Exercise Session 09

Exercise 1.
Recall the recursive algorithm Fib-Rec(n) which computes n-th Fibonacci number $F(n)$ in time $O(2^n)$ by means of a naive implementation the Fibonacci recurrence.

```
Fib-Rec(n)
if n < 2
      return 1
else
      return Fib-Rec(n − 1) + Fib-Rec(n − 2)
```

(a) Implement a top-down memoized version of the above procedure called Memoized-Fib(n).

(b) Perform the asymptotic analysis of the worst-case running-time of Memoized-Fib(n).

(c) Perform the asymptotic analysis of the space used by Memoized-Fib(n)

Exercise 2.
Given a sequence of elements $A = [a_1, a_2, \ldots, a_n]$ we say that $[a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ is a subsequence of A if and only if $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. For example, the following are valid subsequences of $A' = [4, 6, 6, 7, 6, 8, 1, 0, 9, 0, 15, 7, 10]$:

$$[6, 6, 6, 8, 15], \qquad [4, 6, 0, 15, 7, 10], \qquad [4, 6, 7, 10] .$$

Given a sequence of numbers $A[1\,..\,n]$, we are interested in finding an arbitrary nondecreasing subsequence of A of maximal length (a.k.a., longest nondecreasing subsequence).

(a) Argue why a brute-force enumeration of all subsequences leads to an exponential algorithm.

(b) Let $L(i)$ be the maximal length of a nondecreasing subsequence of $A[1\,..\,i]$ that contains the element $A[i]$. Describe a recurrence defining $L(i)$ for $i = 1\,..\,n$.

(c) Note that $L = \max_{i=1..n} L_i$ is the length of a longest nondecreasing subsequence of A. Describe a bottom-up dynamic programming procedure BottomUp-LNDS(A) that computes L.

(d) Describe a procedure Print-LNDS(A) that prints an arbitrary longest nondecreasing subsequence of A.

Remark: the longest subsequence may not be unique. For instance, the length of the longest non decreasing subsequence for A' is 7 as witnessed by the subsequences $[4, 6, 6, 7, 8, 9, 15]$ and $[4, 6, 6, 7, 8, 9, 10]$.

Exercise 3.

The *knapsack problem* is a classic problem in combinatorial optimisation. The problem description goes as follows: one hiker is preparing for a long trip and has to fill up his bag with useful items. He can choose among n items to bring. Each item $i = 1..n$, weights $w_i > 0$ and has a utility value of $v_i \geq 0$. Overall the bag cannot lift more than $W > 0$. Given this information, the hiker wants to find a selection $S \subseteq \{1, \ldots, n\}$ of items that maximises the total value $\sum_{i \in S} v_i$ subject to the overall weight constraint $\sum_{i \in S} w_i \leq W$.
Example: Consider a problem instance with $w = [10, 20, 29]$, $v = [47, 125, 138]$ and $W = 50$. An optimal solution is the set of indices $S = \{2, 3\}$ with value $125+138 = 263$ and weight $20+29 = 49 \leq 50$.

For $i \in \{1..n\}$ and $j \geq 0$, we denote by $V(i, j)$ the optimal value of a selection S among the first i items, i.e., $S \subseteq \{1, \ldots, i\}$ having total weight at most j. $V(i, j)$ can be recursively defined as follows

$$V(i,j) = \begin{cases} 0 & \text{if } j = 0 \text{ or } i = 0 \\ V(i-1, j) & \text{if } w_i > j \\ \max\big(v_i + V(i-1, j-w_i), V(i-1, j)\big) & \text{otherwise} \end{cases}$$

Intuitively, the optimal value of an empty selection ($i = 0$) or a selection of weight at most 0 ($j = 0$) is 0. If the i-th item has a weight that exceeds the maximum weight allowed ($w_i > j$), then the i-th item cannot be part of the selection, hence the optimal value for $V(i, j)$ corresponds the value of an optimal selection among the first $i - 1$ elements. Otherwise $w_i \leq j$, thus the optimal selection can be made either keeping the i-th item in the selection, or discarding it. In the former case the total value is $v_i + V(i-1, j-w_i)$, in the latter case $V(i-1, j)$.

(a) Describe a bottom-up dynamic programming procedure KNAPSACK(v, w, W) that takes as input $v[1..n]$, $w[1..n]$, and $W > 0$ and returns the value of an optimal selection of items. Analyse the worst-case running time of your algorithm.

(b) Describe a procedure PRINT-KNAPSACK(v, w, W) that prints the optimal selection of items.

Exercise 4.

Peter works in Legoland where he is responsible of assembling buildings made of Lego blocks. He noticed that most of his work consists in repeating the following task: assemble a line of given length using Lego blocks. Given an unlimited supply of Lego blocks of given sizes, Peter wants to find the minimum amount of lego blocks required to form a line of length L.
For example, if each Lego block has one of following sizes $S = [3, 5, 7]$, the minimum amount of blocks required to form a line of length $L = 15$ is 3. Indeed, this can be achieved as $(5+5+5)$ or $(3+5+7)$. Sometimes, some lines cannot be formed using only the available blocks, e.g. there is no way to make a line of length $L = 4$.
Let's denote with $P(L, S)$ the minimum amount of blocks required to form a line of length L using Lego blocks of sizes in $S[1..n]$. Then, $P(L, S)$ can be recursively defined as follows.

$$P(L,S) = \begin{cases} \infty & \text{if } L < 0 \\ 0 & \text{if } L = 0 \\ 1 + \min_{i=1..n} P(L - S[i], S) & \text{if } L > 0 \end{cases}$$

Describe a bottom-up dynamic programming algorithm that computes $P(L, S)$ given $L \geq 0$ and an array S of block sizes. Analyse the worst-case running time of your algorithm.