CS3933 Fall 2007 – Solo Assignment 1
Due Wednesday September 26, 2007, at 5pm.
Sample Solution

1. List the following functions in ascending order of growth rate.
   $\sqrt{n}$, $4^{\log_4 n} = n$, $n \log_2 n$, $\sqrt{n}^3$, $4^{\log_2 n} = n^2$, $n^{\log_2 n} = 4^{\log_4 n \cdot \log_2 n}$, 4^n

2. Consider the following algorithm:
   Algorithm *Accumulate*(A, n):
   Input: array A of n integers, indexed from 0.
   $i \leftarrow 0$
   $R \leftarrow A[0]$
   $S \leftarrow 0$
   while $i < n - 1$ do
    while ($R \leq A[i]$ and $i < n - 1$)
     $i \leftarrow i + 1$
     $R \leftarrow 2 * R$
    $S \leftarrow S + R$
    $R \leftarrow A[i]$
   return S

   Analyse the running time of the algorithm using O, Ω, and Θ, as appropriate.

   Each of the nested loops can iterate up to n times, but they cannot both do so at the same time. To determine the overall running time, we need to consider all iterations of the inner while loop, independently of which outer loop iteration they are a part.

   The value of i starts at 0, and is increased by 1 for each iteration. The loops stop iterating when $i = n - 1$, and i is never decreased, so the total number of iterations of the inner while loop is exactly $n - 1$. The rest of the algorithm contributes only an additional constant factor, so the overall running time of this algorithm is $\in \Theta(n)$.

3. Consider the following algorithm:
   Algorithm *Numbers*(A,n):
   Input: array A of n positive integers.
   $S \leftarrow 0$
   for i from 1 to $n - 1$ do
    for j from $i + 1$ to n do
     $P \leftarrow 1$

```
        for k from 1 to i · j do
            P ← P · A[j]
        S ← S + P
return S
```

(a) Analyse the running time of this algorithm, using O, Ω, and Θ, as appropriate.

Upper bound: The outer for loop (on i) iterates exactly $n-1$ times. For each of the outer iterations, the middle loop (on j) iterates at most $n-1$ times. For each of these middle iterations, the innermost loop iterates at most $(n-1)\cdot n$ times. The rest of the algorithm takes constant time, so the algorithm as a whole runs in time $\in O((n-1)^3 \cdot n) = O(n^4)$.

Lower bound: The middle loop requires $j > i$ (so, to have many j values, i cannot be too high), and the inner loop requires $k \leq i \cdot j$ (so, to have many k values, each of i and j cannot be too low).

Consider the outer loop iterations where $\frac{n}{3} < i < \frac{2n}{3}$. Further consider the middle loop iterations where $j > \frac{2n}{3}$. Then $j > i$ as required by the algorithm. There are at least $\frac{n}{3} - 1$ iterations of the outer loop, and each of these outer iterations will have at least $\frac{n}{3} - 1$ iterations of the middle loop. For each of these, the inner loop has $i \cdot j$ iterations, which is at least $\frac{n}{3} \cdot \frac{2n}{3} = \frac{2n^2}{9}$.

Therefore there are at least $(\frac{n}{3} - 1)^2 \cdot \frac{2n^2}{9}$ total interations of the inner loop, so the algorithm runs in time $\in \Omega((\frac{n}{3} - 1)^2 \cdot \frac{2n^2}{9}) \;=\; \Omega(n^4)$.

Since this algorithm runs in time $\in O(n^4)$ and $\in \Omega(n^4)$, it runs in time $\in \Theta(n^4)$.

(b) Now that you have determined the degree of the polynomial, figure out the exact polynomial function for how many times the statement $P \leftarrow P \cdot A[j]$ is executed in terms of the input size n.

Determine the polynomial by implementing the algorithm, testing it for various sizes n, and solving for the unknown polynomial coefficients. Please submit your test case results (n and corresponding count) as well as your polynomial function.

Note that how you solve for your coefficients is not an essential part of the assignment; it's just a step in finding the function (so you do not need to show your work for the coefficient solving, and indeed can solve them however you like).

n	c(n)
1	0
2	2
3	11
4	35
5	85
6	175

This data produces the polynomial $\frac{n^4}{8} + \frac{n^3}{12} - \frac{n^2}{8} - \frac{n}{12}$. Note that $n = 6$ was not needed to find this polynomial, but was included for confirmation.

4. Consider a modified mergesort algorithm so that it splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-quarter. Write an algorithm for this modified sorting algorithm. Asymptotically analyze it by establishing a recurrence equation first, then analysing the recurrence tree of the algorithm. You can make assumptions about the instance sizes considered in order to simplify your math, as long as the sizes can still be arbitrarily large.

   The following recursive algorithm should be called as QUADSORT(A, 0, $n-1$). The merge call is to a linear-time four-way merge.

   Algorithm QUADSORT(array A, integer a, integer b)
   if $(a = b)$ return
   $mid1 \leftarrow a + \lfloor \frac{b-a}{4} \rfloor$
   $mid2 \leftarrow a + \lfloor \frac{b-a}{2} \rfloor$
   $mid3 \leftarrow b - \lceil \frac{b-a}{4} \rceil$
   QUADSORT(A, a, $mid1$)
   QUADSORT(A, $mid1 + 1$, $mid2$)
   QUADSORT(A, $mid2 + 1$, $mid3$)
   QUADSORT(A, $mid3 + 1$, b)
   merge ($A[a..mid1]$, $A[mid1 + 1..mid2]$, $A[mid2 + 1..mid3]$, $A[mid3 + 1..b]$)
   storing result of merge in $A[a..b]$.
   return

   Analysis:
   For simplicity, let us assume that n is a power of 4. Then our running time is given by the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 4 \cdot T(\frac{n}{4}) + c \cdot n & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence will have height $\log_4 n$, since the height h is the level where

$$\frac{n}{4^h} = 1$$

which is true if and only if $h = \log_4 n$.

At level k, where level 0 is the root, there will be 4^k nodes, each of size $\frac{n}{4^k}$. Since the amount of work done for each node is linear, then the amount of time needed at level k is

$$W(k) \;=\; 4^k \cdot c \cdot \frac{n}{4^k} \;=\; cn \cdot \left(\frac{4}{4}\right)^k \;=\; cn$$

Summing over all levels, the total time taken for the algorithm is

$$\begin{aligned} T(n) \;&=\; \sum_{k=0}^{\log_4 n} W(k) \\ &=\; \sum_{k=0}^{\log_4 n} cn \\ &=\; cn \cdot \left(\sum_{k=0}^{\log_4 n} 1\right) \\ &=\; cn \cdot (\log_4 n + 1) \end{aligned}$$

So $T(n) \in \Theta(n \log n)$.