1. This question is concerned with the max subarray problem.  
Apply the dynamic programming algorithm to the array \([-3, 10, -8, 9]\).  
Solution:  
We apply the dynamic programming algorithm by Kadane on the array \([-3, 10, -8, 9]\).  
First step:  
\begin{align*}
B[1] &= \max(-3, 0) = 0 \text{ and } m = 0. \\
\end{align*}

So we get the following array \(B\):  
\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
0 & 10 & 2 & 11 \\
\end{array}
\]

The algorithm returns 11. The subarray yielding this value is \([10, -8, 9]\).

2. Show that the naive algorithm for the max-subarray problem is in \(O(n^2)\),  
and that the dynamic programming algorithm for the max-subarray problem is in \(O(n)\).  
Solution:  
The function for the time complexity of the naive algorithm is determined by the summation \(\sum_{i=1}^{n}(n - i + 1) = n + (n - 1) + \ldots + 1 = \frac{1}{2}n(n + 1)\).  
Hence this algorithm is in \(O(n^2)\).  
The function for the time complexity of the dynamic programming algorithm is determined by the summation \(\sum_{r=1}^{n-1} 1 = n - 1\). Hence this algorithm is in \(O(n)\).

3. This question is concerned with rod cutting.  
Apply the dynamic programming algorithm to the price table from the slides with \(n = 3\).  
Solution:  
We apply the dynamic programming algorithm, using the situation from the slides with \(n = 3\). The list of prices:

<table>
<thead>
<tr>
<th>length</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Applying the algorithm yields an array $b$ where at index $i$ we have the best price for a rod of length $i$.

The first step is the assignment $b[0] = 0$. Next the value for $b[j]$ is calculated in the outermost for-loop.

\[
\begin{align*}
  j &= 1 & i &= 1 & p[1] + b[0] &= 1 + 0 = 1 & b[1] := 1 \\
  j &= 2 & i &= 1 & p[1] + b[1] &= 1 + 1 = 2 \\
\end{align*}
\]

So the array $b$ takes the following form:

\[
\begin{array}{l|l|l|l|l}
  0 & 1 & 2 & 3 \\
  0 & 1 & 5 & 8 \\
\end{array}
\]


4. The recurrence equation for the recursive algorithm for rod cutting is given by $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$ together with $T(0) = 1$. Use induction to show that $T(n) = 2^n$.

Solution:

The base step: $T(0) = 1 = 2^0$.

The induction step: we assume that $T(k) = 2^k$. We have to show that $T(k + 1) = 2^{k+1}$.

\[
\begin{align*}
  T(k + 1) &= 1 + \sum_{j=0}^{k} T(j) \\
           &= 1 + \sum_{j=0}^{k} T(j) + T(k) \\
           &= 2 \cdot T(k) \\
           &= 2 \cdot 2^k \\
           &= 2^{k+1}
\end{align*}
\]

In the one-but-last step the induction hypothesis is used.

5. This question is concerned with rod cutting.

We adapt the problem: now there is a fixed cost of $c$ per cut. The benefit is the sum of the values of the rods, minus the costs for cutting. Apply the dynamic programming algorithm to deal with this situation.

Solution:

In the following $p$ is an array containing prices for rods of length 1, \ldots, $n$ and $c$ is the fixed cost per cut.
**Algorithm** `rodCuttingCost(p, n, c)`:

```plaintext
new array r[0...n]

r[0] := 0
for j = 1 to n do
    q := p[j]
    for i = 1 to j do
        if q < p[i] + r[j - i] - c then
            q := p[i] + r[j - i] - c
    r[j] = q

return r[n]
```

Note the assignment `q := p[j]`. We can do instead `for i = 1 to j - 1 do` because for the case that `i = j` we perform the test `q < p[i] + r[j - i] - c` which always fails because `p[j] ≤ q < p[j] + 0 - c`.

Example of applying the algorithm:

6. This question is concerned with rod cutting.

   Give an alternative algorithm for rod cutting that works top-down but reuses sub-results.

   Solution:

   Here is the algorithm from the book. It works top-down but reuses partial results. The input is an array `p[1...n]` of prices and a length `n`.

   **Algorithm** `rodCuttingMemo(p, n)`

   ```plaintext
   new array r[0...n]
   for i := 1 to n do
       r[i] := -∞
   return rodCuttingAux(p, n, r)
   ```

   The following auxiliary procedure is used:
Algorithm rodCuttingAux\((p, n, r)\)

\[
\begin{align*}
\text{if } r[n] & \geq 0 \text{ then} \\
& \quad \text{return } r[n] \\
\text{if } n & = 0 \text{ then} \\
& \quad q := 0 \\
\text{else} \\
& \quad q := -\infty \\
& \quad \text{for } i := 1 \text{ to } n \text{ do} \\
& \quad \quad q := \max(q, p[i] + \text{rodCuttingAux}(p, n - i, r)) \\
& \quad r[n] := q \\
\text{return } q
\end{align*}
\]

7. This question is concerned with knapsack01.

Apply the algorithm for knapsack01 to the following set \(S\).

<table>
<thead>
<tr>
<th>(b)</th>
<th>(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

with maximal total weight \(W = 5\).

Solution:

\[
\begin{array}{|cc|cccc|}
\hline
k & w & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 2 & 2 & 2 & 2 & 2 & 2 \\
2 & 0 & 2 & 5 & 7 & 7 & 7 & 7 \\
3 & 0 & 2 & 5 & 7 & 7 & 7 & 9 \\
\hline
\end{array}
\]

So the algorithm returns 9.

8. This question is concerned with knapsack10.

Apply the algorithm for knapsack01 with

\[
S = \{ (12, 4), (10, 6), (8, 5), (11, 7), (14, 3), (7, 1), (9, 6) \}
\]

and \(W = 18\).

Solution:

The set \(S\) contains:

\[
(12, 4), (10, 6), (8, 5), (11, 7), (14, 3), (7, 1), (9, 6)
\]
and we consider the elements in that order. So for example \( S_1 = \{(12, 4)\} \).

The next table gives \( B[k, w] \) for \( k = 0, \ldots, 5 \) and \( w = 0, \ldots, 18 \).

<table>
<thead>
<tr>
<th>( k \backslash w )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>20</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>20</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>34</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>26</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>26</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

9. This question is concerned with knapsack01.

Adapt the algorithm for knapsack01 such that it does not only give the best possible benefit, but also the subset of items yielding the best possible benefit.

Apply this adapted algorithm to the following set \( S \) of items \( s_i \) with benefit \( b_i \) and weight \( w_i \):

<table>
<thead>
<tr>
<th></th>
<th>( b )</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

with maximum total weight \( W = 8 \).

Solution:

We use as starting point the version of knapsack01 using a single array.

An adapted algorithm for knapsack01:

```plaintext
Algorithm knapsack01(S, W):
    for \( w := 0, \ldots, W \) do
        \( B[w] := 0 \)
        \( S[w] := \emptyset \)
    for \( k := 1, \ldots, n \) do
        for \( w := W, \ldots, w_k \) do
            if \( B[w - w_k] + b_k > B[w] \) then
                \( B[w] := B[w - w_k] + b_k \)
                \( S[w] := S[w - w_k] \cup \{s_k\} \)
```

We apply this algorithm on the following set \( S \) with items \( s_i \) with benefit \( b_i \) and weight \( w_i \):
Given a set $S$ for the knapsack01 problem with the following property: the order of the items if sorted according to increasing weight ($w$) is the same as the order of the items if sorted according to decreasing benefit ($b$). Example:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

10. This exercise is concerned with knapsack01.

Given a set $S$ for the knapsack01 problem with the following property: the order of the items if sorted according to increasing weight ($w$) is the same as the order of the items if sorted according to decreasing benefit ($b$). Example:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>

Give an efficient algorithm for this special case of knapsack01.

Solution:

Idea: We repeatedly select an item with the lowest weight (which then has a highest benefit), unless no more items can be added because of $W$. 

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
<td>∅</td>
</tr>
</tbody>
</table>