Exercise sheet on dynamic programming for course 30540, 2023/24

Problem 1. You are given a sequence of n integers $a_1, a_2, \ldots, a_n$. You can take any subset of them, as long as you do not take any three consecutive elements. What is the largest sum of elements you can take?

Required running time: $O(n)$.

Solution. For $i \in \{0, 1, \ldots, n\}$, let d_i denote the largest sum you can take out of the first i elements, i.e., $a_1, \ldots, a_i$. We have $d_0 = 0$, $d_1 = \max(a_1, 0)$, $d_2 = \max(a_1, 0) + \max(a_2, 0)$. For $i \geqslant 3$, an optimal solution for i elements must either

a) leave the last element and use an optimal solution for $i-1$ elements, i.e., $d_i = d_{i-1}$, or
b) take the last element but leave the second to last one and use an optimal solution for the first $i-2$ element, and then $d_i = d_{i-2} + a_i$, or
c) take the last two items, leave the third to last, and use an optimal solution for the first $i-3$ items ($d_i = d_{i-3} + a_{i-1} + a_i$).

It follows that

$$d_i = \max\big(d_{i-1}, d_{i-2} + a_i, d_{i-3} + a_{i-1} + a_i\big).$$

We can turn this observation into an algorithm:

$d \leftarrow$ array of length $n+1$
$d[0] \leftarrow 0$
$d[1] \leftarrow \max(a_1, 0)$
$d[2] \leftarrow \max(a_1, 0) + \max(a_2, 0)$
for $i = 3, \ldots, n$ **do**
    $d[i] \leftarrow \max\big(d[i-1], \max(d[i-2] + a_i, d[i-3] + a_{i-1} + a_i)\big)$
return $d[n]$

Problem 2. Solve the knapsack problem in $O(nV)$ time, where $V = v_1 + v_2 + \cdots + v_n$ denotes the sum of item values. Note that the running time cannot depend on the item weights nor the knapsack capacity.

Problem 3. Given a sequence of length n, find its longest subsequence (of not necessarily consecutive elements) that is a palindrome.

Required running time: $O(n^2)$

Solution. Let $s[0], s[1], \ldots, s[n-1]$ be the input sequence. For any i and j such that $0 \leqslant i \leqslant j \leqslant n-1$, we denote by $d[i][j]$ the length of a longest palindromic subsequence of $s[i], s[i+1], \ldots, s[j]$. It is easy to prove that if $s[i] = s[j]$, then $d[i][j] = 2 + d[i+1][j-1]$, and otherwise $d[i][j] = \max(d[i+1][j], d[i][j-1])$. This can be turned into an algorithm as follows:

```
1  d ← two-dimensional array of size n × n
2  for i = 0, ..., n−1 do
3  |  d[i][i] ← 1
4  for i = 0, ..., n−2 do
5  |  if s[i] = s[i+1] then
6  |  |  d[i][i+1] ← 2
7  |  else
8  |  |  d[i][i+1] ← 1
9  for i = n−3, n−4, ..., 0 do
10 |  for j = i+2, i+3, ... n−1 do
11 |  |  if s[i] = s[j] then
12 |  |  |  d[i][j] ← 2 + d[i+1][j−1]
13 |  |  else
14 |  |  |  d[i][j] ← max(d[i+1][j], d[i][j−1])
15 return d[0][n−1]
```

Note that the outermost for-loop in line 9 goes backwards – it ensures that in lines 12 and 14 we are referencing subproblems that have already been computed.

The above algorithm computes only the length of an optimal subsequence, not the subsequence itself. To compute the subsequence we can follow an iterative or a recursive pattern similar to what appears, e.g., in algorithms for the knapsack problem or LCS.

Problem 4. Each field of a square $n \times n$ board either contains a single diamond or it is empty. We start in the top left corner of the board, and take steps either down or to the right, until we reach the bottom right corner. Compute the largest number of diamonds we can collect along the way.

Required running time: $O(n^2)$

Problem 5. Alice and Bob play the following game on a directed acyclic graph with n nodes and m edges. At the beginning, they select uniformly at random one node in the graph and put there a single token. Then, they take turns, starting from Alice. Each turn consists of moving the token along one of the edges leaving the node currently occupied by the token. The player who cannot make a move loses. Assuming both Alice and Bob use an optimal strategy, compute the probability that Alice wins.

Required running time: $O(n + m)$.