please don't grade

CSE 548 – Analysis of Algorithms, Spring 2013

Assignment #1

Duo Xu (#108662210)

partner: Yu-Yao Lin (#109090793)

<u>Problem 1</u>

(1) $blah(n) = \Theta(nlg^2n)$

This is a recursive function, and the only input parameter is n. It first goes through a nested loop, naming it as $f(n)$, then it calls itself twice, which can be regard as a recursion. The base case is when n equals 0, it returns value 1.

The recurrence relation can be written as follows:

$blah(n) = 2 * blah(n/2) + f(n)$ where $n > 0$

$blah(n) = 1$ where $n = 0$

Using the master theorem, we can easily get the time-complexity of this function. $f(n)$ is $\Theta(nlgn)$, and the second case of the master theorem applies to this recurrence relation.

Thus, the time-complexity of this function is $blah(n) = \Theta(nlg^2n)$

(2) $blah(n) = O(2^n)$

This is a recursive function, and the only input parameter is n. It uses a loop to call itself n times with the input parameter ranging from 0 to n-1. The base case is when n equals 0, it returns value 7.

The recurrence relation can be written as follows:

$blah(n) = blah(0) + blah(1) + ... + blah(n-1)$ where $n > 0$ (1)

$blah(n) = 7$ where $n = 0$

So, $blah(n-1) = blah(0) + blah(1) + ... + blah(n-2)$ (2)

(1)-(2), $blah(n) = 2 * blah(n-1)$ (3)

(3) is a geometric progression, thus $blah(n) = 7 * 2^{n-1} = O(2^n)$

Problem 2

(1) False.

A counterexample will be $f(n) = n$ and $g(n) = n^2$.

Obviously, $f(n) = o(g(n))$.

However, $\log(f(n)) = \log(n)$ and $\log(g(n)) = 2\log(n)$,

thus, if $c = \frac{1}{2}$, $\log(f(n))$ always equals $c * log(g(n))$, for any n.

(2) True.

$f(n) = O(g(n))$ means there exist positive constants c and n0 such that $1 < f(n) \leq c * g(n)$ for all $n \geq n_0$

As $\log(x)$ is a monotonically increasing function and $g(n)$ grows faster than $f(n)$, there must exist positive constants c and n0 such that $0 < \log(f(n)) \leq c * \log(g(n))$ for all $n \geq n_0$

(3) True.

$f(n) = o(g(n))$ means for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $1 < f(n) < c * g(n)$ for all $n \geq n_0$

As 2^x is a monotonically increasing function and $g(n)$ grows faster than $f(n)$, for any positive constant $c > 0$, there must exist a constant $n_0 > 0$ such that $0 < 2^{f(n)} < c * 2^{g(n)}$ for all $n \geq n_0$

(4) False.

A counterexample will be $f(n) = n$ and $g(n) = \frac{n}{2}$.

Obviously, $f(n) = O(g(n))$.

However, $2^{f(n)} = 2^n$ and $2^{g(n)} = \sqrt{2^n}$,

thus, when $n \to \infty$, $\frac{2^{f(n)}}{2^{g(n)}} \to \infty$ and we cannot find a n_0 and c such that $0 < 2^{f(n)} \leq c * 2^{g(n)}$ for all $n \geq n_0$.

Problem 3

(1) ℓ multiplications

Name a^N as function $f(N)$, when $N = 2^\ell$,

$f(N) = f(N/2)^2$

$f(N/2) = f(N/4)^2$

$\vdots$

$f(2) = f(1)^2$

$f(1) = a$

Thus it requires $logN$ or ℓ multiplications.

(2) At most $2 * \lfloor logN \rfloor$ multiplications

When N is an arbitrary integer, it is always able to be written as binary format, which means $N = n_k * 2^k + n_{k-1} * 2^{k-1} + ... + n_1 * 2^1 + n_0 * 2^0$, where $n_i \in \{0, 1\}$ and $k = \lfloor logN \rfloor$.

Thus we can use an array of size k+1 to track the result $a^{2^0}, a^{2^1}, ..., a^{2^k}$. a^{2^0} needs 0 multiplication, $a^{2^k} = a^{2^{k-1}} * a^{2^{k-1}}$, only 1 more multiplication than $a^{2^{k-1}}$. So in order to fill this array, we need $k = \lfloor logN \rfloor$ multiplications. Then we need multiply all the subs whose prefix $n_i = 1$, which means we need another $t \leq \lfloor logN \rfloor$ multiplications.

So totally it requires $k + t \leq 2 * \lfloor logN \rfloor$ multiplications, which is $O(logN)$.

Problem 4

a. $T(n) = \Theta(n^{lg3})$

Using the master theorem, $a = 3, b = 2, f(n) = n, f(n) = O(n^{\log_b^{a-\epsilon}})$, case 1 applies to this recurrence. Hence $T(n) = \Theta(n^{lg3})$.

b. $T(n) = \Theta(lglgn)$

Let $m = logn$, thus $2^m = n$, then $S(m) = T(2^m) = T(2^{m/2}) + 1 = S(m/2) + 1$

Using the master theorem, $a = 1, b = 2, f(m) = 1, k = 0, f(m) = \Theta(m^{\log_b^a})$, case 2 applies to this recurrence. Hence $S(m) = \Theta(lgm)$, and $T(n) = \Theta(lglgn)$

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

$$T(n) = T(n-1) + n$$
$$T(n) = T(n-2) + (n-1) + n$$
$$T(n) = T(n-3) + (n-2) + (n-1) + n$$
$$\vdots$$
$$T(n) = \Theta(1) + 3 + 4 + ... + n$$
$$T(n) = (3 + n) * (n-2)/2 + \Theta(1) = \Theta(n^2)$$

d. $T(n) = \Theta(lgn)$

Let $m = lgn$, thus $2^m = n$, then $S(m) = T(2^m) = 2T(2^{m/2}) + 1 = 2S(m/2) + 1$

Using the master theorem, $a = 2, b = 2, f(m) = 1, f(m) = O(m^{\log_b^{a-\epsilon}})$, case 1 applies to this recurrence. Hence $S(m) = \Theta(m)$, and $T(n) = \Theta(lgn)$

Problem 5

$(a + bi)(c + di) = ac + adi + bci - bd = ac - bd + (ad + bc)i$

Thus, we first calculate $(a + b)(c + d)$, which equals $ac + ad + bc + bd$. (1)

Then we calculate ac and bd. (2)

Then from (1) and (2), $ad + bc = (a + b)(c + d) - ac - bd$.

Now we know all the subs and we are able to calculate the final result, only using 3 multiplications with a few extra additions.