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

02. November 2020

Algorithms & Data Structures — Exercise sheet 7 — HS 20

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

Points: _____

Submission: On Monday, 9 November 2020, send your solution by email to your TA and your peer graders between 09:00 and 09:15 in the morning. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 7.1 *Tinder Don*na Juan*a.*

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, after having a date you always tell your best friend how it went before going to your next date.

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 gives you a list K of days on which he/she is not available, and 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. Note that you always need to have a day in which your best friend is available between two of your dates.

Use dynamic programming to determine the number of different date-schedules under these constraints. In an exam, we would give full points for an $\mathcal{O}(T)$ solution, but you may get partial points for larger runtimes like $\mathcal{O}(T \cdot |K|)$.

Address the following aspects in your solution:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\ldots]$?
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:

Definition of the DP table:

Computation of an entry:

Calculation order:

Extracting the solution:

Running time:

Exercise 7.2 *Longest Snake.*

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 neighbors 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 neighbors 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 neighbors of each field. That is, to obtain the neighbors of a field F_i you may call $\mathcal{N}(F_i)$, which will return the set of the neighbors of F_i. 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.

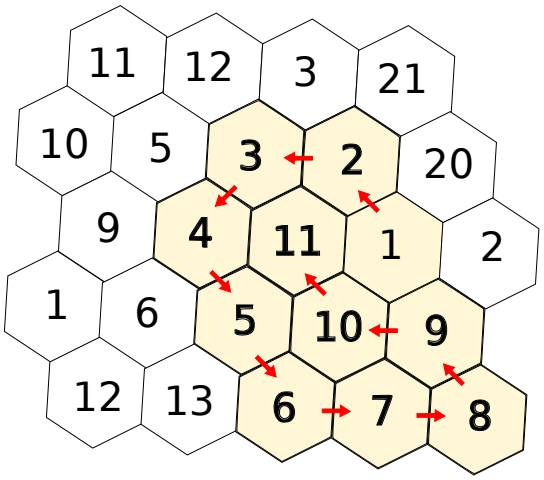

Figure 1: Example of a longest snake.

Hint: *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:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\ldots]$?
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:

Definition of the DP table:

Computation of an entry:

Calculation order:

Extracting the solution:

Running time:

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.

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).

Exercise 7.3 *Making change with few coins* **(2 points)**.

Suppose that you have (infinitely many) coins of different values $x_1, \dots, x_n \in \mathbb{N}$, and you want to make them sum to a given amount with as few coins as possible (where it is allowed to use several coins of the same value). More formally, for some amount $a \in \mathbb{N}$, you want to determine the minimal number of coins k that are needed so that their values sum to a (if it is not possible to get amount a with the given coin values, we will say that $k = \infty$). For example, if you have coins of values 3 and 7, then if $a = 20$ we can get this amount with $k = 4$ coins (and it's not possible to get it with fewer coins), while if $a = 8$ then $k = \infty$ since it's impossible to make such coins sum to 8.

Use dynamic programming to compute the minimal number of coins needed to get amount a. Your solution should have runtime $\mathcal{O}(an)$.

Address the following aspects in your solution:

1. *Dimensions of the DP table*: What are the dimensions of the table $DP[\dots]$?
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:

Definition of the DP table:

Computation of an entry:

Calculation order:

Extracting the solution:

Running time:

Exercise 7.4 *Integer partitions* **(1 point)**.

An integer partition of $n \in \mathbb{Z}_{\geq 0}$ is a way of writing n as a sum of positive integers. For example, the integer partitions of 3 are: 3, $2+1$ and $1+1+1$. The order of the summands does not matter, i.e., $2+1$ and $1+2$ are the same partition of 3.

The partition function $p(n)$ computes the number of integer partitions of n. For example, $p(0) = 1$, $p(1) = 1$, $p(2) = 2$, $p(3) = 3$ and $p(4) = 5$. Despite looking seemingly simple, no closed form for the partition function $p(n)$ is known. However, it is possible to compute $p(n)$ in quadratic time and linear memory by dynamic programming. Your task is to derive this algorithm.

Hint: *Develop first a solution that needs quadratic time and quadratic memory, then think about how to save memory. Such a solution (with quadratic memory) would still give partial points in an exam.*

Dimensions of the DP table:

Definition of the DP table:

Computation of an entry:

Calculation order:

Extracting the solution:

Running time: