Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
François Hublet, Goran Zuzic, Tommaso d'Orsi, Jingqiu Ding

31 October 2022

Algorithms & Data Structures — Exercise sheet 6 — HS 22

The solutions for this sheet are submitted at the beginning of the exercise class on 07 November 2022.

Exercises that are marked by * are "challenge exercises". They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 6.1 *Longest ascending subsequence.*

The longest ascending subsequence problem is concerned with finding a longest subsequence of a given array A of length n such that the subsequence is sorted in ascending order. The subsequence does not have to be contiguous and it may not be unique. For example if $A = [1, 5, 4, 2, 8]$, a longest ascending subsequence is $1, 5, 8$. Other solutions are $1, 4, 8$, and $1, 2, 8$.

Given is the array:

$$[19, 3, 7, 1, 4, 15, 18, 16, 14, 6, 5, 10, 12, 19, 13, 17, 20, 8, 14, 11]$$

Use the dynamic programming algorithm from section 3.2. of the script to find the length of a longest ascending subsequence and the subsequence itself. Provide the intermediate steps, i.e., DP-table updates, of your computation.

Solution:

The solution is given by a one-dimensional DP table that we update in each round. After round i, the entry $DP[j]$ contains the smallest possible endvalue for an ascending sequence of length j that only uses the first i entries of the array. In each round, we need to update exactly one entry. If there is no ascending sequence of length j, we mark it by "-" . In order to visualise the algorithm, we display the table after each round. Note that the algorithm does not create a new array in each round, it just updates the single value that changes

length	1	2	3	4	5	6	7	8	9
round 1	19	-	-	-	-	-	-	-	-
round 2	3	-	-	-	-	-	-	-	-
round 3	3	7	-	-	-	-	-	-	-
round 4	1	7	-	-	-	-	-	-	-
round 5	1	4	-	-	-	-	-	-	-
round 6	1	4	15	-	-	-	-	-	-
round 7	1	4	15	18	-	-	-	-	-
round 8	1	4	15	16	-	-	-	-	-
round 9	1	4	14	16	-	-	-	-	-
round 10	1	4	6	16	-	-	-	-	-
round 11	1	4	5	16	-	-	-	-	-
round 12	1	4	5	10	-	-	-	-	-
round 13	1	4	5	10	12	-	-	-	-
round 14	1	4	5	10	12	19	-	-	-
round 15	1	4	5	10	12	13	-	-	-
round 16	1	4	5	10	12	13	17	-	-
round 17	1	4	5	10	12	13	17	20	-
round 18	1	4	5	8	12	13	17	20	-
round 19	1	4	5	8	12	13	14	20	-
round 20	1	4	5	8	11	13	14	20	-

The longest subsequence has length 8, since this is the largest length for which there is an entry in the table after the final round. To obtain the subsequence itself, we work backwards: The last entry is 20. To get the second-to-last value, we check out the left neighbor of 20 in the round in which 20 was entered (round 17), which is 17. Then we go the left neighbor of 17 in the round in which it entered the table (round 16), and obtain 13. Continuing in this fashion, we obtain the sequence $1, 4, 5, 10, 12, 13, 17, 20$.

Exercise 6.2 *Coin Conversion* **(1 point)**.

Suppose you live in a country where the transactions between people are carried out by exchanging coins denominated in dollars. The country uses coins with k different values, where the smallest coin has value of $b_1 = 1$ dollar, while other coins have values of $b_2, b_3, \ldots, b_k$ dollars. You received a bill for n dollars and want to pay it *exactly* using the smallest number of coins. Assuming you have an unlimited supply of each type of coin, define OPT to be the minimum number of coins you need to pay exactly n dollars. Your task is to calculate OPT. All values $n, k, b_1, \ldots, b_k$ are positive integers.

Example: $n = 17, k = 3$ and $b = [1, 9, 6]$, then $\text{OPT} = 4$ because 17 can be obtained via 4 coins as $1 + 1 + 9 + 6$. No way to obtain 17 with three or less coins exists. (A previous version had a typo "$k = 4$" that was corrected to "$k = 3$".)

(a) Consider the pseudocode of the following algorithm that "tries" to compute OPT.

Algorithm 1

Input: integers n, k and an array $b = [1 = b_1, b_2, b_3, \ldots, b_k]$.

$counter \leftarrow 0$
while $n > 0$ **do**
    Let $b[i]$ be the value of the largest coin $b[i]$ such that $b[i] \leq n$.
    $n \leftarrow n - b[i]$.
    $counter \leftarrow counter + 1$
Print("min. number of required coins = ", $counter$)

Algorithm 1 does not always produce the correct output. Show an example where the above algorithms fails, i.e., when the output does not match OPT. Specify what are the values of n, k, b, what is OPT and what does Algorithm 1 report.

Solution:

Set $n = 12$, $k = 3$, $b = [1, 9, 6]$ (this is the same example as above except $n = 12$). Algorithm 1 returns 4 as it finds the sequence of coins $[9, 1, 1, 1]$. The correct answer is $\text{OPT} = 2$ because $12 = 6 + 6$.

(b) Consider the pseudocode below. Provide an upper bound in O notation that bounds the time it takes a compute $f[n]$ (it should be given in terms of n and k). Give a short high-level explanation of your answer. For full points your upper bound should be tight (but you do not have to prove its tightness).

Algorithm 2

Input: integers n, k. Array $b = [1 = b_1, b_2, b_3, \ldots, b_k]$.

Let $f[0 \ldots n]$ be an array of integers.
$f[0] \leftarrow 0$ ▷ Terminating condition.
for $N \leftarrow 1 \ldots n$ **do**
    $f[N] \leftarrow \infty$ ▷ At first, we need ∞ coins. We try to improve upon that.
    for $i \leftarrow 1 \ldots k$ **do**
        if $b[i] \leq N$ **then**
            $val \leftarrow 1 + f[N - b[i]]$ ▷ Use coin $b[i]$, it remains to optimally pay $N - b[i]$.
            $f[N] \leftarrow \min(f[N], val)$
Print($f[n]$)

Solution:

In worst case, the algorithm completes in $O(n \cdot k)$ time. There are a total of n different states $f[1], \ldots, f[N]$, and computing the answer for each state takes $O(k)$ time (due to the inner **for** loop). Therefore, the total runtime is $O(n \cdot k)$.

(c) Let $\text{OPT}(N)$ be the answer (min. number of coins needed) when $n = N$. Algorithm 2 (correctly) computes a function $f[N]$ that is equal to $\text{OPT}(N)$. Formally prove why this is the case, i.e., why $f[N] = OPT(N)$.

Hint: *Use induction to prove the invariant $f[n] = OPT(n)$. Assume the claim holds for all values of $n \in \{1, 2, \ldots, N-1\}$. Then show the same holds for $n = N$.*

Solution:

We use induction. For the base of the induction, $f[0] = 0 = OPT(0)$ is trivially correct. Suppose that $f[n] = OPT(n)$ for all values of $n \in \{1, 2, \ldots, N-1\}$. It remains the prove the induction step: that $f[N]$ is also correct.

Suppose that the **optimal** way to obtain N is via $OPT(N) = T^*$ coins: $N = a_1 + a_2 + \ldots + a_{T^*}$ where $a_i \in \{b_1, \ldots, b_k\}$ for all i. Let x be the index such that $a_1 = b_x$. Then, in the inner **for** loop (lines 7–10), after the variable i becomes equal to x, we will have that $f[N] \leq 1 + f[N - b_x]$. However, by assumption, we have that $f[N - b_x] = OPT(N - b_x)$ is computed correctly, hence $OPT(N - b_x) \leq T^* - 1$ since $N = a_1 + a_2 + \ldots + a_{T^*}$ can be rewritten as $N - b_x = a_2 + \ldots + a_{T^*}$ (this uses $T^* - 1$ coins). Therefore, $f[N] \leq 1 + OPT(N - b_x) \leq 1 + T^* - 1 = T^* = OPT(N)$.

We shown $f[N] \leq OPT(N)$. It remains to argue that $f[N] \geq T^*$. Suppose the latter is not the case and consider the moment when $f[N]$ (i.e., its corresponding variable $solution$) got assigned a value less than T^*. At that moment, we have that $1 + f[N - b_i] < T^*$. Rewriting, we have that $f[N - b_i] < T^* - 1$. This means, by assumption and $N - b_i < N$, that there exists a way to pay $N - b_i$ using less than $T^* - 1$ coins. However, this implies that we can then pay N using less than T^* coins: simply pay for $N - b_i$ and then use an additional coin b_i. This contradicts the choice of T^*.

Hence, we proved that $f[N] = T^* = OPT(N)$. The claim follows by induction.

(d) Rewrite Algorithm 2 to be recursive and use memoization. The running time and correctness should not be affected.

Solution:

Algorithm 3

```
1:  Input: integers n, k. Array b = [1 = b_1, b_2, b_3, ..., b_k].
2:  Global variable: memo[1...n], initialized to −1.
3:
4:  function f(N)
5:      if N = 0 then return 0
6:      if memo[N] ≠ −1 then return memo[N]
7:      solution ← ∞
8:      for i ← 1...k do
9:          if b[i] ≤ N then
10:             val ← 1 + f(N − b[i])          ▷ Use coin b_i, it remains to optimally pay x − b_i.
11:             solution ← min(solution, val)   ▷ Check if this is the best seen so far?
12:     memo[N] ← solution
13:     return solution
14:
15: Print("OPT = ", f(n))
```

Exercise 6.3 *Longest common subsequence.*

Given are two arrays, A of length n, and B of length m, we want to find the their longest common subsequence and its length. The subsequence does not have to be contiguous. For example, if $A =$

$[1, 8, 5, 2, 3, 4]$ and $B = [8, 2, 5, 1, 9, 3]$, a longest common subsequence is $8, 5, 3$ and its length is 3. Notice that $8, 2, 3$ is another longest common subsequence.

Given are the two arrays:

$$A = [7, 6, 3, 2, 8, 4, 5, 1]$$

and

$$B = [3, 9, 10, 8, 7, 1, 2, 6, 4, 5],$$

Use the dynamic programming algorithm from Section 3.3 of the script to find the length of a longest common subsequence and the subsequence itself. Show all necessary tables and information you used to obtain the solution.

Solution:

As described in the lecture, $DP[i, j]$ denotes the size of the longest common subsequence between the strings $A[1 \dots i]$ and $B[1 \dots j]$. Note that we assume that A has indices between 1 and 8, so $A[1 \dots 0]$ is empty, and similarly for B. Then we get the following DP-table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

To find some longest common subsequence, we create an array S of length $DP[n, m]$ and then we start moving from cell (n, m) of the DP table in the following way:

If we are in cell (i, j) and $DP[i-1, j] = DP[i, j]$, we move to $DP[i-1, j]$.

Otherwise, if $DP[i, j-1] = DP[i, j]$, we move to $DP[i, j-1]$.

Otherwise, by definition of DP table, $DP[i-1, j-1] = DP[i, j] - 1$ and $A[i] = B[j]$, so we assign $S[DP[i, j]] \leftarrow A[i]$ and then we move to $DP[i-1, j-1]$.

We stop when $i = 0$ or $j = 0$.

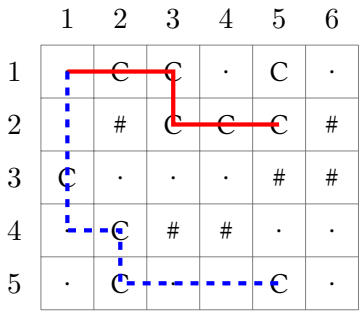
Using this procedure we find the following longest common subsequence: $S = [7, 6, 4, 5]$.

Exercise 6.4 *Coin Collection* **(2 points)**.

Suppose you are playing a video game where your character's goal is to collect as many coins in a two-dimensional $m \times n$ grid world (m rows by n columns). The world is given to you as a table $A[1 \dots m][1 \dots n]$ where each cell is either a coin (denoted as "C"), impassible (denoted as "#"), or passable without coins (denoted as ".").

Your character starts at $(1, 1)$ (this cell will always be passable) and, in each turn, can move either right or down (up to your choice), or stop whenever (ending the game). Moving right corresponds to moving from $(x, y) \to (x, y+1)$ and moving down is $(x, y) \to (x+1, y)$. The goal is to determine the maximum number of coins the player can collect (by moving into a cell).

For example, on the $m \times n = 5 \times 6$ grid depicted right, the player can collect 5 coins by following the solid-red path. This is maximum possible and the answer is 5. A suboptimal path is depicted in dashed-blue, yielding 4 coins.



Remark: Be careful not to peek into an element of the table that is out-of-bounds (i.e., not within $[1, m] \times [1, n]$*), as this can cause undefined behavior on a real computer.*

(a) Write the pseudocode of a **recursive** function $f(x, y)$ which takes as argument a position of the character (x, y), and outputs the maximum number of coins that the character can collect if it started at (x, y) (ignoring all coins it might have previously collected). For example, in the grid above, $f(1, 1) = 5$, $f(2, 1) = 4$, $f(5, 5) = 1$, $f(5, 6) = 0$. The function does not need to be memoized for this subtask.

Solution:

Algorithm 4

```
Input: integers m, n, grid A (seen as global read-only variables).

function f(x, y)
    coinHere ← 1 if A[x][y] = "C" and 0 otherwise
    ret ← coinHere
    if x + 1 ≤ m and A[x + 1][y] ≠ "#" then
        goDown ← coinHere + f(x + 1, y)
        ret ← max(ret, goDown)
    if y + 1 ≤ n and A[x][y + 1] ≠ "#" then
        goRight ← coinHere + f(x, y + 1)
        ret ← max(ret, goRight)
    return ret
```

(b) Prove that your algorithm terminates in finite time (even if possibly exponential in the size of the input). Prove that the algorithm is correct.

Hint: *(This hint is assuming you implemented part (a) in the most natural recursive way.) To prove the algorithm completes in finite time, observe that* $x + y$ *only increases and is bounded, hence no infinite execution paths exist.*

Hint: *To prove the algorithm is correct, we simply need to prove the invariant which describes* $f(x, y)$ *(i.e., the first sentence of part (a)). Assume, by induction, the invariant holds for recursive calls* $f(x, y)$ *with strictly larger values of* $x + y$*, i.e., for those* $f(x', y')$ *such that* $x' + y' > x + y$*. Argue that*

then it also holds for $f(x, y)$ — we do this by considering the optimal path P^* that starts at (x, y) and consider three cases: if P^* ends immediately, if P^* initially goes to the right, or it goes down. Using the inductive hypothesis, argue that in each of those cases $f(x, y)$ becomes a value at least as large as the number of coins collected on P^*. Similarly, by considering the three cases, argue that the final value cannot be larger than that of P^* since otherwise we could find a better P^*. This, by induction, establishes that $f(x, y)$ is always equal to the number of coins on P^*.

Solution:

Finite time. Clearly, the parameters (x, y) of the function f satisfy $x \in \{1, 2, \ldots, m\}$ and $y \in \{1, 2, \ldots, n\}$ since this condition is true when the function is first called and the function ensures it remains true upon subsequent calls. Furthermore, in each subsequent call of f, the value of $x+y$ (the sum of values of parameters) strictly increases; since $x + y$ is also bounded within the range $[2, m + n]$ we conclude that f will eventually terminate.

Correctness. In short: we check all possible paths. Using induction, we prove the invariant that $f(x, y)$ reports the largest number coins we can collect by starting at (x, y). Induction hypothesis: the invariant holds for calls $f(x, y)$ with strictly larger value of $x + y$. Induction step: let P^* be the optimal path starting at (x, y). If P^* stops immediately, clearly $f(x, y)$ is going to (correctly) return 0/1 based on whether there is a coin on (x, y), hence $f(x, y)$ will return the correct value. From now on, let us define with $val(P^*)$ the number of coins on P^*.

If P^* initially goes to the right, let P' be a suffix of P^* without the first cell (i.e., starting at $(x, y + 1)$) and let $coinHere$ be 1 if there is a coin at (x, y) and 0 otherwise. By construction, we have $val(P^*) = coinHere + val(P')$. Then, consider the call $goRight = coinHere + f(x, y + 1)$: by induction, $f(x, y + 1)$ reports a value at least as large as $val(P')$, hence

$$f(x, y) \geq goRight = coinHere + f(x, y + 1) \geq coinHere + val(P') = val(P^*).$$

Analogously, the same claim holds if P^* initially goes down.

We have proven $f(x, y) \geq val(P^*)$. We now prove $f(x, y)$ cannot exceed $val(P^*)$. For the sake of contradiction, suppose $f(x, y) > val(P^*)$. There are 3 cases: (1) if the final value of $f(x, y)$ was assigned in line 5 (of Algorithm 4). This is impossible, as then $f(x, y) = coinHere \leq val(P^*)$, a contradiction. Case (2): if the final value of $f(x, y)$ was assigned in line 11 (i.e., by going right), i.e., $f(x, y) = coinHere + f(x, y + 1)$. Then, let P' be the optimal path starting at $(x, y + 1)$. By the induction hypothesis, we have that $f(x, y + 1) = val(P')$. But then, we can construct a path starting at (x, y) that is better than P^*: simply prepend (x, y) to P' which gives a value of $coinHere + val(P') = f(x, y) > val(P^*)$. However, this contradicts the choice of P^*, resulting in a contradiction. Finally, case (3), when the final value of $f(x, y)$ was assigned in line 8 (i.e., going down), is completely analogous to case (2) and we are done. Hence, $f(x, y)$ cannot contain more coins than the **optimal** path P^*. Hence, we proven the claim.

(c) Rewrite the pseudocode of the subtask (a), but apply memoization to the above f. Prove that calling $f(1, 1)$ will, in the worst-case, complete in $O(m \cdot n)$ time.

Solution:

Algorithm 5 Differences with Algorithm 4 are marked in blue for convenience.

```
1:  Input: integers m, n, grid A (seen as global read-only variables).
2:  Global variable: memo[1...m][1...n], initialized to −1.
3:
4:  function f(x, y)
5:      if memo[x][y] ≠ −1 then return memo[x][y]
6:      coinHere ← 1 if A[x][y] = "C" and 0 otherwise
7:      ret ← coinHere
8:      if x + 1 ≤ m and A[x + 1][y] ≠ "#" then
9:          goDown ← coinHere + f(x + 1, y)
10:         ret ← max(ret, goDown)
11:     if y + 1 ≤ n and A[x][y + 1] ≠ "#" then
12:         goRight ← coinHere + f(x, y + 1)
13:         ret ← max(ret, goRight)
14:     memo[x][y] ← ret
15:     return ret
```

Any two calls to f when the arguments (x, y) have the same value (i.e., on two calls $f(x_1, y_1)$ and $f(x_2, y_2)$ where $x_1 = x_2$ and $y_1 = y_2$), at most one call can proceed beyond the first **if** statement — the second call will short-circuit due to memoization and exit immediately. Therefore, the number of times the function f proceeds beyond the first "memoization" if is $O(m \cdot n)$. In each such call, the number of operations excluding recursive calls is $O(1)$, hence we conclude that the total runtime is $O(m \cdot n)$.

(d) Write the pseudocode for an algorithm that computes the solution in $O(m \cdot n)$ time, but does not use any recursion. Address the following aspects of your solution:

(a) Definition of the DP table: What are the dimensions of the table DP? What is the meaning of each entry?

(b) Computation of an entry: How can an entry be computed from the values of other entries?

(c) Specify the base cases, i.e., the entries that do not depend on others.

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

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

(f) Running time: What is the running time of your solution?

(g) Explicitly write out the pseudocode.

Solution:

(a) $DP[1 \dots m][1 \dots n]$. The entry $DP[x][y]$ corresponds to the maximum number of coins that the character can collect if it started at (x, y) (ignoring all coins it might have previously collected)

(b) Each entry $DP[x][y]$ is equal to the maximum of three things: (1) whether there is a coin at (x, y) (0 if not, 1 if yes; this corresponds to the path stopping here), (2) whether there is a coin at (x, y) plus DP[x+1][y] (corresponds to continuing the path down), (3) whether there is a coin at (x, y) plus DP[x][y+1] (corresponds to continuing the path right).

(c) The base case is essentially case (1) from part (b) above: if the path stops at (x, y) we initialize $DP[x][y]$ with 0 if there is no coin at (x, y) and 1 if there is one.

(d) One way to compute the entries is bottom-to-top: from last to first row in the outer loop, then from last to first column in the inner loop.

(e) $DP[1][1]$ contains the final solution.

(f) Running time is $O(m{\cdot}n)$ since there are $O(m{\cdot}n)$ table entries, and each entry can be computed in $O(1)$ time.

(g) (see below)

Algorithm 6

```
1: Input: integers m, n, grid A (seen as global read-only variables).
2:
3: Define a table dp[1...m][1...n].
4: for x ← m downto 1 do
5:     for y ← n downto 1 do
6:         coinHere ← 1 if A[x][y] = "C" and 0 otherwise
7:         dp[x][y] ← coinHere
8:         if x + 1 ≤ m and A[x + 1][y] ≠ "#" then
9:             goDown ← coinHere + dp[x + 1][y]
10:            dp[x][y] ← max(dp[x][y], goDown)
11:        if y + 1 ≤ n and A[x][y + 1] ≠ "#" then
12:            goRight ← coinHere + dp[x][y + 1]
13:            dp[x][y] ← max(dp[x][y], goRight)
14: Print(dp[1][1])
```