude that $T(n) = O(n)$.

2 Recursion Tree

As we discussed in class, the recursion tree approach is essentially the same as the expanding approach and it just reorganizes the calculation procedure in a tree structure.

3 Guess-and-Verification (Substitution)

This approach essentially uses the definition of the big-O notation and follows the mathematical induction. The first step is to make a guess. In the verification step, we assume the guess is true for smaller values of n, and then we use the recurrence to prove that the guess is also true for any general n.

Consider the following recurrence:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + n, & \text{if } n \geq 2, \\ 1 & \text{if } n = 1. \end{cases}$$

The guess step. We first make a guess: $T(n) = O(n \log n)$. According to the definition of the big-O notation, we want to find a constant $c > 0$ and an integer $n_0 \geq 1$ such that $T(n) \leq c \cdot n \log n$ for all $n \leq n_0$.

The verification step. We assume the above is true for $n/2$, i.e., $T(n/2) \leq c \cdot \frac{n}{2} \cdot \log(n/2)$. Then, we have the following (note that $\log(n/2) = \log n - 1$).

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &\leq 2 \cdot (c \cdot \frac{n}{2} \cdot \log(n/2)) + n \\ &= c \cdot n \log(n/2) + n \\ &= c \cdot n \log n - cn + n \end{aligned}$$

Hence, in order to prove $T(n) \leq c \cdot n \log n$, it is sufficient to prove $c \cdot n \log n - cn + n \leq c \cdot n \log n$, or equivalently to prove $n \leq cn$. It is easy to see that any $c \geq 1$ suffices.

As summary, the above proves that when $c = 1$ (or $c = 2, 3, \ldots$) and $n_0 = 1$ (n_0 can actually be any positive integer), $T(n) \leq c \cdot n \log n$ holds for all $n \geq n_0$.

Therefore, we conclude that $T(n) = O(n \log n)$.

4 Master Theorem

For constants $a \geq 1$, $b > 1$, and $n_0 \geq 1$, Master theorem is used to solve the following recurrence (let $f(n)$ be a function of n):

$$T(n) = \begin{cases} a \cdot T(n/b) + f(n), & \text{if } n \geq n_0, \\ 1 & \text{if } n \leq n_0. \end{cases}$$

Master theorem says the following:

1. If $f(n) = O(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

   Example: $T(n) = 2T(n/2) + \sqrt{n}$.

   Here, $a = 2$, $b = 2$, and $f(n) = \sqrt{n}$. If we let $\epsilon = 0.1$, then $n^{\log_b a - \epsilon} = n^{0.9}$, and thus, $f(n) = O(n^{0.9})$. Therefore, we obtain $T(n) = \Theta(n)$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

   Example: $T(n) = 2T(n/2) + n$.

   Here, $a = 2$, $b = 2$, and $f(n) = n$. Hence, $f(n) = \Theta(n^{\log_b a})$. Therefore, we obtain $T(n) = \Theta(n \log n)$.

3. If the following two conditions hold:

   (a) $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ for some constant $\epsilon > 0$, and
   (b) $af(n/b) \leq c(f(n))$ for some constant $c < 1$ and all sufficiently large n,

   then $T(n) = \Theta(f(n))$.

   Example: $T(n) = 2T(n/2) + n^2$.

   Here, $a = 2$, $b = 2$, and $f(n) = n^2$. If we let $\epsilon = 0.1$, then $n^{\log_b a + \epsilon} = n^{1.1}$, and thus, $f(n) = \Omega(n^{1.1})$. Therefore, the condition (a) is satisfied.

   For condition (b), we want to prove $2f(n/2) \leq cf(n)$, i.e., $2 \cdot (n/2)^2 \leq c \cdot n^2$. It is easy to see that if we let $c = 1/2$, then $2f(n/2) \leq cf(n)$ holds for all $n \geq 1$. Hence, the condition (b) is also satisfied.

   Therefore, we conclude that $T(n) = \Theta(n^2)$.

Note: As we discussed in class, not every such recurrence can be solved by Master theorem.

Comments

1. To solve a recurrence, you may want to try Master theorem first because it can "directly" give you the solution if it works.
2. If Master theorem does not apply, then try expanding or recursion tree approaches.
3. For the guess-and-verification approach, the key is to make a "good" guess, which is based on your experience. However, if you have a clear target to prove, i.e., you want to particularly prove $T(n) = O(n^2)$, then the substitution approach might be the easiest approach.
4. Not every recurrence can be solved by Master theorem. The other three approaches, however, should be able to solve all recurrences in general (although might be not easy).