CS3933 Winter 2007 – Solo Assignment 1
Due Wednesday January 31, 2007, at 5pm.
Sample Solution

1. (3 marks)

   List the following functions in ascending order of growth rate.:
   $\sqrt{n}$, $4^{\log_2 n}$, $n^2 \log_2 n$, $\sqrt{n}^5$, $n^{\log_2 n}$, 2^{2n}, $\sqrt{n}^n$

2. Prove or disprove, using the definitions of O, Ω, and Θ, that

   (a) (3 marks) $3n \log_2 n \ \in \ O(5n^2)$

   Need $c > 0$, $n_0 \geq 0$ such that $\forall n \geq n_0$, $3n \log_2 n \leq c \cdot 5n^2$.

   Let $c = 1$ and $n_0 = 1$. Then, since $\log_2 n < n \ \ \forall n \geq 1$,
   $3n \log_2 n < 3n \cdot n = 3n^2 < 5n^2$.

   (b) (3 marks) $2^n \ \notin \ \Omega(4^n)$

   Need to show that $\forall c > 0, \ n_0 \geq 0, \exists n \geq n_0$ such that $2^n < c \cdot 4^n$.

   Let $n = \max(-\log_2 c + 1, n_0)$. Then, since $-\log_2 c < n$,
   $\frac{1}{c} = 2^{-\log_2 c} < 2^n$, so $1 < c \cdot 2^n$.
   Therefore $2^n < c \cdot 2^n \cdot 2^n = c \cdot 4^n$, which is what was to be shown.

3. (5 marks) Consider the following algorithm:
   Algorithm *Spin*(n):
   Input: positive integer n.
   $x \leftarrow 0$
   for i from 1 to n
    for j from i to n
     $x \leftarrow x + 1$
   return x

   Analyse the running time of this algorithm in terms of the size of its input (rather than in terms of n, the input value itself). Use O, Ω, and Θ, as appropriate.

   Since the input is a single arbitrarily large integer, the size of the integer is the number of bits needed to store it, which is $m = \lceil \log_2 n \rceil$. Note that $\log_2 n \leq m < log_2 n + 1$; we can thus ignore the ceiling as long as we do not use m as a base for exponentiation.
   (I am including the ceiling bounds in this sample solution but it is not necessary to use them or the ceiling.)

Upper Bound:
The outer loop (on i) iterates exactly n times, and for each value of i, the inner loop (on j) iterates at most n times. The rest of the algorithm adds at most a constant factor, so the running time of the algorithm as a whole is $\in O(n^2)$. Since $\log_2 n < m$, $n \leq 2^m$, the algorithm runs in time $\in O((2^m)^2) = O(2^{2m})$.

Lower Bound:
If we consider the values for i such that $i \leq \frac{n}{2}$, then there are at least $\frac{n}{2}$ values for i (and iterations of the outer loop), each of which has at least $\frac{n}{2}$ values for j (and iterations of the inner loop). As a whole, then, the algorithm runs in time $\in \Omega(\frac{n}{2} \cdot \frac{n}{2}) = \Omega(n^2)$. Since $m < \log_2 n + 1$, $n > 2^{m+1}$, the algorithm runs in time $\in \Omega((2^{m+1})^2) = \Omega(2^{2m+2}) = \Omega(4 \cdot 2^{2m}) = \Omega(2^{2m})$.

Since the algorithm runs in time $\in O(2^{2m})$ and $\in \Omega(2^{2m})$, then it runs in time $\in \Theta(2^{2m})$, in terms of the size of the input.

4. (6 marks) Consider the following algorithm:
Algorithm *Run*(A, n, m):
Input: 2-dimensional array A of real numbers, with n rows and m columns.

```
i ← 0
j ← 0
while i < n − 1 or j < m − 1 do
    while (A[i, j] < A[i + 1, j] and i < n − 1)
        i ← i + 1
        print "down"
    j ← j + 1
    while (A[i, j] < A[i, j + 1] and j < m − 1)
        j ← j + 1
        print "right"
    i ← i + 1
```

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

Upper Bound:
If we consider the entire algorithm, each loop iteration (of any loop) will increment at least one of i or j. Since i goes from 0 to $n-1$, and j goes from 0 to $m-1$, i can be incremented n times and j can be incremented m times. Therefore the total number of loop iterations cannot exceed $n+m$ (since the number of iterations cannot be larger than the maximum number of increment operations), and the algorithm runs in time $\in O(n+m)$.

Lower Bound:
Considering all iterations (over the entire algorithm), the maximum number of increment operations for a single loop iteration is 2 (incrementing both i and j). Therefore the loops must have at least $\frac{n+m}{2}$ iterations in the worst case, so the algorithm runs in time $\in \Omega(n+m)$.

Since the algorithm runs in $O(n+m)$ time and $\Omega(n+m)$ time, is runs in time $\in \Theta(n+m)$.

5. Consider the following algorithm:
Algorithm *Numbers*(A,n):
Input: array A of n positive integers.

```
S ← 0
for i from 0 to n − 1 do
    for j from 0 to i do
        P ← 1
        for k from i to n − 1 do
            P ← P ∗ A[k]
        for k from i to j do
            P ← P ∗ A[k]
        S ← S + P
return S
```

(a) (5 marks) Analyse the running time of this algorithm, using O, Ω, and Θ, as appropriate.

Upper bound:
Each loop iterates at most n times. Since two of the loops are sequential and are nested inside the outer two loops, and the rest of the algorithm runs in time bounded by a constant, the algorithm runs in $O(n \cdot n \cdot (n+n)) = O(n^3)$ time.
Lower bound:
At least $\frac{n}{3}$ iterations of the outer loop have $\frac{n}{3} \leq i \leq \frac{2n}{3}$.
When $i \geq \frac{n}{3}$, the loop indexed by j iterates at least $\frac{n}{3}$ times (from 0 to at least $\frac{n}{3}$) for each i value.
When $i \leq \frac{2n}{3}$, the first loop indexed by k iterates at least $\frac{n}{3}$ times (from at most $\frac{2n}{3}$ to $n-1$).
As this is a lower bound, we can ignore the work done by the second loop indexed by k (which iterates at most once anyway, since $j \leq i$).

Therefore the algorithm runs in time $\in \Omega(\frac{n}{3} \cdot \frac{n}{3} \cdot \frac{n}{3}) \;=\; \Omega(n^3)$.

Since the algorithm runs in time $\in O(n^3)$ and $\in \Omega(n^3)$, then we can say that it runs in time $\in \Theta(n^3)$.

(b) (4 marks) Now that you have determined the degree of the polynomial, figure out the exact polynomial function for how many times the statement $P \leftarrow P * A[k]$ 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).

Test data:

n	t
1	2
2	6
3	13
4	24

Substituting this data into the general cubic equation $an^3 + bn^2 + cn + d = T(n)$ produces the linear equations:

$a + b + c + d = 2$

$8a + 4b + 2c + d = 6$

$27a + 9b + 3c + d = 13$

$64a + 16b + 4c + d = 24$

Solving this system of equations produces $a = \frac{1}{6}$, $b = \frac{1}{2}$, $c = \frac{4}{3}$, and $d = 0$, so our equation is

$T(n) = \frac{n^3}{6} + \frac{n^2}{2} + \frac{4n}{3}$

6. (7 marks total – 3 for algorithm, 4 for analysis)
Consider a modified mergesort algorithm so that it splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write an algorithm for this ternary 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 TernarySort(A, 0, $n-1$). The merge call is to a linear-time three-way merge.

Algorithm TernarySort(array A, integer a, integer b)
if $(a = b)$ return
$mid1 \leftarrow a + \lfloor \frac{b-a}{3} \rfloor$
$mid2 \leftarrow b - \lceil \frac{b-a}{3} \rceil$
TernarySort(A, a, $mid1$)
TernarySort(A, $mid1 + 1$, $mid2$)
TernarySort(A, $mid2 + 1$, b)
merge ($A[a..mid1]$, $A[mid1 + 1..mid2]$, $A[mid2 + 1..b]$)
    storing result of merge in $A[a..b]$.
return

Analysis:
For simplicity, let us assume that n is a power of 3. Then our running time is given by the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3 \cdot T(\frac{n}{3}) + c \cdot n & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence will have height $\log_3 n$. At level k, where level 0 is the root, there will be 3^k nodes, each of size $\frac{n}{3^k}$. Since the amount of work done for each node is linear, then the amount of time needed at level k is

$$W(k) \;=\; 3^k \cdot c \cdot \frac{n}{3^k} \;=\; cn \cdot \left(\frac{3}{3}\right)^k \;=\; cn$$

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

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_3 n} W(k) \\ &= \sum_{k=0}^{\log_3 n} cn \\ &= cn \cdot \left(\sum_{k=0}^{\log_3 n} 1 \right) \\ &= cn \cdot (\log_3 n + 1) \end{aligned}$$

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