Problem 1 (DPV 6.18)

First, we will implement a data structure (matrix) in order to define the subproblem. Initially, the matrix M[i][j] will contain Boolean values – false. Then, we change those Boolean values to true if desired sum can be obtained during the runtime of the algorithm. Define a matrix of size M[n][v].
The subproblem, similar to DPV 6.17 (coin change problem), can be defined as:

```
if( M[i-1][j-x_n] == true )
        M[i][j] = true;
else if ( M[i-1][j] == true )
        M[i][j] = true;
else
        M[i][j] = false;
```

And the recurrence relation can be written as:

$$M[i][j] = M[i-1, j] \ V \ M[i-1, j-x_i]$$

So basically, if $\mathbf{M[i-1][j-x_n]}$ == true, this means that it is still possible to come up with the new sum $\mathbf{j-x_n}$ with $x_1 \ldots x_{(i-1)}$ coins. The else if case, $\mathbf{M[i-1][j]}$ == true means that the new sum $\mathbf{j-x_n}$ can still be obtained without having x_i as its sum component. We look up the matrix and backtrack until we reach till the first coin, and

The runtime is going to be O(nv) where n is the number of denominations and v is the sum trying to be obtained. N is given because we subtract x_i from v **at most** n times, because we have $x_1 \ldots x_n$. V is given by how many times we will have to subtract x's from v. The **worst case** scenario would be that x_1 through x_n are all 1's, so we have to subtract x's from v for n times. Therefore O(nv) will be the worst case runtime.

Problem 2 (DPV 6.19) - your algorithm should run in time at most O(nvk) to receive full credit.

First, we initialize a matrix (2D array) a[i][j] of Boolean values. i represents the desired sum (v_i) that we are trying to obtain after subtracting different x's, and i will be at maximum v. j represents how many coins are used, and j will be at max k because we can use up to k number of coins.
The matrix has a size a[v+1][k+1]. The first column will initially have all *false*s, and the first row will have *true*s. If we get to any of the *true* values on the top row, then it means that, within k subtractions, we can obtain the desired sum v.
The recurrence relation is given as:

$$a[\mathbf{i}][\mathbf{j}] = a[\mathbf{i}-\mathbf{x_j}][\mathbf{j}-\mathbf{1}] \qquad \text{if } (\mathbf{i}-\mathbf{x_j}) >= 0$$

And the algorithm is as follows:

```
for i = v … 1
        for j = 1 … k
                for m = 1 … n
                        if (if (i-xj) >= 0)
                        a[i][j] = a[i-xm][j-1]
return a[v][k]
```

This algorithm has obviously O(*nvk*) because the three for-loops iterates *n*v*k* times.
The algorithm starts from a[1][1]. For any coin x_m that is smaller than the currently-desired sum i (initially it's at v), we take i-x_m and then store them in the next column with its appropriate row, that is a[i-x_m][j-1]. At each moment during the iteration, i = x_{m1} + x_{m2} + … where m*i*'s are some arbitrary combination of coins. We continue this iteration until we get to the last column then we check if any of the computational branches was keep returning true, which will be stored at a[v][k].

Problem 3 (DPV 7.18) part a & c – for part (a), show how to use the original maxflow problem to solve this variation; and for part (c), show how to use linear programming to solve this.

(a) First, let's label all the sources as s1, s2, s3… and all the sinks t1, t2, t3,…
Then, we introduce a new source node s whose children are all the previous sources s1, s2, s3... with no edge weights. So, s->s1, s->s2, s->s3… Also, we introduce a new sink node t whose parents are all the previous sinks t1, t2, t3… with no edge weights. t1-> t, t2-> t, t3-> t… Now, all we do is run the same algorithm and fine the max flow, but from s to t.

(c) -use linear programming

Previously, we had a capacity for each edge e as c. The capacity also has a lower bound, a non-negative constraint, so the bounds in whole look like 0 <= e <= c. This max flow problem can be solved with linear programming, and the methodology will be similar to the normal max flow problem. All we do is change the lower bound to be the minimum flow of each edge. The objective function to be maximized does not change how it works.