CS 466/666
Assignment 3
(Due: Noon Friday November 5, 2010)
Justify all answers

1) [13 marks] Consider algorithm for the union-find problem discussed in class (union by size, find with path compression). We showed that the amortized time of makeset/union/find operations is $O(\log^* n)$.

   a) Prove that a sequence of m union/find operations takes total time O(m) if all the unions come before any of the finds.
   b) Suppose the m operations consist of a constant number of runs of unions followed by runs of finds. Prove the total runtime is still O(m).
   c) Assume that we replace path compression with *partial path compression*: After find(x) partially compress the path $x \rightarrow$ root by setting $p(y) \leftarrow p(p(y))$ for each node y on the path. So each node becomes a child of its previous *grandparent*, not a child of the root. (Note that a merit of this approach is that it can be done in a single scan up a path, though that is not relevant to our question.) Prove that the amortized running time of makeset/union/find remains $O(\log^* n)$. (A precise statement of any modifications or why various conditions still hold in the proof given in class for the full path compression technique is sufficient.)

2) [18 marks] One application of the union-find problem is to maintain connected components of an undirected graph G as new edges are added to G. We should support two operations:

   - insert(u, v): inserts the edge (u, v) into G.
   - query(u, v): decides whether u and v are connected.

   a) Show how to implement the operations query and insert using a small number of union and find operations.

   Supporting edge deletion is tougher for general graphs. Here we consider a simple variant in which G is *initially a simple path of n vertices* and we only allow the following two operations:

   - delete(u, v): deletes the edge (u, v) from G.
   - query(u, v): decides whether u and v are still connected.

   b) Describe a list-based method that supports query in O(1) and delete in O(log n) amortized time, assuming that we have $\Theta(n)$ delete operations. Hint: Use an argument similar to the one for the weighted union heuristic.
   c) Describe a data structure that supports both query and delete operations in O(lg lg n) worst-case time (after a linear time set up). Hint: Maintain the set of deleted edges in a suitable data structure.

3) [19 marks] Given an array A of n numbers, we call a number x the *frequent element* of A if it occurs at least 0.7n times in A, i.e., for at least 70% of values for $1 \leq i \leq n$ we have A[i] = x. Note that not all arrays have frequent elements and each array can have at most one frequent element. The *mode* of A is the value that occurs most frequently in A.

Also recall the terminology of two approaches to using randomness in developing algorithms, both using random bits rather than relying on any distribution of the data. A *Monte Carlo* method is one that runs in a given time, but has some probability of failure. A *Las Vegas* algorithm always returns the correct answer, but has some probability of taking "longer than expected".

By a "very efficient" algorithm, we mean one that will run very quickly in practice with a small constant in the "O" term of its runtime.

a) Give an algorithm that finds the mode of A in time O(n log n).
b) Given a number x, describe a very efficient O(n) time algorithm that checks whether x is the frequent element of A.
c) Give an O(n) time deterministic algorithm that finds the frequent element of A or reports that none exists.
d) Give a very efficient Las Vegas algorithm that finds the frequent element of A (or reports there is none). State the expected number of comparisons your method uses, up to o(n) terms.
e) Give a very efficient Monte Carlo algorithm that finds the frequent element of A (or reports there is none). State the probability of error. The probability of error should be at most 0.25. Hint: Start your algorithm by taking 3 random elements.