Intro to Algo HW #6

Problem 6.18

We define a sub-problem as follows. If we have the condition $a + b = c$, and we know b we need to figure out if we can make a. From a we need to solve it's own subproblem. This is the definition.

The sub problem S(n, V) returns the true if the nth value of the input array can be used to make a value V. If we run this from 0 to n we have our answer stored at Table[n][V]

The algorithm is as follows:

```
Store a table[n][V], this will store the coins
Take in the input of the array of coins which is size n
Define the Subproblem S(n, V)

for i to n:
    if S(x[i-1], V-x[i]):
        Table[i-1][V-x[i]] = {True, coin_solution_to_V()}
    if S(x[i-1], V):
        Table[i-1][V] = {True, coin_solution_to_V()}
    else:
        Table[i-1][V-x[i]] = {False, x[i-1]}

Return Table[n-1][V]
```

Analysis: We run the solution from 0 to n which is the size of the input array. At most we check 2V sub problems. Because we store the previous sub problems referencing them from the table is O(1). Thus the runtime is $O(nV)$

Problem 6.19

We define a sub problem S(v) that determines if a certain value is able to be reached based on the coins. We have the same $a + b = c$ relation from the previous question. Meaning we can construct the algorithm is roughly the same way without the item limitation, but a max size limitation. We need to take the smallest value each time.

The algorithm is as follows:

```
Input the array X of coins size n.
Create a Table[n][v]
Define the subproblem S(v)

for i to n:
    Table[i][v-x[i]] = min(S(v-x[i])) +1

if Table[n-1][v] <= k and Table[n-1][v] == True:
    return Table[n-1][v]
```

What happens in this algorithm is that S(v) is solving all the minimal subproblems for all values of v-x[i]. Because it is stored in the table S(v) can reference smaller sub problems stored in the table at a constant rate. At the end of the table the solution to S(v) is stored. It can return false. Because we take the minimal solution to S(v) each time we are assured to properly check against k.

Analysis: We solve n byvu subproblems. Taking the minimum and checking against k takes k amount of times. Storing it in the table means looking up smaller sub problems takes $O(1)$. Thus the algo is $O(n * k * v)$

Problem 7.18

a)

With many sources and sink we need to restructure the graph. We create a super vertex "SUPER-V" and a super sink "$SUPER_{sink}$". Now we have one source and one sink so we can use the linear programming method from the max flow problem.

The algorithm is as follows:

Input the multiple source and sink graph GG.
Define the linear solution for Max Flow problem M(G)

Convert the GG graph to a SUPER SINK/SOURCE graph A and return the SUPER-V and $SUPER_{sink}$ node.

Run M() on A.
M(A)

Return the maximum flow.

c)

The original linear programming method had the constraint of conserving the f_e or the flow across the edge, in addition to not breaking the capacity. We add another constraint l_e that is the lower bound. Thus we can use the same linear Max Flow problem with one source and one sink.

The algorithm is as follows:

Input the double constraint graph G
Define the linear solution for Max Flow problem M(G)

Convert the M() solution to account for the l_e constraint.
New constraint is $l_e \leq f_e \leq c_e$.

Run M() on G.
M(G)

Return the maximum flow.