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?

Solution:

Given n and k, let $m = \lfloor n^{1/k} \rfloor$ be the largest integer such that $m^k \leq n$.

1. *Dimensions of the DP table*: $DP[0 \ldots n][0 \ldots m]$
2. *Definition of the DP table*: $DP[i][j]$ is True if, and only if, i can be written as a sum $i = a_1^k + \cdots + a_p^k$ where $p \in \mathbb{N}$, the $(a_\ell)_{1 \leq \ell \leq p}$ are distinct, and $\{a_1, \ldots, a_p\} \subseteq \{1..j\}$.

3. *Computation of an entry*: DP can be computed recursively as follows:

$$
\begin{aligned}
DP[0][j] &= \texttt{True} && 0 \le j \le m && (1)\\
DP[i][0] &= \texttt{False} && 0 < i \le n && (2)\\
DP[i][j] &= DP[i-j^k][j-1] \texttt{ or } DP[i][j-1] && j^k \le i \le n && (3)\\
DP[i][j] &= DP[i][j-1] && \text{otherwise.} && (4)
\end{aligned}
$$

   Equation (1) expresses that 0 can always be written as an (empty) sum of distinct integers in any interval $\{1..j\}$. Equation (2) says that non-zero values cannot be obtained as a sum of integers in $\{1..0\} = \emptyset$. Equation (3) and (4) provide the recurrence relation. An integer j can be obtained as a sum $i = a_1^k + \cdots + a_p^k$ of distinct integers in $\{1..j\}$ iff <u>either</u>

   (a) Some a_ℓ (for example, a_p) is j and the rest of the sum is $a_1^k + \cdots + a_{p-1}^k = i - a_p^k = i - j^k$, such that $\{a_1, \ldots, a_{p-1}\} \subseteq \{1..j-1\}$ <u>or</u>

   (b) No a_ℓ is j, and $a_1^k + \cdots + a_p^k = i$ is a sum of integers from $\{1..j-1\}$.

   Case (a) corresponds to $DP[i-j^k][j-1]$, case (b) to $DP[i][j-1]$. When $j^k < i$, both cases are possible, and we obtain formula (3); when $j^k > i$, only case (2) is possible, and we obtain (4).

4. *Calculation order*: Following the recurrence relations above, we can compute first by order of increasing j, and then in an arbitrary order for i (for example, in increasing order).

5. *Extracting the solution*: The solution is $DP[n][m]$, since any a_ℓ that appears in a sum $a_1^k + \cdots + a_p^k = n$ is such that $a_\ell^k \le n$, which implies $a_\ell \le \lfloor n^{1/k} \rfloor = m$.

6. *Running time*: The running time of the solution is $O(n \cdot n^{\frac{1}{k}}) = O(n^{1+\frac{1}{k}})$ as there are $(n+1) \cdot (m+1) = O(nm) = O(n \cdot n^{\frac{1}{k}})$ entries in the table, we process each entry in $O(1)$ time, and the solution is extracted in $O(1)$ time.

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 $0 \le i \le n$, 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.

   Solution:

   1. *Dimensions of the DP table:* The DP table is linear, and its size is $n + 1$.

2. *Definition of the DP table:* $DP[i]$ is the number of possible routes from C_0 to C_i (which stop at C_i).

3. *Computation of an entry:* Initialize $DP[0] = 1$.

   For every $i > 0$, we can compute $DP[i]$ using the formula
   $$DP[i] = \sum_{\substack{0 \le j < i \\ k_i \le k_j + d}} DP[j]. \tag{5}$$

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

5. *Extracting the solution:* All we have to do is read the value at $DP[n]$.

6. *Running time:* For $i = 0$, $DP[0]$ is computed in $O(1)$ time. For $i \ge 1$, the entry $DP[i]$ is computed in $O(i)$ time (as we potentially need to take the sum of i entries). Therefore, the total runtime is $O(1) + \sum_{i=1}^{n} O(i) = O(n^2)$.

(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) ?

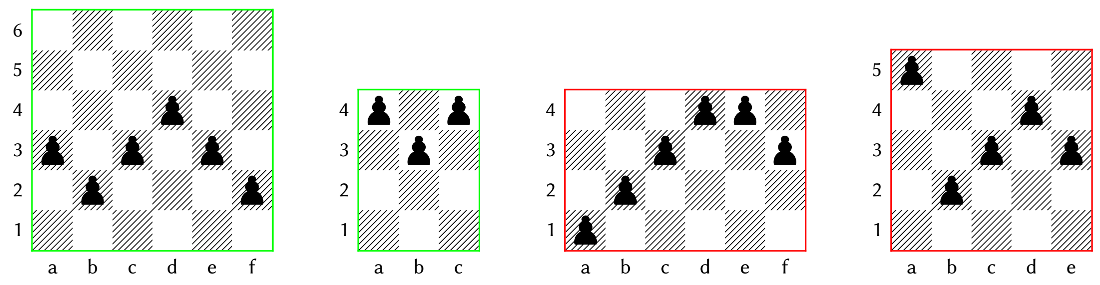
Solution:

Assuming that $k_i > k_{i-1} + d/10$ for all i, we know that $k_i > k_{i-10} + d$, and hence $k_i > k_j + d$ for all $j \le i - 10$. Therefore, the sum in formula (5) contains at most 10 terms $DP[j]$ (and for each of them we can check in constant time whether we should include it or not, i.e., whether $k_i \le k_j + d$). So in this case the computation of the entry $DP[i]$ takes time $O(1)$ for all $0 \le i \le n$, and hence the total runtime is $O(n)$.

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*.

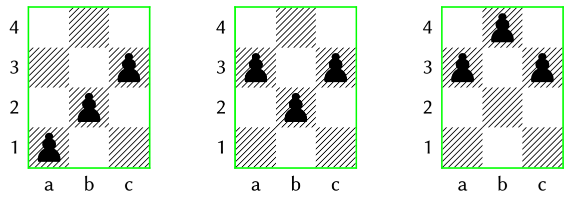
The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.



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 $O(NM)$.

Solution:

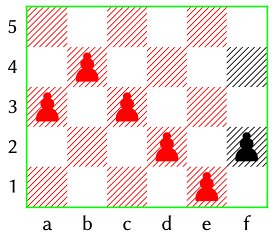
1. *Dimensions of the DP table*: $DP[1 \dots N][1 \dots M]$
2. *Definition of the DP table*: $DP[i][j]$ counts the number of distinct safe pawn lines on an $N \times j$ chessboard with the pawn in the last column located in row i. For example, for $N = 4$, we have $DP[3][3] = 3$, since 3 safe pawn lines on a 4×3 chessboard have their last pawn in row 3, namely:



3. *Computation of an entry*: DP can be computed recursively as follows:

$$\begin{aligned} DP[i][1] &= 1 & 1 \le i \le N \quad (6)\\ DP[1][j] &= DP[2][j-1] & 1 < j \le M \quad (7)\\ DP[N][j] &= DP[N-1][j-1] & 1 < j \le M \quad (8)\\ DP[i][j] &= DP[i-1][j-1] + DP[i+1][j-1] & 1 < i < N, 1 < j \le M \quad (9) \end{aligned}$$

   Equation (6) solves the base case where the chessboard has only one column. In that case, there exists exactly one safe pawn line. Equation (9) provides the general recurrence formula. The rationale behind this formula it is as follows: a pawn line on a $N \times j$ chessboard with its last pawn in row i is obtained by adding a single pawn located at (j, i) (the black pawn on the board below) to a pawn line on a $N \times (j-1)$ chessboard (the red pawns on first board below). Clearly, the last pawn of the smaller line must be on row $i+1$ or $i-1$. Hence, we have $DP[i][j] = DP[i-1][j-1] + DP[i+1][j-1]$. However, this is not true when we have the edge cases $i = 1$ or $i = N$. In these cases, only one position is available for the last pawn of the smaller line, yielding formulae (7) and (8).



4. *Calculation order*: We first compute by order of increasing j, and then in an arbitrary order for i (for example, in increasing order).
5. *Extracting the solution*: The solution is $\sum_{i=1}^{N} DP[i][M]$.
6. *Running time*: The running time of the solution is $O(MN)$, as there are NM entries in the table which are processed in $O(1)$ time, and extracting the solution takes $O(N) \le O(MN)$ time.

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 $f(\text{"}0110001101\underline{111}0001\text{"}) = 3$ 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 **pseudocode** 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.

Solution:

1. *Dimensions of the DP table*: $DP[1 \dots n][0 \dots k+1][0 \dots k+1]$
2. *Definition of the DP table*: Given a string S, let $g(S)$ be the length of the (longest) suffix of "all ones". For example, $g("010\underline{11}") = 2$, $g("010110") = 0$, $g("01101010\underline{111}") = 3$. The entry $DP[len][maks][curr]$ represents the number of binary strings S of length exactly len where $f(S) = maks$ and $g(S) = curr$.
3. *Computation of an entry*: While each entry can be computed directly, in this case it is a bit easier to compute it indirectly. Namely, we take the entire collection of strings $\mathcal{S} = \{S_1, S_2, \dots\}$ represented by some entry $dp[len][maks][curr] = |\mathcal{S}|$ and append "0" to all of them: $\{S_1 + "0", S_2 + "0", \dots\}$. All of them correspond to the entry $dp[len+1][maks][0]$, hence we increase the latter entry by $dp[len][maks][curr]$. Similarly, we append "1" to all of $\mathcal{S}$ and obtain $\{S_1 + "1", S_2 + "1", \dots\}$. All of them correspond to the entry $dp[len+1][\max(maks, curr+1)][curr+1]$, hence we analogously increase that entry by $dp[len][maks][curr]$. The base corresponds when $len = 1$, where the only strings are "0" and "1". Hence, $dp[1][1][1] = 1$ and $dp[1][0][0] = 1$, while $dp[1][1][0] = 0$ and $dp[1][0][1] = 0$.
4. *Calculation order*: The entries can be calculated in order of increasing len. There is no interaction between entries with the same len, hence the order within the same value of len can be arbitrary.
5. *Extracting the solution*: The solution is extracted by summing up over all possible values $curr$ of $g(S)$: $\sum_{curr=0}^{k} dp[n][k][curr]$.
6. *Running time*: The running time of the solution is $O(nk^2)$ as there are $O(nk^2)$ entries in the table, each of which is processed in $O(1)$ time, and the solution is extracted in $O(k) \leq O(nk^2)$.
7. Explicitly write out the full pseudocode.

Algorithm 1

```
1:  Input: integers $n, k$.
2:  Define $dp[1 \dots n][0 \dots k+1][0 \dots k+1]$, initialized to 0.
3:  $dp[1][0][0] \leftarrow 1$
4:  $dp[1][1][1] \leftarrow 1$
5:  for $len \in \{1, \dots, n-1\}$ do
6:      for $maks \in \{0, \dots, k\}$ do
7:          for $curr \in \{0, \dots, k\}$ do
8:              $val \leftarrow dp[len][maks][curr]$
9:              if $val \neq 0$ then                              ▷ Prevents going out-of-bounds.
10:                 (Note: let $a \overset{+}{\leftarrow} b$ be the shorthand for $a \leftarrow a + b$.)
11:                 $dp[len+1][maks][0] \overset{+}{\leftarrow} val$                         ▷ Append 0.
12:                 $dp[len+1][\max(maks, curr+1)][curr+1] \overset{+}{\leftarrow} val$      ▷ Append 1.
13: $sol \leftarrow 0$
14: for $curr \in \{0, 1, \dots, k\}$ do
15:     $sol \leftarrow sol + dp[n][k][curr]$
16: Print(“solution = “, $sol$)
```

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 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 $F_1, \dots, F_n$, computes the length of the longest snake.

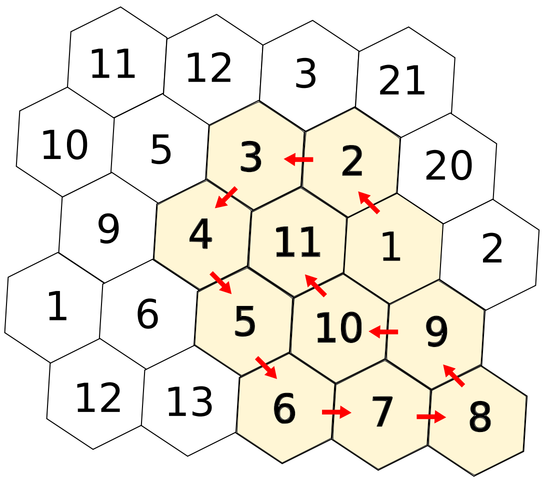


Figure 1: Example of a longest snake.

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.

Solution:

1. *Dimensions of the DP table:* The DP table is linear, its size is n.
2. *Definition of the DP table:* $DP[i]$ is the length of the longest snake with head F_i (that is, the length of the longest snake of the form $(F_{j_1}, \dots, F_{j_{m-1}}, F_i)$).
3. *Computation of an entry:*
$$DP[i] = 1 + \max_{\substack{F_j \in \mathcal{N}(F_i) \\ v_j = v_i - 1}} DP[j].$$
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 define $\max$ in this formula to be 0.
4. *Calculation order:* 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 $j = 1, \dots n-1$.
5. *Extracting the solution:* The output is $\max_{1 \le i \le n} DP[i]$.
6. *Running time:* We compute the order in time $O(n \log n)$ by sorting $v_1, \dots, v_n$. Then each entry can be computed in time $O(1)$ and finally we compute the output in time $O(n)$. 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.

Remark. An alternative solution would be to store the predecessor in a longest snake with head F_i directly in $DP[i]$ (in addition to the length of this longest snake), and store $\emptyset$ if the length of the longest snake is just 1. Then, in order to recover a longest snake, we simply need to find a head of a snake that has maximal length and then follow the sequence of predecessors until we reach an entry $DP[i]$ that has $\emptyset$ as predecessor.

*(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).

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 2 Fill-lengths$(v_1,\dots,v_n)$

```
$S[1],\dots,S[n] \leftarrow 0,\dots,0$
**for** $i = 1,\dots,n$ **do**
    **if** $S[i] = 0$ **then**
        Move-to-tails$(i, S, v_1,\dots,v_n)$
**return** $S$
```

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

Algorithm 3 Move-to-tails$(i, S, v_1,\dots,v_n)$

```
**for** $F_j \in \mathcal{N}(F_i)$ **do**
    **if** $v_j = v_i - 1$ **and** $S[j] = 0$ **then**
        Move-to-tails$(j, S, v_1,\dots,v_n)$
$S[i] = 1 + \max_{\substack{F_j\in\mathcal{N}(F_i)\\ v_j=v_i-1}} S[j]$
```

As in part a), we assume that $\max$ over the empty set is 0. Let us show why this procedure is correct. First, since the algorithm Move-to-tails is recursive, we have to check that it actually finishes. Move-to-tails$(i, S, v_1,\dots,v_n)$ is calling Move-to-tails only for indices j with $v_j < v_i$, and therefore an easy induction on v_j shows that the algorithm will always terminate. We now show the correctness of Move-to-tails$(i, S, v_1,\dots,v_n)$ by induction on v_i.

Base case $v_i = 1$: If $v_i = 1$, then there is no j such that $v_j = v_i - 1$. Therefore, the $\max$ in Move-to-tails$(i, S, v_1,\dots,v_n)$ is empty, so $S[i]$ is set to 1, which is indeed the length of a longest snake with head F_i when $v_i = 1$.

Induction hypothesis: After calling Move-to-tails$(i, S, v_1,\dots,v_n)$ with $v_i = k$, the value of $S[i]$ contains the length of the longest snake with head F_i.

Induction step $k \to k+1$: Let i be an index with $v_i = k+1$. Then for any $F_j \in \mathcal{N}(F_i)$ such that $v_j = v_i - 1$, we have $v_j = k$, so by the induction hypothesis after calling Move-to-tails$(j, S, v_1,\dots,v_n)$ the value of $S[j]$ contains the length of the longest snake with head F_j. Therefore, after setting

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

the value of $S[i]$ indeed contains the length of the longest snake with head F_i.

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).

For the runtime, we will show that for each $i \in \{1,\dots,n\}$ we call Move-to-tails$(i, S, v_1,\dots,v_n)$ exactly once. Indeed, it is called only when $S[i] = 0$, and after the first call of Move-to-tails$(i, S, v_1,\dots,v_n)$ has terminated, we have $S[i] > 0$ by the invariant for the rest of the algorithm. So Move-to-tails$(i, S, v_1,\dots,v_n)$ will not be called a second time after the first call has terminated. While the first call of Move-to-tails$(i, S, v_1,\dots,v_n)$ is running, Move-to-tails is only called for indices

j with $v_j < v_i$, which follows from a very simple induction. So Move-to-tails$(i, S, v_1, \ldots, v_n)$ is also not called a second time while the first call is still running. So we have shown that Move-to-tails$(i, S, v_1, \ldots, v_n)$ is called exactly once for each i. Therefore, the running time is linear in n.

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