Dynamische Programmierung (DP)

am Beispiel des Wechselgeldproblems

Lukas Rost

13. Mai 2019

Das Problem

Das Wechselgeldproblem

- Philip steht in der Schlange vor der Kasse im Supermarkt...
- ...und möchte den Rentnern vor ihm helfen, das Kleingeld herauszusuchen.
- **Problem:** Rentner brauchen viel Zeit, um eine Münze herauszusuchen.
- Philip will Betrag so aufteilen, dass Rentner möglichst wenige Münzen brauchen.

Das Problem

Das Wechselgeldproblem

- Philip steht in der Schlange vor der Kasse im Supermarkt...
- ...und möchte den Rentnern vor ihm helfen, das Kleingeld herauszusuchen.
- **Problem:** Rentner brauchen viel Zeit, um eine Münze herauszusuchen.
- Philip will Betrag so aufteilen, dass Rentner möglichst wenige Münzen brauchen.

Das Problem

Das Wechselgeldproblem

- Philip steht in der Schlange vor der Kasse im Supermarkt...
- ...und möchte den Rentnern vor ihm helfen, das Kleingeld herauszusuchen.
- **Problem:** Rentner brauchen viel Zeit, um eine Münze herauszusuchen.
- Philip will Betrag so aufteilen, dass Rentner möglichst wenige Münzen brauchen.

Das Problem

Das Wechselgeldproblem

- Philip steht in der Schlange vor der Kasse im Supermarkt...
- ...und möchte den Rentnern vor ihm helfen, das Kleingeld herauszusuchen.
- **Problem:** Rentner brauchen viel Zeit, um eine Münze herauszusuchen.
- Philip will Betrag so aufteilen, dass Rentner möglichst wenige Münzen brauchen.

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \ldots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \ldots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \ldots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \dots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \dots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \dots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \ldots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \ldots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \ldots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \ldots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \ldots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \ldots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \ldots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \ldots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \ldots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Das Problem

Das Wechselgeldproblem

Formaler:

- Münzwerte: $w = \{w_1, \ldots, w_n\}$ mit $w_i \in \mathbb{N}$ und $w_1 = 1$
- in Deutschland üblicherweise $w = \{1, 2, 5, 10, \ldots\}$
- Rechnungssumme: W
- Anzahl der einzelnen genutzten Münzen: $x = \{x_1, \ldots, x_n\}$
- Gesamtanzahl der Münzen zu minimieren:

$$\min \sum_{j=1}^{n} x_j$$

- Bedingung:

$$\sum_{j=1}^{n} w_j x_j = W$$

Dynamische Programmierung

DP als Problemlösungsmethode

- **Erfinder:** Richard Bellman (1950er)
- Vereinfachen eines komplexen Problems durch Zerlegung in einfache Teilprobleme
- ...unter Nutzung von Rekursion

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

Bedingungen für DP

Optimale Substruktur:

- Lösung eines Optimierungsproblems kann durch Kombination optimaler Teillösungen gefunden werden
- *Optimalitätsprinzip von Bellman*
- *Beispiel:* Dijkstra

Überlappende Subprobleme:

- nur wenige Subprobleme
- rekursive Lösung würde die gleichen Subprobleme immer wieder lösen
- aber DP löst jedes nur ein Mal!

Dynamische Programmierung

DP-Begriffe

- *DP-Funktion:* $\mathfrak{f} : A \to B$
- *Zustandsraum: A*
  Jeder Zustand wird durch Werte der DP-Variablen $x_1, \ldots, x_n$ beschrieben
- *Werteraum: B*
- $\mathfrak{f}$ sollte rekursiv berechenbar sein, z.B. Fibonaccizahlen:

$$\mathfrak{f}(n) = \begin{cases} 1, & n < 2 \\ \mathfrak{f}(n-1) + \mathfrak{f}(n-2), & n \geq 2 \end{cases}$$

Dynamische Programmierung

DP-Begriffe

- *DP-Funktion:* $\mathfrak{f} : A \to B$
- *Zustandsraum: A*
  Jeder Zustand wird durch Werte der DP-Variablen $x_1, \ldots, x_n$ beschrieben
- *Werteraum: B*
- $\mathfrak{f}$ sollte rekursiv berechenbar sein, z.B. Fibonaccizahlen:

$$\mathfrak{f}(n) = \begin{cases} 1, & n < 2 \\ \mathfrak{f}(n-1) + \mathfrak{f}(n-2), & n \geq 2 \end{cases}$$

Dynamische Programmierung

DP-Begriffe

- *DP-Funktion:* $\mathfrak{f} : A \to B$
- *Zustandsraum: A*
  Jeder Zustand wird durch Werte der DP-Variablen $x_1, \ldots, x_n$ beschrieben
- *Werteraum: B*
- $\mathfrak{f}$ sollte rekursiv berechenbar sein, z.B. Fibonaccizahlen:

$$\mathfrak{f}(n) = \begin{cases} 1, & n < 2 \\ \mathfrak{f}(n-1) + \mathfrak{f}(n-2), & n \geq 2 \end{cases}$$

Dynamische Programmierung

DP-Begriffe

- *DP-Funktion:* $\mathfrak{f} : A \to B$
- *Zustandsraum: A*
  Jeder Zustand wird durch Werte der DP-Variablen $x_1, \ldots, x_n$ beschrieben
- *Werteraum: B*
- $\mathfrak{f}$ sollte rekursiv berechenbar sein, z.B. Fibonaccizahlen:

$$\mathfrak{f}(n) = \begin{cases} 1, & n < 2 \\ \mathfrak{f}(n-1) + \mathfrak{f}(n-2), & n \geq 2 \end{cases}$$

Dynamische Programmierung

Berechnung der DP-Funktion

- **Memoization:** Speichern der Ergebnisse von Funktionsaufrufen
- geringere Laufzeit, aber dafür höherer Speicherverbrauch

Dynamische Programmierung

Berechnung der DP-Funktion

- **Memoization:** Speichern der Ergebnisse von Funktionsaufrufen
- geringere Laufzeit, aber dafür höherer Speicherverbrauch

Dynamische Programmierung

Naiv vs. Memoization

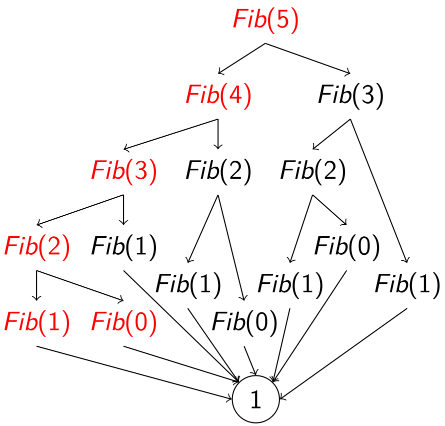

Abb.: Naive Lösung: $O(Fib(n))$

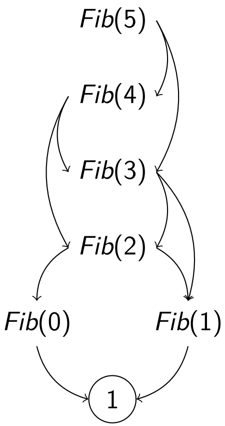

Abb.: Schnelle Lösung: $O(n)$

Dynamische Programmierung

Bottom-Up vs. Top-Down

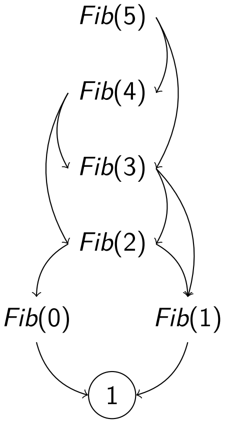

Abb.: „Top-Down“ DP

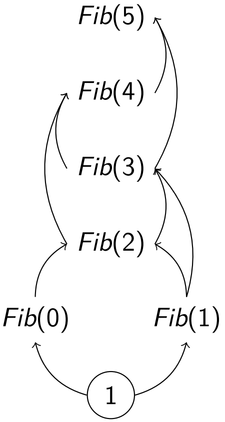

Abb.: „Bottom-Up“ DP

Die Lösung

Greedy-Ansatz

- einfach den größten Münzwert nehmen, der den verbleibenden Betrag nicht übersteigt
- **Problem:** funktioniert nur für *kanonische Münzsysteme* wie in Deutschland oder USA
- Was ist, wenn Philip in *Templonia* lebt?
- Dort gibt es nur 1, 3 und 4 templonische Säulen als Münzwerte!

Die Lösung

Greedy-Ansatz

- einfach den größten Münzwert nehmen, der den verbleibenden Betrag nicht übersteigt
- **Problem:** funktioniert nur für *kanonische Münzsysteme* wie in Deutschland oder USA
- Was ist, wenn Philip in *Templonia* lebt?
- Dort gibt es nur 1, 3 und 4 templonische Säulen als Münzwerte!

Die Lösung

Greedy-Ansatz

- einfach den größten Münzwert nehmen, der den verbleibenden Betrag nicht übersteigt
- **Problem:** funktioniert nur für *kanonische Münzsysteme* wie in Deutschland oder USA
- Was ist, wenn Philip in *Templonia* lebt?
- Dort gibt es nur 1, 3 und 4 templonische Säulen als Münzwerte!

Die Lösung

Greedy-Ansatz

- einfach den größten Münzwert nehmen, der den verbleibenden Betrag nicht übersteigt
- **Problem:** funktioniert nur für *kanonische Münzsysteme* wie in Deutschland oder USA
- Was ist, wenn Philip in *Templonia* lebt?
- Dort gibt es nur 1, 3 und 4 templonische Säulen als Münzwerte!

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- *DP-Variable:* x (zu erreichende Summe)
- *DP-Wert:* $\mathfrak{f}(x)$ (minimale Anzahl benötigter Münzen)
- *DP-Funktion:*

$$\mathfrak{f}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{i=1}^{n} \mathfrak{f}(x - w_i) + 1 & x > 0 \end{cases}$$

- Idee: Möglichkeiten für erste Münze durchprobieren und beste wählen
- gesuchtes Ergebnis: $\mathfrak{f}(W)$
- läuft durch DP in $\mathcal{O}(n \cdot W)$

Die Lösung

DP-Lösung

- Wie erhält man daraus die eigentliche Lösung (Anzahl der einzelnen Münzen)?
- ganz einfach: für jeden Wert die erste benötigte Münze speichern
- ...und dann backtracen

Die Lösung

DP-Lösung

- Wie erhält man daraus die eigentliche Lösung (Anzahl der einzelnen Münzen)?
- ganz einfach: für jeden Wert die erste benötigte Münze speichern
- ...und dann backtracen

Die Lösung

DP-Lösung

- Wie erhält man daraus die eigentliche Lösung (Anzahl der einzelnen Münzen)?
- ganz einfach: für jeden Wert die erste benötigte Münze speichern
- ...und dann backtracen

Die Lösung

Implementierung

```
value[0] = 0;
for(int j = 1; j <= n; j++){
        value[j] = INF;
        for(auto c: coins){
          if(j-c >= 0 && value[j-c]+1 < value[j]){
                  value[j] = value[j-c]+1;
                  first[j] = c;
          }
        }
}
```

Bildquellen

- http://ais.fudder.de/piece/07/12/b7/78/118667128-w-960.jpg
- Vorträge und Aufgaben des IOI-Trainings