

CS 3343 (Spring 2008) Assignment 4

Solution

1. (30 points) On the course web page for homework #4, follow the links to download three Java programs that I wrote to compute the power of two. The algorithms for two of the programs are very similar to what we have discussed in class. Another one is new.

- a. (5 points) Compile the programs on any computer and use each program to compute 2^n with 5 to 10 values of n of your choice (for example, you may use $n = 10, 20, 30, 40, 50, 100, 500, 1000$). To compute 2^{10} using *Alg1*, you can type in *java Alg1 10*. Similar for *Alg2* and *Alg3*. The programs output two values: the value of 2^n , and the number of times that the function *power2* was called. What did you observe? Did the three algorithms give the same result for the same input? What is the largest n that each program can solve? Do you know why? (Answer this after you finished (b)-(d).) Be succinct.

n	2^n	# recursive calls			CPU time		
		Alg1	Alg2	Alg3	Alg1	Alg2	Alg3
10	1024	5	11	2047	< 1s	< 1s	< 1s
20	10^6	6	21	2×10^6	< 1s	< 1s	< 1s
30	10^9	6	31	2×10^9	< 1s	< 1s	10s
40	10^{12}	7	41	–	< 1s	< 1s	2h
50	10^{15}	7	51	–	< 1s	< 1s	0.5yr (projected)
100	10^{30}	8	101	–	< 1s	< 1s	10^{14} yr (projected)
200	10^{60}	9	201	–	< 1s	< 1s	10^{44} yr (projected)
500	10^{150}	10	501	–	< 1s	< 1s	10^{135} yr (projected)
1000	10^{301}	11	1001	–	< 1s	< 1s	10^{285} yr (projected)

As seen from the table above, the numbers of recursive calls made by Alg1, Alg2, and Alg3 to compute 2^n are $\lfloor \log_2 n \rfloor + 2$, $n + 1$, and $2^{n+1} - 1$, respectively. This empirical result suggests that the three algorithms have time complexity in $\Theta(\log n)$, $\Theta(n)$, and $\Theta(2^n)$, respectively. Indeed, Alg1 and Alg2 can compute 2^n for almost any n (limited only by the precision of double numbers) instantly. In contrast, Alg3 computes 2^n in about 10 seconds for $n = 30$, 20 seconds for $n = 31$, 1 minute for $n = 33$, and 1 hour for $n = 39$. It would take about 24 hours for $n = 43$, a week for $n = 45$, a year for $n = 51$, etc.

- b. (8 points) Study the algorithms in the three programs, and write down the recurrence for each algorithm.

Observe that within each call to *power2*, excluding the recursive call to itself, the other statements take constant time. Therefore, to estimate the total running time, we can simply count the number of times that *power2* is called.

Let the numbers of calls to *power2* made by the three algorithms be T_1 , T_2 , and T_3 , respectively. By studying the three algorithms, we can define the following recurrences:

$$T_1(n) = T_1(\lfloor n/2 \rfloor) + 1 \tag{1}$$

$$T_2(n) = T_2(n - 1) + 1 \tag{2}$$

$$T_3(n) = 2T_3(n - 1) + 1 \tag{3}$$

It is also not hard to define the following base cases:

$$T_1(0) = T_2(0) = T_3(0) = 1, T_1(1) = T_2(1) = 2.$$

- c. (12 points) Solve the three recurrence functions using either the recursion tree method or the iteration method. Do not worry about the base cases.

To solve Equation (1), we can use the iteration method:

$$\begin{aligned} T_1(n) &= T_1(\lfloor n/2 \rfloor) + 1 \\ &= (T_1(\lfloor n/4 \rfloor) + 1) + 1 \\ &= ((T_1(\lfloor n/8 \rfloor) + 1) + 1) + 1 \\ &= \dots \\ &= T_1(\lfloor n/2^i \rfloor) + i \end{aligned}$$

We can continue the above iteration until $\lfloor n/2^i \rfloor = 1$. This happens when $i = \lfloor \log_2 n \rfloor$. Therefore,

$$T_1(n) = T_1(1) + \lfloor \log_2 n \rfloor = 2 + \lfloor \log_2 n \rfloor. \quad (4)$$

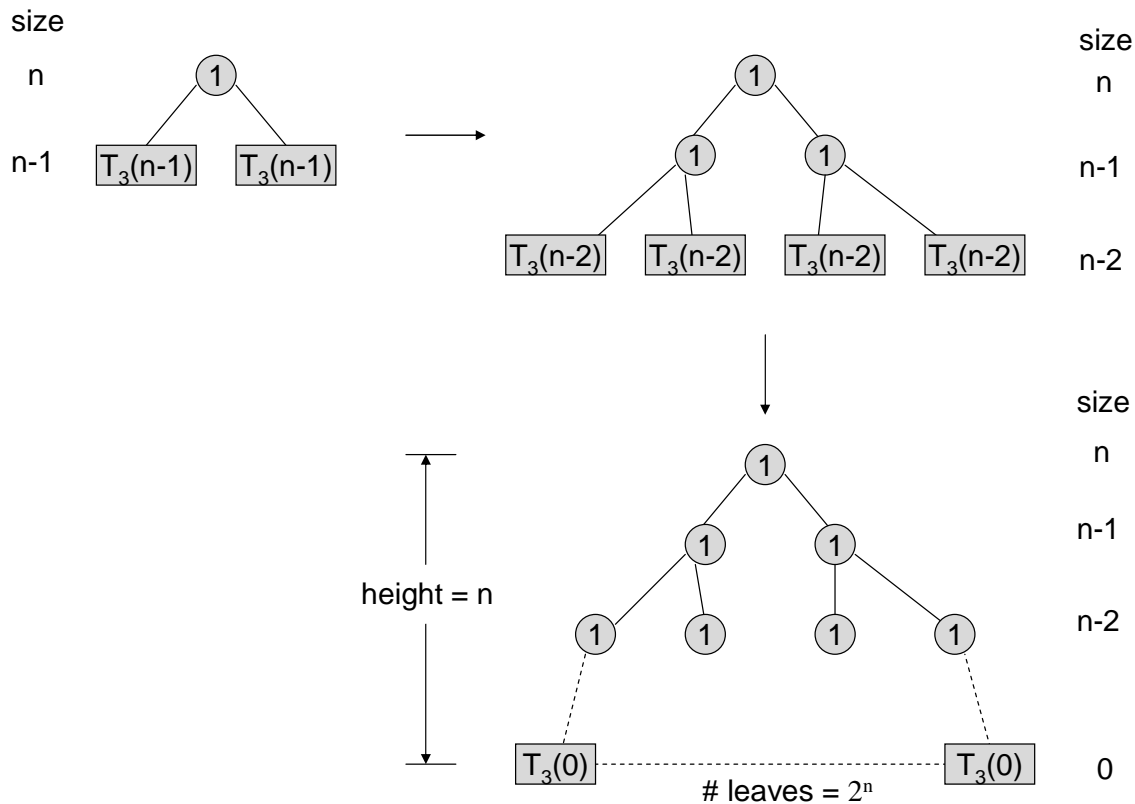
We can also use the iteration method to solve Equation (2):

$$\begin{aligned} T_2(n) &= T_2(n-1) + 1 \\ &= (T_2(n-2) + 1) + 1 \\ &= ((T_2(n-3) + 1) + 1) + 1 \\ &= \dots \\ &= T_2(n-i) + i \end{aligned}$$

Continue the above iteration until $n-i=0$, i.e., $i=n$, we have

$$T_2(n) = T_2(0) + n = 1 + n. \quad (5)$$

We can use either the iteration method or the recursion tree method to solve Equation (3). I use the recursion tree method here:



Since $T_3(0) = 1$, the sum of the leaves is 2^n . The total sum of all the numbers in the tree nodes is therefore $1 + 2 + 4 + \dots + 2^n = \sum_{i=0}^n 2^i$.

Using the formula for geometric series,

$$T_3(n) = 2^{n+1} - 1. \quad (6)$$

If you did not use the base cases, you may not derive the exact forms as Equations (4)-(6), but you should be able to get something in the same orders of growth.

- d. (5 points) Does the second output (i.e., the number of times that *power2* was called) match your expectation? Why or why not?

Equations (4)-(6) are exactly the same as those we derived empirically from the Table above.

2. (30 points) Assume that $T(1) \in \Theta(1)$. Solve the following recurrence functions using the master method to derive a tight bound (Θ). If the master method cannot be applied, state the reason, and give an upper bound (big-Oh) as tight as you can. Always justify your answer.

a. $T(n) = 4T(n/2) + n$;

Solution: $a = 4, b = 2, f(n) = n$. This is CASE 1, since $n^{\log_b a} = n^2$ dominates $f(n)$ by a polynomial factor n . Therefore $T(n) = \Theta(n^2)$.

b. $T(n) = 9T(n/3) + n^2$;

Solution: $a = 9, b = 3, f(n) = n^2$. This is CASE 2, since $n^{\log_b a} = n^2 \in \Theta(f(n))$. Therefore $T(n) = \Theta(n^2 \log n)$.

c. $T(n) = 6T(n/4) + n$;

Solution: $a = 6, b = 4, f(n) = n$. This is CASE 1, since $\frac{n^{\log_b a}}{f(n)} = \frac{n^{\log_4 6}}{n} = n^{\log_4 6 - 1} = n^{\log_4 6/4} = n^{\log_4 1.5}$. Also $\log_4 1.5 \approx 0.3 > 0$. Therefore $T(n) = \Theta(n^{\log_4 6})$.

d. $T(n) = 2T(n/4) + n$;

Solution: $a = 2, b = 4, f(n) = n$. This is CASE 3, since $\frac{f(n)}{n^{\log_b a}} = \frac{n}{n^{1/2}} = n^{1/2} \in \Omega(n^{1/2})$. Therefore $T(n) = \Theta(n)$.

e. $T(n) = T(n/2) + n \log n$;

Solution: $a = 1, b = 2, f(n) = n \log n$. This is CASE 3, since $\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n^0} = n \log n \in \Omega(n)$. Therefore $T(n) = \Theta(n \log n)$.

f. $T(n) = 4T(n/4) + n \log n$.

Solution: $a = 4, b = 4, f(n) = n \log n$. The master method does not apply, since $\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n} = \log n \in o(n^\epsilon)$ for any positive ϵ .

We know that $\log n \in o(n^\epsilon)$ for any $\epsilon > 0$, therefore $n \log n < n^{1+\epsilon}$ for sufficiently large n . Hence, $T(n) < 4T(n/4) + n^{1+\epsilon}$ for sufficiently large n . To get an upper bound for $T(n)$, we can solve the following recurrence:

$$S(n) = 4S(n/4) + n^{1+\epsilon}$$

This recurrence can be solved using CASE 3 of the master method, which gives $S(n) \in \Theta(n^{1+\epsilon})$. The solution to the original recurrence is, therefore, $T(n) \in O(n^{1+\epsilon})$ (because it is an upper bound).

If you remember the extension to CASE 2 of the master method, you can in fact directly get $T(n) \in \Theta(n \log^2 n)$.