Solving Recurrences

Recurrences

Estimations of the time complexity of recursive algorithms are usually done using *recurrencies*: equations representing the time complexity of a problem of size n via the time complexity of smaller problems.

Examples. All of the following are recurrencies:

$$T(n) = 2T(n-1) + 4$$
$$T(n) = 3T(\frac{n}{2}) + \log_2(n)$$
$$T(n) = 2T(\frac{n}{2})$$
$$T(n) = 4T(n-6) + 2n\log_2(n)$$

The general form of a recurrence we will consider is

$$T(n) = a \cdot T(\frac{n}{b} - d) + f(n)$$

Such a recurrence describes the time complexity of a recursive divide-and-conquer algorithm which:

- Reduces the problem of size n to finding answers to a subproblems...
- ...each of which has the size $\frac{n}{b} - d$ ...
- ...and takes $f(n)$ time to assemble the solution of the problem of size n from the a computed solutions.

Solving Recurrences: Master Method

The *master method* is applicable to the recurrences of the form

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

It is derived from the following statement, called the Master theorem:

Theorem. Lab a and b be two constants, $f(n)$ be an asymptotically positive function. Let $T(n)$ be defined as:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

Then:

1. If $f(n) = O(n^{\log_b(a-\epsilon)})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$.
2. If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log_2(n))$.
3. If $f(n) = \Omega(n^{\log_b(a+\epsilon)})$ for some $\epsilon > 0$, **and** $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$ for all $n > n_0$, then $T(n) = \Theta(f(n))$.

Note. Essentially, the master theorem checks whether $f(n)$ grows faster, slower or the same as $n^{\log_b(a)}$, and chooses the asymptotically faster of the two functions as the solution. If they grow in a similar way, then $\log_2(n)$ is added to the solution.

In the recurrence:

- $f(n)$ controls the amount of time an algorithm spends on each recursive step. If $f(n)$ is sufficiently fast-growing, it starts dominating the computation.
- $n^{\log_b(a)}$ controls the thime it takes to process the pure recursion. It is based on the number of subproblems to be solved on each step (represented by a) and the "shrinkage" factor of each subproblem (represented by b). If this function is asymptotically faster growing than $f(n)$, it means that processing recursion starts dominating the running time of the algorithm.

Example 1. $T(n) = 2T(n/2) + 5$.

Here: $a = 2$, $b = 2$, $f(n) = 5$ and $f(n) = O(1) = O(n^{\log_2(2)-\epsilon}) = o(n)$. Therefore,

$T(n) = \Theta(n) = n^{\log_2(2)}$.

Example 2. $T(n) = 8T(n/2) + n^3$.

$a = 8$, $b = 2$, $\log_b(a) = \log_2(8) = 3$, $f(n) = n^3$;
$f(n) = n^3 = \Theta(n^{\log_b(a)}) = \Theta(n^3)$. Therefore,

$T(n) = \Theta(n^2 \log_2(n))$.

Example 3. $T(n) = 2T(n/2) + n^2$.

$a = 2$; $b = 2$; $\log_b(a) = 1$, $f(n) = n^2$; $f(n) = \Omega(n)$ and $af(n/b) = 2(n/2)^2 = 0.5n^2 \leq 0.5f(n)$. Therefore,

$T(n) = \Theta(n^2)$.

Example 4. $T(n) = 2(n/2) + n\log_2(n)$.

Here, $a = 1$, $b = 1$, $\log_b(a) = 1$; $f(n) = n\log_2(n)$.

We know that $f(n) = \Theta(n)$, **however**, for any $\epsilon > 0$, $f(n) \neq \Theta(n^{\epsilon})$.

Therefore, neither of the cases of the master theorem is applicable.

Exercise 4.6.-2 of the textbook provides the bound for this case:

- if $f(n) = \Theta(n^{\log_b(a)} \log_2^k n)$, $k \leq 0$, then
  $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n)$.

This gives the solution of the recurrence in Example 4 as

$T(n) = \Theta(n\log_2^2(n))$.

Other Cases

Some recurrences cannot be solved using master theorem.

Example 5. $T(n) = 2T(n-1) + 2$.

Here, we have $T(n-d)$ on the right side of the recurrence, rather than $T(n/b)$.

You can use (cautiously) a *guess-and-check* or *substitution* method for such equations.

1. **Step 1.** Gueass the form of the solution.
2. **Step 2.** Substitute and check.

(note, technically, this is a proof by induction. By guessing $T(n) = O(f(n))$, we make an inductive assumption that for all $m < n$, we have already established $T(m) = O(f(m))$. We would also need to establish the base cases, but those, typically, are straighforward.)

Let us guess that $T(n) \leq c(2^n - 1)$. We assume that we have already shown for $n-1$ that $T(n-1) \leq c(2^{n-1} - 1)$. Then,

$T(n) = 2T(n-1)+2 \leq 2c(2^{n-1}-1)+2 = c2^n-2c+2 = c2^n-2(c-1) \leq c2^n$.

(notice that setting $T(n) \leq c2^n$ would not have worked, as we'd get $T(n) \leq c2^n + 2$, which is not what we needed to prove).