COS 226	Algorithms and Data Structures	Fall 2010

Final Solutions

1. **Analysis of algorithms.**

U Find a maximum spanning tree in a connected edge-weighted graph.
This problem and the MST linear-time reduce to one another (by negating the weights). The existence of a deterministic linear-time algorithm for the MST is an open problem.

P Find all vertices reachable from a given *set* of source vertices in a digraph.
We used this subroutine in the NFA simulation algorithm for regular expressions.

I Find a Hamilton path in a digraph (if one exists).
This problem is NP-complete, so unless $P = NP$, it cannot be solved in polynomial time, let alone in linear time.

P Find a Hamilton path in a DAG (if one exists).
This can be found by computing the topological order and checking that there is an edge between each consecutive pair of vertices in the order.

P Find the strong components of a digraph.
Kosaraju's or Tarjan's algorithm does this.

P Insert N `Comparable` keys into a binary heap.
The sink-based heap construction method used in heapsort achieves this.

I Sort an array of N `Comparable` keys.
This would violate the $N \lg N$ sorting lower bound because `Comparable` *keys can be accessed only through the* `compareTo()` *method.*

I Insert N `Comparable` keys into a binary search tree.
This would violate the $N \lg N$ sorting lower bound because `Comparable` *keys can be accessed only through the* `compareTo()` *method and an inorder traversal of the BST would yield the keys in sorted order.*

P Compute the inverse Burrows-Wheeler transform.
You did this on Assignment 8.

P Insert N strings into an R-way trie.
Inserting each string takes time proportional to its length.

P *The number of nodes in a TST is bounded by the the total number of characters.*

P Perform a nearest-neighbor query in a 2-d tree.
You algorithm from Assignment 7 has this property (even though it's running time is typically much better in practice).

2. **Equivalence relations.**

   ✓ `v.equals(w)` for objects in a Java class that correctly implements the `equals()` method

   ___ `v.compareTo(w) < 0` for objects in a Java class that correctly implements the `Comparable` interface

   ✓ `connected(v, w)` in `CC` for connectivity in an undirected graph

   ___ `reachable(v, w)` in `TransitiveClosure` for reachability in a digraph

   ✓ `stronglyConnected(v, w)` in `KosarajuSCC` for strong connectivity in a digraph

3. **Depth-first search.**

   (a)
   ```
   preorder:   A B G C D F E H I
   postorder:  G B E I H F D C A
   ```

   (b) I and II only
   The function-call stack always contains a sequence of vertices on a directed path from s to the current vertex (with s at the bottom and the current vertex at the top).

4. **Minimum spanning tree.**

   (a) 1 2 3 4 6 7 9 15

   (b) 6 1 3 2 4 7 9 15

5. **Shortest paths.**

   (a)
   ```
   v   edgeTo[]     distTo[]
   -------------------------
   0    -             0.0
   1   2->1 33.0     34.0
   2   0->2  1.0      1.0
   3   2->3 11.0     12.0
   4   5->4 33.0     53.0
   5   3->5  8.0     20.0
   6   1->6  5.0     39.0
   7   5->7 46.0     66.0
   8    -            infinity
   ```

   (b)
   ```
   v   edgeTo[]     distTo[]
   -------------------------
   0     -            0.0
   1    2->1 33.0    34.0
   2    0->2  1.0     1.0
   3    2->3 11.0    12.0
   4   *6->4 13.0   *52.0
   5    3->5  8.0    20.0
   6    1->6  5.0    39.0
   7   *6->7 46.0   *45.0
   8   *6->8 10.0   *49.0
   ```

   (c) $2 \to 1$, $0 \to 2$, $2 \to 3$, $7 \to 4$, $3 \to 5$, $1 \to 6$, $6 \to 7$, $6 \to 8$

6. **Polar sort.**

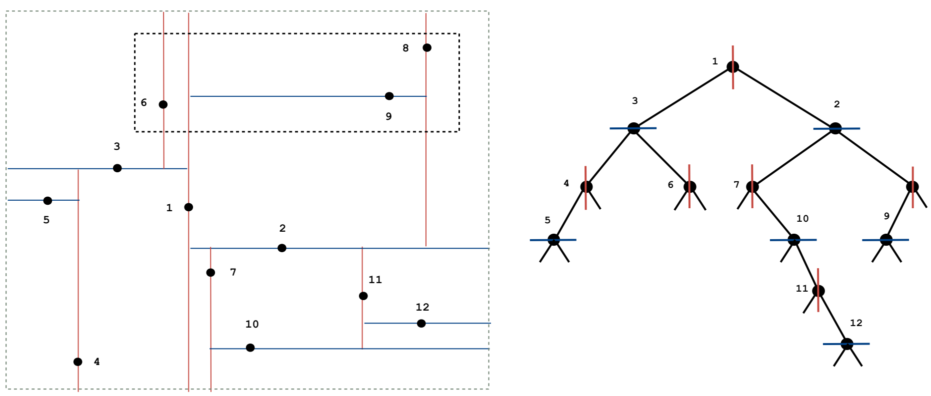
   (a) Not a transitive relation, as required by the `compareTo()` contract. To see this, let p be $(0,0)$; let q_1 be $(1,1)$; let q_2 be $(-1,1)$; let q_3 be $(0,-1)$. Then, with respect to p, q_2 is counterclockwise of q_1; q_3 is counterclockwise of q_2; and q_1 is counterclockwise of q_3.

   (b) When comparing two points q_1 and q_2 by the polar angle they make with p, first compare the y-coordinates of q_1 and q_2 to that of p. If one has a bigger y-coordinate than p and the other has a smaller y-coordinate than p, then the one with the smaller y-coordinate makes a greater polar angle with p; otherwise if both have bigger or both have smaller y-coordinates, then the ccw-based code works. (A special case is needed to handle the case when q_1 and q_2 have the same y-coordinate as p.)

7. **Kd-trees.**

   (a) 1 2 3 6 8 9 (though the search may go one extra level, depending on implementation)

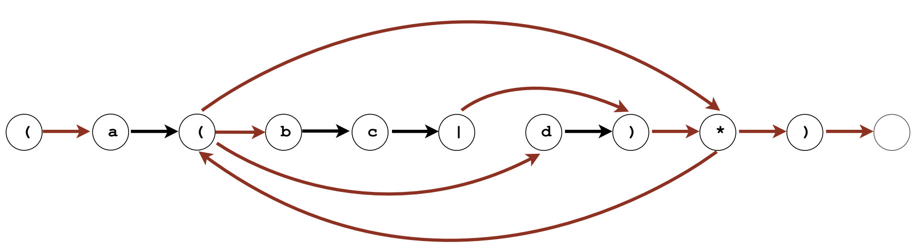
   (b)



8. **Substring search.**

	0	1	2	3	4	5	6
a	1	1	1	4	1	6	1
b	0	2	0	0	5	0	2
c	0	1	3	0	0	3	7

9. **Regular expressions.**



10. **Substring search and pattern matching.**

E	brute-force substring search	A. M
C/D	Knuth-Morris Pratt	B. N/M
E	Boyer-Moore (with only mismatch heuristic)	C. N
C/D	Monte Carlo version of Rabin-Karp	D. $M + N$
E	regular-expression pattern matching	E. MN
C	simulating a DFA	F. 2^M
E	simulating an NFA	G. 2^N

11. **Huffman codes.**

code 1: a Huffman (and optimal) prefix-free code

code 2: not a prefix-free code because 00 is a prefix of 001

code 3: a Huffman (and optimal) prefix-free code

code 4: a prefix-free code, but not optimal (or Huffman) because it produces a 91-bit code (instead of 90)

code 5: an optimal prefix-free code (produces a 90-bit code), but it's not a Huffman code because C and D both start with 0

12. **Cyclic rotation of a string.**

(a)
- Form the new string $t' = t + t$ by concatenating two copies of t.
- Do a substring search for the query string s within the text string t' using Knuth-Morris-Pratt. s is a cyclic rotation of t if and only if KMP finds a match.

For example search for `winterbreak` in `breakwinterbreakwinter`.

(b) The order-of-growth of the worst-case running time is N.

13. **Reductions.**

(a) This direction is easy. Solve an instance of MULTIPLICATION with x as both arguments. This computes $x \times x$, as desired.

(b) i. Compute $(x - y)$.
ii. Solve the following three instances of SQUARING: x^2, y^2, $(x - y)^2$.
iii. Compute $z = x^2 + y^2 - (x - y)^2 = 2xy$
iv. Compute $z/2 = xy$, noting that z is even.

Steps i, iii, and iv take linear time using the grade-school algorithms for addition, subtraction, and division by two. Step 2 calls the subroutine SQUARING a constant number of times on inputs of N bits (or fewer).

(c) I, II, and III