COS 226 **Algorithms and Data Structures** **Spring 2020**

Final Exam Solutions

1. **Initialization.** Don't forget to do this.

2. **Memory usage.**

   (a) 262,144 (2^{18})
   The memory usage for $V = E = 80$ *is* $8192 = 2^{13}$*. Going down one row makes the memory go up by a factor of* 8*; going right one column makes the memory go up by a factor of* 4*. So, the memory usage for* $V = E = 160$ *is* $2^{13} \times 8 \times 4 = 2^{18}$.

   (b) $\Theta(V^3 E^2)$
   When you double V*, the memory goes up by a factor of 8, so the exponent for* V *is* $\log_2 8 = 3$*. When you double* E*, the memory goes up by a factor of 4, so the exponent for* E *is* $\log_2 4 = 2$.

3. **String operations.**

   (3.1) n^2
   String concatenation takes time proportional to the length of the resulting string, so the running time is $\Theta(1 + 2 + 3 + \ldots + n) = \Theta(n^2)$.

   (3.2) n
   Appending each character to the end of a `StringBuilder` *object takes* $\Theta(1)$ *amortized time. So, starting from an empty* `StringBuilder`*, appending* n *characters takes* $\Theta(n)$ *time in the worst case.*

   (3.3) 2^n
   Immediately after the `for` *loop, the length of* s *is* 2^n.

   (3.4) n
   String concatenation takes time proportional to the length of the resulting string, so the running time is $\Theta(1+2+4+8+\ldots+n) = \Theta(n)$. The substring extraction part also takes $\Theta(n)$ time because `substring()` is called only once.

   (3.5) $n \log n$
   As with mergesort, the recursive function divides the problem into two subproblems of size $n/2$ *and does* $\Theta(n)$ *work (for the string concatenation). So, the running time is* $\Theta(n \log n)$.

4. **String sorts.**

   (4.1) *LSD radix sort after 1 pass.*

   (4.2) *LSD radix sort after 2 passes.*

   (4.3) *MSD radix sort after the second call to key-indexed counting.*

   (4.4) *3-way radix quicksort after the first partitioning step.*

   (4.5) *MSD radix sort after the first call to key-indexed counting.*

5. **Graph search.**

(5.1) 0 1 2 8 4 3 6 5 7 9

(5.2) 2 8 1 5 7 6 3 4 9 0

(5.3) 0 1 4 8 9 2 3 5 6 7

6. **Properties of shortest paths algorithms.**

(6.1) Dijkstra, Topological, and Bellman–Ford

(6.2) Dijkstra, Topological

- *The existence of the edge $v \to w$ implies that v appears before w in any topological order. Thus, the topological sort algorithm must relax vertex v before vertex w.*
- *Since a shortest path from s to t visits vertex v before w (and all edge weights are positive), it follows that the length of the shortest path from s to v is strictly less than the length of the shortest path from s to w. Dijkstra relaxes vertices in order of increasing order of distance from s, so it must relax vertex v before w.*

(6.3) Dijkstra, Topological
The topological sort algorithm relaxes each vertex once, in topological order. Dijkstra's algorithm relaxes each vertex once, in increasing order of distance from s.

(6.4) Topological

(6.5) Topological, Bellman–Ford
Bellman–Ford takes $\Theta(E+V)$ time if the vertices happen to be numbered so that they are relaxed in increasing order of distance from s.

(6.6) Topological, Bellman–Ford
The topological sort algorithm works even if the edge weights are negative. Bellman–Ford works if there are no negative cycles, which will be the case here since there are no directed cycles.

7. **Minimum spanning trees.**

(7.1) 10 20 30 50 60 80 130

(7.2) 80 10 20 60 30 50 130

8. **Maximum flow.**

(8.1) 65 = 7 + 30 + 28

(8.2) 104 = 8 + 45 + 10 + 41

(8.3) $A \to G \to B \to C \to H \to I \to J$

(8.4) 68
The bottleneck capacity of the augmenting path found in 8.3 is 3. Augmenting flow along this path yields a maxflow.

(8.5) $S = \{A, B, C, D, F, G, H\}$
With respect to the maxflow f^* found in 8.4, find all vertices reachable from s via forward edges (that aren't full) or backward edges (that aren't empty).

9. **R-way trie.**

   K C J G A F

```
    public Integer get(String key) {
        Node x = root;
        for (int d = 0; d < key.length(); d++) {
            if (x == null) return null;
            char c = key.charAt(d);
            x = x.next[c];
        }
        if (x == null) return null;
        return x.val;
    }
```

10. **Knuth–Morris-Pratt.**

    ABABAC, CBCBCC

    One way to solve this problem is to build the DFA for all 5 strings and check which have the requisite pattern.

    Here's a more systematic approach. Consider the partial DFA below, where unknown characters are labeled with p, q, r, s, and t.

$$\begin{tikzcd}
0 \arrow[r, "p"] & 1 \arrow[r, "q"] & 2 \arrow[r, "r"] & 3 \arrow[r, "s"] & 4 \arrow[r, "t"] & 5 \arrow[l, bend right, "B"'] \arrow[r, "C"] & 6
\end{tikzcd}$$

Recall that being in state 4 implies that the last 4 characters of the text match the first 4 characters in the pattern. One way to get to state 4 is if the last 6 characters in the text are $pqrstB$ (corresponding to $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 4$). In this case, the last 4 characters in the text ($rstB$) must match the first 4 characters in the pattern ($pqrs$). That is, $p = r$, $q = s$, $r = t$, and $s = B$. Simplifying, we get $p = r = t$ and $q = s = B$.

The strings $ABABAC$ and $CBCBCC$ match the partial DFA. The string $BBBBBC$ almost matches the partial DFA, except the transition in state 5 for a B goes to state 5 (not 4).

11. **Regular expressions.**

    (11.1) $\epsilon, B, BBBB, BABBAABBAAAB$
    *If a string matching the RE contains an A, it must match the regular expression (BA*B)*, which implies it contains an even number of Bs (e.g., not BBBBAB).*

    (11.2) $\epsilon, B, BBBB, BBBBAB$
    Because there is an ϵ*-transition* $3 \to 4$ *and no* ϵ*-transition* $10 \to 4$*, strings accepted by the dummy NFA end up matching the regular expression B*(BA*B).*

    (11.3) BBAB

    (11.4) BABBB

    (11.5) Add edge $10 \to 4$; delete edge $3 \to 4$

12. **Data compression.**

    (12.1) 7 M M O O O B Z B

    (12.2) C A B C C C C B D B

    (12.3) C A C A B C A C A B

13. **Ternary search tries.**

    13.1 K Q
    The left and right links connecting the nodes A, H, J, ?, R, U form a mini-BST (corresponding to the second character of strings starting with T). So, the mystery character must be strictly between J and R.

    13.2 EAR, THE, TRIE, TUB, TUBA

14. **Problem identification.**

14.1 Possible

Run DFS from an arbitrary vertex. If all vertices get marked, the `edgeTo[]` *array provides a unique path between s and each vertex. These edges correspond to a spanning tree. BFS works similarly.*

14.2 Possible

You solved this problem on the WordNet *assignment. Such a vertex exists if and only if there is exactly one vertex of outdegree 0.*

14.3 Possible

Prim's algorithm and Kruskal's algorithm both work with negative edge weights. So, to find a maximum spanning tree in an edge-weighted graph, negate all of the weights and find a minimum spanning tree.

14.4 Possible

Sort the integers (using LSD or MSD radix sort). This takes $\Theta(n)$ time because the integers are fixed size (64 bits). Compute the squared distance between adjacent integers in the sorted array, maintaining a champion to keep track of the closest pair encountered.

14.5 Impossible

The total size of the input is n^2, so it takes $\Theta(n^2)$ time to read the input (or write the output).

14.6 Possible

Insert the n strings in a binary trie, marking each node that corresponds to a string. Check whether any non-leaf node is marked—that would correspond to a string that is a prefix of another string.

14.7 Possible

This is the classic substring search problem. Knuth–Morris–Pratt solves this problem in $\Theta(m+n)$ time.

14.8 Possible

This is a variant of the classic regular expression pattern matching problem. Search for the regular expression $.*(regexp).*$, where regexp is the regular expression.

14.9 Impossible

This would imply that every input of length 100 (or more) could be compressed down to 99 (or fewer bits) because you could repeatedly apply the algorithm to the compressed message. But there are infinitely many such inputs and only finitely many bitstrings of length 99 (or less). So, multiple inputs would have to map to the same compressed representation, which is not allowed.

15. **Tiger coloring.**

15.1 $\Theta(E+V)$

The inner `for` *loop is executed V times. The body of the inner loop is executed twice for each edge $e = v - w$ (once in v's adjacency and once in w's adjacency list).*

15.2 $\Theta(1)$

The function terminates as soon as it detects an inconsistency in the purported tiger coloring. For example if there are enemy edges 0-1, 1-2, and 2-0, then the v loop wouldn't make it past 2. (With self-loops or parallel edges, it could terminate even sooner.)

15.3 spanning tree, bipartite graph, no odd-length cycle

If there are only enemy edges, this is equivalent to asking whether an undirected graph is bipartite. *An undirected graph is bipartite if and only if it has no odd-length cycles. A spanning tree is an example of a bipartite graph—it has no cycles.*

15.4 The main idea is to pick an arbitrary starting vertex s, color it orange, and run a modified version of either BFS or DFS from vertex s. When visiting vertex v and considering an incident edge $e = v\text{-}w$, color w as forced by edge e (same color as v if a friendship edge and opposite color as v if an enemy edge). If w was already colored with an incompatible color, stop (and declare the graph as not tiger colorable).

Here's a Java implementation that (for brevity and variety) defers checking for incompatibilities until the end. Since G is connected, all vertices will get marked and colored.

```java
public class TigerColorable {
    private[] boolean marked;
    private[] boolean isOrange;
    private boolean isTigerColorable;

    public TigerColorable(TigerGraph G) {
        marked = new boolean[G.V()];
        isOrange = new boolean[G.V()];
        isOrange[0] = true;
        dfs(G, 0);
        isTigerColorable = isTigerColoring(G, isOrange);  // from 15.1
    }

    private void dfs(TigerGraph G, int v) {
        marked[v] = true;
        for (TigerEdge e : G.adj(v)) {
            int w = e.other(v);
            if (!marked[w]) {
                if (e.isFriendshipEdge()) isOrange[w] =  isOrange[v];
                else if (e.isEnemyEdge()) isOrange[w] = !isOrange[v];
                dfs(G, w);
            }
        }
    }

    public boolean isOrange(int v)    { return isOrange[v];      }
    public boolean isTigerColorable() { return isTigerColorable; }
}
```