ETH zürich

Übung 10 – Dynamische Programmierung

Informatik II, D-ITET, ETH Zurich

Heutiges Programm

Wiederholung Theorie
- Dynamische Programmierung
- Beispiel: Münzwechsel Problem
- Beispiel: Der dynamische Frosch
- Beispiel: Längste aufsteigende 2D Sequenz

Kahoot

Vorbesprechung Serie 10

1. Wiederholung Theorie

Dynamische Programmierung: Idee

- Aufteilen eines komplexen Problems in kleinere Teilprobleme; Teillösungen werden zu komplexeren Lösungen kombiniert
  = Top-Down-Rekursion
- Identische Teilprobleme werden nur einmal berechnet
  = Memoization
  - Die Idee ist, einfach **die Ergebnisse von Teilproblemen zu speichern**, damit wir sie später bei Bedarf nicht neu berechnen müssen.
- Rekursion eliminieren
  = Bottom-Up-Algorithmen
- Optional, nicht immer möglich: Platz sparen, indem die DP-Tabelle nur das Nötigste zwischenspeichert

Dynamic Programming = Divide-And-Conquer ?

- In beiden Fällen ist das Ursprungsproblem (einfacher) lösbar, indem Lösungen von Teilproblemen herangezogen werden können.
  Das Problem hat **optimale Substruktur**.
- Bei Divide-And-Conquer Algorithmen (z.B. Mergesort) sind Teilprobleme unabhängig; deren Lösungen werden im Algorithmus nur einmal benötigt.
- Beim DP sind Teilprobleme nicht unabhängig. Das Problem hat **überlappende Teilprobleme**, welche im Algorithmus mehrfach gebraucht werden.
- Damit sie nur einmal gerechnet werden müssen, werden Resultate tabelliert. Dafür darf es **zwischen Teilproblemen keine zirkulären Abhängigkeiten** geben.

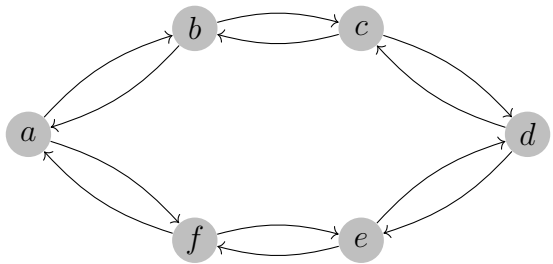
Vergleich Memoization vs. Dynamic Programming

- **Memoisierung:**
  - Top-down-Ansatz
  - Rekursion mit Caching von Ergebnissen
  - Berechnet Werte bei Bedarf träge
  - Kann effizienter sein, wenn nur wenige Werte benötigt werden
- **Dynamische Programmierung:**
  - Iterativer Bottom-up-Ansatz
  - Erstellt Lösungen aus kleineren Teilproblemen
  - Berechnet alle Werte in einer vordefinierten Reihenfolge
  - Kann effizienter sein, wenn alle Werte benötigt werden

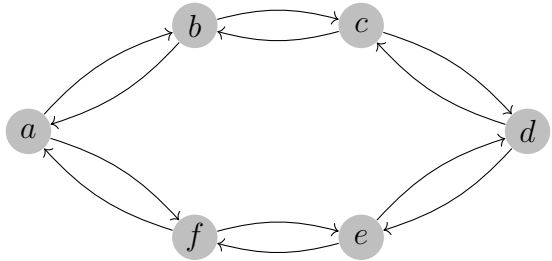
Problem ohne optimale Substruktur

Frage: Problem ohne optimale Substruktur?

Beispiel: Längster (einfacher) Weg



Problem ohne optimale Substruktur: Längster Weg



- Längster Weg von z.B. a nach e ist a, b, c, d, e, geht also über c
- Aber der längste Weg von a nach c ist *nicht* a, b, c (und analog für c nach e)

$\Rightarrow$ Die Kombination optimaler Teillösungen führt nicht zur optimalen Gesamtlösung

$\Rightarrow$ Das Problem hat keine optimale Substruktur

Dynamische Programmierung

Eine vollständige Beschreibung eines dynamischen Programms behandelt **immer** die folgenden Aspekte:

- **Definition der Teilprobleme / der DP-Tabelle**: Welche Dimensionen hat die Tabelle? Was ist die Bedeutung jedes Eintrags?
- **Rekursion: Berechnung eines Eintrags**: Wie berechnet sich ein Eintrag aus den Werten von anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?
- **Berechnungsreihenfolge (topologische Ordnung)**: In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden?
- **Lösung und Laufzeit**: Wie lässt sich die Lösung am Ende aus der Tabelle auslesen? Laufzeit des Algorithmus?

Beispiel: Münzwechsel-Problem

Definition

Gegeben sei ein Satz von Münzwerten und ein Zielbetrag. Finde die minimale Anzahl von Münzen, die benötigt werden, um den Zielbetrag zu erreichen. Beachte, dass derselbe Münzwert mehrmals verwendet werden kann.

Beispiel

Gegeben seien Münzen = [1, 2, 4] und Zielbetrag = 8, die Lösung ist 2 (4 + 4).

Aufgabe

Entwerfen Sie einen rekursiven Algorithmus, um die Aufgabe zu lösen.

Münzwechsel: rekursive Idee

Wie kann ich 8 erreichen?

3 Möglichkeiten:

1. Ich habe 7 und verwende eine 1 Münze.
2. Ich habe 6 und verwende eine 2 Münze.
3. Ich habe 4 und verwende eine 4 Münze.

Wie viele Münzen habe ich verwendet?

3 Möglichkeiten:

1. Münzen(7) + 1
2. Münzen(6) + 1
3. Münzen(4) + 1

$\Rightarrow$ Münzen(8) =
1 + min{Münzen(7), Münzen(6), Münzen(4)}

Münzwechsel: rekursiver Code

```
int coinChange(const std::vector<int>& coins, int amount) {
    if (amount == 0) {
        return 0;
    }
    int minCoins = INT_MAX;
    for (int coin : coins) {
        if (amount - coin >= 0) {
            int temp = coinChange(coins, amount - coin);
            if (temp != -1) {
                minCoins = std::min(minCoins, temp + 1);
            }
        }
    }
    return minCoins == INT_MAX ? -1 : minCoins;
}
```

Münzwechsel-Problem

Aufgabe

Entwerfen Sie einen DP-Algorithmus, um die Aufgabe zu lösen.

Münzwechsel: dynamische Programmierung

Wir können die dynamische Programmierung verwenden, um dieses Problem zu lösen, indem wir ein eindimensionales Array erstellen, bei dem `dp[i]` die minimale Anzahl von Münzen darstellt, die zur Herstellung des Betrags i benötigt werden:

- Setze jedes Element in `dp` auf einen Wert, der größer ist als die maximal mögliche Anzahl von Münzen.
- Setze `dp[0] = 0`.
- Für jede Münze `c`, iteriere durch das Array und aktualisiere `dp[i]`, wenn `dp[i-c]+1` einen niedrigeren Wert hat.

Münzwechsel: DP-Visualisierung

Münzen: [1, 2, 4] Ziel: 8

`i`	0	1	2	3	4	5	6	7	8
`dp[i]`	0	1	1	2	1	2	2	3	2

Nach der Verarbeitung der dritten und letzten Münze. Antwort: **`dp[8] = 2`**.

Münzwechsel: DP-Code

```
int coinChange(const std::vector<int>& coins, int amount) {
    std::vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int coin : coins) {
        for (int i = coin; i <= amount; ++i) {
            dp[i] = std::min(dp[i], dp[i - coin] + 1);
        }
    }
    return dp[amount] <= amount ? dp[amount] : -1;
}
```

Münzwechsel: Zeitkomplexität

Rekursiver Algorithmus

Der rekursive Algorithmus hat eine exponentielle Zeitkomplexität von $\mathcal{O}(c^n)$, wobei c die Anzahl der Münzwerte und n der Zielbetrag ist.

Dynamischer Programmieralgorithmus

Der Algorithmus der dynamischen Programmierung hat eine polynomielle Zeitkomplexität von $\mathcal{O}(c \cdot n)$, wobei c die Anzahl der Münzwerte und n der Zielbetrag ist.

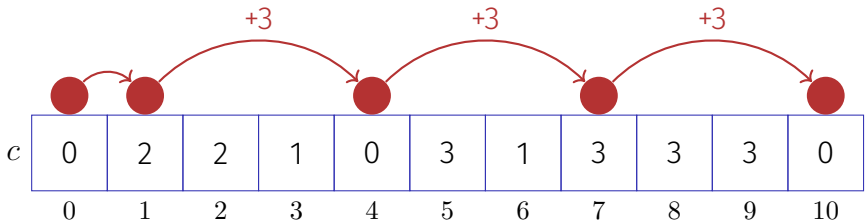
Der dynamische Frosch: Aufgabe

Gegeben: $n+1$ Felder mit Werten: $c_0, c_1, \ldots, c_n$, $c_i \geq 0$ für alle i.

Frosch will von Feld 0 zu, Feld n. Er kann vorwärts zum nächsten Feld laufen oder über zwei Felder springen.

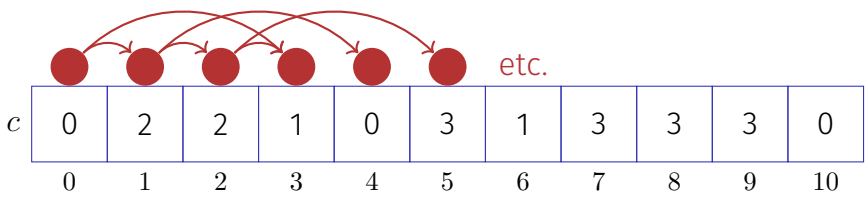
Laufen kostet nichts, Springen kostet 3. Darüber hinaus fallen die Kosten der besuchten Felder an.

Gesucht: geringste Kosten zu gegebenem c.

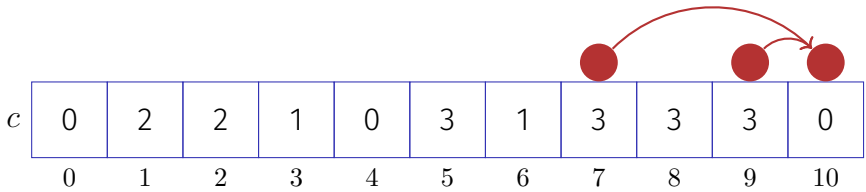


Kosten: 0+2+3+0+3+3+3+0 = 14 (nicht minimal)

Möglichkeiten



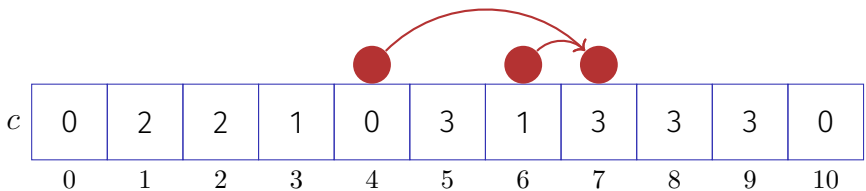
Wichtige Erkenntnis



- **Angenommen, ich kann** die minimalen Kosten M_7 und M_9 nach Feld 7 und 9 berechnen
- **Dann kann ich auch** die minimalen Kosten nach Feld 10 berechnen:

$$M_{10} = c_{10} + \min\{M_9, 3 + M_7\}$$

Rekursion



$$M_i = c_i + \begin{cases} \min\{M_{i-1}, 3 + M_{i-3}\} & \text{wenn } 3 \leq i \\ M_{i-1} & \text{wenn } 0 < i < 3 \\ 0 & \text{sonst} \end{cases}$$

Algorithmus frogRec(n)

Input: $n \geq 0$, Kosten-Array c
Output: Geringstmögliche Kosten M_n

$m \leftarrow c_n$
if $n > 3$ **then**
    $m_1 \leftarrow$ `frogRec`$(n-1)$
    $m_2 \leftarrow$ `frogRec`$(n-3)$
    $m \leftarrow m + \min(m_1, 3 + m_2)$
else if $n > 0$ **then**
    $m \leftarrow m +$ `frogRec`$(n-1)$
return m

Algorithmus frogMem(n)

Input: $n \geq 0$, Kosten-Array c
Output: Geringstmögliche Kosten M_n

if $\exists$ memo$[n]$ **then**
    $m \leftarrow$ memo$[n]$
else
    $m \leftarrow c_n$
    if $n > 3$ **then**
        $m_1 \leftarrow$ **frogMem**$(n-1)$
        $m_2 \leftarrow$ **frogMem**$(n-3)$
        $m \leftarrow m + \min(m_1, 3 + m_2)$
    else if $n > 0$ **then**
        $m \leftarrow m +$ **frogMem**$(n-1)$
    memo$[n] \leftarrow m$
return m

DP-Algorithmus: Beschreibung

	Dimension der Tabelle? Bedeutung der Einträge?
1.	Tabelle der Grösse $n \times 1$. n-ter Eintrag von M enthält minimale Kosten zum Erreichen von Feld n.
	Welche Einträge hängen nicht von anderen ab?
2.	Wert M_0 ist unabhängig einfach „berechenbar“.
	Berechnungsreihenfolge?
3.	M_i mit aufsteigenden i.
	Rekonstruktion einer Lösung?
4.	M_n enthält die minimalen Kosten zum Erreichen des Feldes n.

Längste aufsteigende Spur

Gegeben $n \times m$ Matrix A:

9	27	42	41	48
35	39	8	3	5
12	49	2	38	4
15	47	29	28	6
19	1	25	33	10

Gesucht: längste aufsteigende Spur:

$$4, 6, 28, 29, 47, 49$$

Definition der DP-Tabelle

- Welche Dimensionen hat die Tabelle?
  - $n \times m \quad (\times 2)$
- Was ist die Bedeutung jedes Eintrags?
  - In $T[x][y]$ steht Länge der längsten aufsteigenden Spur, die im Feld $A[x][y]$ endet
  - In $S[x][y]$ stehen Koordinaten des Vorgängers von (x, y) in der aufsteigenden Spur (falls existent)

Berechnung eines Eintrags

- Wie berechnet sich ein aus anderen Einträgen? Welche Einträge hängen nicht von anderen Einträgen ab?
  - Betrachte Nachbarn mit kleineren Einträgen in A
  - Wähle von diesen den mit dem grössten Eintrag in T
  - Aktualisiere T und S (S erhält Koordinaten des ausgewählten Nachbarn, T erhält Wert + 1 des ausgewählten Nachbarn).

Berechnungsreihenfolge

- In welcher Reihenfolge kann man die Einträge berechnen, so dass die jeweils benötigten anderen Einträge bereits vorher berechnet wurden?

- Bottom-Up: Beginne mit kleinstem Element in A, dann nächstkleinstes, usw. (bedeutet, dass man A sortieren muss)

- Rekursiv: Beliebige Reihenfolge; falls Eintrag schon berechnet, dann überspringen, sonst rekursiv von kleineren Nachbarn aus berechnen

Auslesen der Lösung

- Wie lässt sich die Lösung am Ende aus der Tabelle auslesen?
  - Betrachte alle Einträge, um den Eintrag zu finden, in dem eine längste Spur endet. Von dort aus können wir die Lösung rekonstruieren, indem wir den jeweiligen Vorgängern folgen.

2. Kahoot

3. Vorbesprechung Serie 10

Vorbesprechung Serie 10

1. **Max Sum Increasing Subsequence**, Programmieren;
   Maximale Summe einer aufsteigenden Teilfolge finden
2. **Max Sum Triangle**, Programmieren;
   Fragen zu 2D DP Aufgabe beantworten
3. **Enumerating Palindromes**, Text;
   Optimale stückweise konstante Funktion finden um Daten anzunähern

Tips Serie 10

1. Schaut euch die Vorlesungsfolien ab ?? noch einmal an. Ihr müsst den Ansatz jedoch von längster Folge auf Folge mit grösster Summe anpassen
2. Verwendet als DP-Tabelle eine **`Triangle`** Datenstruktur
3. -