Question 2

(a)

$$T(n) = \begin{cases} min(T(n-c_y))+1, & \text{if } c_y < n. \\ T(n), & \text{otherwise.} \end{cases} \tag{1}$$

This recurrence shows that the number of coins for any sum n is dependent on the value of the coins in the coin set. If the sum is smaller in value than the coin we are checking (c_y), then 0 of this coin can be used. Therefore, the optimal count is equal to the previously found number of coins for the sum n. If the coin is less in value than n, then we can calculate how many coins it took to minimize n-c_y and add 1 to it.

(b) We can use an iterative approach to solve this question by creating a 2D array with rows of size k+1 and columns of size n+1. The first n columns will represent how many of each coin we will use and the last column will contain the total number of coins used per row. For the row representing a total of 0, we fill every column with 0. For the row representing a total of 1, we fill the 0 column with 0, the 1 column with 1, the 2 through n-1 columns with 0, and the n column with 1. For all other rows, we fill every cell with 0. We then iterate through the array, and for every row index i, find all coins n_x in the set of all coins such that $n_x <= i$. For every one of these coins, we will use one more coin than the value at $[i-n_x]$[n] as this is the total number of coins for every sum. Therefore, we check which coin has the minimum value at $[i-n_x]$[n] and set every value in row i equal to the value in $i-n_x$ while incrementing [i][n_x] and [i][n] by 1 each. We can repeat this logic until we finish the row representing a total of k (the k+1th row), at which point we can return the array at this row from index 0 to index n-1.

(c) Our code will run in $O(cn)$ time because we will have to iterate through every coin once for every number from 1 to the total change required (in the worst case). Our code will use $O(cn)$ memory because our only data structure is the 2D array with n rows and c+1 columns.

(d) The greedy algorithm could be better than the dynamic programming algorithm if we are looking for an algorithm with less memory space. The only caveat is that we would need prior knowledge of the set of coins that we have so we can judge whether or not greedy can always work. If we would prefer an algorithm that will always work regardless of memory used, the dynamic programming algorithm is better at it will work without prior knowledge of the coin set.

Question 3

(a) Let DP(i) be the maximum possible happiness at i for some array of integers.

RECURRENCE:

DP(i) = max(DP(i - 1) + a[i], DP(i - 2) + a[i], DP(i - 3) + a[i], DP(i - 4) + a[i] , DP(i - 1), DP(i - 2), DP(i - 3))-(20 if SKIPS(i)=2, 20 if SKIPS(i)=3, ∞ if SKIPS(i)=4) IF $i > 3$

SKIPS(i) = 0 IF a[i] > 0 AND i > 0

SKIPS(i) = x+SKIPS(i-x) IF a[i] < 0 AND i > 0 AND DP(i) = DP(i - x)+A(i))

SKIPS(i) = x+SKIPS(i-x)+1 IF a[i] < 0 AND i > 0 AND DP(i) = DP(i - x))

BASE CASE(S):

DP(i) = a[0] IF i = 0

DP(i) = max(a[0], a[1], a[0] + a[1]) IF i = 1

DP(i) = max(a[0], a[1], a[2], a[0] + a[1] + a[2], a[0] + a[1], a[0] + a[2], a[1] + a[2]) IF i = 2

DP(i) = max(a[0] + a[1] + a[2] + a[3], a[0] + a[1] + a[2], a[0] + a[1]-20, a[0]-20, a[1] + a[2] + a[3], a[1] + a[2], a[1]-20, a[2] + a[3], a[2], a[3]) IF i = 3

SKIPS(i) = 0 IF a[i] > 0 AND i = 0

SKIPS(i) = 1 IF a[i] < 0 AND i = 0

DP(i) is the maximum happiness for the sub-array $0, i$ and is calculated using DP at previously calculated indices and SKIPS(i). SKIPS is calculated during the iterations and is based on which index results in the maximum at DP(i).

(b) In order to do this recurrence, we will use the fact that we know the maximum possible sum up to any one point i by storing that as DP(i). We will also have our passed in int[] happy array and store the number of skips we have available at SKIPS(i), which stores how many we have skipped at i to get the maximum sum in DP. With this recurrence, for our base cases, we fill out the array from indices 0 to 3 with the maximum happiness at each step. This involves some combination of the previous indices in the happy array while subtracting 20 if we ever skip 2 or 3 indices. For the recursive cases, we calculate the maximum between the 4 previous indices plus the current index and the 4 previous indices. At each step, we account for the possibility of there being multiple skipped by checking the value of skips at the same index, which is calculated based on which previous index we are using to maximize DP(i) (DP(i-1), DP(i-2), DP(i-3), or DP(i-4)).

(c) Our memoization structure is an array with n indices (assuming the happy array has n indices) where each is filled iteratively so that the first calculated index is 0, then 1, and so on until n-1. In the end, our final answer is the value at index n-1.

(d) The runtime for this code is $O(n)$ because we will need to iterate through the array once for all n indices.