The dynamic programming algorithm for CMP (change making problem).

Problem description:

- a sequence of positive integers \( c_1, c_2, \ldots, c_n \) which represent coin denominations
- non-negative integer \( T \) representing a target amount
- assuming an unlimited supply of coins of each denomination
- Goal: try to find minimum amount number \( M \) of coins needed to form \( T \) exactly, or output \( \infty \) if no combination of coins of the given denomination.

For example, if \( n = 3 \) and \( c_1, c_2, c_3 = 5, 9, 13 \) and \( T = 19 \) then \( M = 3 \) (since \( 19=5+5+9 \)), and if \( T = 6 \), then \( M = \infty \).

Two solve the problem, we can use dynamic programming algorithm (remember that Greedy Algorithm won’t work for this problem).

1. The First Method: two dimensional array.
   Define array \( A(i, t) \) \((0 \leq i \leq n, 0 \leq t \leq T)\) as follows:
   \( A(i, t) \) is the minimum number of the coins needed to form \( t \) by using only \( c_1, c_2, \ldots, c_i \), or else \( A(i, t) = \infty \) if there is no combination of the coins \( c_1, c_2, \ldots, c_i \) sums to \( t \).

   So our solution to this problem given \( A(i, t) \) are filled is \( M = A(n, T) \).

   To fill \( A(i, t) \), we give out the following recurrence, and you can prove the correctness of the recurrence by using induction (Please try it by yourselves as an exercises).

\[
A(i, t) = \begin{cases} 
0 & i = 0, t = 0 \\
\infty & i = 0, t > 0 \\
\min_{k \in \{0, 1, \ldots, \lfloor \frac{t}{c_i} \rfloor\}} \left[k + A(i-1, t-k \times c_i)\right] & \text{otherwise}
\end{cases}
\]

The high level algorithm for efficiently outputting a set of coins summing to \( M \) if \( M < \infty \), using the filled-in array \( A(i, t) \).

Suppose we use global array \( \text{coin}[i] \) to enstore the coins of denomination \( c_1 \) we used that are sum up to \( M \).

```c
void main(){
input n, c_1, c_2, \ldots, c_n, T;
Initialize: for (int j=1, j<n+1, j++) coin[i]=0;
int M:=A(n, T);
output coin[1], \ldots, coin[n], M;
}
```
int A(int i, int t)
{
    if (i==0 && t==0) return 0;
    if (i==0 && t>0) return INF;  % INF is a predefined big number
    if (i>0 && t>0)
    {
        int min:= INF;
        int p:= 0;
        int m:= t div c_i;
        for (k=0,k<m+1,k++)
        {
            int x:=k+A(i-1, t-k*c_i);
            if (x<min) { min:=x; p:=k;}
        }
        coin[i]:=p;
        return min;
    }
}

2. The Second Method: one dimentional array.

Given \(c_1, \ldots, c_n\), array \(A(t)\) \((0 \leq t \leq T)\) as follows:
\(A(t)\) is the minimum number of the coins needed to form \(t\) by using \(c_1, c_2, \ldots, c_n\), or else \(A(t) = \infty\) if there is no combination of the coins \(c_1, c_2, \ldots, c_n\) sums to \(t\).

So our solution to this problem given \(A(t)\) are filled is \(M = A(T)\).

To fill \(A(t)\), we give out the following recurrence:

\[
A(t) = \begin{cases} 
0 & t = 0 \\
\infty & t < 0 \\
\min_{j \in \{0,1,\ldots,n\}}[1 + A(t - c_j)] & \text{otherwise}
\end{cases}
\]

The high level algorithm for efficiently outputting a set of coins summing to \(M\) if \(M < \infty\), using the filled-in array \(A(i,t)\).

Suppose we use global array \(\text{coin}[i]\) to enstore the coins of denomination \(c_1\) we used that are sum up to \(M\).

void main()
{
input n,c_1,c_2,\ldots,c_n,T;
Initialize: for (int j=1,j<n+1,j++) \text{coin}[i]=0;
int M:=A(T);
output coin[1],\ldots,\,coin[n],\,M;
}

int A(int t)
{
    if (t=0) return 0;
    if (t<0) return INF; % INF is a predefined big number
    if (t>0)
    {
        int min:= INF;
        int p:= 0;
        for (j=1,j<n+1,j++)
        {
            int x:=1+A(t-c_j);
            if (x<min) { min:=x; p:=j;}
        }
        coin[p]:=coin[p]+1;
        return min;
    }
}