CS3933 Winter 2007 – Solo Assignment 2
Due Wednesday February 14, 2007, at 5pm.
Sample Solution

1. (7 marks) Design and write a $\Theta(n \log n)$ divide and conquer algorithm that will, given a sequence of n numbers, find the number of significant inversions in the sequence.

Algorithm Sig_Invs(list L, number n):
returns $(S, count)$ where S is the sorted list and $count$ is the number of significant inversions

```
if n = 1
  S ← L
  count = 0
else
  divide L into two halves:
        A containing the first ⌈n/2⌉ elements
        B containing the remaining ⌊n/2⌋ elements
  (S_A, c_A) ← Sig_Invs(A, ⌈n/2⌉)
  (S_B, c_B) ← Sig_Invs(B, ⌊n/2⌋)
  c ← 0
  S ← empty list
  c_sig ← ⌈n/2⌉
  head_A ← front of S_A
  head_B ← front of S_B
  sig ← front of S_A
  while there are still elements remaining in S_A or in S_B
        while c_sig ≠ 0 and sig ≤ 2 · head_B
              advance sig to next element
              c_sig ← c_sig − 1
        if head_B < head_A
              append head_B to S
              advance head_B
              c ← c + c_sig
        else (head_A is larger)
              append head_A to S
              advance head_A
return (S, c_A + c_B + c)
```

Despite the nested while loops, this is still a linear time combine step, since we cannot advance *sig* more than the length of S_A over all iterations.

Note to marker: since an algorithm for counting the number of inversions (not necessarily significant ones) was presented in class and is in the text, a significant amount of similarity in solutions is expected.

2. (7 marks) You are analysing a divide & conquer algorithm, and have determined that it has the following characteristics:

   - The algorithm processes the data in constant time to determine 5 parts of the data for the recursion step, where each part has size $\frac{n}{4}$
   - The algorithm recurses 5 times, each time on data of size $\frac{n}{4}$ (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 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).

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 5 \cdot T(\frac{n}{4}) + c \cdot \sqrt{n} & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence will have deepest level $\log_4 n$, since it divides the size by 4 until it is reduced to 1. At level k, where level 0 is the root, there will be 5^i nodes, each of size $\frac{n}{4^k}$. Since the amount of work done for each node is $c \cdot \sqrt{n}$, then the amount of time needed at level k is

$$W(k) \;=\; 5^k \cdot c \cdot \left(\frac{n}{4^k}\right)^{\frac{1}{2}} \;=\; cn^{\frac{1}{2}} \cdot \left(\frac{5}{2}\right)^k$$

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

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_4 n} W(k) \\ &= \sum_{k=0}^{\log_4 n} cn^{\frac{1}{2}} \cdot \left(\frac{5}{2}\right)^k \\ &= cn^{\frac{1}{2}} \cdot \left(\sum_{k=0}^{\log_4 n} \left(\frac{5}{2}\right)^k\right) \end{aligned}$$

$$
\begin{aligned}
&= cn^{\frac{1}{2}} \cdot \left(\frac{\left(\frac{5}{2}\right)^{\log_4 n+1} - 1}{\frac{5}{2} - 1} \right) \\
&= \frac{2}{3} \cdot \frac{5}{2} cn^{\frac{1}{2}} \left(\frac{5}{2}\right)^{\log_4 n} - \frac{2}{3} cn^{\frac{1}{2}} \\
&= \frac{5}{3} cn^{\frac{1}{2}} \cdot n^{\log_4\left(\frac{5}{2}\right)} - \frac{2}{3} cn^{\frac{1}{2}} \\
&= \frac{5}{3} cn^{\frac{1}{2}} \cdot n^{\log_4 5 - \frac{1}{2}} - \frac{2}{3} cn^{\frac{1}{2}} \\
&= \frac{5}{3} cn^{\log_4 5} - \frac{2}{3} cn^{\frac{1}{2}}
\end{aligned}
$$

Since $\log_4 5 > \frac{1}{2}$, $T(n) \in \Theta(n^{\log_4 5})$, which is approximately $\Theta(n^{1.16})$.

3. (5 marks) Solve the following recurrence relation by analysing the recursion tree. Note that you will need to calculate upper and lower bounds, since the division is asymmetric and produces a non-complete recursion tree.

$$
T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(\frac{2n}{3}) + T(\frac{n}{3}) + c \cdot n & \text{otherwise} \end{cases}
$$

Since the relation for $T(n)$ recurses on two parts of sizes that sum to n, each complete level contains n data items. Furthermore, since the nonrecursive part of the running time is linear, and $c \cdot \frac{n}{3} + c \cdot \frac{2n}{3} = c \cdot n$, the time required for each complete level is $c \cdot n$; the time required for incomplete levels will be less than $c \cdot n$.

Based on the $T(\frac{n}{3})$ part of the recurrence, the first leaf will appear at level $\log_3 n$, since $\frac{n}{3^{\log_3 n}} = 1$. So, all levels from the root (level 0) to level $\log_3 n$ will be complete. Since there are $\log_3 n + 1$ complete levels, each requiring $c \cdot n$ time to compute, $T(n) \geq c \cdot n \cdot (\log_3 n + 1)$, so $T(n) \in \Omega(n \log n)$.

Based on the $T(\frac{2n}{3})$ part of the recurrence, the last leaf will appear at level $\log_{\frac{3}{2}} n$, since

$$
\frac{n}{\left(\frac{3}{2}\right)^{\log_{\frac{3}{2}} n}} = 1
$$

So, there are $\log_{\frac{3}{2}} n + 1$ levels in the recursion tree, each requiring at most $c \cdot n$ time to compute. Thus $T(n) \leq c \cdot n \cdot (\log_{\frac{3}{2}} n + 1)$, so $T(n) \in O(n \log n)$. Therefore $T(n) \in \Theta(n \log n)$.

4. (a) (8 marks) Design and write an algorithm that, given a simple undirected graph $G = (V, E)$ as input, will determine if it can be turned into a tree by removing exactly one edge. Output of your algorithm should be either the edge whose removal will turn the remaining graph into a tree, or "no" if there is no such edge.

Analyse your algorithm's running time.

```
Algorithm MoreThanTree(graph G = (V, E))
    flag = 0
    if |E| = |V|
      flag = 1
      mark all vertices as unvisited
      mark all edges as unused
      let s be any vertex
      e = DFS_Edge(G, s)
      for all u ∈ V
          if u is unvisited
            flag = 0
    if flag = 1
      output e
    else
      output "no"

DFS_Edge(graph G = (V, E), vertex s)
    e = null
    for all u such that (s, u) ∈ E
      if u is unvisited
          mark u as visited
          mark (s, u) as used
          e' = DFS_Edge(G, u)
          if e = null
            e = e'
      else
          if (s, u) is unused
            e = (s, u)
    return e
```

Analysis:

The depth-first search performed by DFS_Edge will, in the worst case, visit all vertices once and consider all edges twice (once per endpoint). However, the DFS is not performed unless $m = n$. There are additionally three non-nested loops that iterate n times each (one to mark vertices, one to mark the $m = n$ edges, and one explicit loop to check for connectivity). Thus the algorithm runs in $\Theta(n)$ time.

(b) (8 marks) Design and write an algorithm that, given a simple undirected graph $G = (V, E)$ as input, will determine if it can be turned into a tree by adding exactly one edge. Output of your algorithm should be either the edge whose addition will turn the remaining graph into a tree, or "no" if there is no such edge.

Analyse your algorithm's running time.

```
Algorithm AlmostTree(graph G = (V, E))
    mark all vertices w ∈ V as unvisited
    flag = 0
    if |E| = |V| − 2
      Let u be any vertex
      DFS(G, u)
      for all w ∈ V
          if w is unvisited
            v ← w
            flag ← 1
      if flag = 1
          DFS(G, v)
          for all w ∈ V
            if w is unvisited
              flag ← 0
    if flag = 0
      output "no"
    else
      output (u, v)

DFS(graph G = (V, E), vertex s)
    for all u such that (s, u) ∈ E
      if u is unvisited
          mark u as visited
          DFS(G, u)
```

Analysis:
In the worst case, the algorithm will visit all vertices and consider all edges twice. However, the traversals only occur if $m = n - 2$. DFS is called at most twice, and there are three non-nested loops that either mark or check all vertices, which add at most a constant factor to the algorithm. Thus the algorithm runs in time $\in \Theta(m + n) = \Theta(n)$.