CMPS 102 — Fall 2018 – Homework 3

Alyssa Melton

I have read and agree to the collaboration policy.

Collaborators: none

Solution to Problem 4

Alice has a gift card with *N* dollars on it and has a choice of *m* cakes to buy. We want to find how many different ways she can buy the cakes so that she uses all of the money on her gift card.

There is a counting problem like this one that is very similar that asks to count how many ways we can sum to a given number using positive integers. In this case, we check every possible combination of numbers until the sum exceeds n. Using that as a basis I constructed the following algorithm which involves some definitions a subroutine:

thisCombo = a subset of the indices of the set of cake prices, where the sum of the values of denominations at the indices found in this array will or will not equal the total n.
sum(thisCombo) = helper function that adds all values of denominations at the indices found in this array. Example: if thisCombo = [1, 2], the sum returned is denominations[1] + denominations[2]
comboIndex = the index of thisCombo, so we can track the next empty spot and previously added spot.
denominations = set of cake prices, or denominations we can use to add to the given total n.
n = gift card amount.
total = count of how many total ways there are to buy the cakes which is updated throughout getWays.

getWays(thisCombo, comboIndex, denominations, n)
    if $(sum(thisCombo) > n)$
        (thisCombo exceeds n)
        break loop

    if $sum(thisCombo) == n$
        (thisCombo is exactly n, so it is a valid combination)
        total++
        break loop

    (else $sum(thisCombo < n)$, so want to try to add more to it:)

    if comboIndex = 0 (haven't added anything to thisCombo yet, start at beginning of combo array)
        prev = 0
    else (we want to know what the denominations index of the last number added was)
        previous = thisCombo[comboIndex-1]

    (basically want to check all possible combinations:)
    for $(i = previous; denominations[i] \leq n; i++)$
        thisCombo[comboIndex] = i
        getWays(thisCombo, comboIndex+1, denominations, n)

Claim 1. *This algorithm is running time $O(mn)$.*

Proof. Assumption: only cakes less than or equal to n will be included in m.
At the first call of getWays, we get a loop that will go through all the denominations which is m. Now, we will calculate how many calls to getWays each subcall will do. Because each call to getWays we are adding a cake price to our combination, at the worst case we can only add up to n cakes, in which they are all at price 1. Thus, the runtime is $O(mn)$ where we are calling in a loop that goes through m times, and getWays is recursively called at most n times within each iteration of that loop. □

Claim 2. *This algorithm works.*

Proof. This algorithm is necessarily checking all possible combinations, and it incrimenting total when a valid combination is found. To say this algorithm doesn't work is to say that it is not checking all possible combinations, so that is what we will prove.

Assume that not all combinations were checked, and a valid combination was not accounted for. In getWays, we loop through all denominations from the previous, until greater than n. Then, there must be a denomination outside of the bounds previous to greater than n. Well, if a denomination is greater than n, we cannot buy that cake because we don't have enough money on the gift card, so this must not give a valid combination. The combination was be found in an element less than the previous. But, if the number is less than the previous, it is already accounted for and we would have already created this combination. Thus we have a contradiction as all numbers were accounted for. □

Claim 3. *This algorithm is space complexity O(n)*

Proof. The algorithm has an array that holds combination, which will be $O(n)$ in the worst case (a denomination = 1). We also have a few single variables. □