

Übungslektion 9 – Dynamic Programming I

Informatik II

15. / 16. April 2025

Willkommen!

Polybox



Passwort: jschul

Personal Website



<https://n.ethz.ch/~jschul>

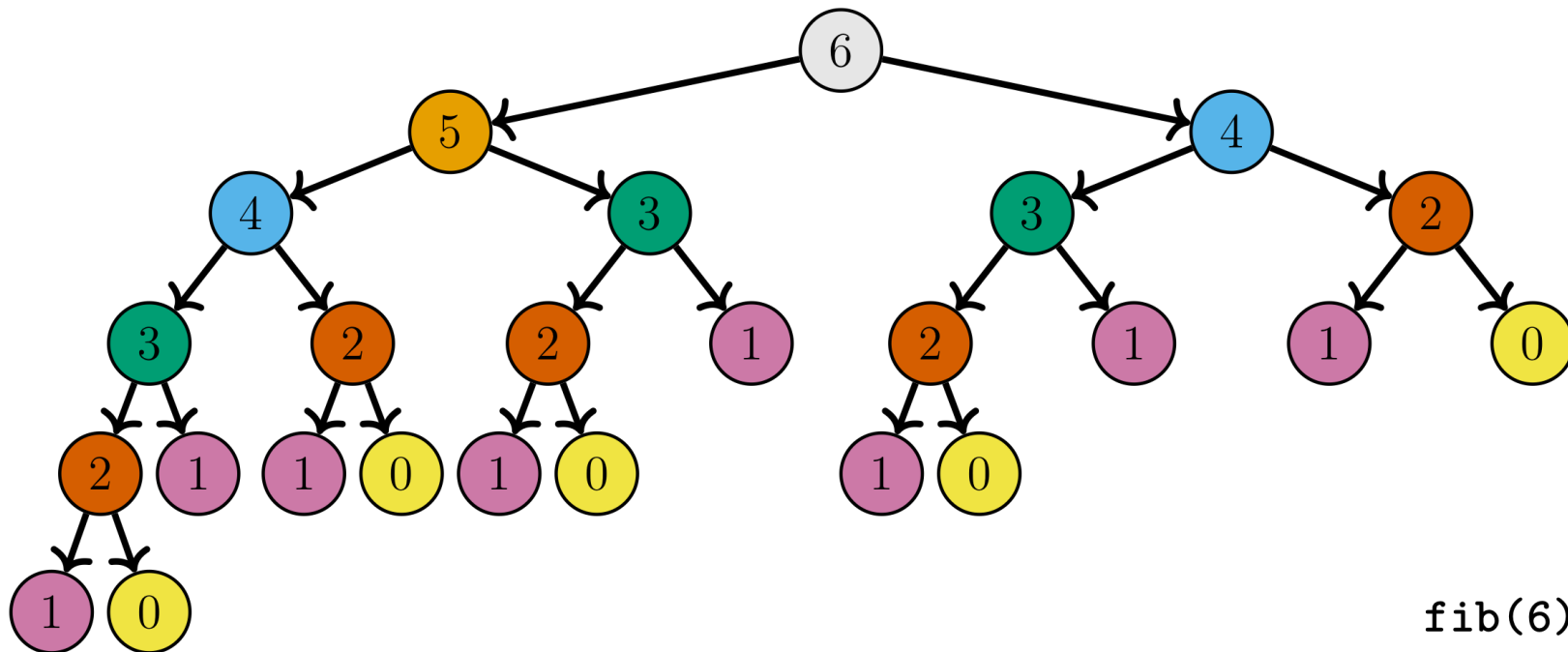
Heutiges Programm

- Recap
- King's Way
- Rod Cutting
- Wrap-Up

1. Recap

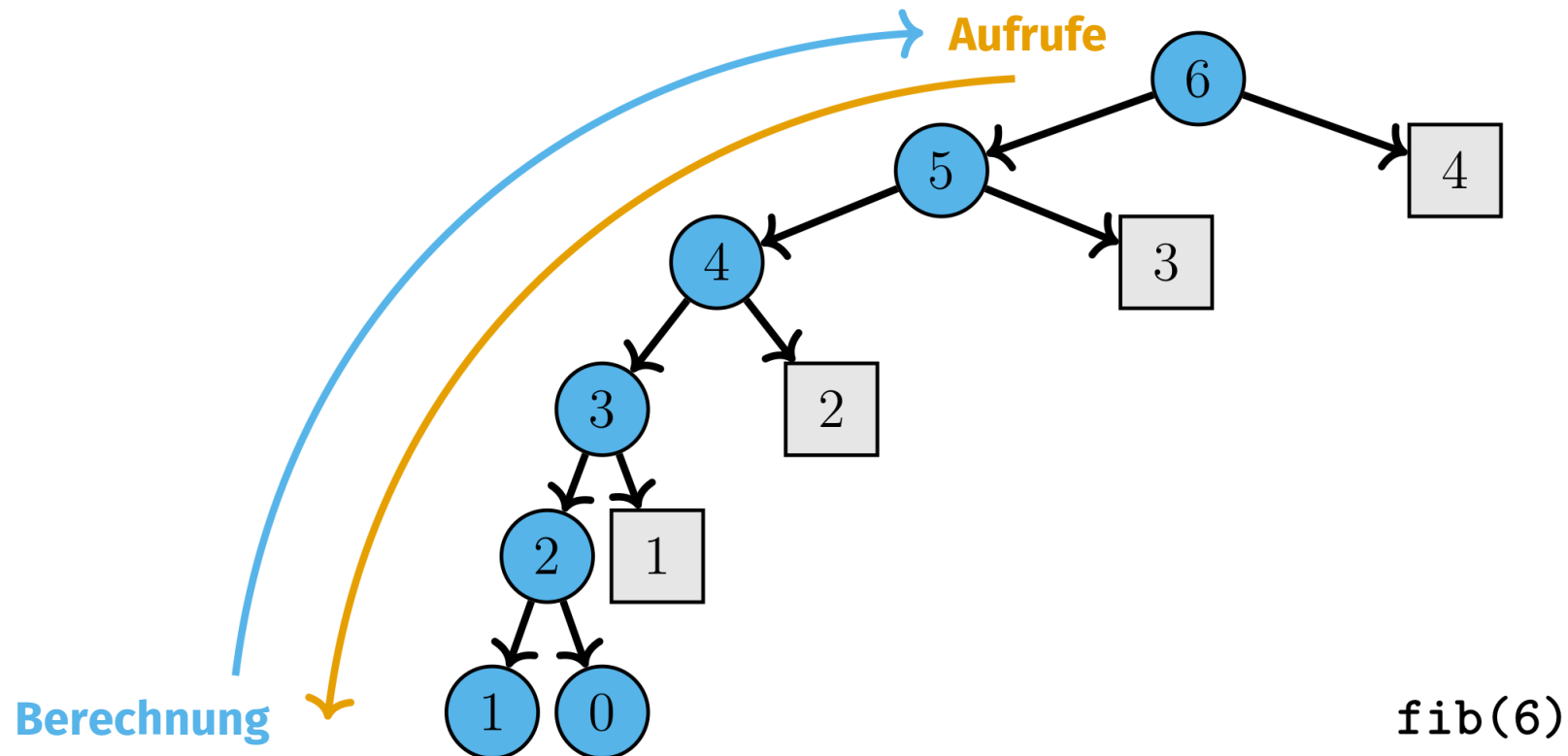
Repetition der Vorlesung

Rekursive Probleme wie die Fibonacci-Folge können mit Memoisierung oder Dynamic Programming schneller gelöst werden, weil unnötige Wiederholungen vermieden werden.



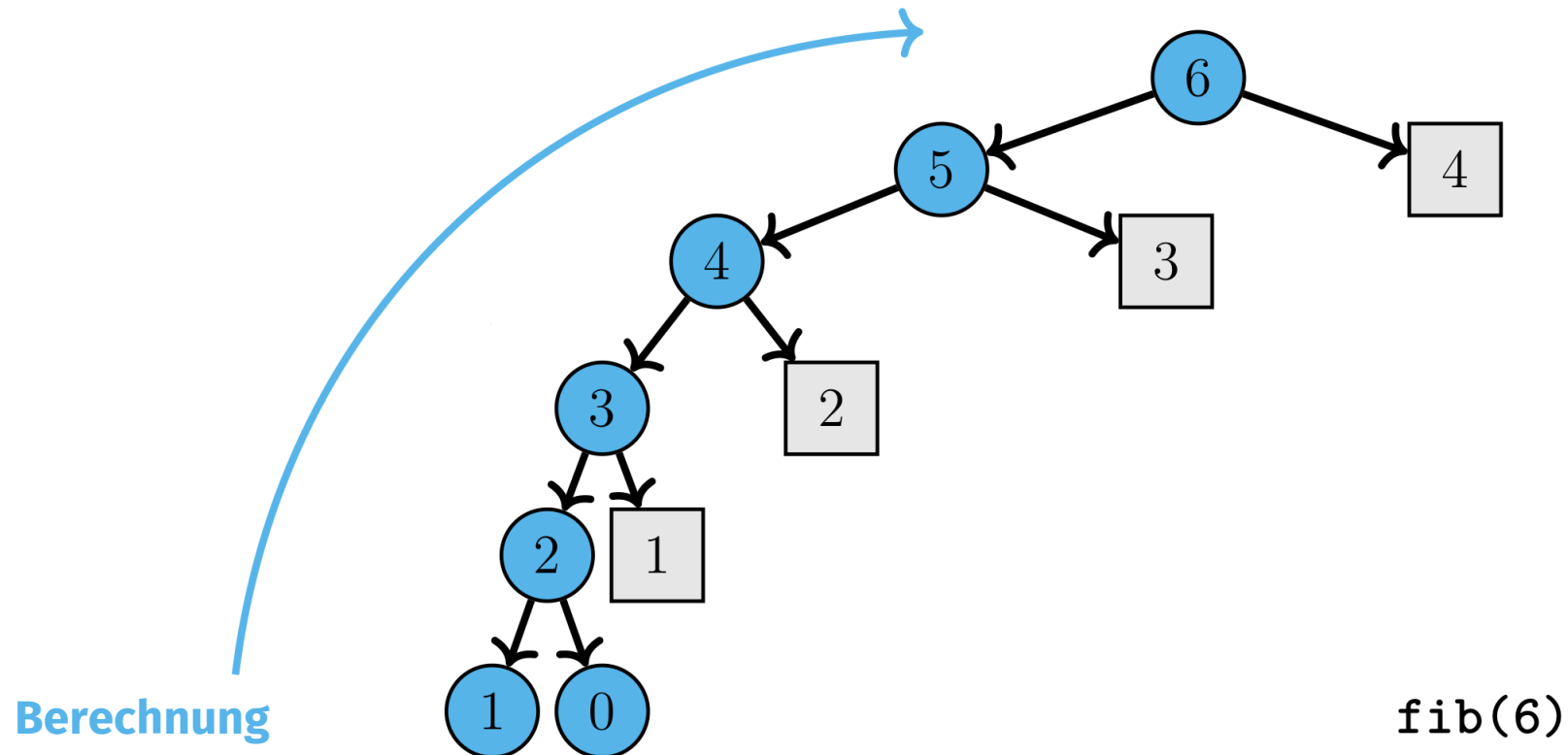
Memoisierung

Bei der Memoisierung werden Lösungen von Teilproblemen in einer Tabelle gespeichert. So können frühere Ergebnisse wiederverwendet werden, statt sie bei jedem rekursiven Aufruf neu zu berechnen.



Dynamic Programming - Tabellierung

Bei der Bottom-Up Dynamic Programmierung werden Teilprobleme ab dem Basisfall gelöst und Ergebnisse gespeichert, um doppelte Berechnungen zu vermeiden.



Memoisierung != Dynamic Programming

- Memoisierung

- Ansatz?

- Löst nur die notwendigen Teilprobleme.

- Dynamic Programming

Memoisierung != Dynamic Programming

- Memoisierung

- Ansatz?

- Löst nur die notwendigen Teilprobleme.

- Dynamic Programming

- Ansatz?

- Löst alle Teilprobleme im Voraus, auch wenn einige davon am Ende nicht benötigt werden.

Memoisierung != Dynamic Programming

■ Memoisierung

■ Ansatz?

Löst nur die notwendigen Teilprobleme.

■ Rekursion?

Wird mit Rekursion implementiert, dabei wird eine Tabelle bei jedem Aufruf mitgegeben.

■ Dynamic Programming

■ Ansatz?

Löst alle Teilprobleme im Voraus, auch wenn einige davon am Ende nicht benötigt werden.

Memoisierung != Dynamic Programming

■ Memoisierung

■ Ansatz?

Löst nur die notwendigen Teilprobleme.

■ Rekursion?

Wird mit Rekursion implementiert, dabei wird eine Tabelle bei jedem Aufruf mitgegeben.

■ Dynamic Programming

■ Ansatz?

Löst alle Teilprobleme im Voraus, auch wenn einige davon am Ende nicht benötigt werden.

■ Rekursion?

Wird typischerweise iterativ implementiert.

Memoisierung != Dynamic Programming

■ Memoisierung

- Ansatz?
Löst nur die notwendigen Teilprobleme.
- Rekursion?
Wird mit Rekursion implementiert, dabei wird eine Tabelle bei jedem Aufruf mitgegeben.
- Berechnungsrichtung?
Top-down.

■ Dynamic Programming

- Ansatz?
Löst alle Teilprobleme im Voraus, auch wenn einige davon am Ende nicht benötigt werden.
- Rekursion?
Wird typischerweise iterativ implementiert.

Memoisierung != Dynamic Programming

■ Memoisierung

- Ansatz?
Löst nur die notwendigen Teilprobleme.
- Rekursion?
Wird mit Rekursion implementiert, dabei wird eine Tabelle bei jedem Aufruf mitgegeben.
- Berechnungsrichtung?
Top-down.

■ Dynamic Programming


- Ansatz?
Löst alle Teilprobleme im Voraus, auch wenn einige davon am Ende nicht benötigt werden.
- Rekursion?
Wird typischerweise iterativ implementiert.
- Berechnungsrichtung?
Bottom-Up.

2. King's Way

Problemstellung: King's Way

- Gegeben: Ein Schachbrett a , auf dem der König auf dem Feld $[0,0]$ startet. Jedes Feld enthält eine nicht-negative Belohnung (0 oder mehr).

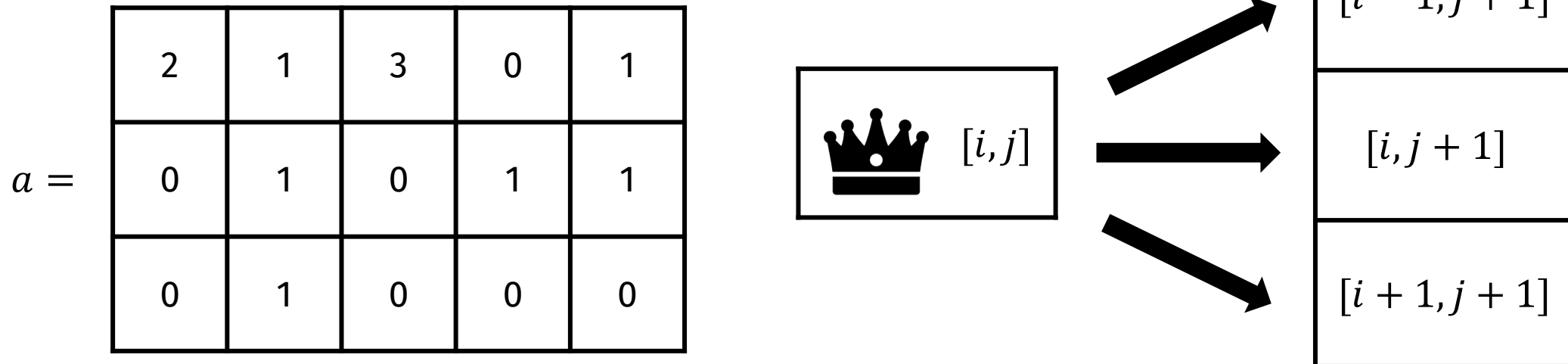
$a =$



2	1	3	0	1
0	1	0	1	1
0	1	0	0	0

Problemstellung: King's Way

- Gegeben: Ein Schachbrett a , auf dem der König auf dem Feld $[0,0]$ startet. Jedes Feld enthält eine nicht-negative Belohnung (0 oder mehr).
- An jeder Position $[i,j]$ hat der König drei mögliche Züge, solange er das Schachbrett nicht verlässt.



Problemstellung: King's Way

- **Input:** Ein Schachbrett, dargestellt als 2D-Array der Grösse $n \times m$, wobei jedes Feld eine nicht-negative Ganzzahl (≥ 0) enthält, die die jeweilige Belohnung angibt.

Problemstellung: King's Way

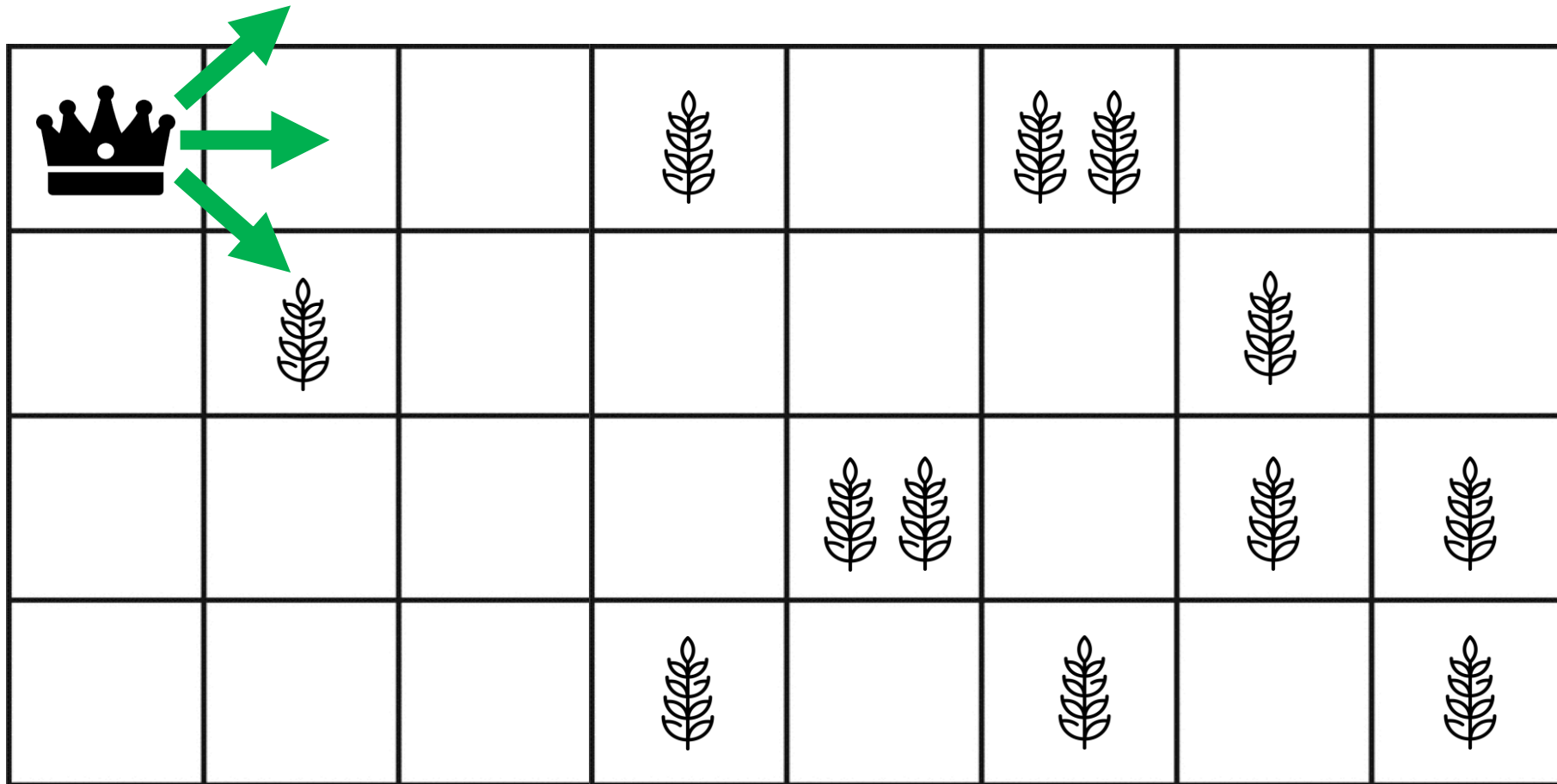
- **Input:** Ein Schachbrett, dargestellt als 2D-Array der Grösse $n \times m$, wobei jedes Feld eine nicht-negative Ganzzahl (≥ 0) enthält, die die jeweilige Belohnung angibt.
- **Ziel:** Der König startet auf Feld $[0,0]$ und kann sich in drei Richtungen bewegen: Nordost, Ost und Südost. Ziel ist es, den optimalen Pfad zu finden, um die Summe der Belohnungen auf dem Weg zur rechten Seite des Schachbretts zu maximieren.

Problemstellung: King's Way

- **Input:** Ein Schachbrett, dargestellt als 2D-Array der Grösse $n \times m$, wobei jedes Feld eine nicht-negative Ganzzahl (> 0) enthält, die die jeweilige Belohnung angibt.
- **Ziel:** Der König startet auf Feld $[0,0]$ und kann sich in drei Richtungen bewegen: Nordost, Ost und Südost. Ziel ist es, den optimalen Pfad zu finden, um die Summe der Belohnungen auf dem Weg zur rechten Seite des Schachbretts zu maximieren.
- **Output:** Die maximale Belohnung, die der König auf seinem Weg zur rechten Seite des Schachbretts einsammeln kann.

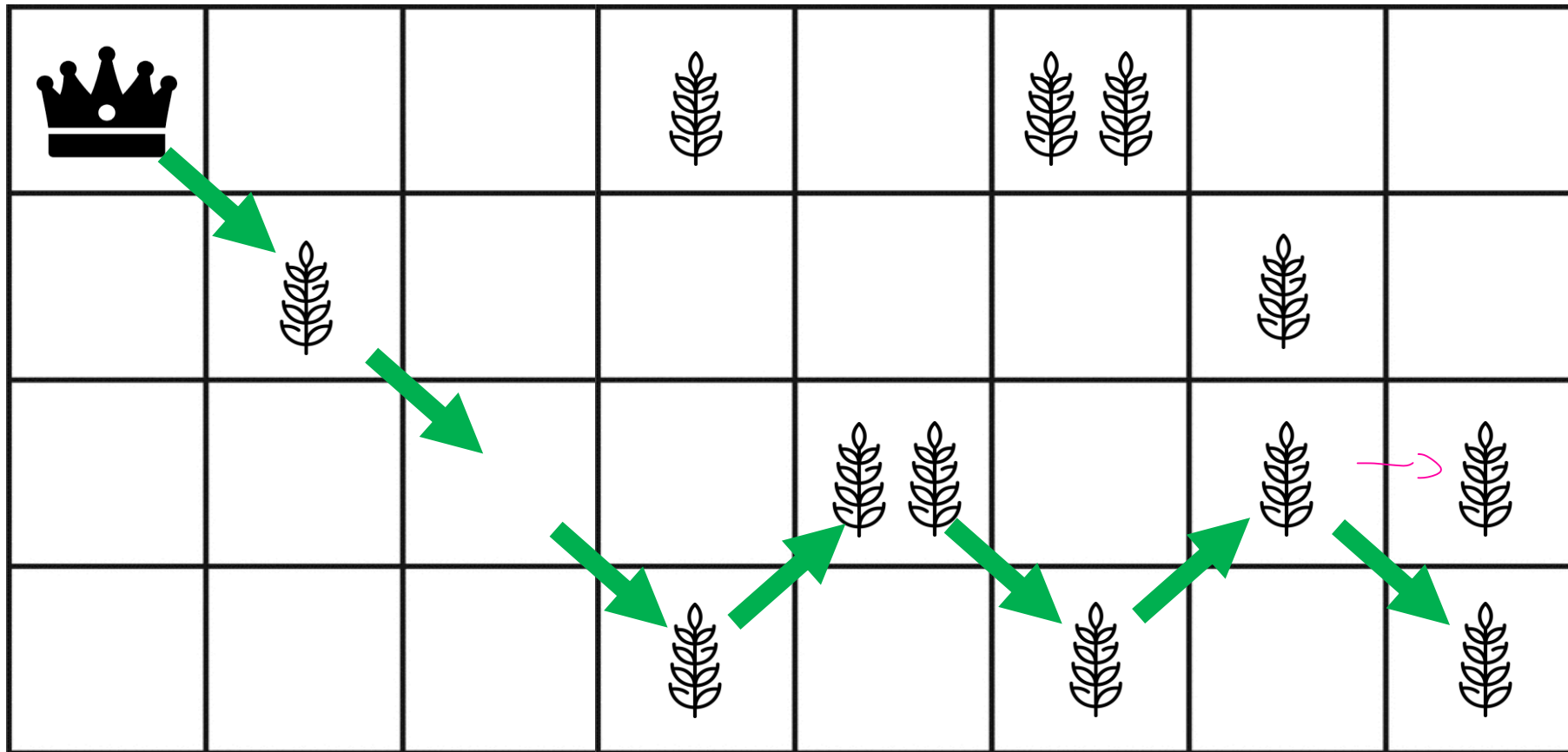
Problemstellung: King's Way Beispiel

Was ist die maximale Belohnung, die der König einsammeln kann?

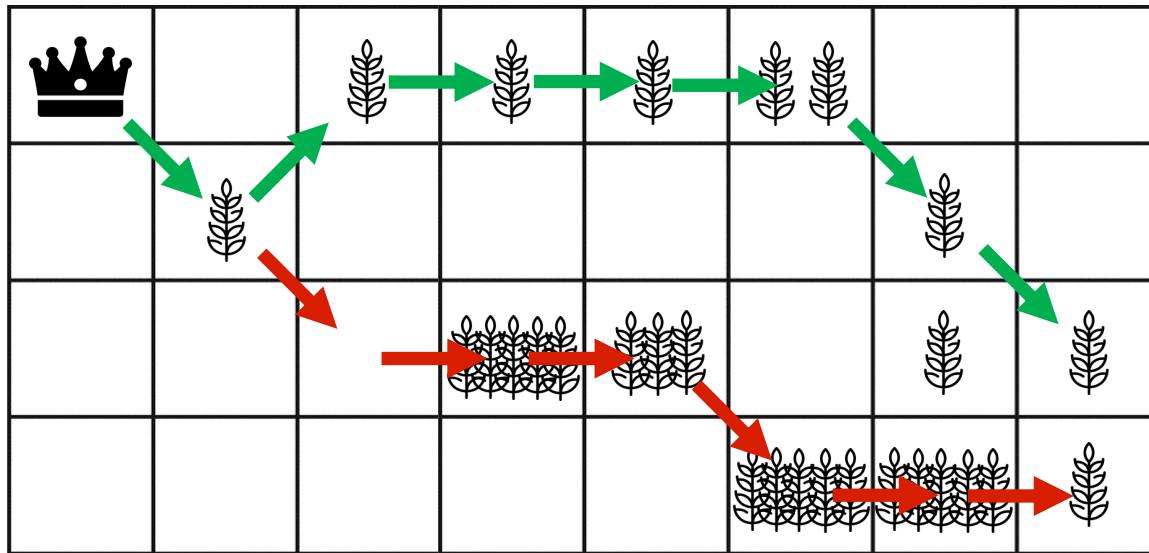


Problemstellung: King's Way Beispiel

Der König kann maximal 7 Belohnungen einsammeln.

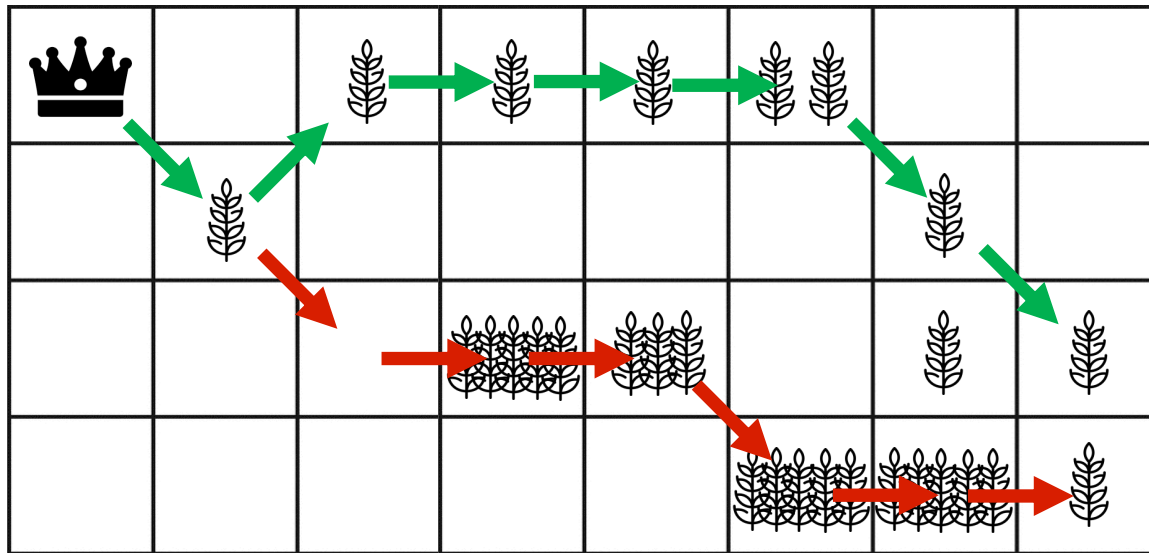


Gierige Lösung



- Nur die sofortige Belohnung zu nehmen, kann zu suboptimalen Entscheidungen führen, da dadurch grössere Belohnungen weiter vorne verpasst werden.

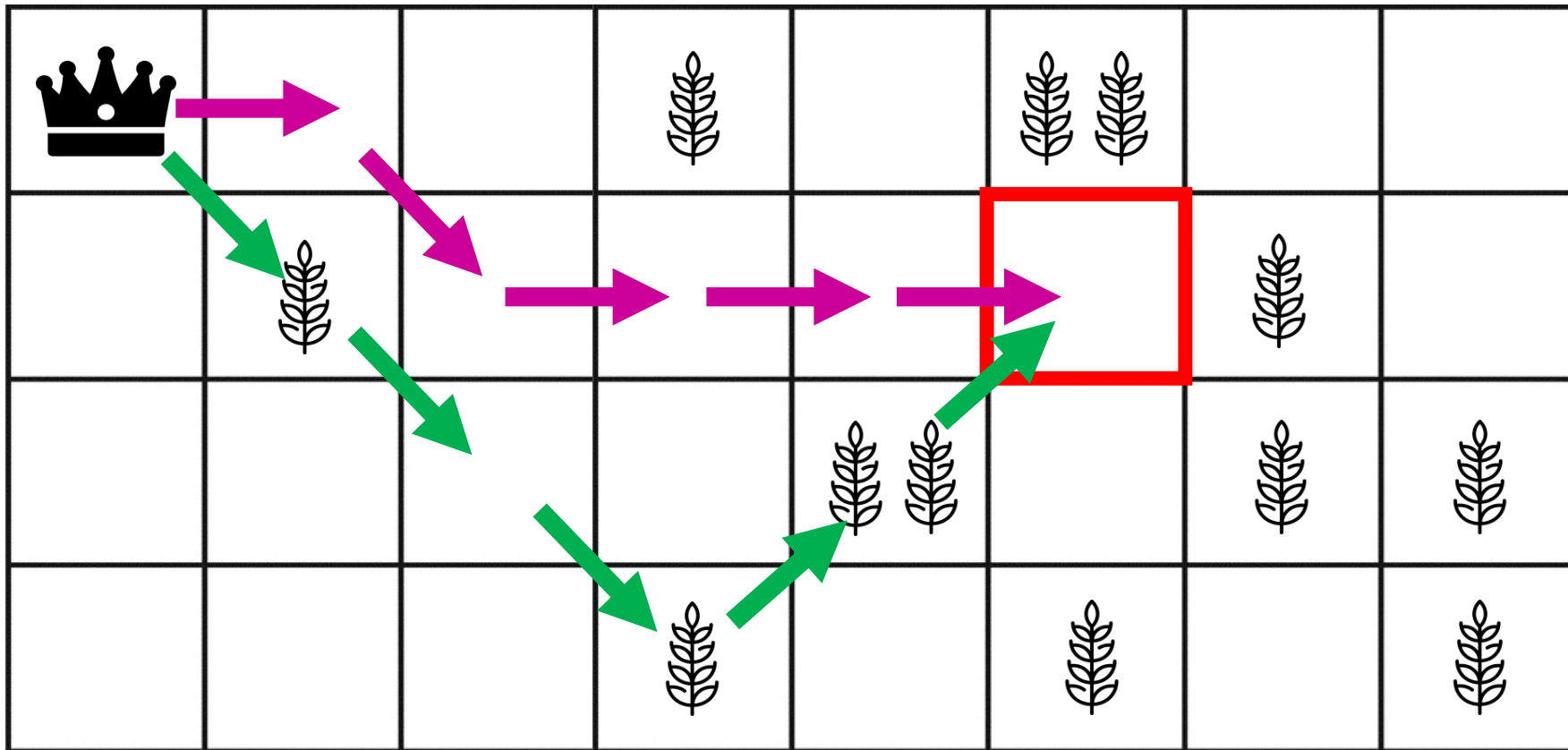
Gierige Lösung



- Nur die sofortige Belohnung zu nehmen, kann zu suboptimalen Entscheidungen führen, da dadurch grössere Belohnungen weiter vorne verpasst werden.
- Daher müssen wir alle möglichen Pfade bewerten, um den global optimalen Pfad zu finden.

Überlappende Teilprobleme

Da mehrere Pfade zum selben Feld führen können, wird das Teilproblem mehrfach gelöst, was zu überflüssigen Berechnungen führt.



Wieso ist DP so nützlich?

- **Brute Force:** Das Vergleichen aller Pfade hat eine exponentielle Laufzeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.

Wieso ist DP so nützlich?

- **Brute Force:** Das Vergleichen aller Pfade hat eine exponentielle Laufzeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamic Programming** zerlegt das Problem in kleinere Teilprobleme und speichert deren Lösungen zur Wiederverwendung.

Wieso ist DP so nützlich?

- **Brute Force:** Das Vergleichen aller Pfade hat eine exponentielle Laufzeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamic Programming** zerlegt das Problem in kleinere Teilprobleme und speichert deren Lösungen zur Wiederverwendung.
- **Zeitkomplexität:** $\mathcal{O}(n \times m)$ da die Anzahl der Felder mit dem Aufwand pro Feld $\mathcal{O}(1)$ multipliziert wird.

Wieso ist DP so nützlich?

- **Brute Force:** Das Vergleichen aller Pfade hat eine exponentielle Laufzeitkomplexität von $\mathcal{O}(3^{n \times m})$, was ineffizient ist.
- **Dynamic Programming** zerlegt das Problem in kleinere Teilprobleme und speichert deren Lösungen zur Wiederverwendung.
- **Zeitkomplexität:** $\mathcal{O}(n \times m)$ da die Anzahl der Felder mit dem Aufwand pro Feld $\mathcal{O}(1)$ multipliziert wird.
- **Effizienz:** Dynamic Programming vermeidet redundante Berechnungen und ist dadurch deutlich effizienter.

Die drei Schritte von DP

- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.

Die drei Schritte von DP

- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.
- 2) **Definiere die Rekurrenz**, indem du die Schritte zwischen Teilproblemen beschreibst und die besten Entscheidungen berücksichtigst.

Die drei Schritte von DP

- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.
- 2) **Definiere die Rekurrenz**, indem du die Schritte zwischen Teilproblemen beschreibst und die besten Entscheidungen berücksichtigst.
- 3) **Implementiere die Lösung** mit einer geeigneten Datenstruktur und der richtigen Reihenfolge des Ausfüllens.

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Für jedes Feld $[i, j]$ auf dem Schachbrett ($i < n, j < m$) steht $S[i, j]$ für die maximale Belohnung, die der König einsammeln kann, wenn er von diesem Feld bis zur letzten Spalte $j = m - 1$ zieht, unter Berücksichtigung aller möglichen Züge.

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Für jedes Feld $[i, j]$ auf dem Schachbrett ($i < n, j < m$) steht $S[i, j]$ für die maximale Belohnung, die der König einsammeln kann, wenn er von diesem Feld bis zur letzten Spalte $j = m - 1$ zieht, unter Berücksichtigung aller möglichen Züge.

- Welche Optionen hat der König an der Position $[i, j]$?

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Für jedes Feld $[i, j]$ auf dem Schachbrett ($i < n, j < m$) steht $S[i, j]$ für die maximale Belohnung, die der König einsammeln kann, wenn er von diesem Feld bis zur letzten Spalte $j = m - 1$ zieht, unter Berücksichtigung aller möglichen Züge.

- Welche Optionen hat der König an der Position $[i, j]$?
 - Zug nach Nordosten $[i - 1, j + 1]$
 - Zug nach Osten $[i, j + 1]$
 - Zug nach Südosten $[i + 1, j + 1]$

Randbedingungen

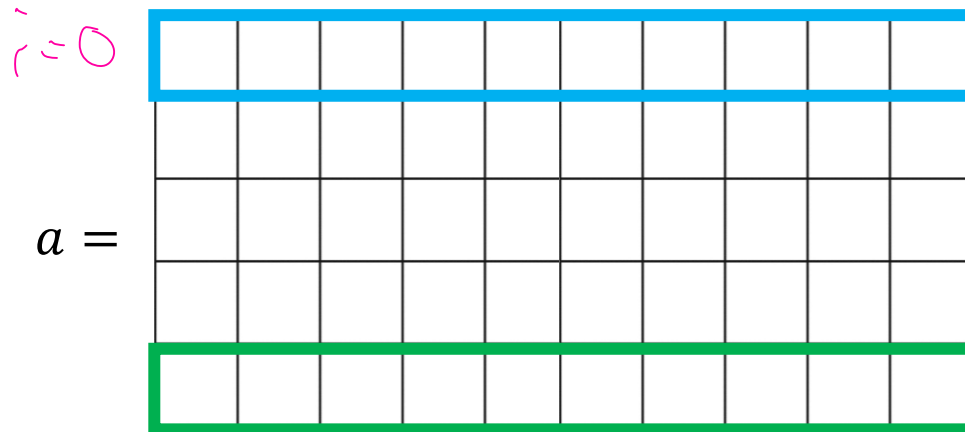
Kann der König immer nach Nordost oder Südost ziehen?

Randbedingungen

Kann der König immer nach Nordost oder Südost ziehen?

Nein, die Randbedingungen müssen beachtet werden!

- Oberste Zeile ($i = 0$): Der König kann nicht nach Nordost ziehen.
- Unterste Zeile ($i = n - 1$): Der König kann nicht nach Südost ziehen.



Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Für jedes Feld $[i, j]$ auf dem Schachbrett ($i < n, j < m$) steht $S[i, j]$ für die maximale Belohnung, die der König einsammeln kann, wenn er von diesem Feld bis zur letzten Spalte $j = m - 1$ zieht, unter Berücksichtigung aller möglichen Züge.

- Welche Optionen hat der König an der Position $[i, j]$?
 - Zug nach Nordosten $[i - 1, j + 1]$
 - Zug nach Osten $[i, j + 1]$
 - Zug nach Südosten $[i + 1, j + 1]$

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Für jedes Feld $[i, j]$ auf dem Schachbrett ($i < n, j < m$) steht $S[i, j]$ für die maximale Belohnung, die der König einsammeln kann, wenn er von diesem Feld bis zur letzten Spalte $j = m - 1$ zieht, unter Berücksichtigung aller möglichen Züge.

- Welche Optionen hat der König an der Position $[i, j]$?
 - Zug nach Nordosten $[i - 1, j + 1]$, **if $i > 0$**
 - Zug nach Osten $[i, j + 1]$
 - Zug nach Südosten $[i + 1, j + 1]$, **if $i > 0$**

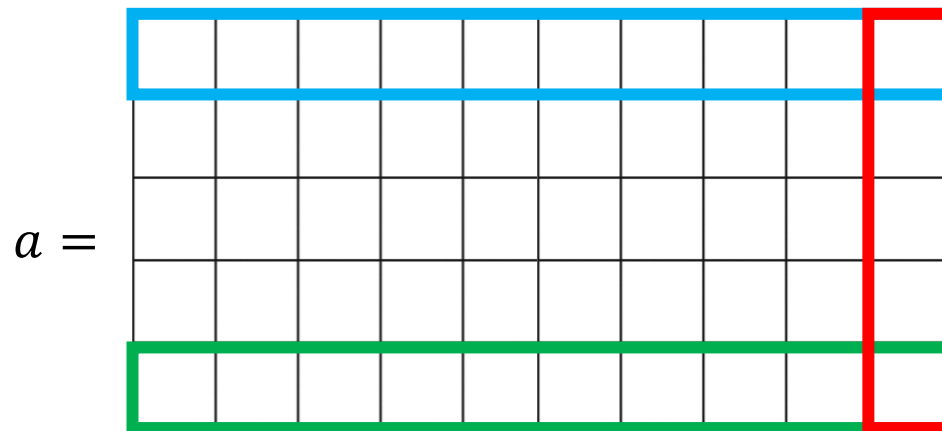
Basisfall

Gibt es noch weitere Einschränkungen für den König auf dem Brett?

Basisfall

Gibt es noch weitere Einschränkungen für den König auf dem Brett?

Ja, eine weitere wichtige Einschränkung ist, dass der König anhalten muss, wenn er die rechteste Spalte erreicht ($j = m - 1$), da keine weiteren Züge mehr möglich sind. Dies ist der **Basisfall**: Der König kann sich nicht weiterbewegen, daher gilt $S[i, j] = a[i, j]$.



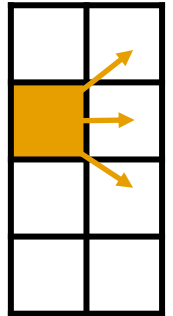
Schritt 1: Identifiziere die Teilprobleme

- Eine Spalte → Der König erhält einfach die Belohnung des jeweiligen Feldes (Basisfall).



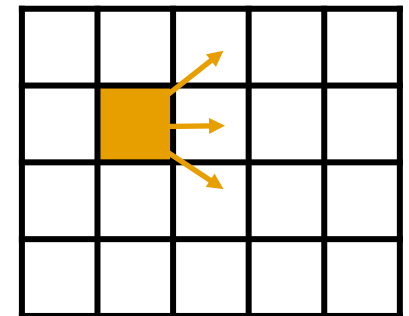
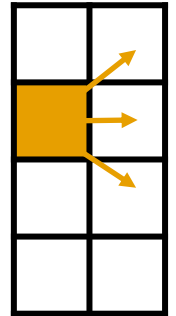
Schritt 1: Identifiziere die Teilprobleme

- Eine Spalte → Der König erhält einfach die Belohnung des jeweiligen Feldes (Basisfall).
- Zwei Spalten → Der König hat drei mögliche Züge: Nordosten, Osten und Südosten, und wählt den Zug mit der höchsten Belohnung.



Schritt 1: Identifiziere die Teilprobleme

- Eine Spalte → Der König erhält einfach die Belohnung des jeweiligen Feldes (Basisfall).
- Zwei Spalten → Der König hat drei mögliche Züge: Nordosten, Osten und Südosten, und wählt den Zug mit der höchsten Belohnung.
- Vollständiges Schachbrett → Die Entscheidung des Königs an jedem Feld hängt von den Werten in der nächsten Spalte ab – er wählt die Option mit dem grössten möglichen Gewinn.



Schritt 2: Definiere die rekursive Formel

$S[i, j]$ stellt die maximale Belohnung dar, die der König sammeln kann, wenn er beim Feld $[i, j]$ startet und optimal bis zur rechten Spalte zieht.

$$S[i, j] = \begin{cases} a[i, j] & j = m - 1 \\ a[i, j] + \max(S[i - 1, j + 1], S[i, j + 1]) & i = n - 1, j < m - 1 \\ a[i, j] + \max(S[i, j + 1], S[i + 1, j + 1]) & i = 0, j < m - 1 \\ a[i, j] + \max \begin{cases} S[i - 1, j + 1] \\ S[i, j + 1] \\ S[i + 1, j + 1] \end{cases} & 0 < i < n - 1, 0 < j < m - 1 \end{cases}$$

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Eine **Matrix** S der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < n$ und $j < m$.

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Eine **Matrix** S der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < n$ und $j < m$.

- Von welchen Teilproblemen hängt die Berechnung von $S[i, j]$ ab?

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Eine **Matrix** S der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < n$ und $j < m$.

- Von welchen Teilproblemen hängt die Berechnung von $S[i, j]$ ab?

Von $S[i, j + 1]$, $S[i, j - 1]$ wenn $i > 0$ und $S[i + 1, j]$ wenn $i < n - 1$
– also unter Berücksichtigung der Randfälle.

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Eine **Matrix** S der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < n$ und $j < m$.

- Von welchen Teilproblemen hängt die Berechnung von $S[i, j]$ ab?

Von $S[i, j + 1]$, $S[i, j - 1]$ wenn $i > 0$ und $S[i + 1, j]$ wenn $i < n - 1$ – also unter Berücksichtigung der Randfälle.

- In welche Richtung sollte die Datenstruktur ausgefüllt werden?

Schritt 3: Lösung implementieren

- Welche Datenstruktur wird für das King's Way Problem benötigt?

Eine **Matrix** S der Grösse $n \times m$, wobei $S[i, j]$ die maximale Belohnung speichert, die eingesammelt werden kann, wenn man bei $[i, j]$ startet, für alle $i < n$ und $j < m$.

- Von welchen Teilproblemen hängt die Berechnung von $S[i, j]$ ab?

Von $S[i, j + 1]$, $S[i, j - 1]$ wenn $i > 0$ und $S[i + 1, j]$ wenn $i < n - 1$ – also unter Berücksichtigung der Randfälle.

- In welche Richtung sollte die Datenstruktur ausgefüllt werden?

Von rechts nach links, Spalte für Spalte. Die Reihenfolge der Zeilen innerhalb einer Spalte ist dabei nicht wichtig.

DP Tabelle

Während dem Ausfüllen der Datenstruktur müssen Randbedingungen sorgfältig berücksichtigt werden.

DP Tabelle

Während dem Ausfüllen der Datenstruktur müssen Randbedingungen sorgfältig berücksichtigt werden.

- **Grösse der DP-Tabelle:** Wir verwenden eine DP-Tabelle in derselben Grösse wie das Schachbrett, um die optimale Lösung für jedes Feld zu speichern.

DP Tabelle

Während dem Ausfüllen der Datenstruktur müssen Randbedingungen sorgfältig berücksichtigt werden.

- **Grösse der DP-Tabelle:** Wir verwenden eine DP-Tabelle in derselben Grösse wie das Schachbrett, um die optimale Lösung für jedes Feld zu speichern.
- **Richtung des Durchlaufs:** Die Tabelle muss so durchlaufen werden, dass alle abhängigen Teilprobleme bereits gelöst sind, bevor das aktuelle Feld berechnet wird.

DP Tabelle

Randbedingungen:

- Bei $j = m - 1$ (letzte Spalte) kann der König nicht mehr weiter. Dies ist der Basisfall.

DP Tabelle

Randbedingungen:

- Bei $j = m - 1$ (letzte Spalte) kann der König nicht mehr weiter. Dies ist der Basisfall.
- Wenn $i = 0$ (oberste Zeile) kann der König nicht nach Nordost ziehen.

DP Tabelle

Randbedingungen:

- Bei $j = m - 1$ (letzte Spalte) kann der König nicht mehr weiter. Dies ist der Basisfall.
- Wenn $i = 0$ (oberste Zeile) kann der König nicht nach Nordost ziehen.
- Wenn $i = n - 1$ (unterste Zeile), kann der König nicht nach Südost ziehen.

DP Tabelle

Randbedingungen:

- Bei $j = m - 1$ (letzte Spalte) kann der König nicht mehr weiter. Dies ist der Basisfall.
- Wenn $i = 0$ (oberste Zeile) kann der König nicht nach Nordost ziehen.
- Wenn $i = n - 1$ (unterste Zeile), kann der König nicht nach Südost ziehen.
- An diesen Randpositionen sind nicht alle drei Zugoptionen (Ost, Nordost, Südost) gültig – es dürfen nur die zulässigen Bewegungen berücksichtigt werden.

Daten Struktur

Durch die rekursive Formulierung haben wir die Abhängigkeiten definiert, die für die Berechnung der Lösung des Königswegs an der Position $[i, j]$ erforderlich sind.

$$S[i, j] = \begin{cases} a[i, j] & j = m - 1 \\ a[i, j] + \max(S[i - 1, j + 1], S[i, j + 1]) & i = n - 1, j < m \\ a[i, j] + \max(S[i, j + 1], S[i + 1, j + 1]) & i = 0, j < m \\ a[i, j] + \max \begin{cases} S[i - 1, j + 1] \\ S[i, j + 1] \\ S[i + 1, j + 1] \end{cases} & 0 < i < n - 1, j < m \end{cases}$$

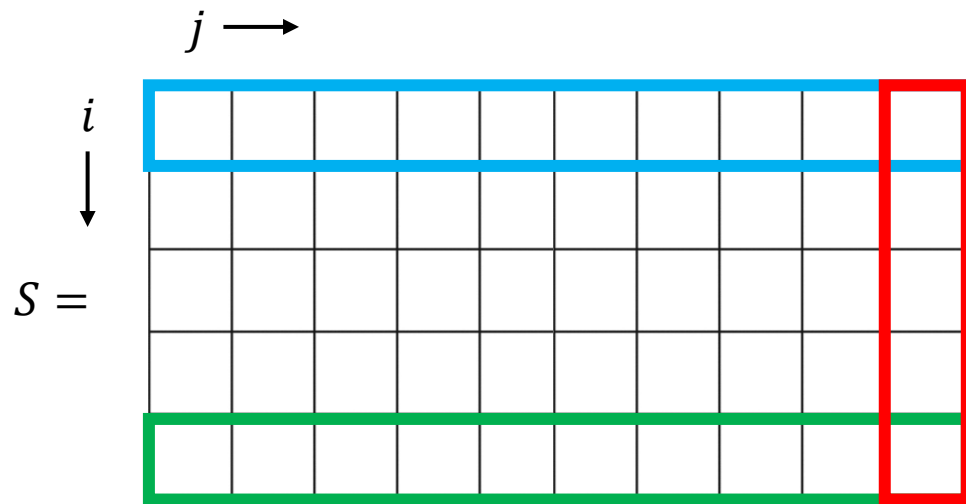
Die Rekursive Lösung in der Tabelle

- Sei $S[i, j]$ die maximale Belohnung, die der König ab diesem Feld aus erhalten kann.

Die Rekursive Lösung in der Tabelle

- Sei $S[i, j]$ die maximale Belohnung, die der König ab diesem Feld aus erhalten kann.
- Sobald die gesamte Tabelle berechnet wurde, kann der König auf jedes beliebige Feld gesetzt werden, und es ist sofort ersichtlich, welche maximale Belohnung von diesem Startpunkt aus erreicht werden kann.

DP Tabelle



Randbedingungen

- $j = m - 1$ $a[i, j]$
- $i = 0, j < m$ $a[i, j] + \max(\rightarrow, \searrow)$
- $i = n - 1, j < m$ $a[i, j] + \max(\rightarrow, \nearrow)$

sonst

- $a[i, j] + \max(\nearrow, \rightarrow, \searrow)$

DP Tabelle

Die endgültige Lösung ist $s[0,0]$, links oben in der DP-Tabelle.

$S =$

Beispiel: Wie füllt man die DP-Tabelle aus?

↙



















→ X

👑		🌿			🌿		
		🌿🌿		🌿		🌿	🌿
X	🌿🌿		🌿			🌿🌿	
				🌿			🌿🌿
		🌿					🌿

S =

8				4	3	1	0
10				5	4	2	1
10			4	4	4	4	0
10				5	4	2	2
6				4	2	2	1

Beispiel: Wie füllt man die DP-Tabelle aus?

$S =$

8	8	6	5	4	3	1	0
10	8	8	5	5	4	2	1
10	10	6	6	4	4	4	0
10	6	6	5	5	4	2	2
6	6	6	5	4	2	2	1

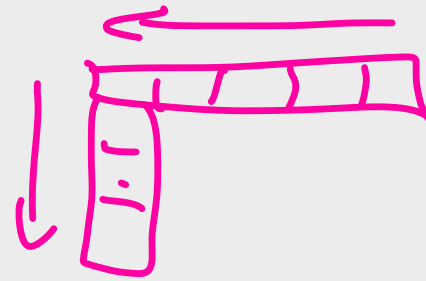
Python Implementierung: Maximum Belohnung

```
import numpy as np

def bestway(a):
    a = np.array(a) # Falls a noch kein numpy array ist
    n,m = a.shape
    S = np.zeros((n,m))

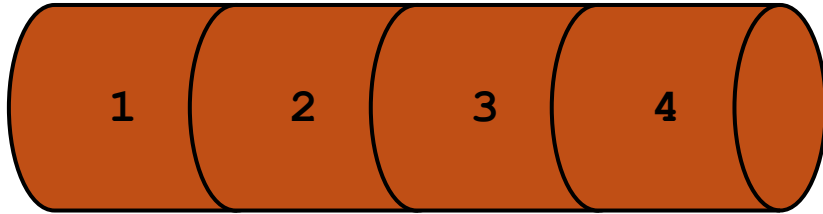
    for j in range(m-1, -1, -1):
        for i in range(n):
            if j == m-1:
                S[i,j] = a[i,j]
            else:
                northeast = S[i-1,j+1] if i > 0 else -1
                east = S[i,j+1]
                southeast = S[i+1,j+1] if i < n - 1 else -1
                S[i,j] = a[i,j] + max(northeast, east, southeast)

    return S[0,0]
```



3. Rod Cutting

Problemstellung

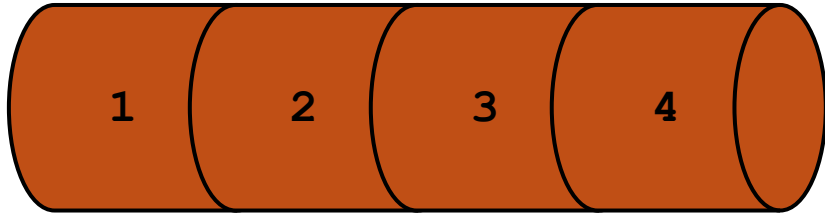


Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9

■ Input:

- Eine Stange der Länge n , die wir zerschneiden.

Problemstellung

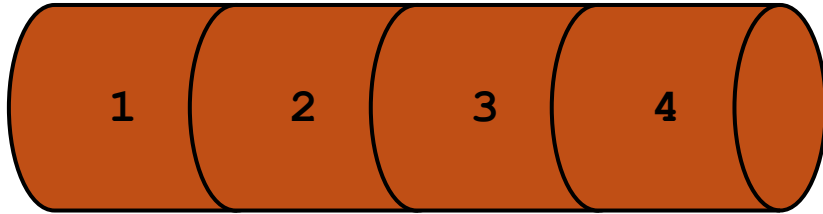


Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9

■ Input:

- Eine Stange der Länge n , die wir zerschneiden.
- Eine Preisliste p für Stangen von unterschiedlicher Länge.

Problemstellung



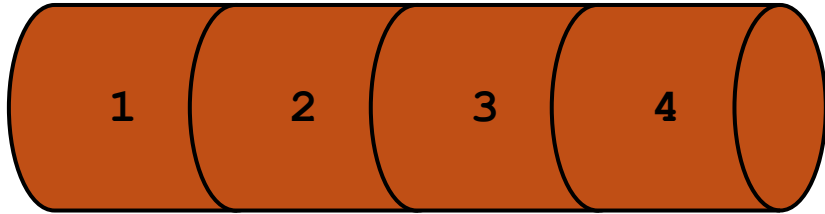
Länge i	0	1	2	3	4
Preis $p[i]$	0	1	5	8	9

■ Input:

- Eine Stange der Länge n , die wir zerschneiden.
- Eine Preisliste p für Stangen von unterschiedlicher Länge.

- ## ■ Ziel:
- Die Stange so zerschneiden, dass die Schnittstücke zusammen für den höchstmöglichen Preis verkauft werden können.

Problemstellung



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

■ Input:

- Eine Stange der Länge n , die wir zerschneiden.
- Eine Preisliste p für Stangen von unterschiedlicher Länge.

■ Ziel:

Die Stange so zerschneiden, dass die Schnittstücke zusammen für den höchstmöglichen Preis verkauft werden können.

■ Output:

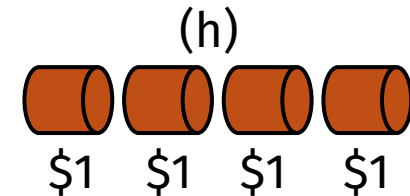
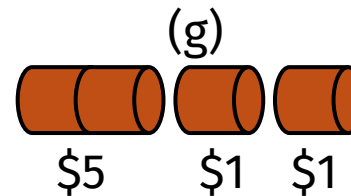
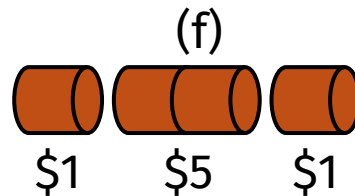
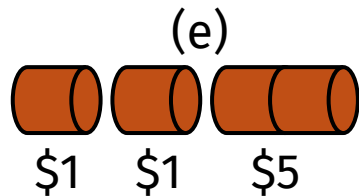
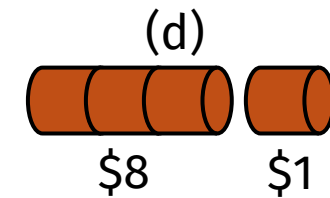
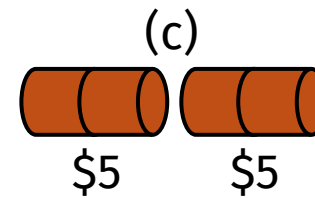
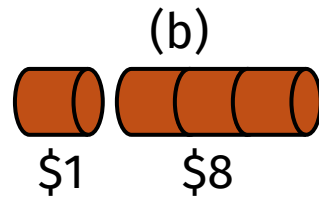
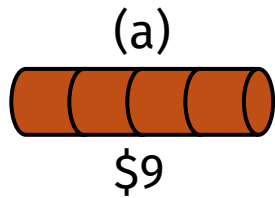
Der maximale Wert $S[n]$, der durch das Verschneiden der Stange erlangt werden kann.

Beispiel

Für eine Stange der Länge 4 gibt es 8 Möglichkeiten, sie zu schneiden:

$n = 7$

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



Wobei (c) mit $\$5 + \$5 = \$10$ den höchsten Ertrag liefert.

Problemstellung

- **Problem:** Für einen Stab der Länge n gibt es 2^{n-1} Optionen.

Problemstellung

- **Problem:** Für einen Stab der Länge n gibt es 2^{n-1} Optionen.
→ Um diese zu verstehen, fangen wir bei den kleinsten Längen an.

Die drei Schritte von DP

- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.

Die drei Schritte von DP


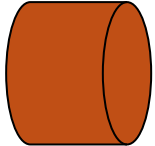
- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.
- 2) **Definiere die Rekurrenz**, indem du die Schritte zwischen Teilproblemen beschreibst und die besten Entscheidungen berücksichtigst.

Die drei Schritte von DP

- 1) **Identifiziere die Teilprobleme** und analysiere, wie sie sich überschneiden.
- 2) **Definiere die Rekurrenz**, indem du die Schritte zwischen Teilproblemen beschreibst und die besten Entscheidungen berücksichtigst.
- 3) **Implementiere die Lösung** mit einer geeigneten Datenstruktur und der richtigen Reihenfolge des Ausfüllens.

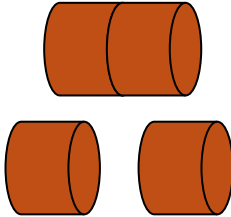
Optimale Substruktur

- Für Stangen der Länge 0 und 1 haben wir jeweils nur eine Option.

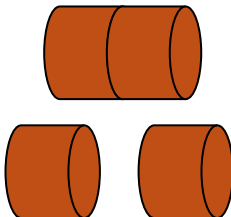
Länge	Optionen	Optimaler Preis
0		0
1		$p[1]$

Optimale Substruktur

- Für eine Stange der Länge 2 haben wir bereits mehrere Optionen.

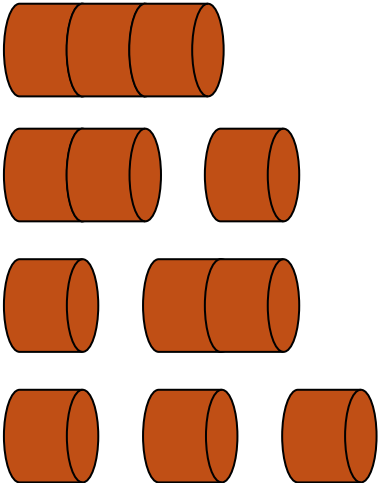
Länge	Optionen	Preis-Optionen
2		$p[2]$ $p[1] + p[1]$

- Die bessere Option von beiden bestimmt den optimalen Preis.

Länge	Optionen	Optimaler Preis
2		$\max \left\{ \begin{array}{l} p[2] \\ p[1] + p[1] \end{array} \right\} = s[2]$

Optimale Substruktur

- Der Preis für die 3er Stange hängt vom Preis der 2er Stange ab.
→ Folgend hängt der optimale Preis der 3er Stange vom optimalen Preis der 2er Stange ab.

Länge	Optionen	Optimaler Preis
3		$\max \left\{ \begin{array}{l} p[3] \\ S[2] + p[1] \\ p[1] + S[2] \\ p[1] + p[1] + p[1] \end{array} \right\} = S[3]$

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

Schritt 1: Identifiziere die Teilprobleme

- Welche Teilprobleme gibt es?

$S[i]$ bezeichnet den optimalen Preis, den man für eine Stange der Länge i erzielen kann.

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis einer Stange der Länge i setzt sich zusammen aus dem Preis des Schnittstücks der Länge k und dem Preis vom Reststück der Länge $i - k$.

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

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis einer Stange der Länge i setzt sich zusammen aus dem Preis des Schnittstücks der Länge k und dem Preis vom Reststück der Länge $i - k$.

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

- Da nicht bekannt ist, bei welcher Länge k der optimale Schnitt liegt, müssen alle Optionen geprüft und die Beste gewählt werden.

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis einer Stange der Länge i setzt sich zusammen aus dem Preis des Schnittstücks der Länge k und dem Preis vom Reststück der Länge $i - k$.

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

- Da nicht bekannt ist, bei welcher Länge k der optimale Schnitt liegt, müssen alle Optionen geprüft und die Beste gewählt werden.

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

Schritt 2: Definiere die rekursive Formel

- Der optimale Preis einer Stange der Länge i setzt sich zusammen aus dem Preis des Schnittstücks der Länge k und dem Preis vom Reststück der Länge $i - k$.

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

- Da nicht bekannt ist, bei welcher Länge k der optimale Schnitt liegt, müssen alle Optionen geprüft und die Beste gewählt werden.

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

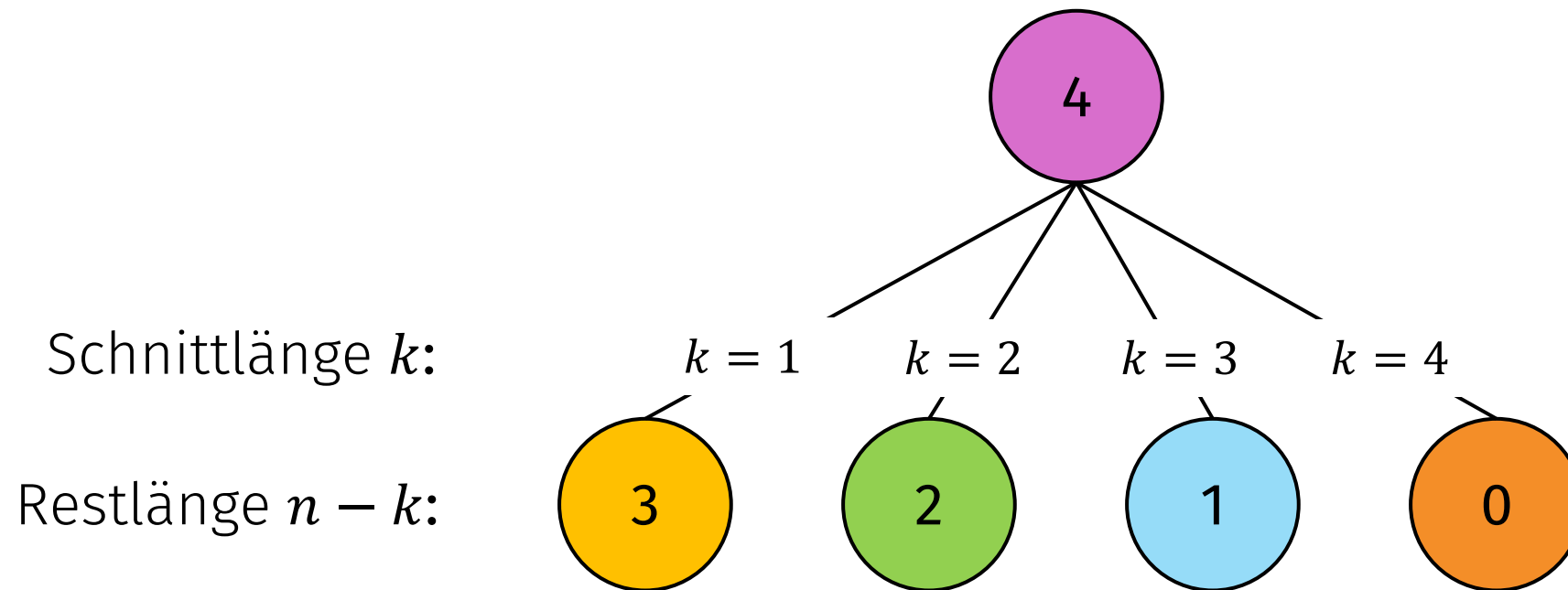
- Beachte, dass diese Formel nur gilt, wenn p für jede Länge von 0 bis i einen Preis angibt. Andernfalls wird $i - k$ irgendwann negativ, was zu einem ungültigen Zugriff führt.

Rekursiver Algorithmus

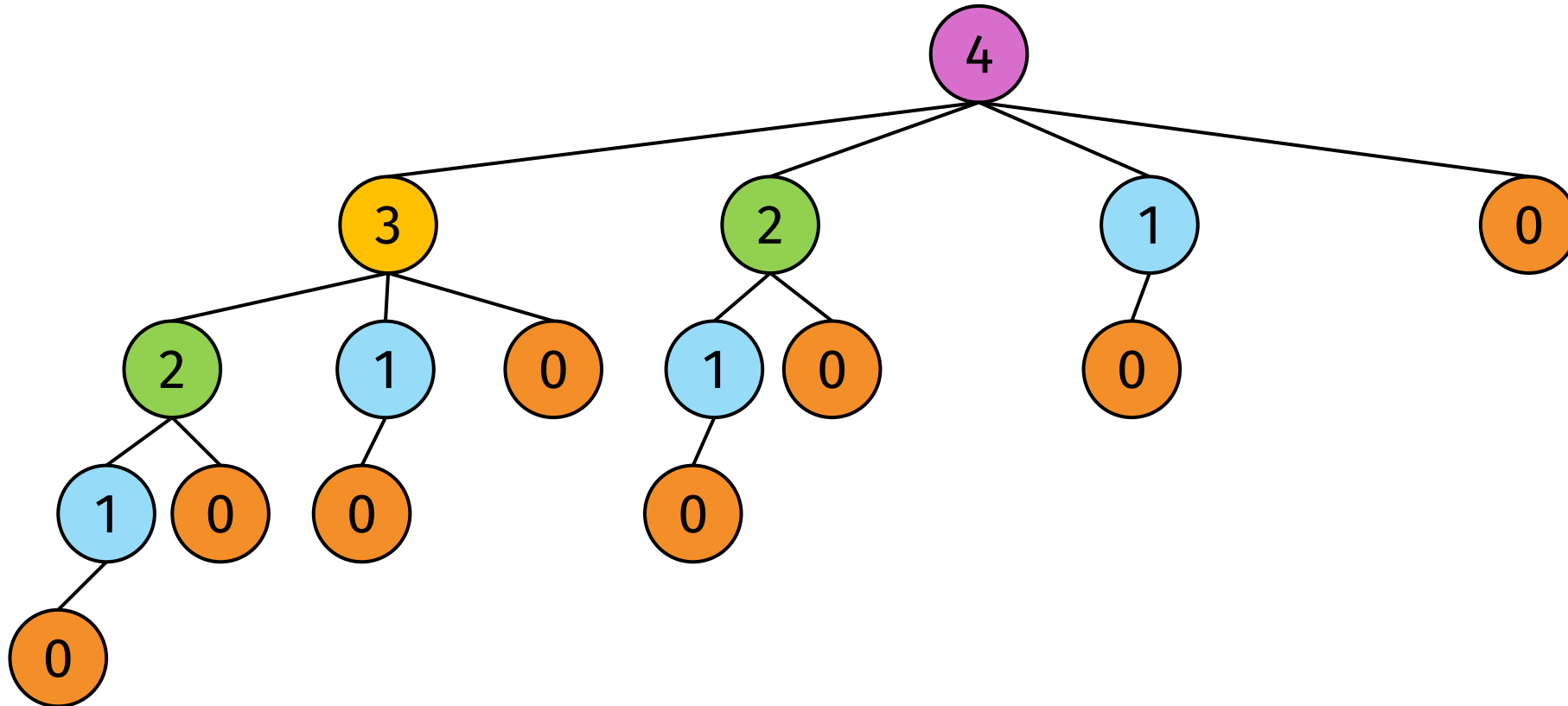
```
def max_value(p, n):  
  
    #Base case  
    if n == 0:  
        return 0  
  
    #Optimaler Preis  
    options = [p[k] + max_value(p, n-k) for k in range(1, n+1)]  
    return max(options)
```

Überlappende Teilprobleme

Anhand des Beispiels für eine Stange der Länge 4.



Überlappende Teilprobleme



- Die Teilprobleme werden mehrfach berechnet! Das resultiert in einer exponentiellen asymptotischen Laufzeit von $\mathcal{O}(2^{n-1})$.

Dynamic Programming

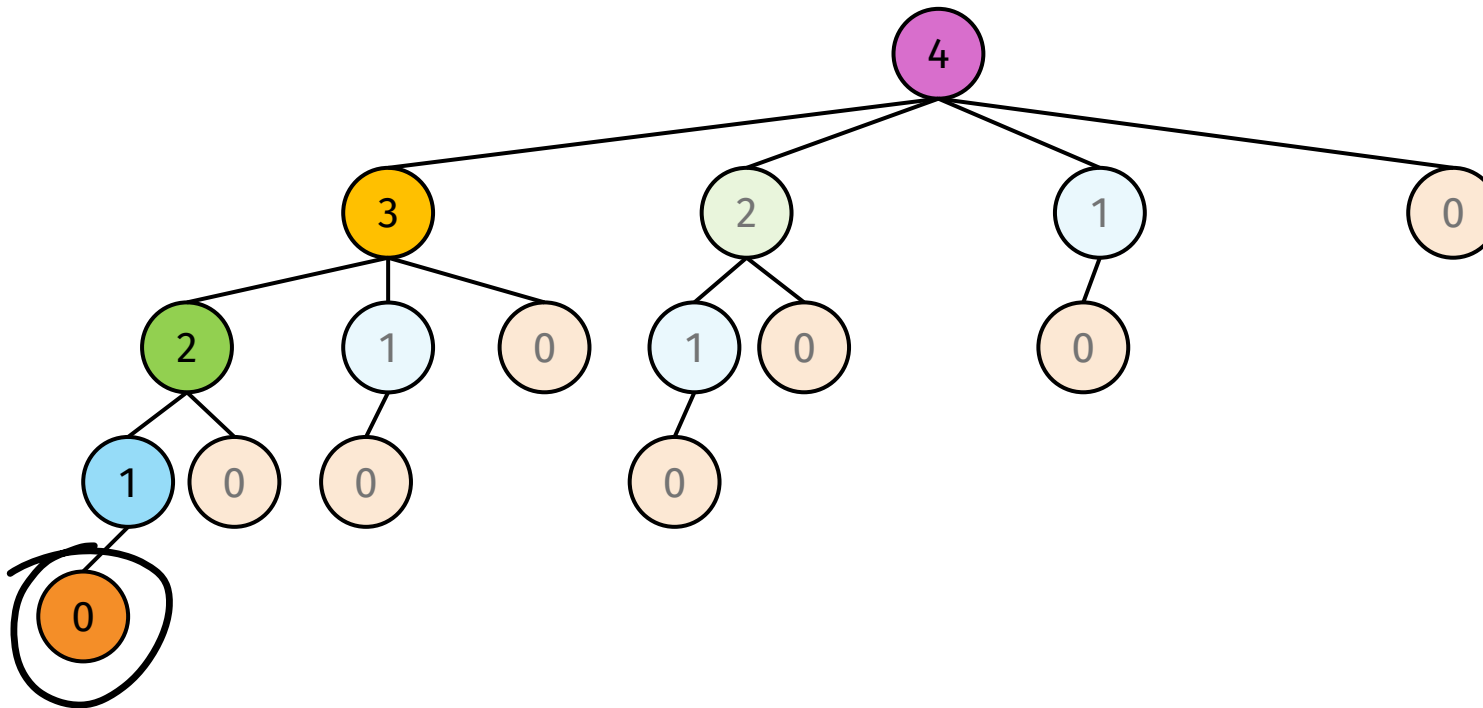
- Gegeben sind ein Rekursives Problem mit überlappenden Teilproblemen.

Dynamic Programming

- Gegeben sind ein Rekursives Problem mit überlappenden Teilproblemen.
- Somit kann Rod Cutting durch Dynamic Programming optimiert werden.

Memoisierung

- Wie bereits gezeigt, speichern wir die Subprobleme nach dem ersten Lösen. Ab da können wir direkt auf die Lösung zugreifen und vermeiden mehrfaches Berechnen des Subproblems.



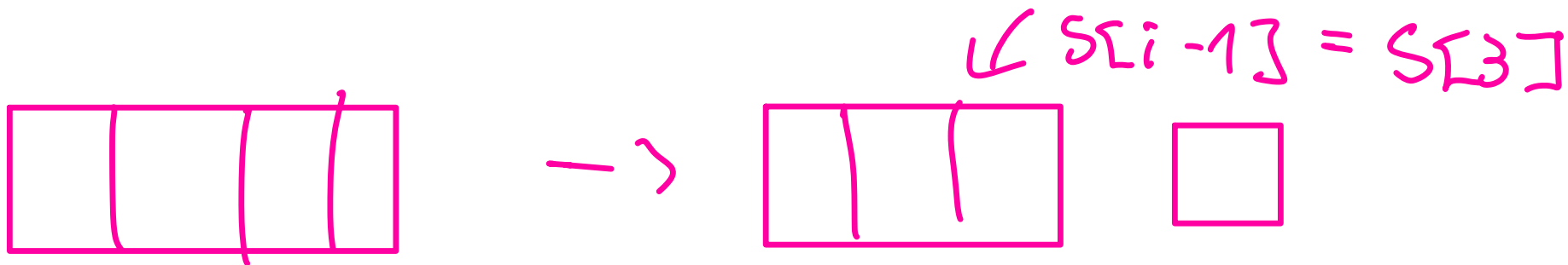
- Dies reduziert die asymptotische Laufzeit von $\mathcal{O}(2^{n-1})$ auf $\mathcal{O}(n^2)$.

Schritt 3: Lösung Implementieren

- Bestimme eine geeignete Datenstruktur:
 - Ein 1D-Array der Länge $n + 1$

Schritt 3: Lösung Implementieren

- Bestimme eine geeignete Datenstruktur:
 - Ein 1D-Array der Länge $n + 1$
- Ermittle die Abhängigkeiten:
 - Um $S[i]$ zu berechnen, müssen zuerst $S[i - 1], \dots, S[i - k]$ bekannt sein.



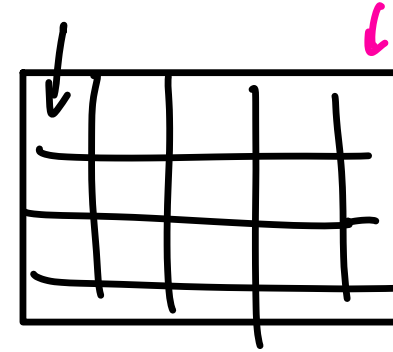
Schritt 3: Lösung Implementieren

- Bestimme eine geeignete Datenstruktur:
 - Ein 1D-Array der Länge $n + 1$
- Ermittle die Abhängigkeiten:
 - Um $S[i]$ zu berechnen, müssen zuerst $S[i - 1], \dots, S[i - k]$ bekannt sein.
 - Letztendlich hängt alles von $S[0]$ ab – dem Basisfall.

Schritt 3: Lösung Implementieren

- Bestimme eine geeignete Datenstruktur:

- Ein 1D-Array der Länge $n + 1$



- Ermittle die Abhängigkeiten:

- Um $S[i]$ zu berechnen, müssen zuerst $S[i - 1], \dots, S[i - k]$ bekannt sein.
- Letztendlich hängt alles von $S[0]$ ab – dem Basisfall.

- Bestimme die richtige Berechnungsreihenfolge:

- Aus den Abhängigkeiten ergibt sich, dass wir bei $i = 0$ starten und das Array von links nach rechts ausfüllen.



Zur Erinnerung: Rekursiver Algorithmus

```
def max_value(p, n):  
  
    #Base case  
    if n == 0:  
        return 0  
  
    #Optimaler Preis  
    options = [p[k] + max_value(p, n-k) for k in range(1, n+1)]  
    return max(options)
```

Memoisierung

→ In rot die Änderungen am naiv-rekursiven Algorithmus.

```
def max_value_memo(p, n, memo = None):  
    # Datenstruktur erstellen, falls nicht gegeben  
    if memo is None:  
        memo = dict()  
  
    # Subproblem nur berechnen, falls es noch nicht berechnet wurde  
    if n not in memo:  
  
        if n == 0:  
            memo[0] = 0  
        else:  
            options = [p[k] + max_value_memo(p, n-k, memo) for k in range(1, n+1)]  
            memo[n] = max(options) # Resultat speichern  
  
    return memo[n]
```

Handwritten annotations:

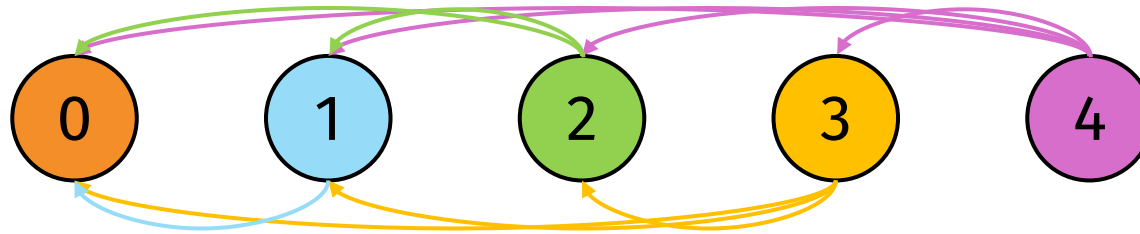
- $h=3$ (green)
- $h=2$ (black)
- Green checkmarks (✓) and a red 'X' (✗) with arrows pointing to the recursive call and the memoization line.
- A large black box on the right side of the code.

Memoisierung

```
def max_value_memo(p, n, memo = None):  
    # Datenstruktur erstellen, falls nicht gegeben  
    if memo is None:  
        memo = dict()  
  
    # Subproblem nur berechnen, falls es noch nicht berechnet wurde  
    if n not in memo:  
  
        if n == 0:  
            memo[0] = 0  
        else:  
            options = [p[k] + max_value_memo(p, n-k, memo) for k in range(1, n+1)]  
            memo[n] = max(options)  
  
    return memo[n]
```

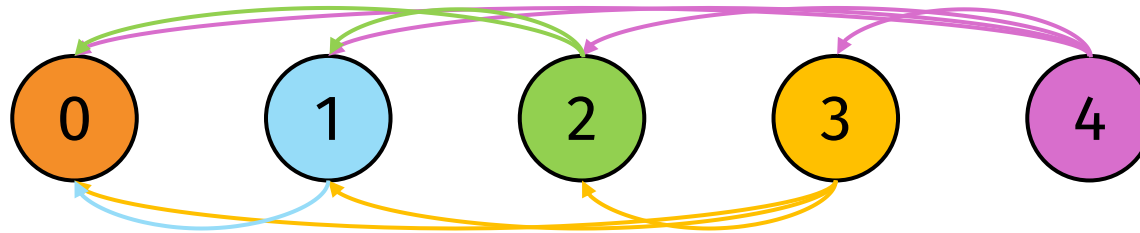
Dynamic Programming: Theorie

- Der optimale Preis vom 3er Stück $S[3]$ hängt vom optimalen Preis des 2er Stücks $S[2]$, des 1er Stücks $S[1]$, und des Basisfalls $S[0]$ ab.



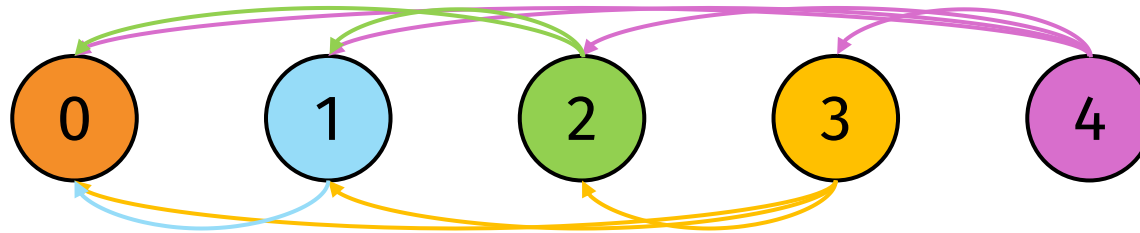
Dynamic Programming: Theorie

- Der optimale Preis vom 3er Stück $S[3]$ hängt vom optimalen Preis des 2er Stücks $S[2]$, des 1er Stücks $S[1]$, und des Basisfalls $S[0]$ ab.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.



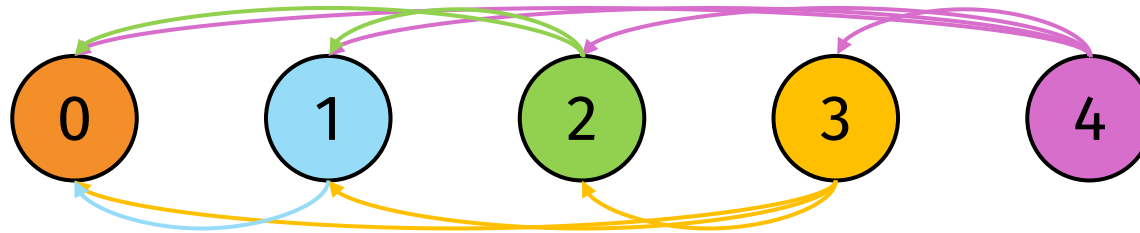
Dynamic Programming: Theorie

- Der optimale Preis vom 3er Stück $S[3]$ hängt vom optimalen Preis des 2er Stücks $S[2]$, des 1er Stücks $S[1]$, und des Basisfalls $S[0]$ ab.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



Dynamic Programming: Theorie

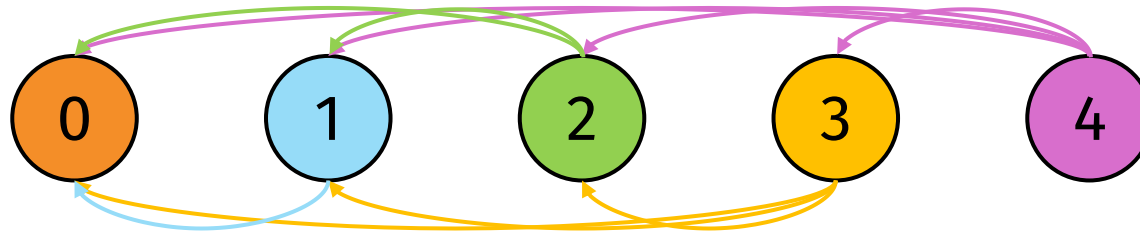
- Der optimale Preis vom 3er Stück $S[3]$ hängt vom optimalen Preis des 2er Stücks $S[2]$, des 1er Stücks $S[1]$, und des Basisfalls $S[0]$ ab.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



- Weil $S[0] = 0$ bereits bekannt ist, beginnen wir bei $S[1]$ und berechnen Schritt für Schritt die Resultate bis $S[n]$.

Dynamic Programming: Theorie

- Der optimale Preis vom 3er Stück $S[3]$ hängt vom optimalen Preis des 2er Stücks $S[2]$, des 1er Stücks $S[1]$, und des Basisfalls $S[0]$ ab.
- $S[2]$ hängt von $S[1]$ und $S[0]$ ab.
- $S[1]$ hängt nur von $S[0]$ ab.



- Weil $S[0] = 0$ bereits bekannt ist, beginnen wir bei $S[1]$ und berechnen Schritt für Schritt die Resultate bis $S[n]$.
- Dafür benutzen wir eine geeignete Datenstruktur, z.B. eine Liste der Länge $n+1$

Dynamic Programming

```
import numpy as np
def max_value_dp(p, n):
```

```
    #Datenstruktur erstellen
```

```
    S = np.zeros(n+1)
```

$S = [0, 0, 0, 0]$

```
    #Basisfall implementieren
```

```
    S[0] = 0 # Überflüssig aufgrund np.zeros
```

```
    #Datenstruktur in geeignete Richtung ausfüllen
```

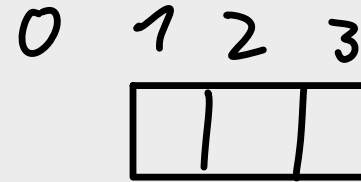
```
    for i in range(1, n+1):
```

$i=1$

```
        options = [p[k] + S[i-k] for k in range(1, i+1)]
```

```
        S[i] = max(options)
```

```
    return S[n]
```



$n=3$

$i=2$ options: $[p[2] + S[0]]$

options: $[p[1] + S[0], \dots]$

$S[1] = p[1]$

→ Auf die Richtung des Ausfüllens & Indizes (n oder $n + 1$) achten!

4. Wrap-Up

Wrap-Up: Dynamic Programming

Allgemeines Vorgehen bei Dynamic Programming:

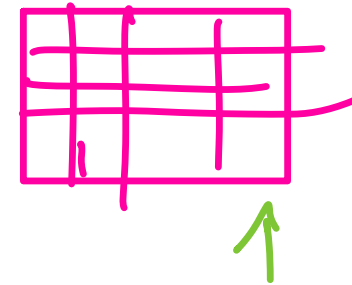
1. Datenstruktur initialisieren
2. Basisfall implementieren
3. Datenstruktur ausfüllen *R-L, L-R?*
4. Resultat zurückgeben

Wrap-Up: King's Way

1. Datenstruktur initialisieren:

- 2D Matrix

```
S = np.zeros( (n,m) )
```



Wrap-Up: King's Way

1. Datenstruktur initialisieren:

- 2D Matrix

```
S = np.zeros( (n,m) )
```

2. Basisfall implementieren:

- Rechte Spalte

```
S[:,m] = a[:,m]
```

Wrap-Up: King's Way

1. Datenstruktur initialisieren:

- 2D Matrix

```
S = np.zeros( (n,m) )
```

2. Basisfall implementieren:

- Rechte Spalte

```
S[:,m] = a[:,m]
```

3. Datenstruktur ausfüllen:

- Von rechts nach links

```
for j in [...]:  
    for i in [...]:  
        opt1 = [...]  
        opt2 = [...]  
        opt3 = [...]
```

Wrap-Up: King's Way

1. Datenstruktur initialisieren:

- 2D Matrix

```
S = np.zeros( (n,m) )
```

2. Basisfall implementieren:

- Rechte Spalte

```
S[:,m] = a[:,m]
```

3. Datenstruktur ausfüllen:

- Von rechts nach links

```
for j in [...]:  
    for i in [...]:  
        opt1 = [...]  
        opt2 = [...]  
        opt3 = [...]
```

4. Resultat zurückgeben

```
return S[0,0]
```

Wrap-Up: Rod Cutting

1. Datenstruktur initialisieren:
 - 1D Array

```
S = np.zeros(n+1)
```


Wrap-Up: Rod Cutting

1. Datenstruktur initialisieren:
 - 1D Array
2. Basisfall implementieren:
 - Stange der Länge 0

```
S = np.zeros(n+1)
```

```
S[0] = 0 # Überflüssig durch np.zeros
```

Wrap-Up: Rod Cutting

1. Datenstruktur initialisieren:
 - 1D Array
2. Basisfall implementieren:
 - Stange der Länge 0
3. Datenstruktur ausfüllen:
 - Variable Anzahl an Optionen

```
S = np.zeros(n+1)
```

```
S[0] = 0 # Überflüssig durch np.zeros
```

```
for i in [...]:  
    options = ["opt for k in cuts"]  
    data[i] = max(options)
```

S[i]

Wrap-Up: Rod Cutting

1. Datenstruktur initialisieren:
 - 1D Array
2. Basisfall implementieren:
 - Stange der Länge 0
3. Datenstruktur ausfüllen:
 - Variable Anzahl an Optionen
4. Resultat zurückgeben

```
S = np.zeros(n+1)
```

```
S[0] = 0 # Überflüssig durch np.zeros
```

```
for i in [...]:  
    options = ["opt for k in cuts"]  
    data[i] = max(options)
```



```
return S[n]
```

5. Hausaufgaben

Übung 8: Dynamic Programming I

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

Übung 8: DP I

- Tribonacci 
- Catalan Numbers 
- Blocks
- Task Scheduling v2.0

Abgabedatum: Montag 28.04.2025, 20:00 CET

KEINE HARDCODIERUNG