DESIGN AND ANALYSIS OF ALGORITHMS
LAB 08: Dynamic Programming

I. Dynamic Programming Idea

1. "Programming" relates to planning/use of tables, rather than computer programming.
2. Solve smaller problems first, record solutions in a table; use solutions of smaller problems to get solutions for bigger problems.
3. Differs from Divide and Conquer in that it stores the solutions, and subproblems are "overlapping"

II. Programming Exercises

A. Requirements for all problems:

1. Implement a Dynamic Programming algorithm to solve problem in Python programming language
2. Implement a Divide-and-Conquer algorithm (in the form of a recursive algorithm) to solve problem in Python programming language
3. Generate different inputs of different size
4. Draw the running time of each program as a function of input size

B. Problems

1. Calculate Fibonacci Numbers

   Fibonacci Numbers are described as follows

   $F_1 = F_2 = 1.$

   $F_n = F_{n-1} + F_{n-2}$, for $n \geq 3.$

```
Def F(n):
    If n <= 1:
        Return 1
    Return F(n-1) + F(n-2)
```

   The Dynamic Programming algorithm to calculate Fibonacci Numbers is presented as follows

```
Fib(n):
If n = 1 or n = 2, then return 1
Else
    prevprev ← 1
    prev ← 1
    For i = 3 to n {
        f ← prev + prevprev
        prevprev ← prev
        prev ← f
    }
    Return f
End
```

2. Solve 0/1 knapsack problem

The 0/1 knapsack problem is described as follows

- Some objects $O_1, O_2, \ldots, O_n$
- Their weights $W_1, W_2, \ldots, W_n$ (assumed to be integral values)
- Their values $V_1, V_2, \ldots, V_n$
- Capacity C
- To find a set $S \subseteq \{1, 2, \ldots, n\}$ such that
  $\Sigma_{i \in S} W_i \leq C$ (capacity constraint)
  and
  $\Sigma_{i \in S} V_i$ is maximsed
- Note that here we cannot use fractional items! We either take the whole or nothing of each item.

Intuition of Dynamic programming

- Let $F(C, j)$ denote the maximum value one can obtain using capacity C such that objects chosen are only from $O_1, \ldots, O_j$.
- Then, $F(C, 0) = 0$.
- If $W_j \leq C$, then
  $F(C, j) = max(F(C, j-1), F(C - W_j, j-1) + V_j)$.
- If $W_j > C$, then $F(C, j) = F(C, j-1)$.

The Dynamic Programming algorithm to solve 0/1 knapsack problem is presented as follows

For $s = 0$ to C do
    $F(s, 0) = 0$
EndFor.
    For $s = 1$ to C do
        For $j = 1$ to n do
            If $W_j \leq s$, then
              $F(s, j) = max(F(s, j-1), F(s-W_j, j-1)+V_j)$;
            Else $F(s, j) = F(s, j-1)$ Endif
        EndFor
    EndFor

Complexity: $O(C * n)$

If we want to see which objects are chosen, apply the following algorithm

For $s = 0$ to C do $F(s, 0) = 0; Used(s, 0) = false$ EndFor
    For $s = 1$ to C do
        For $j = 1$ to n do
            If $W_j \leq s$, then
                $F(s, j) =$
                  $max(F(s, j-1), F(s-W_j, j-1)+V_j)$;
                If $F(s, j) = F(s, j-1)$, then
                  $Used(s, j) = false$
                Else $Used(s, j) = true$ Endif
            Else $F(s, j) = F(s, j-1)$; $Used(s, j) = false$
            Endif
        EndFor
    EndFor
$Left = C$
For $j = n$ down to 1 do {
    If $Used(Left, j) = true$, then
        Print("Pick item" j); $Left = Left - W_j$;
EndIf
EndFor

3. Solve coin changing problem
   The problem is described as follows

   - Given some denominations $d[1] > d[2] > \ldots > d[n] = 1$.
   - To find the minimal number of coins needed to make change for certain amount S.

   Intuition of Dynamic programming to solve the problem

- Let $C[i, j]$ denote the number of coins needed to obtain value j, when one is only allowed to use coins $d[i], d[i+1], \ldots, d[n]$.
  Then, $C[n, j] = j$, for $0 \leq j \leq S$.
- Computing: $C[i-1, j]$:
  - If we use at least one coin of denomination $d[i-1]$: $1 + C[i-1, j - d[i-1]]$
  - If we do not use any coins of denomination $d[i-1]$: $C[i, j]$
- Taking minimum of above, we get:
  $C[i-1, j] = min(1 + C[i-1, j - d[i-1]], C[i, j])$

The Dynamic Programming algorithm to solve the problem is presented as follows

CoinChange
For $j = 0$ to S do
    $C[n, j] = j$
Endfor
For $i = n$ down to 2 do
    For $j = 0$ to S do
        If $j \geq d[i-1]$, then
        $C[i-1, j] = min(1 + C[i-1, j - d[i-1]], C[i, j])$
        Else $C[i-1, j] = C[i, j]$
    EndFor
EndFor

Complexity: $\Theta(S * n)$

To give coins used:
For $j = 0$ to S do $C[n, j] \leftarrow j$; $used[n, j] \leftarrow true$ EndFor
For $i = n$ down to 2 do
    For $j = 0$ to S do
        If $j \geq d[i-1]$ and $1 + C[i-1, j - d[i-1]] < C[i, j]$, then
            $C[i-1, j] \leftarrow 1 + C[i-1, j - d[i-1]]$;
            $used[i-1, j] \leftarrow true$
        Else $C[i-1, j] \leftarrow C[i, j]$;
            $used[i-1, j] \leftarrow false$
    EndFor
EndFor

4. Calculate Binomial Coefficients.

The Binomial Coefficients problem is described as follows

- $C(n,k) = {}^nC_k = \binom{n}{k}$: number of ways to choose k out of n objects
- $C(n,k) = C(n-1,k-1) + C(n-1,k)$, for $n > k > 0$.
- $C(n,0) = 1 = C(n,n)$.
- So $C(n,k)$ can be computed using smaller problems.

	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1