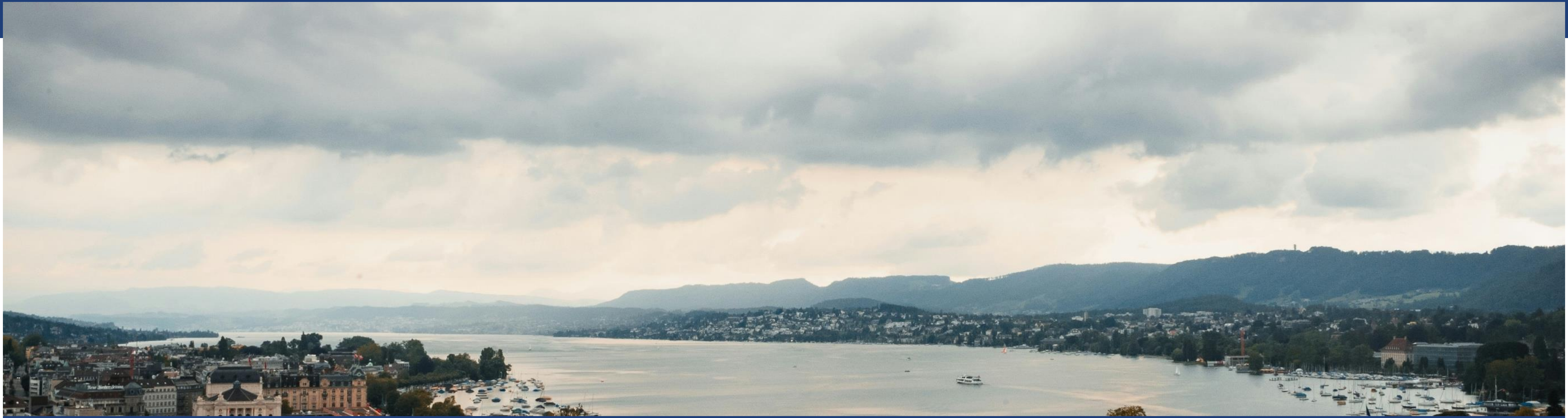


Übungslektion 10 – Dynamic Programming II

Informatik II

29. / 30. April 2025

Willkommen!

Polybox

Passwort: jschul

Personal Website

`https://n.ethz.ch/~jschul`

Heutiges Programm

- Wiederholung: Rod Cutting
- Limited Rod Cutting
- Frog Jumps
- Wrap-Up

1. Wiederholung: Rod Cutting

Wiederholung: Rod Cutting

Problemstellung:

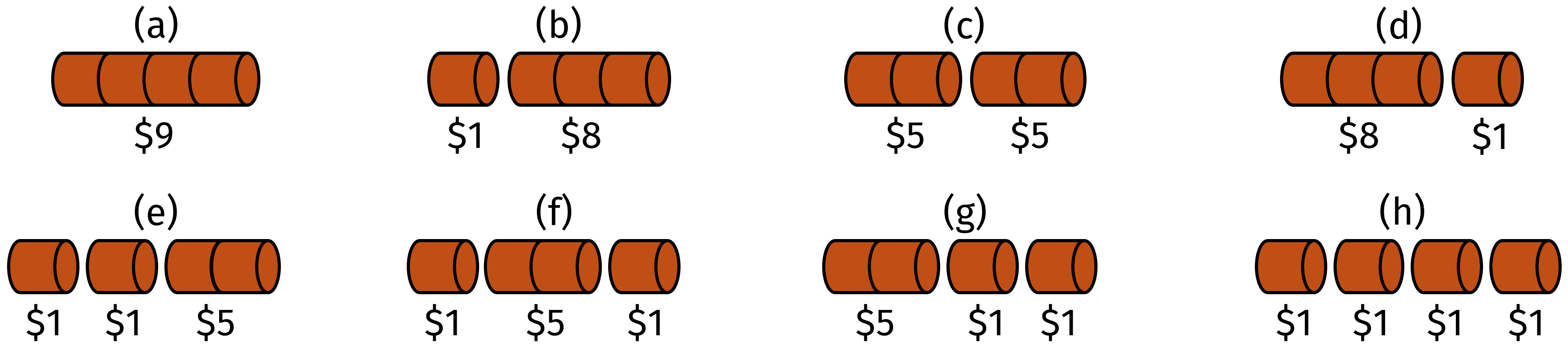
Eine Stange der Länge n soll in Teilstücke zerlegt werden, wobei jedes Stück der Länge i einen in der Preisliste p bestimmten Preis $p[i]$ hat.

- **Input**: Länge n und Preisliste p.
- **Ziel**: Die Stange so zerschneiden, dass der maximale Gesamtpreis erzielt wird.
- **Output**: Der maximale Erlös durch optimales Schneiden.

Wiederholung: Rod Cutting

Brute-Force Ansatz: Jede Möglichkeit berechnen

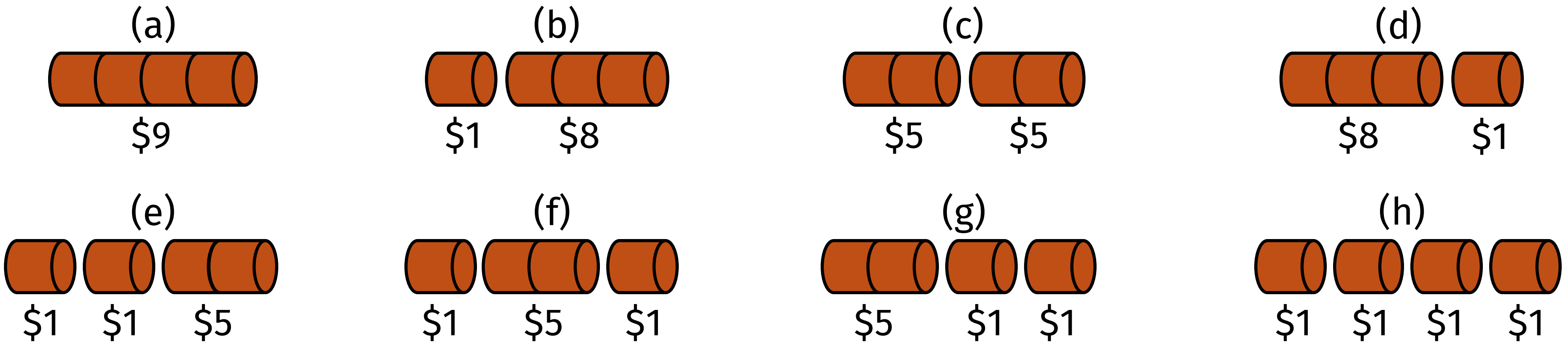
Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9



Wiederholung: Rod Cutting

Brute-Force Ansatz: Jede Möglichkeit berechnen

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



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

Wiederholung: Rod Cutting

Warum können wir bei Rod Cutting Dynamic Programming verwenden?

Wiederholung: Rod Cutting

Warum können wir bei Rod Cutting Dynamic Programming verwenden?

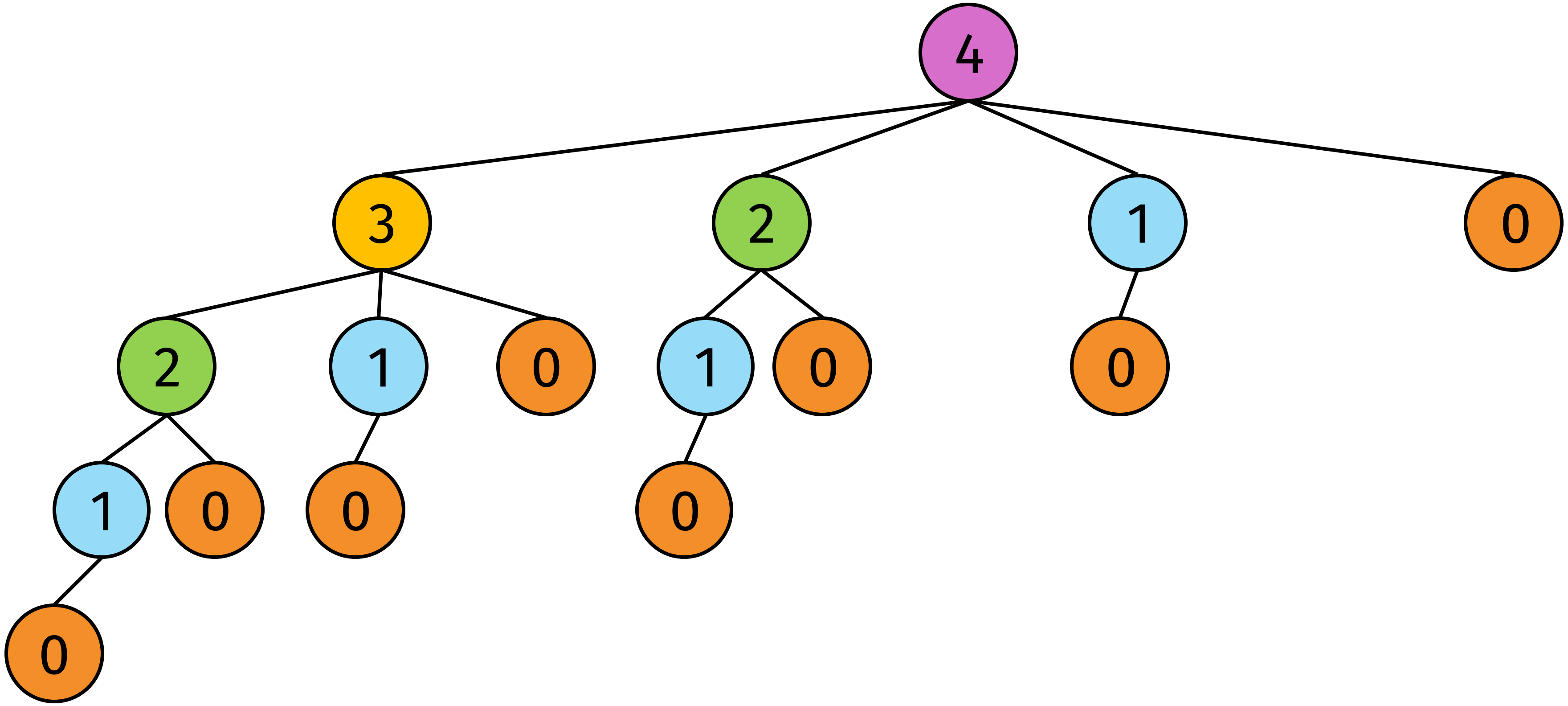
1. Überlappende Teilprobleme

Wiederholung: Rod Cutting

Warum können wir bei Rod Cutting Dynamic Programming verwenden?

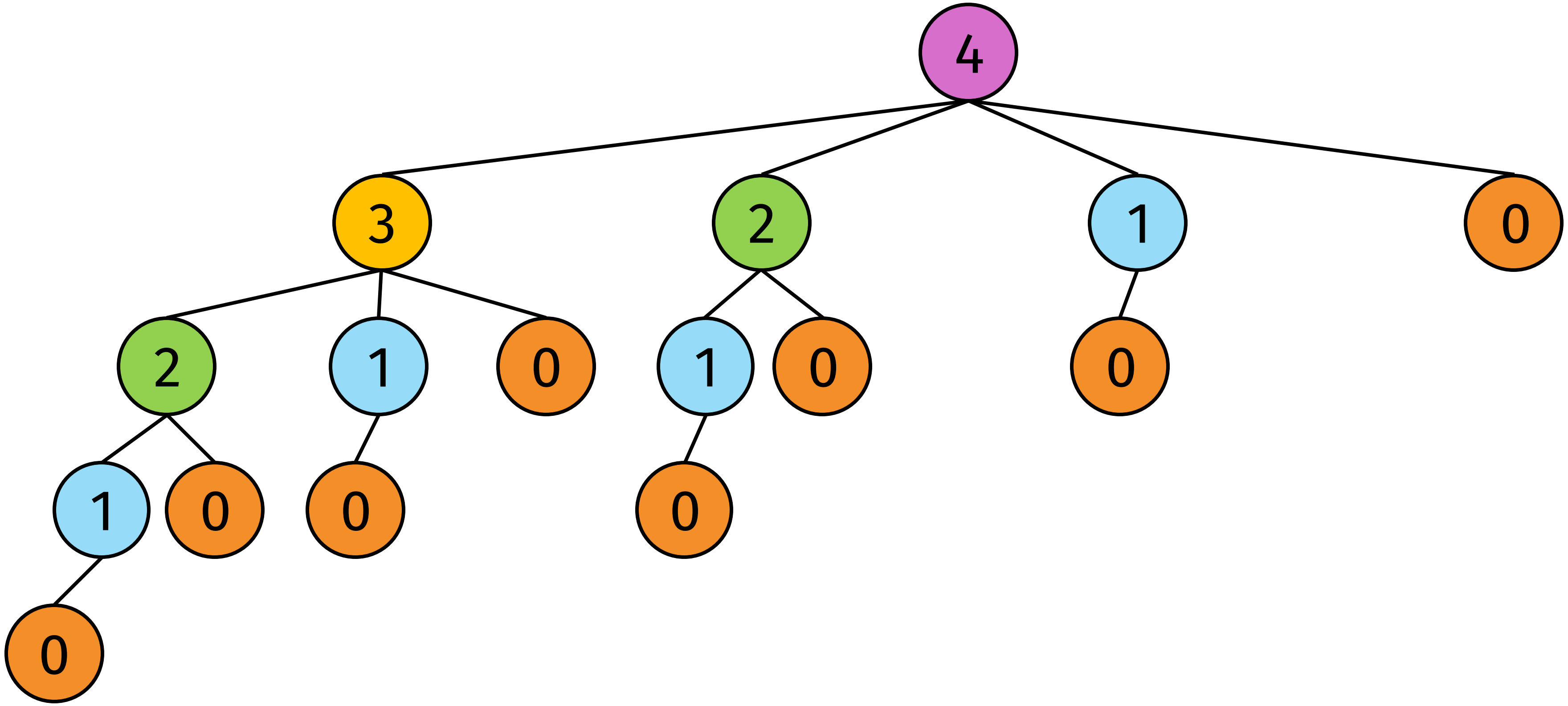
1. Überlappende Teilprobleme
2. Optimale Substruktur

Zur Erinnerung: Überlappende Teilprobleme



- Die Teilprobleme werden mehrfach berechnet. Dies resultiert in einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^n)$.

Zur Erinnerung: Überlappende Teilprobleme



- Die Teilprobleme werden mehrfach berechnet. Dies resultiert in einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^n)$.
- DP führt generell zu polynomieller Laufzeit. Hier: $\mathcal{O}(n^2)$.

Was ist eine Optimale Substruktur?

- Sei $S[5]$ der optimale Erlös für eine Stange der Länge 5.

Was ist eine Optimale Substruktur?

- Sei $S[5]$ der optimale Erlös für eine Stange der Länge 5.
- $S[5]$ hängt (unter anderem) von $S[4]$ ab.

Was ist eine Optimale Substruktur?

- Sei $S[5]$ der optimale Erlös für eine Stange der Länge 5.
- $S[5]$ hängt (unter anderem) von $S[4]$ ab.
- In Worten:

  "Der optimale Erlös für eine Stange der Länge 5 ($S[5]$) hängt vom optimalen Erlös für eine Stange der Länge 4 ($S[4]$) ab."

Was ist eine Optimale Substruktur?

“Der optimale Erlös für eine Stange der Länge 5 ($S[5]$) hängt vom optimalen Erlös für eine Stange der Länge 4 ($S[4]$) ab.”

Beweis durch Widerspruch:

1. Ist $S[5]$ optimal, so muss $S[4]$ ebenfalls optimal sein.

Was ist eine Optimale Substruktur?

“Der optimale Erlös für eine Stange der Länge 5 ($S[5]$) hängt vom optimalen Erlös für eine Stange der Länge 4 ($S[4]$) ab.”

Beweis durch Widerspruch:

1. Ist $S[5]$ optimal, so muss $S[4]$ ebenfalls optimal sein.
2. Wäre $S[4]$ nicht optimal, so könnte man einen höheren / optimaleren Erlös für $S[4]$ erhalten.

Was ist eine Optimale Substruktur?

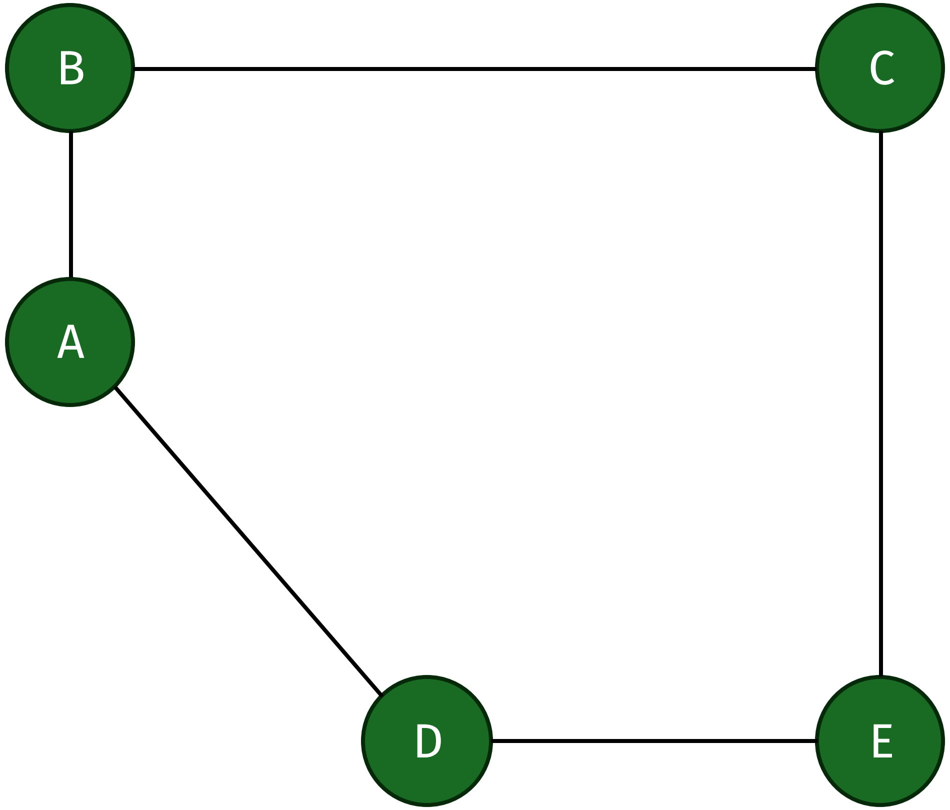
“Der optimale Erlös für eine Stange der Länge 5 ($S[5]$) hängt vom optimalen Erlös für eine Stange der Länge 4 ($S[4]$) ab.”

Beweis durch Widerspruch:

1. Ist $S[5]$ optimal, so muss $S[4]$ ebenfalls optimal sein.
2. Wäre $S[4]$ nicht optimal, so könnte man einen höheren / optimaleren Erlös für $S[4]$ erhalten.
3. Als Konsequenz könnte man auch einen höheren Erlös für $S[5]$ erhalten. Somit ist $S[5]$ nicht optimal.

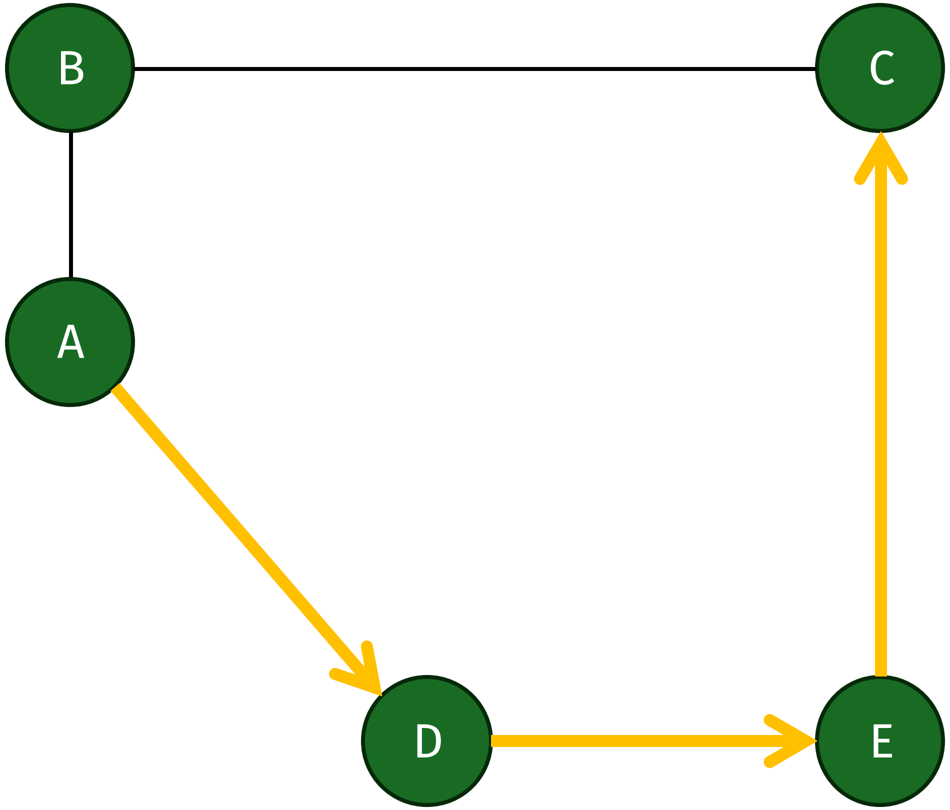
Beispiel: Nicht-Optimale Substruktur

- Gesucht ist der längste geometrische Weg von A nach C.



Beispiel: Nicht-Optimale Substruktur

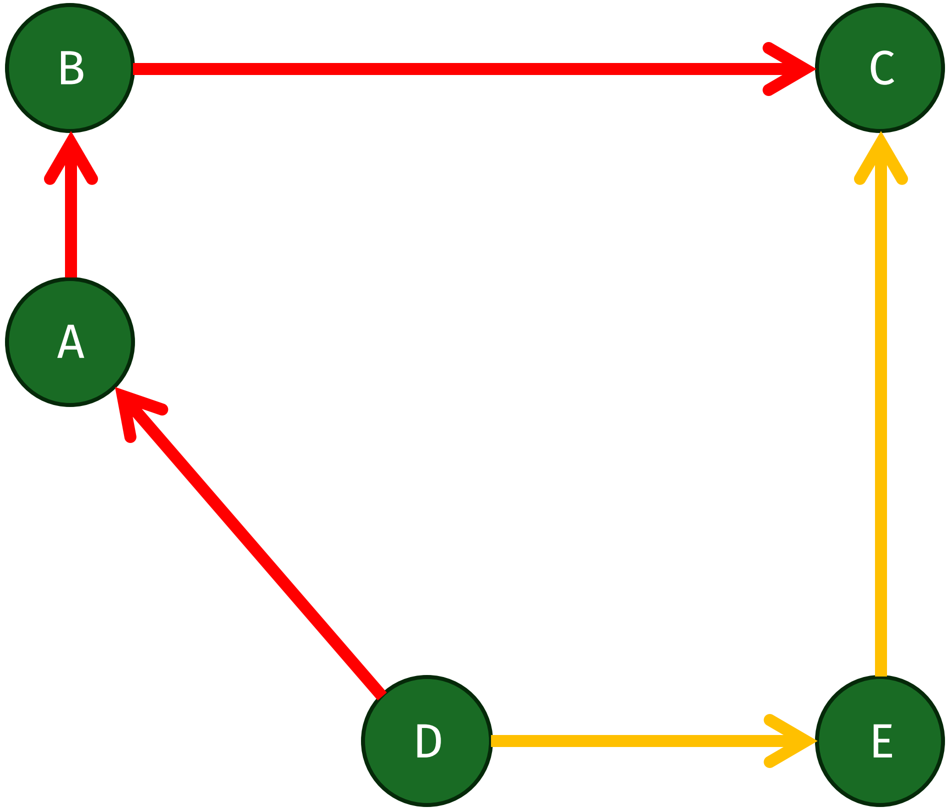
- Gesucht ist der längste geometrische Weg von A nach C.



- Der Weg kann in A-D und D-C aufgeteilt werden. (Teilprobleme)

Beispiel: Nicht-Optimale Substruktur

- Der optimale Weg von D-C ist aber nicht Teil vom optimalen Weg von A-C.



Wiederholung: Rod Cutting

Warum können wir Dynamic Programming verwenden?

1. Überlappende Teilprobleme

Wiederholung: Rod Cutting

Warum können wir Dynamic Programming verwenden?

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

Wiederholung: Rod Cutting

Warum können wir Dynamic Programming verwenden?

1. Überlappende Teilprobleme
   - Ohne Überlappungen würde der DP Algorithmus ebenfalls in exponentieller Zeit laufen $\mathcal{O}(2^n)$.
     → DP führt nicht bedingungslos zu polynomieller Laufzeit.
2. Optimale Substruktur

Wiederholung: Rod Cutting

Warum können wir Dynamic Programming verwenden?

1. Überlappende Teilprobleme
   - Ohne Überlappungen würde der DP Algorithmus ebenfalls in exponentieller Zeit laufen $\mathcal{O}(2^n)$.
     → DP führt nicht bedingungslos zu polynomieller Laufzeit.
2. Optimale Substruktur
   - Ohne optimale Substruktur kann die Lösung nicht mit DP berechnet werden.

2. Limited Rod Cutting

Was bedeutet “Limited”

Unlimited:

- Die Stange kann beliebig oft geschnitten werden.

Limited:

- Die Stange darf höchstens c mal geschnitten werden.

Was bedeutet "Limited"

Unlimited:

- Die Stange kann beliebig oft geschnitten werden.

Limited:

- Die Stange darf höchstens c mal geschnitten werden.

→ Der maximale Erlös einer Stange der Länge 20 unterscheidet sich, je nachdem, ob man sie bis zu 10-mal oder nur 2-mal schneiden darf.

Drei Schritte von Dynamic Programming

1) Teilprobleme & Optionen identifizieren

Drei Schritte von Dynamic Programming

1) Teilprobleme & Optionen identifizieren
2) Rekurrenz definieren

Drei Schritte von Dynamic Programming

1) Teilprobleme & Optionen identifizieren
2) Rekurrenz definieren
3) Lösung implementieren
   - Geeignete Datenstruktur definieren
   - Abhängigkeit identifizieren
   - Richtung des Ausfüllens bestimmen

Schritt 1: Teilprobleme & Optionen identifizieren

- $S[i]$ war der optimale Erlös für eine Stange der Länge i.

Schritt 1: Teilprobleme & Optionen identifizieren

- $S[i]$ war der optimale Erlös für eine Stange der Länge i.
- Neu hängt der optimale Erlös auch davon ab, wie viele Schnitte noch zur Verfügung stehen.

→ $S[i,0], \ S[i,1], \ S[i,2], \ \dots, \ S[i,c]$

Schritt 1: Teilprobleme & Optionen identifizieren

- $S[i, t]$ bezeichnet den optimalen Erlös, den man mit einer Stange der Länge i und t verbleibenden Schnitten erzielen kann.

Schritt 1: Teilprobleme & Optionen identifizieren

- $S[i,t]$ bezeichnet den optimalen Erlös, den man mit einer Stange der Länge i und t verbleibenden Schnitten erzielen kann.

→ **Zweidimensionale Teilprobleme!**

Schritt 1: Teilprobleme & Optionen identifizieren

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

Schritt 1: Teilprobleme & Optionen identifizieren

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

- Option 1: Schneiden nach Länge 1

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

Schritt 1: Teilprobleme & Optionen identifizieren

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

- Option 1: Schneiden nach Länge 1
  $$S[i,t] \;=\; p[1] \;+\; S[i-1,t-1]$$
- Option 2: Schneiden nach Länge 2
  $$S[i,t] \;=\; p[2] \;+\; S[i-2,t-1]$$

Schritt 1: Teilprobleme & Optionen identifizieren

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

- Option 1: Schneiden nach Länge 1
  $$S[i,t] \;=\; p[1] \;+\; S[i-1,t-1]$$
- Option 2: Schneiden nach Länge 2
  $$S[i,t] \;=\; p[2] \;+\; S[i-2,t-1]$$
- Option 3: Schneiden nach Länge 3
  $$S[i,t] = p[3] + S[i-3,t-1]$$

Schritt 1: Teilprobleme & Optionen identifizieren

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

- Option 1: Schneiden nach Länge 1
  $$S[i, t] \;=\; p[1] \;+\; S[i-1, t-1]$$
- Option 2: Schneiden nach Länge 2
  $$S[i, t] \;=\; p[2] \;+\; S[i-2, t-1]$$
- Option 3: Schneiden nach Länge 3
  $$S[i, t] = p[3] + S[i-3, t-1]$$

  [...]

- Option i: Schneiden nach Länge i
  $$S[i, t] = p[i] + S[0, t-1]$$

Schritt 2: Rekurrenz definieren

- Von allen Optionen wählen wir jene mit dem höchsten Erlös.

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

Schritt 2: Rekurrenz definieren

Was war der Basisfall bei Unlimited Rod Cutting?

Schritt 2: Rekurrenz definieren

Was war der Basisfall bei Unlimited Rod Cutting?

- $S[0] = 0$, für eine Stange der Länge 0 erhält man keinen Erlös.

Schritt 2: Rekurrenz definieren

Was war der Basisfall bei Unlimited Rod Cutting?

- $S[0] = 0$, für eine Stange der Länge 0 erhält man keinen Erlös.
  → Dies ändert sich auch nicht mit der Anzahl an verfügbaren Schnitten.

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

Schritt 2: Rekurrenz definieren

Was war der Basisfall bei Unlimited Rod Cutting?

- $S[0] = 0$, für eine Stange der Länge 0 erhält man keinen Erlös.
  → Dies ändert sich auch nicht mit der Anzahl an verfügbaren Schnitten.

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

Was passiert, wenn keine Schnitte mehr verbleiben?

Schritt 2: Rekurrenz definieren

Was war der Basisfall bei Unlimited Rod Cutting?

- $S[0] = 0$, für eine Stange der Länge 0 erhält man keinen Erlös.
  → Dies ändert sich auch nicht mit der Anzahl an verfügbaren Schnitten.

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

Was passiert, wenn keine Schnitte mehr verbleiben?

- Es kann nur noch der Erlös für das verbleibende Stück in seiner Gesamtlänge erzielt werden.

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

Schritt 2: Rekurrenz definieren

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

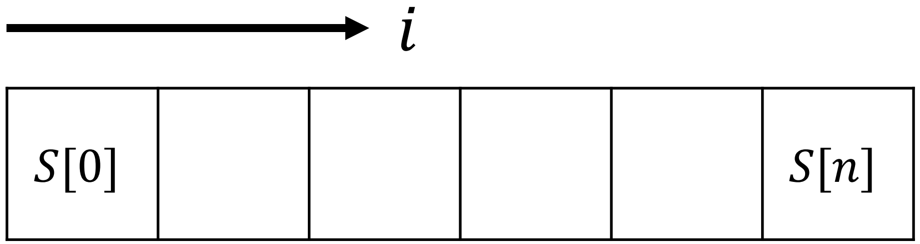
Schritt 2: Rekurrenz definieren

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

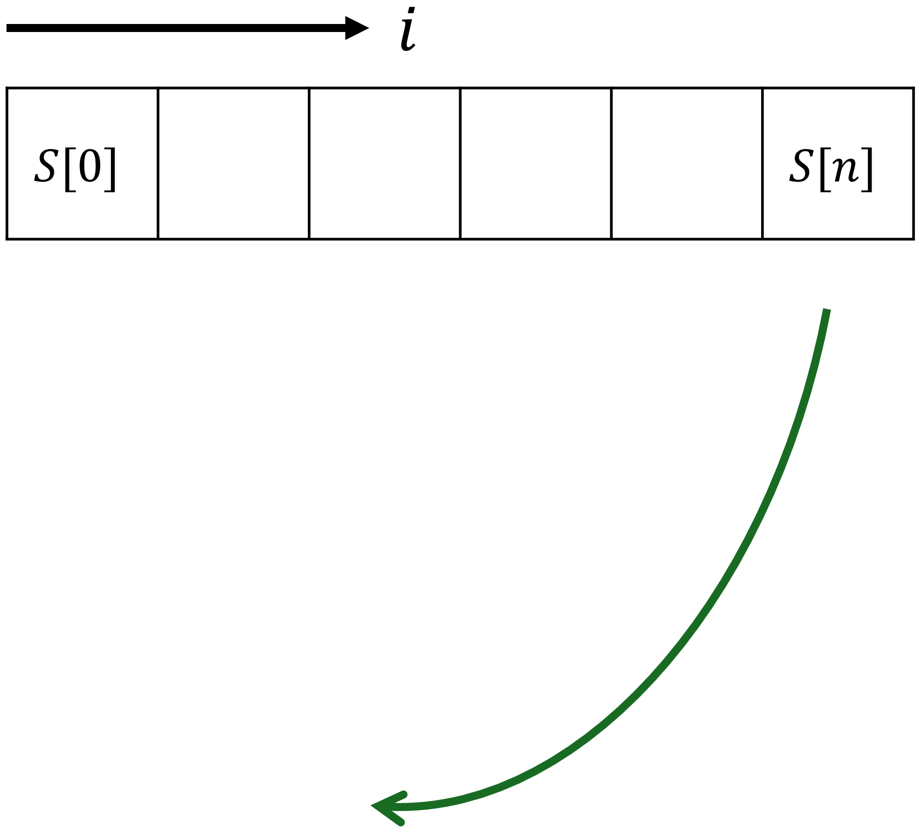
- Bemerkung: da $p[0] = 0$ können die beiden Basisfälle zusammengelegt werden.

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

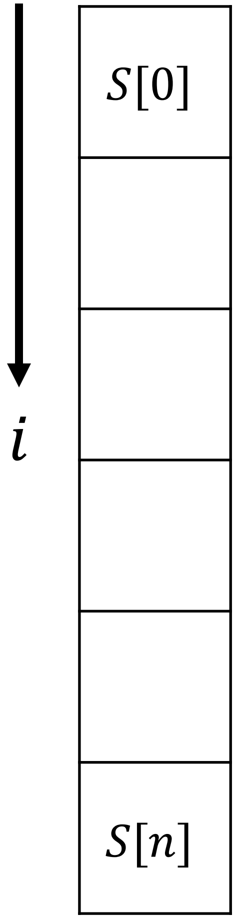
Schritt 2: Graphisch



Schritt 2: Graphisch



Schritt 2: Graphisch



Schritt 2: Graphisch

$t \rightarrow$

$i \downarrow$

$S[0,0]$	0	0	0	0	$S[0,c]$
$p[1]$					
$p[2]$					
…					
$p[n-1]$					
$S[n,0]$					$S[n,c]$

Basisfälle

Ziel

Schritt 2: Graphisch

t

i

$S[0,0]$	0	0	0	0	$S[0,c]$
$p[1]$					
$p[2]$					
…	$S[i,t]$				
$p[n-1]$				$S[i,t]$	
$S[n,0]$					$S[n,c]$

Abhängigkeiten

Abhängigkeiten

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 2D-Array der Grösse $(n+1) \times (c+1)$.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 2D-Array der Grösse $(n+1) \times (c+1)$.

- Abhängigkeiten identifizieren:
  - Um $S[i,t]$ zu berechnen, müssen $S[i-1,t-1], S[i-2,t-1], \dots, S[0,t-1]$ bekannt sein.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 2D-Array der Grösse $(n + 1) \times (c + 1)$.
- Abhängigkeiten identifizieren:
  - Um $S[i, t]$ zu berechnen, müssen $S[i - 1, t - 1], S[i - 2, t - 1], \ldots, S[0, t - 1]$ bekannt sein.
- Richtung des Ausfüllens bestimmen:
  - Beginne bei $S[0,0]$ und fülle das Array von links nach rechts und von oben nach unten aus.

Schritt 3: Dynamic Programming

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

  S = np.zeros((n+1,c+1)) #Erledigt auch gleich den ersten Basisfall
  S[:,0] = p #Zweiter 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]
```

Schritt 3: Memoisierung

```python
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 betrachtet 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)

- Die Stange nicht zu schneiden ist ebenfalls eine Option.
- Das Nicht-Schneiden wird durch den Fall $k = i$ abgedeckt.
- Die Stange wird in ein Stück der Länge i und ein Stück der Länge 0 „geteilt“.
- Dabei zählt der Code ein Schnitt, obwohl faktisch keiner erfolgt.
- Da die Rekursion endet, beeinflusst der zusätzliche Schnitt das Optimierungsergebnis nicht.
- Eine korrekte Rekonstruktion der tatsächlichen Schnittfolge erfordert jedoch eine Anpassung des Algorithmus.

3. Frog Jumps

Problemstellung

- **Input**: Ein 2D-Array a mit Ganzzahlen aus der Menge $[-1] \cup [1, \infty)$.

Problemstellung

- **Input**: Ein 2D-Array a mit Ganzzahlen aus der Menge $[-1] \cup [1, \infty)$.
- **Ziel**: Ein Frosch startet in der linken oberen Ecke und muss mit möglichst wenigen Sprüngen zur rechten unteren Ecke gelangen, unter Berücksichtigung der Regeln auf der folgenden Folie.

Problemstellung

- **Input**: Ein 2D-Array a mit Ganzzahlen aus der Menge $[-1] \cup [1, \infty)$.

- **Ziel**: Ein Frosch startet in der linken oberen Ecke und muss mit möglichst wenigen Sprüngen zur rechten unteren Ecke gelangen, unter Berücksichtigung der Regeln auf der folgenden Folie.

- **Output**: Die minimale Anzahl an Sprüngen, die erforderlich sind, um von links oben nach rechts unten zu kommen.

Problemstellung: Regeln

- Freies Feld ($a[i, j] > 0$):

Problemstellung: Regeln

- Freies Feld ($a[i,j] > 0$):
  - Der Frosch kann entweder nach Osten oder Süden springen, wobei er maximal $1, 2, 3, \dots, a[i,j]$ Felder weit springen darf.

Problemstellung: Regeln

- Freies Feld ($a[i,j] > 0$):
  - Der Frosch kann entweder nach Osten oder Süden springen, wobei er maximal $1, 2, 3, \dots, a[i,j]$ Felder weit springen darf.

- Bananenschale ($a[i,j] = -1$):

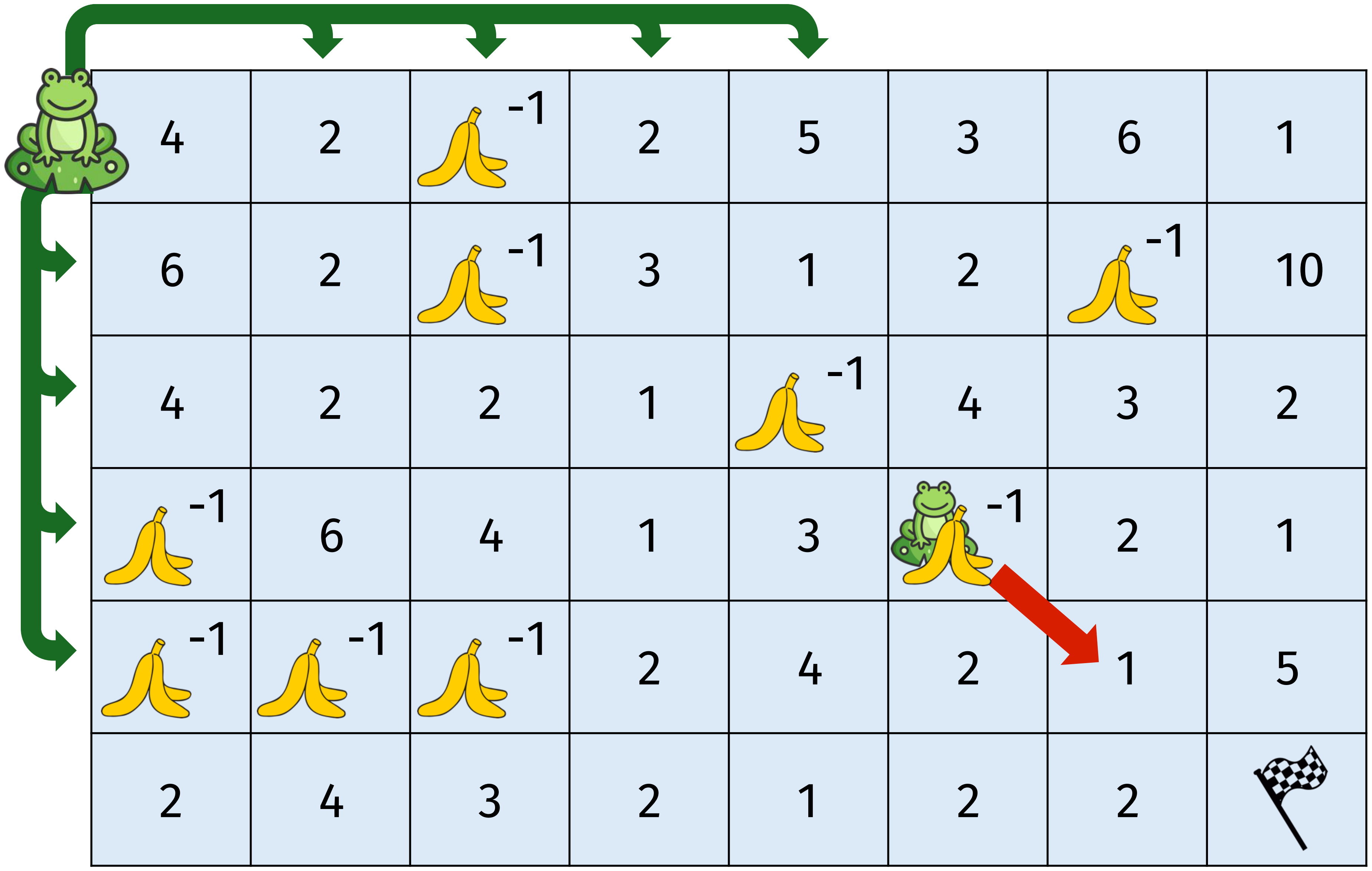
Problemstellung: Regeln

- Freies Feld ($a[i,j] > 0$):
  - Der Frosch kann entweder nach Osten oder Süden springen, wobei er maximal $1, 2, 3, \dots, a[i,j]$ Felder weit springen darf.
- Bananenschale ($a[i,j] = -1$):
  - Der Frosch rutscht auf der Bananenschale aus und landet ein Feld weiter süd-östlich (diagonal).

Problemstellung: Regeln

- Freies Feld ($a[i,j] > 0$):
  - Der Frosch kann entweder nach Osten oder Süden springen, wobei er maximal $1, 2, 3, \dots, a[i,j]$ Felder weit springen darf.
- Bananenschale ($a[i,j] = -1$):
  - Der Frosch rutscht auf der Bananenschale aus und landet ein Feld weiter süd-östlich (diagonal).
  - Das Ausrutschen zählt <u>nicht</u> als Sprung.

Problemstellung: Beispiel



Problemzerlegung

Das Problem ist sehr komplex. Es kann jedoch in Teilprobleme zerlegt werden, die unabhängig gelöst und kombiniert werden können.

1) Basisproblem
2) Variable Sprungweite
3) Zweidimensionalität
4) Bananenschale

Problemzerlegung

Wir erhalten unser Basisproblem, indem wir die Bananenschalen, die zweite Dimension und die variable Sprungweite entfernen.

1) Basisproblem ⟵
2) Variable Sprungweite
3) Zweidimensionalität
4) Bananenschale

Basisproblem

- **Input**: Ein 1D-Array a mit positiven Ganzzahlen $(a[j] > 0)$.

Basisproblem

- **Input**: Ein 1D-Array a mit positiven Ganzzahlen $(a[j] > 0)$.

- **Ziel**: Der Frosch muss mit möglichst wenigen Sprüngen von Index 0 bis zum Index $m - 1$ gelangen. Der Frosch hat zwei Optionen:
  - Er kann genau ein Feld nach vorne springen.
  - Er kann so viele Felder weit springen, wie der Wert des aktuellen Feldes $a[j]$ angibt.

Basisproblem

- **Input**: Ein 1D-Array a mit positiven Ganzzahlen $(a[j] > 0)$.

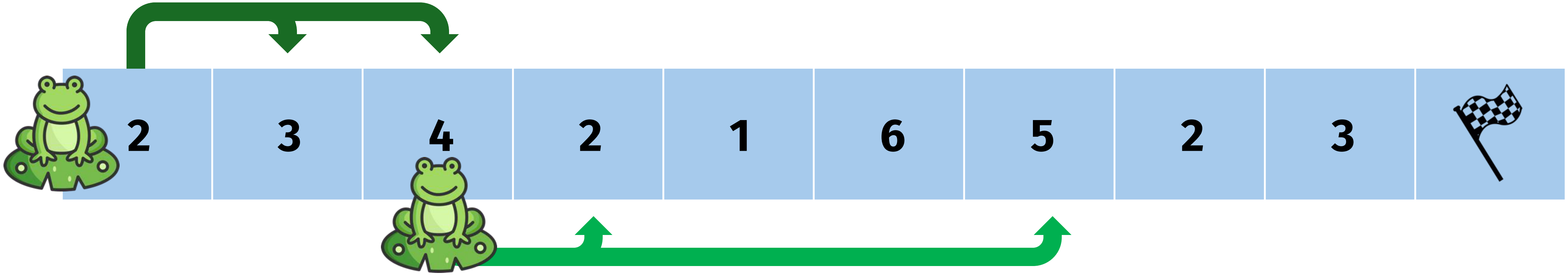
- **Ziel**: Der Frosch muss mit möglichst wenigen Sprüngen von Index 0 bis zum Index $m - 1$ gelangen. Der Frosch hat zwei Optionen:
  - Er kann genau ein Feld nach vorne springen.
  - Er kann so viele Felder weit springen, wie der Wert des aktuellen Feldes $a[j]$ angibt.

- **Output**: Die minimale Anzahl an Sprüngen, die erforderlich sind, um von links nach rechts zu kommen.

Basisproblem: Bemerkung

- Nicht alle DP-Probleme können in Teilprobleme zerlegt werden.
- Insbesondere 2D-Probleme (z.B. King's Way) haben im Allgemeinen kein 1D-Basisproblem.

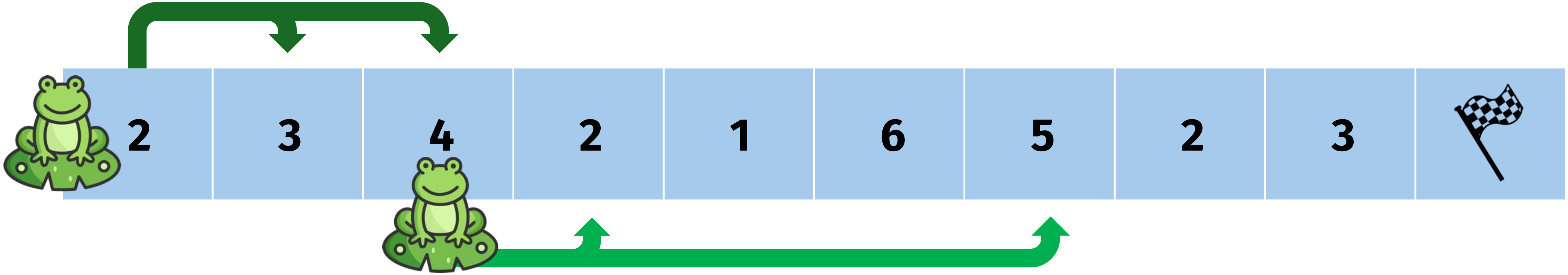
Basisproblem: Beispiel



Drei Schritte von Dynamic Programming

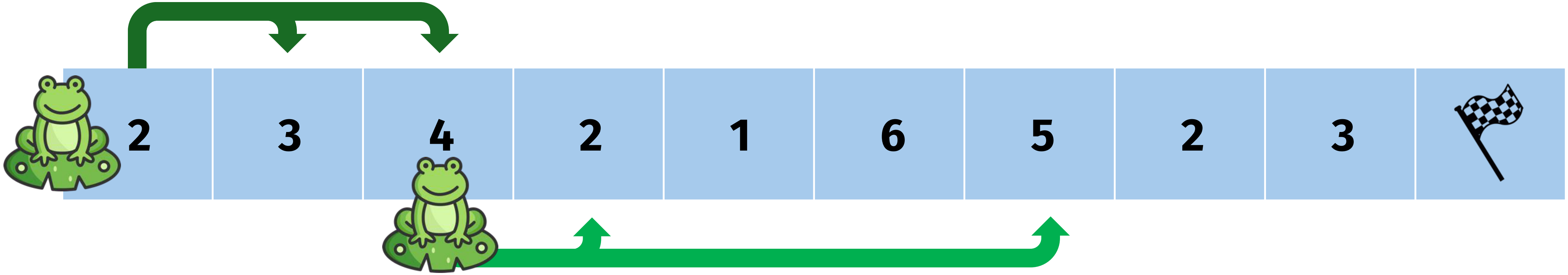
1) Teilprobleme & Optionen identifizieren
2) Rekurrenz definieren
3) Lösung implementieren
   - Geeignete Datenstruktur definieren
   - Abhängigkeit identifizieren
   - Richtung des Ausfüllens bestimmen

Schritt 1: Teilprobleme & Optionen identifizieren



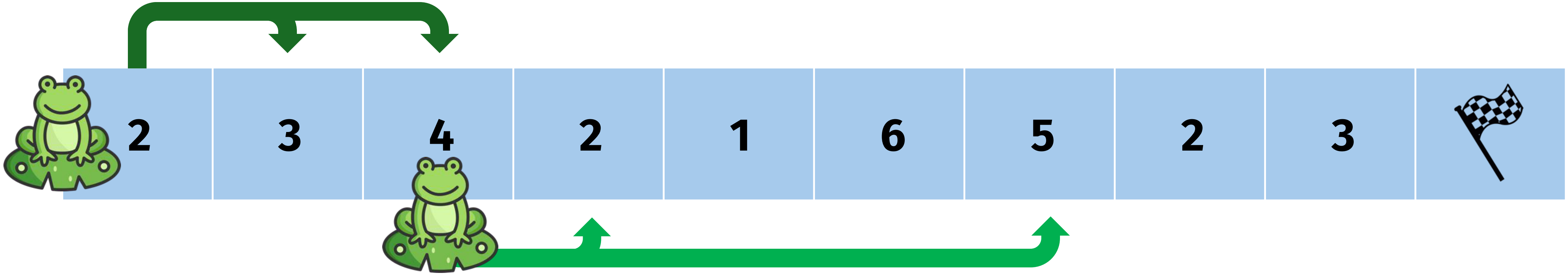
- Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um von Index j zu Index $m-1$ zu gelangen.

Schritt 1: Teilprobleme & Optionen identifizieren



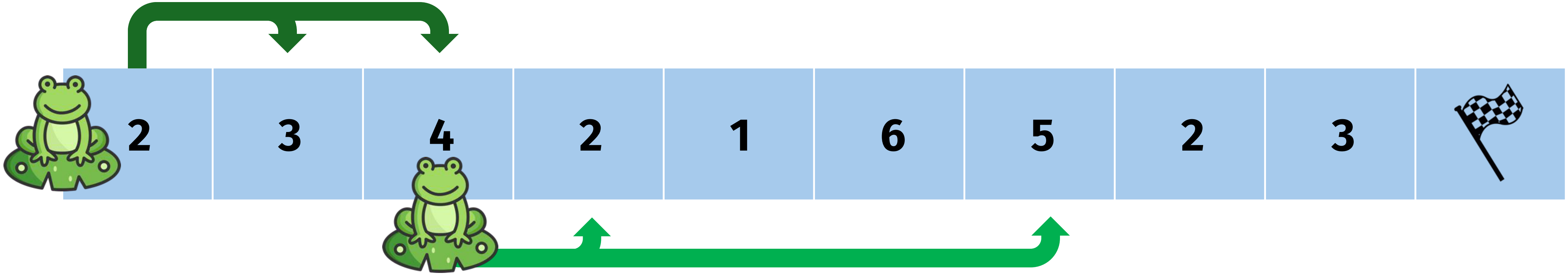
- Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um von Index j zu Index $m-1$ zu gelangen.
- Gesucht ist $S[0]$, die minimale Anzahl an Sprüngen vom Beginn des Arrays aus.

Schritt 1: Teilprobleme & Optionen identifizieren



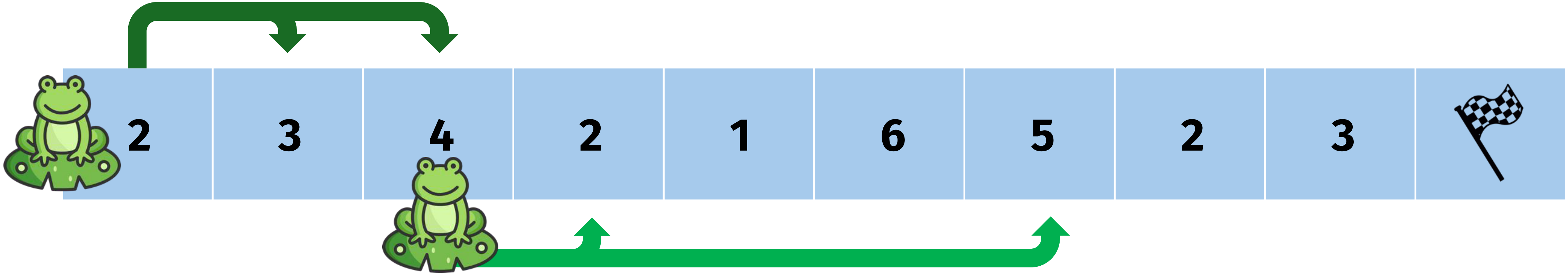
- Welche Optionen hat der Frosch bei Index j?

Schritt 1: Teilprobleme & Optionen identifizieren



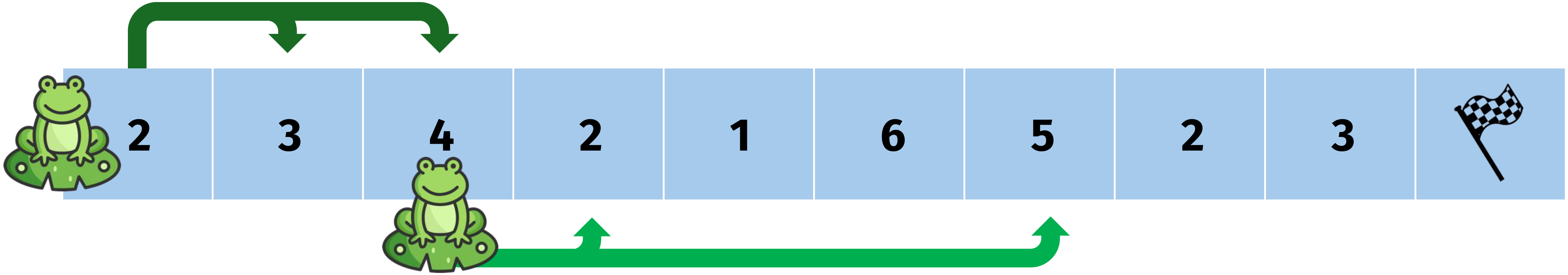
- Welche Optionen hat der Frosch bei Index j?
  - **Option A**: 1 Feld weiter springen

Schritt 1: Teilprobleme & Optionen identifizieren



- Welche Optionen hat der Frosch bei Index j?
  - **Option A**: 1 Feld weiter springen
  - **Option B**: $a[j]$ Felder weiter springen

Schritt 1: Teilprobleme & Optionen identifizieren



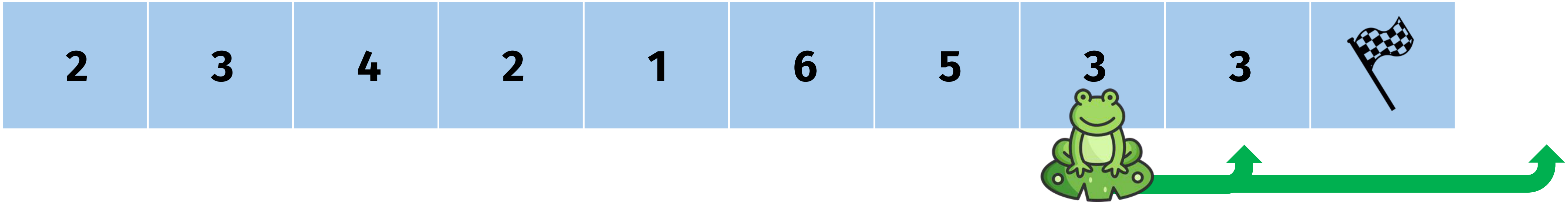
- Welche Optionen hat der Frosch bei Index j?
  - **Option A**: 1 Feld weiter springen
  - **Option B**: $a[j]$ Felder weiter springen

→ Der Frosch wählt die bessere Option.

Schritt 2: Rekurrenz definieren

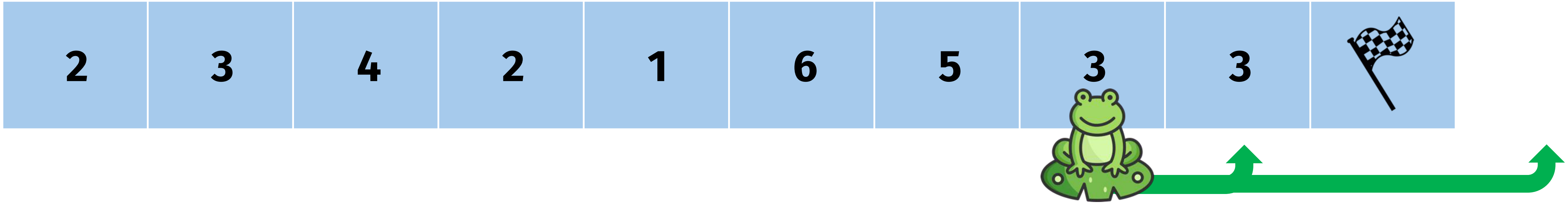
$$S[j] = \begin{cases} 0 & \text{if } j = m - 1 \text{ (Basisfall)} \\ 1 + \min(S[j+1], S[j + a[j]]) & \text{else} \end{cases}$$

Randfall



- Problem: Der Frosch springt über das Ziel hinaus!
  → Wir erhalten einen **IndexError: out of bounds**.

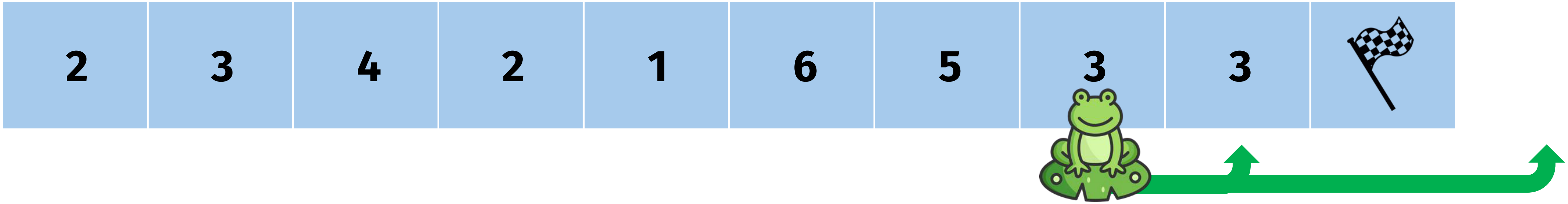
Randfall



- Problem: Der Frosch springt über das Ziel hinaus!
  → Wir erhalten einen **IndexError: out of bounds**.

Wie lösen wir das?

Randfall



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

- Dieser Ausdruck gibt $m - 1$ zurück, sollte der Sprung ungültig sein.

Schritt 2: Rekurrenz definieren

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

$$\Downarrow$$

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

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array um die Teilprobleme $S[j]$ zu speichern.

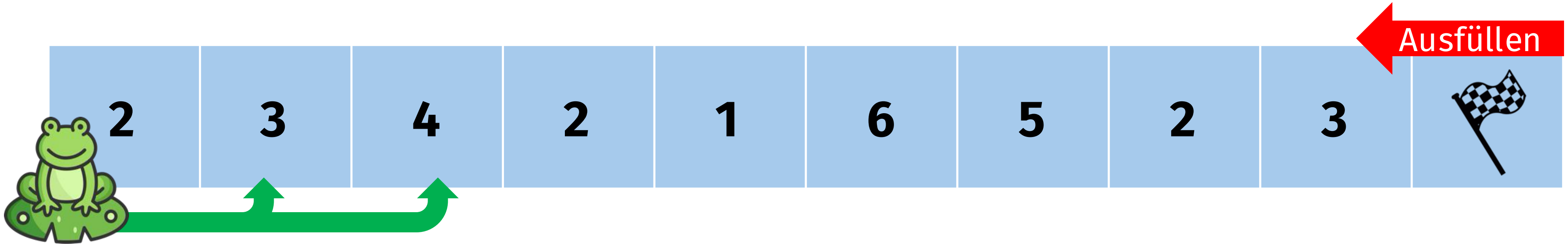
Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array um die Teilprobleme $S[j]$ zu speichern.

- Abhängigkeit identifizieren:
  - Zur Berechnung von $S[j]$ müssen die Werte von $S[j+1]$ und $S[j+a[i]]$ bekannt sein.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array um die Teilprobleme $S[j]$ zu speichern.
- Abhängigkeit identifizieren:
  - Zur Berechnung von $S[j]$ müssen die Werte von $S[j+1]$ und $S[j+a[i]]$ bekannt sein.
- Richtung des Ausfüllens bestimmen:
  - Die Werte müssen demnach **von rechts nach links** berechnet werden.



Schritt 3: Lösung implementieren

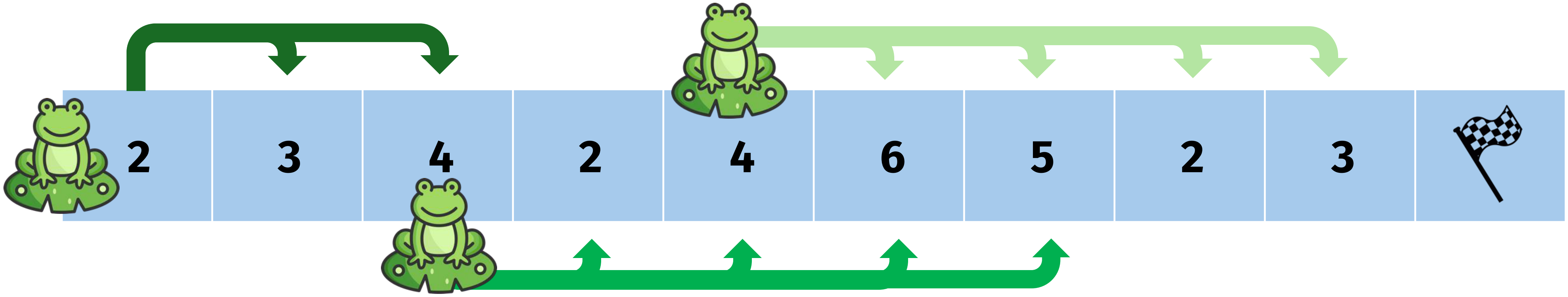
```python
import numpy as np

def base_problem(a):

  m = len(a)
  S = np.zeros(m)
  S[-1] = 0 #Basisfall (redundant durch np.zeros)

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

  return S[0]
```

Problemzerlegung

- Es besteht nun ein Verständnis des Basisproblems, sodass weitere Aspekte hinzugefügt werden können.

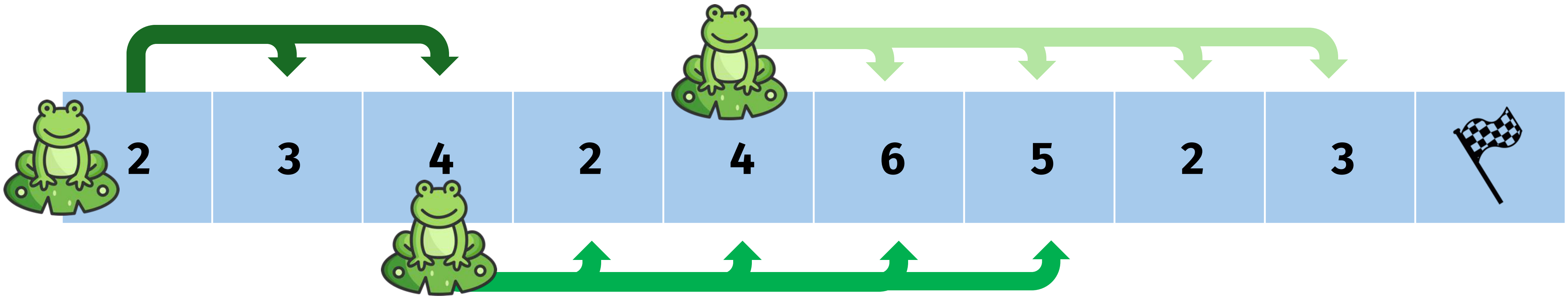
  1) Basisproblem
  2) Variable Sprungweite
  3) Zweidimensionalität
  4) Bananenschale

Variable Sprungweite



- Nun kann der Frosch nicht nur 1 oder a[j] Felder weit springen, sondern eine beliebige Distanz zwischen 1 und $a[j]$.

Variable Sprungweite



- Nun kann der Frosch nicht nur 1 oder a[j] Felder weit springen, sondern eine beliebige Distanz zwischen 1 und $a[j]$.
- Einen Grossteil der Basisproblem-Lösung kann beibehalten werden.

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme:
  - Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um vom Index j den Index $m-1$ zu erreichen.

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme:
  - Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um vom Index j den Index $m-1$ zu erreichen.
- Optionen:
  - Der Frosch hat a[j] Optionen und kann eine Distanz von 1, 2, …, $a[j]$ springen.

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme:
  - Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um vom Index j den Index $m-1$ zu erreichen.

- Optionen:
  - Der Frosch hat $a[\mathrm{j}]$ Optionen und kann eine Distanz von 1, 2, ..., $a[j]$ springen.
  - Um die beste Option zu finden, müssen alle geprüft werden.

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme:
  - Sei $S[j]$ die minimale Anzahl an Sprüngen, die der Frosch benötigt, um vom Index j den Index $m-1$ zu erreichen.

- Optionen:
  - Der Frosch hat $a[\mathrm{j}]$ Optionen und kann eine Distanz von 1, 2, ..., $a[j]$ springen.
  - Um die beste Option zu finden, müssen alle geprüft werden.

Wie prüfen wir alle Optionen?

Schritt 2: Rekurrenz definieren

→ Wir überprüfen alle Optionen mittels for-Schleife.

Schritt 2: Rekurrenz definieren

→ Wir überprüfen alle Optionen mittels for-Schleife.

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

$$\Downarrow$$

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

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array, wie beim Basisproblem.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array, wie beim Basisproblem.

- Abhängigkeit identifizieren:
  - Zur Berechnung von $S[j]$ müssen die Teilprobleme $S[j+1]$, $S[j+2]$, ..., $S[j+a[i]]$ gelöst sein.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Ein 1D-Array, wie beim Basisproblem.
- Abhängigkeit identifizieren:
  - Zur Berechnung von $S[j]$ müssen die Teilprobleme $S[j+1]$, $S[j+2]$, ..., $S[j+a[i]]$ gelöst sein.
- Richtung des Ausfüllens bestimmen:
  - Das Array wird von rechts nach links ausgefüllt, wie beim Basisproblem.

Schritt 3: Lösung implementieren

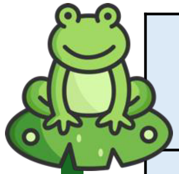
```
import numpy as np

def variable_jumping_distance(a):

  m = len(a)
  S = np.zeros(m)
  S[-1] = 0 #Basisfall (redundant durch np.zeros)

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

  return S[0]
```

Problemzerlegung

Nächster Schritt: Basisproblem + Variable Sprungweite + Zweidimensionalität.

1) Basisproblem
2) Variable Sprungweite
3) Zweidimensionalität
4) Bananenschale

Der Input ist neu ein 2D-Array a. Der Frosch kann nun nach Süden springen.



4	3	2	2	3	3	2	4
6	2	1	5	4	2	2	5
7	2	3	4	3	1	2	2
3	2	5	6	3	3	2	3
2	5	3	2	2	4	4	3
3	2	1	2	3	3	2	

Schritt 1: Teilprobleme & Optionen identifizieren

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

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme: $S[i, j]$ bezeichnet die minimale Anzahl an Sprüngen, um von der Position $[i, j]$ das Ziel $[n-1, m-1]$ zu erreichen.
- Optionen: Durch die neue Dimension kann der Frosch auch nach Süden springen. Das heisst, er kann 1 bis $a[i, j]$ Felder weit springen, entweder nach rechts oder nach unten.

Schritt 2: Rekurrenz definieren

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

Mit

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

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Neue Dimension → ein 2D-Array mit der gleichen Grösse wie das Input Array.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Neue Dimension → ein 2D-Array mit der gleichen Grösse wie das Input Array.

- Abhängigkeit identifizieren:
  - Um $S[i,j]$ zu berechnen müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i,j+1], S[i,j+2], \ldots, S\big[i,j+a[i,j]\big]$
    - Im Süden: $S[i+1,j], S[i+2,j], \ldots, S[i+a[i,j],j]$

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Neue Dimension → ein 2D-Array mit der gleichen Grösse wie das Input Array.

- Abhängigkeit identifizieren:
  - Um $S[i, j]$ zu berechnen müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i, j+1], S[i, j+2], \ldots, S\big[i, j + a[i, j]\big]$
    - Im Süden: $S[i+1, j], S[i+2, j], \ldots, S[i + a[i, j], j]$

- Richtung des Ausfüllens:
  - Beginne bei $S[n-1, m-1]$ und fülle das Array von rechts nach links und von unten nach oben aus.

Schritt 3: Lösung implementieren

```
import numpy as np

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

   for i in range(n-1, -1, -1):
       for j in range(m-1, -1, -1):
           best_east = min(S[i, min(j+k, m-1)] for k in range(1, a[i,j]+1))
           best_south = min(S[min(i+k, m-1), j] for k in range(1, a[i,j]+1))
           S[i,j] = 1 + min(best_east, best_south)

   return S[0,0]
```

Problemzerlegung

Nun kombinieren wir alle vier Teile und lösen somit das ursprüngliche Frog Jumps Problem.

1) Basisproblem
2) Variable Sprungweite
3) Zweidimensionalität
4) Bananenschale

Bananenschale

- Bananenschalen haben im Input Array den Wert -1.

Bananenschale

- Bananenschalen haben im Input Array den Wert -1.
- Landet der Frosch auf eine Bananenschale, so rutscht er aus und fällt auf das nächste Feld in süd-östlicher Richtung.

$$a[i,j] \rightarrow a[i+1,j+1]$$

Bananenschale

- Bananenschalen haben im Input Array den Wert -1.
- Landet der Frosch auf eine Bananenschale, so rutscht er aus und fällt auf das nächste Feld in süd-östlicher Richtung.

$$a[i,j] \rightarrow a[i+1,j+1]$$

- Das Ausrutschen zählt <u>nicht</u> als Sprung.

Bananenschale

- Bananenschalen haben im Input Array den Wert -1.
- Landet der Frosch auf eine Bananenschale, so rutscht er aus und fällt auf das nächste Feld in süd-östlicher Richtung.

$$a[i,j] \rightarrow a[i+1, j+1]$$

- Das Ausrutschen zählt <u>nicht</u> als Sprung.
- Bananenschalen Felder kommen nicht an den Rändern $i = n - 1$ oder $\mathrm{j} = \mathrm{m} - 1$ vor, da dort das diagonale Ausrutschen in einem ungültigen Speicherzugriff resultieren würde.

Frog Jumps: Beispiel

4	2	-1	2	5	3	6	1
6	2	-1	3	1	2	-1	10
4	2	2	1	-1	4	3	2
-1	6	4	1	3	-1	2	1
-1	-1	-1	2	4	2	1	5
2	4	3	2	1	2	2	

Schritt 1: Teilprobleme & Optionen identifizieren

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

Schritt 1: Teilprobleme & Optionen identifizieren

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

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme: unverändert, $S[i,j]$ bezeichnet die minimale Anzahl an Sprüngen, um von der Position $[i,j]$ das Ziel $[n-1,m-1]$ zu erreichen.
- Optionen: Neu hängen die Optionen davon ab, ob der Frosch auf einer Bananenschale landet:
  - Bananenschale: Wenn der Frosch auf einer Bananenschale landet, hat er nur eine Möglichkeit: Er rutscht aus und landet auf $a[i+1,j+1]$. Dieser Schritt zählt <u>nicht</u> als Sprung.

Schritt 1: Teilprobleme & Optionen identifizieren

- Teilprobleme: unverändert, $S[i,j]$ bezeichnet die minimale Anzahl an Sprüngen, um von der Position $[i,j]$ das Ziel $[n-1, m-1]$ zu erreichen.
- Optionen: Neu hängen die Optionen davon ab, ob der Frosch auf einer Bananenschale landet:
  - Bananenschale: Wenn der Frosch auf einer Bananenschale landet, hat er nur eine Möglichkeit: Er rutscht aus und landet auf $a[i+1, j+1]$. Dieser Schritt zählt <u>nicht</u> als Sprung.
  - Freie Felder: Optionen bleiben unverändert.

Schritt 2: Rekurrenz definieren

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

Mit

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

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

- Abhängigkeit identifizieren:
  - Um $S[i, j]$ zu berechnen, müssen folgende Teilprobleme gelöst sein:

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

- Abhängigkeit identifizieren:
  - Um $S[i,j]$ zu berechnen, müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i,j+1], S[i,j+2], \ldots, S\big[i,j+a[i]\big]$

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

- Abhängigkeit identifizieren:
  - Um $S[i,j]$ zu berechnen, müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i,j+1], S[i,j+2], \dots, S\big[i,j+a[i]\big]$
    - Im Süden: $S[i+1,j], S[i+2,j], \dots, S[i+a[i],j]$

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

- Abhängigkeit identifizieren:
  - Um $S[i,j]$ zu berechnen, müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i,j+1], S[i,j+2], \ldots, S\big[i,j+a[i]\big]$
    - Im Süden: $S[i+1,j], S[i+2,j], \ldots, S[i+a[i],j]$
    - Diagonal: $S[i+1,j+1]$

Schritt 3: Lösung implementieren

- Geeignete Datenstruktur definieren:
  - Unverändert, ein 2D-Array mit der gleichen Grösse wie das Input-Array.

- Abhängigkeit identifizieren:
  - Um $S[i,j]$ zu berechnen, müssen folgende Teilprobleme gelöst sein:
    - Im Osten: $S[i,j+1], S[i,j+2], \ldots, S\big[i,j+a[i]\big]$
    - Im Süden: $S[i+1,j], S[i+2,j], \ldots, S[i+a[i],j]$
    - Diagonal: $S[i+1,j+1]$

- Richtung des Ausfüllens:
  - Unverändert, Beginne bei $S[n-1,m-1]$ und fülle das Array von rechts nach links und unten nach oben aus.

Schritt 3: Lösung implementieren

```
import numpy as np

def frog_jumps(a):
   n,m = a.shape
   S = np.zeros((n,m))
   S[n-1,m-1] = 0 #Basisfall (redundant durch np.zeros)

   for i in range(n-1, -1, -1):
        for j in range(m-1, -1, -1):
             if i == n-1 and j == m-1: continue
             if a[i,j] == -1:
                  S[i,j] = S[i+1,j+1]
             else:
                  best_east = min(S[i, min(j+k, m-1)] for k in range(1,a[i,j]+1))
                  best_south = min(S[min(i+k, m-1), j] for k in range(1,a[i,j]+1))
                  S[i,j] = 1 + min(best_east, best_south)

   return S[0,0]
```

Problemzerlegung

Somit sind alle vier Teilprobleme kombiniert und das anfängliche Frog Jumps Problem gelöst.

1) Basisproblem
2) Variable Sprungweite
3) Zweidimensionalität
4) Bananenschale

4. Wrap-Up

Problemzerlegung: Elemente der Teilprobleme

Das Basisproblem kann beliebig mit den Teilproblemen ergänzt werden, jedes führt ein zusätzliches Element in der DP-Lösung ein:

Problemzerlegung: Elemente der Teilprobleme

Das Basisproblem kann beliebig mit den Teilproblemen ergänzt werden, jedes führt ein zusätzliches Element in der DP-Lösung ein:

- Variable Sprungweite: Eine **zusätzliche for-Schleife** ist erforderlich, da nun eine variable Anzahl an Sprungoptionen vorhanden ist.

Problemzerlegung: Elemente der Teilprobleme

Das Basisproblem kann beliebig mit den Teilproblemen ergänzt werden, jedes führt ein zusätzliches Element in der DP-Lösung ein:

- Variable Sprungweite: Eine **zusätzliche for-Schleife** ist erforderlich, da nun eine variable Anzahl an Sprungoptionen vorhanden ist.
- Zweidimensionalität: Das Problem wird mit einer **2D-Datenstruktur** erweitert, bei der zwei Laufvariablen i und j verwendet werden.

Problemzerlegung: Elemente der Teilprobleme

Das Basisproblem kann beliebig mit den Teilproblemen ergänzt werden, jedes führt ein zusätzliches Element in der DP-Lösung ein:

- Variable Sprungweite: Eine **zusätzliche for-Schleife** ist erforderlich, da nun eine variable Anzahl an Sprungoptionen vorhanden ist.
- Zweidimensionalität: Das Problem wird mit einer **2D-Datenstruktur** erweitert, bei der zwei Laufvariablen i und j verwendet werden.
- Bananenschale Felder: Eine **zusätzliche if-Bedingung** ist notwendig, da je nach Feld unterschiedliche Optionen zur Verfügung stehen.

5. Hausaufgaben

Übung 9: Dynamic Programming II

Auf `https://expert.ethz.ch/enrolled/SS25/mavt2/exercises`

Übung 9: DP II

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

Abgabedatum: Montag 05.05.2025, 20:00 MEZ

KEINE HARDCODIERUNG