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

7 November 2022

Algorithms & Data Structures — Exercise sheet 7 — HS 22

The solutions for this sheet are submitted at the beginning of the exercise class on 14 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 7.1 *k-sums* (1 point).

We say that an integer $n \in \mathbb{N}$ is a *k-sum* if it can be written as a sum $n = a_1^k + \cdots + a_p^k$ where $a_1, \ldots, a_p$ are distinct natural numbers, for some arbitrary $p \in \mathbb{N}$.

For example, 36 is a 3-*sum*, since it can be written as $36 = 1^3 + 2^3 + 3^3$.

Describe a DP algorithm that, given two integers n and k, returns True if and only if n is a k-sum. Your algorithm should have asymptotic runtime complexity at most $O(n^{1+\frac{2}{k}})$.

Hint: *The intended solution has complexity* $O(n^{1+\frac{1}{k}})$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
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 solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Exercise 7.2 *Road trip.*

You are planning a road trip for your summer holidays. You want to start from city C_0, and follow the only road that goes to city C_n from there. On this road from C_0 to C_n, there are $n-1$ other cities $C_1, \ldots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \ldots, C_{n-1}$ are right on the road from C_0 to C_n). For each $0 \leq i \leq n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \ldots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \ldots, C_{n-1}$ you will make an additional stop (you will stop in C_0 and C_n anyway). However, you do not want to drive more than d kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city C_i you can only go forward to cities C_j with $j > i$).

(a) Provide a *dynamic programming* algorithm that computes the number of possible routes from C_0 to C_n that satisfies these conditions, i.e., the number of allowed subsets of stop-cities. In order to get full points, your algorithm should have $O(n^2)$ runtime.

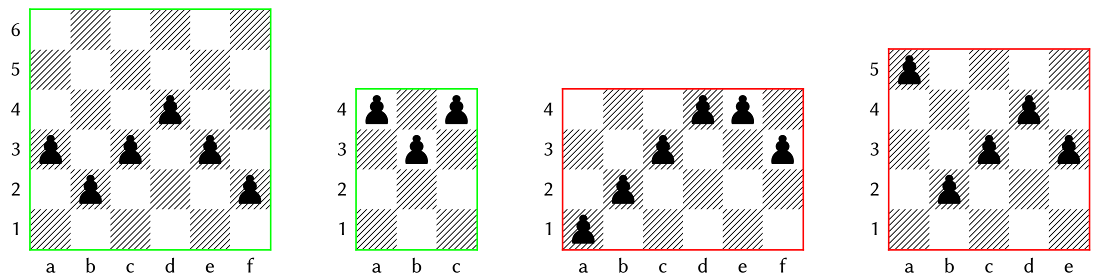
Address the same six aspects as in Exercise 7.1 in your solution.

(b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

Exercise 7.3 *Safe pawn lines* **(1 point)**.

On an $N \times M$ chessboard (N being the number of rows and M the number of columns), a *safe pawn line* is a set of M pawns with exactly one pawn per column of the chessboard, and such that every two pawns from adjacent columns are located diagonally to each other. When a pawn line is not safe, it is called *unsafe*.

The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.



Describe a DP algorithm that, given $N, M > 0$, counts the number of safe pawn lines on an $N \times M$ chessboard. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(NM)$.

Exercise 7.4 *String Counting* **(1 point)**.

Given a binary string $S \in \{0, 1\}^n$ of length n, let $f(S)$ be the length of the longest substring of consecutive 1s. For example $f(\text{"}011000110\underline{111}0001\text{"}) = 3$ because the string contains "111" (underlined) but not "1111". Given n and k, the goal is to count the number of binary strings S of length n where $f(S) = k$.

Write the <u>**pseudocode**</u> of an algorithm that, given positive integers n and k where $k \leq n$, reports the required answer. For full points, the running time of your solution can be any polynomial in n and k (e.g., even $O(n^{11}k^{20})$ is acceptable).

Hint: *The intended solution has complexity* $O(nk^2)$.

In your solution, address the same six aspects as in Exercise 7.1.

Exercise 7.5 *Longest Snake.*

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

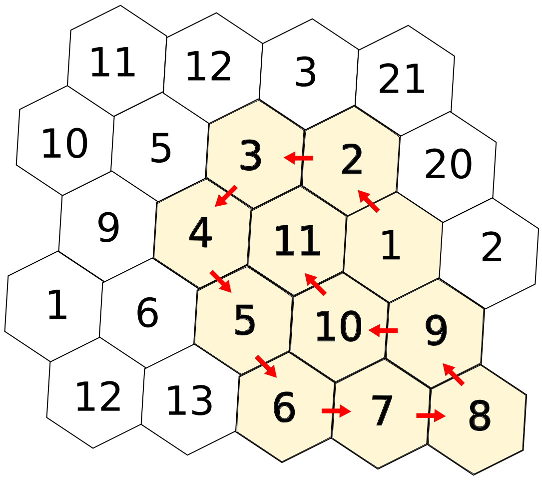


Figure 1: Example of a longest snake.

__Hint:__ Your algorithm should solve this problem using $O(n \log n)$ time, where n is the number of hexagonal fields.

Address the same six aspects as in Exercise 7.1 in your solution.

(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 $O(n)$ 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).