Jason Caplan
*Design and Analysis of Algorithms*
Professor Leff
*HW12*

1. Explanation of dynamic programming approach
   - We use the 2-d array with the following semantics: the $i^{th}$ row represents the $i^{th}$ denomination, the $j^{th}$ column represents the target amount of change to be made, and the value stored in $[i, j]$ represents the fewest number of coins needed to make the target amount of change $j$ using the first $1 \dots i$ denominations.
   - Furtherore, we can define our sub-problems as follows: using only the first $1 \dots i$ denominations, the way to make change for a target $j$ with the fewest number of coins will either be:
     - **CASE I** | The same as when using the first $i - 1$ denominations (if throwing another coin of this denomination into the mix does not produce a more optimal set of coins)

       *-or-*
     - **CASE II** | The best way to make change for a target that is $denominations[i]$ smaller than the current target, or $j - denominations[i]$, plus $1$ more coin of the $i^{th}$ denomination to reach the original target $j$
   - The two cases above can be compared, and the minimum chosen as the value for $[i, j]$
   - We also realize that the far-left of every row can be trivially filled in without even doing any comparisons
     - In detail, the first $denominations[i]$ elements of a given row will always fall under case I, since no change can be made for a target $x$ involving a coin of a value greater than $x$. Furthermore, the $denominations[i]^{th}$ element of each row will contain a 1, because the best change is simply using 1 coin of the given denomination to reach said target
2. Notation and Recurrence
   - Calling the 2-d array $C$, using the semantics for $i$ and $j$ described above, and using $S^*$ for the optimal set of coins for $[i, j]$, we have:

$$C[i,j] = \begin{cases} C[i-1,j] & i \notin S^* \\ C\big[i, j - denominations[i]\big] + 1 & i \in S^* \\ C[i-1,j] & j < denominations[i] \\ 1 & j = denominations[i] \\ \infty & C[i-1,j] = 0 \text{ or } \infty \end{cases} \quad \text{Base Cases}$$

   - Or more concretely, since $S^*$ is defined by whichever set of coins has the smallest size, (ignoring base cases for simplicity):

$$C[i,j] = \min(C[i-1,j], C\big[i, j - denominations[i]\big] + 1)$$

3. Payout Method and "Big-O" Space/Computation
   - The payout method "retraces our steps" to find the coins used, as follows:
     - Start at "bottom right" of $C$, or $C[denominations.length - 1, N]$
     - Until we get to the "top left" of $C$, or $C[0,0]$, we move along the traceback path with the following rules:

        - If $C[i,j] == C[i,j-1]$, we know we did not use any more of the $i^{th}$ denomination, so we decrement $i$ to reflect as such
        - If $C[i,j] < C[i,j-1]$, we did utilize a coin of the $i^{th}$ denomination, so we increment $payout[i]$ and decrement $j$ by $denominations[i]$ to reflect as such
- Space
    - The only space utilized is that of the 2-d array, which is of size $n \times N$, where $n$ is the number of denominations, and $N$ is the target amount of change. Therefore we have $O(n \times N)$
- Time
    - Filling in the 2-d array we visit each element exactly once, and do $O(1)$ work each time, for a total of $O(n \times N)$
    - Retracing our steps can take a maximum of $N$ iterations, in the worst-case where we used $N$ coins to make change for $N$. This is dwarfed by the filling-in process