ECE-250 – Algorithms and Data Structures (Winter 2012)

Assignment 2 – Suggested solutions

1 - Place the following functions in ascending order by asymptotic behaviour, justifying for each case (you may use either limits or the formal definition to justify). That is, put them in sequence f_1, f_2, f_3, $\cdots$ such that $\mathrm{f}_k(n) = O(\mathrm{f}_{k+1}(n))$:

$\mathrm{f}(n) = 10n^2 + 1$
$\mathrm{f}(n) = 5$
$\mathrm{f}(n) = \ln(n+1)$
$\mathrm{f}(n) = 2^n$
$\mathrm{f}(n) = 10^n$
$\mathrm{f}(n) = \log_{10} n$
$\mathrm{f}(n) = \lg(64\,n^2)$
$\mathrm{f}(n) = n^{0.00001}$

Solution:

The order (or rather, one possible order, given that a few of them are equivalent — so, we could see this as a weak ordering) is: $\mathrm{f}_1(n) = 5$, $\mathrm{f}_2(n) = \ln(n+1)$, $\mathrm{f}_3(n) = \log_{10} n$, $\mathrm{f}_4(n) = \lg(64\,n^2)$, $\mathrm{f}_5(n) = n^{0.00001}$, $\mathrm{f}_6(n) = 10n^2 + 1$, $\mathrm{f}_7(n) = 2^n$, $\mathrm{f}_8(n) = 10^n$

From f_1 to f_2, the reason is quite obvious, since $\displaystyle\lim_{n\to\infty} \frac{5}{\ln(n+1)}$ clearly is 0.

The trickier ones are perhaps f_2, f_3 and f_4; but we recall that all logarithms (to different bases) are multiples of each other ($\log_a x = \log_b x \log_a b$). Having $(n+1)$ as the argument clearly makes no difference (for $n > 1$, $\lg(n+1) < 1 + \lg n$ — *why?*), and f_4 may look tricky, but we notice that it is nothing more than $\lg 64 + 2\,\lg n = \Theta(\log n)$

For f_5, we recall that any power n^α grows faster than $\log n$ for every $\alpha > 0$, no matter how small (the limit may be easily determined using L'Hôpital's rule). Quadratic growth in f_6 clearly follows, and then exponential.

We recall that different exponentials are not Θ of each other; larger bases dominate, as can be seen by the limit:

$$\lim_{n\to\infty} \frac{10^n}{2^n} = \lim_{n\to\infty} 5^n = \infty$$

Showing that $\mathrm{f}_7(n) = o(\mathrm{f}_8(n))$ and therefore big-Oh as well.

2 - Show that (or rather, explain in detail why) the run time of the following loop is $\Theta(\log n)$. You may assume that $n > 5$ in every instance of running that loop:

```
for (int i = 5; i < n; i *= 2)
{
    sum += i*i;
}
```

Solution:

We recognize that loop, where the control variable is doubled at each iteration. If the loop started at `i = 1`, then it would execute $\lg n$ times (plus/minus rounding, of course, but in asymptotic notation, that detail makes no difference).

The observation being that this loop executes that number of times minus three (because the only difference between the two is the initial three passes to get from 1 to past 5 — again, plus/minus rounding), and so it executes $\Theta(\log n) - \Theta(1)$ times, which is $\Theta(\log n)$).

A more formal way to look at it could be noticing that after one iteration, i reaches 5×2; after two iterations, 5×2^2; then 5×2^3; after k iterations, we reach 5×2^k — the loop stops after k iterations if i reaches n; that is, when $5 \times 2^k = n$, or $\lg 5 + \lg 2^k = \lg n \;\Rightarrow\; k = \lg n - \lg 5$, showing that the loop executes $\Theta(\log n)$ times.

3 - Determine the run time of the following function. You may assume that the value of n is a power of 2.

```
void merge_sort (int * array, int n)
{
    if (n == 1)
    {
        return;
    }

    merge_sort (array, n/2);
    merge_sort (array + n/2, n/2);

    merge (array, n);   // This function works in linear time when
                        // measured with respect to its second argument
}
```

Solution:

This recursive function leads to a recurrence relation; if we use $\mathrm{T}(n)$ to denote its run time, simply adding the run time of the fragments that contribute to it, we get: $\Theta(1)$ for the initial if and return; then, twice the run time of the function for half-size arguments, or $2 \cdot \mathrm{T}(n/2)$, and then `merge`, taking $\Theta(n)$:

$$\mathrm{T}(n) = 2\mathrm{T}(n/2) + \Theta(n)$$

The initial condition is clearly given by the recursion's base case (`if (n == 1)`), which is $\Theta(1)$.

Replacing $\Theta(n)$ by a generic representative, say, $c_1 n$ for a constant $c_1 > 0$, and solving by repeated substitution, we obtain, after k levels of recursion:

$$\mathrm{T}(n) = 2^k \mathrm{T}(n/2^k) + k\, c_1 n \quad \text{with } \mathrm{T}(1) = c_2$$

At $k = \lg n$ we reach T(1); we substitute and obtain the result:

$$\mathrm{T}(n) \;=\; nc_2 + \lg n \, c_1 n \;=\; \Theta(n \log n)$$

4 - Determine the run time of the following function, and briefly explain why we obtain the given result [. . .]

```
bool mystery_function (const int * primes, int n, int subset_product)
{
    if (validation_function (subset_product, primes, n))
    {
        return true;
    }

    if (mystery_function (primes + 1, n-1, subset_product)
        || mystery_function (primes + 1, n-1, subset_product * primes[0]))
    {
        return true;
    }

    return false;
}
```

Solution:

By visual inspection, we notice that this function ends up checking all possible combinations of including or not each prime number in a partial product (subset_product), and thus would execute 2^n times, with linear time each execution, for a runtime of $\Theta(n\,2^n)$

The recurrence relation that we obtain is as follows: we have $\Theta(n)$ for the validation function (it is a condition, but evaluating it requires calling a function that we're told has run time $\Theta(n)$). Then, we add twice the run time of mystery function for size $n-1$ — the rest of the steps are just expressions and simple operations, each one taking $\Theta(1)$, and being a fixed number of them, they add up to $\Theta(1)$. Thus, we have:

$$\mathrm{T}(n) = 2\mathrm{T}(n-1) + \Theta(n) \quad \text{with} \;\; \mathrm{T}(0) = \Theta(1)$$

(it will be clear why in this case it is more convenient to use 0 as the initial condition — clearly, for a problem of size 0, the cost is constant time)

Once again, repeated substitution (assuming $c_1 n$ and c_2 as generic representatives of the classes in the recurrence relation) can be used to obtain the solution:

$$\begin{aligned}
\mathrm{T}(n) &= 2(\mathrm{T}(n-2) + c_1(n-1)) + c_1 n \\
&= 4\mathrm{T}(n-2) + 3c_1 n - 2c_1 \\
&= 8\mathrm{T}(n-3) + 7c_1 n - 10c_1 \\
&= \cdots \\
&= 2^k \mathrm{T}(n-k) + (2^k - 1)n + \Theta(1)
\end{aligned}$$

At $k = n$ we reach $\mathrm{T}(0) = c_2$, so we obtain:

$$\mathrm{T}(n) \;=\; 2^n c_2 + (2^n - 1)n + \Theta(1) \;=\; \Theta(n\,2^n)$$

10% Bonus Marks:

Determine the run time of the function `populate_array`

```
void resize (int * & array, int size, int new_size)
{
    int * new_array = new int [new_size];
    for (int i = 0; i < size && i < new_size; i++)
    {
        new_array[i] = array[i];
    }
    delete [] array;
    array = new_array;
}

int sum (const int * array, int size)
{
    int total = 0;
    for (int i = 0; i < size; i++)
    {
        total += array[i];
    }
    return total;
}

int * populate_array (int n)
{
    int arr_size = 1;
    int * array = new int [arr_size];
    array[0] = 1;

    for (int i = 1; i < n; i++)
    {
        if (i >= arr_size)
        {
                // If running out of space, double the array size
            resize (array, arr_size, 2*arr_size);
            arr_size *= 2;
        }
            // Each element gets assigned with the sum of
            // all previous elements
        array[i] = sum (array, i);
    }

    return array;    // return pointer to allocated and populated array
}
```

Solution:

The function `resize` takes linear time (with respect to its `size` argument), since it has to copy these many elements when increasing the size (which is always the case when called from `populate_array`).

The function `sum` also takes linear time with respect to its `size` argument.

The key detail to observe is the fact that the array is resized to increase in geometric progression; as discussed in class, at the end of the slides from 2012-01-20, this leads to an append operation that, on average, runs in constant time (some appends require a resize, but then the following several appends do not).

Thus, apart from all the $\Theta(1)$ contributions, the only terms we need to consider are the calls to the function `sum`. We see that the first call adds 1 element, then 2, then 3, and so on, until $n-1$ (when called for element $n-1$, it adds the elements 0 to $n-2$—a total of $n-1$ elements added).

We recognize this as the arithmetic sum from 1 to $(n-1)$, leading to $n(n-1)/2$ operations. Thus, `populate_array` runs in $\Theta(n^2)$.

10% Bonus Marks:

Prove, using the formal definition of $\boldsymbol{\Theta}$, that $\lg(n!) = \Theta(n \lg n)$

Solution:

We need to find positive constants c_1, c_2, and N such that

$$0 \leqslant c_1 n \lg n \leqslant \lg n! \leqslant c_2 n \lg n$$

The left-most inequality is trivial, since the function is positive, and c_1 is positive. The right-most inequality is trivially achieved with $c_2 = 1$, since $\lg n! \leqslant n \lg n$

Thus, we have to find c_1 and N to satisfy the remaining inequality, $c_1 n \lg n \leqslant \lg n!$ for all $n \geqslant N$.

Consider the expression

$$\lg n! \;=\; \lg n + \lg(n-1) + \lg(n-2) + \;\cdots\; + \lg 3 + \lg 2$$

and consider the first half of those terms (the logs with argument greater than $n/2$). Given that $\lg(n/2) = \lg n - \lg 2 = \lg(n) - 1$, and given that lg is a strictly increasing function, that means that for all $k \geqslant n/2$, we have $\lg k \geqslant \lg(n) - 1$.

Thus,

$$\lg n! > \lg n + \lg(n-1) + \lg(n-2) + \;\cdots\; + \lg(n/2 + 1) + \lg(n/2) \;\geqslant\; \frac{n}{2}(\lg(n) - 1)$$

Thus, if we can find c_1 that satisfies the inequality $c_1\, n \lg n \leqslant \frac{n}{2}(\lg(n) - 1)$, then we're done, since the expression on the right is less than $\lg n$!

$$c_1\, n \lg n \leqslant \frac{n}{2}(\lg(n) - 1) \Rightarrow c_1 \lg n \leqslant \frac{1}{2}(\lg(n) - 1)$$

And this is satisfied, for some N with *any* $c_1 < \frac{1}{2}$. We can easily verify this; $0 < c_1 < \frac{1}{2}$ translates into $\lg n \leqslant a(\lg(n) - 1)$ with $a > 1$ (since $a = (1/2)/c_1$ and $c_1 < \frac{1}{2}$), and this means that $(a-1)\lg n \geqslant 1$, which holds for $n > 2^{1/(a-1)}$.