Prof. Sergio A. Alvarez
21 Campanella Way, room 569
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS383, Algorithms
Spring 2009
HW4 Solutions

1. Suppose that $f(n)$ and $g(n)$ are strictly positive functions defined for all positive integers n, such that $f(n) = O(g(n))$. For each of the following, state whether the given relationship necessarily follows from the above assumption alone or not. If you state that the stated relationship follows, provide a proof. Otherwise, provide a specific counterexample.

(a)
$$f(n) = \Theta(g(n))$$

Solution

This is certainly not implied by $f(n) = O(g(n))$. Consider, for example, the following functions:
$$f(n) = \log n, \quad g(n) = n,$$
for which $f(n)/g(n) \to 0$ as $n \to \infty$ (by L'Hôpital's rule – see my notes on asymptotic time complexity), so that $f(n) = O(g(n))$ but not $g(n) = O(f(n))$.

(b)
$$\frac{1}{g(n)} = O\left(\frac{1}{f(n)}\right)$$

Solution

This one is true! The assumption that $f(n) = O(g(n))$ means that there exist finite constants $C > 0$ and n_0 such that
$$f(n) \le Cg(n) \quad \text{for all } n \ge n_0$$
Since the values $f(n)$ and $g(n)$ are both strictly positive and hence nonzero for all n (as is C), we can take their reciprocals, which satisfy:
$$\frac{1}{f(n)} \ge \frac{1}{Cg(n)} \quad \text{for all } n \ge n_0$$
This shows that
$$\frac{1}{g(n)} \le C\frac{1}{f(n)},$$
so that
$$\frac{1}{g(n)} = O\left(\frac{1}{f(n)}\right)$$

(c)

$$f(n)^2 = O\left(g(n)^2\right)$$

Solution

True also. Just write out the meaning of the assumption $f(n) = O(g(n))$ in terms of the definition of big O and square both sides. Since the squaring function $x \mapsto x^2$ is monotone increasing for $x > 0$, its application preserves the relative ordering of the left and right sides of the original relationship. Write out the details as an exercise.

(d)

$$2^{f(n)} = O\left(2^{g(n)}\right)$$

Solution

This one is false! Yes, base 2 exponentiation is monotone in the exponent, so it preserves relative ordering as in the case of squaring. However, multiplying the exponent by a constant actually changes the asymptotic growth rate, which can mess things up. Follow along:

$$f(n) \le Cg(n) \Rightarrow 2^{f(n)} \le 2^{Cg(n)}$$

However:

$$2^{Cg(n)} = \left(2^{g(n)}\right)^C$$

The C has gone from a mere multiplicative constant to an outer exponent.
Here is a specific counterexample:

$$f(n) = 2\log n, \quad g(n) = \log n$$

For these particular functions, the conclusion clearly fails:

$$2^{f(n)} = n^2, \quad 2^{g(n)} = n$$

2. Algorithm 1 draws a recursive pattern (finite fractal) based on a given image. An example based on a photo of the Prague Astronomical Clock is displayed approximately in Fig. 1, where finer details have been suppressed (the recursion depth has been limited to three).

```
Algorithm 1: Recursive Image Generation
Input: An image $im$, location coordinates $x, y$, size $n$.
Output: Displays recursive image based on $im$ of size $n \times n$ at $(x, y)$ (e.g., Fig. 1).
DRAW($im, x, y, n$)
(1)   if $n > 0$
(2)       DRAWONCE($im, x, y, n$)
(3)       DRAW($im, x, y + n, \lfloor n/2 \rfloor$)
(4)       DRAW($im, x + n, y, \lfloor n/2 \rfloor$)
(5)       DRAW($im, x, y - n, \lfloor n/2 \rfloor$)
(6)       DRAW($im, x - n, y, \lfloor n/2 \rfloor$)
```

Let $C(n)$ denote the total number of times that the drawOnce function is called by the invocation draw(im, x, y, n).

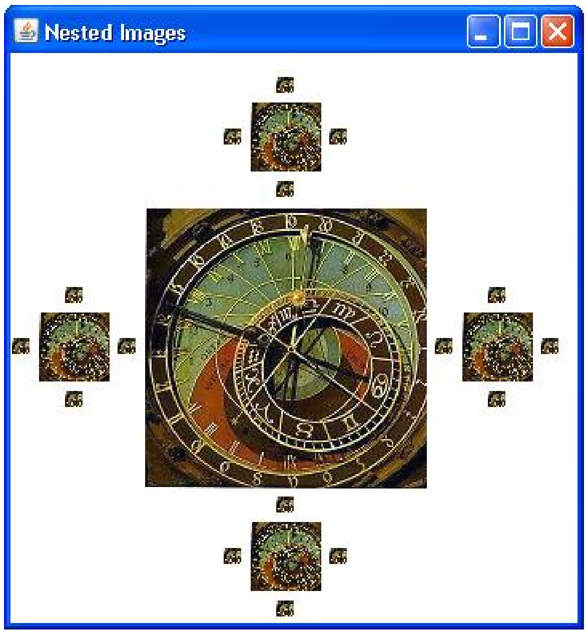

Figure 1: Recursive Image (individual clock image from http://utf.mff.cuni.cz/Relativity/orloj.htm)

(a) Write a recurrence relation for $C(n)$. Include a base case. Explain how to derive the recurrence relation from the pseudocode for Algorithm 1.

Solution

Since each level of the recursion calls drawOnce exactly once, and since no calls are made when $n = 0$, we have the following:

$$C(n) = 1 + 4 * C(\lfloor n/2 \rfloor) \text{ for } n > 0$$
$$C(0) = 0$$

(b) Solve the recurrence relation to find a big Θ expression for the number of drawOnce calls as a function of n. Explain.

Solution

At recursion depth k, there are 4^k different copies of draw, each of which makes one drawOnce call (unless the base level is reached). Thus, the total number of calls at depth k is 4^k. The recursion depth as determined by the base case together with the successive divisions by 2 is $\log_2 n$. Thus, the total number of steps is equal to the following sum:

$$s(n) = \sum_{k=0 \ldots \log_2 n} 4^k$$

This is the n-th partial sum of a geometric series, which equals

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

The same result follows more expediently by applying the Master Theorem from chapter 2 of the textbook (note that $a = 4,\ b = 2,\ d = 0$ here, so $d = 0 < 2 = \log_b a$).

(c) How would the recurrence relation for the number of drawOnce calls and the resulting big Θ growth rate change if there were *five* recursive calls to draw instead of four in Algorithm 1 (with the rest of the algorithm remaining the same)? Explain.

Solution

By the Master Theorem, with $a = 5,\ b = 2,\ d = 0$ (note that $d = 0 < \log_b a$ here also), we find that

$$C(n) = \Theta(n^{\log_2 5})$$

The exponent of n here is approximately 2.32. The algorithm is therefore asymptotically slower in the case of five recursive calls.

3. You are designing a divide and conquer algorithm for a certain computational task. Your approach will involve dividing an input instance of size n into a certain number of subproblems, each of size $n/3$. Assuming that the gluing time needed to recombine the subsolutions is $\Theta(n^2)$, what is the largest number of subproblems for which the overall running time of the algorithm on instances of size n will be $O(n^2)$? Give a concise explanation, and state your answer clearly.

Solution

Apply the Master Theorem with $b = 3,\ p = 2$. Since the exponent in the desired asymptotic upper bound on the running time, n^2, is the same as the gluing time exponent in this situation, the Master Theorem shows that the running time will only be $O(n^2)$ if the recurrence is dominated by the gluing time, that is, only if the number of subproblems a is strictly less than the value $b^p = 3^2 = 9$. Therefore, the largest possible number of subproblems is eight.

4. Consider the task of detecting whether a given array has an element that is repeated in more than half of the positions of the array. For example. the value 2 is the majority element in the array $\{3, 2, 5, 2, 3, 2, 7, 2, 2\}$, while the array $\{5, 8, 8, 3, 10, 8, 5, 8\}$ has no majority element. In the present task you will develop a divide and conquer algorithm for solving this problem, and you will analyze its running time.

(a) Suppose that an array $a[1...n]$ has an element v_L that occurs in strictly more than half of the first $\lfloor n/2 \rfloor$ positions of a, and an element v_H that occurs in strictly more than half of the remaining $\lceil n/2 \rceil$ positions. Argue carefully why if there is an element v of a that occurs in over half of all n positions of a, then v must be either v_L or v_H. Note that it would *not* be enough for v to occur more times in a than either v_L or v_H. We explicitly require that v occur in strictly more than half of all positions of the entire array.

Solution

Suppose that v is any element of $a[1...n]$ other than the elements v_L and v_H described above. Then v occurs in fewer than half of the positions among $1...\lfloor n/2 \rfloor$ (otherwise v

would be v_L) and in fewer than half of the remaining positions $\lfloor n/2 \rfloor + 1...n$ (otherwise v would be v_H). Note that there can't be more than one majority element in either half, since a majority element leaves fewer than half of all positions available for other elements. Adding up, the total number of occurrences of v in the entire array $a[1...n]$ is therefore less than $n/2$. This shows that such a v cannot be the majority element of $a[1...n]$. Equivalently, a majority element of $a[1...n]$ must necessarily be either v_L or v_H as stated.

(b) Using 4a, design a divide and conquer algorithm that returns the majority element of an array or the value NO_SUCH_ELEMENT as appropriate. Give detailed pseudocode.

Solution

Algorithm 2 provides a divide and conquer algorithm for the majority element task based on the above analysis. The countOccurrences function just does a linear scan of the array argument, counting occurrences of the first argument in the array.

Algorithm 2: dcMajority
Input: array $a[1...n]$.
Output: An element v of a that occurs in at least $\lfloor n/2 \rfloor + 1$ positions of a, or the value NO_SUCH_ELEMENT if no such v exists.

```
dcMajority($a[1...n]$)
(1)   if $n$ equals 0 then return NO_SUCH_ELEMENT
(2)   if $n$ equals 1 then return $a[1]$
(3)   $v_L$ = dcMajority($a[1...\lfloor n/2 \rfloor]$)
(4)   $v_H$ = dcMajority($a[\lfloor n/2 \rfloor + 1...n]$)
(5)   if $v_L$ equals $v_H$ then return $v_L$
(6)   count$_L$ = countOccurrences($v_L$, $a[1...n]$)
(7)   count$_H$ = countOccurrences($v_H$, $a[1...n]$)
(8)   if count$_L > n/2$ then return $v_L$
(9)   if count$_H > n/2$ then return $v_H$
(10)  return NO_SUCH_ELEMENT
```

The fact that Algorithm 2 correctly solves the majority element task as stated above follows from the observation in 4a.

(c) Analyze the running time of your algorithm from 4b. Provide a careful explanation, including the relevant recurrence relation.

Solution

The running time of countOccurrences(v, $a[1...n]$) is $\Theta(n)$. In the worst case, the time needed by dcMajority($a[1...n]$) to combine the subsolutions resulting from the recursive calls to dcMajority is $\Theta(n)$, due to the countOccurrences calls. In any case, the combination time is $O(n)$. Since there are two recursive calls, we get the following recurrence relation for the running time, $T(n)$:

$$T(n) = 2T(\lceil n/2 \rceil) + O(n)$$
$$T(0) = O(1)$$

We apply the Master Theorem to find the solution. We have $a = 2,\ b = 2,\ p = 1$ since there are 2 subproblems, each of size $n/2$, and since the combination time is $O(n^1)$. For these values, $a = b^p$, hence we are in the critical case where both the branching factor of the recursion tree and the exponent of the combination term contribute to the overall time. The solution is

$$T(n) = O(n \log n)$$

This is indeed more efficient than calling numOccurrences for every element of the array and returning the element that occurs more than $n/2$ times if it exists – that would take time $\Omega(n^2)$ in the worst case.