Question 1.1

(a)

$$ONE(i) = \begin{cases} ARR(i) & \text{if i=0} \\ MAX(ONE(i-1) + ARR(i), ARR(i)) & \\ \text{otherwise} & \end{cases} \quad (1)$$

$$ZERO(i) = \begin{cases} 0 & \text{if i=0} \\ MAX(ONE(i-1), ZERO(i-1) + ARR(i)) & \\ \text{otherwise} & \end{cases} \quad (2)$$

$$MAXSOFAR(i) = \begin{cases} ARR(i) & \text{if i=0} \\ MAX(ONE(i), ZERO(i), MAXSOFAR(i-1)) & \\ \text{otherwise} & \end{cases} \quad (3)$$

(b) Our recurrence calculates the maximum contiguous sum for a subarray at index i. There are two arrays- one calculating the maximum sum with one skip remaining and the other calculating the maximum sum with zero skips remaining and each are compared to the maximum sum so far to calculate the new maximum. In the example, ONE(i) represents the maximum sum with one skip remaining at index i, ZERO(i) represents the maximmum sum with no skips remaining at index i, and MAXSOFAR(i) represents the maximum possible sum so far at i (whether or not it includes i itself).

(c) The final answer is calculated by returning the maximum value calculated at the final index of the array. We can do this by simply calling MAXSOFAR(n-1) where n is the length of the array, which would return the largest value between ONE(n-1), ZERO(n-1), and MAXSOFAR(n-2).

(d) We would use 3 arrays as our memoization structure- one would keep track of the maximum value with one skip remaining, another would keep track of the maximum value with 0 skips remaining, and the last would keep track of the maximum sum so far at each index.

(e) The run time will be O(n) because we will have to run through the entire array once. We will need to compare elements in constant time to come up with the maximum sums at each point.

2.2

(a) Our algorithm will need to have a setS that has all nodes that are even reachable (initially just the source), a queue Q that keeps track of even distance nodes that have not yet been explored (initially just the source), and a map M of type (node : set of nodes) where the key-value pairs are every node and the nodes that can be reached within 2 edges of it. To begin, we run a BFS from the source to everything that is within 2 edges away from it. Everything that is 1 edge away will be added to the map's value with the source node as the key and everything 2 edges away will be added to the map's value with the source node as the key and to Q and to S. We would also update each of the boolean fields of all nodes 2 edges away to be true. Once we have finished all these steps, we have officially visited the source node. We then repeat the same step from every node in Q, each time removing it once we have finished visiting it. If at any point we are visiting a node X that is marked as even and encounter a node 2 edges away, we update this node's boolean field to be true and add it to S, Q, and and M with the key of X. If we are ever visiting a node X that is marked as even and find that we are one edge away from another marked-as-even node Y, this means we have found an odd-length cycle in the graph. To account for this, we can find the set of values in M for nodes X and Y, and mark every node in these two sets' boolean fields as true. We then add these nodes to S and Q. We repeat this until there are no nodes left to visit in Q. At this point, we check if every vertex in the graph is in S by comparing the sizes, and if S is smaller than the total number of nodes, then return false; otherwise we return true.

(b) Our algorithm will traverse each edge once and if any edge results in an even length path, we only need to visit this node once. We will never re-visit a node, so in the worst case we will visit n vertices and m edges. Since adding to S, M, and Q take constant time, this means our end runtime will be O(m+n).

(c) Our algorithm initially finds nodes that are distance 2 away from the source and then uses the basis that every node that is even-length from the source will be 2 away from a node that is 2 away from a node... that is 2 away from the source. The exceptions to this are when we encounter an odd-length cycle that enables us to reconfigure a path to certain nodes since adding two odd numbers creates an even number. We account for this by manually editing the nodes that are odd distance and connected to a node that is part of an odd-length cycle. We then check if every node is visited after Q is empty because this means there are no more possible nodes that are 2-edges away from any node we have visited already. We then return true if we have visited every vertex because this means that all vertices were even-length away, and if not we return false because there were some vertices that we were unable to visit in even steps.

Question 3.1

(a) When given a list of variables and a list of constraints in standard 3-SAT form, the constraints are given in the form of $(x \wedge / \vee y \wedge / \vee z$ where each $\wedge/\vee$ means one of and, or. To transform this into DOUBLE-3-SAT, we can find some variable x that is not in the set of variables passed in, and create a clause $(x \vee \neg x)$. We then add this clause to the set of constraints passed in, which can be done in polynomial time. We then run 3-SAT using the original variable and constraints and return if it is satisfiable or not.

(b) If 3-SAT returns true, then our DOUBLE-3-SAT will have every single constraint from the original parameters equal to true. Then, we have 2 possible true return options with x = True or x = False as in both of these cases $(x \vee \neg x)$ is true. Our algorithm will return true every time 3-SAT returns true then.

If 3-SAT returns false, then that means there was some constraint that was not met in the original set of constraints. Our algorithm will then return false as regardless of x being true or false, we will not be able to return true for the original set of constraints, therefore returning false.

Question 4.1

(a) For an array A with length n, our algorithm will need a new array B of size n to keep track of the maximum difference at each index so far and keep track of x, the last possible point for a new contiguous subarray (or when the elements stop increasing at any point). We start with setting x=0 and A[0] = 0. Then, while $A[i] <= A[i+1]$, we set B[i] to be A[i+1] -A[x]. If $A[i] > A[i+1]$, then we set B[i+1] = B[i] and x = i+1. We repeat this for the entire array and return the value at B[n-1].

(b) We use an array as our memoization structure and the array at each index i will store the maximum value between the maximum difference at the previous index and that between the current index and the start of the longest contiguous subarray containing index i. This accounts for any possible decrease in the array as we will restart our contiguous subarray, and will ensure we have the maximum difference possible because if the values are increasing, the difference will be greater.

(c) This will run in O(n) time as we will have to iterate through the entire array once while only needing to check the value at B[i-1] and A[i]-A[x] to calculate B[i], which can be done in constant time.

Question 5

How long did you spend on the exam (approximate to the nearest hour?

Four hours roughly.

How long did you study for the exam before the exam started (approximate to the nearest hour)?

0

How many other students did you talk to (please state their names as well)?

One- Simin Liu

If you talked to others, was it helpful?

Yes absolutely, it helped to bounce ideas off a lot.

In how many of the four sections did you spend significant time attempting both problems?

Sections 1 and 2

Did you like selecting between problems, or would you rather have just gotten one question per topic?

Selecting helped a lot- we could pick things we were more confident in or were easier to understand/less time consuming.