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

04. November 2019

Algorithms & Data Structures — Exercise sheet 7 — HS 19

Exercise Class (Room & TA): ____________________

Submitted by: ____________________

Peer Feedback by: ____________________

Points: _____

Submission: On Monday, 11 November 2019, 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 7.1 *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 as described in class or the script to find the length of a longest ascending subsequence and the subsequence itself. Show all necessary tables and information you used to obtain the solution.

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 7.2 *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 as described in class or 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 7.3 *Tinder Don*na Juan*a* **(1 Point)**.

You registered on Tinder and you got a lot of matches (you may assume that you have an endless amount of matches). Now, you would like to create a schedule for your dates. You don't date more than one person per day. Further, the day after having a date you always tell your best friend how it went and, thus, do not have time for a date on that day.

You tell your best friend about your success on Tinder and that you are trying to find a nice schedule for your dates. Your best friend challenges you to enumerate all possible date-schedules for the next T days. A schedule consists of T entries, where the i-th entry contains whether you have a date on this day or not.

Use dynamic programming to determine the number of different date-schedules under your constraints.

***Hint:** In order to achieve full points your algorithm should solve this problem using $\Theta(T)$ time and memory.*

Address the following aspects in your solution:

1. *Definition of the DP table: What are the dimensions of the table $DP[\dots]$?* What is the meaning of each entry?
2. *Definition of the DP table*: What is the meaning of each entry?

3. *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the final solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

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

Definition of the DP table: $DP[i]$ contains the number of possible shedules if there are i days.

Calculation of an entry: Initialize $DP[1]$ to 2: you can either have a date on day 1 or not. Initialize $DP[2]$ to 3: you can have date on the first day or on the second day or don't have dates.

An entry $i > 2$ can be calculated as follows: $DP[i]$ can be calculated by adding up the number of possible shedules if you have a date on day i plus the number of possible shedules if you do not have a date on day i.

If you don't have a date on day i, this places no restriction on the shedules, so the number of possible shedules in this event is $DP[i-1]$. If you have a date on day i, you can't have a date on day $i-1$. We've placed a restriction on day $i-1$ and i but not on any days before that, so the number of possible schedules in this case is equal to $DP[i-2]$.

Hence $DP[i] = DP[i-1] + DP[i-2]$.

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 $DP[T]$.

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

Remark 1. The running time is $\Theta(T)$ only if we work with unit cost model, where arithmetic operations cost $\Theta(1)$. Notice that in order to compute the k-th entry of the DP table we should add two $\Theta(k)$-digit numbers, and in practice we need $\Theta(k)$ time to do it. Hence in more realistic computational model the running time of the algorithm described above is $\Theta(n^2)$.

Remark 2. It is easy to prove inductively that $DP[i] = \mathrm{Fib}_{i+2}$ is the $i+2$-nd Fibonacci number. Therefore, it is also possible to solve the problem without dynamic programming, by computing directly

$$DP[T] = \mathrm{Fib}_{T+2} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{T+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{T+2}\right).$$

In fact, by iterated squaring, this is possible with $O(\log T)$ floating point operations. However, Remark 1 still applies: $DP[T]$ is a $\Theta(T)$-digit number, so even to output the result one actually needs $\Omega(T)$ time. Moreover, one would need to think about the floating point decision that is needed so that the result is correctly rounded.

Exercise 7.4 *Longest Snake* **(2 points)**.

You are given a game-board consisting of hexagonal fields $F_1, \ldots, F_n$. The fields contain natural numbers $v_1, \ldots, v_n \in \mathbb{N}$. Two fields are neighbours if they share a border. We call a sequence of fields $(F_{i_1}, \ldots, F_{i_k})$ a *snake* of length k if, for $j \in \{1, \ldots, k-1\}$, F_{i_j} and $F_{i_{j+1}}$ are neighbours and their values satisfy $v_{i_{j+1}} = v_{i_j} + 1$. Figure 1 illustrates an example game board in which we highlighted the longest snake.

For simplicity you can assume that F_i are represented by their indices. Also you may assume that you know the neighbours of each field. That is, to obtain the neighbours of a field F_i you may call $\mathcal{N}(F_i)$, which will return the neighbours $\mathcal{N}(F_i) = \{F_{j_1}, \ldots, F_{j_6}\}$. Each call of $\mathcal{N}$ takes unit time.

a) Provide a *dynamic programming* algorithm that, given a game-board $F_1, \ldots, F_n$, computes the length of the longest snake.

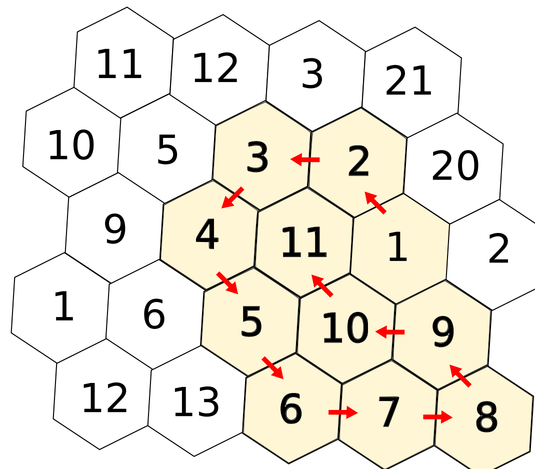

Figure 1: Example of a longest snake.

Hint: *In order to achieve full points your algorithm should solve this problem using $\mathcal{O}(n \log n)$ time, where n is the number of hexagonal fields.*

Address the following aspects in your solution:

(a) *Definition of the DP table: What are the dimensions of the table $DP[\ldots]$?* What is the meaning of each entry?

(b) *Definition of the DP table*: What is the meaning of each entry?

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

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

Meaning of a table entry (in words): $DP[i]$ is the length of the longest snake with head F_i (that is, the length of the longest snake of the form $(F_{j_1}, \ldots, F_{j_{m-1}}, F_i)$).

Computation of an entry (initialization and recursion):

$$DP[i] = 1 + \max_{\substack{F_j \in \mathcal{N}(F_i)\\ v_j = v_i - 1}} DP[j].$$

That is, we look at those neighbors of F_i that have values v_j smaller than v_i exactly by 1, and choose the maximal value in the DP table among them. If there are no such neighbors, we assume that $\max$ in this formula is 0.

Order of computation: We first sort the hexagons by their values. Then we fill the table in ascending order, that is, $i_1, \dots, i_n$ such that $v_{i_j} \le v_{i_{j+1}}$ for all $j = 1, \dots n-1$.

Computing the output: The output is $\max_{1 \le i \le n} DP[i]$.

Running time: We compute the order in time $\mathcal{O}(n \log n)$ by sorting $v_1, \dots, v_n$. Then each entry can be computed in time $\mathcal{O}(1)$ and finally we compute the output in time $\mathcal{O}(n)$. So the running time of the algorithm is $O(n \log n)$.

b) Provide an algorithm that takes as input $F_1, \dots F_n$ and a DP table from part a) and outputs the longest snake. If there are more than one longest snake, your algorithm can output any of them. State the running time of your algorithm in Θ-notation in terms of n.

Solution: At the beginning we find a head of a snake that is some F_{j_1} such that $DP[j_1] = \max_{1 \le i \le n} DP[i]$. If $DP[j_1] \ne 1$, we look at its neigbours and find some F_{j_2} such that $DP[j_2] = DP[j_1] - 1$. If $DP[j_2] \ne 1$, then among neighbors of F_{j_2} we find some F_{j_3} such that $DP[j_3] = DP[j_2] - 1$ and so on. We stop when $DP[j_m] = 1$ (where m is exactly the length of the longest snake). Then we output the snake $(F_{j_1}, \dots, F_{j_m})$.

The running time of this algorithm is $\Theta(n)$, since we use $\Theta(n)$ operations to find F_{j_1} and we need $\Theta(1)$ time to find each F_{j_k} for $1 < k \le m \le n$ and $\Theta(m)$ time to output the snake.

c)* Find a linear time algorithm that finds the longest snake. That is, provide an $\mathcal{O}(n)$ time algorithm that, given a game-board $F_1, \dots, F_n$, outputs the longest snake (if there are more than one longest snake, your algorithm can output any of them).

Solution: We can use recursion with memorization. Similar to part a), we will fill an array $S[1, \dots, n]$ of lengths of longest snakes, that is, $S[i]$ is the length of the longest snake with head F_i. Consider the following pseudocode:

Algorithm 1 Fill-lengths($v_1, \dots, v_n$)

```
S[1], ..., S[n] ← 0, ..., 0
for i = 1, ..., n do
    if S[i] = 0 then
        Move-to-tails(i, S, v_1, ..., v_n)
return S
```

where the procedure Move-to-tails($i, v_1, \dots, v_n$) is:

Algorithm 2 Move-to-tails$(i, S, v_1, \dots, v_n)$

```
for F_j ∈ N(F_i) do
    if v_j = v_i − 1 and S[j] = 0 then
        Move-to-tails(j, S, v_1, ..., v_n)
S[i] = 1 + max_{F_j ∈ N(F_i), v_j = v_i − 1} S[j]
```

As in part a), we assume that $\max$ over the empty set is 0. After we fill S, we can use the same algorithm as in part b) to find a longest snake (we should replace DP by S in the description of that algorithm).

Note that it is important that we do not make a copy of S each time when we call Move-to-tails (so if we implement this algorithm in a real programming language, we should pass S by reference or by pointer).

The running time is linear in n since for each $i \in \{1, \dots, n\}$ we call Move-to-tails$(i, S, v_1, \dots, v_n)$ exactly once. Let's denote the running time of Move-to-tails$(i, S, v_1, \dots, v_n)$ by $T[i]$. We have

$$T[i] = \sum_{j \in R_i} \big(T[j] + \Theta(1)\big) + \Theta(1) = \sum_{j \in R_i} T[j] + \Theta(1),$$

where R_i is the set of j such that Move-to-tails$(j, S, v_1, \dots, v_n)$ is called in the for-loop during the execution of Move-to-tails$(i, S, v_1, \dots, v_n)$ (the empty sum is 0). The total running time is

$$\sum_{i=1}^{n} \big(T[i] + \sum_{j \in R_i} T[j]\big) = \sum_{i=1}^{n} \Theta(1) = \Theta(n).$$

In fact the technique that we used here is closely related to the depth-first search and topological ordering of a graph. These topics will be studied later in this course.