John Higgins
Programming Bootcamp
20 July, 2022

Note: All code and plots can be found here!

Problem 1.

Solution: Code and putput for problem 1 is included below:

```
function matrix_fill(N::Int64, coins::Array{Int64})
    #initialize an empty array of size N+1 x the amount of coins + 1
    #note: because I need a row of ones at the beginning (so that successful ways of making change get counted), the indices of everything will be shifted up by 1. It's not pretty, but it works
    res = zeros(N+1, length(coins)+1)
    #create the aforementioned row of ones
    res[1, :] .= 1
    #sort the list of coins we have, because the way I have written it depends on the list of coins being ordered
    coins = sort(coins)
    #iteratively fill in the rows of the matrix
    for val=1:N
        #iterate over possible coins to use
        for (i, coin) in enumerate(coins)
            if coin > val #if the coin is larger than the remaining balance
                res[val+1,i+1] = res[val+1, i] #we cannot make change using this coin, so we use the previous count for the first i-1 coins
            elseif coin == val #if the coin is exactly the same as the remaining balance
                res[val+1,i+1] = 1 + res[val+1, i] #use this coin to make exact change and thus we need to increment the current count by 1
            else
                #can always make change with the lower denomination coins and ignore the new coin
                res[val+1,i+1] = res[val+1,i]
                #find the highest quantity of coin i that can be used
```

```
                j_range = Int(floor(val/coin))
                #loop over possible multiples of the current coin, up to
    j_range
                for j=1:j_range
                    #add the number of ways to make change for the
    resulting balance with the first i-1 coins and add to the current entry
                    res[val+1, i+1] += res[val + 1  - j*coin, i]
                end
            end
        end
    end
    #return the resulting matrix, as well as the total number of ways (it
    will be the entry in the bottom right corner of the matrix)
    res, res[N+1, length(coins)+1]
end

#find the total amount of ways
full_matrix, total_ways = matrix_fill(10, [2,3,5,6])
total_ways
>>5.0

```

□

Problem 2.

Solution: Code and output for problem 2 is included below:

```
#create bellman function for problem: takes rod length N and price vector
    P
function rod_bellman(N::Int64, P::Vector{Int64})
    #assign empty policy and value arrays
    value_mat = zeros(N)
    pol_func = zeros(N)
    #loop over state variables n
```

```
    for n=1:N
        #start with the candidate choice being to just sell the remaining rod and the candidate max being the value of the remaining
        cand_pol = n
        cand_max = P[n]
        #loop over all rod lengths to cut off; since n is the default, it is not included here
        for i=1:n-1
            #value of choosing length i to cut off is the price of a rod of length i plus the continuation value at n-i
            val = P[i] + value_mat[n-i]
            #if we get a value higher than the candidate max (note: it is possible that there is a tie between two different policies - this doesn't really matter though since we are interested in maximizing the value)
            if val >= cand_max
                #update the max and argmax candidates
                cand_max = val
                cand_pol = i
            end
        end
        #find the value of having a rod of length n by choosing the above candidate
        value_mat[n] = cand_max
        pol_func[n] = cand_pol
    end
    #return the value and policy vectors
    value_mat, pol_func
end

function rod_solver(P::Vector{Int64})
    #start with price vector corresponding to rod of length N
    N = length(P)
```

```
    #initialize empty policy sequence
    pol_seq = []
    #fill in the value function and policy vectors
    val, pol = rod_bellman(N, P)
    #use while loop to iteratively find the policy choices required to
   achieve the max value
    while N >= 1
        #find the policy function for a rod of length N
        cur_pol = Int(pol[N])
        #add the current policy to the policy sequence
        append!(pol_seq, cur_pol)
        #decrease N by the amount of the current policy
        N = N - cur_pol
    end
    #return the overall value as well as the sequence required to attain
   it
    val[length(P)],pol_seq
end

value_n, cuts = rod_solver( [1,5,8,9,10,17,17,20])
value_n
>>22.0
cuts
>>[6,2]

value_n, cuts = rod_solver( [1,5,45,9,10,17,17,20])
value_n
>>95.0
cuts
>>[3,3,2]

```

□

Problem 3.

Solution: Code and output for problem 3 is included below:

```
#note: I assume item array is ordered by ascending weight
function knap_bellman(values::Vector{Int64}, weights::Vector{Int64}, C::Int64)
    N = length(values)
    #create empty value array
    v_mat = zeros(N+1, C+1)
    #create empty weight array associated with the values in the value array
    w_mat = zeros(N+1,C+1)
    #create policy array which attains the values in the value array
    p_mat = fill([], N+1, C+1)
    for c=1:C+1
        for (i,item) in enumerate(values)
            #for convenience, define the weight and value of item i
            w_i = weights[i]
            v_i = values[i]
            if w_i > c #if i is not feasible, ignore it and set arrays to previous choice
                v_mat[i+1,c] = v_mat[i, c]
                w_mat[i+1,c] = w_mat[i,c]
                w_mat[i+1, c] = w_mat[i, c]
            elseif w_i ==  c #if item i is exactly the weight limit
                if values[i] >= v_mat[i,c] #if i is better than the previous optimal choice
                    if weights[i] <= w_mat[i,c] #if i weighs less than the previous optimal choice, set values, weights, and policy so that i is chosen at weight c
                        p_mat = [i]
                        v_mat[i+1, c] = v_i
                        w_mat[i+1, c] = w_i
```

```
                end
            end
        else
            #start with i being the candidate optimal choice
            cand_max = v_i
            cand_weight = w_i
            cand_pol = [i]
            for j=1:i-1 #loop over possible choices of smaller subsets
                v_ij = v_i + v_mat[j+1, c-w_i] #the value of choosing i and the optimal choice of items (1,...,j) with weight at most c - w_i
                w_ij = w_i+ w_mat[j+1, c-w_i]#weight of the above
                #println(v_ij, " ", w_ij, " ", values[i], " ", v_mat[j+1, c-w_i+1], w_ij, c)
                if w_ij <= c
                    if v_ij >= cand_max #if the value of using i with the optimal bundle of items (1,...,j) of weight less than c-w_i is greater than the candidate maximum, set the candidates equal to this new combination
                        cand_max = v_ij
                        cand_weight = w_ij
                        cand_pol = p_mat[j+1, c-w_i]
                    end
                end
            end
            #set value and weight for optimal bundle of weight less than c which potentially includes up to item i
            v_mat[i+1, c] = cand_max
            w_mat[i+1, c] = cand_weight

            if cand_pol ==[i] #if we are sticking with policy i
                p_mat[i+1, c] = cand_pol #set the policy to i
            else #if we are combining i with an existing policy vector
                new_pol = copy(cand_pol)
```

```
                    append!(new_pol, i) #append to previous policy vector
                    p_mat[i+1, c] = new_pol #create entry for new policy
   vector
                end
            end
        end
    end
    #return value and optimal policy vector
    v_mat[N+1, C+1], p_mat[N+1, C+1]
end

#determine value and optimal policies
knap_bellman([4,3,8], [1,1,2], 3)
>> 12, [1,3]
knap_bellman([60,100,120], [10,20,30], 50)
>> 220, [2,3]
```

To see how this is a generalization of the problem in part 2, we note that if we set C = 8 and the weight of a rod of length x to be x, then it is exactly the same problem. □

Problem 4.

Solution: Suppose there is an infinitely-lived firm with discount factor β which faces linear inverse demand $P(q) = a - bq$ in each period, where q is the quantity supplied by the firm. The firm starts with marginal cost c_0 in period $t = 0$. Each period, the firm can choose to invest in cost-reducing research. Specifically, if their marginal cost is c_t in period t, investing x_t dollars in research in period t lowers their marginal cost in period $t+1$ to $c_{t+1} = f(c_t, x_t)$ for some function f which is increasing in c_t and decreasing in x_t. We specialize by using $f(c_t, x_t) = \alpha^{x_t} c_t$, where $\alpha \in (0, 1)$. The firm's profit in period t is given by

$$\pi(q_t, x_t) = (P(q_t) - f(c_{t-1}, x_{t-1}))q_t - x_t$$

However, we can consolidate this. If the firm wishes to transition from c_t to c_{t+1}, they must pay an adjustment cost of $\Phi(c_t, c_{t+1}) = f^{-1}(c_t, c_{t+1}) = \log_\alpha\left(\frac{c_{t+1}}{c_t}\right)$ when $c_{t+1} \geq c_t$ and $\Phi(c_t, c_{t+1}) = 0$ otherwise. Hence, the firm's period-t profit given current cost c_t and next period cost c_{t+1} becomes

$$\pi(q_t, c_t, c_{t+1}) = (P(q_t) - c_t)q_t - \Phi(c_t, c_{t+1})$$

The firm's problem is thus to choose $\{q_t, c_{t+1}\}_{t=0}^{\infty}$ to solve

$$\max_{\{q_t, c_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t \left[(P(q_t) - c_t)q_t - \Phi(c_t, c_{t+1})\right]$$

subject to the constraints that $q_t, c_{t+1} \geq 0 \ \forall t$ and $c_0 > 0$. Alternatively, we note that we can express the firm's problem in a recursive manner. The value function of the firm with current cost c is given by

$$V(c) = \max_{q, c'} \left[\pi(q, c, c') + \beta V(c')\right]$$

We focus first on the first order condition for q. The optimal quantity must solve

$$0 = a - 2bq - c \iff q = \frac{a - c}{2b}$$

This is independent of the choice of c'. Thus, a firm with cost c will optimally set $q(c) = \frac{a-c}{2b}$. We can rewrite the firm's Bellman equation as follows:

$$V(c) = \max_{c'} \left[\pi(q(c), c, c') + \beta V(c')\right]$$

We can see now that this is a fixed point of the contraction mapping $T : f \to Tf$ where

$$(Tf)(c) = \max_{c'} \left[\pi(q(c), c, c') + \beta f(c')\right]$$

Thus, we can find the value function through value function iteration. We will do this in Julia. □