QUIZ-NACHBESPRECHUNG

In der Vorlesung haben wir um das *Teilsummenproblem* für ein Array $A[1 \dots n]$ zu lösen das folgende *Teilproblem* gesehen:

$$T(i, s) = \text{Ist } s \text{ eine Teilsumme von } A[1 \dots i]?$$

Wahr oder falsch: Eine Implementierung mit dynamischer Programmierung mit diesem Teilproblem wird die Lösung des Teilsummenproblems in Zeit $O(n^C)$ für eine Konstante $C > 0$ berechnen.

O(nS)

Bitte wählen Sie eine Antwort:

- ○ Wahr
- ☒ Falsch

In der Vorlesung haben wir um das *Teilsummenproblem* zu lösen das folgende *Teilproblem* gesehen:

$$T(i, s) = \text{Ist } s \text{ eine Teilsumme von } A[1 \dots i]?$$

Betrachten Sie die folgende Tabelle welche für ein Array $A[1 \dots 3]$ mit dynamischer Programierung basierend auf diesem Teilproblem erstellt wurde:

$T(i,s)$	$s=0$	$s=1$	$s=2$	$s=3$	$s=4$
$i=0$	Wahr	Falsch	Falsch	Falsch	Falsch
$i=1$	Wahr	Falsch	Wahr	Falsch	Falsch
$i=2$	Wahr	Falsch	Wahr	Wahr	Falsch
$i=3$	Wahr	Falsch	Wahr	Wahr	Wahr

Basierend auf dieser Tabelle, welche der folgenden Aussagen sind korrekt? *(Achten Sie auf die Grenzen der Arrays!)*

Wählen Sie eine oder mehrere Antworten:

- ☐ a. 1 ist eine Teilsumme von $A[1 \dots 3]$
- ☒ b. 2 ist eine Teilsumme von $A[1 \dots 3]$
- ☒ c. 3 ist eine Teilsumme von $A[1 \dots 2]$
- ☐ d. 4 ist eine Teilsumme von $A[1 \dots 2]$

In der Vorlesung haben wir um das *Rucksackproblem* zu lösen das folgende Teilproblem gesehen:

$$M(i, w) = \text{Der maximale Profit, der mit den Gegenständen in } A[1 \dots i] \text{ bei einem Gewichtslimit von } w \text{ erzielt werden kann.}$$

Welche der folgenden Rekursionsformeln berechnet den Wert von $M(i, w)$ richtig?

(Unten ist p_i der Profit von Gegenstand i, und w_i ist das Gewicht von Gegenstand i.)

Wählen Sie eine Antwort:

- ○ a. $M(i,w) = \max\{M(i-1,w), \quad p_i + M(i, w-w_i)\}$
- ☒ b. $M(i,w) = \max\{M(i-1,w), \quad p_i + M(i-1, w-w_i)\}$
- ○ c. $M(i,w) = \max\{M(i-1,w-w_i), \quad p_i + M(i-1, w-w_i)\}$

Sei $A[1 \dots 10]$ ein Array von 10 verschiedenen ganzen Zahlen.

Wahr oder falsch: Eine längste aufsteigende Teilfolge in $A[1 \dots 5]$ endet immer in einer strikt kleineren Zahl als eine längste aufsteigende Teilfolge in $A[1 \dots 10]$.

[1,2,3,4,10,6,7,8,9,5]

Bitte wählen Sie eine Antwort:

- ○ Wahr
- ☒ Falsch

In der Vorlesung haben wir um die *längste aufsteigende Teilfolge* zu finden das folgende *Teilproblem* gesehen:

$$M(i, \ell) = \text{Die kleinstmögliche Endung einer aufsteigenden Teilfolge der Länge } \ell \text{ in } A[1 \dots i].$$

Betrachten Sie die folgende Tabelle welche für das Array $A[1 \dots 3]$ mit dynamischer Programierung basierend auf diesem Teilproblem erstellt wurde:

$M(i,\ell)$	$\ell=1$	$\ell=2$	$\ell=3$
$i=1$	4	∞	∞
$i=2$	4	7	∞
$i=3$	4	8	8

Wahr oder falsch: Die obige Tabelle muss einen Fehler enthalten.

Bitte wählen Sie eine Antwort:

- ☒ Wahr
- ○ Falsch

SUBSET SUM

Theory Task T3. (HS21) **/ 9 P**

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry ?
2) *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.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, A and B, and justify your answer.

Size of the DP table / Number of entries: $(n+1) \times (A+1) \times (B+1)$

Meaning of a table entry: $DP[i,j,k] = \begin{cases} 1 & \text{falls es ein Subset } I \subseteq \{1,\ldots,k\} \text{ gibt, so dass } i = \sum_{m \in I} a_m \text{ und } j = \sum_{m \in I} a_m^2 \\ 0 & \text{sonst} \end{cases}$

Computation of an entry (initialization and recursion):

Base Case: $DP[i,j,0] = 0 \quad \forall i,j > 0$

$DP[0,0,k] = 1 \quad \forall k$

Rekursion: $DP[i,j,k] = \underbrace{DP[i-A[k], j-A[k]^2, k-1]}_{\text{falls } i < A[k] \text{ oder } j < A[k]^2 = 0} \vee DP[i,j,k-1]$

Order of computation:

mit steigendem k, Reihenfolge von i und j ist egal

Extracting the result:

$DP[A,B,n]$

Running time: $O(A \cdot B \cdot n)$

KNAPSACK

Problem:

Gewichtslimit W, n Gegenstände mit Gewicht $w_i \in \mathbb{N}$ und Profit $p_i \in \mathbb{N}$ für $1,\dots,n$.

gesucht: Teilmenge $I \subseteq \{1,\dots,n\}$, sodass

(i) $\sum_{i \in I} w_i \leq W$ (Gewichtslimit wird eingehalten)

(ii) $\sum_{i \in I} p_i$ maximal

Alternativansatz:
$DP[i,w]$:= maximaler Profit, der mit Gewichtslimit w und den ersten i Gegenständen erreichbar ist

Teilproblem: $DP[i,p]$ = minimal benötigtes Gewicht, um mit den ersten i Gegenständen mind. Profit p zu erreichen

Rekursion: Base Case: $DP[0,0]=0$, $DP[0,p]=\infty$ für $0 < p \leq \underbrace{\hat{\sum_{i=1}^{n}} p_i}_{\text{max. mögliche Profit}}$

$$DP[i,p] = \min\{G(i-1,p), w_i + G(i-1,p-p_i)\}$$

($G(i-1,p)$: wir nutzen i-ten Gegenstand nicht; $w_i + G(i-1,p-p_i)$: wir nutzen i-ten Gegenstand; falls $p < p_i$: $= \infty$)

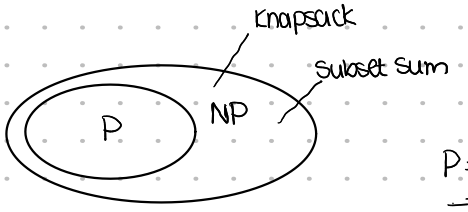
Auslesen der Lösung: Schau in Zeile $i=n$ nach größtem Profit p_0 mit $DP[n,p_0] \leq W$

Laufzeit: $O(n \cdot \hat{P})$ (pseudopolynomiell; $\hat{P}$: maximal erreichbarer Profit)

P vs NP

P := Menge aller Entscheidungsprobleme, die in polynomieller Zeit lösbar sind

NP := Menge aller Entscheidungsprobleme, bei denen man eine mögliche Lösung in polynomieller Zeit lösen kann



Die Probleme Subset Sum und Knapsack sind NP-vollständig, d.h. falls man eine polynomielle Lösung für diese Probleme findet, gilt $P = NP$.

$P = NP$?
→ keiner weiß es
→ Vermutung: $P \neq NP$

APPROXIMATION FÜR KNAPSACK

Idee: Profite runden: $\overline{p_i} := K \cdot \lfloor \frac{p_i}{K} \rfloor$ für ein $K \in \mathbb{N}$

→ runde p_i auf das nächst Vielfache von K ab

DP-Dimensionen: $(n+1) \times (P+1) \longrightarrow (n+1) \times (\frac{P}{K}+1)$

ungerundeter Fall

Laufzeit: $\Theta(nP) \longrightarrow \Theta(n\frac{P}{K})$

Was ist die Qualität dieser Lösung?

$K = 10$

27 $\qquad 20 \leq 27 \leq 30$

- $\overline{p_i} \leq p_i \leq \overline{p_i} + K \quad (1)$ $\quad \hookrightarrow \overline{p_i} \geq p_i - K$
- Gewichte bleiben gleich, es passen gleich viele Gegenstände in den Rucksack
- Aussortieren der Gewichte mit $w_i > W$ in $O(n)$

→ $\sum_{i \in OPT} p_i \geq p_{max} = \max\{p_1, \dots, p_n\}$

→ $\sum_{i \in OPT} p_i \leq \sum_{i=1}^{n} p_i \leq n \cdot p_{max}$

OPT: optimale Lösung für ursprüngliches Problem

$\overline{OPT}$: optimale Lösung für Problem mit gerundeten Profiten

- $\sum_{i \in \overline{OPT}} \overline{p_i} \geq \sum_{i \in OPT} \overline{p_i} \quad (4)$

$$
\begin{aligned}
p(\overline{OPT}) = \sum_{i \in \overline{OPT}} p_i &\overset{(1)}{\geq} \sum_{i \in \overline{OPT}} \overline{p_i} \\
&\overset{(4)}{\geq} \sum_{i \in OPT} \overline{p_i} \\
&\overset{(1)}{\geq} \sum_{i \in OPT} (p_i - K) \\
&= p(OPT) - K \cdot \underbrace{|OPT|}_{\leq n} \quad \leftarrow \text{Anzahl Elemente in OPT} \\
&\geq p(OPT) - K \cdot n
\end{aligned}
$$

- wollen K so wählen, dass $K \cdot n$ im Vergleich zu $p(OPT)$ klein ist

  Parameter $\varepsilon > 0$: $K \cdot n \leq \varepsilon \cdot p_{max} \leq \varepsilon \cdot p(OPT)$

  → $K := \frac{\varepsilon \cdot p_{max}}{n}$

$$
\begin{aligned}
\Rightarrow p(\overline{OPT}) &\geq p(OPT) - K \cdot n \\
&= p(OPT) - \varepsilon \cdot p_{max} \\
&\geq p(OPT) - \varepsilon \cdot p(OPT) \\
&= (1 - \varepsilon) \cdot p(OPT)
\end{aligned}
$$

- Laufzeit für $K = \frac{\varepsilon \cdot p_{max}}{n}$: $\qquad P = \sum_{i=1}^{n} p_i$

$$
\frac{nP}{K} = \frac{n^2 P}{\varepsilon \cdot p_{max}} \leq \frac{n^3 \cdot p_{max}}{\varepsilon \cdot p_{max}} = \frac{n^3}{\varepsilon} \leq O(\tfrac{n^3}{\varepsilon})
$$

($P \leq n \cdot p_{max}$)

- für 99% Approximation: $\varepsilon = 0{,}01$, $O(n^3)$

  für $(1 - \frac{1}{n})$-Approximation: $\varepsilon = \frac{1}{n}$, $O(n^4)$

LONGEST INCREASING SUBSEQUENCE

Problem: finde längste aufsteigende Teilfolge in $A[1...n]$

Teilproblem: $DP[i,\ell]$:= kleinstmögliche Endung einer aufsteigenden Teilfolge der Länge ℓ in $A[1...i]$, falls es keine gibt: ∞

Rekursion: Base Cases: $DP[1,\ell] = \begin{cases} A[1] & \text{falls } \ell = 1 \\ \infty & \text{sonst} \end{cases}$

$$DP[i,\ell] = \begin{cases} \min\{DP[i-1,\ell], A[i]\} & \text{falls } DP[i-1,\ell-1] < A[i] \\ DP[i-1,\ell] & \text{sonst} \end{cases}$$

Beispiel: $A = [3,7,8,4,5]$

DP[i,ℓ]	1	2	3	4	5
1	3	∞	∞	∞	∞
2	3	7	∞	∞	∞
3	3	7	8	∞	∞
4	3	4	8	∞	∞
5	3	4	5	∞	∞

Optimierungen:

→ eindimensionales DP-Array
→ Suchen des zu ändernden Elements mit BinarySearch
→ Laufzeit: $O(n \log n)$, Speicher $O(n)$

Zeile $i \to i+1$: einzige Änderung ist
$DP[i+1,\ell]$ falls $DP[i,\ell-1] < A[i+1] \leq DP[i,\ell]$

Theory Task T3. FS21 / 11 P

Given a square matrix $M \in \mathbb{N}^{n \times n}$ of non-negative integers, with $n \geq 1$, the goal of this task is to compute the *longest snake* within it. In Figure 1 the top-left element is M_{11}, the rows are indexed by the first index and the columns by the second. M_{ij} denotes the jth entry in the ith row.

A *snake* is a sequence of entries of M, i.e., $s = (s_1, \ldots, s_k)$ with $s_\ell = M_{i_\ell j_\ell}$ for $i_\ell, j_\ell \in \{1, \ldots, n\}$ and $\ell \in \{1, \ldots, k\}$, that satisfies that the next entry s_{m+1} is either below the current one s_m (i.e., $i_{m+1} = i_m + 1$ and $j_{m+1} = j_m$) or to its left (i.e., $i_{m+1} = i_m$ and $j_{m+1} = j_m - 1$). This means that there are no cycles. Additionally, the corresponding matrix entries are required to differ exactly by 1, i.e., s satisfies $|s_m - s_{m+1}| = 1$, for all $m \in \{1, \ldots, k-1\}$. The *length* of a snake $s = (s_1, \ldots, s_k)$ is equal to its sequence length, which in this case is k. Figure 1 depicts an example of a longest snake of length 8.

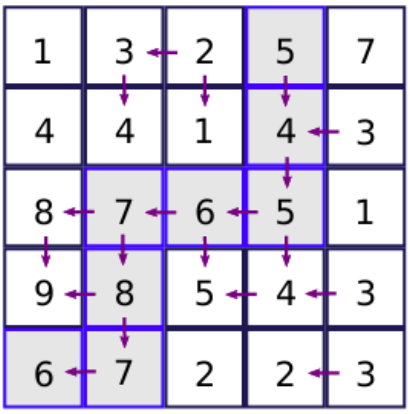


Figure 1: Sample matrix M and corresponding longest snake. The purple arrows indicate eligible ways of creating a snake. Note that multiple longest snakes are possible, e.g., $(3, 4, 5, 6, 7, 8, 7, 6)$ would be another snake with the same length as the highlighted one $(5, 4, 5, 6, 7, 8, 7, 6)$.

Use dynamic programming to design an algorithm that, given a square matrix M of non-negative integers, computes a longest snake. In case M contains multiple snakes of maximal length, it suffices if your algorithm computes only one of them.

In order to achieve full points, your solution must run in $\mathcal{O}(n^2)$ time and address the following aspects:

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

2) *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.

3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?

4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?

5) *Running time*: What is the running time of your algorithm? Provide it in Θ-notation in terms of n, and justify your answer.

1) $n \times n$

2) $DP[i,j]$:= max. Länge einer Schlange, die in A_{ij} endet

3) $$DP[i,j] = \begin{cases} DP[i][j+1]+1 & \text{falls ① und nicht ②} \\ DP[i-1][j]+1 & \text{falls ② und nicht ①} \\ \min\{DP[i][j+1], DP[i-1][j]\}+1 & \text{falls ① und ②} \\ 1 & \text{falls weder ① noch ②} \end{cases}$$

Fall ①: $|A_{i(j+1)} - A_{ij}| = 1$

Fall ②: $|A_{(i-1)j} - A_{ij}| = 1$

Base Case: $DP[1][n] = 1$

4) maximales Element k in DP-Tabelle bei $DP[i,j]$

$s_k = A_{ij}$

$$s_{k-1} = \begin{cases} A_{(i-1)j} & |A_{ij} - A_{(i-1)j}| = 1 \quad \text{und} \quad DP[i,j] = DP[i-1,j]+1 \\ A_{i(j+1)} & |A_{ij} - A_{i(j+1)}| = 1 \quad \text{und} \quad DP[i,j] = DP[i,j+1]+1 \end{cases}$$

5) n^2 Einträge, Berechnung in $O(1)$ $\Rightarrow \Theta(n^2)$

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

Given a binary string $S \in \{0,1\}^n$ of length n, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{“}\underline{11}10\underline{1}1\text{”}) = 3$ because the string contains three 1 that are followed by another 1 (underlined). Given n and k, the goal is to count the number of binary strings S of length n with $f(S) = k$.

Describe a DP algorithm that, given positive integers n and k with $k < n$, reports the required number. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(nk)$.

Hint: *Define a three dimensional DP table* $DP[1 \dots n][0 \dots k][0 \dots 1]$.

Hint: *The entry* $DP[i][j][l]$ *counts the number of strings of length* i *with* j *occurrences of* "11" *that end in* l *(where* $1 \leq i \leq n, 0 \leq j \leq k$ *and* $0 \leq l \leq 1$*).*

1) s. Hint

2) Base Case: DP[1][0][l] = 1 für $l \in \{0,1\}$

DP[1][j][l] = 0 für $l \in \{0,1\}$ und $1 \leq j \leq k$

Rekursion: DP[i][j][0] = DP[i-1][j][0] + DP[i-1][j][1]

$$DP[i][j][1] = \begin{cases} DP[i-1][j][0] & \text{falls } j = 0 \\ DP[i-1][j][0] + DP[i-1][j-1][1] & \text{sonst} \end{cases}$$

3) Berechnungsreihenfolge:

mit steigendem i, bei gleichem Wert von i ist Reihenfolge zwischen j und k nicht relevant

4) Lösung = DP[n][k][0] + DP[n][k][1]

5) O(nk)