COS 226	Algorithms and Data Structures	Spring 2022

Final Exam Solutions

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

2. **Analysis of algorithms.**

   (2.1) $\sim 48n$ bytes
   Each of the n `Node` objects uses 48 bytes:
   - 16 bytes object overhead
   - 16 bytes for the two `Node` references
   - 16 bytes for the two `double` instance variables

   (2.2) $\Theta(E + V)$
   This is the idiomatic double nested loop that iterates over each vertex and each edge exactly once.

3. **String operations.**

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

   (3.2) $\Theta(n^4)$
   String concatenation takes time proportional to the length of the resulting string. So, the running time is $\Theta(1 + 2 + 3 + \ldots + n^2) = \Theta(n^4)$.

   (3.3) $\Theta(n^2)$
   Appending each character to the end of a `StringBuilder` *takes* $\Theta(1)$ *amortized time. So, starting from an empty* `StringBuilder`*, appending* n^2 *characters takes* $\Theta(n^2)$ *time.*

   (3.4) $\Theta(n^2)$
   String concatenation takes time proportional to the length of the resulting string. Assuming n *is a power of 2, the running time is* $\Theta(1 + 2 + 4 + 8 + \ldots + n^2) = \Theta(n^2)$. *If* n *is not a power of 2, then the string is doubled until reaching the smallest power of 2 greater than* n^2*, but this doesn't affect the asymptotic running time. The substring extraction also take* $\Theta(n^2)$ *time.*

4. **String sorts.**

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

   (4.2) *3-way radix quicksort (after the first partitioning step)*

   (4.3) *LSD radix sort (after 2 passes)*

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

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

5. **Graph search.**

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

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

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

(5.4) ☒ ☒ ☐ ☐
Connected components and bipartiteness can be computed using either BFS or DFS. Finding a topological ordering is specific to DFS. Finding a shortest path (fewest edges) in a digraph is specific to BFS.

6. **Minimum spanning trees.**

(6.1) 0 10 20 30 50 70 110 120

(6.2) 20 30 10 50 70 0 120 110

(6.3) ☐ ☒ ☒ ☐

7. **Shortest paths.**

(7.1) 0 4 3 1 2 5
Dijkstra's algorithm relaxes the vertices in increasing order of distance from s.

(7.2) 0 4 3 5 2 1
The topological order happens to be unique (because of the path $0 \to 4 \to 3 \to 5 \to 2 \to 1$).

(7.3) 150, 140

(7.4) ☒ ☐ ☐ ☒

8. **Maxflow and mincut.**

(8.1) 52

(8.2) 57

(8.3) $A \to B \to C \to I \to D \to E \to J$, bottleneck capacity = 3

(8.4) $\{A, F, G, H\}$ or $\{A, B, C, D, F, G, H, I\}$

9. **Dynamic programming.**

(9.1) F B C H G I J
The solution is unique.

10. **Ternary search tries.**

    (10.1) A, AN, IN, PET, POT

    (10.2) JADWIN, MATHEY, NASSAU, ORANGE

11. **Data compression.**

    (11.1) 2/3
    Each of the 16 0s in the input is replaced with one 8-bit count (16) and each of the 8 1s in the input is replaced with one 8-bit count (8). So, run-length coding using 16 (8 + 8) bits to represent 24 (16 + 8) input bits.

    (11.2) A B B A B A

    (11.3) ☐ ☐ ☒ ☒

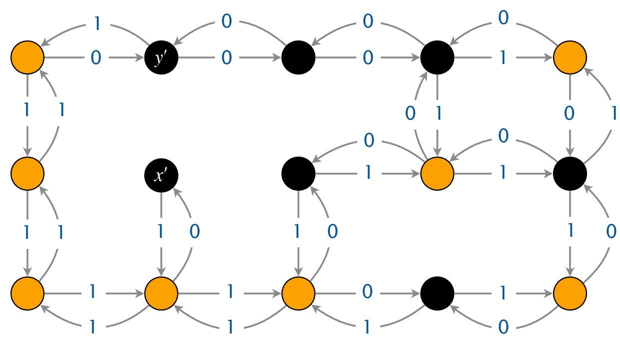
    (11.4) A B B A A B B A A B B

    (11.5) A B C

12. **Orange and black directed cycle.**

There are three key ideas to modeling the orange-and-black cycle problem as a single-source shortest paths problem in a directed graph:

- Remove the edge x–y. Now, any *path* between x and y corresponds to a cycle containing edge x–y.
- Replace each undirected edge with two anti-parallel directed edges. Now, any simple *directed path* from x to y corresponds to a cycle containing edge x–y.
- Assign a weight of a weight of 1 to all edges $v \to w$ with vertex w colored orange. Now, the *weight* of a directed path is equal to its number of orange vertices (ignoring the color of the first vertex in the path).



(12.1) The digraph G' has V vertices, one vertex v' for each vertex v in G. Identify vertex x' as the source vertex.

(12.2) The digraph G' has $2E-2$ edges. For each edge v–w in G other than x–y, include two antiparallel edges $v' \to w'$ and $w' \to v'$ in G'. If v is colored orange, the weight of the edge $w' \to v'$ is 1; otherwise its weight is 0. if w is colored orange, the weight of edge $v' \to w'$ is 1; otherwise its weight is 0.

(12.3) The shortest path from x' to y' in G' (plus the deleted edge) corresponds to the desired cycle in G.

Alternative solutions. There are a few variants that also work:

- Assign a weight of 1 to any edge $v' \to w'$ with v orange (instead of with w orange).
- Assign a weight of 1 to any edge $v' \to w'$ with either v orange *or* w orange. This effectively doubles the length of all paths, but doesn't change the shortest paths.
- Use a weight other than 1. Any positive constant works.
- Identify vertex y' as the source vertex (instead of x'). The edges are symmetric.
- Deletes edges of the form $v' \to x'$ or $y' \to w'$. These are optional (since the shortest path from x' to y' won't use them).

We note that the orange-and-black cycle problem can be solved in $\Theta(E+V)$ time in the worst case using a variant of breadth-first search. But the goal here is to model it as a classical shortest path problem.

13. **Equivalent BSTs.**

The algorithm has three key steps:

- **Uniquely identify a BST.** For each BST, compute its *level-order traversal* to uniquely identify it. You can think of each level-order traversal as a string of length m, where each "character" in the string is a 64-bit integer. Here are the level-order traversals of the four example BSTs:
  - 50 20 80 10 30
  - 50 20 99 10 30
  - 50 20 80 10 30
  - 50 10 80 20 30
- **Sort.** Sort the level-order traversals to bring equivalent BSTs together.
  - 50 10 80 20 30
  - 50 20 80 10 30
  - 50 20 80 10 30
  - 50 20 99 10 30

  The main challenge here is that the alphabet is of size $R = 2^{64}$. So, we can't directly apply LSD or MSD radix sort since that would involve creating an array of length 2^{64}. So, instead, treat each 64-bit integer as a sequence of eight 8-bit integers. Now, the strings are of length $8m$ (instead of m) but the alphabet is of size $R = 2^8$ (instead of 2^{64}). Now, we can sort these "strings" efficiently using either LSD or MSD radix sort.
- **Find equivalent BSTs.** The sorting brings equivalent BSTs together. To check for equivalent BSTs, it suffices to compare adjacent entries in the sorted order.

Each of the three steps takes $\Theta(mn)$ time in the worst case.

Alternative solutions.

- *Other tree traversals.* A BST can also be uniquely defined by its *preorder* or *postorder* traversal. (Inorder traversal does not work because all BSTs on the same set of keys have the same inorder traversals.)
- *Different alphabets.* We can treat each 64-bit integer as either a sequence of eight 8-bit integers or as a sequence of 64 individual bits. Using $R = 2$ will be slower than using $R = 2^8$, but only by a constant factor.
- *Multiway trie.* This wastes some space but still meets the performance requirements (assuming you break up each 64-bit integer into 8 bytes (or 64 bits) so that $R = 2^8$ (or $R = 2$) instead of $R = 2^{64}$).

Partial-credit solutions.

- *String concatenation.* Use a `StringBuilder` to concatenate the *decimal* integers in a BST traversal. Two non-equivalent BSTs may be incorrectly identified as equivalent, e.g., the BSTs with traversals of [12, 345] and [1, 2345] would both result in `"12345"`. It's important to treat each integer as a sequence of a fixed number of bits or bytes. A variant of this approach that does work is to separate the decimal integers with spaces (or some other delimiter) when concatenating. Now, the alphabet is effectively of size $R = 11$.
- *Compare-base sorting algorithms.* Mergesort (or heapsort) makes $\Theta(n \log n)$ string compares in the worst case. Each compare takes $\Theta(m)$ time in the worst case, This leads to a worst-case running time $\Theta(mn \log n)$.
- *3-way radix quicksort.* In the worst case, 3-way radix quicksort takes $O(mnR)$ time. If the alphabet size R were a small constant (e.g., 2 or 8 or 11), then this meets the performance requirement. However, when R is astronomical (e.g., $R = 2^{64}$ and bigger than n), you can't treat it as a constant. For example, even when $m = 1$, all of the partitioning steps could be degenerate. This would lead to a running time of $\Theta(n^2)$ in the worst case.