CS3933 Winter 2007 – Group Assignment 1
Due Tuesday January 23, 2007, at 5pm.
Sample Solution

Groups can be of up to 3 people; please list all names of the group members at the top of the assignment.

1. (Taken from Kleinberg and Tardos, ch. 1, questions 1 and 2)
   Consider an arbitrary instance of the Stable Matching Problem.

   (a) Will there always exist a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m? Either explain why or give a counterexample.

   No. Counterexample preferences, for men $\{X, Y\}$ and women $\{A, B\}$:

A	X	Y
B	Y	X

X	B	A
Y	A	B

   In this example, A's first choice is X, but A is not X's first choice. B's first choice is Y, but B is not Y's first choice.

   (b) If such a man and woman exist, will they always be matched with each other in any stable matchings? Either explain why or give a counterexample.

   Yes, if such a man and woman exist, they will always be matched with each other in any stable matching. If not, so man m ranks w first but is matched to w', and woman w ranks m first but is matched to m', then m and w would be an unstable pair and the matching would not be stable.

2. Prove, using the definition of Ω, that for any integer $k \geq 2$, $n! \in \Omega(k^n)$.

   Basic idea: $n!$ has (n-k+1) factors that are all at least k, so we only need to compensate for the other (k-1) factors (which are all at least 1).

   We need to show that $\exists c > 0, n_0 \geq 0$ such that $\forall n \geq n_0, \quad n! \geq c \cdot k^n$.

   Let $c = \frac{1}{k^{k-1}}$. Then we have, $\forall n \geq k$,

$$n! \;=\; \prod_{i=1}^{n} i \;\geq\; \prod_{i=k}^{n} i \;\geq\; \prod_{i=k}^{n} k \;=\; k^{n-k+1} \;=\; \frac{1}{k^{k-1}} \cdot k^n$$

3. Consider the following algorithm:

Algorithm *FactorSum*(A,n):

```
Input: Array A of n real numbers.
S ← 0
for i from 0 to n − 1 do
    for j from i to n − 1 do
        P ← 1
        for k from i to j do
            P ← P ∗ A[k]
        S ← S + P
return S
```

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

Upper bound:
The outer loop (on i) iterates exactly n times. The middle loop (on j) iterates at most n times for each value of i, and the innermost loop (on k) iterates at most n times for each value of j. The rest of the steps in the algorithm contribute at most a constant factor, so the algorithm as a whole takes time $\in O(n^3)$.

Lower bound:
In order for the innermost loop (on k) to have many iterations, there needs to be a large gap between i and j. However, we still need to have many possible values for i and j that have that gap.

If we consider the values for i and j that have $i \leq \frac{n}{3}$ and $j \geq \frac{2n}{3}$, there will be at least $\frac{n}{3}$ values for k that meet the inner loop requirement of $i \leq k \leq j$. These restrictions also satisfy the middle loop condition that $j \geq i$. Since there are at least $\frac{n}{3}$ such values for i and $\frac{n}{3}$ such values for j, the algorithm runs in time $\in \Omega(\frac{n}{3} \cdot \frac{n}{3} \cdot \frac{n}{3}) = \Omega(n^3)$.

Since the algorithm's running time is both $\in O(n^3)$ and $\in \Omega(n^3)$, it is $\in \Theta(n^3)$.

4. Consider the following algorithm:

Algorithm *Jumps*(A, n, m):

Input: Array A of n positive integers.

```
y ← 0
x ← n
for i from 1 to n do
    j ← A[i]
    while j > 0 and x > 0
        x ← x − 1
        j ← j − 1
        y ← y + 1
    x ← x + 1
```

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

Upper bound:
The number of the iterations of the while loop varies depending on the input values and also on the number of previous iterations. To count the number of while loop iterations, we need to look at the total number of iterations (not just considering a single iteration of the for loop, but all iterations together).

Every time the while loop iterates, x is decremented by one. However, the condition on the while loop stop it from iterating further if x is 0, so x will never be negative. Therefore, we only need to determine how many times (over the entire algorithm) x can be decremented without becoming negative.

The starting value of x is n, and x is incremented by one each time the for loop iterates, so it is incremented n times for a total positive contribution of $2n$. So, x can only be decremented $2n$ times without becoming negative, and thus the while loop has a total number of iterations of at most $2n$.

Since this analysis considers the total number of iterations of the innermost loop, we do not multiply it by the number of iterations of the outer loop (the $2n$ result is not "per i value" but over all i values). Since the rest of the algorithm produces at most an additional constant factor, the algorithm takes time $\in O(n)$.

Lower bound:
Since the for loop will iterate exactly n times, this algorithm runs in time $\in \Omega(n)$.

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