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
Johannes Lengler, Gleb Novikov, Chris Wendler, Ulysse Schaller

26. October 2020

Algorithms & Data Structures Exercise sheet 6 HS 20

Exercise Class (Room & TA): ____________________

Submitted by: ____________________

Peer Feedback by: ____________________

Points: _____

Submission: On Monday, 2. November 2020, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 6.1 *Minimum edit distance: finding an invariant* (1 point).

Let $\Sigma = \{a, b, c, \dots, z\}$ denote the alphabet. The minimum edit distance is equal to the minimal amount of insertions, deletions and substitutions required to change the string α to the string β. E.g., `editDistance`((t,i,g,e,r), (z,i,e,g,e)) = 3. The following algorithm takes two strings $\alpha = (\alpha_1, \dots, \alpha_m) \in \Sigma^m$ and $\beta = (\beta_1, \dots, \beta_n) \in \Sigma^n$, with $m, n > 1$, as input and computes the minimum edit distance.

Algorithm 1 `editDistance`(α, β)

```
$d \leftarrow \mathbf{0}^{(m+1)\times(n+1)}$ an $(m+1) \times (n+1)$ matrix of zeros
for $i = 1, \dots, m$ do
    $d_{i,0} = i$
for $j = 1, \dots, n$ do
    $d_{0,j} = j$
for $i = 1, \dots, m$ do
    for $j = 1, \dots, n$ do
        if $\alpha_i = \beta_j$ then
            $c = 0$
        else
            $c = 1$
        $d_{i,j} = \min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + c)$
        // Your invariant from a) must hold here.
return $d_{m,n}$
```

Note that for d we started our indexing from 0 and for α and β from 1.

a) Formulate an invariant $INV(i, j)$ that holds after the (i, j)-th iteration of the for loops, i.e., after the computation of $d_{i,j}$ in the pseudo code.

Solution: Let $\alpha_{:i} = (\alpha_1, \ldots, \alpha_i)$ and $\beta_{:j} = (\beta_1, \ldots, \beta_j)$. We formulate the following invariant $INV(i,j)$:

After the computation of $d_{i,j}$ (which happens in one of the for-loops, except for $d_{0,0}$ for which it happens in the initialization), the value of $d_{i,j}$ is equal to the minimal edit distance from $\alpha_{:i}$ to $\beta_{:j}$.

b) Prove the correctness of the algorithm `editDistance` by induction over i and j using your invariant.

Hint: You may ignore the exact order of computation and use the following induction hypothesis of the form: '$INV(i,j)$ holds for all i,j with $i+j \leq k$'. Then, you do an induction step from k to $k+1$.

Solution:

We will show that $INV(i,j)$ holds for all pairs $(i,j) \in \{0,\ldots,m\} \times \{0,\ldots,n\}$ (note that we also cover the cases when $i=0$ or $j=0$). The proof is by induction on $k := i+j$.

Base case If $k=0$ then $i=j=0$. In this case, both strings are the empty string, so their edit distance is zero. We observe that this is what the algorithm sets.

Induction hypothesis Assume, for some $0 \leq k \leq m+n$, $INV(i,j)$ holds for all $0 \leq i \leq m$ and $0 \leq j \leq n$ with $i+j \leq k$.

Note here that $d_{i,j}$ is only computed once. Thus, if the invariant $INV(i,j)$ holds at the point in time when $d_{i,j}$ is computed, it also holds throughout the remaining computation of the algorithm.

Induction step $k \to k+1$ We need to show something for all i,j with $i+j \leq k+1$. For $i+j < k+1$, the desired statement is contained in the induction hypothesis, and there is nothing to show. So the only new case is if $i+j = k+1$. We discriminate two cases. If $i=0$, then the first string is the empty string, and the edit distance to a string of length j is obviously exactly j. Similarly, if $j=0$, then the edit distance is exactly i. We observe that this is set correctly in the first two for loops.

In the remaining case, we have $i,j \geq 1$. For the strings $\alpha_{:i}$ and $\beta_{:j}$, there is some sequence of optimal edits. We distinguish another three cases, depending on what the edit sequence does with the last characters α_i and β_j. Either α_i is deleted by some edit (first case). If it is not deleted, then it must (possibly after some substitution) be matched to some character of $\beta_{:j}$ in the end. If that character is not β_j, then β_j must have been added, since α_i is the last character of the start string (second case). Finally, it might be that α_i is matched to β_j, possibly after substituion (third case).

Case 1: Delete $\boldsymbol{\alpha_i}$ In this case the optimal number of edits required is the edit distance from $\alpha_{:i-1}$ to $\beta_{:j}$ plus one, which is equal to $d_{i-1,j}+1$ by our induction hypothesis. The hypothesis is applicable for the index pair $(i-1,j)$ since $(i-1)+j \leq k$.

Case 2: Add $\boldsymbol{\beta_j}$ In this case the optimal number of edits required is the edit distance from $\alpha_{:i}$ to $\beta_{:j-1}$ plus one, which is equal to $d_{i,j-1}+1$ by our induction hypothesis. Again the hypothesis is applicable for the index pair $(i,j-1)$ since $i+(j-1) \leq k$.

Case 3: Substitute If $\alpha_i = \beta_j$, we don't need to substitute and thus the optimal number of edits required is the edit distance from $\alpha_{:i-1}$ to $\beta_{:j-1}$, which is equal to $d_{i-1,j-1}$ by our induction hypothesis. Else, we substitute and thus the optimal number of edits required is the edit distance from $\alpha_{:i-1}$ to $\beta_{:j-1}$ plus one, which is equal to $d_{i-1,j-1}+1$ by our induction hypothesis. Again the hypothesis is applicable for the index pair $(i-1,j-1)$ since $(i-1)+(j-1) \leq k$.

The minimal number of edits is achieved by the minimum over these three cases which is exactly what `editDistance` does.

This concludes the induction.

At the end of the execution of `editDistance`, $d_{i,j}$ is equal to the amount of edits required to transform $\alpha_{:i}$ to $\beta_{:j}$ for all $0 \leq i \leq m$ and $0 \leq j \leq n$ and as a consequence $d_{m,n}$ is the minimum edit distance.

Exercise 6.2 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_3 &= 1 \\ F_i &= F_{i-1} + F_{i-2} + F_{i-3}. \end{aligned}$$

a) Provide a recursive function (using pseudo code) that computes F_i for $i \in \mathbb{N}$.

Solution:

Algorithm 2 $F(i)$

```
if $i \leq 3$ then
    return $1$
else
    return $F(i-1) + F(i-2) + F(i-3)$
```

b) Lower bound the running time of your recursion from a) using Ω notation.

Solution: The number of operations for a call $F(i)$ is given by the recurrence $T(1) = T(2) = T(3) = 1$ and $T(i) = T(i-1) + T(i-2) + T(i-3) + c$, in which c is a positive constant.

We will show by induction that $T(i) \geq \frac{1}{3} \cdot 3^{i/3}$. For $i = 1, 2, 3$ this is satisfied since $i/3 \leq 1$ for these values, and thus $T(i) = 1 \geq \frac{1}{3} \cdot 3^{i/3}$.

For the inductive step, we bound $T(i)$ from below by the following chain of inequalities

$$\begin{aligned} T(i) &\geq T(i-1) + T(i-2) + T(i-3) \\ &\overset{\text{IV}}{\geq} \frac{1}{3} \cdot \left(3^{(i-1)/3} + 3^{(i-2)/3} + 3^{(i-3)/3}\right) \\ &\geq \frac{1}{3} \cdot \left(3^{(i-3)/3} + 3^{(i-3)/3} + 3^{(i-3)/3}\right) = \frac{1}{3} \cdot 3^{i/3}. \end{aligned}$$

Thus, $T(i) \geq \Omega(3^{i/3})$.

Remark: With a bit more care, it can be shown by induction that $T(i) = \Theta(\phi^i)$, where $\phi \approx 1.8393$ is the unique positive solution of $x^3 = x^2 + x + 1$.

Edit on Nov. 6: The proof above is complete, but it doesn't provide much help for imitating the proof in similar situations. In particular, it does not contain guidelines on how to find a guess for the inductive statement $T(i) \geq \frac{1}{3} \cdot 3^{i/3}$. So here we give some intuitive reasoning on how to find a candidate for such a statement. Note that this is informal reasoning, not part of a proof or of the solution. So we don't need to justify our steps.

First, it seems plausible that T is monotone, i.e., for $j > i$ we have $T(j) \geq T(i)$. Then we can bound

$$T(i) = T(i-1) + T(i-2) + T(i-3) \geq 3T(i-3).$$

Iterating this formula, we get

$$T(i) \geq 3T(i-3) \geq 9 \cdot T(i-6) \geq \ldots \geq 3^k \cdot T(i-3k).$$

For $k \approx i/3$, we obtain the conjecture $T(i) \geq 3^{i/3} =: f(i)$. This conjecture is not yet correct. The inductive step would go through with this conjecture, but it fails for the base cases $i = 1, 2, 3$. However, let us look into the inductive step. It is $T(i) \geq 3T(i-3) \geq 3f(i-3) \geq f(i)$, so the only requirement for f is $3f(i-3) \geq f(i)$. This requirement is still satisfied if we replace f by $c \cdot f$, essentially because both sides of the inequality $3f(i-3) \geq f(i)$ are linear in f. So the inductive step would still work if we use the inductive statement $T(i) \geq c \cdot f(i)$, for a constant c that we can still choose. Now we choose c such that the base cases are also satisfied, i.e., such that $T(1) \geq cf(1)$, $T(2) \geq cf(2)$, and $T(3) \geq cf(3)$. One such choice is $c = 1/3$, and this is used in the solution above.

c) Improve the running time of your algorithm using the memoization. Provide pseudo code of the improved algorithm and analyze its running time.

Solution:

Algorithm 3 Compute F_i using memoization

```
memory ← $i$-dimensional array filled with $(-1)$s
function F_MEM(i)
    if memory[i] $\neq -1$ then                              ▷ If $F_i$ is already computed.
        return memory[i]
    if $i \leq 3$ then
        return 1
    else
        $F_i \leftarrow$ F_Mem($i-1$) + F_Mem($i-2$) + F_Mem($i-3$)
        memory[i] ← $F_i$
        return $F_i$
```

When calling F_Mem(i), each F_j for $1 \leq j \leq i$ is computed only once and then stored in memory. Thus the running time of F_Mem(i) is $\mathcal{O}(i)$.

d) Compute F_i using dynamic programming and state the running time of your algorithm:

Solution:

Dimensions of the DP table: The DP table is linear, its size is i.

Definition of the DP table: $DP[j]$ contains F_j for $1 \leq j \leq i$.

Calculation of an entry: Initialize $DP[1], DP[2]$ and $DP[3]$ to 1.

The entries with $j > 3$ are computed by $DP[j] = DP[j-1] + DP[j-2] + DP[j-3]$.

Calculation order: We can calculate the entries of DP from smallest to largest.

Reading the solution: All we have to do is read the value at $DP[i]$.

Running time: Each entry can be computed in time $\Theta(1)$, so the running time is $\Theta(i)$.

Exercise 6.3 *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 neighbour of 20 in the round in which 20 was entered (round 17), which is 17. Then we go the left neighbour 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 6.4 *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 $DP[i - 1, j]$.

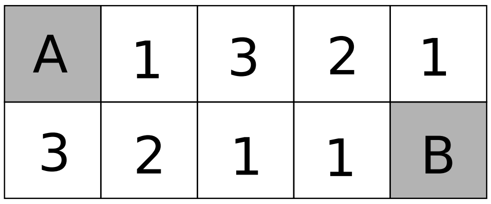
Otherwise, if $DP[i, j - 1] = DP[i, j]$, we move to $DP[i, j - 1]$.

Otherwise, by definition of DP table, $DP[i - 1, j - 1] = DP[i, j] - 1$ and $A[i - 1] = B[j - 1]$, so we assign $S[DP[i - 1, j - 1]] \leftarrow A[i - 1]$ 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 6.5 *Dynamic programming marathon* **(1 point)**.



A	1	3	2	1
3	2	1	1	B

Figure 1: Runner problem for a cost array of size 2×5.

Imagine, a runner wants to run from A to B in Fig. 1. There are two lanes available. One is represented by the first row and the other by the second row. On some sections, the first lane is faster than the second lane, and vice versa. The runner can change lanes at any time, but this costs 1 minute every time. In this exercise, you are supposed to provide a dynamic programming algorithm that computes the optimal track.

Formally, the problem is defined in terms of a cost array $c \in \mathbb{N}^{2\times n}$. In Fig. 1 $n = 5$. Now, the runner starts at position $(1, 1)$ and wants to run to $(2, n)$. Running along a lane from field (i, j) to the field $(i, j + 1)$ requires $c_{i,j+1}$ minutes. Changing lanes from field $(1, j)$ to $(2, j)$ requires $1 + c_{2,j}$ minutes, and from field $(2, j)$ to $(1, j)$ requires $1 + c_{1,j}$ minutes.

Provide an algorithm using dynamic programming that computes the optimal track from A to B. Your algorithm should compute the optimal sequence $(1, 1), (i_1, j_1), \ldots, (i_k, j_k), (2, n)$ and its cost (= the time required by the runner to run the sequence).

Solution:

Dimensions of the DP table:

The DP table is linear, its size is $2 \times n$.

Definition of the DP table:

$DP[i, j]$ contains the smallest cost for reaching the field (i, j) without running backwards.

Calculation of an entry:

Initialize $DP[1, 1] = 0$, $DP[2, 1] = 1 + c_{2,1}$.

The entries with $j > 1$ are computed by

$$DP[1, j] = \min(DP[1, j-1] + c_{1,j}, DP[2, j-1] + c_{2,j} + c_{1,j} + 1)$$

and

$$DP[2, j] = \min(DP[2, j-1] + c_{2,j}, DP[1, j-1] + c_{1,j} + c_{2,j} + 1).$$

This is a correct recursion, since to reach $(1, j)$ the runner must either pass through $(1, j-1)$ or through $(2, j-1)$, and likewise for $(2, j)$.

Calculation order: We compute the entries column by column from left to right.

Reading the solution: The cost of the optimal track equal to the entry $DP[2, n]$.

We obtain an optimal track by backtracking. That is, we start at field $(2, n)$ and we check whether we got there with or without swapping the lane: If we got there without swapping the lane, the equality $DP[2, n] = DP[2, n-1] + c_{2,n}$ must hold and we add $(2, n-1)$ to our optimal track and proceed by checking how we got to $(2, n-1)$. If we got there with a lane swap, the equality $DP[2, n] = DP[1, n-1] + c_{1,n} + c_{2,n} + 1$ must hold and we add $(1, n)$ and $(1, n-1)$ to our optimal track and proceed by checking how we got to $(1, n-1)$. We continue like this until we reach one of the fields $(1, 1)$ or $(2, 1)$. If we reach $(2, 1)$ we add $(1, 1)$ and are done.

Running time: Each entry can be computed in time $\Theta(1)$, so the running time for the computation of the table is $\Theta(n)$. Reading the solution requires $\Theta(1)$. Our backtracking is also $\Theta(n)$, as we have to check exactly $n-1$ times from which field we came from. Thus, our solution requires $\Theta(n)$ time.