COS 226	Algorithms and Data Structures	Spring 2008

Final Solutions

1. **Graph search. (10 points)**

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

   (b) `A B D E C F G H I`

   (c) Shortest path (fewest number of edges) from s to t.

   (d) Topological sort, strongly connected components, Euler tour.

2. **Minimum spanning tree.**

   (a) `B-D D-G E-F C-H F-H C-G A-H`

   (b) `A-H C-H F-H E-F C-G D-G B-D`

3. **Minimum spanning tree.** For simplicity, we'll assume all edge weights are distinct.

   Find the unique path between v and w in the MST. This takes $O(V)$ time using BFS or DFS because there are only $V - 1$ edges in the MST subgraph. We claim that the MST in G is the same as the MST in G' if and only if every edge on the path has length less than c.

   - If any edge on the path has weight greater than c, we can decrease the weight of the MST by swapping the largest weight edge on the path with v-w. Hence weight of the MST for G' is strictly less than the weight of the MST for G.
   - If the weight of v-w is larger than any edge on the path between v and w, then the cycle property asserts that v-w is not in the MST for G' (because it is the largest weight edge on the cycle consisting of the path from v to w plus the edge v-w). Thus, the MST for G is also the MST for G'.

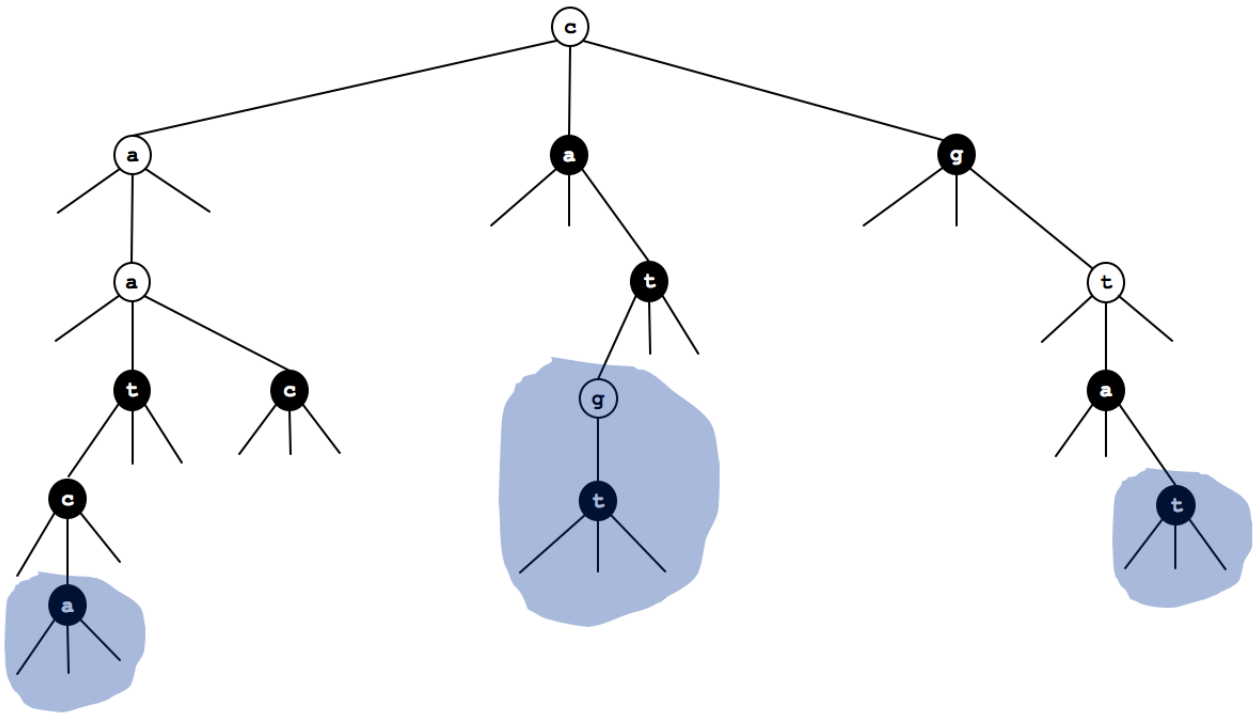
4. **Data compression.**

   $16 * 1 + 8 * 2 + 4 * 3 + 2 * 4 + 1 * 4 = 56$

5. **Ternary search tries.**

(a) `aac, aat, ac, ca, ct, g, ta`

(b)



6. **String searching.**

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

7. **Algorithm matching.**

A T9 texting in a cell phone	A. Trie
D 1D range search	B. Hashing
E 2D range search	C. 3-way radix quicksort
B Document similarity	D. Binary search tree
K Traveling salesperson problem	E. Kd tree
L Sudoku solver	F. Depth-first search
J Arbitrage detection	G. Breadth-first search
F Mark-sweep garbage collector	H. Dijkstra's algorithm
G Web crawler	I. Topological sort
H Google maps	J. Bellman-Ford
I PERT/CPM	K. Enumerate permutations
C Longest repeated substring	L. Enumerate base-R integers

8. **Regular expressions.**

Draw the NFA that results from the RE-to-NFA conversion algorithm described in lecture when applied to the regular expression `(a | (bc)*) d*`. Label the start state 0, the accept state 1, and remaining states in the order they are created by the RE-to-NFA algorithm.

9. **Convex hull.**

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

   (b)
   ```
   1. F -> G -> H
   2. F -> G -> H -> I
   3. F -> G -> E
   4. F -> G -> E -> D
   5. F -> G -> E -> D -> B
   6. F -> G -> E -> A
   7. F -> G -> E -> A -> C
   ```

10. **4-sum.**

    Consider the 4-SUM problem: *Given N integers, do any 4 of them sum up to exactly 0?*

    (a) N^4

    (b) For each i and j, put the integer key $a[i] + a[j]$ in a hash table. Assoicate with each integer key the list of all pairs of indices that sum to that value.

    For each k and l, check whether the key $-(a[k] + a[l])$ is in the hash table. If so, scan the list of all pairs of indices taht sum to that value. If such a pair exists whose indices i and j are disjoint from k and l, then we have four distinct array entries that sum to exactly 0.

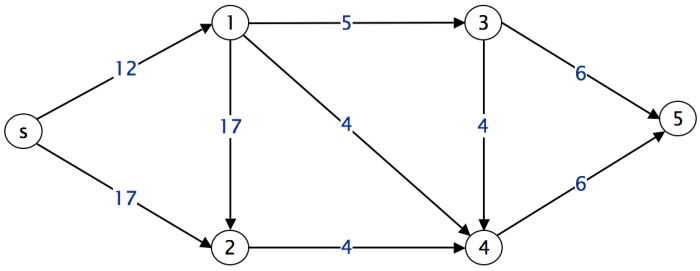
    The first phase takes $O(N^2)$ time and $O(N^2)$ space.

    The second phase can be inefficient because searching through the list can take too long (if too many of the entries at the beginning of the list have indices that are not disjoint from k and l). For example, if the array contains $20, 20, 20, \ldots, 20, 10, -40, 30, 0$, then the list of all pairs that sum to 30 will be $20 + 10, 20 + 10, 20 + 10, \ldots 20 + 10, 30 + 0$. Thus, in the second pass, the algorithm will waste alot of time trying to find a match for 10-40 because it is not disjoint from all of the 20+10 entries.

    To avoid this bottleneck, preprocess the original array so that at most 4 copies of each value remain. Now, when scanning the list for a sum that complements k and l, only scan the first 9 entries in the list for the sum $-(a[k] + a[l])$. Ther can be at most 4 pairs of indices in the list with k as one of the indices and at most 4 pairs with l as one of the indices. By the 9th entry, we will have found a pair of indices disjoint from k and l (or we will have exhausted the list). Thus, we only need to do a constant amount of work for each k and l.

11. **Reductions and shortest paths.**

    - **Solution 1.** Create a weighted digraph G' with the same set of vertices and edges. For each edge v-w assign its weight to be the weight of vertex w. The shortest vertex-weighted path from s to v in G is the same as the shortest edge-weighted path from s to v in G'. (The distance of the paths differ by the vertex weight of s.)



- **Solution 2.** Replace each vertex v with two vertices v' and v'' and include and edge v'-v'' with weigh equal to the vertex weight of v. Replace each directed edge v-w with the edge v''-w' and assign it a weight of 0. The shortest path from s' to v'' in G' corresponds to the shortest path from s to v in G.

12. **Suffix sorting.**

(a) String concatenation takes time and space proportional to the output. Thus, the loop takes quadratic time and uses quadratic space.

(b) Substring extraction takes only constant time and uses only constant space.

```
// form the N suffixes, appending '\0' to the end of each string
s = s + "\0";
String[] suffixes = new String[N];
for (int i = 0; i < N; i++) {
   suffixes[i] = s.substring(i, N+1);
}

// sort the N strings
RadixQuicksort3way.sort(suffixes);
```