Programming Assignment 3 - Rishab Dudhia (SID: 862141444)

a. To begin the program, I declared a priority queue. However, the default priority queue is maintained as a max heap, but since we needed to look at the minimum values of the bags of candies, I needed the extra parameter “greater<int>” in the declaration of the priority queue. After reading the initial sizes of the bags of candies into the priority queue, I passed in the priority queue, named candyQ, and an empty vector, named pastPiles, into my function, countCandies. The function pops the two values off the top of the priority queue, the two smallest piles. The sum of the two piles, the merged pile, is pushed back into the priority queue and the merged pile is also pushed into the pastPiles vector. The function is called again with the same parameters, this time with candyQ having size n-1. The base case of the function is when the priority queue is only of size one, meaning that all the piles have been merged. After the countCandies function returns and we are back in the main function, the pastPiles vector contains all the piles that were made in merging to one bag. Then I add all the values in the vector and multiply the value by two since each candy was counted twice, once before the two piles were merged and once after.

b. This problem was the 0/1 knapsack problem. I followed to solve the problem through dynamic programming. I created a vector of pair<int, int>’s with the pair’s first value being the weight of item i and the second value being the value of item i, with i being the i^{th} element, item, in the vector. I also remember the lowest weight in a variable called lowWeight. Then I created a vector of vector<int>’s called weightVals to remember the values of the calculated calls of weight remaining and item. The vector was of size *number of items*, *n*, + 1 x *total weight allowed* + 1. All the values in the table of n = 0 are set to 0 and also all the values in the table of weight remaining = 0 are set to 0. Then the

function inSuitcase is called with parameters maxWeight, the number of items, weightVals, the vector of pair<int, int>'s called items, and the lowWeight. The base cases, or boundary cases, are if the value for the called remaining weight and item has already been calculated, if the remaining weight is less than the lowWeight, and when item = 0 because the 0^{th} row in the table does not represent any item since there are n+1 rows. If none of the base cases are achieved with the call to the function, the decision for the function to make is to pick the current passed in item with the remaining weight or not to pick it. The value currBest is calculated to simulate the situation of not picking the item. Then, if the weight of the item is less than the remaining weight, the temp variable is calculated to simulate picking the item. Then, if the temp value is greater than the currBest value, the currBest value is set to temp. Then in the table, I set the value of the remaining weight and the current item to currBest since that's the best value we could achieve with the decision to pick the item or not to. The value is saved into the table so that the next time the function is called with this item and remaining weight, the base case saying that the value has been calculated will be reached and the value will be returned right away.

c. This problem was the bounded knapsack problem. To solve it, I just had to manipulate my solution to part b. Instead of my items vector being a vector of pair<int, int>'s, I made it a vector of tuple<int, int, int>'s with the last value representing how many times the item has been picked. It was initialized to 0 when reading in the weights and values for the other items. My weightVals vector needed one more dimension to remember the values of the situation of weight remaining, item, and how many times it has been used. Therefore, the weightVals vector was declared as "vector<vector<vector<int>>>

weightVals.” In the function, we have the same base cases as part 3. We also remember the value of how many times the current item had been used prior to making the decision. Then, currBest is calculated and represents the same as in part b. However, for temp to be calculated, not only must the item’s weight be less than the remaining weight, but the item must be used less than the max amount of times that it can be used. I also created a bool variable to represent if the last value of the tuple for the item was manipulated or not. Then, the rest of the function is the same as part b, except for one if statement. This statement checks if the third value of the item’s tuple was incremented in the calculations, and decrements it if so. I do this so that the item seems unchanged so that it can be used in further calculations in other calls to the function after returning from the current function.

d. For this problem, I read in the map of the resort in a vector called resort declared as “vector<vector<int>> resort.” Then, I created an identical vector with respect to size called resortVals which will represent the calculated values of calls to the corresponding cells in the map. Then I passed in the two tables with their dimensions into my function allPaths. The function starts at the top left corner of the resort map and calls pathFrom (parameters being the current cell, resort map, resortVals, and dimensions) on each cell to calculate the longest route from that cell. We keep track of the longest path with variable max to be returned. The function pathFrom calls its helper function with the same parameters, but with an additional being the current value of the cell. The helper needs this value to check one of its base cases. The base cases are if the location values are outside of the map, if the passed in x value, the value of last cell, is larger than the value of the current cell, and if the value of item, weight remaining, and times the item was

used has been calculated. If none of the base cases have been reached, we calculate four values: left, right, up, down. These values are calculated by calling the helper function with the location of the passed in cell corresponding to the name of the variable being calculated. The max of these four values is remembered in weightVals and it is returned to the pathFrom function. The pathFrom function contains the same body as the helper function without the base cases since it is only the initial call. Once the allPaths' double for loop finishes calculating the longest paths from each cell on the map, the max of the longest paths is returned to the main which is our final answer. Not sure where my code is wrong as my answers are being accepted, but just using too much memory on 6 inputs.