Quiz

In a linked list data structure L, suppose you have a pointer to an object o in L. Then inserting a key k into the list after o (i.e. `insertAfter`(o, k, L)) takes $O(1)$ time.

- (●) True
- ( ) False

programming 2

12 Minuten

Let T be a 2-3 tree with depth 4. Let x be the number of leaves. It is possible for x to be equal to:

$2^4 \leq$ number of leaves $\leq 3^4$

$2^4 = 16$, $3^4 = 81$

True	False	
	X	5
	X	10
X		20
X		50
	X	100
	X	200
	X	1000

Let $B1, B2$ be two leaves in a 2-3-tree. If an insert operation increases the depth of $B1$, then it also increases the depth of $B2$.

- (●) True
- ( ) False

alle Blätter sind immer auf dem gleichen Level

We can find the maximum key of a 2-3 tree with n leaves in time $O(\log(n))$.

True

False

Consider the subset sum problem with input $A[1 \dots n]$ and target value b, where the subproblem $T(i, s)$ denotes whether s is a subset sum of $A[1 \dots i]$. Fill in X such that the recursion is $T(i, s) = T(i-1, s) \vee T(i-1, X)$ (for $2 \leq i \leq n$). If $X < 0$ then $T(i-1, X)$ is considered to be false.

a. $X = A[i-1]$

b. $X = s - A[i]$

c. $X = s - b$

d. $X = b$

There is a known algorithm which solves subset sum in time $O(n^3)$.

True

False

Consider the knapsack problem with weights w_i, profits p_i and weight limit W. Let $P = p_1 + \dots + p_n$. Which of the following statements are true/false:

True	False			
X		There is an algorithm solving the problem in time $O(nW)$.		
X		There is an algorithm solving the problem in time $O(nP)$.		

There is an algorithm for knapsack which computes in polynomial time a solution which is at least half as good as the optimal solution.

(True)

False

$(1-\varepsilon)$-Approximationsalgorithmus in $O(n^3 \cdot \varepsilon^{-1})$

wähle $\varepsilon = \frac{1}{2}$

For the knapsack problem, let OPT be the optimal solution for the original profits p_i and $\widetilde{OPT}$ the optimal solution for the rounded profits $\tilde{p}_i$. Which of the following is always true?

(a.) $\sum_{i \in \widetilde{OPT}} \tilde{p}_i \geq \sum_{i \in OPT} \tilde{p}_i$.

b. $\sum_{i \in \widetilde{OPT}} \tilde{p}_i = \sum_{i \in OPT} \tilde{p}_i$.

c. $\sum_{i \in \widetilde{OPT}} \tilde{p}_i \leq \sum_{i \in OPT} \tilde{p}_i$.

d. None of the above.

Let L be a longest ascending subsequence of $A[1 \ldots i]$ and L' be a longest ascending subsequence of $A[1 \ldots i+1]$. Then L' is either identical to L, or L' is obtained from L by adding $A[i+1]$ at the end.

True

(False)

Counterexample: $A[1,3,2]$

$L = 1,3$

$L' = 1,2$

- Neue Gruppen

Wie Teilproblem finden?

Drei Kategorien:

Optimierungsproblem:

→ Was ist der maximale/minimale/kleinste/grösste/höchste ..

Entscheidungsproblem:

→ Gibt es .. sodass ...

Berechnung: (selten)

→ Was ist die Anzahl ...

Anz. Pins der Länge ≤ n

DP[1] = 10

DP[i] = 10 DP[i-10]

Lösung: $\sum_{i=1}^{n} DP[i]$

Zusätzliche Bedingungen die das Problem einfacher machen

Arrays, Strings, Mengen, alles was man indexieren kann:

→ ... wenn wir nur die ersten i Elemente betrachten

→ ... wenn wir nur die ersten i Elemente betrachten und das i-te Element gewählt werden muss

→ ... wenn wir bei i beginnen

→ ... wenn wir bei i beginnen und das i-te Element gewählt werden muss

→ ... wenn wir bei i beginnen und j enden

→ ... wenn wir bei i beginnen und höchstens/genau Länge ℓ haben

Geld, Zeit, Tickets, und alles was man ausgeben kann:

→ ... und wir mit t Zeit/Geld/etc starten

Grenzen, Limits die man einhalten muss z.B. Gewichtslimit, Zeitlimit

→ ... und wir Limit ℓ einhalten

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$\begin{aligned} A_1 &= 1 \\ A_2 &= 2 \\ A_3 &= 3 \\ A_4 &= 4 \\ A_n &= A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5. \end{aligned}$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

Solution:

Algorithm 1 $A(n)$

```
if n ≤ 4 then
    return n
else
    return A(n − 1) + A(n − 3) + 2A(n − 4)
```

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

Solution:

Algorithm 2 Compute A_n using memoization

```
memory ← n-dimensional array filled with (−1)s
function A_Mem(n)
    if memory[n] ≠ −1 then                ▷ If A_n is already computed.
        return memory[n]
    if n ≤ 4 then
        memory[n] ← n
        return n
    else
        A_n ← A_Mem(n − 1) + A_Mem(n − 3) + 2A_Mem(n − 4)
        memory[n] ← A_n
        return A_n
```

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. Address the following aspects in your solution:

- **Dimensions of the DP table:** The DP table is linear, its size is n.
- **Definition of the DP table:** $DP[k]$ contains A_k for $1 \leq k \leq n$.
- **Calculation of an entry:** Initialize $DP[1]$ to 1, $DP[2]$ to 2, $DP[3]$ to 3 and $DP[4]$ to 4. The entries with $k \geq 5$ are computed by $DP[k] = DP[k-1] + DP[k-3] + 2DP[k-4]$.
- **Calculation order:** We can calculate the entries of DP from smallest to largest.
- **Reading the solution:** All we have to do is read the value at $DP[n]$.
- **Run time:** Each entry can be computed in time $\Theta(1)$, so the run time is $\Theta(n)$.

Subset Sum ($\neq$ MSS)

Gibt es eine Teilmenge an Indices $I \subseteq \{1 \dots n\}$ sodass $\sum_{i \in I} A[i] = b$

Dimension: $T[0..n][0 \dots s]$

Teilproblem: $T[i,s] = \begin{cases} 1 & \text{falls es eine Teilmenge } I \subseteq \{1..i\} \text{ gibt sodass } \sum_{j \in I} A[j] = s \\ 0 & \text{sonst} \end{cases}$

Base Case: $T[0,0] = 1$

$T[0,s] = 0 \quad \forall 1 \leq s \leq b$

Rekursion: $T[i,s] = \underbrace{T[i-1,s]}_{\text{wir nehmen } A[i] \text{ nicht}} \vee \overbrace{T[i-1, s-A[i]]}^{=0 \text{ falls } A[i] > s}$ $\quad \forall 1 \leq i \leq n \; \forall 0 \leq s \leq b$

wir nehmen $A[i]$

Berechnungsreihenfolge: For $i = 0 \dots n$:

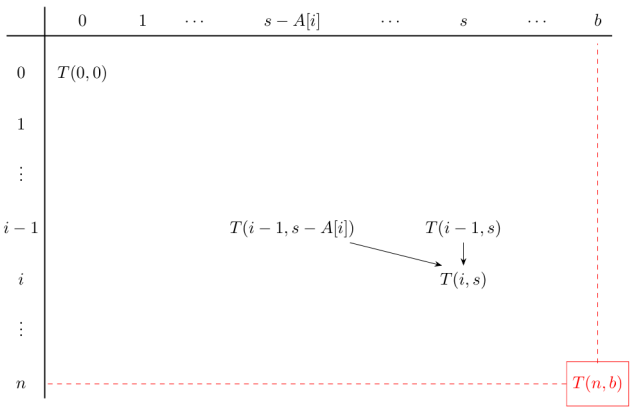
For $s = 0 \dots b$:

compute $T[i,s]$

Lösung: $T[n,b]$ sagt ob eine Lösung existiert,

falls ja dann machen wir backtracking

um I zu finden



Laufzeit: $O(n\,b)$

z.B. $b = n, b = n^4$ — z.B. $b = 2^n, b = n\,2^n$

Pseudopolynomiell: Ob der Algorithmus in polynomieller oder exponentieller Zeit läuft hängt von der Eingabe b ab

Rucksackproblem

Gewichtslimit W, n Gegenstände mit Gewicht w_i und Profit p_i

Finde eine Teilmenge $I \subseteq \{1 \dots n\}$ mit $\sum_{i \in I} w_i \leq W$ sodass $\sum_{i \in I} p_i$ maximal ist

Dimension: $P[0 \dots n][0 \dots W]$

Teilproblem: $P[i, w]$ = max Profit wenn wir nur die ersten i Elemente betrachten und Gewichtslimit w einhalten

Base Case: $P[0, w] = 0 \quad \forall 0 \leq w \leq W$

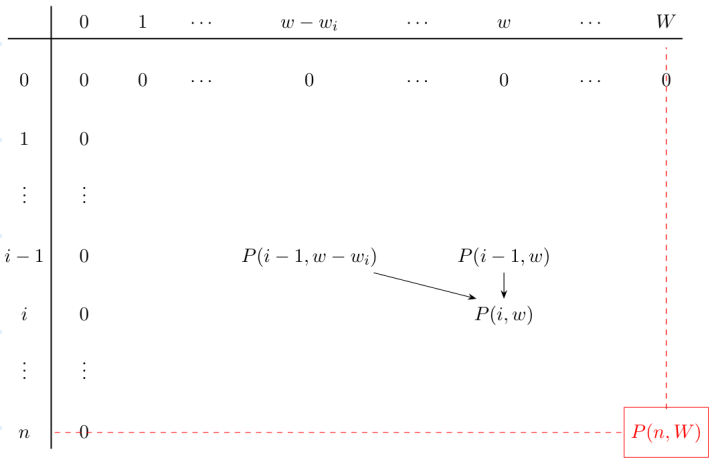
Rekursion: $P[i, w] = \max \{ P[i-1, w] , \overbrace{P[i-1, w - w_i] + p_i}^{= 0 \text{ falls } w < w_i} \}$ $\quad \forall 1 \leq i \leq n \; \forall 0 \leq w \leq W$

$P[i-1, w]$: wir nehmen Gegenstand i nicht

$P[i-1, w - w_i] + p_i$: wir nehmen Gegenstand i

Berechnungsreihenfolge:
For $i = 0 \dots n$:
  For $w = 0 \dots W$:
    compute $P[i, w]$

Lösung: $P[n, W]$ sagt ob eine Lösung existiert, falls ja dann machen wir backtracking um I zu finden



Laufzeit: $O(n \cdot W)$, pseudopolynomiell

Alternative Lösung

Dimension: $G[0 \dots n][0 \dots P]$ wobei $P = \sum_{i=1}^{n} p_i$

Teilproblem: $G[i, p]$ = minimales Gewicht, wenn wir nur die ersten i Gegenstände betrachten, um Profit p zu erreichen (oder überschreiten).
($= \infty$ falls dies nicht möglich ist)

Base Case: $G[0, 0] = 0$

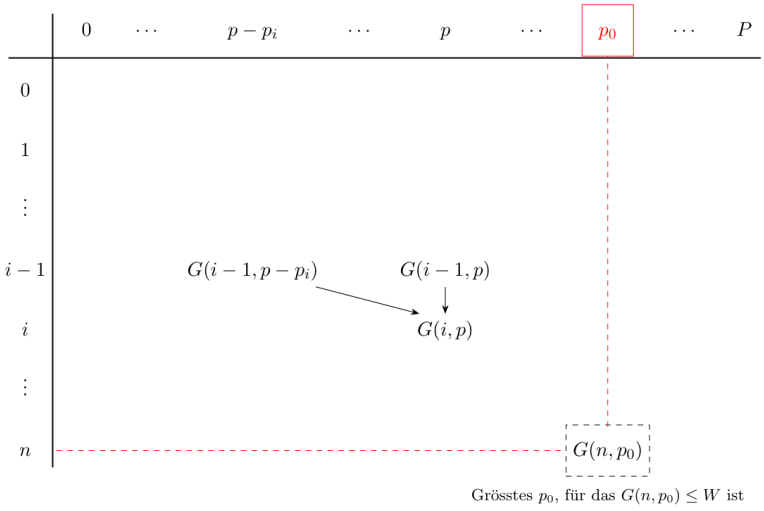
$G[0, p] = \infty \quad \forall 1 \leq p \leq P$

Rekursion: $G[i, p] = \min \{ G[i-1, p], \underbrace{G[i-1, p-p_i] + w_i}_{= 0 \text{ falls } p < p_i} \}$

- $G[i-1, p]$: wir nehmen Gegenstand i nicht
- $G[i-1, p-p_i] + w_i$: wir nehmen Gegenstand i

Berechnungsreihenfolge: For $i = 0 \dots n$:
For $p = 0 \dots P$:
compute $G[i, p]$

Lösung: $\max \{ p_0 \mid G[n, p_0] \leq W \}$



Grösstes p_0, für das $G(n, p_0) \leq W$ ist

Laufzeit: $O(n \cdot P)$, pseudopolynomiell

Längste aufsteigende Teilfolge

Gesucht ist die Länge einer LAT in einem Array $A[1..n]$

Dimension: $M[1...n][1...n]$

Teilproblem: $M[i,\ell]$ = kleinstmögliche Endung einer AT der Länge ℓ im Bereich $A[1..i]$

(= ∞ falls es keine solche Teilfolge gibt)

Base Case: $M[1,1] = A[1]$

$M[1,\ell] = \infty \quad \forall 2 \le \ell \le n$

Rekursion: $M[i,\ell] = \min \begin{cases} \underbrace{\infty}_{A[i] \text{ anhängen}} & \text{falls } A[i] \le M[i-1,\ell-1] \quad \text{(können wir } A[i] \text{ anhängen?)} \\ M[i-1,\ell] & \rightarrow A[i] \text{ nicht verwenden} \end{cases}$ $\quad \forall 2 \le i \le n \; \forall 1 \le \ell \le n$

Berechnungsreihenfolge:
```
For i = 1...n:
    For ℓ = 1...n:
        compute M[i,ℓ]
```

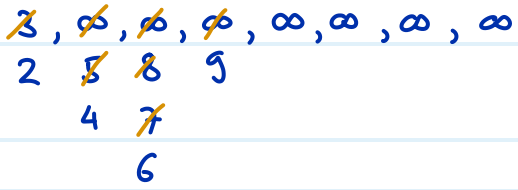
Lösung: $\max\{\ell \mid M[n,\ell] < \infty\}$

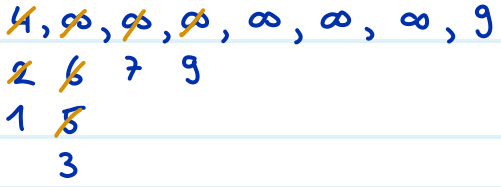
Laufzeit: $O(n^2)$

Verbesserung in $O(n \log n)$ mit Binary Search

Idee: Es verändert sich nur ein DP Eintrag in jeder Reihe

Bsp: $A = [3,5,4,8,9,7,2,6]$ $\quad B = [4,6,2,7,5,1,3,9]$





~ 20 min Zeit

/ 8 P von 100 Punkten

Theory Task T3.

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). Address the following aspects in your solution:

↳ Hinweis für die Dimension!

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 $\max A$, and justify your answer.

↳ immer wenn ihr das lest müsst ihr mindestens einen Satz schreiben

Dimension: $DP[0..n][0..\max A]$

Teilproblem: $DP[i,k] = \begin{cases} 1 & \text{falls es eine Teilmenge } I \subseteq \{1 \ldots i\} \text{ gibt mit } \sum_{j \in I} a_j = k \\ 0 & \end{cases}$

Base Case: $DP[0,0] = 1$

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

Rekursion: $DP[i,k] = \underbrace{DP[i-1,k]}_{\text{wir nehmen } a_i \text{ nicht}} \vee \overbrace{DP[i-1,k-a_i]}^{=0 \text{ falls } a_i > k}$ (wir nehmen a_i)

$\forall 1 \le i \le n \; \forall 0 \le k \le \max A$

Berechnungsreihenfolge: For $i = 0 \dots n$:
For $k = 0 \dots \max A$:
compute $DP[i, k]$

Lösung: $\begin{cases} 1 & \text{falls } \forall 2 \leq i \leq n \cdot DP[i-1, a_i] = 1 \\ 0 & \text{sonst} \end{cases}$

Laufzeit: $\Theta(n \max A)$ pseudopolynomiell

You are a history student, "Alex," standing at the entrance of the "Grand Boulevard of Museums" (museum 0). You have a research paper due tomorrow, and your goal is to acquire the maximum amount of "knowledge" possible.

You have two limited resources:

- A starting budget of M dollars.
- A total time limit of T hours.

The boulevard has N museums, numbered 0 to $N-1$. For each museum i, you are given three pieces of information:

- `ticket_cost[i]` : The cost in dollars to buy a ticket.
- `visit_time[i]` : The time in hours it takes to see the exhibit.
- `knowledge[i]` : The knowledge points (KP) you gain from the exhibit.

The Rules of the Tour

When you are at the entrance of museum i, you have two choices:

1. **Visit the Museum:**
   - You **can only** do this if you have enough money (`>= ticket_cost[i]`) AND enough time (`>= visit_time[i]`).
   - If you visit, you pay the `ticket_cost[i]` , spend the `visit_time[i]` , and gain the `knowledge[i]` .
   - After your visit, you immediately move to the front of the next museum ($i+1$).
2. **Skip the Museum:**
   - You **can only** do this if you have at least **1 hour** of time remaining.
   - Skipping costs you no money, but it costs **1 hour** of time to walk to the next museum.
   - You gain 0 knowledge.
   - You move to the front of the next museum ($i+1$).

Your tour ends when you have considered the last museum (i.e., you arrive at index N) or when you run out of time and cannot even afford to skip (i.e., you have 0 time left).

What is the absolute maximum amount of knowledge you can gain?

Example

- $N = 2$ museums
- $M = 10$ (dollars)
- $T = 5$ (hours)

Museum (i)	ticket_cost[i]	visit_time[i]	knowledge[i]
0	7	3	10
1	4	2	8

Optimal Path Analysis

1. **At Museum 0:** Alex has $M = 10, T = 5$.
   - *Choice 1 (Visit):* Cost=7, Time=3. Alex can afford this.
     - New state: $M = 3, T = 2$. Knowledge=10. Move to Museum 1.
   - *Choice 2 (Skip):* Cost=0, Time=1. Alex can afford this.
     - New state: $M = 10, T = 4$. Knowledge=0. Move to Museum 1.
2. **Path 1 (Visit Museum 0):**
   - **At Museum 1:** Alex has $M = 3, T = 2$.
   - *Choice 2 (Skip):* Cost=0, Time=1. Alex can afford this.
     - New state: $M = 3, T = 1$. Knowledge=0. Move to end.
   - **Total Knowledge (Path 1):** $10 + 0 = 10$.
3. **Path 2 (Skip Museum 0):**
   - **At Museum 1:** Alex has $M = 10, T = 4$.
   - *Choice 1 (Visit):* Cost=4, Time=2. Alex **can** afford this.
     - New state: $M = 6, T = 2$. Knowledge=8. Move to end.
   - *Choice 2 (Skip):* Cost=0, Time=1. Alex can afford this.
     - New state: $M = 10, T = 3$. Knowledge=0. Move to end.
   - **Total Knowledge (Path 2):** $0 + 8 = 8$.

Comparing the two possible complete tours, the max knowledge is **10**.

Dimension: $(N+1) \times (T+1) \times (M+1)$

Teilproblem: $DP[i,t,m]$ = max knowledge Alex can gain starting at museum i, with t hours and m dollars left

Base Case: $DP[N,t,m] = 0 \quad \forall 0 \le t \le T \; \forall 0 \le m \le M$

Rekursion:

$$DP[i,t,m] = \max \begin{cases} DP[i+1, t-1, m] & \text{weitergehen} \\ DP[i+1, t-visit\_time[i], m-ticket\_cost[i]] + knowledge[i] & \text{Museum } i \text{ besuchen} \end{cases}$$

only possible if ≥ 0 (für $t - visit\_time[i]$ und $m - ticket\_cost[i]$)

Berechnungsreihenfolge:

For $i = N \dots 0$:
  For $t = 0 \dots T$:
    For $m = 0 \dots M$:
      compute $DP[i,t,m]$

Lösung: $DP[0,T,M]$

Laufzeit: $O(N \cdot T \, M)$

```java
public static int solve(int N, int M, int T, int[] ticketCost, int[] visitTime, int[] knowledge) {
  DP = new int[N + 1][T + 1][M + 1];
  // Base Case: automatisch gemacht weil alle Einträge auf 0 initialisiert werden

  for (int i = N - 1; i >= 0; i--) {
      for (int t = 0; t <= T; t++) {
          for (int m = 0; m <= M; m++) {
              // Rekursion:
              int knowledgeFromSkip = 0;
              if (t >= 1) knowledgeFromSkip = DP[i + 1][t - 1][m];
              int knowledgeFromVisit = 0;
              if (t >= visitTime[i] && m >= ticketCost[i]) {
                  knowledgeFromVisit = DP[i + 1][t - visitTime[i]][m - ticketCost[i]] + knowledge[i];
              }
              DP[i][t][m] = Math.max(knowledgeFromSkip, knowledgeFromVisit);
          }
      }
  }
  // Lösung:
  return DP[0][T][M];
}
```

```java
public static int solve2(int N, int M, int T, int[] ticketCost, int[] visitTime, int[] knowledge) {
  DP = new int[N + 1][T + 1][M + 1];
  for (int i = 0; i<DP.length; i++) {
    for (int j = 0; j<DP[0].length; j++) {
      for (int k = 0; k<DP[0][0].length; k++) DP[i][j][k] = -1;
    }
  }
  // Lösung:
  return compute(0, T, M);
}


public static int compute(int i, int t, int m) {
  // Base Case:
  if (i >= N) return 0;

  // Memoization:
  if (DP[i][t][m] != -1) return DP[i][t][m];

  // Rekursion:
  int knowledgeFromSkip = 0;
  if (t >= 1) knowledgeFromSkip = compute(i + 1, t - 1, m);

  int knowledgeFromVisit = 0;
  if (t >= visitTime[i] && m >= ticketCost[i]) { // check if we have enough time and money
    knowledgeFromVisit = knowledge[i] + compute(i + 1, t - visitTime[i], m - ticketCost[i]);
  }
  // Store result in DP table:
  DP[i][t][m] = Math.max(knowledgeFromSkip, knowledgeFromVisit);
  return DP[i][t][m];
}
```

Given a binary string $S \in \{0,1\}^n$ of length n, let $g(S)$ be the **positional sum** of the string. We define the positional sum as the sum of all 1-based indices i such that the i-th character of S is a '1'.

For example, if $S = \text{"}01011\text{"}$:

- The '1's are at indices 2, 4, and 5.
- $g(S) = 2 + 4 + 5 = 11$.
- For $S = \text{"}1000\text{"}$, $g(S) = 1$.
- For $S = \text{"}0000\text{"}$, $g(S) = 0$.

Given n and k, the goal is to count the number of binary strings S of length n with $g(S) = k$.

Describe a DP algorithm that, given positive integers n and k, reports the required number. Your solution should have a time complexity of at most $O(nk)$.

Dimension: DP[0...n][0...k]

Teilproblem: DP[i,j] = number of binary Strings of length i with positional sum = j

Base Case: DP[0,0] = 1

DP[0,j] = 0 $\forall 1 \le j \le k$

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

(DP[i-1,j]: wir fügen eine 0 hinzu; DP[i-1,j-1]: wir fügen eine 1 hinzu; = 0 falls j-i<0)

Berechnungsreihenfolge:
For i = 0...n:
  For j = 0...k:
    compute DP[i,j]

Lösung: DP[n,k]

Laufzeit: O(n k)