Solutions to Problem Set No. 4

1. Algorithm: Perform a depth-first search of G and direct each edge in the direction the search advances along it. (Tree edges are directed from parent to child; non-tree edges from descendant to ancestor.) Let $\overrightarrow{G}$ be the resulting directed graph. Suppose G is connected. Then the search reaches all the vertices from a single start vertex. Suppose further that G contains no bridges. Then by page 8 of Class Notes No. 3, $L(v) < pre(v)$ for every vertex other than the root, where $pre(v)$ is the preorder number assigned by the depth-first search to v and $L(v) = \min\{pre(w)|$ there is a path consisting of zero or more tree edges followed by at most one non-tree edge from v to $w\}$. Identify vertices by their preorder numbers. Then, in $\overrightarrow{G}$, every vertex is reachable from vertex 1 (the tree root) by a path of tree edges, and for every vertex $v \neq 1$, $L(v)$ is reachable from v. The latter implies by induction that 1 is reachable from every vertex v, since $L(v) < v$ for $v \neq 1$. Thus $\overrightarrow{G}$ is strongly corrected.

   Conversely, if G is disconnected, there is no way to get between any two connected components no matter how the edges are directed. If G is connected but has a bridge $\{v, w\}$, there is no way to get from w to v if the bridge is directed from v to w, and no way to get from v to w if the bridge is directed from w to v.

   We conclude that the edges can be directed to make G strongly connected if and only if G is connected and has no bridges. Furthermore, if such directing is possible, a depth-first search will do it in linear time.

2. (a) Algorithm:

   Step 1: Sort the integers.

   Step 2: For each pair of integers i, j with $i < j$, construct the triple $(i, j, j - i)$. This produces $\binom{n}{2}$ triples.

   Step 3: Group the triples based on their third coordinate.

   Step 4: Perform the following steps for each group of triples: Let k be the common third coordinate of the triples. Construct a graph G_k whose edges are (i, j) such that (i, j, k) is a triple. Graph G_k is acyclic. Find a longest path in G_k via topological search.

   The longest path in all the G_k's corresponds to the longest arithmetic sequence; "k" is the difference between successive members of the sequence.

   Proving the correctness of this algorithm is straightforward: each path in a G_k corresponds to an arithmetic sequence, and each arithmetic sequence corresponds to a path in a G_k. We will describe an $O(n^2)$-time implementation; the hardest step is Step 3, which we deal with last.

Step 1 requires $O(n \log n)$ time with any fast sorting algorithm. For the purpose of Step 3, we index each integer from 1 to n by its position in the sorted sequence. Step 2 requires $O(n^2)$ time. Once Step 3 is completed, we perform Step 4 as follows. We construct an empty graph whose vertices are $1, 2, \ldots, n$. For each successive group of triples, for each triple (i, j, k) in the group, we construct an edge from $\mathit{index}(i)$ to $\mathit{index}(j)$ by adding $\mathit{index}(j)$ to the adjacency list for $\mathit{index}(i)$. We also record that $\mathit{index}(i)$ has at least one outgoing edge, and we increase the in-degree of $\mathit{index}(j)$ by one. Once the graph is constructed, we build a list of vertices with at least one outgoing edge but with in-degree zero. We maintain a count for each vertex incident to an edge, of the length of the longest path into it so far found. Initially this count is zero. We successively delete and and process vertices of in-degree zero but initially positive out-degree. To process a vertex v, for each edge (v, w) we replace $\mathit{count}(w)$ by $\max\{\mathit{count}(w),\ \mathit{count}(v) + 1\}$, delete (v, w), decrease the in-degree of w, and add w to the list of in-degree zero vertices if its in-degree has decreased to zero. When we are done processing v, we record $\mathit{count}(v)$ as a possible maximum path length.

This method will compute the maximum path length in G_k, by processing the vertices in topological order, in time proportional to the number of edges in G_k: no vertex not incident to an edge in G_k is examined. Thus the total time for Step 4 is $O(n^2)$.

I know of three ways to perform Step 3 in $O(n^2)$ time, none completely satisfactory. We can use a huge array with one slot for each possible difference $j - i$, and insert each pair (i, j) into slot $j - i$, recording non-empty slots. This method takes $O(n^2)$ time for Step 3 but uses an amount of space equal to the difference between the largest and smallest integers. Alternatively, we can use a hash table to store the pairs (i, j) based upon $j - i$. This uses $O(n^2)$ space and takes $O(n^2)$ average time, averaging over the choice of hash function. Finally, we can sort the pairs (i, j) based upon $j - i$. This is equivalent to the problem of sorting "$X + Y$", which can be done in $O(n^2)$ comparisons, but the algorithm I know about is non-constructive and the overhead to determine which comparisons to make may be greater than $O(n^2)$. See M. L. Fredman, "How good is the information theory bound in sorting?," *Theoretical Computer Science* **1** (1976), 355–361.

3. Algorithm:

Step 1: Compute the strongly connected components of G and shrink each to a single vertex. The resulting graph G is acyclic.

Step 2: Find a topological numbering of the vertices of G', and identify vertices by their numbers. Suppose G' has k vertices. Graph G is semiconnected if and only if G' contains $(i, i+1)$ for $i = 1, 2, \ldots, k-1$. That is, G is a single path plus extra edges.

This algorithm clearly runs in linear time. We prove it correct as follows. We claim that G is semiconnected if and only if G' is. This is true because there is a path from x to y in G if and only if there is a path from x' to y' in G', where x' (y', respectively) is the vertex of G' into which x (y, respectively) is contracted. If G' is a path plus extra edges, then G' is certainly semiconnected. Conversely, if there is a vertex i such that $(i,\ i+1)$ is not an edge, there is no path from i to $i+1$ and no path from $i+1$ to i. Hence G' is not semiconnected.

4. (a) To insert a new pair (x, y), we proceed as follows. If the tree is empty, put (x, y) a new root. Otherwise, let (a, b) be the pair in the root. If $x < a$, swap (x, y) and (a, b). Now the pair (x, y) to be inserted has $x \geq a$. If $y < b$, insert (x, y) in the left subtree by proceeding recursively. Otherwise, insert (x, y) in the right subtree by proceeding recursively. This method of insertion preserves both the heap order on the 1^{st} coordinate and symmetric order on the 2^{nd} coordinate.

   To delete a pair (x, y), we bubble-down an empty node. Let e be the node containing (x, y). If e has no children, just delete e. If e has a single child, delete e and replace it by its single subtree. If e has two children, move into e the pair among the two in e's two children that has smaller 1^{st} coordinate, and proceed recursively to fill or delete the newly vacant child. This deletion method preserves both orders. Both insertion and deletion proceed along a single tree path and take time proportional to the tree depth.

   (b) If the tree is empty, stop. Otherwise, let (x, y) be the pair in the root. If $x \leq a$ and $b \leq y \leq c$, return (x, y) as a pair in the answer set and proceed recursively in both subtrees of the root. If $x \leq a$ and $y < b$, proceed recursively in the right subtree. If $x \leq a$ and $y > c$, proceed recursively in the left subtree. (If $x > a$, stop: since all pairs below (x, y) have 1^{st} coordinate at least x, no pair below (x, y) can satisfy the query.)

   In case $x \leq a$ and $y < b$ ($y > c$, respectively), no pair in the right subtree (left subtree, respectively) can satisfy the query. It follows that the algorithm returns all the correct pairs, and no incorrect ones. We verify the time bound as follows. If we walk down through the tree concurrently level-by-level following the search for query answers, at most two nodes per level (the leftmost and the rightmost) can be visited without having their contents satisfy the query. Suppose to the contrary that there are pairs (x_1, y_1) and (x_2, y_2) on the same level with $y_1 \leq y_2 < b$. Let (x, y) be the pair that is in the nearest common ancestor of the nodes containing the pairs (x_1, y_1) and (x_2, y_2). Then $y \leq y_2 < b$, which means that the search will never reach (x_1, y_1). The symmetric argument applies if $c < y_1 \leq y_2$. It follows that the total number of nodes visited during the search is at most the number of pairs that satisfy the query plus twice the tree depth plus one.

(c) A treap containing the pairs $(1, 1), (2, 2), (3, 3), \ldots, (n, n)$ is just a path of depth $n - 1$.

(d) See R. E. Tarjan, "Updating a balanced search tree in $O(1)$ rotations," *Infor. Proc. Lett.* **16** (1983), 253–257 and E. M. McCreight, "Priority search trees," *SIAM J. Comput.* **14** (1985), 257–276.