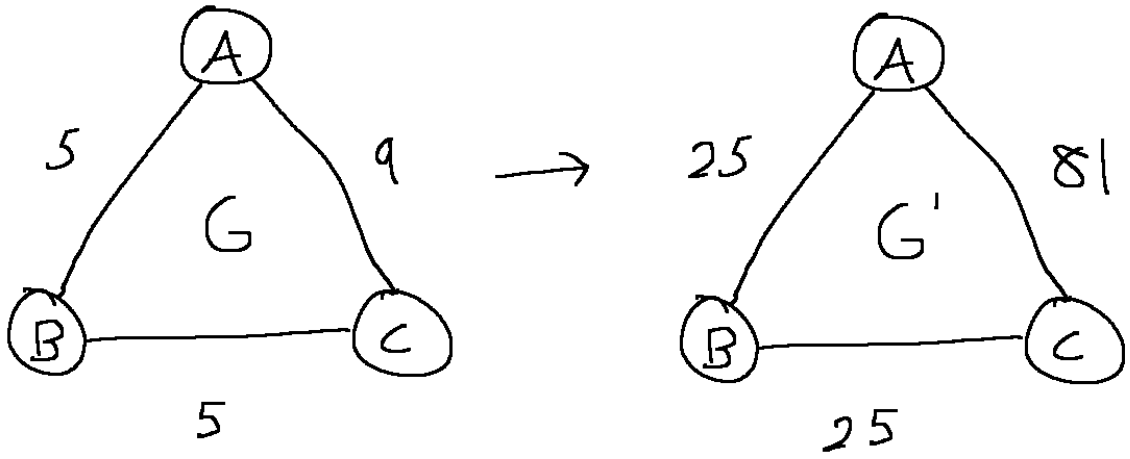
1.
Proof

Let S be the solution reached by the cashier's algorithm, and let S* be an optimal solution for the coin changing problem. Then,

- Case 1: S* chose the same coins as S. Then, S = S*.
- Case 2: S* chose different coins than S. In this case, it is possible to exchange c number of some coin denomination for a single coin of the next higher denomination. This would reduce the amount of coins used in the solution by (c-1) and thus contradicts the idea that S* is an optimal solution. It would also bring the solution S* closer to S.

Therefore the solution S chosen by the cashier's algorithm is always optimal when the denominations are c^0, c^1, …, c^k for some positive integers c > 1 and k ≥ 1.

2a. The statement is true. The minimum spanning tree for a graph can be constructed using Kruskal's algorithm, which chooses edges in ascending order of cost. Squaring each edge cost of G will not change the order in which Kruskal's algorithm chooses edges.
2b. The statement is false. Consider the graph drawn below. Originally, the shortest path from A to C involves only a single edge with a total cost of 9. After squaring each edge, the new shortest path from A to C involves two edges with a total cost of 50.



3.
Proof

Let S be the solution reached by the current greedy algorithm, and let S* be the solution reached by the new algorithm being considered. Also, let i and j be two consecutive trucks in S* that were sent off from New York where i was sent off less than full to allow j (and future trucks after j) to be better packed. We know that the greedy algorithm "stays ahead" as i is being sent off, since it is filled with as much weight as possible and therefore no other algorithm could have shipped more weight at this point. We can also show that the greedy algorithm stays ahead as j is being sent off, since all of the weight in S*'s j-truck can either be completely stored in S's j-truck, or more likely, distributed between S's j-truck and the extra space in S's i-truck. Thus the

belief that future trucks after j in S* will be better packed is also proven false. Given any two consecutive trucks i and j in S, there is no better way to pack them with weight than the greedy algorithm currently in use.

4.

Algorithm description in Plain English:

First, enqueue each contestant into a priority queue that is sorted by their combined projected biking time and projected running time. The contestants with the longest combined time should be at the front of the queue, and the contestants with the shortest combined time should be at the back. To start the triathlon, dequeue the first contestant, who will begin swimming. When they complete their 20 laps, dequeue again.

Proof of Correctness:

Let S be the solution reached by the proposed greedy algorithm, and let S* be an optimal solution that results in the fastest triathlon possible with the fewest inversions. An inversion in S* would be a pair of adjacent contestants i and j such that j has a shorter combined projected biking time and projected running time, but j is scheduled before i. Exchanging two adjacent, inverted contestants i and j reduces the number of inversions by 1 and does not increase the maximum amount of time that the triathlon will take, which contradicts the idea that S* already has the fewest inversions. Thus, the greedy solution is always optimal.

Complexity Analysis:

For a priority queue, each enqueue and dequeue occurs in O(log n) time, and this is repeated for all n contestants. Therefore, the total time complexity is O(nlog n).

5.

Algorithm description in Plain English:

The algorithm will be a modification of Dijkstra's algorithm. First, initialize a list for the v node and run Dijkstra's algorithm starting on node u. If v is about to be added to the explored area of Dijkstra's algorithm, add all $\pi(v)$ values to the list. By the end, the list should save the $\pi(v)$ values in consideration when a shortest path to v was being chosen. After the Dijkstra's algorithm completes, return False if there are duplicates of d[v]'s value in v's list, since the shortest path from u to v is not unique. Otherwise, return True if d[v] only appears once in the list.

Proof of Correctness:

- Claim (Loop Invariant): For each element $\pi(v)$ in the list, $\pi(v)$ is equal to the length of a unique path from u to v.
- Initialization: Dijkstra's algorithm initializes on node u, then begins to explore paths outwards.
- Maintenance: When the algorithm is calculating different values for a shortest path to v, it only stores the values permanently if they were being considered at the time that v was added to the explored area of Dijkstra's algorithm. By the properties of Dijkstra's

algorithm, it is impossible for a more optimal path to have been undiscovered at this point. Thus the solution will be optimal and the invariant holds.

- Termination: The algorithm ends once all nodes have been visited at which point the priority queue used for exploring the graph will be empty and d[v] will have been calculated as the shortest path length to v. At this point, the algorithm makes a final check to see if this shortest path length was unique.

Complexity Analysis:

It is known that Dijkstra's algorithm requires O(mlog n) time when implemented using a binary heap, where n is the amount of nodes and m is the combined amount of edges and nodes. The modifications proposed will require O(degree(v)) time for the list operations. Therefore, the total running time is still O(mlog n).