COS 226	Algorithms and Data Structures	Spring 2021

Final Exam Solutions

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

2. **Memory usage.**

   (a) 6,553,600 (100×2^{16})

   The running time for $V = E = 80$ *is* $102400 = 100 \times 2^{10}$. *Going down one row makes the running time go up by a factor of* 4*; going right one column makes the running time go up by a factor of* 4. *So, the running time for* $V = 320$ *and* $E = 160$ *is* $100 \times 2^{10} \times 4^2 \times 4 = 100 \times 2^{16}$.

   (b) $\Theta(V^2E^2)$

   When you double V*, the running time goes up by a factor of 4, so the exponent for* V *is* $\log_2 4 = 2$. *When you double* E*, the running time goes up by a factor of 4, so the exponent for* E *is also* $\log_2 4 = 2$.

   (c) $\Theta(V + E)$

   This code fragment iterates over all of the edges in the graph (twice).

3. **String operations.**

   (3.1) n^2

   String concatenation takes time proportional to the length of the resulting string, so the running time is $\Theta(1 + 2 + 3 + \ldots + n) = \Theta(n^2)$ *in the worst case (when the input string has no space characters).*

   (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) $n \log n$

   As with mergesort, each recursive call takes $\Theta(n)$ *time, plus the time for two subproblems of length* $n/2$. *So, the overall running time is* $\Theta(n \log n)$.

   (3.4) `removeSpaces1()`

   The best case occurs when the input string consists of n *space characters. In this case, the running time is* $\Theta(n)$.

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 first call to key-indexed counting)*

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

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

5. **Graph search.**

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

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

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

6. **Minimum spanning trees.**

(6.1) 10 20 40 50 70 80 110

(6.2) 50 10 20 40 70 80 110

(6.3) 91

If $x > 90$, the edge of weight 90 will replace v-w in the MST. If $x = 90$, there will be to MSTs—one containing v–w and one containing the other edge of weight 90.

7. **Shortest paths.**

0 3 1 4 2 5

Dijkstra's algorithm relaxes the vertices in increasing order of distance from s.

8. **Maximum flow.**

(7.1) $33 = 7 + 3 + 23$

Flow conservation implies that the flow leaving s equals the flow arriving at t.

(7.2) 6

The flow leaving s is 33. The flow arriving at t is $x + 7 + 20$. These quantities are equal.

(7.3) $38 = 34 + 4$

(7.4) $A \to G \to B \to H \to I \to D \to J$

(7.5) $3 = \min\{4, 8, 4, 5, 3, 6\}$

(7.6) $\{A, B, C, F, G, H, I\}$.

After augmenting 3 units of flow along the augment path above, there are no remaining augmenting paths, so it is a maxflow. We can find a mincut by finding the vertices reachable from s via forward edges (that aren't full) or backward edges (that aren't empty).

9. **Ternary search trie search.**

D G I H F B

10. **Data compression.**

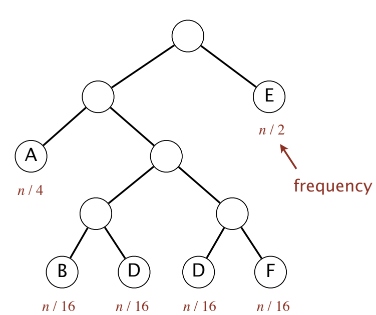
(9.1) P E T A L S

(9.2) $2n$

The corresponding Huffman trie is at right.

The number of bits to encode a message of length n is

$4(n/16 + n/16 + n/16 + n/16) + 2(n/4) + 1(n/2) = 2n$.



(9.3) 41 81 82 83 80

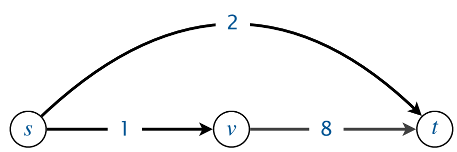
11. **Properties of graph algorithms.**

(11.1) X X X

- *Maximizing a function is equivalent to minimizing its negative. Prim's algorithm (and Kruskal's algorithm) each work with negative weights.*
- *Prim's algorithm (and Kruskal's algorithm) each depend only the relative order of the edge weights (and not on the actual values). For positive edge weights, $x > y$ if and only if $1/x < 1/y$. So, both algorithm will compute a maximum spanning tree when using the reciprocal weights.*
- *This is equivalent to negating the edge weights since, after negating the edge weights, Kruskal's algorithm processes the vertices in decreasing order of their original weight.*

(11.2) X O X

- *Maximizing a function is equivalent to minimizing its negative. Bellman–Ford works with negative edge weights provided there are no negative cycles. The digraph is a DAG, so there are no directed cycles.*
- *Here's a small counterexample to see why taking reciprocals doesn't lead to longest paths. Using the original weights, the longest path is $s \to v \to t$. But, if we use the reciprocal weights, the shortest path is $s \to t$.*



- *This is the same dynamic programming recurrence that we saw in lecture for computing shortest paths in DAGs, except with* min *replaced by* max*. It was also an iClicker question.*

12. **Properties of string algorithms.**

(12.1) X X X

Key-indexed counting takes $\Theta(n+R)$ time, uses $\Theta(n+R)$ extra memory, and is stable

(12.2) O O X

- *In the worst case, it takes $\Theta(mR)$ time to search for a string of length m since there might be R left/right links to follow before matching each character.*
- *The memory of a TST is proportional to the number of nodes. But there might be more than one node per string, e.g, if the strings are long and don't overlap much.*
- *The shape of the TST is different depending on whether you insert the string* `"AAAAA"` *before or after the string* `"BBBBB"`.

(12.3) O X X

- *The best compression ratio is achieve on inputs that alternate between 255 0s and 255 1s, as in the iClicker question.*
- *This leads to a compression ratio is 8/1, which is the worst possible.*
- *The running time of run-length coding is $\Theta(n)$. After reading each bit, the run-length coding algorithm updates a counter and, possibly, writes an 8-bit integer.*

13. **Problem identification.**

(13.1) Possible

Use either DFS or BFS, as in precept.

(13.2) Possible

Use either DFS or BFS, as in precept.

(13.3) Possible

Add super source s (connected to all vertices in top row); replace each undirected edge with two anti-parallel directed edges; and run Dijkstra's algorithm from s.

(13.4) Possible

Sort using LSD or MSD radix sort. Since $R = 2$, this takes $\Theta(n)$ time. Duplicate IPv4 addresses will be adjacent.

(13.5) Impossible

Any Huffman code can be represented as a binary trie in which each node is either an internal node (with two non-null links) or an external node corresponding to a character (with two null links). Thus, the sibling of the node containing the longest codeword must also be an external node.

(13.6) Impossible

If the LZW table contains a codeword for s, it contains a codeword for all prefixes of s.

14. **Simple directed path.**

The key idea is to repeatedly push the vertices in the path P onto a stack in the order they appear in P), keeping track of which vertices are currently on the stack. Before pushing vertex v onto the stack, check if v is already on the stack:

- *If v is not on stack, push v onto stack.*
- *If v is on the stack, repeatedly pop vertices from the stack until reaching the other copy of v.*

This algorithm maintains the invariant that the stack defines a simple path from s to the vertex v in P under consideration.

We can meet the performance requirements by using the following data structures:

- Use a *linked list* (or *resizing array*) for the stack. The stack operations take $\Theta(n)$ time since each vertex in P is pushed onto the stack once and popped from the stack at most once.
- Use a *red–black BST* to determine which vertices are on the stack. Maintaining the set of vertices on the stack and answering queries takes $O(n \log n)$ time.

  Note that a *vertex-index array* does not meet the full-credit performance requirements. It takes $\Theta(V)$ time to initialize the array and n can be much smaller than V.

```
Stack<Integer> stack = new Stack<>();   // stack of vertices
SET<Integer> set = new SET<>();         // vertices on stack
for (int v : inputPath) {

   // v not currently on stack
   if (!set.contains(v)) {
      stack.push(v);
      set.add(v);
   }

   // v already on stack
   else {
      while (stack.peek() != v) {
         int w = stack.pop();
         set.remove(w);
      }
   }

   // reverse the stack
   Stack<Integer> simplePath = new Stack<>();   // simple path from s to t
   for (int v : stack)
      simplePath.push(v);
}
```

Alternative approaches:

- Run BFS or DFS from s in the digraph G. This will find a simple path. But it takes $\Theta(E+V)$ time in the worst case, which doesn't meet the performance requirements.

- Build a digraph G' containing only the edges in P and run BFS or DFS on this subgraph of G. Naively, this takes $\Theta(n+V)$ time, which doesn't meet the performance requirements. The reason for this is that G' still has V vertices (even if some of those vertices have no incident edges).

  To turn this into an $O(n \log n)$ time algorithm, we represent the digraph as a symbol table mapping vertices to adjacency lists (instead of an array of length V mapping vertices to adjacency lists). Specifically, we use a red–black BST to map from vertices in P to linked lists of adjacent vertices. Now, run BFS or DFS in this digraph G' (replacing vertex-indexed arrays such as `marked[]` and `edgeTo[]` with red–black BSTs). Since G' has at most n vertices and edges, BFS or DFS takes $O(n \log n)$ time, with the bottleneck being the red–black BST operations.

- Here's an elegant, full-credit solution. Create a red–black BST whose keys and values are vertices, with the interpretation that `st.get(v)` gives the next vertex in the simple path from s to t after v. To initialize the symbol table, insert they key–value pairs corresponding to the sequence of edges in P.

  ```
  // initialize the symbol table st
  for (int i = 0; i < path.length - 1)
      st.put(path[i], path[i+1]);
  ```

  Then you can construct the path P' by starting at s and following the `st.get(v)` pointers until reaching t.

  ```
  // construct the simple path
  Queue<Integer> simplePath = new Queue<>();
  int v = s;
  simplePath.enqueue(v);
  while (v != t) {
      v = st.get(v);
      simplePath.enqueue(v);
  }
  ```

  Since, when inserting a key–value pair for a key already in the data structure, a symbol tables overwrite the old value with the new value, the algorithm maintains the invariant that there are no repeated vertices in the `st.get()` graph (except possibly the last one). This lone cycle is not problematic because the construction algorithm stops as soon as it finds the first copy of t.

15. **Missing string.**

(a) $\Theta(nk)$
There are n strings, each containing k bits.

(b) *The main idea of the full-credit solution is to sort the strings and check for gaps between adjacent strings in the sorted array.*

More specifically

- Sort the strings using *3-way radix quicksort*. Since, $R = 2$, the worst-case running time is $\Theta(nk)$ and it uses only $\Theta(k)$ extra space (for the function-call stack).
- To look for a gap, first check whether `a[0]` is all 0s and `a[n-1]` is all 1s. If not, return either all 0s or all 1s as the missing string, respectively. Otherwise, manipulate `a[i]` and `a[i+1]` as k-bit integers; if `a[i+1]` does not equal `a[i] + 1`, return `a[i] + 1` as the missing string.

  "Adding 1" to a binary string of length k takes $\Theta(k)$ time, using an array. So, looking for gaps takes $O(nk)$ time and uses $\Theta(k)$ extra space.

As an alternative to 3-way radix quicksort, you can modify MSD radix sort to replace key-indexed counting with an in-place variant. This can be done ala 2-way quicksort partitioning because $R = 2$.

Here are a few partial-credit variants that do not quite meet the performance requirements:

- Using LSD or MSD radix sort takes $\Theta(nk)$ time but uses $\Theta(n + k)$ extra space for key-indexed counting.
- Using mergesort takes $\Theta(nk \log n)$ time in the worst case. For example, if all of the strings start with $k/2$ 1s, then every compare takes $\Theta(k)$ time and there are $\Theta(n \log n)$ compares.
- Using a 2-way trie takes $\Theta(nk)$ time but uses $\Theta(nk)$ extra space in the worst case. For example, if all of the strings end in $k/2$ 1s.

(b′) *The main idea of the extra-credit solution is to determine the first bit in a missing string; then find the remaining bits by recursively examining only the strings that start with that bit. To do so, rearrange the strings into two subarrays: those that start with 0 and those that start with 1 (e.g., using 2-way quicksort partitioning). Use whichever bit appears least frequently as the first bit in the missing string. This takes $\Theta(n)$ time and uses $\Theta(1)$ space. Finally, recur in the smaller subarray to determine the remaining $k - 1$ bits.*

The overall running time of this approach is $\Theta(n + k)$.

- The $\Theta(k)$ term comes from building the missing string by repeatedly appending one bit at a time, using a `StringBuilder`.
- The $\Theta(n)$ term accounts for all of the work from rearranging the strings and determining which bit appears least frequently. For the first bit, there are n strings to consider; for the second bit, there are at most $n/2$ strings to consider; for the third bit, there are at most $n/4$ strings to consider, and so forth. Recall $n + n/2 + n/4 + \ldots = \Theta(n)$.