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($n$)
1  if $n < 2$
2      return 1
3  else
4      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)

Solution 1.

(a) A top-down memoized implementation of the Fibonacci recurrence is

```
MEMOIZED-FIB($n$)
1  let $F[0..n]$ be a new array
2  $F[0] = F[1] = 1$
3  for $i = 2$ to $n$
4      $F[i] = 0$
5  return MEMOIZED-FIB-AUX($n, F$)
```

```
MEMOIZED-FIB-AUX($n, F$)
1  if $F[n] > 0$
2      return $F[n]$
3  $F[n]$ = MEMOIZED-FIB-AUX($n - 1, F$) + MEMOIZED-FIB-AUX($n - 2, F$)
4  return $F[n]$
```

(b) For the worst-case running time of MEMOIZED-FIB(n), we perform an aggregate analysis. Each call to a subproblem is computed at most once because every time one subproblem is called again, the procedure returns immediately the value stored in the array F. Since all other operations take constant time, overall the running-time of MEMOIZED-FIB(n) grows linearly in the number of subproblems, that is $\Theta(n)$.

(c) The algorithm employs an array F of length n to store the values of the Fibonacci sequence, thus the space requirement to run MEMOIZED-FIB(n) is $O(n)$.

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 $[4, 6, 6, 7, 8, 9, 10]$.

Solution 2.

(a) Any subsequence of $A[1\,..\,n]$ can be encoded by an n-digit binary number, where the i-th digit equals 1 if a_i belong to the sequence, and 0 otherwise. Thus the number of subsequences of A corresponds to the number of n-digit binary number, i.e., 2^n. Therefore a naive enumeration of all the sequences would yield an exponential algorithm.

(b) Observe that in a nondecreasing subsequence the last element is the greatest. Then for $1 \leq i \leq n$, $L(i)$ can be recursively characterised as follows

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max\{L(j) \colon 1 \leq j < i \text{ and } A[j] \leq A[i]\} & \text{if } i > 1 \end{cases}$$

Trivially, the longest nondecreasing subsequence of $A[1\,..\,1]$ is $[A[1]]$. For $i > 1$, we have that the a longest nondecreasing subsequence of $A[1\,..\,i]$ that contains $A[i]$ must have $A[i]$ as last element, thus its prefix can be found as the maximal among the longest subsequences of $A[1\,..\,j]$ containing $A[j]$ such that $1 \leq j < i$ and $A[j] \leq A[i]$.

(c) The above recurrence for $L(i)$ exhibits a number of overlapping subproblems, therefore dynamic programming is a suitable option for saving computational time at the expense of additional memory usage. A bottom-up dynamic programming implementation of the computation of L is described below (for the moment you can ignore the parts in blue)

```
BottomUp-LNDS($A$)
$n = A.length$
initialise the array $L[1\,..\,n+1]$ with 0s
Initialise the array $S[1\,..\,n]$ with 0s
$L[1] = 1$
for $i = 2$ to $n$
    // Compute $q = \max\{L(j) \colon 1 \leq j < i$ and $A[j] \leq A[i]\}$
    $q = 0$
    for $j = 1$ to $i - 1$
        if $A[j] \leq A[i]$ and $L[j] \geq q$
            $q = L[j]$
            $S[i] = j$
    $L[i] = q + 1$
// Compute $l = \max_{i=1..n} L[i]$
$l = L[1]$
$m = 1$
for $i = 2$ to $n$
    if $l < L[i]$
        $l = L[i]$
        $m = i$
return $l$, $m$, $S$
```

The running time of BottomUp-LNDS(A) is $\Theta(n^2)$ where n is the length of the array A. The leading term in the asymptotic complexity of the problem is due to the two nested for-loops in lines 5–12.

(d) To print out an optimal nondecreasing subsequence, we need to keep track of the choice of the index made while finding an optimal solution of each subproblem. This is done in the parts of the code that are coloured in blue. The array S stores in $S[i]$ the index of the second last element of a longest nondecreasing subsequence of $A[1..i]$ that contains $A[i]$ (as last element), that is

$$S[i] = \arg\max\{L(j)\colon 1 \leq j < i \text{ and } A[j] \leq A[i]\}\,,$$

note that we use the convention that $S[i] = 0$ if the set $\{L(j)\colon 1 \leq j < i \text{ and } A[j] \leq A[i]\}$ is empty. Additionally, we return the value $m = \arg\max_{i=1..n} L(i)$ which corresponds to the index of the last element of a longest nondecreasing subsequence of $A[1\,..\,n]$.

Given this information we print a longest nondecreasing subsequence of $A[1\,..\,n]$ as follows.

```
Print-LNDS($A$)
1   $(l, m, S)$ = BottomUp-LNDS($A$)
2   let $B[1\,..\,l]$ be an array
3   // Store the optimal subsequence in $B$
4   $B[l] = A[m]$
5   for $i = l - 1$ downto 1
6       $B[i] = A[S[m]]$
7       $m = S[m]$
8   // Print the array $B$
9   for $i = 1$ to $l$
10      print $B[i]$
```

The above procedure first stores the optimal subsequence in the array B starting from the last element down to the first, then prints the content of the array B. This double pass allows us to print the subsequence in increasing order.

Alternatively, one can implement a recursive version of the above procedure which does not require to store the optimal subsequence in the local array B.

```
Print-LNDS($A$)
1   $(l, m, S)$ = BottomUp-LNDS($A$)
2   Print-LNDS-Aux($A, S, l, m$)
```

```
Print-LNDS-Aux($A, S, l, m$)
1   if $l > 0$
2       Print-LNDS-Aux($A, S, l - 1, S[m]$)
3       print $A[m]$
```

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.

Solution 3.

(a) Note that Knapsack(v, w, W) has to compute the value $V(n, W)$. Therefore we can exploit the above described recurrence to implement a bottom-up dynamic programming procedure which computes $V(n, W)$ (for the moment you can ignore the parts in blue).

```
Knapsack($v, w, W$)
 1  $n = v.length$
 2  initialise the matrix $V[0\,..\,n, 0\,..\,W]$ with 0s
 3  let be a boolean matrix $K[0\,..\,n, 0\,..\,W]$ initialised with FALSE
 4  for $i = 1$ to $n$
 5      for $j = 1$ to $W$
 6          if $w[i] > j$
 7              $V[i,j] = V[i-1,j]$
 8          else
 9              if $V[i-1,j] \geq v[i] + V[i-1, j-w[i]]$
10                  $V[i,j] = V[i-1,j]$
11              else
12                  $V[i,j] = v[i] + V[i-1, j-w[i]]$
13                  $K[i,j] =$ TRUE
14  return $V[n,W]$, $K$
```

The worst-case running time is $O(W \cdot n)$ where n is the size of the arrays v and w.

(b) To print out an optimal selection, we need to keep track of which elements we kept in the bag while computing the optimal value V. This is done in the parts coloured in blue. The matrix $K[0\,..\,n, 0\,..\,W]$ is filled in according to the following criterion $K[i, j] = \text{TRUE}$ if and only if the optimal selection among the first i items having at most weight j contains the element i.

```
PRINT-KNAPSACK($v, w, W$)
    $(V, K) = $ KNAPSACK($v, w, W$)
    $j = W$
    for $i = v.length$ downto 1
        if $K[i, j]$
            print $i$
            $j = j - w[i]$
```

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.

Solution 4.
We can compute $P(L, S)$ in sequence as follows: first we compute $P(1, S)$, then $P(2, S)$, and so on until $P(L, S)$. The following procedure iteratively computes $P(i, S)$ for $i \in \{0, .., L\}$ and returns the value $P(L, S)$. This is done by implementing the computation of the recursive characterisation for the function $P(L, S)$ in a bottom-up fashion, using an array $P[0\,..\,L]$ to to store the values $P[i] = P(i, S)$ for $i \in \{0, .., L\}$.

```
BOTTOM-UP-BUILDLINE($L, S$)
    create the array $P[0\,..\,L]$
    $P[0] = 0$
    for $j = 1$ to $L$
        // Compute $\min_{i=1..S.length} P(L - S[i], S)$
        $q = \infty$
        for $i = 1$ to $S.length$
            if $j - S[i] \geq 0$ and $q > P[j - S[i]]$
                $q = P[j - S[i]]$
        $P[j] = 1 + q$
    return $P[L]$
```

The overall running time of the above procedure is $\Theta(L \cdot |S|)$, since initialising P takes $\Theta(L)$, and the two nested for-loops overall take $\Theta(L \cdot |S|)$. This suffices to solve the question with full marks.

For those interested on how one can actually retrieve the optimal list of bricks, here is how you can do that by simply using another array to store the optimal choices. We modify the above solution to keep track of optimal brick choices in the array $B[1\,..\,L]$ as follows (in blue are the parts that differ from the previous pseudocode above)

```
BOTTOM-UP-BUILDLINE($L, S$)
1   create the arrays $P[0\,..\,L]$ and $B[1\,..\,L]$
2   $P[0] = 0$
3   for $j = 1$ to $L$
4       // Compute $\min_{i=1..S.length} P(L - S[i], S)$
5       $q = \infty$
6       for $i = 1$ to $S.length$
7           if $j - S[i] \geq 0$ and $q > P[j - S[i]]$
8               $q = P[j - S[i]]$
9               $B[j] = i$
10      $P[j] = 1 + q$
11  return $P[L]$ and $B$
```

Now, one use the array of choices B and print an optimal list of bricks forming a line of length L as follows.

```
PRINTBLOCKS($L, S$)
1   $(v, B) =$ BOTTOM-UP-BUILDLINE($L, S$)
2   if $v < \infty$ // Check if there is a solution
3       $j = L$ // Initialise length of the subproblem
4       while $j \neq 0$
5           print $S[B[j]]$ // Print chosen brick
6           $j = j - S[B[j]]$ // Update length of the subproblem
```

The worst-case running time of the PRINTBLOCKS algorithm is $\Theta(L \cdot |S|)$: the call to the subroutine BOTTOM-UP-BUILDLINE(L, S) takes $\Theta(L \cdot |S|)$ and the **while**-loop takes $\Theta(L)$.