CS325 Winter 2013: HW 1

Daniel Reichert
Trevor Bramwell
Lance Stringham

January 18, 2013

0.2

Problem: Show that, if c is a positive real number, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is:

(a) $\Theta(1)$ if $c < 1$

(b) $\Theta(n)$ if $c = 1$

(c) $\Theta(c^n)$ if $c > 1$

The moral: in big-Θ terms, the sum of a geometric series is simply the first term if the series is strictly decreasing, the last term if the series is strictly increasing, or the number of terms if the series is unchanging.

Solution:

(a) If $c < 1$ then $g(n)$ will approach 1.

(b) If $c = 1$ then $g(n)$ will approach n.

(c) If $c > 1$ then $g(n)$ will approach c^n.

0.3(a)

Problem: The Fibonacci numbers $F_0, F_1, F_2, \ldots$, are defined by the rule

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}.$$

In this problem we will confirm that this sequence grows exponentially fast and obtain some bounds on its growth.

(a) Use induction to prove that $F_n \geq 2^{0.5n}$ for $n \geq 6$.

Solution: Proof by induction: Base cases: $n = 6$,$n = 7$. $F_6 = F_5 + F_4 = 8$ $F_7 = F_6 + F_5 = 13$ $2^{0.5(6)} = 8$ $2^{0.5(7)} \approx 11.314$ Since both cases hold, our base case is established. Inductive step: Assume: $F_k \geq 2^{0.5k}$ for $6 \geq k \geq n$ Prove: $F_k + 1 \geq 2^{0.5(k+1)}$

$F_k \geq 2^{0.5k}$ $F_k + F_k - 1 \geq 2^{0.5k} + 2^{0.5(k-1)}$ In the above step we added to both sides the base case where k is replaced by k-1. $F_k + 1 \geq 2^{0.5k} + 2^{0.5(k-1)}$ A Simplification from the definition of the Fibonacci numbers. $F_k + 1 \geq 2^{0.5k} + 2^{0.5k}2^{-.5}$ Expansion of terms. $F_k + 1 \geq 2^{0.5k}(1 + 2^{-.5})$ Factoring like terms. $F_K + 1 \geq 2^{0.5(k+1)}$ Here we reduce the left hand side term to $2^{0.5(k+1)}$ because $2^{0.5(k+1)} > 2^{0.5k} + 2^{0.5k}2^{-.5}$ Thus, by the principle of mathematical induction we have shown that $F_n \geq 2^{0.5n}$ for $n \geq 6$

2.3 from text book

Section 2.2 describes a method for solving recurrence relations which is based on analyzing the recursion tree and deriving a formula for the work done at each level. Another (closely related) method is to expand out the recurrence a few times, until a pattern emerges. For instance, lets start with the familiar $T(n) = 2T(n/2) + \mathrm{O}(n)$. Think of $\mathrm{O}(n)$ as being $\leq cn$ for some constant c, so: $T(n) \leq 2T(n/2) + cn$.

By repeatedly applying this rule, we can bound $T(n)$ in terms of $T(n/2)$, then $T(n/4)$, then $T(n/8)$, and so on, at each step getting closer to the value of $T(\cdot)$ we do know, namely $T(1) = \mathrm{O}(1)$.

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + cn \\
&\leq 2[2T(n/4) + cn/2] + cn = 4T(n/4) + 2cn \\
&\leq 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn \\
&\leq 8[2T(n/16) + cn/8] + 3cn = 16T(n/16) + 4cn \\
&\vdots
\end{aligned}
$$

A pattern is emerging... the general term is

$$T(n) \leq 2^k T(n/2^k) + kcn.$$

Plugging in $k = \log_2 n$, we get $T(n) \leq nT(1) + cn \log_2 n = \mathrm{O}(n \log n)$.

1. Do the same thing for the recurrence $T(n) = 3T(n/2) + \mathrm{O}(n)$. What is the general kth term in this case? And what value of k should be plugged in to get the answer?

2. Now try the recurrence $T(n) = T(n1) + \mathrm{O}(1)$, a case which is not covered by the master theorem. Can you solve this too?

2.4 from text book

Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n1$ and then combining the solutions in constant time.

3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\mathrm{O}(n^2)$ time.

A) $T(n) = 5T(n/2) + \mathrm{O}(n)$ From the master theorem we know that the complexity case is determined by a/b^d. Since a/b^d is 2.5 in algorithm A,

the complexity is $\mathrm{O}\left(n^{log_2 5}\right)$ B) $T(n) = 2T(n-1) + c$ This case does not fall under the master theorem. In each level of the recursive tree there are $2^i(n-(i-1))+c$ operations to perform. Simplifying for the dominant term we find that the complexity is $\mathrm{O}(2^n)$. c) $T(n) = 9T(n/3) + \mathrm{O}(n^2)$ From the master theorem we know that the complexity case is determined by a/b^d. Since a/b^d is 1 in algorithm C, the complexity is $\mathrm{O}(n^2 log_3 n)$

Given the choice between the three algorithms, A has the best asymptotic complexity because $\mathrm{O}\left(n^{log_2 5}\right) < \mathrm{O}(n^2) < \mathrm{O}(n^2 log_3 n)$. What are the running times of each of these algorithms (in big-O notation), and which would you choose?

2.17 from text book

Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $\mathrm{O}(logn)$.

This can be accomplished with binary search. Pseudo code:

```
search(A,min, max, i)
while (max >= min )
mid = midpoint of min and max
if A[mid] < i
min = mid + 1
else if A[mid] > i
max = mid - 1
else //this means that A[i] == i
return "found A[i] == i"
return "A[i] never equals i"
```