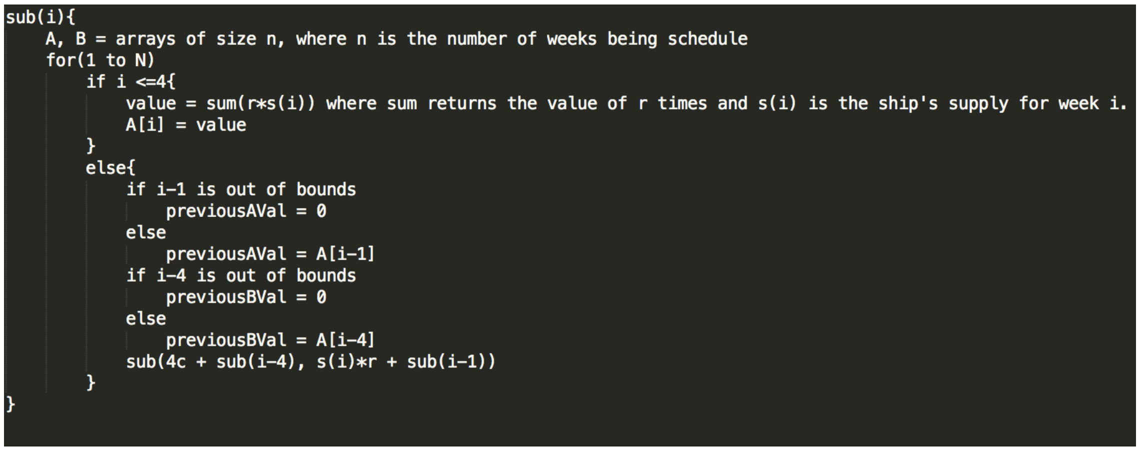
James Sweetman (jts2939)
Jung Yoon (jey283)
Unique #51835

HW #5

Q1

Let sub(i) return the optimal minimum cost of a solution for weeks 1 through i. In the solution, we either use company A or B for a week (the ith week). Company A costs rs_i and Company B costs 4c for week i. This means that Company A will work through week i-1 and Company B will work for i-4.
sub(i) = min(4c + sub(i-4), rs_i + sub(i-1))

In order to memoize this, we can build up a cache by calling the function in order starting from week 1 and increasing to week i and storing the values for each sub(i) into an array and just referencing it in constant time. We will have another array that stores the schedule. This can be done with the following algorithm:



```
sub(i){
    A, B = arrays of size n, where n is the number of weeks being schedule
    for(1 to N)
        if i <=4{
            value = sum(r*s(i)) where sum returns the value of r times and s(i) is the ship's supply for week i.
            A[i] = value
        }
        else{
            if i-1 is out of bounds
                previousAVal = 0
            else
                previousAVal = A[i-1]
            if i-4 is out of bounds
                previousBVal = 0
            else
                previousBVal = A[i-4]
            sub(4c + sub(i-4), s(i)*r + sub(i-1))
        }
}
```

Q2

a. **Minimum # of coins to make change given unlimited coins:**
   i. **Algorithm:**
      Suppose that we must find the minimum number of coins to make change for a value N and that we have an infinite supply of coins C = c1, c2,...,cm where each ci is a positive integer between 1 and N-1. Given this and everything stated in the question itself, one possible algorithm would be to use a variation of the algorithm already given in the hw link from geeksforgeeks.org. The only difference is that instead of adding together all the counts of different combinations, we would take

the minimum of all solutions for each subproblem and store the number of coins needed for the solution in a 2D array.

→ **Subproblems:** We would get a subproblem for each coin ci.
Further explained below in the **Recurrence/Memoize** section because we didn't want to say the same thing twice and moved it down there.

→ **Base case:** N = 0
We don't need to make any change for 0 so the minimum number of coins would be 0 as well. That is, given N = 0 we would return 0.

Also note that while this doesn't pertain to the base case,
If m = 0 but N > 0
Then the minimum number of coins would be 0 since there aren't any coins to make change with.

If N < 0 then there's no need to make change so minimum number of coins would be 0 again.

→ **Recurrence/Memoize:**
We would have a 1D array to store the denominations of coins. Then, we would use a 2D array as our table/data structure to store the solutions. D[1...m, 0...N], where each cell would tell us the minimum number of coins needed to produce N with *m* denominations of coins. This would mean that D[i, j] would be the minimum number of coins needed to make change for *j* using coins in the C = c1,c2,...,ci. In turn, the overall minimum number of joins for N would be given in cell D[m, N].

Then, we recursively fill the 2D array using the bottom-up approach. We know from the base case that when j = 0, D[i, j] is 0. This means that the entire column where j = 0 (the first column) should be filled with 0's.

1) We would consider combinations without ci such that D[i-1, j]. As such, we would reduce the amount of coin denominations available by 1. We would find the minimum number of coins to make change for every combination of denominations of coins excluding each and every c. So, m-c1, m-c2, etc.
2) We would consider the last coin in each combination of coins where the last coin has to be one of the possible C denominations. As such, we would reduce the

amount needed by N-ci. We would find the minimum number of coins to make change for N-c1, N-c2,…, N-cm.

Then, for N and m for a given set of coins C, we would need to select the solution with minimum number of coins:

That is, instead of this as given in the original solution,
`count( S, m - 1, n ) + count( S, m, n-S[m-1] );`
we would make the recurrence find the minimum by replacing it with
`min ( find_min( S, m - 1, n ), find_min( S, m, n-S[m-1]) );`
Such that the recursive structure might look something like this:

```
// Returns the count of ways we can sum  S[0...m-1] coins to get sum n
int find_min( int S[], int m, int n )
{
   // If n is 0 then there is 1 solution (do not include any coin)
   if (n == 0)
      return 1;
   // If n is less than 0 then no solution exists
   if (n < 0)
      return 0;
   // If there are no coins and n is greater than 0, then no solution exist
   if (m <=0 && n >= 1)
      return 0;
   return min ( find_min( S, m - 1, n ), find_min( S, m, n-S[m-1]) );
}
```

We would start computing in a bottom-up fashion and upon getting a returned minimum value, we would fill the 2D array until we finally got to filling the last cell on the last row, last column, which would give us our solution.

ii. **Correctness:**
Note how we are using only a variation of the proven algorithm given in the hw link (http://www.geeksforgeeks.org/dynamicprogramming-set-7-coin-change/). Given that the algorithm is proven, we know that it holds for the minimum case since all we did was change it so that we recursively find the minimum of each combination of denominators.

iii. **Runtime:**
The runtime in this algorithm would be O(mN) where m is the number of coin denominations and N is the amount that we want to make change for, as is reflected in the 2D array data structure. Note that this is not polynomial.

b. **Given limited coins:**

We need to keep a third array which happens to be 1D array of the the number of coins available for each denomination. We already have a 1D array to keep track of the different denominations so it would be like so →

To represent denominations:
C[m-1] = {c1, c2,...,cm}

To represent limit on coins: (THE NEW ARRAY)
L[m-1] = {#of_c1, #of_c2,...,#of_cm}

Then, using a modified version of the algorithm in part a, we'd find the minimum coin change solution with respect to the limit on the number of coins. More specifically, we would want to use the recurrence min (find_min(S, L-1, m - 1, n), find_min(S, L, m, n-S[m-1]));

c. One possible method is to have a descending array of denominations of coins (greatest to least) since the denominations are unique by the definition we were given. Then starting from the smallest coin we can traverse the array, subtracting each coin ci from N along the way. This traversal ensures a runtime of O(m) where m is the number of denominations.

Q3

You use the Bellman-Ford algorithm, with some modifications. A graph G can be made with the stocks as nodes and the ratios as directed edges for each pair of stocks. Instead of multiplying and analyzing large products, we will add the negative logarithms of each ratio. The problem states that the product of all r_{ij} in the cycle will be greater than 1 if it is an opportunity cycle. Taking the log of both sides and using the log properties, we know that this is equivalent to take the sum of the logs of r_{ij} and would be greater than 0. Bellman-Ford has been proven to be polynomial time, so this is a polynomial time algorithm that will find such a "trade cycle" as described in the question.