CS3933 Winter 2007 – Solo Assignment 2
Due Wednesday October 10, 2007, at 5pm.

1. (a) Design a *divide and conquer* algorithm to solve the following problem in $O(\log n)$ time.

   Given: a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ of distinct integers, that you know is in nondecreasing (ascending) order.

   Find: an index i such that $a_i = i$, or determine that no such index exists

   Note that if there are multiple i for which $a_i = i$, any of them will do.

   (b) Asymptotically analyse your algorithm to show that its running time is $\in O(\log n)$.

This algorithm would be called as *stillpoint*$(A, 1, n)$. It returns 0 if there is no index that meets the criterion.

```
Algorithm stillpoint(A, x, y): returns index value j
if (x > y)
  j ← 0
else
  if a_i = i
    j ← i
  else
    if a_i < i
      j ← stillpoint(i + 1, y)
    else     (a_i > i)
      j ← stillpoint(x, i − 1)
return j
```

Analysis:
This algorithm is recursive. It contains no loops, so each instantiation takes constant time. It recurses at most once per call, on an instance size of at most half (of the size it was called with). Our time recurrence is thus:

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ 1 \cdot T(\frac{n}{2}) + c & \text{otherwise} \end{cases}$$

The recursion depth is at most $h = \log_2 n$, since the size is divided by 2 until it reaches 1 (so h satisfies $\frac{n}{2^h} = 1$). Therefore the time taken by this algorithm is:

$$T(n) \leq \sum_{k=0}^{\log_2 n} 1^k \cdot c$$

so

$$T(n) \leq c \cdot \sum_{k=0}^{\log_2 n} 1 \;=\; c \log_2 n + c$$

Therefore this algorithm runs in time $O(\log n)$.

2. You are analysing a divide & conquer algorithm, and have determined that it has the following characteristics, for input of size n:

   - The algorithm processes the data in constant time to determine 4 parts of the data for the recursion step, where each part has size $\frac{n}{9}$
   - The algorithm recurses 4 times, each time on data of size $\frac{n}{9}$ (as long as $n > 1$)
   - the algorithm merges the results from the recursion in $c \cdot \sqrt{n}$ time (for some constant c)

   Complete the asymptotic analysis of the running time of this algorithm, by setting up a recurrence for $T(n)$ and solving it. Use the recursion tree analysis method (in which you would sum up all levels of the recursion tree). You can make simplifying assumptions about the values of n, as long as n can still be arbitrarily large.

   Assume that n is an integer power of 9. Then we have the followinf recurrence for the running time $T(n)$:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 4 \cdot T(\frac{n}{9}) + c \cdot \sqrt{n} & \text{otherwise} \end{cases}$$

   The recursion tree for this recurrence will have deepest level $h = \log_9 n$, since it divides the size by 9 until it is reduced to 1 (so $\frac{n}{9^h} = 1$).

   At level k, where level 0 is the root, there will be 4^k nodes, each of size $\frac{n}{9^k}$. Since the amount of work done for each node is $c \cdot \sqrt{n}$, then the amount of time needed at level k is

$$W(k) \;=\; 4^k \cdot c \cdot \left(\frac{n}{9^k}\right)^{\frac{1}{2}} \;=\; cn^{\frac{1}{2}} \cdot \left(\frac{4}{3}\right)^k$$

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

$$\begin{aligned}
T(n) &= \sum_{k=0}^{\log_9 n} W(k) \\
&= \sum_{k=0}^{\log_9 n} cn^{\frac{1}{2}} \cdot \left(\frac{4}{3}\right)^k \\
&= cn^{\frac{1}{2}} \cdot \left(\sum_{k=0}^{\log_9 n} \left(\frac{4}{3}\right)^k\right) \\
&= cn^{\frac{1}{2}} \cdot \left(\frac{\left(\frac{4}{3}\right)^{\log_9 n+1} - 1}{\frac{4}{3} - 1}\right) \\
&= 3 \cdot \frac{4}{3} cn^{\frac{1}{2}} \left(\frac{4}{3}\right)^{\log_9 n} - 3 \cdot cn^{\frac{1}{2}} \\
&= 4 \cdot cn^{\frac{1}{2}} \cdot n^{\log_9\left(\frac{4}{3}\right)} - 3 \cdot cn^{\frac{1}{2}} \\
&= 4 \cdot cn^{\frac{1}{2}} \cdot n^{\log_9 4 - \frac{1}{2}} - 3 \cdot cn^{\frac{1}{2}} \\
&= 4 \cdot cn^{\log_9 4} - 3 \cdot cn^{\frac{1}{2}}
\end{aligned}$$

Since $\log_9 4 > \frac{1}{2}$ (we know this since $9^{(\frac{1}{2})} = 3 < 4$), $T(n) \in \Theta(n^{\log_9 4})$, which is approximately $\Theta(n^{0.63})$.

3. (a) Design and write an algorithm that, given a simple undirected graph $G = (V, E)$ as input, will determine how many edges would need to be added in order to connect the graph. If the graph is already connected, then your algorithm should answer 0.

For a solution, we need to realize that the minimum number of edges needed to link together components is one less than the number of components; essentially, you would need to build a tree where the components are the nodes.

```
initialize all vertices as unvisited
count ← 0
for all v ∈ V
        if v is unvisited
            BFS(G, v)
            count ← count + 1
        return count − 1
```

```
BFS(G, s)
    let Q be an empty queue
    enqueue s on Q
    mark s as visited
    while Q is not empty
        u ← dequeue(Q)
        for all v such that (u, v) ∈ E
            if v is not visited
                enqueue v on Q
                mark v as visited
```

(b) Analyse your algorithm's running time.

The vertices are initially marked as unvisited in cn time. The for loop in the first part of the algorithm loops through all vertices. However, it will only call $BFS()$ if it has found an unvisited vertex, so $BFS()$ will be called exactly once per component. Over all calls to $BFS()$, each vertex will be enqueued, marked as visited, and dequeued exactly once, so the total number of iterations of the while loop is n. For each vertex u, the number of iterations of the inner for loop is $deg(u)$, so the total number of iterations of this loop (over the entire execution of the algorithm) is

$$\sum_{u \in V} deg(u) = 2m \ .$$

The rest of the algorithm contributes at most a constant factor, so the algorithm as a whole runs in time $\Theta(m + n)$.