CS 170 Discussion Notes: September 12, 2007

1 Two Master Theorems

1.1 Big-O Version

Here's the master theorem from the textbook:

Suppose $T(n) = aT(n/b) + n^d$ and $T(1) = \Theta(1)$.

(1) If $d < \log_b a$, then $T(n) = O(n^{log_b a})$.

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

(3) If $d > \log_b a$, then $T(n) = O(n^d)$.

This version of the master theorem is most useful for approximating an upper bound for $T(n)$ when the recurrence cannot be solved by the Θ-version below.

1.2 Big-Theta Version

The O-version of the master theorem only works for polynomials n^d, and it only gives you an upper bound instead of a Θ bound. Here's a stronger version of the master theorem:

Suppose $T(n) = aT(n/b) + f(n)$ and $T(1) = \Theta(1)$.

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

(3) If $f(n) = \Omega(n^{log_b a + \epsilon})$, for some $\epsilon > 0$ and the regularity condition ($af(n/b) \leq cf(n)$ for some $c < 1$ and sufficiently large n) holds, then $T(n) = \Theta(f(n))$.

You will not need to worry about the regularity condition in case (3) as all functions $f(n)$ you encounter in this course should satisfy it.

Case (1) is satisfied iff $f(n)$ is polynomially smaller than $n^{log_b a}$, case (2) is satisfied iff $f(n)$ and $n^{log_b a}$ are asymptotically equivalent, and case (3) is satisfied iff $f(n)$ is polynomially bigger than $n^{log_b a}$ (and the regularity condition holds). $f(n)$ is polynomially bigger than $g(n)$ if we can find some $\epsilon > 0$ such that $f(n)$ is $\Omega(g(n)n^\epsilon)$. Roughly speaking, $f(n)$ is polynomially bigger than $g(n)$ when $f(n)$ asymptotically dominates $g(n)$ even after we multiply $g(n)$ by a very small order polynomial (e.g., $n^{0.00001}$). For example, n^2 is polynomially bigger than $n \log n$, however, neither $n \log n$ nor $2n$ are polynomially bigger than n.

Note that there are some recurrences that will not satisfy any of the three conditions. In these cases, you can replace $f(n)$ with a polynomial n^d such that $f(n)$ is $O(n^d)$ and solve $T(n) = aT(n/b) + n^d$ using the O-version of the theorem. You can then obtain an upper bound of the true solution.

1.3 Master Theorem Examples

(i) If $T(n) = 3T(n/2) + \Theta(n \log n)$, then $f(n) = n \log n$ and $n^{\log_b a} = n^{\log_2 3}$. Therefore, $T(n) = \Theta(n^{log_2 3})$.

(ii) If $T(n) = 2T(n/2) + \Theta(n \log n)$, then $f(n) = n \log n$ and $n^{\log_b a} = n$. Since $n \log n$ does not polynomially dominate n, the Θ-version cannot be applied. However, we can still give an upper bound for $T(n)$ by writing $T(n) = 2T(n/2) + O(n^2)$. Thus, $T(n) = O(n^2)$ by the O-version.

(iii) If $T(n) = T(n/2) + \Theta(n \log n)$, then $f(n) = n \log n$ and $n^{\log_b a} = n^0 = 1$. Therefore, $T(n) = \Theta(n \log n)$.

2 Divide-and-Conquer Examples

(a) The nth Fibonacci number, F_n, satisfies the following:

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ (F_{\lceil n/2 \rceil})^2 + (F_{\lceil n/2 \rceil - 1})^2, & \text{if } n \geq 2 \text{ and } n \text{ is even} \\ (F_{\lceil n/2 \rceil})^2 + 2F_{\lceil n/2 \rceil} F_{\lceil n/2 \rceil - 1}, & \text{if } n \geq 2 \text{ and } n \text{ is odd} \end{cases}$$

We can formulate this into a recursive divide-and-conquer algorithm:

```
Fib(n)
    If n <= 0, return 0
    If n = 1, return 1
    x = Fib(ceil(n/2))
    y = Fib(ceil(n/2) - 1)
    if n is odd, return x*x + y*y
    if n is even, return x * (x + 2*y)
End
```

To determine the runtime complexity, assume we can add two n-bit integers in $\Theta(n)$ time and multiply two n-bit integers in $\Theta(n^c)$ time, with $1 < c < 2$. The algorithm makes two recursive calls to subproblems of half the size. Also, the nth Fibonacci number is $\Theta((\frac{1+\sqrt{5}}{2})^n)$, so it must have $\Theta(n)$ bits and each iteration does $\Theta(n^c)$ amount of work. This is the amount of work Fib(n) does outside of recursive calls is a constant number of multiplications and additions on $\Theta(n/2)$ bit integers.

Therefore, we have the recurrence $T(n) = 2T(n/2) + \Theta(n^c)$, with $1 < c < 2$ and $T(1) = \Theta(1)$. By the master theorem, since $f(n) = n^c$ is polynomially bigger than $n^{\log_b a} = n$, we have $T(n) = \Theta(n^c)$.

(B) Problem 2.23 from the textbook.

3 Multiplication Algorithms

So why did we use $\Theta(n^c)$, with $1 < c < 2$, for the time complexity of multiplying two n-bit integers, instead of just using the value of c corresponding to the fastest-known multiplication algorithm? The answer is that, in practice, the choice of multiplication algorithm depends on the size of n.

Popular multiplication algorithms include:

(1) Long multiplication. This is the paper-and-pencil method you learned in school. Running time is $\Theta(n^2)$. Outperforms other algorithms for small n (because of the small constant hidden in the Θ-notation).

(2) Karatsuba algorithm. This is the method you learned in lecture that splits the integers into halves and cleverly reduces the number of multiplications. Running time is $\Theta(n^{\log_2 3})$. Outperforms long multiplication once n is bigger than about 100 or 1,000 (the actual threshold depends on the particular computer architecture).

(3) Schonhage-Strassen algorithm. It uses the Fast Fourier Transform and is asymptotically the fastest known multiplication algorithm. Running time is $O(n \cdot \log n \cdot \log(\log n))$. Outperforms Karatsuba's algorithm only when n is bigger than about 10,000 or 100,000 (because of the large constant hidden in the Θ-notation). Remember, n is the number of bits, so this algorithm is seldom used for normal applications.

A linear algebra package, for example, might employ a recursive algorithm that uses Karatsuba's algorithm for the top-level calls and then switches to long multiplication once subproblems reach a small enough size (under 100 or 1,000 bits).