Analysis of Algorithms: Solutions 9

```
                            X
                   X        X
                   X        X
                   X        X
number of          X        X
homeworks          X        X
                X  X     X  X     X
                X  X     X  X     X
       X        X  X     X  X     X
       X  X     X  X     X  X     X
       X  X  X  X  X     X  X     X
       -----------------------------
       3  4  5  6  7  8  9  10 11 12
                 grades
```

Problem 1
Determine a longest common subsequence of $\langle 0, 1, 1, 0, 1, 1 \rangle$ and $\langle 1, 0, 1, 0, 0, 1, 0, 1 \rangle$, and draw the table constructed by the LCS-Length algorithm for these two sequences.

The longest common subsequences are $\langle 0, 1, 1, 0, 1 \rangle$, $\langle 0, 1, 0, 1, 1 \rangle$, and $\langle 1, 1, 0, 1, 1 \rangle$. The table computed by LCS-Length is as follows:

```
6 | 0 1 2 3 3 3 4 4 5
5 | 0 1 2 3 3 3 4 4 5
4 | 0 1 2 2 3 3 3 4 4
3 | 0 1 1 2 2 2 3 3 3
2 | 0 1 1 2 2 2 2 2 2
1 | 0 0 1 1 1 1 1 1 1
0 | 0 0 0 0 0 0 0 0 0
    -----------------
    0 1 2 3 4 5 6 7 8
```

Problem 2
Write an algorithm INCREASING-LENGTH(A, n) that determines the *length* of a longest increasing subsequence of an array $A[n]$.

Let $d[i]$ denote the length of a longest increasing subsequence whose last element is $A[i]$. For example, suppose that $A = \langle 1, 2, 1, 2, 3 \rangle$ and $i = 3$; then, the longest increasing subsequence that ends at $A[3]$ is $\langle 1, 1 \rangle$, which implies that $d[3] = 2$.

Observe that this subsequence contains at least one element, which means that $d[i] \geq 1$. Furthermore, if $j < i$ and $A[j] \leq A[i]$, then $d[i] \geq d[j] + 1$, because we may construct a $(d[j]+1)$-element increasing subsequence that ends at $A[i]$, by adding $A[i]$ to the longest increasing subsequence that ends at $A[j]$. These observations lead to the following algorithm, whose running time is $\Theta(n^2)$:

```
INCREASING-LENGTH(A, n)
d-max ← 0
for i ← 1 to n
    do d[i] ← 1
        for j ← 1 to i − 1
            do if A[i] ≥ A[j] and d[i] < d[j] + 1
                    then d[i] ← d[j] + 1
    if d-max < d[i]
        then d-max ← d[i]
return d-max
```

Problem 3
Suppose that you drive along some road, and you need to reach its end. Initially, you have a full tank, which holds enough gas to cover a certain distance d. The road has n gas stations, where you can refill your tank. The distances between gas stations are represented by an array $A[1..n]$, and the last gas station is located exactly at the end of the road. You wish to make as few stops as possible along the way. Give an algorithm CHOOSE-STOPS(d, A, n) that identifies all places where you have to refuel, and returns the set of selected gas stations.

```
CHOOSE-STOPS(d, A, n)
stations ← ∅   ▷ set of selected gas stations
d-left ← d   ▷ distance that corresponds to the remaining gas
for i ← 1 to n
    do if d-left < A[i]   ▷ cannot reach the next gas station? then refuel
          then stations ← stations ∪ {i − 1}
               d-left ← d
    d-left ← d-left − A[i]   ▷ drive to the next station
return stations
```

The algorithm runs in linear time, that is, its complexity is $\Theta(n)$.

Problem 4

Suppose that the weights of all items in the 0-1 Knapsack Problem are integers, and the weight limit W is also an integer. Design an algorithm that finds a *globally optimal* solution, and give its time complexity in terms of the number of items n and weight limit W.

We use dynamic programming with two arrays, $\mathit{item}[1..W]$ and $\mathit{value}[0..W]$, which are indexed on the size of a knapsack. For every size i between 0 and W, we compute the maximal value of items that can be loaded into a knapsack, and store this result in $\mathit{value}[i]$. If $\mathit{value}[i]$ is larger than $\mathit{value}[i-1]$, then $\mathit{item}[i]$ is the last added item; otherwise, $\mathit{item}[i]$ is 0.

We add items in their numerical order; that is, if items j_1 and j_2 must be in the knapsack, and $j_1 < j_2$, then we add j_1 before j_2.

The following algorithm computes the arrays $\mathit{item}[1..W]$ and $\mathit{value}[0..W]$, and returns the maximal value of items for size W; its time complexity is $\Theta(nW)$.

```
DYNAMIC-KNAPSACK(W, v, w, n)
value[0] ← 0
for i ← 1 to W    ▷ consider every size of a knapsack
    do item[i] ← 0
       value[i] ← value[i − 1]    ▷ initialize the maximal value for size i
       for j ← 1 to n    ▷ look through items, to find the best addition to a smaller load
           do if w[j] ≤ i    ▷ item j fits into the knapsack
                    and j > item[i − w[j]]    ▷ it does not violate the numerical order
                    and value[i] < value[i − w[j]] + v[j]    ▷ we get a good value by adding j
                 then item[i] ← j    ▷ add j to the knapsack
                      value[i] ← value[i − w[j]] + v[j]
return value[W]
```

We also need an algorithm for printing out the list of selected items. The following output procedure uses the array $\mathit{item}[1..W]$, built by DYNAMIC-KNAPSACK, to print items in their numerical order; its running time is $O(n)$.

```
PRINT-KNAPSACK(item, W, w, i)
if i = 0
   then "do nothing"
elseif item[i] = 0
   then PRINT-KNAPSACK(item, W, w, i − 1)
else PRINT-KNAPSACK(item, W, w, i − w[item[i]])
     print item[i]
```