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: The DP table is an array of length T.

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

Computation 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 schedules if you have a date on day i plus the number of possible schedules 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 schedules, so the number of possible schedules in this event is $DP[i-1]$. If you have a date on day i, then there needs to be a day during which your best friend is available between your previous date and day i. In particular, if $j < i$ is the last day in which your friend was available before day i, then you can't have a date on any of the days $j, j+1, \ldots, i-1$, and in this case the number of possible schedules is $DP[j-1]$. Therefore,

$$DP[i] = DP[i-1] + DP[\max\{j < i : j \notin K\} - 1], \tag{1}$$

where the second term is replaced by 1 if the maximum is empty (indeed, if your best friend was not available any of the first $i-1$ days, then the only way to have a date on day i is if this is your first date, i.e. all the entries before day i are scheduled with no date). For simplicity, we will simply define $DP[0] = 1$ and say that the maximum in equation (1) is 1 if it is over the empty set.

In order to get the desired $\mathcal{O}(T)$ runtime, we will actually compute the maximum values $M[i] := \max\{j < i : j \notin K\}$ in advance and store them in an array M. First we convert K into an array of length T which has a one in exactly those positions which are blocked, and zeroes elsewhere. In this way, we can decide in constant time whether $j \in K$ for some given date j.

Now we compute M recursively. We initialize with $M[1] = M[2] = 1$. We can then compute the values $M[i]$ from smallest to largest i using

$$M[i] = \begin{cases} M[i-1] & \text{if } i-1 \in K, \\ i-1 & \text{if } i-1 \notin K. \end{cases}$$

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

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

Running time: First, it takes time $\mathcal{O}(T)$ to compute the values $M[i]$ for all $1 \leq i \leq T$. Using these values, it takes time $\Theta(1)$ to compute a new entry in the DP table, since equation (1) becomes

$$DP[i] = DP[i-1] + DP[M[i] - 1].$$

Since there are T entries in the DP table, we therefore need $\Theta(T)$ operations to fill DP. Therefore, the total running time is $\mathcal{O}(T)$.

Remark 1. It is also possible to design an algorithm with $\mathcal{O}(|K| + L)$ runtime, where L is the longest interval without entries in K, i.e., the largest difference between any two consecutive entries in K. This runtime may be better in some situations.

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.

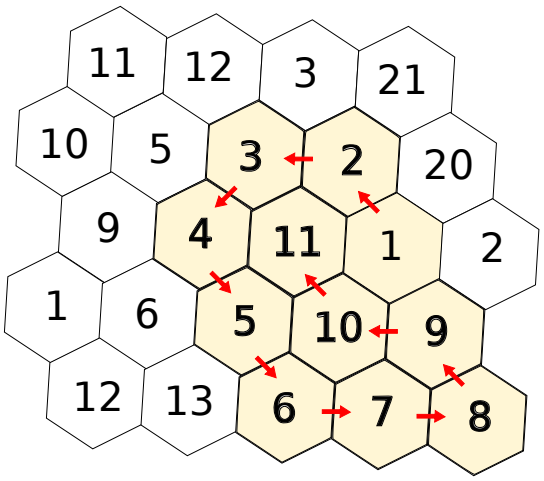

Figure 1: Example of a longest snake.

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: The DP table is linear, its size is n.

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 $(F_{j_1}, \ldots, F_{j_{m-1}}, F_i)$).

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.

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

Extracting the solution: The output is $\max_{1 \leq i \leq n} DP[i]$.

Running time: We compute the order in time $\mathcal{O}(n \log n)$ by sorting $v_1, \ldots, v_n$. 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 $F_1, \ldots F_n$ 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 $DP[j_1] = \max_{1 \leq i \leq n} DP[i]$. If $DP[j_1] \neq 1$, we look at its neigbours and find some F_{j_2} such that $DP[j_2] = DP[j_1] - 1$. If $DP[j_2] \neq 1$, 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 $(F_{j_1}, \ldots, F_{j_m})$.

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 $1 < k \leq m \leq n$ and $\Theta(m)$ time to output the snake.
Remark 2. 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 $\mathcal{O}(n)$ time algorithm that, given a game-board $F_1, \ldots, F_n$, 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 $S[1, \ldots, n]$ 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$(v_1, \ldots, v_n)$

```
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$(i, v_1, \ldots, v_n)$ is:

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

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

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

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

Suppose that you have (infinitely many) coins of different values $x_1, \ldots, x_n \in \mathbb{N}$, 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 $DP[\ldots]$?
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: The DP table is an array of length $a + 1$, indexed by $b = 0, \ldots, a$.

Definition of the DP table: $DP[b]$ is the minimal number of coins required to get amount b (and it is ∞ if it's impossible to get this amount).

Computation of an entry: We initialize the first entry as $DP[0] = 0$ since we can get an amount of 0 using 0 coins.

An amount $b \geq 1$ requires at least one coin, and if we remove this coin (say of value $x_j \leq b$) from (one of) the smallest set of coins that sums to b, then the remaining set will sum to $b - x_j$ and should also be (one of) the smallest sets of coins that does so. Therefore we have

$$DP[b] = 1 + \min_{x_j \leq b} DP[b - x_j],$$

where the minimum is ∞ if it is empty (i.e. if $x_j > b$ for all $j = 1, \ldots, n$).

Calculation order: We can calculate the entries of DP from smallest to largest amount.

Extracting the solution: The minimal number of coins required to get a is then simply $DP[a]$.

Running time: There are $a + 1$ entries in the DP table, and each of them requires $\mathcal{O}(n)$ time to be computed, so the total running time is $\mathcal{O}(an)$.

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.*

Solution:

In order to derive our algorithm we consider a simpler problem: How many ways are there to partition n into numbers that are smaller or equal k? Let's call this number $P(n, k)$. We have $p(n) = P(n, n)$.

Let's consider a few examples. We observe $P(4, 1) = 1$, because there is only one way to partition 4 using only numbers ≤ 1, namely, $1+1+1+1$. As soon as we go to $P(4, 2)$ things get more interesting. There are three ways partitioning 4 using numbers ≤ 2, namely, $1 + 1 + 1 + 1$, $2 + 1 + 1$ and $2 + 2$. Within those, we can distinguish two types of partitions: the two that contain 2s and the one that does not.

Formally, we can write $P(4, 2) = P(4, 1) + P(2, 2)$. Similarly, for arbitrary $n \geq 0$ and $k \geq 1$ we get $P(n, k) = P(n, k-1) + P(n-k, k)$ if $n - k \geq 0$ and $P(n, k) = P(n, k-1)$ otherwise. Additionally, we have $P(0, 0) = 1$ and $P(n, 0) = 0$ for $n \geq 1$. Let's think about P as if it was a two dimensional array of size $(n + 1) \times (n + 1)$. The computation of $P(n, k)$ only depends on elements of the previous column $P(n, k-1)$ and the current column $P(n-k, k)$.

Thus, we can solve the problem by iteratively applying a dynamic program that given the previous column `previous`, with `previous[i]` $= P(i, k-1)$ for $0 \leq i \leq n$, and an integer k, computes the current column by filling an $(n + 1)$-dimensional DP table.

We start by initializing `previous` with `previous[i]` $= P(i, 0)$ for $0 \leq i \leq n$ and we set $k = 1$. We will proceed in rounds, and in the k-th round we re-compute the DP table such that $DP[i] = P(i, k)$. At the end of the round we overwrite `previous` with the new values of the DP-table.

Dimensions of the DP table: The DP table is linear, its size is $n + 1$. The auxiliary table `previous` has the same size and dimension.

Definition of the DP table: Before the k-th round, `previous[i]` $= P(i, k-1)$ is the number of ways to partition i using numbers $\leq k - 1$. After the k-th round, $DP[i] = P(i, k)$ is the number of ways to partition i using numbers $\leq k$.

Computation of an entry: $DP[i] =$ `previous`$[i]$, for $i < k$, and $DP[i] =$ `previous`$[i] + DP[i - k]$, for $i \geq k$.

Calculation order: From left to right.

In each round, after filling the DP table we set `previous` $= DP$ and increase k by one.

Extracting the solution: Once we reached $k = n$, the solution is in $DP[n]$.

Running time: Our DP table has size $n+1$ and the computation of each entry is in $\Theta(1)$. Thus, solving the DP once is in $\Theta(n)$ and solving it n times in $\Theta(n^2)$ as desired.