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
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

1. November 2021

Algorithms & Data Structures — Exercise sheet 6 — HS 21

Exercise Class (Room & TA): ____________________

Submitted by: ____________________

Peer Feedback by: ____________________

Points: _____

Submission: On Monday, 8. November 2021, 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 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$F_1 = 1$$
$$F_n = \left(\min_{1\le i<n} F_i^2 + F_{n-i}^2\right) \bmod 3n \quad \text{for } n \ge 2,$$

where $a \bmod b$ is the remainder of dividing a by b.

a) Consider the following algorithm that computes F top-down

Algorithm 1 Computing $F(n)$

```
function $F(n)$
    if $n = 1$ then
        return $1$
    else
        $x \leftarrow F(1)^2 + F(n-1)^2$
        for $i = 2 \dots \lfloor \frac{n}{2} \rfloor$ do
            $x \leftarrow \min(x, F(i)^2 + F(n-i)^2)$
        return $x \bmod 3n$
```

Lower bound the running time $T(n)$ of the above algorithm (i.e., give a simple function $g(n)$ such that $T(n) \ge \Omega(g(n))$) and show that it has an exponential running time.

Hint: *Prove by induction that* $T(n) \ge (3/2)^{n-1}$.

Solution: Note that for $n = 1$, $T(1) \ge 1$ and for $n = 2$, $T(2) \ge T(1) + T(1) \ge 2$. For $n > 1$,

$$T(n) \ge \sum_{i=1}^{n-1} T(i)\,,$$

Let's prove by induction that $T(n) \ge (3/2)^{n-1}$.

- **Base Case.**
  For $n = 1$, $T(n) \geq 1 = (3/2)^{n-1}$, and for $n = 2$, $T(n) \geq 2 \geq (3/2)^{n-1}$.
- **Induction Hypothesis.**
  Assume that for some $k \geq 2$ the property holds all positive integers $m \leq k$. That is, $T(m) \geq (3/2)^{m-1}$.
- **Inductive Step.**
  We must show that the property holds for $k + 1$.

$$T(k+1) \geq \sum_{i=1}^{k} T(i) \geq \sum_{i=1}^{k} (3/2)^{i-1} = \frac{(3/2)^k - 1}{(3/2) - 1} = 2 \cdot (3/2)^k - 2 \geq (3/2)^k + ((3/2)^k - 2) \geq (3/2)^k \,,$$

  where we used the fact that for $k \geq 2$, $(3/2)^k \geq 2$.

b) Improve the running time of the algorithm in (a) using memoization. Provide pseudo code of the improved algorithm.

Solution:

Algorithm 2 Computing $F(n)$ using memoization

```
memory ← array of size $n$ filled with $(-1)$s
function $F_{mem}(n)$
    if memory[$n$] $\neq -1$ then                              ▷ If $F(n)$ is already computed.
        return memory[n]
    if $n = 1$ then
        return $1$
    else
        $x \leftarrow F_{mem}(1)^2 + F_{mem}(n-1)^2$
        for $i = 2 \dots \lfloor \frac{n}{2} \rfloor$ do
            $x \leftarrow \min(x, F_{mem}(i)^2 + F_{mem}(n-i)^2)$
        memory[$n$] ← $x \bmod 3n$
        return memory[$n$]
```

When calling $F_{mem}(n)$, each $F(i)$ for $1 \leq i \leq n$ is computed only once and then stored in `memory`. This substantially enhances the running time of the algorithm.
Remark. The running time of the above memoization algorithm has the same asymptotic growth as the runtime of the algorithm in c) that uses dynamic programming, namely, $\Theta(n^2)$.

c) Compute $F(n)$ bottom-up 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 n.

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

Calculation of an entry: Initialize $DP[1]$ to 1.

The entries with $n > 1$ are computed by

$$DP[n] = \left(\min_{1 \leq i \leq \lfloor \frac{n}{2} \rfloor} DP[i] \cdot DP[i] + DP[n-i] \cdot DP[n-i] \right) \bmod 3n.$$

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 $DP[n]$.

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

$$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2).$$

Exercise 6.2 *Longest common substring: finding an invariant* **(1 point)**.

Let $\Sigma = \{a, b, c, \dots, z\}$ denote the alphabet. Given two strings $\alpha = (\alpha_1, \dots, \alpha_m) \in \Sigma^m$ and $\beta = (\beta_1, \dots, \beta_n) \in \Sigma^n$, we are interested in the length of their longest common *substring*, which is the largest integer k such that there are indices i and j with $(\alpha_i, \alpha_{i+1}, \dots, \alpha_{i+k-1}) = (\beta_j, \beta_{j+1}, \dots, \beta_{j+k-1})$. Note that this problem is different from the longest common subsequence problem that you saw in the lecture. For example, the longest common substring of $\alpha = (a, a, b, c, b, a)$ and $\beta = (a, b, a, b, c, a)$ is (a, b, c), which is of length 3.

Below is the pseudo-code of an algorithm that computes the length of the longest common substring of two strings $\alpha \in \Sigma^m$ and $\beta \in \Sigma^n$ using $\Theta(mn)$ elementary operations:

Algorithm 3 `LongestCommonSubstring`(α, β)

```
$L \leftarrow \mathbf{0}^{m \times n}$ an $m \times n$ matrix of zeros.
for $i = 1, \dots, m$ do
    for $j = 1, \dots, n$ do
        if $\alpha_i = \beta_j$ then
            if $i = 1$ or $j = 1$ then
                $L_{i,j} = 1$
            else
                $L_{i,j} = L_{i-1,j-1} + 1$
        else
            $L_{i,j} = 0$
        // Your invariant from a) must hold here.
return $\max\{L_{i,j} : 1 \le i \le m, 1 \le j \le n\}$
```

a) Execute the algorithm on the strings $\alpha = (a, b, c)$ and $\beta = (c, a, b)$. Write down the value of the matrix L after each pass of the inner for-loop.

Solution: Since $\alpha_1 = a \neq c = \beta_1$, we keep $L_{1,1} = 0$ in the first iteration, which yields

$$L = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_1 = a = \beta_2$, we set $L_{1,2} = 1$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_1 = a \neq b = \beta_3$, we keep $L_{1,3} = 0$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_2 = b \neq c = \beta_1$, we keep $L_{2,1} = 0$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_2 = b \neq a = \beta_2$, we keep $L_{2,2} = 0$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_2 = b = \beta_3$, we set $L_{2,3} = L_{1,2} + 1 = 2$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_3 = c = \beta_1$, we set $L_{3,1} = 1$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 1 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_3 = c \neq a = \beta_2$, we keep $L_{3,2} = 0$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 1 & 0 & 0 \end{pmatrix}.$$

Since $\alpha_3 = c \neq b = \beta_3$, we keep $L_{3,3} = 0$ in the next iteration, which yields

$$L = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 1 & 0 & 0 \end{pmatrix}.$$

The algorithm then returns $\max\{L_{i,j} : 1 \leq i \leq 3, 1 \leq j \leq 3\} = 2$.

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

Hint: *Consider the subproblem of finding the longest common substring that ends at some given indices of α and β.*

Solution: We formulate the following invariant $INV(i,j)$:

After the computation of $L_{i,j}$, for all $1 \le i' \le i$ and all $1 \le j' \le j$, the value of $L_{i',j'}$ is equal to the length of the longest common substring of α and β that ends at $\alpha_{i'}$ and $\beta_{j'}$ respectively.

c) Prove by induction that $INV(i,j)$ holds for all $1 \le i \le m$ and $1 \le j \le n$. Deduce that the algorithm `LongestCommonSubstring` is correct.

Hint: *You can perform induction over the minimum index $k := \min(i,j)$.*

Solution:

We prove that $INV(i,j)$ holds for all $1 \le i \le m$ and all $1 \le j \le n$ by induction on $k = \min(i,j)$.

Base case If $k = 1$ then either $i = 1$ or $j = 1$. The two cases are similar, so without loss of generality we assume that $i = 1$. For any $1 \le j \le n$, the length of the longest common substring ending at α_1 and β_j is 1 if $\alpha_1 = \beta_j$, and 0 otherwise. This is exactly the value given to $L_{1,j}$ by the algorithm in the $(1,j)$-th pass of the for-loop.

Induction hypothesis Assume, for some $1 \le k < \min(m,n)$, that $INV(i,j)$ holds for all $1 \le i \le m$ and $1 \le j \le n$ with $\min(i,j) \le k$.

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

Induction step $k \to k+1$ Let $1 \le i \le m$ and $1 \le j \le n$ be such that $\min(i,j) = k+1$, and consider some $1 \le i' \le i$ and $1 \le j' \le j$. Then $\min(i',j') \le \min(i,j) = k+1$.

If $\min(i',j') \le k$, then $INV(i',j')$ holds by the induction hypothesis, and therefore $L_{i',j'}$ is equal to the length of the longest common substring of α and β that ends at $\alpha_{i'}$ and $\beta_{j'}$.

If $\min(i',j') = k+1$, then $\min(i'-1,j'-1) = k$, so again using the induction hypothesis we know that $L_{i'-1,j'-1}$ is equal to the length of the longest common substring of α and β that ends at $\alpha_{i'-1}$ and $\beta_{j'-1}$. If $\alpha_{i'} \ne \beta_{j'}$, then the longest common substring of α and β that ends at $\alpha_{i'}$ and $\beta_{j'}$ has length 0, and indeed the algorithm will set $L_{i',j'} = 0$. On the other hand, if $\alpha_{i'} = \beta_{j'}$, then the length of the longest common substring of α and β that ends at $\alpha_{i'}$ and $\beta_{j'}$ is simply 1 plus the length of the longest common substring of α and β that ends at $\alpha_{i'-1}$ and $\beta_{j'-1}$, i.e. it is $1 + L_{i'-1,j'-1}$. This is exactly the value to which $L_{i',j'}$ will be set by `LongestCommonSubstring`(α,β).

This concludes the induction. Therefore, at the end of the execution of `LongestCommonSubstring`, $INV(m,n)$ holds. This means that for all $1 \le i \le m$ and all $1 \le j \le n$, the value of $L_{i,j}$ is equal to the length of the longest common substring of α and β that ends at α_i and β_j. In particular, the length of the longest common substring of α and β is $\max\{L_{i,j} : 1 \le i \le m, 1 \le j \le n\}$, which is what is returned by `LongestCommonSubstring`(α,β).

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 $A[i] = B[j]$, 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 6.5 *Optimizing Starduck's profit* **(1 point)**.

The coffeeshop chain Starduck's is planning to open several cafés in Bahnhofstrasse Zürich. There are n possible locations $1, \dots, n$ for their shops on Bahnhofstrasse, ordered by their distance to Zürich main station $m_1 < \ldots < m_n$. Opening a shop at location i would yield Starduck's a profit of $p_i > 0$.

However, they are not allowed to open cafés that are too close to each other, namely any two cafés should have distance at least d from each other, for some given value $d > 0$.

a) Provide an algorithm using dynamic programming that computes the maximum total profit that Starduck's can make on Bahnhofstrasse. In order to get full points, your algorithm should have $O(n \log n)$ runtime.

   Hint: *Consider the subproblem of finding the maximum total profit that Starduck's can make if only locations* $1, \dots, i$ *are available*

   Solution:

   Dimensions of the DP table:

   The DP table is linear and contains entries from $DP[0]$ to $DP[n]$, its size is $n + 1$.

   Definition of the DP table:

   $DP[i]$ contains the maximum total profit that Starduck's can make using only the locations $1, \dots, i$ (taking the distance restriction into account).

   Calculation of an entry:

   Initialize $DP[0] = 0$.

   Let $i \geq 1$ and denote by $j(i)$ the largest index j such that $m_j \leq m_i - d$ (and $j(i) = 0$ if there is no such index). Then $DP[i]$ is computed by

$$DP[i] = \max\{DP[i-1], p_i + DP[j(i)]\}. \tag{1}$$

   Let us show the correctness of this formula. The optimal opening strategy of Starduck's using only the locations $1, \dots, i$ either opens a café at location i or does not. If it does not, then clearly the strategy is the same as the optimal startegy using only the locations $1, \dots, i-1$, whose profit is $DP[i-1]$. If it does open a café at location i, then it is not allowed to open cafés at any location j with $m_j + d > m_i$ because of the distance restriction. Since the m_j's are in inreasing order ($m_1 < \dots < m_n$), this means that one cannot open cafés at any of the locations $j(i)+1, j(i)+2, \dots, i-1$. In the (potentially empty) set of remaining locations, namely all locations up to $j(i)$, one should clearly apply the optimal strategy that yields $DP[j(i)]$ profit, so overall this strategy will yield total profit $p_i + DP[j(i)]$. The maximum total profit that can be made using locations $1, \dots, i$ is then the maximum between these two possibilities ($DP[i-1]$ if no café is opened at locations i, and $p_i + DP[j(i)]$ if a café is opened at location i), which proves formula (1).

   Calculation order: We compute the entries from left to right (i.e. from 0 to n).

   Reading the solution: The maximum total profit is simply $DP[n]$.

   Running time: In order to compute $DP[i]$, we first need to find out the value of $j(i)$. This can be done in time $\lceil \log_2 i \rceil = O(\log n)$ using binary search. Therefore, the computation of an entry takes $O(\log n)$ time, and since there are $n + 1$ entries the total running time is $O(n \log n)$.

b)* You now would like to recover not only the maximum total profit, but the corresponding locations where shops should be opened in order to achieve this profit. How can you get this out of your DP table in time $O(n)$?
   Remark. There might be multiple optimal opening strategies, and it is enough if you can recover just one of them from the DP table.

   Solution:

We obtain an optimal strategy by backtracking. More precisely, remembering the way we built our DP table and justified the correctness of this construction, we know that if $DP[i]$ and $DP[i-1]$ are different, then the optimal strategy restricted to locations $1, \ldots, i$ actually opens a café at location i.

Let us initialize an empty list L. We start from the right of the DP table (namely at index $i = n$) and decrease index i gradually (i.e. to $n-1$, then $n-2$ and so on) until we find the first index i_1 for which the DP-value changes (i.e. i_1 is the largest index such that $DP[i_1] \neq DP[i_1 - 1]$). Since by definition $DP[i_1] = DP[n]$, we deduce that the optimal strategy opens a café at location i_1, so we add i_1 to the list L . Moreover since $DP[i_1] \neq DP[i_1 - 1]$, by equation (1) we know that $DP[i_1] = p_{i_1} + DP[j(i_1)]$. We continue decreasing our index i gradually until we find some index j with $m_j \leq m_{i_1} - d$, namely until we find $j(i_1)$. We then continue to decrease i to $j(i_1)-1, j(i_1)-2$ and so on until we find the first index i_2 for which the DP-value decreases, i.e. $DP[i_2] \neq DP[i_2-1]$. Again, since $DP[i_2] = DP[j(i_1)]$, this location i_2 must be part of the optimal strategy restricted to locations $1, \ldots, j(i_1)$, and therefore of the total optimal strategy since $DP[n] = DP[i_1] = p_{i_1} + DP[j(i_1)]$, so we add i_2 to the list L. Again, we continue decreasing the index until we find $j(i_2)$, and then find the first index smaller or equal to $j(i_2)$ for which the DP-value decreases, and continue this until we reach the index $i = 0$.

Here the pseudo-code of the described procedure. It returns the list of locations where a café should be opened to yield maximum profit.

Algorithm 4 Backtrack(DP)

```
$L \leftarrow []$
$i \leftarrow n - 1$
$OPT \leftarrow DP[n]$
**while** $i \geq 0$ **do**
    **if** $DP[i] < OPT$ **then**
        Add $i + 1$ to $L$
        $m \leftarrow m_{i+1} - d$
        **while** $m_i > m$ **do**
            $i \leftarrow i - 1$
        $OPT \leftarrow DP[i]$
    $i \leftarrow i - 1$
**return** $L$
```

Note that this runs in time $O(n)$, since for each decreasing of i we perform a constant number of operations, and i is decreased exatcly n times (from $n-1$ to -1).