This exercise sheet is concerned with dynamic programming. Without experience it can be very hard to apply dynamic programming directly. It may help to first design a recursive solution for understanding the problem, use memoization and then transform the solution into a dynamic program. An example of this transformation can be found in the example solution for programming exercise 1.4.

A complete description of a dynamic program always considers the following aspects (interesting also for the exam!):

1) **Definition of the DP Table**: What are the dimensions of the table? What is the meaning of each entry?

2) **Computation of an Entry**: How can an entry be computed from the values of other entries? Which entries do not depend on others?

3) **Calculation order**: In which order can entries be computed so that values needed for each entry have been determined in previous steps?

4) **Extracting the solution**: How can the final solution be extracted once the table has been filled?

The running time of a dynamic program is usually easy to calculate by multiplying the size of the table with the time required to compute each entry. Sometimes, however, the time to extract the solution dominates the time to compute the entries.

**Exercise 6.1  Mars mission.**

The rover *Curiosity* landed on Mars and is located at a starting position *S*. The goal is to move to a target position *Z*, and to collect rock samples that are as valuable as possible. To not use too much energy, the rover is only allowed to take a step to the east (right) and to the south (down). The value of each rock sample is stored in an \((m \times n)\) matrix, e.g.

<table>
<thead>
<tr>
<th></th>
<th>9</th>
<th>2</th>
<th>5</th>
<th>11</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>17</td>
<td>21</td>
<td>32</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

In the matrix above, an example of a south-east path from *S* to *Z* is shown where the value of the collected rock samples is maximum. This path can be described by enumerating the corresponding matrix positions: \((1, 1) \rightarrow (2, 1) \rightarrow (2, 2) \rightarrow (2, 3) \rightarrow (2, 4) \rightarrow \cdots \rightarrow (5, 6)\).

*Please turn over.*
Provide a *dynamic programming* algorithm that takes an \((m \times n)\) matrix \(A\) with \(A[1,1] = A[m,n] = 0\), and that computes a south-east path from \(S = (1,1)\) to \(Z = (m,n)\) where the value of the collected rock samples is maximal. Notice that we search for the path itself, and not just for the maximum value. Provide also the running time of your solution in dependency of \(m\) and \(n\).

**Exercise 6.2  Cooking.**

A student wants to cook fried potatoes, one portion today and one portion tomorrow. For this purpose he has \(n\) potatoes \(\{1, \ldots , n\}\) available that have an overall weight of \(G \in \mathbb{N}\) gram. The potato \(i\) weighs \(g_i \in \mathbb{N}\) gram. We want to partition *all* \(n\) potatoes into the portions \(A\) and \(B\) such that these portions have roughly the same weight. Since the portions are cooked on different days, every potato must either be completely used for portion \(A\), or completely used for portion \(B\) (every potato must either be used immediately, or must be *completely* saved for tomorrow). More concretely we want to compute two sets \(A\) and \(B\) of potatoes with \(A \cap B = \emptyset\), \(A \cup B = \{1, \ldots , n\}\) whose weight difference is minimum.

Provide a dynamic programming algorithm for computing two such sets \(A\) and \(B\) with minimum weight difference (as described above). Provide the running time of your algorithm. Is the running time polynomial?

**Hand-in:** until Wednesday, 1st April 2015.