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

Assignment 2

Due on Friday, January 27, in class.

All questions have equal weight for the assignment grade (except for the bonus marks questions).

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 $f_k(n) = O(f_{k+1}(n))$:

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

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;
}
```

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
}
```

4 - Determine the run time of the following function, and briefly explain why we obtain the given result (that is, describe the typical behaviour that explains the given run time class — for example, binary search is a typical program's behaviour that leads to logarithmic time; checking all the values for each iteration over each of the values is a typical behaviour exhibiting quadratic time, etc.).

Doesn't really matter what the function does (in fact, I doubt that it would do anything useful). The parameter primes "magically" contains the list of the first `n` prime numbers, and since it is given to the function, the run time does not include the time to load those values (if any).

You may assume that `validation_function` runs in linear time measured with respect to its second parameter. You may also assume that the main program (or in any case the initial caller) will always call the function passing a value of 1 as the `subset_product` parameter.

```
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;
}
```

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
}
```

10% Bonus Marks:

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

Hint: Think of the "intuition" that we could use to see why this is the case:

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

But then, the function lg grows so slowly, that for the first half of those terms (up to $\lg(n/2)$), the value of the function practically has not changed... So that's at least $n/2$ times that we're adding something that is very close to $\lg n$ — so, we should be sufficiently close to a multiple of $n \lg n$ that we might even neglect the rest of the terms.

Comment about the relevance of the above:

When proving that any sort algorithm can not run faster than $n \log n$, the argument uses the fact that there are $n!$ (n factorial) permutations, one of them being the one with the values sorted, and if we put those permutations in order (some lexicographical order), then that means that we can do binary search on the set of possible permutations — and clearly, there is no way that we can do better than binary search.

Thus, we can find the right permutation in $\lg n!$ (binary search on a set of size $n!$), and no faster than that, asserting this as a lower-bound. So, it seems interesting to show that this is asymptotically equivalent to $n \log n$