ETH zürich

Übungslektion 11 – DP II

Informatik II

5. Mai 2026

Willkommen!

Polybox

Passwort: jschul

Personal Website

`https://jschultev.github.io/personal_website/`

Heutiges Programm

Rückblick: Stab schneiden

Begrenztes Stab schneiden

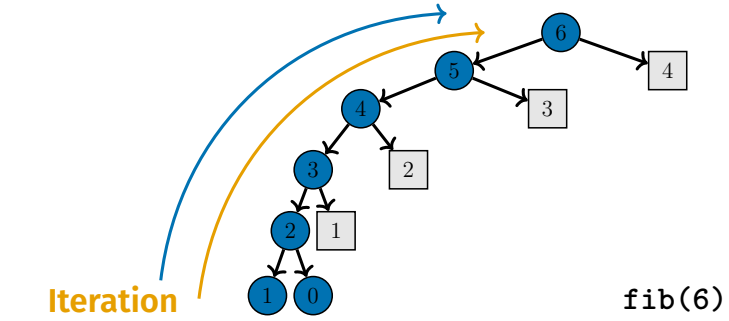
Frosch-Sprünge

Zusammenfassung

Hausaufgaben

1. Rückblick: Stab schneiden

Bottom-Up



Rückblick: Stab schneiden

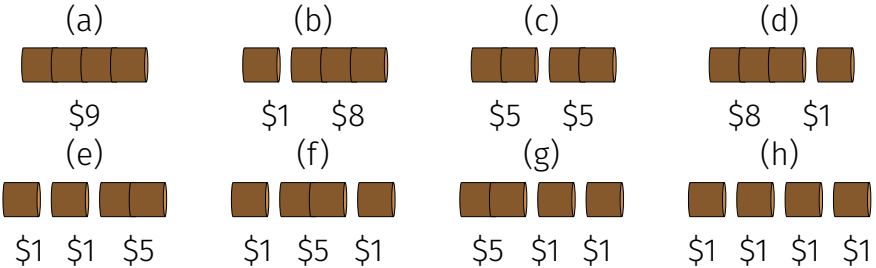
Aufgabenbeschreibung:
Ein Stab der Länge n muss in kleinere Stücke geschnitten werden. Jedes Stück der Länge i hat einen Preis $p[i]$, angegeben in einer Preisliste p.

- **Eingabe**: Länge n und Preisliste p.
- **Ziel**: Den Stab so zerschneiden, dass das Ergebnis den maximalen Wert hat.
- **Ausgabe**: Der maximale Wert, der durch das Zerschneiden des Stabes erzielt werden kann.

Rückblick: Stab schneiden

Brute-Force-Ansatz: Jede Möglichkeit prüfen

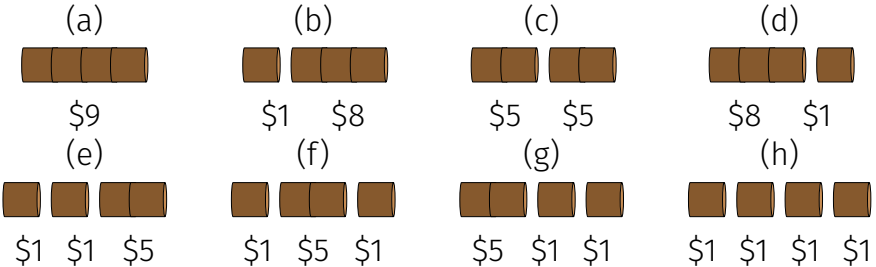
Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9



Rückblick: Stab schneiden

Brute-Force-Ansatz: Jede Möglichkeit prüfen

Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9



Problem: **exponentielle Laufzeit!** 2^{n-1} Optionen $\rightarrow \mathcal{O}(2^n)$

Rückblick: Stab schneiden

- Der optimale Preis eines Stabes der Länge i besteht aus dem Preis des abgeschnittenen Stücks der Länge k und dem Preis des verbleibenden Stücks der Länge $i - k$. Wie sieht das als Formel aus?

$$S[i] = p[k] + S[i - k]$$

- Da die optimale Schnittlänge k unbekannt ist, müssen alle möglichen Optionen bewertet werden, um die beste auszuwählen. Wie machen wir das?

$$S[i] = \begin{cases} 0 & \text{if } i = 0 \\ \max\{p[k] + S[i - k] : k \in [1, i]\} & \text{if } i \neq 0 \end{cases}$$

- Beachte, dass diese Formel nur gilt, wenn p einen Preis für jede Länge von 1 bis i angibt.

Rückblick: Stab schneiden

```
def max_value(p, n):

    #Base case
    if n == 0:
        return 0

    #Optimal Price
    options = [p[k] + max_value(p, n-k) for k in range(1, n+1)]
    return max(options)
```

Rückblick: Stab schneiden

Warum können wir Dynamische Programmierung bei Stab schneiden verwenden?

Rückblick: Stab schneiden

Warum können wir Dynamische Programmierung bei Stab schneiden verwenden?

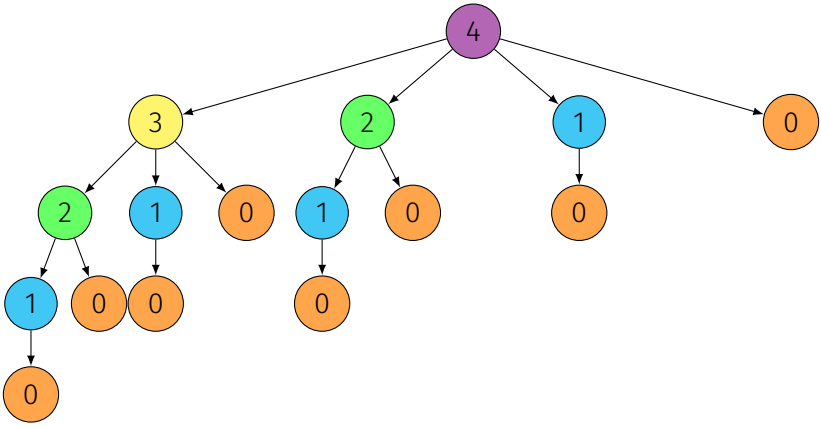
1. Überlappende Teilprobleme
   - Ohne Überlappungen würde ein DP-Algorithmus auch in exponentieller Zeit $\mathcal{O}(2^n)$ laufen.
   - DP führt allerdings nicht bedingungslos zu polynomieller Laufzeit.

Rückblick: Stab schneiden

Warum können wir Dynamische Programmierung bei Stab schneiden verwenden?

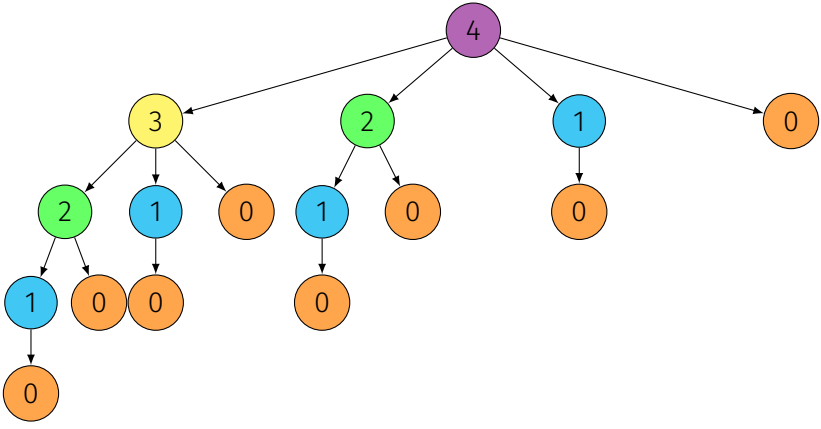
1. Überlappende Teilprobleme
   - Ohne Überlappungen würde ein DP-Algorithmus auch in exponentieller Zeit $\mathcal{O}(2^n)$ laufen.
   - DP führt allerdings nicht bedingungslos zu polynomieller Laufzeit.
2. Optimale Teilstruktur
   - Ohne optimale Teilstruktur sind die überlappenden Teilprobleme nutzlos.

Erinnerung: Überlappende Teilprobleme



- Einzelne Teilprobleme werden mehrfach gelöst. Dies führt zu einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^n)$.

Erinnerung: Überlappende Teilprobleme



- Einzelne Teilprobleme werden mehrfach gelöst. Dies führt zu einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^n)$.
- DP führt allgemein zu einer polynomiellen Laufzeit. Hier: $\mathcal{O}(n^2)$.

Was ist eine optimale Teilstruktur?

- Sei $S[5]$ der optimale Wert, der für einen Stab der Länge 5 erzielbar ist.

Was ist eine optimale Teilstruktur?

- Sei $S[5]$ der optimale Wert, der für einen Stab der Länge 5 erzielbar ist.
- $S[5]$ hängt (unter anderem) von $S[4]$ ab.

Was ist eine optimale Teilstruktur?

- Sei $S[5]$ der optimale Wert, der für einen Stab der Länge 5 erzielbar ist.
- $S[5]$ hängt (unter anderem) von $S[4]$ ab.
- Mit anderen Worten:
  „Der optimale Wert für einen Stab der Länge 5 ($S[5]$) hängt vom optimalen Wert für einen Stab der Länge 4 ($S[4]$) ab.“

Was ist eine optimale Teilstruktur?

„Der optimale Wert für einen Stab der Länge 5 ($S[5]$) hängt vom optimalen Wert für einen Stab der Länge 4 ($S[4]$) ab.“

Beweis durch Widerspruch:

1. Wenn $S[5]$ optimal ist, muss $S[4]$ ebenfalls optimal sein.

Was ist eine optimale Teilstruktur?

„Der optimale Wert für einen Stab der Länge 5 ($S[5]$) hängt vom optimalen Wert für einen Stab der Länge 4 ($S[4]$) ab.“

Beweis durch Widerspruch:

1. Wenn $S[5]$ optimal ist, muss $S[4]$ ebenfalls optimal sein.
2. Wäre $S[4]$ nicht optimal, könnte man einen höheren Wert für $S[4]$ erhalten.

Was ist eine optimale Teilstruktur?

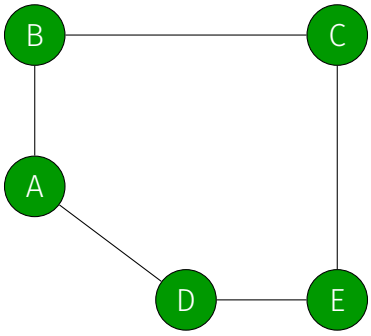
„Der optimale Wert für einen Stab der Länge 5 ($S[5]$) hängt vom optimalen Wert für einen Stab der Länge 4 ($S[4]$) ab.“

Beweis durch Widerspruch:

1. Wenn $S[5]$ optimal ist, muss $S[4]$ ebenfalls optimal sein.
2. Wäre $S[4]$ nicht optimal, könnte man einen höheren Wert für $S[4]$ erhalten.
3. Folglich könnte man auch einen höheren Wert für $S[5]$ finden. Daher kann $S[5]$ nicht optimal sein.

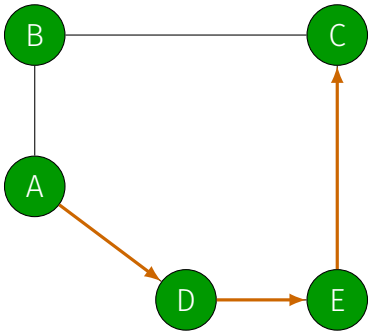
Beispiel: Nicht-Optimale Teilstruktur

- Wir suchen den längsten geometrischen Pfad von A nach C.



Beispiel: Nicht-Optimale Teilstruktur

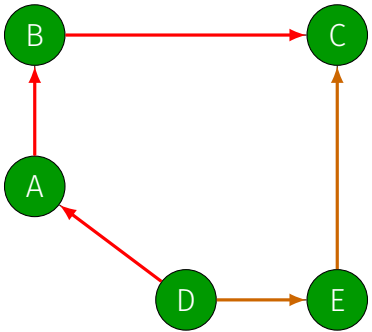
- Wir suchen den längsten geometrischen Pfad von A nach C.



Dieser Pfad kann in Pfade A–D und D–C aufgeteilt werden. (Teilprobleme)

Beispiel: Nicht-Optimale Teilstruktur

- Wir suchen den längsten geometrischen Pfad von A nach C.



Dieser Pfad kann in Pfade A–D und D–C aufgeteilt werden. (Teilprobleme)

- Der optimale (längste) Pfad von D nach C ist jedoch nicht Teil des optimalen Pfades von A nach C.

Rückblick: Stab schneiden

Warum können wir Dynamische Programmierung bei Stab schneiden verwenden?

1. Überlappende Teilprobleme
   - Ohne Überlappungen würde ein DP-Algorithmus auch in exponentieller Zeit $\mathcal{O}(2^n)$ laufen.
     $\rightarrow$ DP führt nicht bedingungslos zu polynomieller Laufzeit.
2. Optimale Teilstruktur
   - Ohne optimale Teilstruktur kann die Lösung nicht mit DP erhalten werden.

2. Begrenztes Stab schneiden

Was bedeutet “begrenzt”?

Unbegrenzt:

- Der Stab kann so oft wie nötig geschnitten werden.

Begrenzt:

- Der Stab darf maximal c Mal geschnitten werden.

Was bedeutet “begrenzt”?

Unbegrenzt:

- Der Stab kann so oft wie nötig geschnitten werden.

Begrenzt:

- Der Stab darf maximal c Mal geschnitten werden.

$\rightarrow$ Der maximale Wert eines Stabes der Länge 20 unterscheidet sich je nachdem, ob er bis zu 10 Mal oder nur zweimal geschnitten werden darf.

Wiederkehrende Muster

- Vielleicht erkennt ihr bei der Lösung dieser Aufgabe ein Muster, die ihr bereits aus einem DP-Problem der Vorlesung kennen.
- Es gibt einige bekannte DP-Probleme, die im Programmierer-Alltag wiederholt in verschiedenen Formen auftauchen.
- Diese bekannten Muster zu erkennen kann enorm hilfreich sein.

Drei Schritte der Dynamischen Programmierung

1. Identifiziere die Teilprobleme & Optionen

Drei Schritte der Dynamischen Programmierung

1. Identifiziere die Teilprobleme & Optionen
2. Definiere die Rekursion

Drei Schritte der Dynamischen Programmierung

1. Identifiziere die Teilprobleme & Optionen
2. Definiere die Rekursion
3. Implementiere eine Lösung
   - Finde eine geeignete Datenstruktur
   - Identifiziere die Abhängigkeiten
   - Bestimme die Reihenfolge beim Ausfüllen der Struktur

Schritt 1: Identifiziere die Teilprobleme & Optionen

- $S[i]$ war der optimale Wert für einen Stab der Länge i.

Schritt 1: Identifiziere die Teilprobleme & Optionen

- $S[i]$ war der optimale Wert für einen Stab der Länge i.
- Jetzt hängt es auch davon ab, wie viele Schnitte noch übrig sind.
  $\rightarrow S[i,0], \quad S[i,1], \quad S[i,2], \quad \ldots, \quad S[i,c]$

Schritt 1: Identifiziere die Teilprobleme & Optionen

- $S[i]$ war der optimale Wert für einen Stab der Länge i.
- Jetzt hängt es auch davon ab, wie viele Schnitte noch übrig sind.
  $\rightarrow S[i,0], \quad S[i,1], \quad S[i,2], \quad \ldots, \quad S[i,c]$
- $S[i,t]$ bezeichnet den optimalen Wert, den man für einen Stab der Länge i mit t verbleibenden Schnitten erzielen kann.
  $\rightarrow$ **Zweidimensionale Teilprobleme!**

Schritt 1: Identifiziere die Teilprobleme & Optionen

Was sind die Optionen beim Teilproblem $S[i, t]$?

Schritt 1: Identifiziere die Teilprobleme & Optionen

Was sind die Optionen beim Teilproblem $S[i, t]$?

- Option 1: Schnitt nach Länge 1

$$S[i, t] = p[1] + S[i - 1, t - 1]$$

Schritt 1: Identifiziere die Teilprobleme & Optionen

Was sind die Optionen beim Teilproblem $S[i, t]$?

- Option 1: Schnitt nach Länge 1

$$S[i, t] = p[1] + S[i - 1, t - 1]$$

- Option 2: Schnitt nach Länge 2

$$S[i, t] = p[2] + S[i - 2, t - 1]$$

Schritt 1: Identifiziere die Teilprobleme & Optionen

Was sind die Optionen beim Teilproblem $S[i, t]$?

- Option 1: Schnitt nach Länge 1

$$S[i, t] = p[1] + S[i - 1, t - 1]$$

- Option 2: Schnitt nach Länge 2

$$S[i, t] = p[2] + S[i - 2, t - 1]$$

- Option 3: Schnitt nach Länge 3

$$S[i, t] = p[3] + S[i - 3, t - 1]$$

Schritt 1: Identifiziere die Teilprobleme & Optionen

Was sind die Optionen beim Teilproblem $S[i, t]$?

- Option 1: Schnitt nach Länge 1

$$S[i, t] = p[1] + S[i - 1, t - 1]$$

- Option 2: Schnitt nach Länge 2

$$S[i, t] = p[2] + S[i - 2, t - 1]$$

- Option 3: Schnitt nach Länge 3

$$S[i, t] = p[3] + S[i - 3, t - 1]$$

- Option i: Schnitt nach Länge i (eg. Stück als ganzes verkaufen.)

$$S[i, t] = p[i] + S[0, t - 1]$$

Schritt 2: Definiere die Rekursion

- Aus allen Optionen wählen wir diejenige mit dem höchsten Wert.

$$S[i,t] = \max(p[k] + S[i-k, t-1] \text{ for } k \text{ in range}(1, i+1))$$

Schritt 2: Definiere die Rekursion

Was war der Basisfall des unbegrenzten Stab-schneiden-Problems?

Schritt 2: Definiere die Rekursion

Was war der Basisfall des unbegrenzten Stab-schneiden-Problems?

- $S[0] = 0$, ein Stab der Länge 0 hat keinen Wert.
  $\rightarrow$ Das ändert sich nicht, egal wie viele Schnitte noch übrig sind.

$$S[0, t] = 0, \quad t \in [0, c]$$

Schritt 2: Definiere die Rekursion

Was war der Basisfall des unbegrenzten Stab-schneiden-Problems?

- $S[0] = 0$, ein Stab der Länge 0 hat keinen Wert.
  $\rightarrow$ Das ändert sich nicht, egal wie viele Schnitte noch übrig sind.

$$S[0, t] = 0, \quad t \in [0, c]$$

Was passiert, wenn wir keine Schnitte mehr haben?

- Nur der Wert des gesamten verbleibenden Stücks kann erzielt werden.

$$S[i, 0] = p[i], \quad i \in [0, n]$$

Schritt 2: Definiere die Rekursion

$$S[i,t] = \begin{cases} p[i] & \text{if } i = 0 \text{ or } t = 0 \\ \max(p[k] + S[i-k, t-1] \text{ for } k \in [1,i]) & \text{else} \end{cases}$$

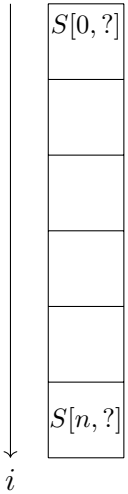
Schritt 2: Definiere die Rekursion

$$S[i,t] = \begin{cases} p[i] & \text{if } i = 0 \text{ or } t = 0 \\ \max(p[k] + S[i-k, t-1] \text{ for } k \in [1,i]) & \text{else} \end{cases}$$

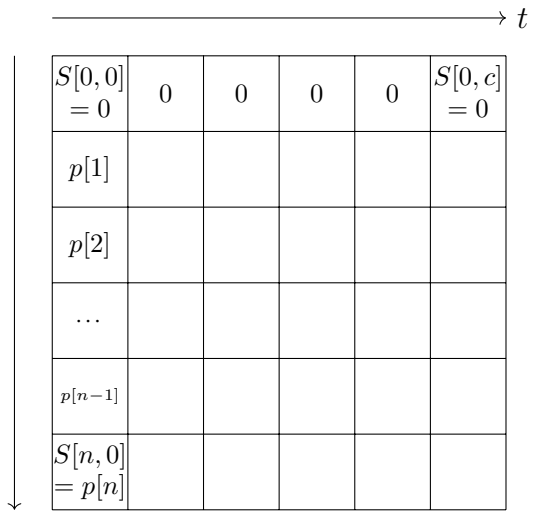
- Hier gehen wir davon aus, dass der Preis 0 für Länge 0 beim Index 0 unserer Tabelle steht. So korrespondiert der Preis immer mit der Länge i
- Da $p[0] = 0$, können die beiden Basisfälle kombiniert werden.

Länge $\boldsymbol{i}$	0	1	2	3	4
Preis $\boldsymbol{p[i]}$	0	1	5	8	9

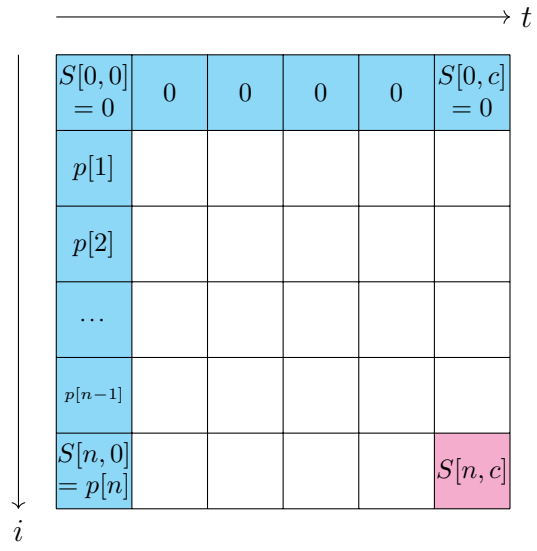
Schritt 2: Grafisch



Schritt 2: Grafisch



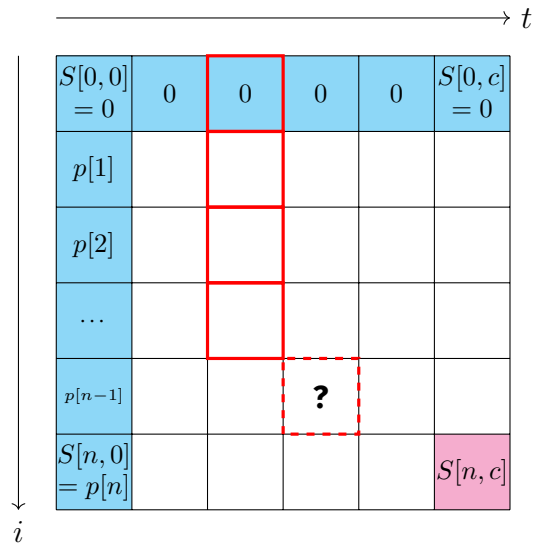
Schritt 2: Grafisch



Basisfälle

Ziel

Schritt 2: Grafisch



Basisfälle

Abhängigkeiten

Ziel

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 2D-Array der Größe $(n+1) \times (c+1)$.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 2D-Array der Größe $(n+1) \times (c+1)$.
- Identifiziere die Abhängigkeiten:
  - Um $S[i,t]$ zu berechnen, müssen wir die Lösungen zu $S[i-1,t-1], S[i-2,t-1], \ldots, S[0,t-1]$ kennen.

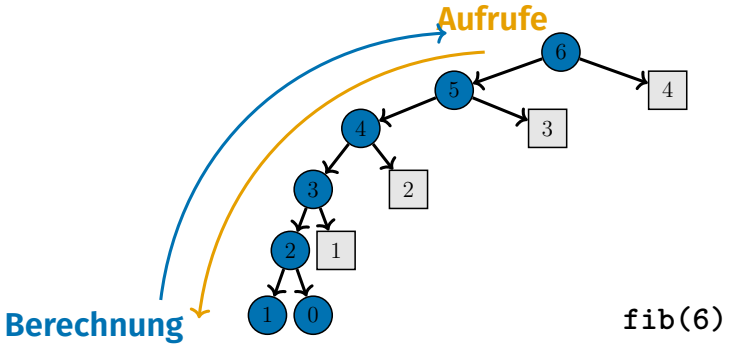
Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 2D-Array der Größe $(n+1) \times (c+1)$.
- Identifiziere die Abhängigkeiten:
  - Um $S[i,t]$ zu berechnen, müssen wir die Lösungen zu $S[i-1,t-1], S[i-2,t-1], \ldots, S[0,t-1]$ kennen.
- Bestimme die Reihenfolge beim Ausfüllen der Struktur:
  - Beginne bei $S[0,0]$ und fülle das Array von links nach rechts und oben nach unten aus.

Schritt 3: Dynamische Programmierung

```python
import numpy as np
#LRC: LimitedRodCutting
def LRC_dp(n,p,c):

    S = np.zeros((n+1,c+1)) #Behandelt schon den ersten Basisfall
    S[:,0] = p #1. Basisfall mit Slicing

    for t in range(1, c+1):
        for i in range(1, n+1):
            options = [p[k] + S[i-k,t-1] for k in range(1,i+1)]
            S[i,t] = max(options)

    return S[n,c]
```

Top-Down Memoisierung



Schritt 3: Memoisierung

```
def LRC_memo(i,p,t,memo = None):

    if memo is None:
        memo = dict()

    if (i,t) not in memo:
        if i < 2 or t == 0: #i == 1 kann auch als Basisfall gesehen werden
            memo[(i,t)] = p[i]
        else:
            options = [p[k] + LRC_memo(i-k,p,t-1,memo) for k in range(1,i+1)
            memo[(i,t)] = max(options)

    return memo[(i,t)]
```

(Bemerkung zur Implementierung)

- Es gibt auch die Option, den Stab gar nicht zu schneiden.
- Dieser Fall ist durch $k = i$ abgedeckt.
- Der Stab wird in ein Stück der Länge i und ein Stück der Länge 0 geschnitten.
- Unser Code zählt einen Schnitt, obwohl der Stab technisch gesehen nicht geschnitten wird.
- Da die Rekursion jedoch endet, wird dieser fehlende Schnitt die Lösung nicht beeinflussen.
- Eine Rekonstruktion der tatsächlichen Schnittsequenz erfordert jedoch eine Anpassung des Algorithmus.

3. Frosch-Sprünge

Aufgabenbeschreibung

- **Eingabe**: Ein 2D-Array a mit ganzen Zahlen aus der Menge $\{-1\} \cup [1, \infty)$.

Aufgabenbeschreibung

- **Eingabe**: Ein 2D-Array a mit ganzen Zahlen aus der Menge $\{-1\} \cup [1, \infty)$.
- **Ziel**: Ein Frosch startet in der oberen linken Ecke und muss die untere rechte Ecke mit möglichst wenigen Sprüngen erreichen. Die geltenden Regeln werden auf der folgenden Folie erläutert.

Aufgabenbeschreibung

- **Eingabe**: Ein 2D-Array a mit ganzen Zahlen aus der Menge $\{-1\} \cup [1, \infty)$.
- **Ziel**: Ein Frosch startet in der oberen linken Ecke und muss die untere rechte Ecke mit möglichst wenigen Sprüngen erreichen. Die geltenden Regeln werden auf der folgenden Folie erläutert.
- **Ausgabe**: Die minimale Anzahl an Sprüngen, die benötigt wird.

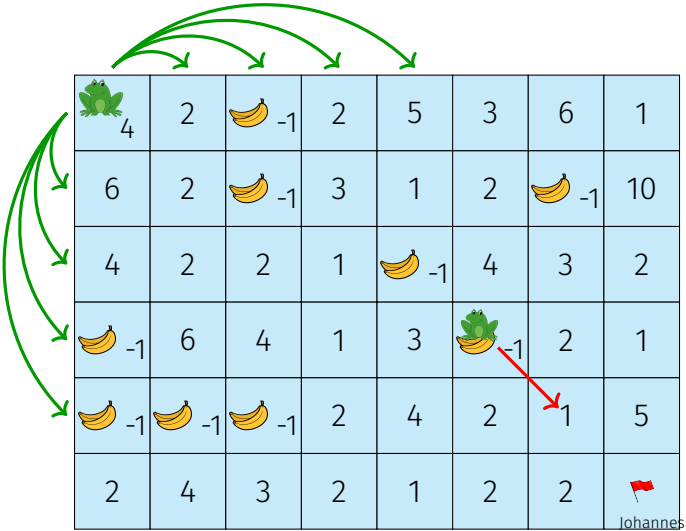
Aufgabenbeschreibung: Regeln

- Freies Feld ($a[i, j] > 0$):
  - Der Frosch kann entweder nach Osten oder nach Süden springen. Er kann eine Distanz von $1, 2, 3, \ldots, a[i, j]$ Felder springen.

Aufgabenbeschreibung: Regeln

- Freies Feld ($a[i, j] > 0$):
  - Der Frosch kann entweder nach Osten oder nach Süden springen. Er kann eine Distanz von $1, 2, 3, \ldots, a[i, j]$ Felder springen.
- Bananenschale ($a[i, j] = -1$):
  - Der Frosch rutscht auf der Bananenschale aus und landet ein Feld weiter in süd-östlicher Richtung (diagonal).
  - Das Ausrutschen auf einer Bananenschale zählt nicht als Sprung.

Aufgabenbeschreibung: Beispiel



Problemzerlegung

Dieses Problem ist sehr komplex. Es kann jedoch in Teilprobleme aufgeteilt werden, die unabhängig voneinander gelöst und kombiniert werden können.

1. Basisproblem
2. Variable Sprungdistanz
3. Zweidimensionalität
4. Bananenschale

Problemzerlegung

Unser Basisproblem bekommen wir, indem wir die Bananenschalen, die zweite Dimension und die variable Sprungdistanz entfernen.

1. Basisproblem
2. Variable Sprungdistanz
3. Zweidimensionalität
4. Bananenschale

Basisproblem

- **Eingabe**: Ein 1D-Array a mit positiven ganzen Zahlen ($a[j] > 0$).

Basisproblem

- **Eingabe**: Ein 1D-Array a mit positiven ganzen Zahlen ($a[j] > 0$).
- **Ziel**: Der Frosch muss Index $n - 1$ von Index 0 aus mit möglichst wenigen Sprüngen erreichen. Er hat zwei Optionen:
  - Er kann ein einzelnes Feld springen.
  - Er kann so viele Felder springen, wie der Wert des aktuellen Feldes $a[j]$ angibt.

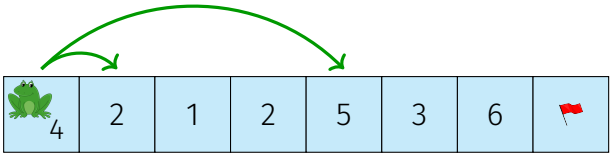
Basisproblem

- **Eingabe**: Ein 1D-Array a mit positiven ganzen Zahlen ($a[j] > 0$).
- **Ziel**: Der Frosch muss Index $n - 1$ von Index 0 aus mit möglichst wenigen Sprüngen erreichen. Er hat zwei Optionen:
  - Er kann ein einzelnes Feld springen.
  - Er kann so viele Felder springen, wie der Wert des aktuellen Feldes $a[j]$ angibt.
- **Ausgabe**: Die minimale Anzahl an Sprüngen, um von links nach rechts zu gelangen.

Bemerkung: Basisproblem

- Nicht alle DP-Probleme können intuitiv in simplere Probleme zerlegt werden.
- Speziell 2D-Probleme haben oft kein simpleres 1D-Problem (z.B. Königsweg).

Aufgabenbeschreibung: Beispiel



Drei Schritte der Dynamischen Programmierung

1. Identifiziere die Teilprobleme & Optionen

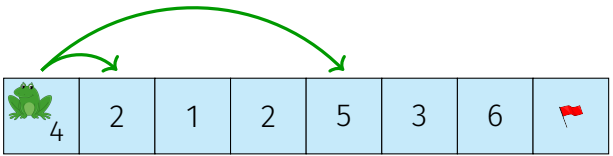
Drei Schritte der Dynamischen Programmierung

1. Identifiziere die Teilprobleme & Optionen
2. Definiere die Rekursion

Drei Schritte der Dynamischen Programmierung

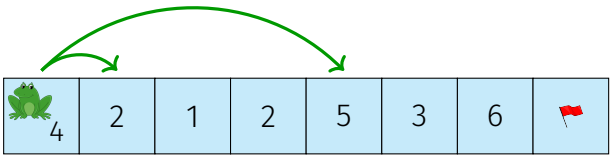
1. Identifiziere die Teilprobleme & Optionen
2. Definiere die Rekursion
3. Implementiere eine Lösung
   - Finde eine geeignete Datenstruktur
   - Identifiziere die Abhängigkeiten
   - Bestimme die Reihenfolge beim Ausfüllen der Struktur

Schritt 1: Identifiziere die Teilprobleme & Optionen



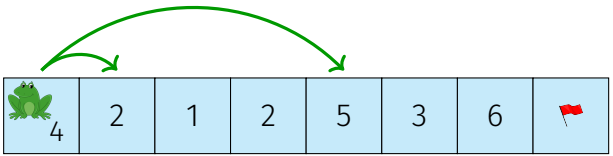
- $S[j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Index j zum Ziel $n-1$ zu gelangen.

Schritt 1: Identifiziere die Teilprobleme & Optionen



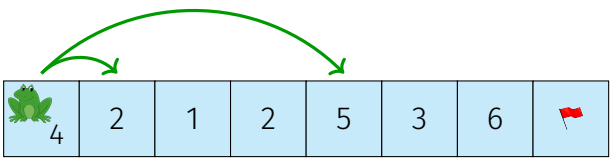
- $S[j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Index j zum Ziel $n-1$ zu gelangen.
- Wir suchen nach $S[0]$, der minimalen Anzahl Sprünge vom Start bis zum Schluss

Schritt 1: Identifiziere die Teilprobleme & Optionen



Was sind die Optionen des Frosches bei Index j?

Schritt 1: Identifiziere die Teilprobleme & Optionen



Was sind die Optionen des Frosches bei Index j?

- **Option A**: Springe 1 Feld
- **Option B**: Springe $a[j]$ Felder

$\rightarrow$ Der Frosch wählt die bessere Option.

Schritt 2: Definiere die Rekursion

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \text{ (base case)} \\ 1 + \min(S[j+1],\, S[j + a[j]]) & \text{else} \end{cases}$$

Randfall

Problem: Der Frosch springt über das Ziel hinaus.
$\rightarrow$ Dies verursacht einen **IndexError: out of bounds**.

Wie können wir das lösen?

Randfall

Problem: Der Frosch springt über das Ziel hinaus.
$\rightarrow$ Dies verursacht einen **IndexError: out of bounds**.

Wie können wir das lösen?

$$\min(j + a[j],\ n - 1)$$

- Dieser Ausdruck gibt $n - 1$ zurück, falls die Sprungdistanz ungültig ist.

Schritt 2: Definiere die Rekursion

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \text{ (base case)} \\ 1 + \min(S[j+1],\, S[j+a[j]]) & \text{else} \end{cases}$$

...wird zu...

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \text{ (base case)} \\ 1 + \min(S[j+1],\, S[\min(j+a[j],\, n-1)]) & \text{else} \end{cases}$$

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 1D-Array, um die Teilprobleme $S[j]$ zu speichern.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 1D-Array, um die Teilprobleme $S[j]$ zu speichern.
- Identifiziere die Abhängigkeiten:
  - Um $S[j]$ zu berechnen, brauchen wir die Lösungen von $S[j+1]$ und $S[j+a[j]]$.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - 1D-Array, um die Teilprobleme $S[j]$ zu speichern.
- Identifiziere die Abhängigkeiten:
  - Um $S[j]$ zu berechnen, brauchen wir die Lösungen von $S[j+1]$ und $S[j+a[j]]$.
- Bestimme die Reihenfolge beim Ausfüllen der Struktur:
  - Das Array muss daher **von rechts nach links** ausgefüllt werden.

Schritt 3: Implementiere eine Lösung

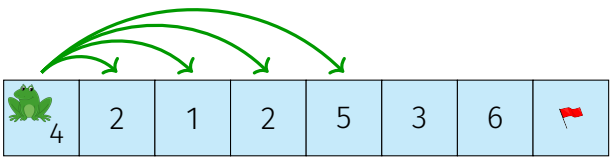
```python
import numpy as np

def base_problem(a):
  n = len(a)

  S = np.zeros(n)
  S[-1] = 0 #Basisfall (redundant wegen "np.zeros")

  for j in range(n-2, -1, -1):
      S[j] = 1 + min(S[j+1], S[min(j+a[j], n-1)])

  return S[0]
```

Problemzerlegung

Nun fügen wir mehr Aspekte des originalen Problems hinzu.

1. Basisproblem
2. Variable Sprungdistanz
3. Zweidimensionalität
4. Bananenschale

Variable Sprungdistanz



- Jetzt kann der Frosch nicht nur ein oder $a[j]$ Felder springen, sondern jede Distanz von 1 bis $a[j]$ Felder.
- Wie müssen wir die bisherige Lösung verändern?

Schritt 2: Definiere die Rekursion

→ Wir prüfen alle Optionen mit einer for-Schleife.

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \\ 1 + \min(S[j+1],\, S[\min(j + a[j],\, n-1)]) & \text{else} \end{cases}$$

$$\Downarrow$$

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \\ 1 + \min(S[\min(j + k,\, n-1)] \text{ for } k \text{ in range}(1,\, a[j]+1)) & \text{else} \end{cases}$$

Schritt 2: Definiere die Rekursion

Wir können Iterationen sparen, indem wir den out-of-bounds-Vergleich in die `range` verschieben. Diese Änderung wird später weitere Vorteile haben.

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \\ 1 + \min(S[\min(j+k,\ n-1)] \text{ for } k \text{ in range}(1,\ a[j]+1)) & \text{else} \end{cases}$$

$$\Downarrow$$

$$S[j] = \begin{cases} 0 & \text{if } j = n-1 \\ 1 + \min(S[k] \text{ for } k \text{ in range}(j+1,\ min(j+a[j], n-1)+1)) & \text{else} \end{cases}$$

Schritt 3: Implementiere eine Lösung

```python
import numpy as np

def variable_jumping_distance(a):

  n = len(a)
  S = np.zeros(n)
  S[-1] = 0 #base case (redundant through np.zeros)

  for j in range(n-2, -1, -1):
      options = [S[k] for k in range(j+1, min(j+a[j], n-1)+1)]
      S[j] = 1 + min(options)

  return S[0]
```

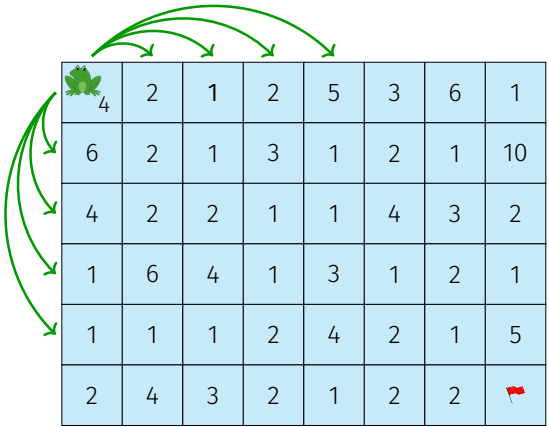
Problemzerlegung

Nun fügen wir mehr Aspekte des originalen Problems hinzu.

1. Basisproblem
2. Variable Sprungdistanz
3. Zweidimensionalität
4. Bananenschale

Zweidimensionalität

Jetzt ist die Eingabe ein 2D-Array. Der Frosch kann auch nach Süden springen.



Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[m-1, n-1]$ zu gelangen.

Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[m-1, n-1]$ zu gelangen.
- Optionen: Mit der neu hinzugefügten Dimension kann der Frosch auch nach Süden springen. Das bedeutet, dass wir entscheiden müssen, ob wir bis zu $a[i, j]$ Felder nach unten oder nach rechts springen.

Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[m-1, n-1]$ zu gelangen.
- Optionen: Mit der neu hinzugefügten Dimension kann der Frosch auch nach Süden springen. Das bedeutet, dass wir entscheiden müssen, ob wir bis zu $a[i, j]$ Felder nach unten oder nach rechts springen.
- Der bisherige Algorithmus wird kopiert für die neue Dimension. Anschliessend werden die beiden Ergebnisse verglichen.

Schritt 2: Definiere die Rekursion

→ Neu hinzugefügt wird der Vergleich zwischen Süd und Ost

$$S[i,j] = \begin{cases} 0 & \text{if } j = n-1 \text{ and } i = m-1 \\ 1 + \min(*south, *east) & \text{else} \end{cases}$$

mit...

$$south = [S[k] \text{ for } k \text{ in range}(i+1,\ min(i + a[i,j], m-1) + 1)]$$
$$east = [S[k] \text{ for } k \text{ in range}(j+1,\ min(j + a[i,j], n-1) + 1)]$$

Either the south or east list can be empty, but never both.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Neue Dimension $\rightarrow$ ein 2D-Array mit der gleichen Größe wie das Eingabe-Array.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Neue Dimension $\rightarrow$ ein 2D-Array mit der gleichen Größe wie das Eingabe-Array.
- Identifiziere die Abhängigkeiten:
  - Um $S[i, j]$ zu berechnen, müssen $a[i, j]$ Teilprobleme rechts und unterhalb gelöst sein. Für ein unbekanntes $a[i, j]$ heisst das: Alle Teilprobleme unterhalb und rechts.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Neue Dimension $\rightarrow$ ein 2D-Array mit der gleichen Größe wie das Eingabe-Array.
- Identifiziere die Abhängigkeiten:
  - Um $S[i, j]$ zu berechnen, müssen $a[i, j]$ Teilprobleme rechts und unterhalb gelöst sein. Für ein unbekanntes $a[i, j]$ heisst das: Alle Teilprobleme unterhalb und rechts.
- Bestimme die Reihenfolge beim Ausfüllen der Struktur:
  - Beginne bei $S[n-1, m-1]$ und fülle das Array von rechts nach links und unten nach oben aus.

Schritt 3: Implementiere eine Lösung

```
import numpy as np

def two_dimensionality(a):
  m,n = a.shape
  S = np.zeros((m,n))
  S[m-1,n-1] = 0 # base case (redundant through np.zeros)

  for i in range(m-1, -1, -1):
      for j in range(n-1, -1, -1):
          if (i == m - 1) and (j == n - 1):
            continue

          south = [S[k] for k in range(i+1, min(i+a[i,j], m-1) + 1)]
          east = [S[k] for k in range(j+1, min(j+a[i,j], n-1) + 1)]
          S[i,j] = 1 + min(*south, *east)

  return S[0,0]
```

Problemzerlegung

Nun fügen wir den letzten Apsekt hinzu, und alle vier Probleme werden kombiniert um das ursprüngliche Frosch-Sprünge-Problem zu lösen.

1. Basisproblem
2. Variable Sprungdistanz
3. Zweidimensionalität
4. **Bananenschale**

Bananenschalen

- Im Eingabe-Array haben Bananenschalen den Wert -1.

Bananenschalen

- Im Eingabe-Array haben Bananenschalen den Wert -1.
- Wenn der Frosch auf einer Bananenschale landet, rutscht er aus und fällt auf das nächste Feld in süd-östlicher Richtung.

$$S[i, j] \to S[i + 1, j + 1]$$

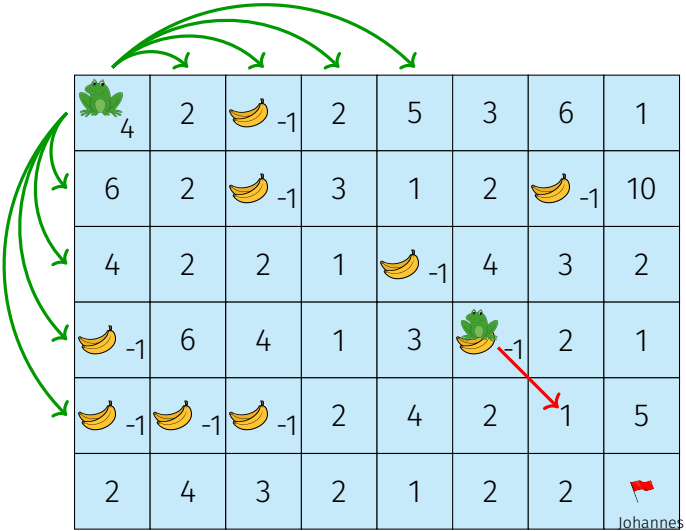
Bananenschalen

- Im Eingabe-Array haben Bananenschalen den Wert -1.
- Wenn der Frosch auf einer Bananenschale landet, rutscht er aus und fällt auf das nächste Feld in süd-östlicher Richtung.

$$S[i, j] \to S[i + 1, j + 1]$$

- Auf einer Bananenschale ausrutschen zählt nicht als Sprung.
- Bananenschalen komme nicht an den Enden des Arrays $i = m - 1$ und $j = n - 1$ vor. Dort würde ausrutschen zu out-of-bounds-Zugriff führen.

Aufgabenbeschreibung: Erinnerung



Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[n-1, m-1]$ zu gelangen.

Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[n-1, m-1]$ zu gelangen.
- Optionen: Nun hängen die Optionen davon ab, ob der Frosch auf einer Bananenschale landet.

Schritt 1: Identifiziere die Teilprobleme & Optionen

- Teilprobleme: unverändert. $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von Position $[i, j]$ zum Ziel $[n-1, m-1]$ zu gelangen.
- Optionen: Nun hängen die Optionen davon ab, ob der Frosch auf einer Bananenschale landet.
- Im Fall, dass $a[i, j] = -1$, müssen wir nichts vergleichen. Wir übernehmen den Wert des Felds, auf welches der Frosch rutscht.

Schritt 2: Definiere die Rekursion

$$S[i,j] = \begin{cases} 0 & \text{if } j = n-1 \text{ and } i = m-1 \\ S[i+1, j+1] & \text{if } a[i,j] = -1 \\ 1 + \min(*south, *east) & \text{else} \end{cases}$$

mit $south$, $east$ definiert wie bisher.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Unverändert, ein 2D-Array mit der gleichen Größe wie das Eingabe-Array.
    Neu: Kann positive Ganzzahlen *oder* -1 enthalten.

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Unverändert, ein 2D-Array mit der gleichen Größe wie das Eingabe-Array. Neu: Kann positive Ganzzahlen *oder* -1 enthalten.
- Identifiziere die Abhängigkeiten:
  - Unverändert

Schritt 3: Implementiere eine Lösung

- Finde eine geeignete Datenstruktur:
  - Unverändert, ein 2D-Array mit der gleichen Größe wie das Eingabe-Array. Neu: Kann positive Ganzzahlen *oder* -1 enthalten.
- Identifiziere die Abhängigkeiten:
  - Unverändert
- Bestimme die Reihenfolge beim Ausfüllen der Struktur:
  - Unverändert

Schritt 3: Implementiere eine Lösung

```
import numpy as np

def frog_jumps_full(a):
  m,n = a.shape
  S = np.zeros((m,n))
  S[m-1,n-1] = 0 # base case (redundant through np.zeros)
  for i in range(m-1, -1, -1):
      for j in range(n-1, -1, -1):
          if (i == m - 1) and (j == n - 1):
            continue
          if a[i, j] == -1:
            S[i, j] = S[i + 1, j + 1]
          else:
            south = [S[k] for k in range(i+1, min(i+a[i,j], m-1) + 1)]
            east = [S[k] for k in range(j+1, min(j+a[i,j], n-1) + 1)]
            S[i,j] = 1 + min(*south, *east)
  return S[0,0]
```

4. Zusammenfassung

Problemzerlegung: Elemente

Das Basisproblem kann mit jeder Kombination von Teilproblemen ergänzt werden, die jeweils ein zusätzliches Element in die DP-Lösung einführen:

Problemzerlegung: Elemente

Das Basisproblem kann mit jeder Kombination von Teilproblemen ergänzt werden, die jeweils ein zusätzliches Element in die DP-Lösung einführen:

- Variable Sprungdistanz: Eine **zusätzliche for-Schleife** wird benötigt, da die Anzahl der Optionen nicht mehr konstant ist.

Problemzerlegung: Elemente

Das Basisproblem kann mit jeder Kombination von Teilproblemen ergänzt werden, die jeweils ein zusätzliches Element in die DP-Lösung einführen:

- Variable Sprungdistanz: Eine **zusätzliche for-Schleife** wird benötigt, da die Anzahl der Optionen nicht mehr konstant ist.
- Zweidimensionalität: Das Problem wird zweidimensional, wir verwenden die Variablen i und j.

Problemzerlegung: Elemente

Das Basisproblem kann mit jeder Kombination von Teilproblemen ergänzt werden, die jeweils ein zusätzliches Element in die DP-Lösung einführen:

- Variable Sprungdistanz: Eine **zusätzliche for-Schleife** wird benötigt, da die Anzahl der Optionen nicht mehr konstant ist.
- Zweidimensionalität: Das Problem wird zweidimensional, wir verwenden die Variablen i und j.
- Bananenschalenfelder: Eine **zusätzliche if-Anweisung** wird benötigt, da sich die Optionen ändern können.

5. Hausaufgaben

Übung 10: DP II

Übung 10: DP II

- Mission Mars mit Lava
- Binomialkoeffizienten
- Dreieck
- Längste gemeinsame Teilsequenz

Abgabedatum: Montag 11.05.2026, 20:00 MEZ

KEINE HARDCODIERUNG

Fragen?

Feedback

https://jschultev.github.io/personal_website/Feedback