1. Exercise 10 Page 110

We can find the number of shortest v-w paths with slight modification of BFS algorithm. In addition to pushing a vertex to a queue, we will also store information about what level that vertex belongs to. Once we encounter vertex w at level l, then we dequeue a vertex until its level is $l+1$. For each vertex we dequeued, we check if that vertex can reach w. If it does, we increment the number of shortest v-w paths by 1. Since it's basically a BFS with some extra O(1) operations in each step, the time complexity of this algorithm is the same BFS, which is O($V+E$) where V denotes the number of nodes and E denotes number of edges.

The algorithm is the following:

```
Let Q be a queue
Let count be 0
Push vertex v to Q
Mark v's level as 0
Mark v as visited
While Q is not empty:
   Dequeue a vertex v1 from Q
   For each vertex v2 that is reachable from v1:
      If v2 is not visited:
         Mark v2's level as (v1's level + 1)
         Mark v2 as visited
         Push v2 to Q
      Else If v2 is visited and v2 is w and (v1's level + 1) is v2's level:
         Increment count
Return count
```

To optimize the above algorithm, we could terminate the loop once we are processing a vertex whose level is greater than w's level. However, adding this logic to the algorithm could reduce the clarity of the pseudocode above, thus I did not include it.

2. Exercise 6 Page 108

We want to show G cannot contain any edges that do not belong to T. In other words, we want to show G is a tree. We will prove this by contradiction. Suppose G is not a tree (i.e. There exists an edge $(v_1, v_2) \in G$ such that $(v_1, v_2) \notin T$). Since T is a DFS tree, without loss of generality, v_1 is an ancestor of v_2. In other words, the distance between v_1 and v_2 is at least two. However, since T is a BFS tree, v_1's level and v_2's level can differ by at most one. This contradicts the assumption that distance between v_1 and v_2 is at least two. Thus, $(v_1, v_2) \in T$. Thus, G must be a tree (i.e. $G = T$).

3. Exercise 7 Page 108

We want to show G has only one connected component. Suppose on the contrary, G has at least two connected components. Then the smallest connected component (SCC) must have at most $2/n$ vertices. Arbitrary pick a vertex v from SCC. Since v can reach any vertex in SCC, v 's degree can be at most $2/n - 1$. This contradicts the assumption that every vertex in G has degree at least $2/n$.

4. Exercise 3 Page 189

We want to show the greedy packing algorithm “stays ahead” of all other solutions (i.e. Given n trucks, the greedy algorithm packs boxes $b_1, ..., b_k$. There is another solution that packs $b_1, ..., b_m$ boxes. Then $m \leq k$. In other words, the greedy algorithm packs more or equal to any other solution). We will prove this by induction.

Base Case: ($n = 1$)

Since the greedy algorithm packs every box until the next box does not fit, clearly the greedy algorithm packs more or equal to any other solution.

Inductive Step:

Suppose the greedy algorithm packs more or equal to any other solution for n trucks

We want to show the greedy algorithm packs more or equal to any other solution for $n + 1$ trucks

Up to nth truck, the greedy algorithm packs $b_1, ..., b_k$ boxes, and another solution packs $b_1, ..., b_m$ where $m \leq k$.

Thus, for $(n + 1)$th truck, if another solution packs $b_{m+1}, ..., b_l$, then the greedy algorithm can pack at least $b_{k+1}, ..., b_l$ because $m \leq k$ from assumption. In other words, the greedy algorithm can pack more or equal to any other solution for $n + 1$ trucks.

5. Exercise 6 Page 191

Claim:

Pick a contestant in descending order of the sum of biking time and running time minimizes the completion time.

Proof of Claim:

Let s_i, b_i, r_i be the swimming time, biking time, and running time of contestant i. Suppose there is another solution that has smaller completion time. Then, there exists contestant i and j such that j is picked right after i and $b_i + r_i < b_j + r_j$. Note completion time of these two contestants is $max(s_i + s_j + b_j + r_j,\ s_i + b_i + r_i)$. But if we swap i and j, then the completion time of these two contestants is $max(s_i + s_j + b_i + r_i,\ s_j + b_j + r_j)$. Since $b_i + r_i < b_j + r_j$, $s_i + s_j + b_j + r_j > s_i + b_i + r_i$, $s_i + s_j + b_j + r_j > s_i + s_j + b_i + r_i$, $s_i + s_j + b_j + r_j > s_j + b_j + r_j$. Therefore, $max(s_i + s_j + b_j + r_j,\ s_i + b_i + r_i) > max(s_i + s_j + b_i + r_i,\ s_j + b_j + r_j)$. In other words, swapping contestant i and j such that j is picked right after i and $b_i + r_i < b_j + r_j$ can only decrease the completion time (not changing the completion time in the worst case). Therefore, we can continuously sort contestant by $b_i + r_i$ in descending order without increasing the completion time.

6. (a)

We can run BFS from each vertex and record the longest path and its level for each BFS tree. Then we simply choose the longest path with the largest level. Since we are running BFS for each vertex, the time complexity of this algorithm is $O(V(V+E))$, where V denotes number of vertices and E denotes number of edges.

The algorithm is the following:

Let P be an array of paths
Let L be an array of levels
For each vertex v in G:
    Perform BFS and arbitrary pick a vertex w from the largest level l
    Push the path from v to w to P
    Push the level l to L
Find the index i of the largest level in L
Return $P[i]$

6. (b)

Since the graph is a DAG, we can run topological sort on the graph. Then we visit each vertex in topological order and assign distance and path to it. If a vertex v hasn't been assigned a distance, we set the distance of the vertex to be 0 and the path is v. Additionally, for each vertex we visit, we assign distance and path to each of its adjacent vertex w. w's distance will be v's distance + 1 and the path will be v's path + w if the new distance is larger and previously assigned one. Finally, we pick the vertex with the largest distance and return its path. The topological ordering of DAG takes $O(V+E)$, where V denotes the number of vertices and E denotes the number of edges. Iterating over all the vertices in topological order and check its edges also takes $O(V+E)$. Thus, the time complexity of this algorithm is $O(V+E)$.

The algorithm is the following:

```
Perform topological ordering on G and get the list of vertices L
Let target be null
For each vertex v in L:
    If v is not visited:
        Mark v as visited
        Mark v's distance as 0
        Mark v's path as v
    For each adjacent vertex w of v:
        If w is not visited or w's distance is smaller than v's distance + 1:
            Mark w as visited
            Mark w's distance as v's distance + 1
            If target is null or target's distance is smaller than w's:
                Let target be w
            Mark w's path be v's path + w
Return target's path
```