```julia
# ------------------------------------------------------------------------------
# Author: Philip Coyle
# Date Created: 06/17/2020
# ps3.jl
# ------------------------------------------------------------------------------

## Question 1
function change_making(cents::Int64, coins_vec::Vector{Int64})
    # Preallocate space for number of ways to make i cents from coins
    # (i being index of vec

    num_ways = zeros(cents+1) # We add one to account for 0 case
    num_ways[1] = 1  # There is one way to make zero cents given coins (Base Case)

    # Dynamic Programming Step
    for i = 1:length(coins_vec) # Iterate over each coin denomination
        for j = 1:length(num_ways) # Iterate over number of ways to make each cent (0:cents)

            if coins_vec[i] <= j-1
                # if coin denomination leq number of cents trying to make,
                # update num_ways by seeing how much is left over and
                # adding num_ways[left_over] to current index

                # Ex: if cents = 3 and coins_vec = [1,2]
                # num_ways = [1, 0, 0, 0] (making 0 cents)
                # num_ways = [1, 1, 0, 0] (making 1 cent with 1 cent coin) -- i = 1; j = 1
                # num_ways = [1, 1, 1, 0] (making 2 cents with 1 cent coin) -- i = 1; j = 2
                # num_ways = [1, 1, 1, 0] (making 2 cents with 1 cent coin) -- i = 1; j = 3
                # num_ways = [1, 1, 1, 1] (making 3 cents with 1 cent coin) -- i = 1; j = 4

                # num_ways = [1, 1, 2, 1] (making 2 cents with 1 & 2 cent coins) -- i = 2; j = 3
                # num_ways = [1, 1, 2, 2] (making 3 cents with 1 & 2 cent coins) -- i = 2; j = 4

                # There are 2 ways to make 3 cents from a 1 and 2 cent coin

                left_over_inx = j - coins_vec[i]
                num_ways[j] = num_ways[left_over_inx] + num_ways[j]
            end
        end
    end
    return num_ways[end]

end

N = 10
S = [2, 5, 3, 6]

z = change_making(N,S)

## Question 2
function rod_cutting(inches::Int64, prices::Vector{Int64})
    val = zeros(inches,1)
    cuts = zeros(inches,2)

    # Base Case
    val[1] = prices[1]
    cuts[1, 1] = 1

    # Dynamic Programming Part
    for i = 2:inches
        candidate_max = 0
        opt_cut = 0

        for j = 1:i # Loop over possible cut points
            cut = j

```

```
            if cut == i # Condition on if its most profitable to sell as whole
                temp_val = 0
            else
                temp_val = val[i - cut]
            end
            temp_val += prices[cut]

            # Update value
            if temp_val > candidate_max
                candidate_max = temp_val
                opt_cut = cut

            end
        end

        # Store value and cut point
        val[i] = candidate_max
        cuts[i,:] = [opt_cut, i-opt_cut]
    end

    return val[end], cuts
end


n = 8
P = [1, 5, 8, 9, 10, 17, 17, 20]
val, cuts = rod_cutting(n,P)

## Question 3
function knapsack(weight::Vector{Int64}, val::Vector{Int64}, cap::Int64)
    # Allocate space for maximum value
    W = zeros(cap,1)
    V = 0
    item = zeros(length(weight),1)

    # Base case: capacity = minimum weight
    # Note, we sort the weight vector (lowest to highest) for siplicity
    permutation = sortperm(weight)
    weight = weight[permutation]
    val = val[permutation]

    min_weight = weight[1]
    W[min_weight] = min_weight

    # Dynamic Programming Step
    for i =  1:length(weight) # Loop over all item weights
        w = weight[i]

        for j = min_weight+1:cap # Loop over all capacities
            if j - w > 0
                wght_tmp = w + W[j - w] # Leftover weight (We call past solved for weights -- D.P. step)

                if wght_tmp >= W[j]
                    W[j] = wght_tmp

                    # Determine which items are used
                    candidate_item = get_items(i, j, w, weight)
                    candidate_max = sum(val.*candidate_item)

                    # Update the global value and items
                    if candidate_max > V
                        V = candidate_max
                        item = candidate_item
                    end
                end
            end
```

```
        end
    end
    return V, convert.(Int64,item)
end

function get_items(i::Int64, j::Int64, w::Int64, weight::Vector{Int64})
    item = zeros(length(weight),1)
    item[i] = 1
    diff = j - w

    min_weight = weight[1]

    k = max(i-1, 1)
    weight = weight[1:k]

    while diff >= min_weight
        vec_diff = diff .- weight
        min_diff = minimum(vec_diff)
        tmp_inx = findfirst(isequal(min_diff), vec_diff)

        if typeof(tmp_inx)  != Nothing
            item[tmp_inx] = 1
            diff = diff - weight[tmp_inx]
            if tmp_inx > 1
                weight = weight[1:tmp_inx - 1]
            else
                break
            end
        end
    end
    return item
end


W = [10, 20, 30]
val = [60, 100, 120]
C = 50
val_max, item = knapsack(W, val, C)
```