



Exercise Session 9 – Dynamic Programming I

Informatik II

15. / 16. April 2025

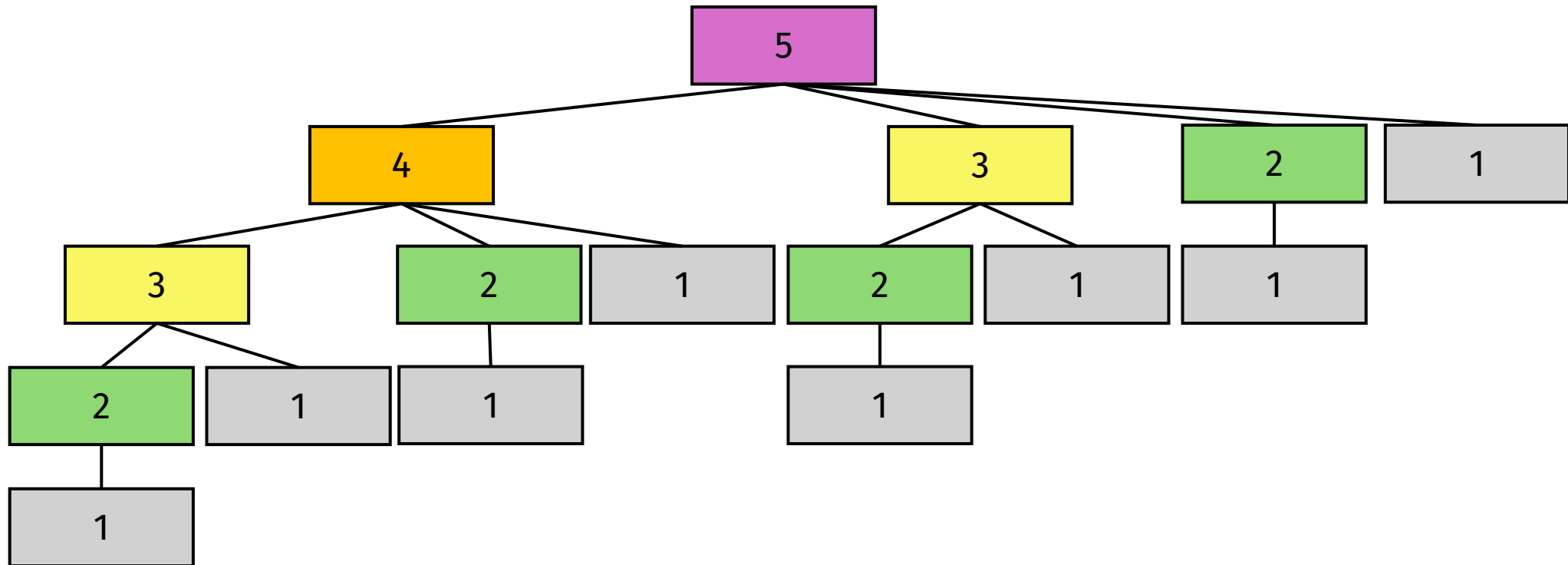
Programm Heute

- Recap
- Schneiden von Eisenstäben
- Matrix-Ketten-Multiplikation
- Zusammenfassung

1. Recap

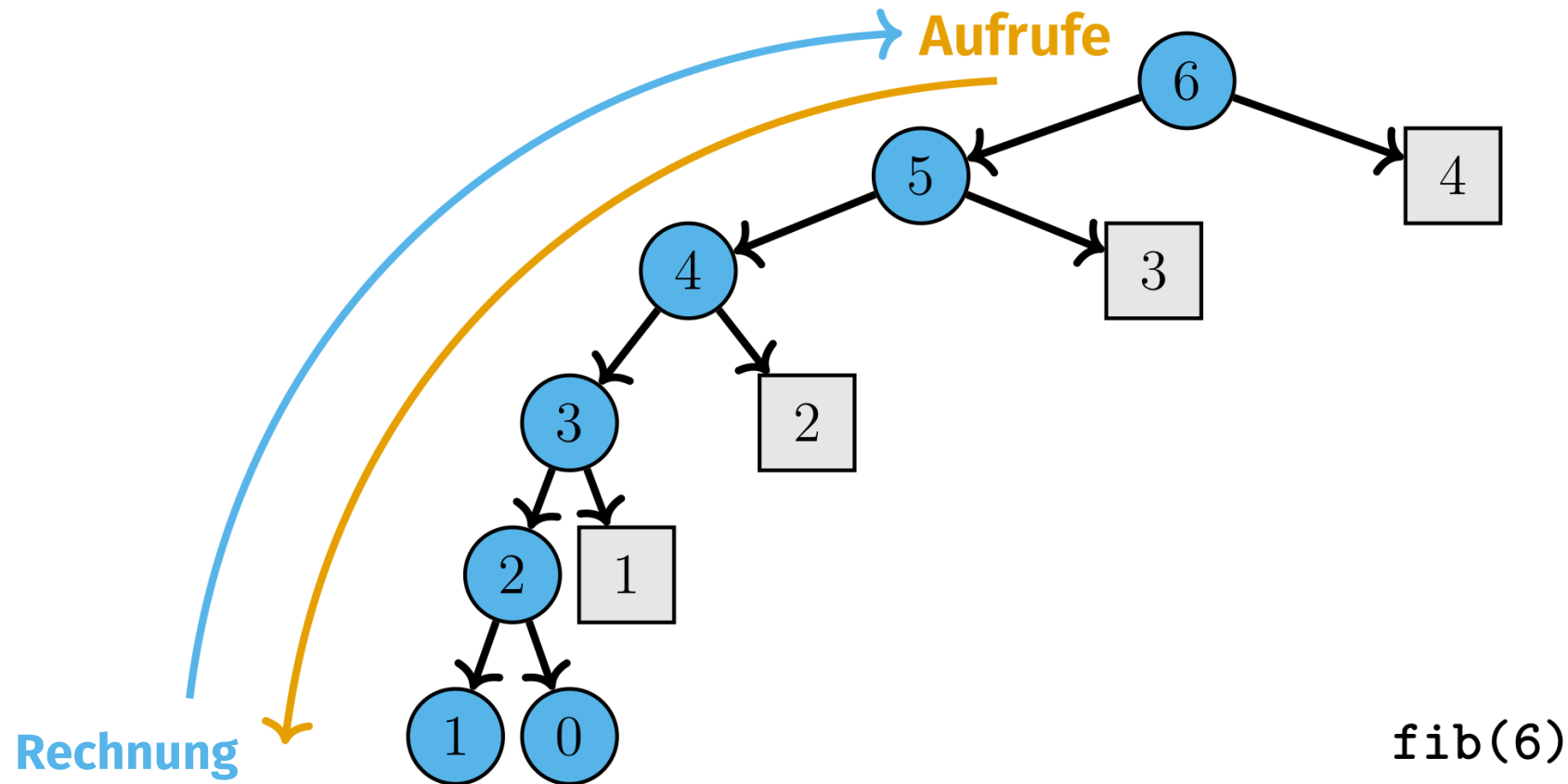
Zusammenfassung der Vorlesung

Für Probleme, die rekursiv gelöst werden können, können wir optimierte Lösungen wie Memoisierung (Top-Down) oder dynamische Programmierung (Bottom-Up) nutzen, um wiederholte Rechnungen zu vermeiden.



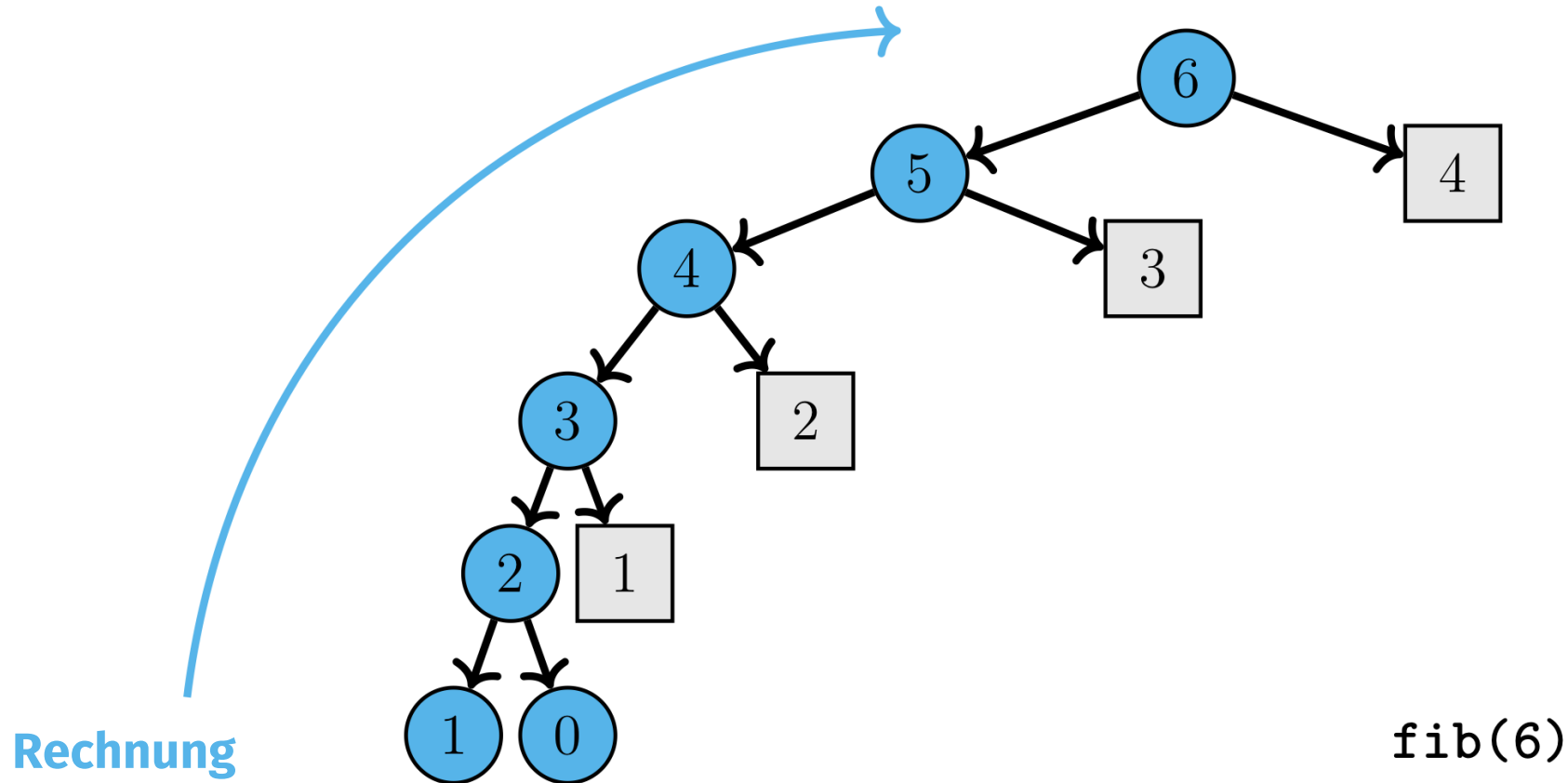
Memoisierung

Wir nutzen Memoisierung, indem wir Lösungen für Subprobleme in einer Tabelle speichern (memoisieren) und diese Tabelle mit jedem Funktionsaufruf mitgeben.



Dynamic Programming

Wir können das Problem iterativ lösen, indem wir beim Basisfall beginnen und eine passende Datenstruktur wie eine DP-Tabelle nutzen, um alle Subprobleme effizient zu berechnen.



Memoisierung != Dynamic Programming

- Memoisierung

- Dynamic Programming

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

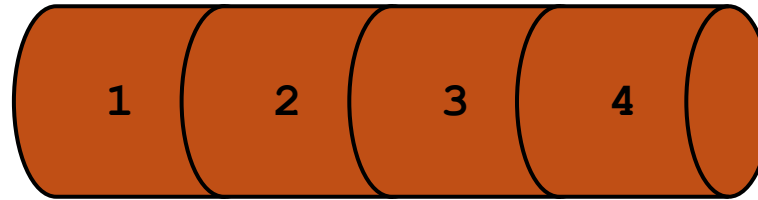
- Ansatz?
Löst nur die notwendigen Teilprobleme.
- Rekursion?
Wird mit Rekursion implementiert, dabei wird eine Tabelle bei jedem Aufruf mitgegeben.
- Berechnungsrichtung?
Top-down.

■ 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. Schneiden von Eisenstäben

Problemstellung

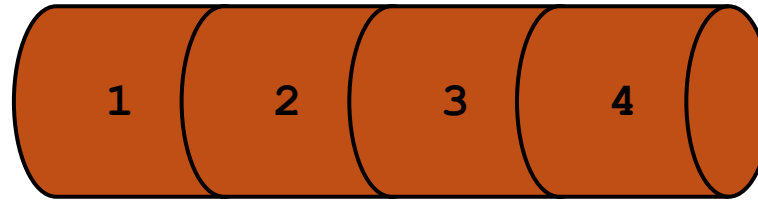


Length i	1	2	3	4
Full price $p[i]$	5	12	1	25

■ Input:

- Eine Stange mit Länge n

Problemstellung

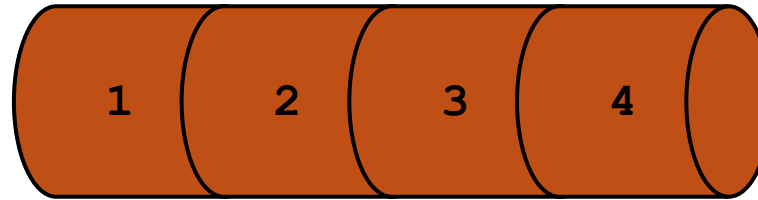


Length i	1	2	3	4
Full price $p[i]$	5	12	1	25

■ Input:

- Eine Stange mit Länge n
- Verschieden lange Stücke haben verschiedene Preise

Problemstellung



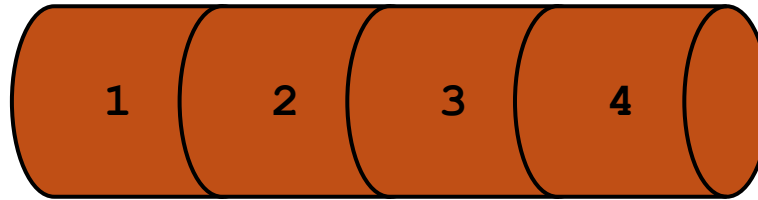
Length i	1	2	3	4
Full price $p[i]$	5	12	1	25

■ Input:

- Eine Stange mit Länge n
- Verschieden lange Stücke haben verschiedene Preise

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

Problemstellung



Length i	1	2	3	4
Full price $p[i]$	5	12	1	25

■ Input:

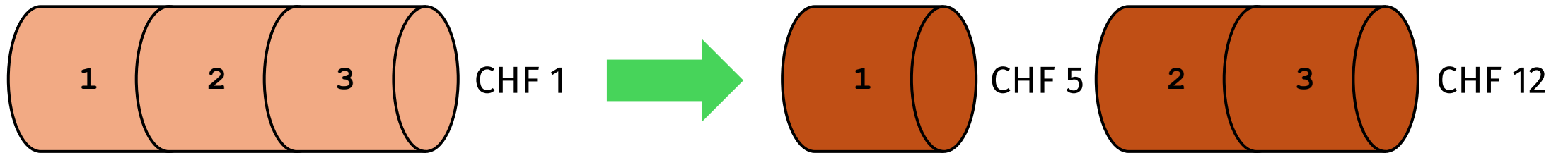
- Eine Stange mit Länge n
- Verschieden lange Stücke haben verschiedene Preise





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

■ Output: Maximaler Preis für die Stange

Beschreibung des Problems

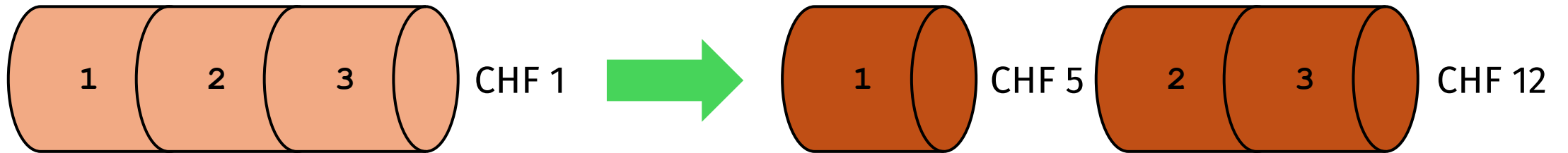
- Stange der Länge 3 hat tiefen Preis $p[3]$







				
Länge i	1	2	3	4
Ganzpreis $p[i]$	5	12	1	25

Beschreibung des Problems


- Stange der Länge 3 hat tiefen Preis $p[3]$
- Man zersägt eine 3er-Stange also in zwei bessere Teile!



				
Länge i	1	2	3	4
Ganzpreis $p[i]$	5	12	1	25

Beschreibung des Problems


- Die besten Schnitte sollen systematisch gesucht werden.



Länge i	1	2	3	4
Ganzpreis $p[i]$	5	12	1	25
Bestpreis $r[i]$	5	12	17	25

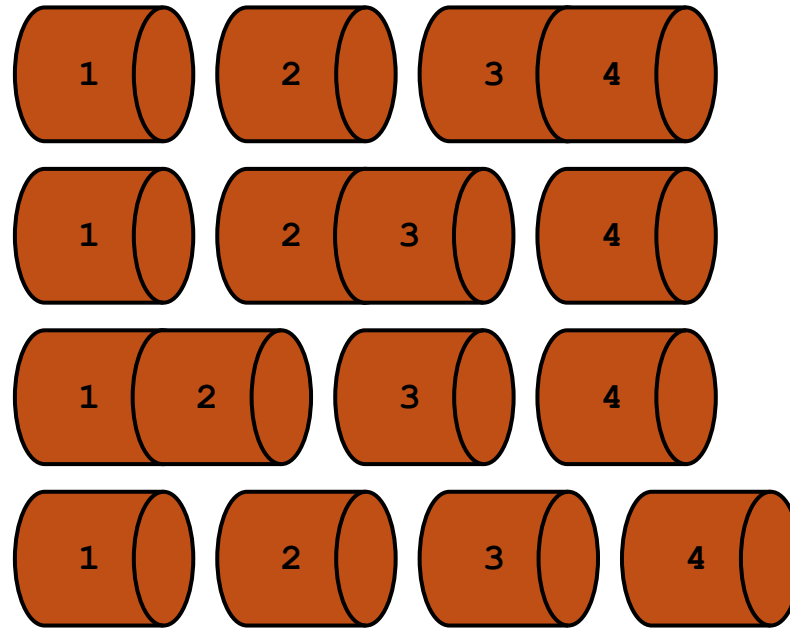
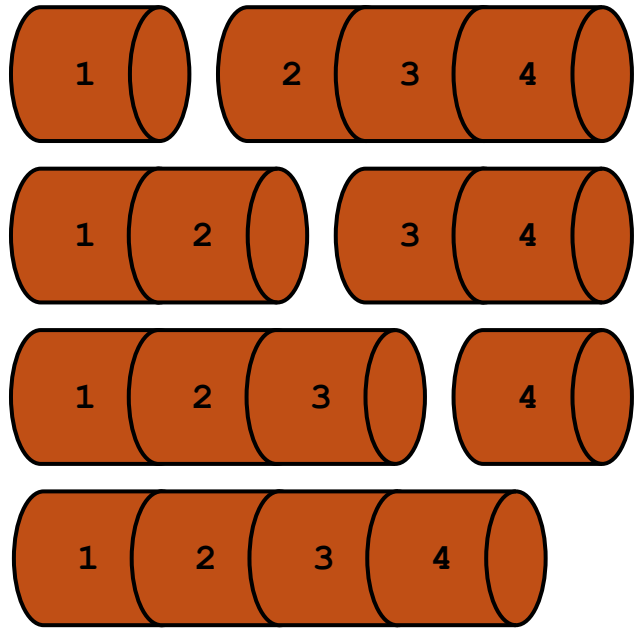
Beschreibung des Problems

- Die besten Schnitte sollen systematisch gesucht werden.
- Gesucht ist *der beste Preis für die Länge i , genannt $r[i]$* , wenn die Stange beliebig geschnitten werden darf.



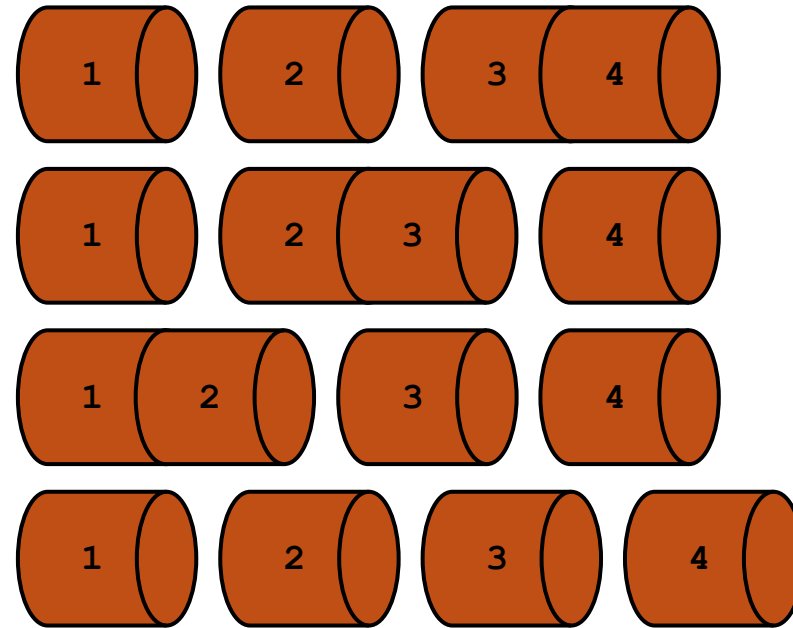
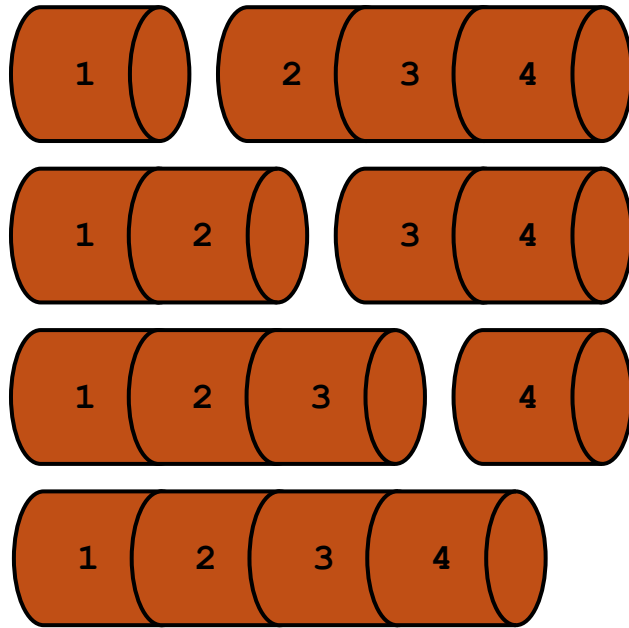
Länge i	1	2	3	4
Ganzpreis $p[i]$	5	12	1	25
Bestpreis $r[i]$	5	12	17	25

Mögliche Lösungsansätze



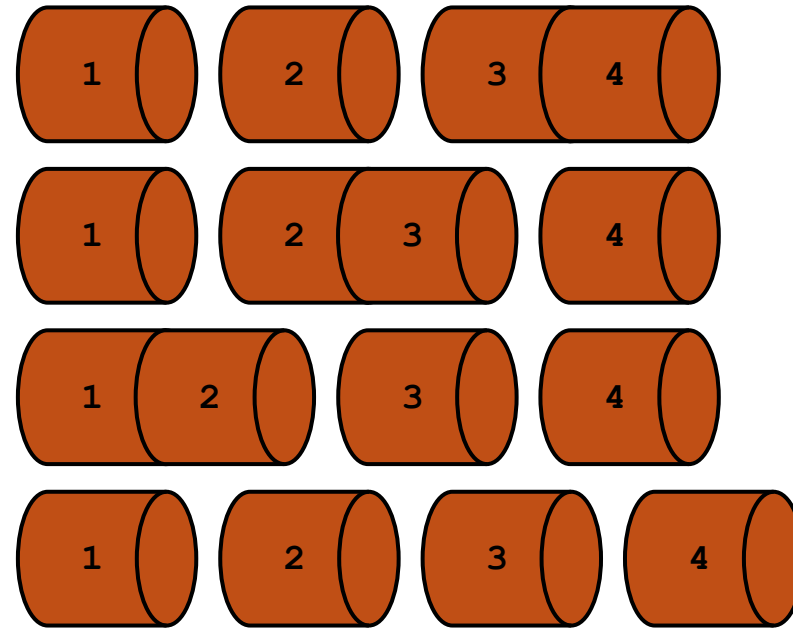
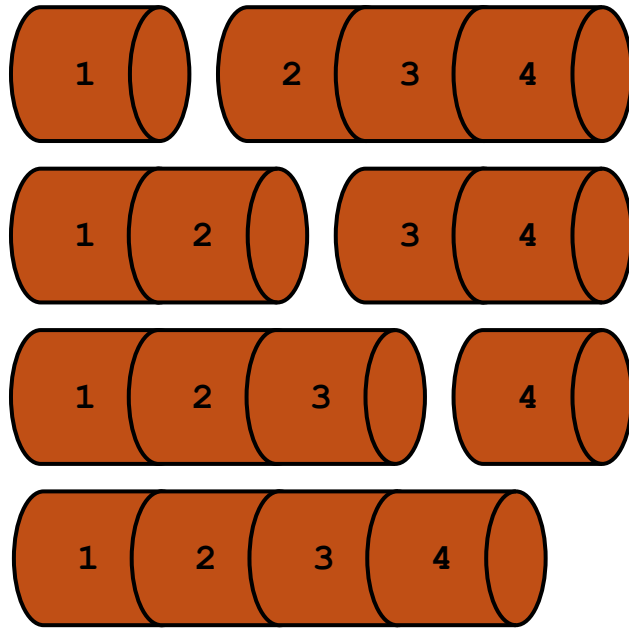
- Alle möglichen Schritte ausprobieren wäre unpraktisch.

Mögliche Lösungsansätze



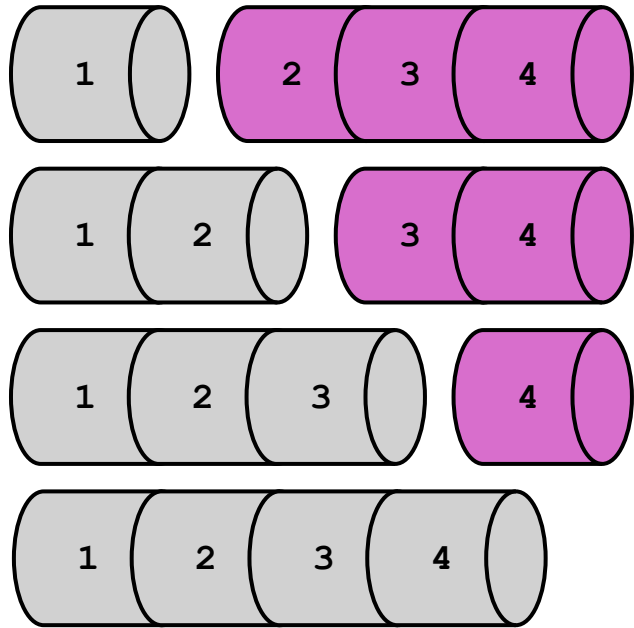
- Alle möglichen Schritte ausprobieren wäre unpraktisch.
 - Die Zahl möglicher Schnitte skaliert exponentiell.

Mögliche Lösungsansätze

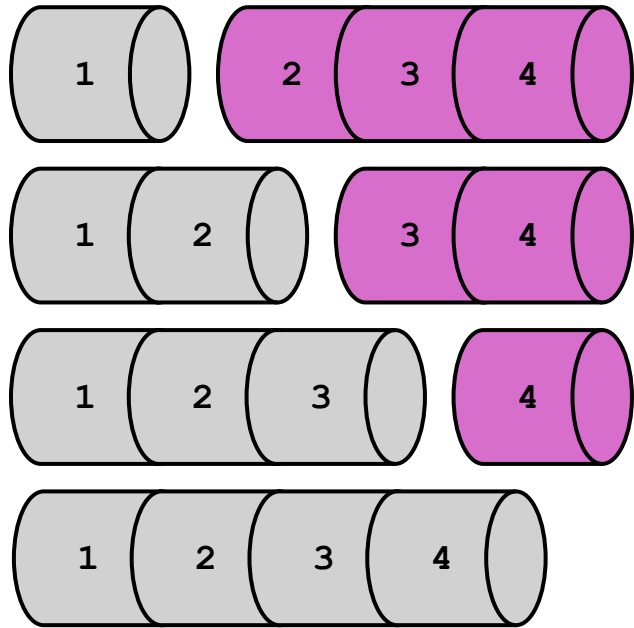


- Alle möglichen Schritte ausprobieren wäre unpraktisch.
 - Die Zahl möglicher Schritte skaliert exponentiell.
- Gesucht sind Vereinfachungen!

Schritt 1: Optimale Substruktur

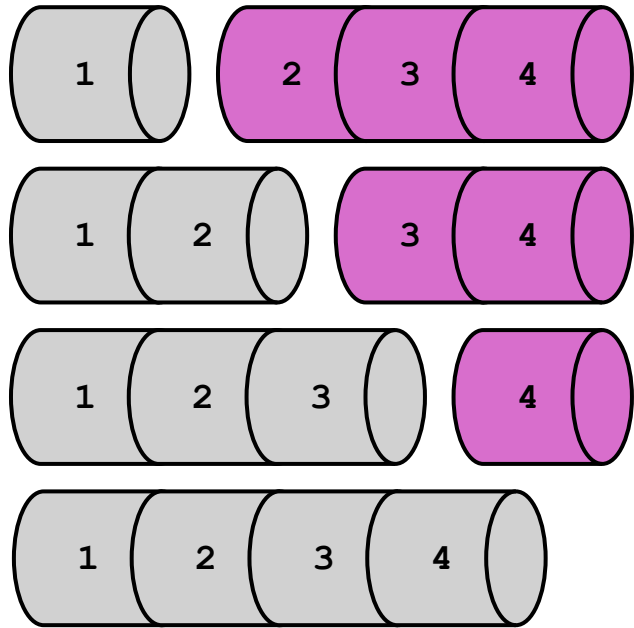


Schritt 1: Optimale Substruktur



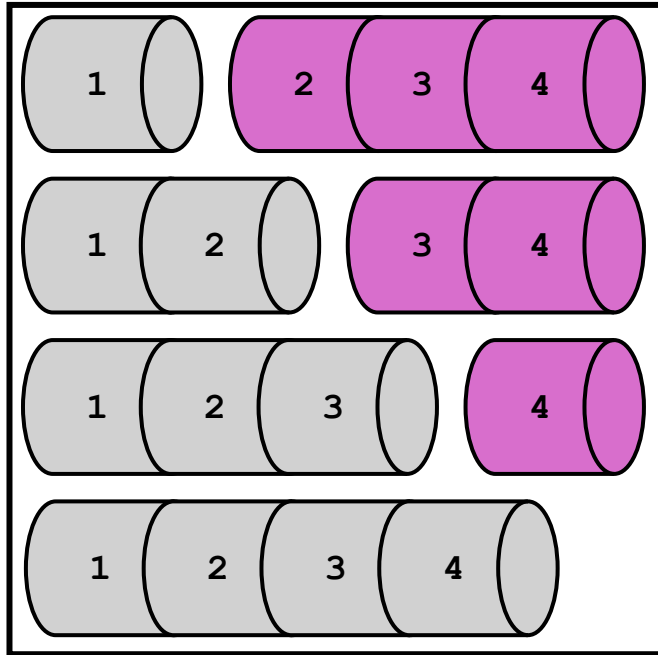
- Die besten Preise von kleinen i können zum berechnen möglicher Preise für grosse i benutzt werden.

Schritt 1: Optimale Substruktur



- Die besten Preise von kleinen i können zum berechnen möglicher Preise für grosse i benutzt werden.
- Die **rechte** Stange wird nochmal weiter zerschnitten. Die **Linke** bleibt ein Stück.

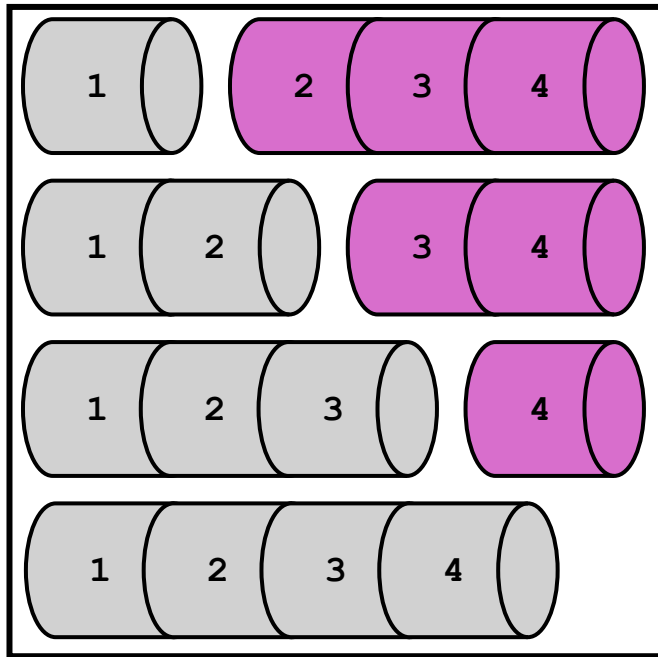
Schritt 2: Konstruktion der optimalen Lösung



max. Preis

- Zwei Schritte zur Lösung

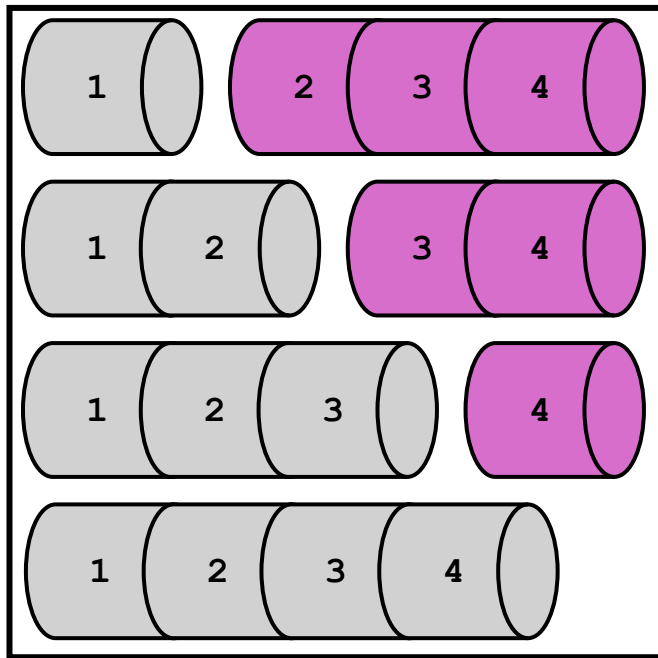
Schritt 2: Konstruktion der optimalen Lösung



max. Preis

- Zwei Schritte zur Lösung
 - **Preise für alle möglichen Schnitte** berechnen.

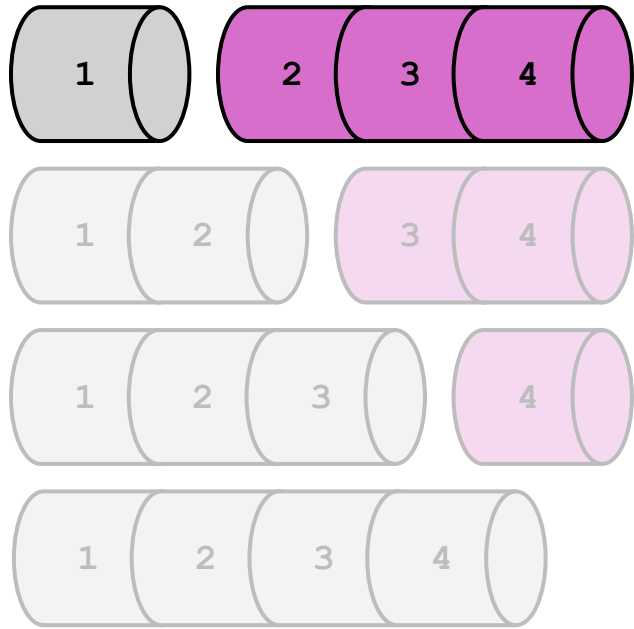
Schritt 2: Konstruktion der optimalen Lösung



max. Preis

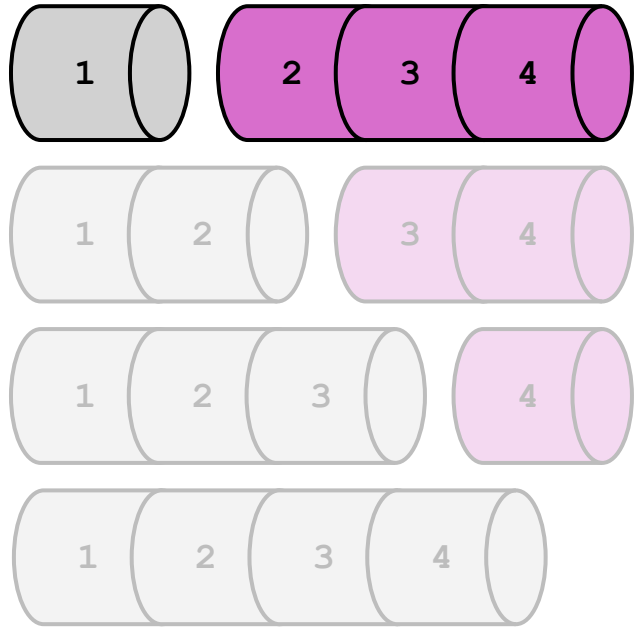
- Zwei Schritte zur Lösung
 - **Preise für alle möglichen Schnitte** berechnen.
 - Das **Maximum finden**, und als $r[i]$ speichern.

Schritt 3: Rekursive Beschreibung



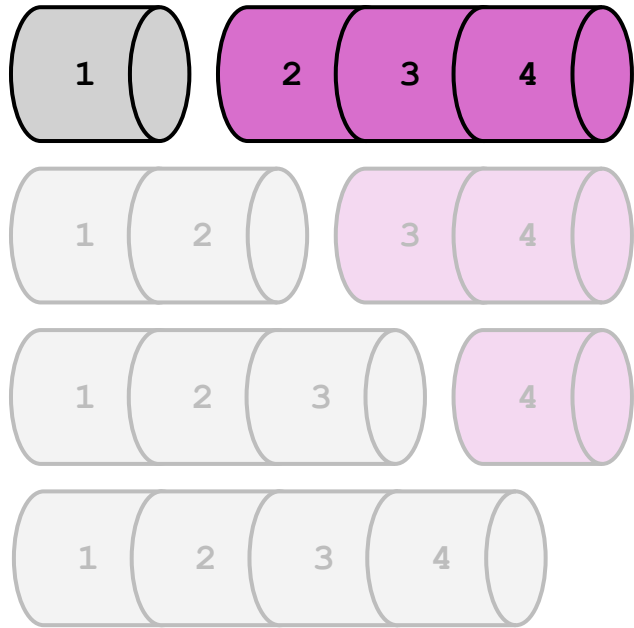
- Beispiel: Was wäre der Preis nach diesem Schnitt?

Schritt 3: Rekursive Beschreibung



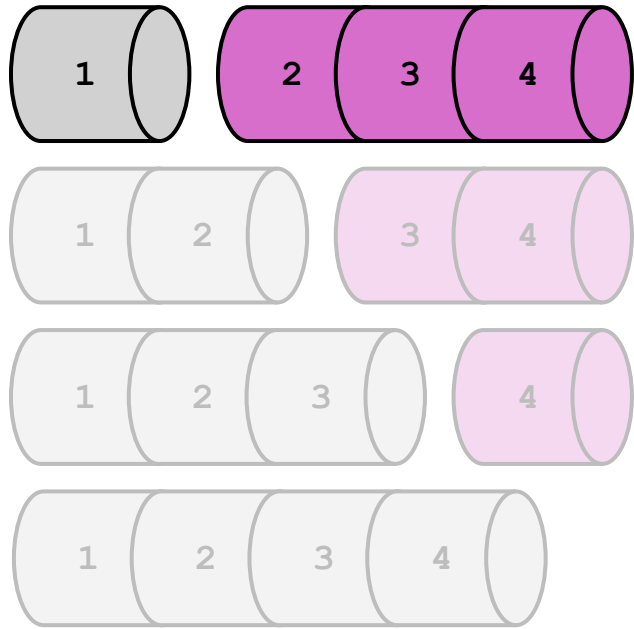
- Beispiel: Was wäre der Preis nach diesem Schnitt?
 - Preis für **1er-Stange** ist bekannt: $p[1]$

Schritt 3: Rekursive Beschreibung



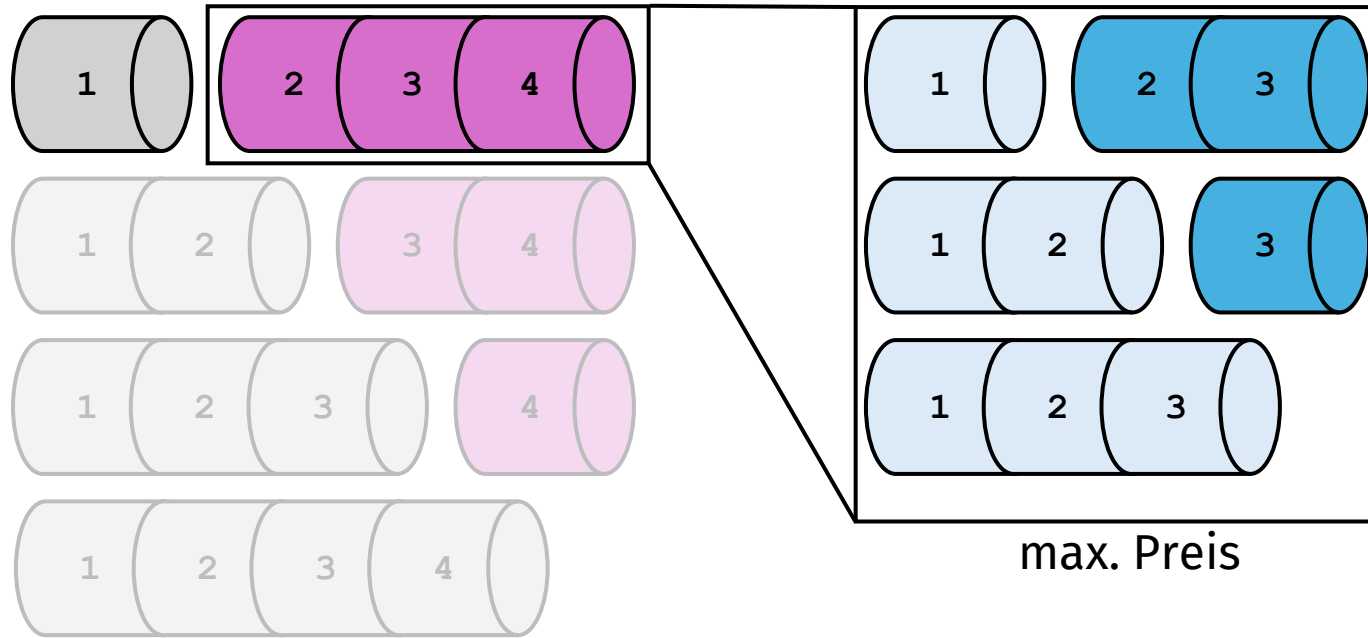
- Beispiel: Was wäre der Preis nach diesem Schnitt?
 - Preis für **1er-Stange** ist bekannt: $p[1]$
 - Gesucht ist noch der Bestpreis für **3er-Stange**: $r[3]$

Schritt 3: Rekursive Beschreibung



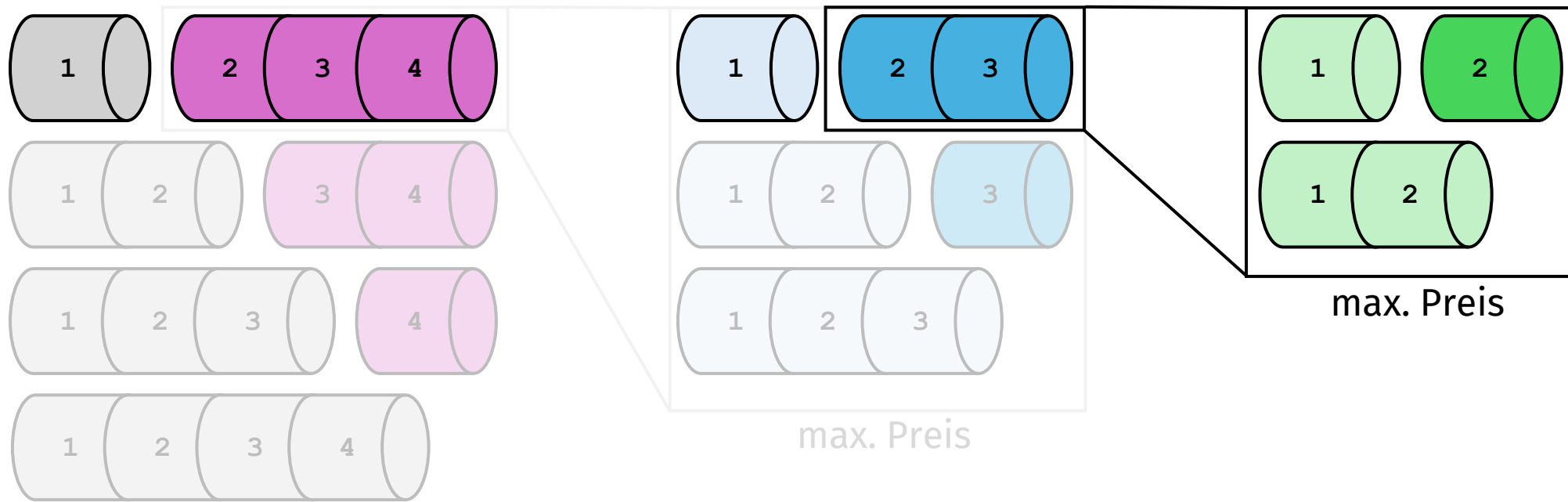
- Beispiel: Was wäre der Preis nach diesem Schnitt?
 - Preis für **1er-Stange** ist bekannt: $p[1]$
 - Gesucht ist noch der Bestpreis für **3er-Stange**: $r[3]$
 - Preis ist dann $p[1] + r[3]$

Schritt 3: Rekursive Beschreibung



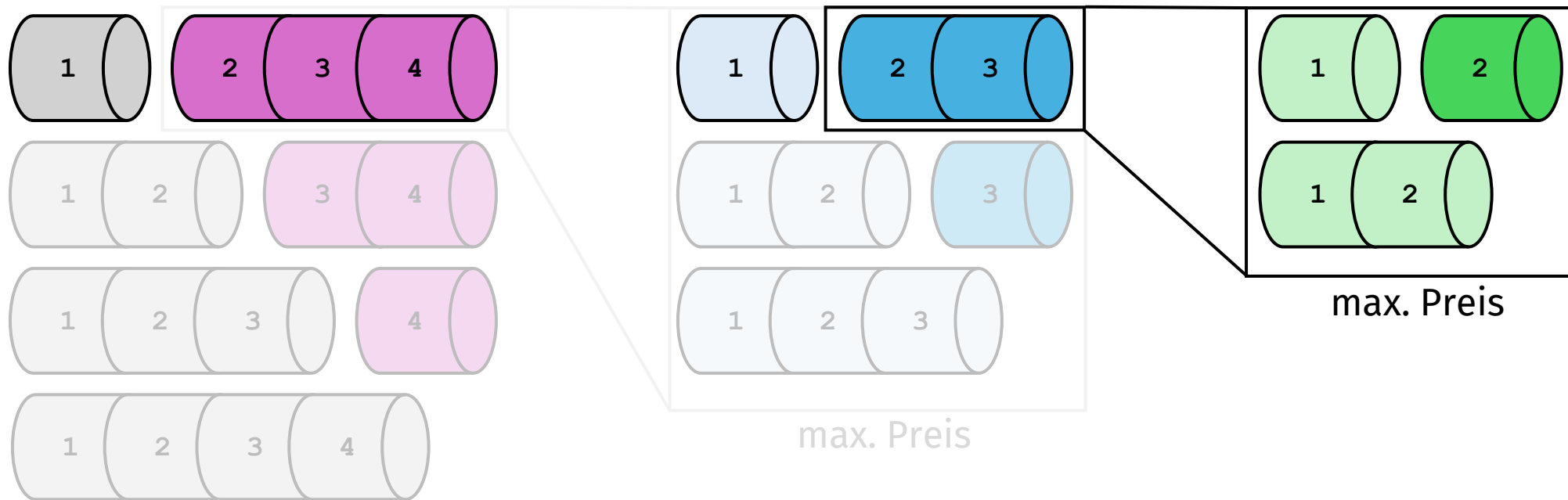
- Bestpreis für 3er-Stange $r[3]$ kann nach gleichem Prinzip berechnet werden...

Schritt 3: Rekursive Beschreibung



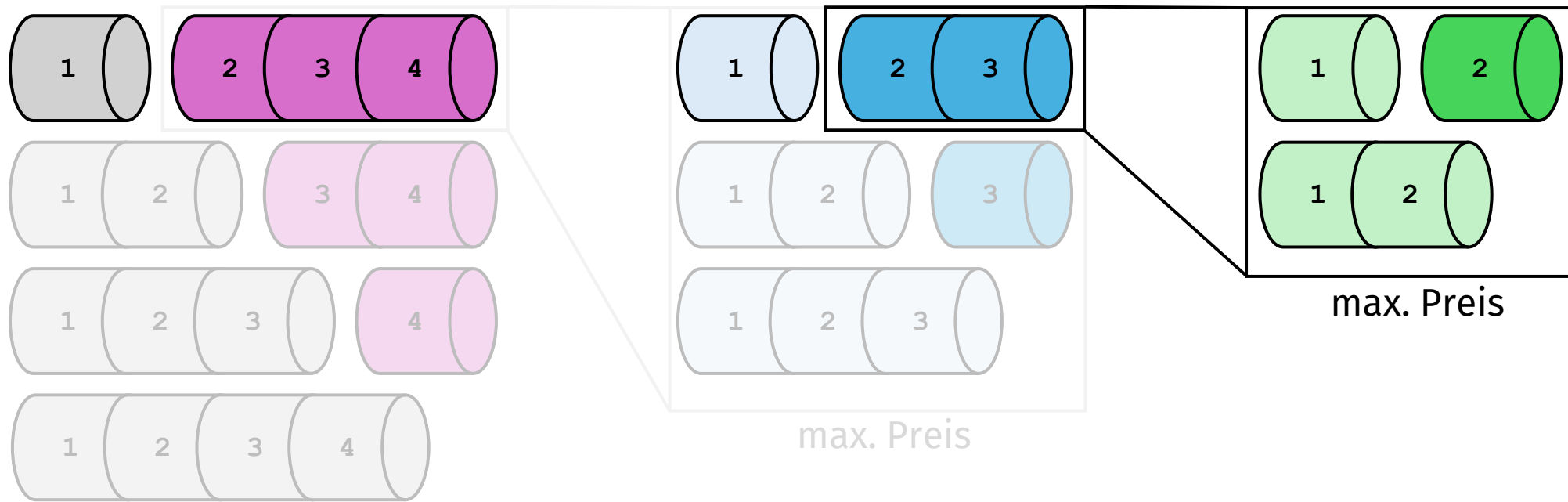
- Für diesen Schnitt muss wiederum der Bestpreis für 2er-Stangen $r[2]$ gefunden werden...

Schritt 3: Rekursive Beschreibung



- Für diesen Schnitt muss wiederum der Bestpreis für 2er-Stangen $r[2]$ gefunden werden...
 - **Ende der Rekursion, 1er-Stücke können nicht zersägt werden.**

Schritt 3: Rekursive Beschreibung

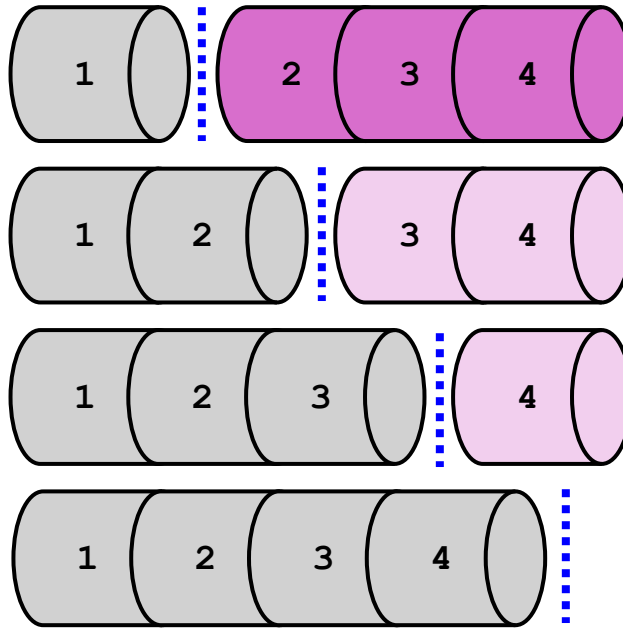


- Für diesen Schnitt muss wiederum der Bestpreis für 2er-Stangen $r[2]$ gefunden werden...

■ **Ende der Rekursion, 1er-Stücke können nicht zersägt werden.**

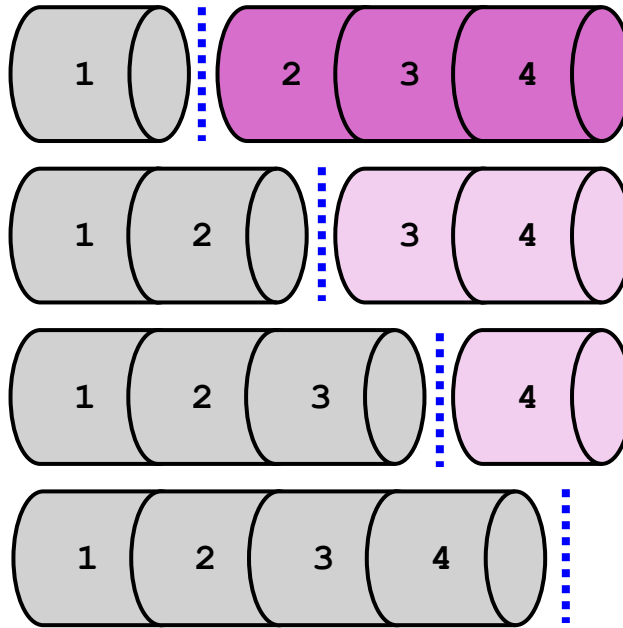
↑ Das ist der Basisfall!

Schritt 4: Berechnung aus Zwischenergebnissen



$$r[i] = \begin{cases} p[i] & \text{wenn } i = 1 \\ \max(p[i] + r[i - j] \text{ für } j \in [1, i]) & \text{wenn } i > 1 \end{cases}$$

Schritt 4: Berechnung aus Zwischenergebnissen



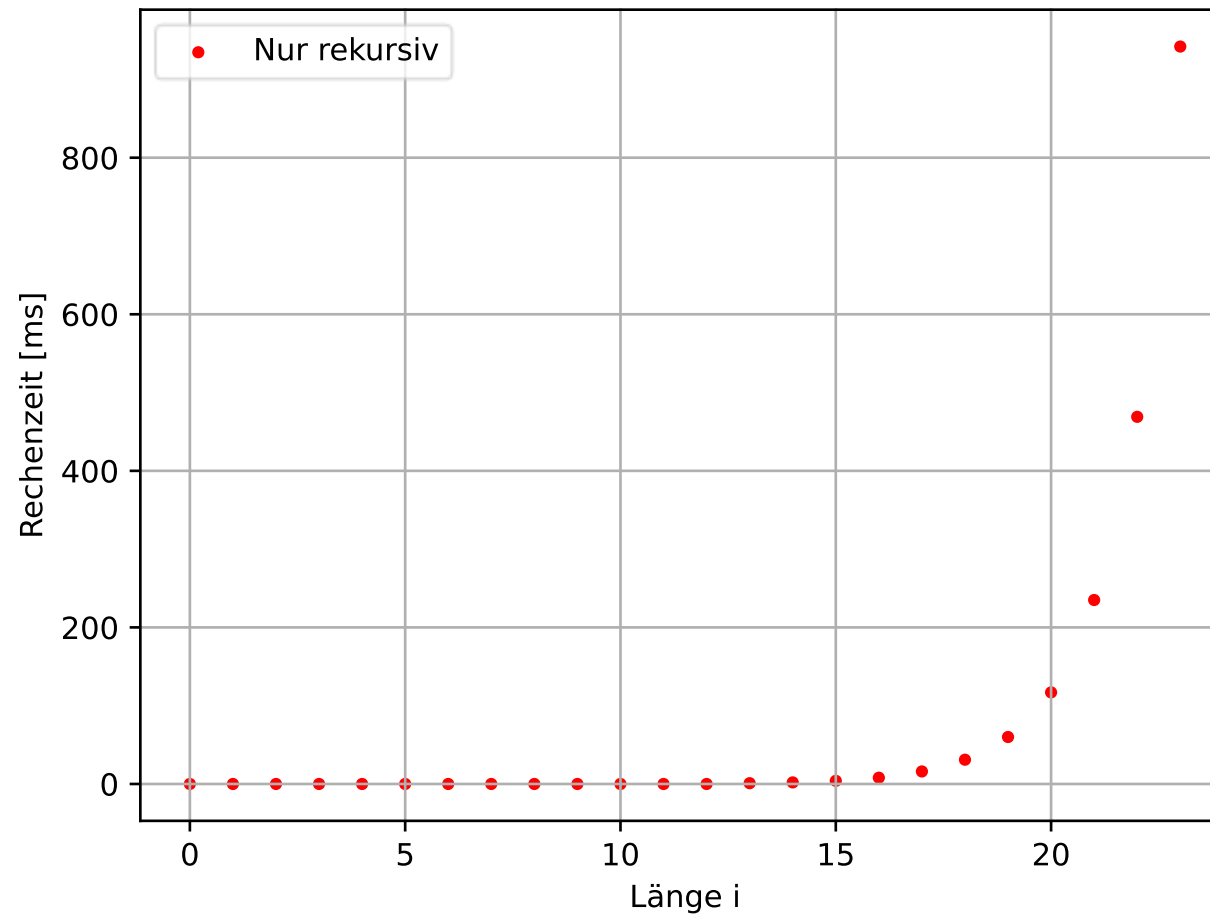
$$r[i] = \begin{cases} p[i] & \text{wenn } i = 1 \\ \max(p[i] + r[i-j] \text{ für } j \in [1, i]) & \text{wenn } i > 1 \end{cases}$$

- $r[i-j]$ wird rekursiv berechnet, wie im Beispiel.

Lösungsansatz in Python

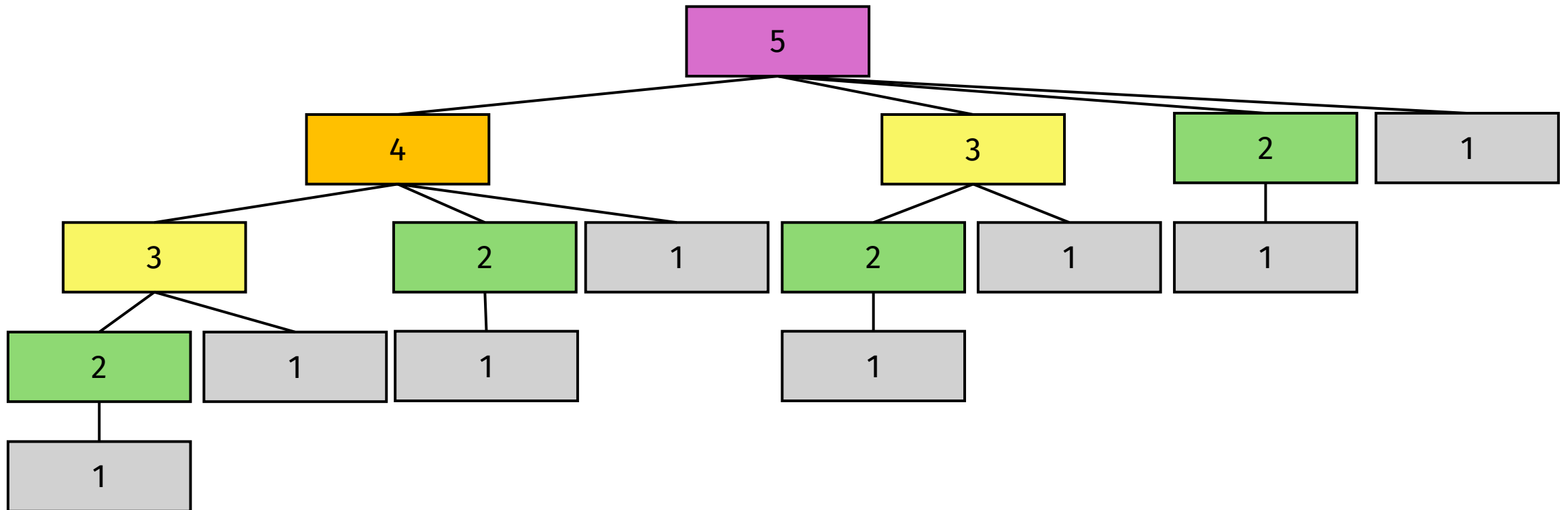
```
def best_price(prices, i):  
    # Basisfall der Rekursion:  
    if i == 1:  
        return prices[1]  
  
    # Rekursive Fälle  
    max_p = prices[i-1]  
    for j in range(1, i):  
        max_p = max(prices[j-1] + best_price(prices, i-j), max_p)  
    return max_p
```

Laufzeit-Diagramm



Rekursive Laufzeit: $\Theta(2^n)$

Funktions-Aufrufe



- Insgesamt $T(i) = 1 + \sum_{j=1}^i T(i - j)$ Aufrufe

Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

Schlaue "Richtung" finden

Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!

Schlaue "Richtung" finden

Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!
- Was, wenn sich das Programm an bekannte Werte erinnern könnte?

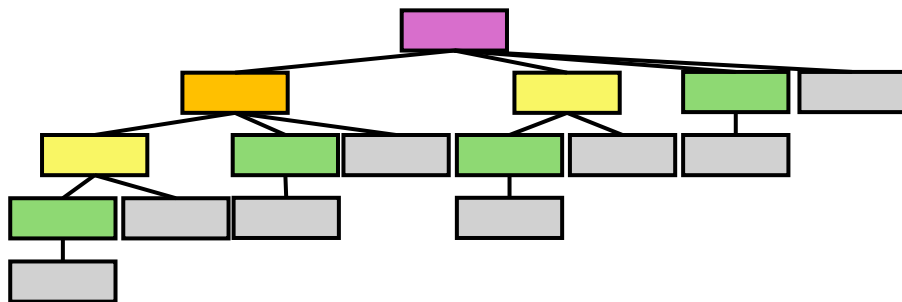
Schlaue "Richtung" finden

Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!
- Was, wenn sich das Programm an bekannte Werte erinnern könnte?

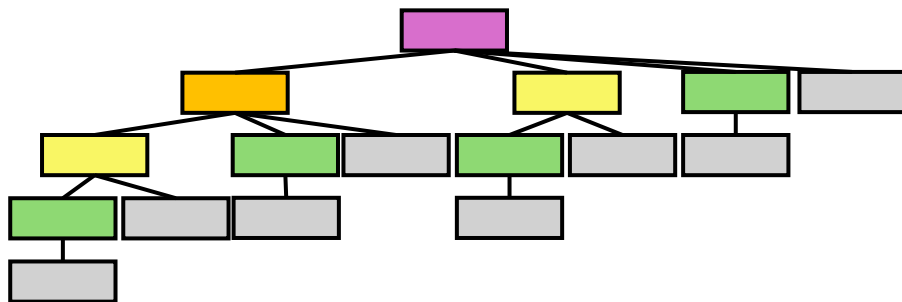
Schlaue "Richtung" finden



Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!
- Was, wenn sich das Programm an bekannte Werte erinnern könnte?



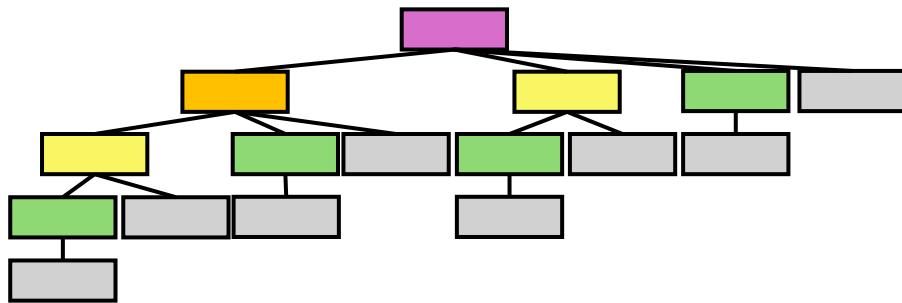
Schlaue "Richtung" finden

- $r[4]$ braucht $r[3]$, aber $r[3]$ braucht *nie* $r[4]$!

Zwei mögliche Verbesserungen für Schritt 4

An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!
- Was, wenn sich das Programm an bekannte Werte erinnern könnte?



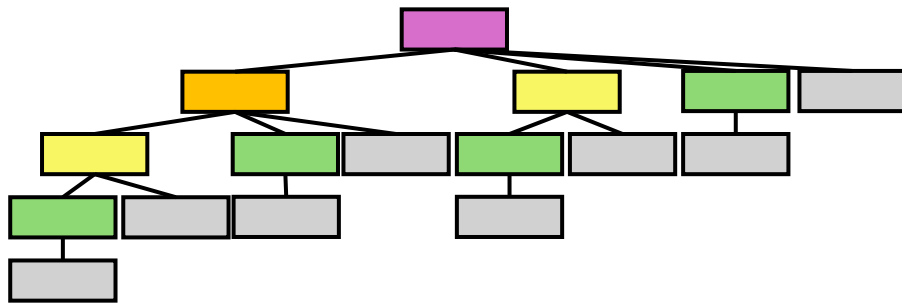
Schlaue "Richtung" finden

- $r[4]$ braucht $r[3]$, aber $r[3]$ braucht *nie* $r[4]$!
- Man könnte Probleme in schlauer Reihenfolge von "unten nach oben" berechnen!

Zwei mögliche Verbesserungen für Schritt 4

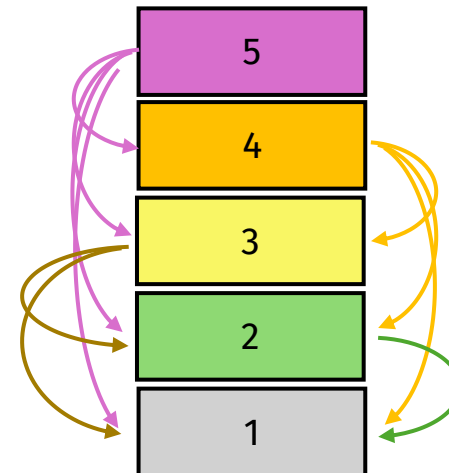
An Lösungen erinnern

- Jeder Wert wird mehrfach berechnet!
- Was, wenn sich das Programm an bekannte Werte erinnern könnte?

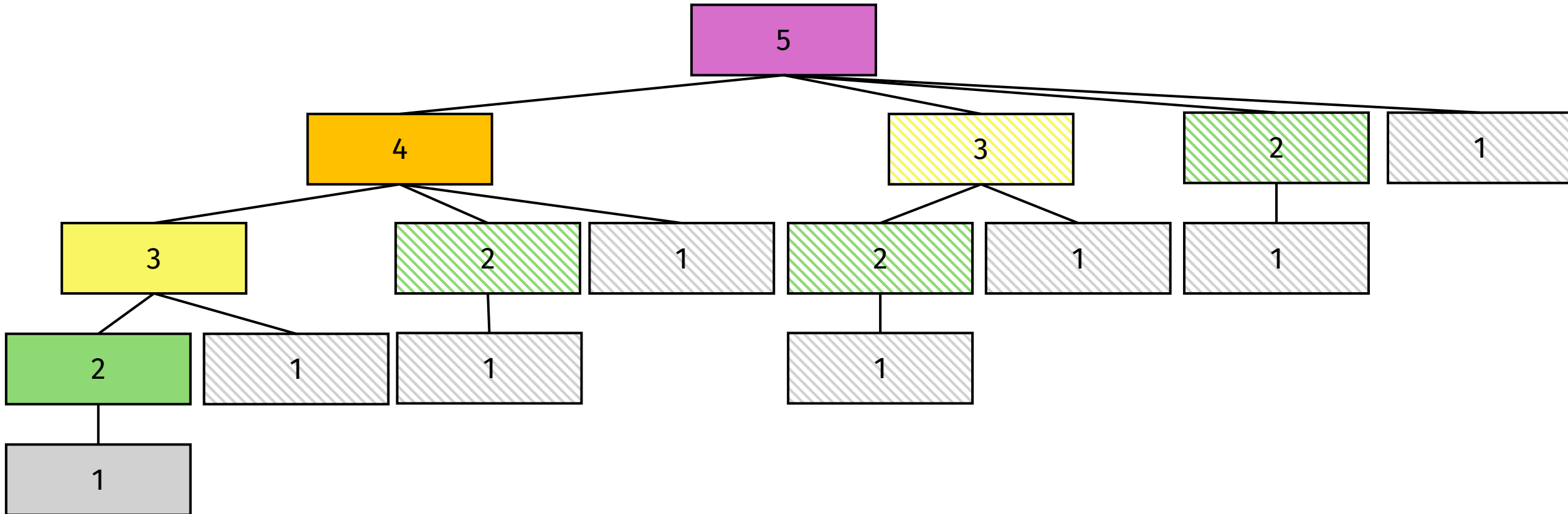


Schlaue "Richtung" finden

- $r[4]$ braucht $r[3]$, aber $r[3]$ braucht *nie* $r[4]$!
- Man könnte Probleme in schlauer Reihenfolge von "unten nach oben" berechnen!



Funktions-Aufrufe mit Memoisierung



- i Aufrufe, in jedem Aufruf eine max-Funktion, total also $T(i) = i^2$

Memoisierung in Python, Theorie

- In Python kann Memoisierung einfach mit einem Dictionary implementiert werden.

Memoisierung in Python, Theorie

- In Python kann Memoisierung einfach mit einem Dictionary implementiert werden.
- Für jeden Aufruf, suche zuerst im Dictionary nach einer Lösung...

Memoisierung in Python, Theorie

- In Python kann Memoisierung einfach mit einem Dictionary implementiert werden.
- Für jeden Aufruf, suche zuerst im Dictionary nach einer Lösung...
 - Wenn gefunden: Gebe Wert aus Dictionary zurück, kein rechnen

Memoisierung in Python, Theorie

- In Python kann Memoisierung einfach mit einem Dictionary implementiert werden.
- Für jeden Aufruf, suche zuerst im Dictionary nach einer Lösung...
 - Wenn gefunden: Gebe Wert aus Dictionary zurück, kein rechnen
 - Wenn nicht: Berechne Wert, speichere vor Rückgabe im Dictionary

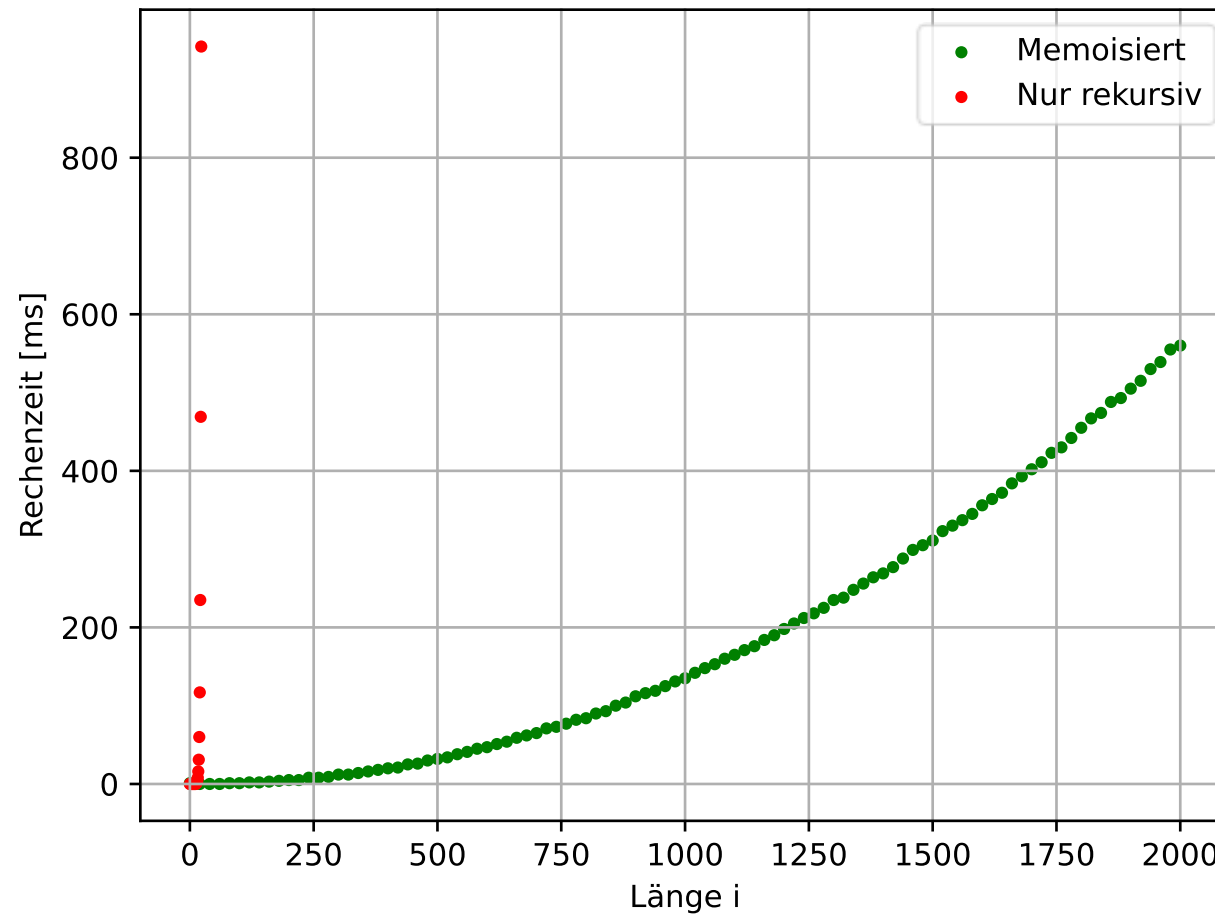
Memoisierung in Python, Theorie

- In Python kann Memoisierung einfach mit einem Dictionary implementiert werden.
- Für jeden Aufruf, suche zuerst im Dictionary nach einer Lösung...
 - Wenn gefunden: Gebe Wert aus Dictionary zurück, kein rechnen
 - Wenn nicht: Berechne Wert, speichere vor Rückgabe im Dictionary
- Gebe *dasselbe* Dictionary an alle Funktions-Aufrufe weiter!

Memoisierung in Python

```
def best_p_memo(prices, i, mem=None):  
    # Erinnerung  
    if mem is None:  
        mem = dict()  
    if i in mem:  
        return mem[i]  
  
    # Basisfall der Rekursion:  
    ...  
  
    # Rekursive Fälle  
    ... best_p_memo(prices, i-j, mem) ...  
  
    mem[i] = max_p  
    return max_p
```

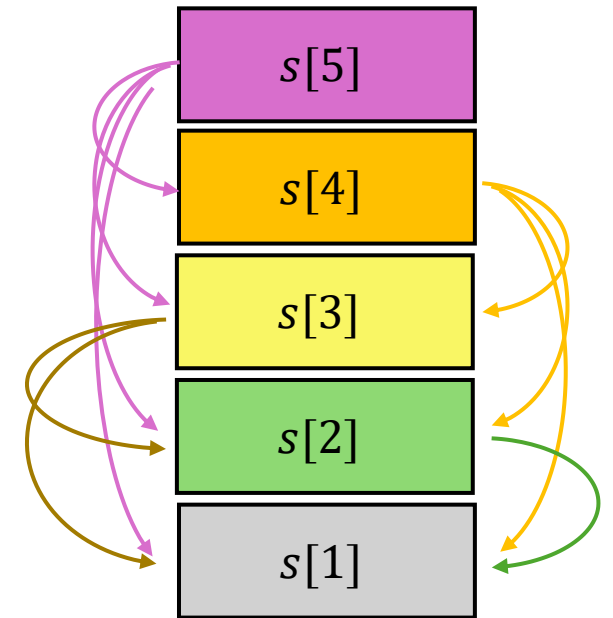
Laufzeit-Diagramm



Memoisiert Laufzeit: $\Theta(n^2)$

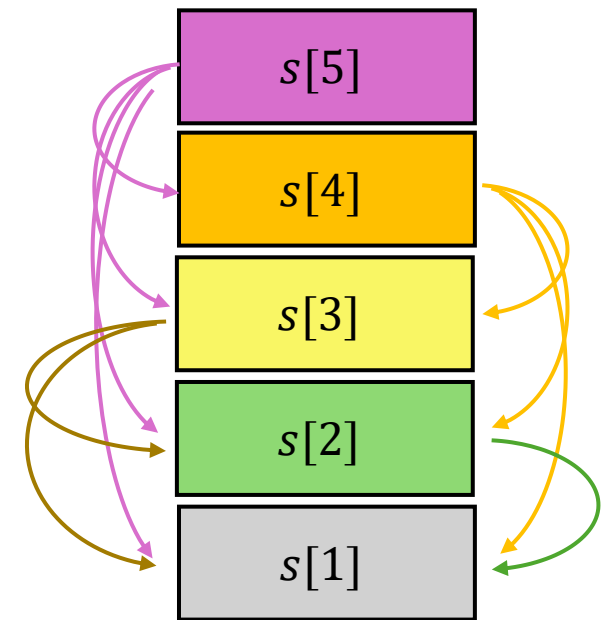
Bottom-Up in Python, Theorie

- Um Länge 3 zu berechnen, muss nie Länge 4 berechnet werden.



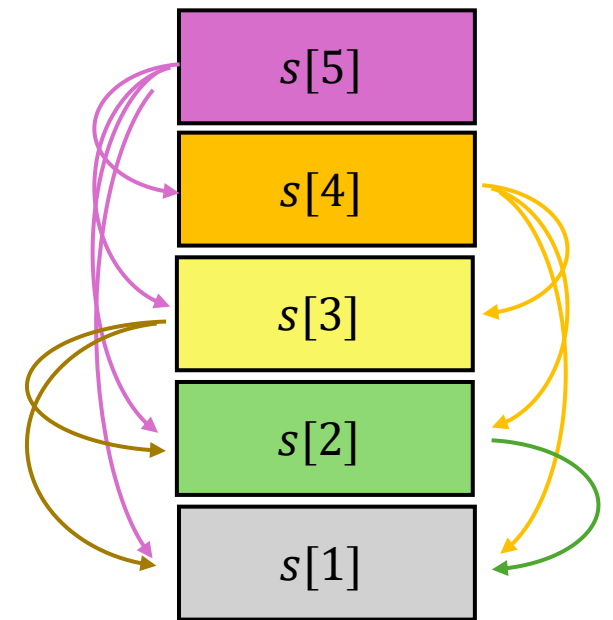
Bottom-Up in Python, Theorie

- Um Länge 3 zu berechnen, muss nie Länge 4 berechnet werden.
 - Logisch, eine 3er-Stange kann nicht in 4er-Stangen zerschnitten werden.



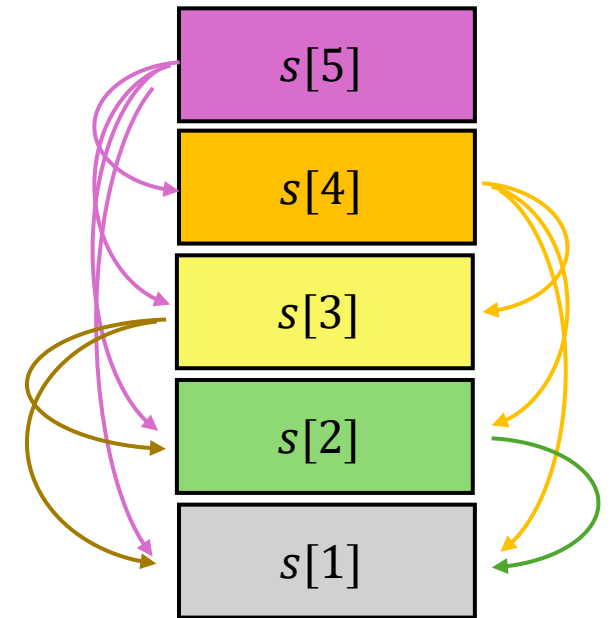
Bottom-Up in Python, Theorie

- Um Länge 3 zu berechnen, muss nie Länge 4 berechnet werden.
 - Logisch, eine 3er-Stange kann nicht in 4er-Stangen zerschnitten werden.
 - Wir nennen dieses Subproblem $s[i]$



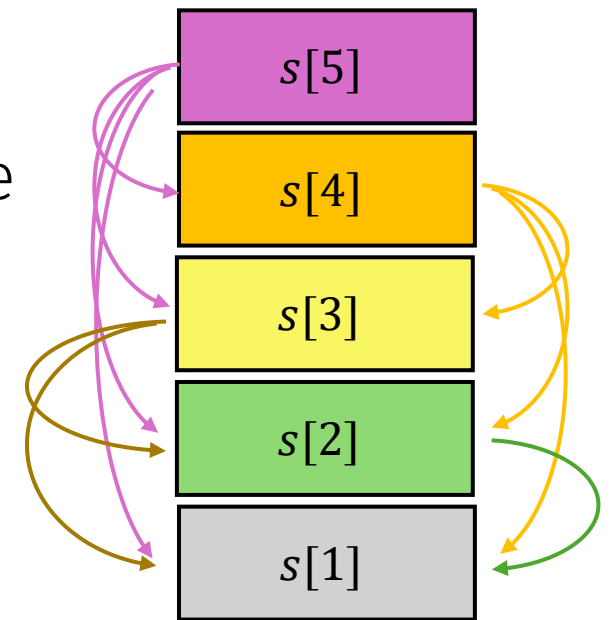
Bottom-Up in Python, Theorie

- Um Länge 3 zu berechnen, muss nie Länge 4 berechnet werden.
 - Logisch, eine 3er-Stange kann nicht in 4er-Stangen zerschnitten werden.
 - Wir nennen dieses Subproblem $s[i]$
- Das Muster setzt sich fort – Bis zur Länge 1!



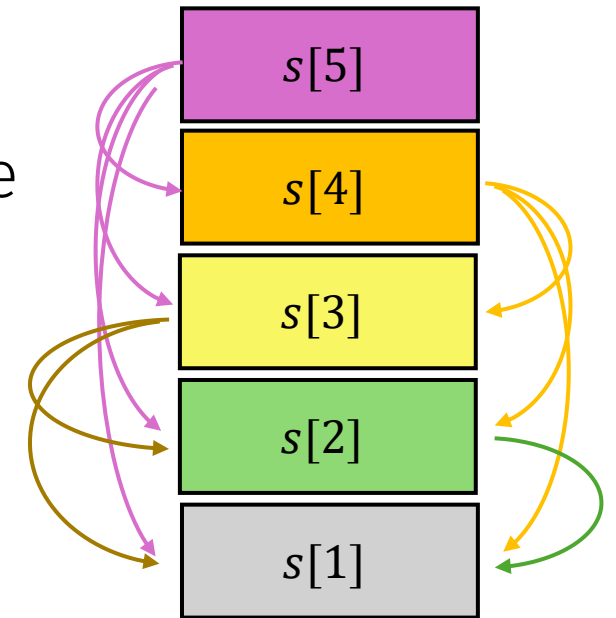
Bottom-Up in Python, Theorie

- Das Problem hat eine Richtung, und deswegen ein unterstes Sub-Problem, das bekannt ist. Seine **Lösung ist selbst nicht auf andere Lösungen angewiesen.**



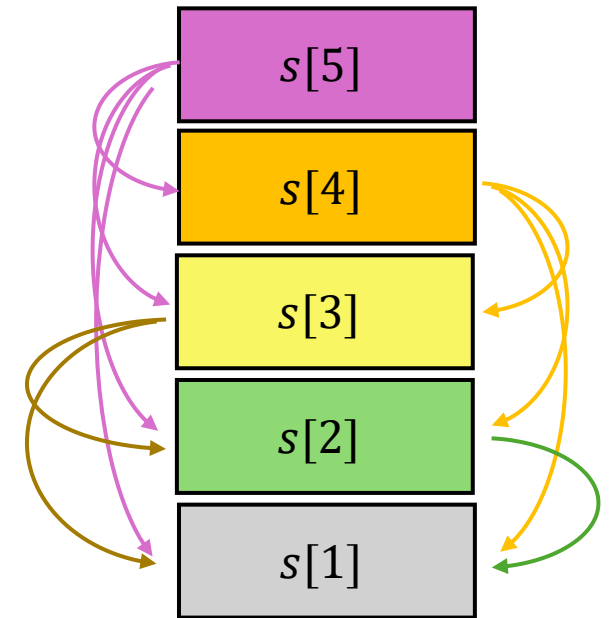
Bottom-Up in Python, Theorie

- Das Problem hat eine Richtung, und deswegen ein unterstes Sub-Problem, das bekannt ist. Seine **Lösung ist selbst nicht auf andere Lösungen angewiesen.**
- Herausforderung: Die "Richtung" für ein Problem finden → oft ersichtlich aus rekursiver Lösung!



Bottom-Up in Python, Theorie

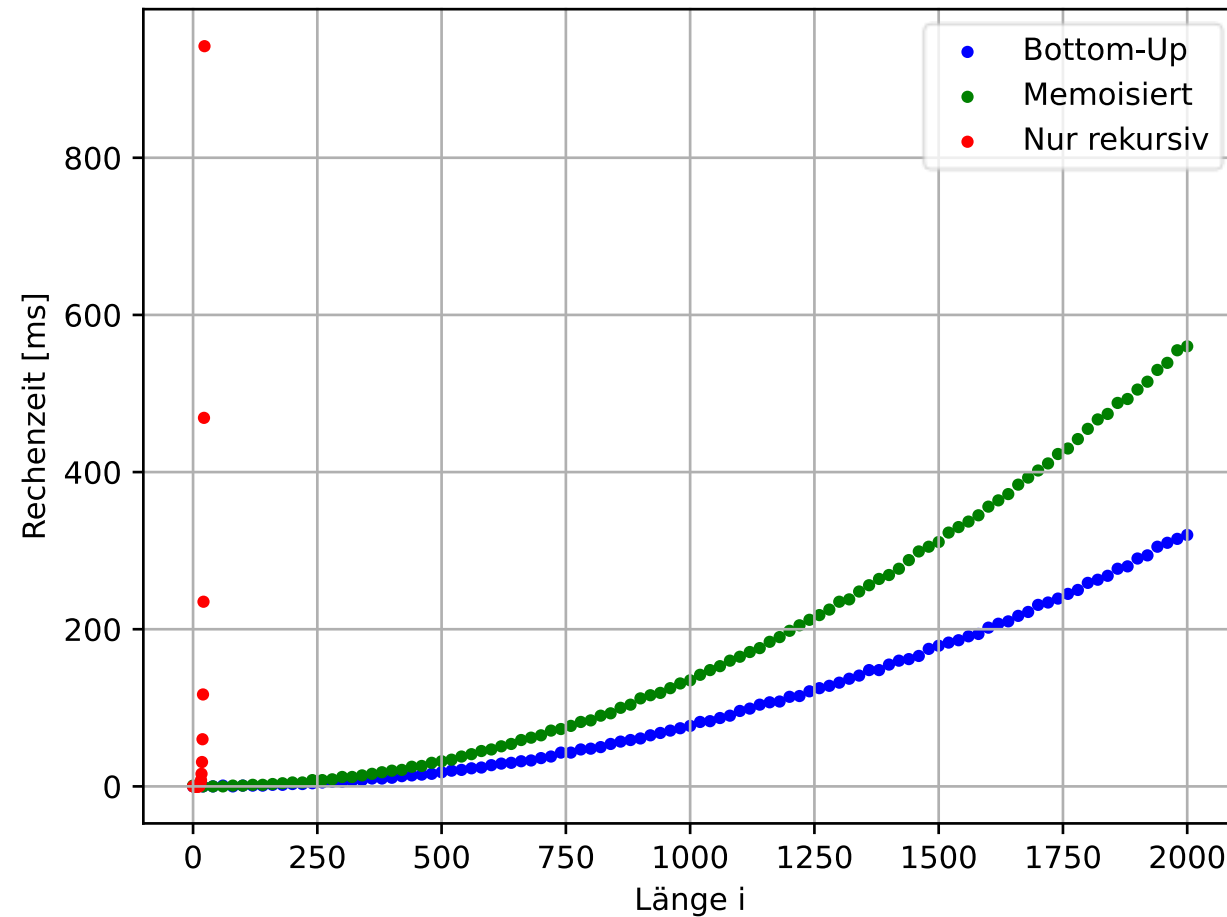
- Python: Eine Struktur (z.B. Liste, wie rechts) speichert berechnete Probleme. Berechne immer "Ende der Kette" mit einer Schleife – jedes vorherige Ergebnis wird an seinem Index in der Struktur gespeichert!



Bottom-Up in Python

```
def best_p_iter(prices, i):  
    # "Struktur" erstellen  
    s = [0] + [None]*i  
  
    # Iteration statt Funktionsaufrufe  
    for j in range(1, i+1):  
        # Genau gleiche Rechnung wie mit Rekursion  
        max_p = prices[j-1]  
        for k in range(1, j):  
            max_p = max(max_p, s[k] + prices[j-k-1])  
        # Fülle Struktur statt Memoisierung  
        s[j] = max_p  
  
    return s[i]
```

Laufzeit-Diagramm



*Memoisiert und Bottom-Up beides $\Theta(n^2)$, mit verschiedenen Konstanten multipliziert. Grund: Mehr Overhead (Aufwand) für "Erinnern".

Bottom-Up Laufzeit: $\Theta(n^2)^*$

3. Matrixmultiplikation

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Matrizen A_1, A_2, \dots, A_n wobei Matrix A_i nicht quadratisch sein muss.

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Matrizen A_1, A_2, \dots, A_n wobei Matrix A_i nicht quadratisch sein muss.
- **Ziel:** Die Reihenfolge finden, in der die Matrixkettenmultiplikation am schnellsten berechnet wird.

Optimale Reihenfolge

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Problem:** Die Berechnung von $A_1 \cdot A_2 \cdot \dots \cdot A_n$ dauert je nach Reihenfolge unterschiedlich lange!

Beispiel

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix}$$

könnte berechnet werden als...

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [6] \quad \text{oder} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Beispiel

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix}$$

könnte berechnet werden als...

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [6] \quad \text{oder} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- Hier macht es keinen Sinn, die Matrix $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ zu berechnen.

Schlussfolgerung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- Für ein gegebenes Problem sind mehrere Berechnungs-Reihenfolgen möglich.

Schlussfolgerung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- Für ein gegebenes Problem sind mehrere Berechnungs-Reihenfolgen möglich.
- Wir legen diese Reihenfolge mit Klammern fest.

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Matrizen A_1, A_2, \dots, A_n wobei Matrix A_i nicht quadratisch sein muss.

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Matrizen A_1, A_2, \dots, A_n wobei Matrix A_i nicht quadratisch sein muss.
- **Ziel:** Die Reihenfolge finden, in der die Matrixkettenmultiplikation $A_1 \cdot A_2 \cdot \dots \cdot A_n$ am schnellsten berechnet wird.

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Matrizen A_1, A_2, \dots, A_n wobei Matrix A_i nicht quadratisch sein muss.
- **Ziel:** Die Reihenfolge finden, in der die Matrixkettenmultiplikation $A_1 \cdot A_2 \cdot \dots \cdot A_n$ am schnellsten berechnet wird.
- **Output:** Zahl Rechnungen der optimalen Reihenfolge mit Klammern.

Nötige Informationen

- Für $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ genügt es zu wissen, dass A_1 eine 3×1 -Matrix ist, sowie A_2 eine 1×3 -Matrix und A_3 eine 3×1 -Matrix.

Problemstellung

$$A_1 A_2 A_3$$

$$(A_1 A_2) \cdot A_3$$

oder

$$A_1 \cdot (A_2 A_3)$$

- **Input:** Eine Sequenz von Dimensionen p_0, p_1, \dots, p_n wobei Matrix A_i die Dimensionen $p_{i-1} \times p_i$ hat. Also $p_0 \times p_1$ für A_1 .
- **Ziel:** Die Reihenfolge finden, in der die Matrixmultiplikation $A_1 \cdot A_2 \cdot \dots \cdot A_n$ am schnellsten berechnet wird.
- **Output:** Zahl Rechnungen der optimalen Reihenfolge mit Klammern.

Erster Algorithmus

- **Idee:** Teste alle möglichen Reihenfolgen von Matrizen und wählen die optimale Lösung.

Erster Algorithmus

- **Idee:** Teste alle möglichen Reihenfolgen von Matrizen und wählen die optimale Lösung.
- **Problem:** Exponentiell viele Lösungen für längere Matrix-Ketten!

Lösung mit Dynamic Programming

1. Charakterisiere Struktur einer optimalen Lösung.

Lösung mit Dynamic Programming

1. Charakterisiere Struktur einer optimalen Lösung.
2. Definiere rekursiv den Wert einer optimalen Lösung.

Lösung mit Dynamic Programming

1. Charakterisiere Struktur einer optimalen Lösung.
2. Definiere rekursiv den Wert einer optimalen Lösung.
3. Berechne Wert einer idealen Lösung.

Lösung mit Dynamic Programming

1. Charakterisiere Struktur einer optimalen Lösung.
2. Definiere rekursiv den Wert einer optimalen Lösung.
3. Berechne Wert einer idealen Lösung.
4. Konstruiere optimale Lösung aus den Zwischenergebnissen.

Vereinfachungen

- Wir berechnen als Ergebnis nur die optimale Anzahl Rechnungen.

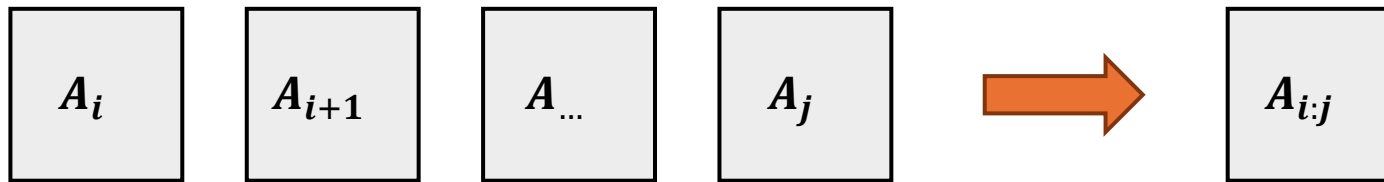
Vereinfachungen

- Wir berechnen als Ergebnis nur die optimale Anzahl Rechnungen.
- Notiz: Es wäre möglich, während der Berechnung auch die Matrix-Kette zu bilden.

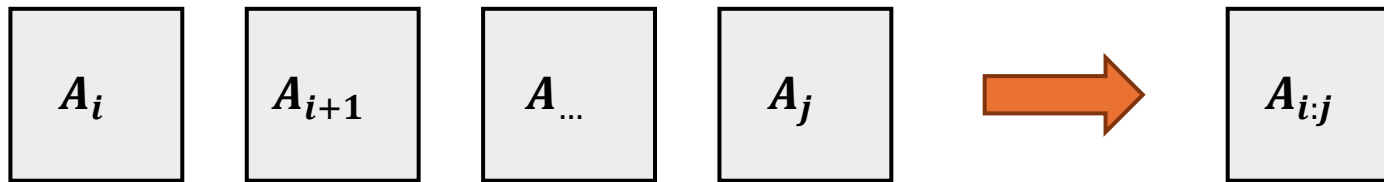
Vereinfachungen

- Wir berechnen als Ergebnis nur die optimale Anzahl Rechnungen.
- Notiz: Es wäre möglich, während der Berechnung auch die Matrix-Kette zu bilden.
 - Wir konzentrieren uns für Einfachheit auf die Rechnungen.

Struktur der optimalen Reihenfolge

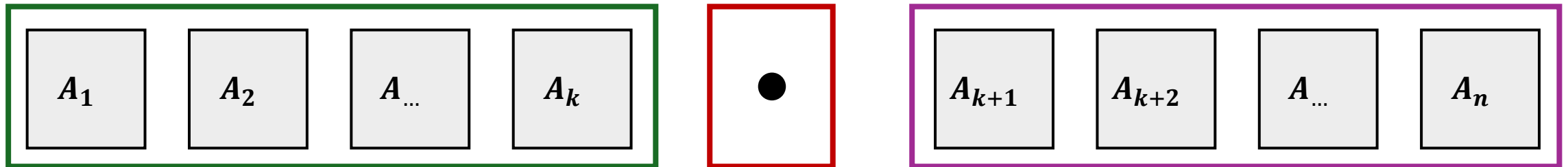


Struktur der optimalen Reihenfolge



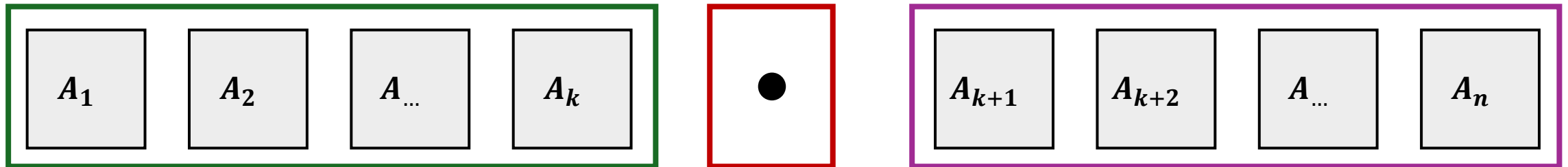
- $A_{i:j}$ bezeichnet die berechnete Matrix-Ketten-Multiplikation $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ für $i < j$

Struktur der optimalen Reihenfolge



- Optimale Berechnung für die ganze Kette kann in die optimale Berechnung für **zwei Ketten** plus die **Kombinationskosten** verwandelt werden.

Struktur der optimalen Reihenfolge



- Optimale Berechnung für die ganze Kette kann in die optimale Berechnung für **zwei Ketten** plus die **Kombinationskosten** verwandelt werden.
- Die Subprobleme sind $A_{1:k}$ und $A_{k+1:n}$

Subprobleme kombinieren

- Bezeichnen wir mit $opt(A_{p:q})$ die optimale Anzahl Berechnungen für eine Matrix-Kette.

Subprobleme kombinieren

- Bezeichnen wir mit $opt(A_{p:q})$ die optimale Anzahl Berechnungen für eine Matrix-Kette.
- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:

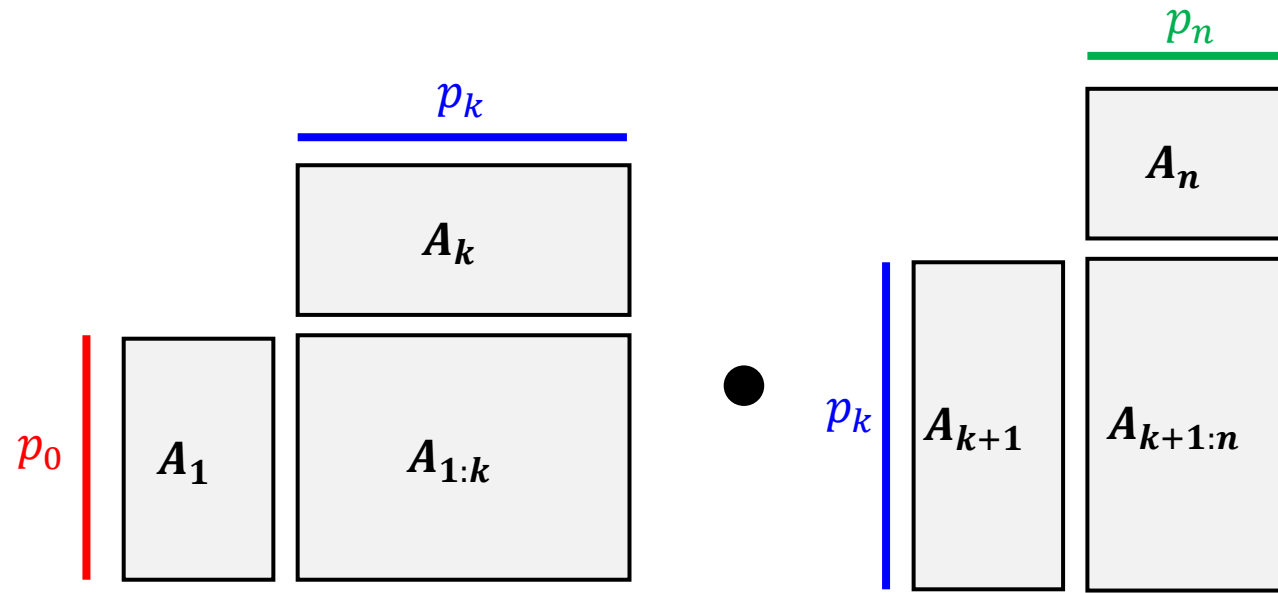
Subprobleme kombinieren

- Bezeichnen wir mit $opt(A_{p:q})$ die optimale Anzahl Berechnungen für eine Matrix-Kette.
- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:
 - $opt(A_{1:n}) = opt(A_{1:k}) + opt(A_{k+1:n}) + p_0 p_k p_n$

Subprobleme kombinieren

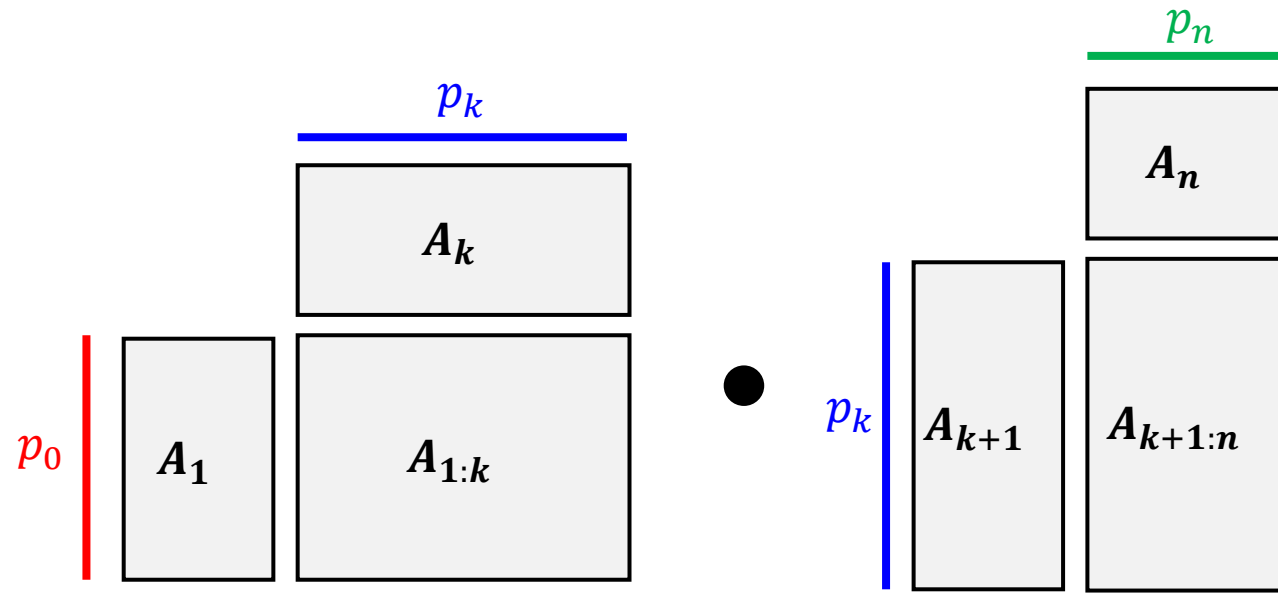
- Bezeichnen wir mit $opt(A_{p:q})$ die optimale Anzahl Berechnungen für eine Matrix-Kette.
- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:
 - $opt(A_{1:n}) = opt(A_{1:k}) + opt(A_{k+1:n}) + p_0 p_k p_n$
 - $p_0 p_k p_n$ ist die Zeit, die es braucht, um das linke und rechte Ergebnis zu kombinieren.

Kombinationskosten



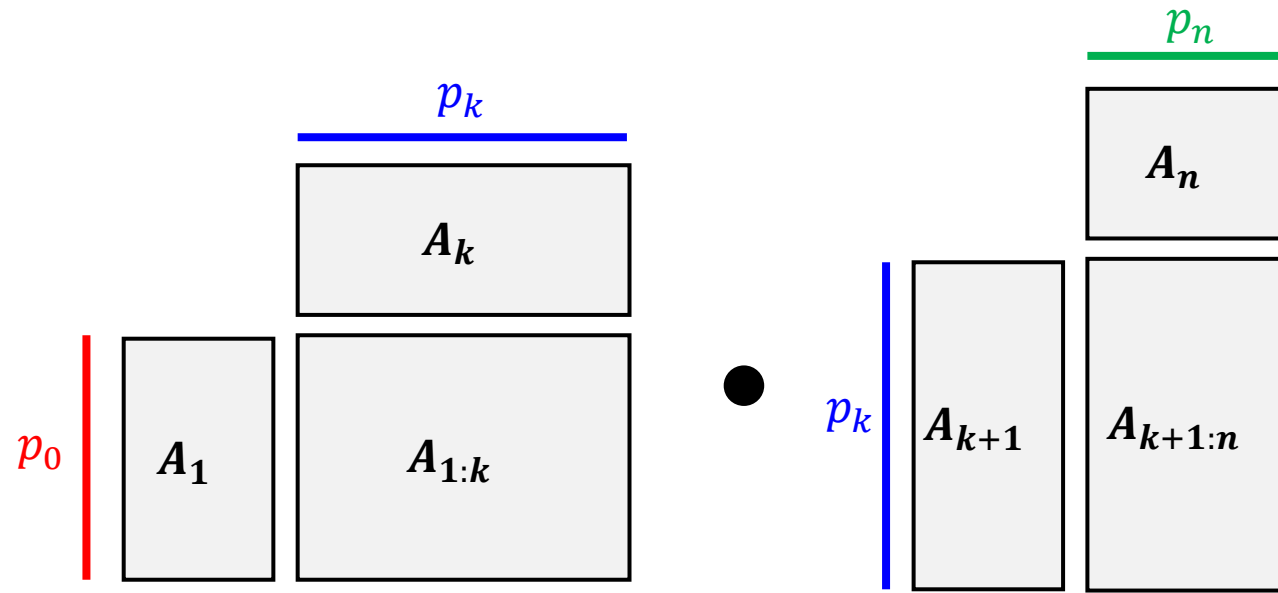
- $A_{1:k}$ hat zwingend Höhe von A_1 und Breite von A_k .

Kombinationskosten



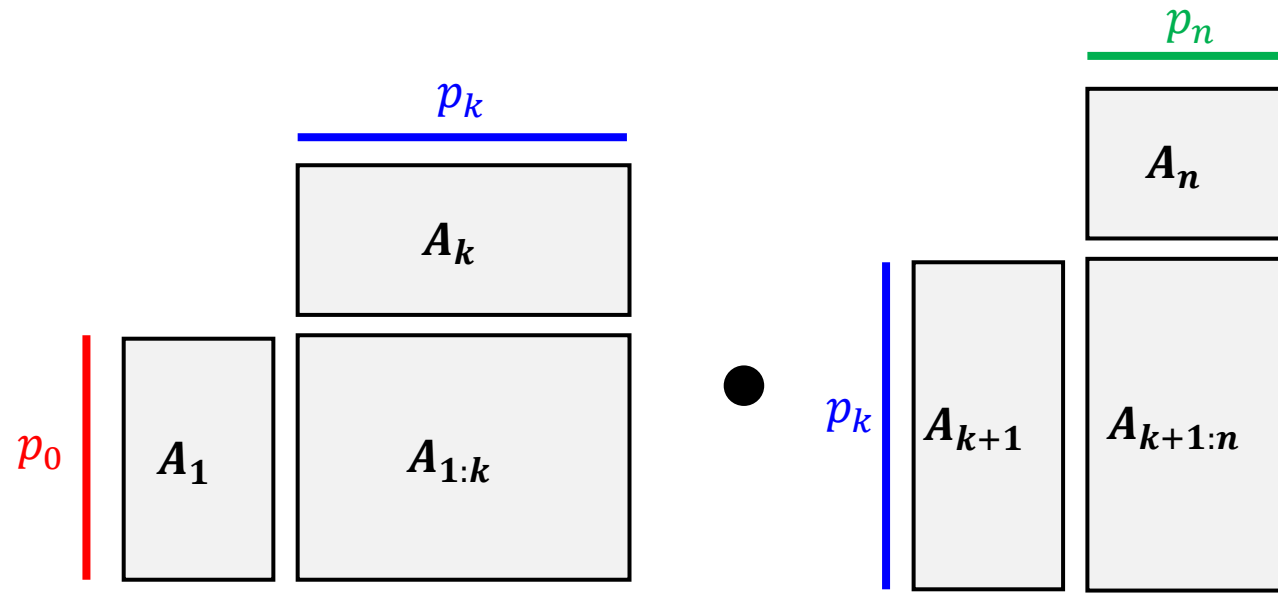
- $A_{1:k}$ hat zwingend **Höhe von A_1** und **Breite von A_k** .
- $A_{k+1:n}$ hat zwingend **Höhe von A_{k+1}** und **Breite von A_n** .

Kombinationskosten



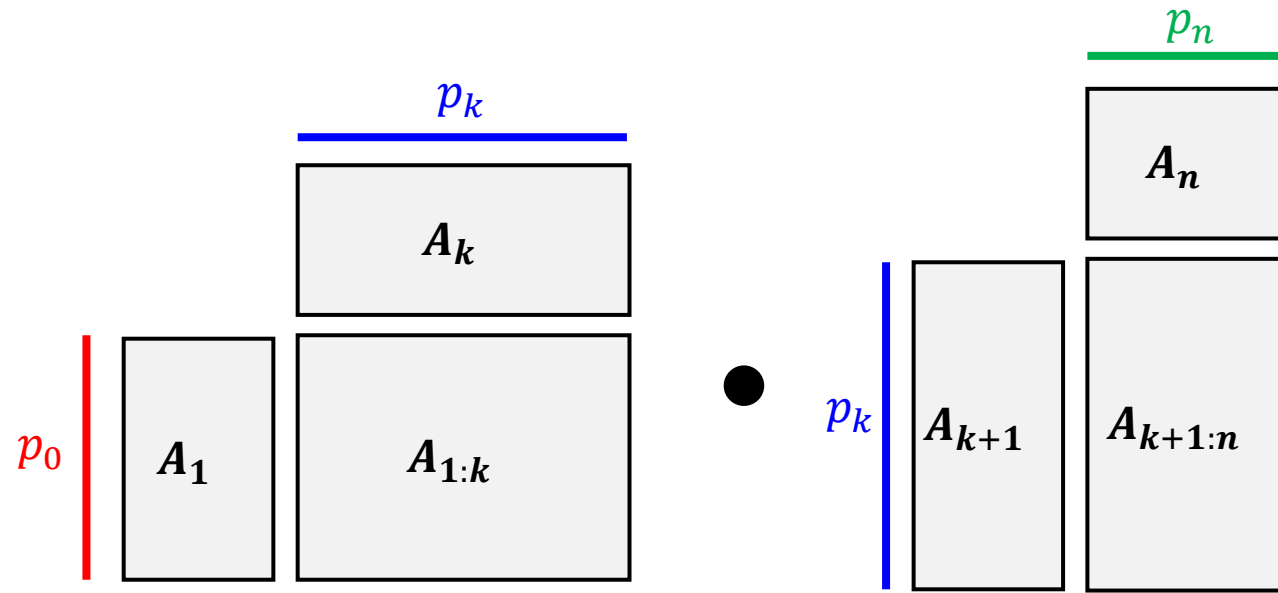
- $A_{1:k}$ hat zwingend **Höhe von A_1** und **Breite von A_k** .
- $A_{k+1:n}$ hat zwingend **Höhe von A_{k+1}** und **Breite von A_n** .
- Die **Höhe von A_{k+1}** ist die **Breite von A_k** .

Kombinationskosten



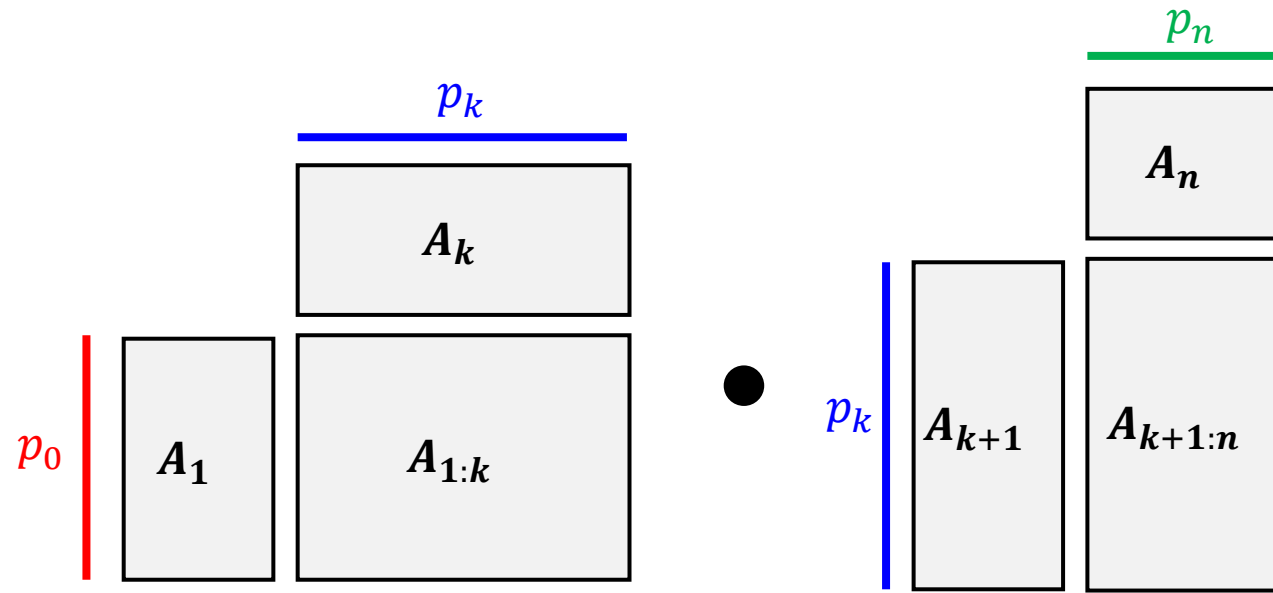
- Die Ergebnis-Matrix hat Dimensionen $p_0 \times p_n$.

Kombinationskosten



- Die Ergebnis-Matrix hat Dimensionen $p_0 \times p_n$.
- Für jedes Element der Ergebnis-Matrix machen wir p_k Rechnungen, um die linke und rechte Matrix zu kombinieren.

Kombinationskosten



- Die Ergebnis-Matrix hat Dimensionen $p_0 \times p_n$.
- Für jedes Element der Ergebnis-Matrix machen wir p_k Rechnungen, um die linke und rechte Matrix zu kombinieren.
- Also total $p_0 p_k p_n$ Rechnungen.

Optimierungsproblem

- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:

Optimierungsproblem

- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:
 - $opt(A_{1:n}) = opt(A_{1:k}) + opt(A_{k+1:n}) + p_0 p_k p_n$

Optimierungsproblem

- Wenn wir die optimalen Anzahlen Berechnungen für $A_{1:k}$ und $A_{k+1:n}$ kennen, ist das Ergebnis für $A_{1:n}$ gegeben als:
 - $opt(A_{1:n}) = opt(A_{1:k}) + opt(A_{k+1:n}) + p_0 p_k p_n$
- Wir wollen die optimalen Subprobleme $A_{1:k}$ und $A_{k+1:n}$ identifizieren, welche die Anzahl Rechnungen minimieren.

Rekursive Lösung

- Wir bezeichnen mit $m[i, j]$ die minimale Anzahl Rechnungen, um die Matrixkette $A_{i:j}$ zu berechnen.

Rekursive Lösung

- Wir bezeichnen mit $m[i, j]$ die minimale Anzahl Rechnungen, um die Matrixkette $A_{i:j}$ zu berechnen.
- Es gilt also, dass:

$$\text{■ } m[i, j] = \begin{cases} 0 & \text{wenn } i = j \\ \text{opt}(A_{i:k}) + \text{opt}(A_{k+1:j}) + p_{i-1}p_kp_j & \text{wenn } i < j \end{cases}$$

Die optimale Lösung berechnen

- Wir besitzen nun eine rekursive Lösung für das Problem. An sich benötigt diese immer noch exponentielle Zeit.

Die optimale Lösung berechnen

- Wir besitzen nun eine rekursive Lösung für das Problem. An sich benötigt diese immer noch exponentielle Zeit.
- Wir können dynamic Programming mit Bottom-Up benutzen, um das Resultat effizienter zu berechnen.

DP-Tabelle

$$m[i, j] = \begin{cases} 0 & \text{wenn } i = j \\ \text{opt}(A_{i:k}) + \text{opt}(A_{k+1:j}) + p_{i-1}p_kp_j & \text{wenn } i < j \end{cases}$$

DP-Tabelle

$$m[i, j] = \begin{cases} 0 & \text{wenn } i = j \\ \min_k \{ \text{opt}(A_{i:k}) + \text{opt}(A_{k+1:j}) + p_{i-1}p_kp_j \} & \text{wenn } i < j \end{cases}$$

- Wir können eine Tabelle m für Zwischenergebnisse benutzen.

DP-Tabelle

$$m[i, j] = \begin{cases} 0 & \text{wenn } i = j \\ \text{opt}(A_{i:k}) + \text{opt}(A_{k+1:j}) + p_{i-1}p_kp_j & \text{wenn } i < j \end{cases}$$

- Wir können eine Tabelle m für Zwischenergebnisse benutzen.
- Wir müssen die Tabelle (wie schon gesehen) in einer schlaunen Reihenfolge durchlaufen!

DP-Tabelle

		i			
		1	2	3	4
j	1	A $= A_1$			
	2	AB	B $= A_2$		
	3	ABC	BC	C $= A_3$	
	4	ABCD	BCD	CD	D $= A_4$

DP-Tabelle

		i			
		1	2	3	4
j	1	A $= A_1$			
	2	AB	B $= A_2$		
	3	ABC	BC	C $= A_3$	
	4	ABCD	BCD	CD	D $= A_4$

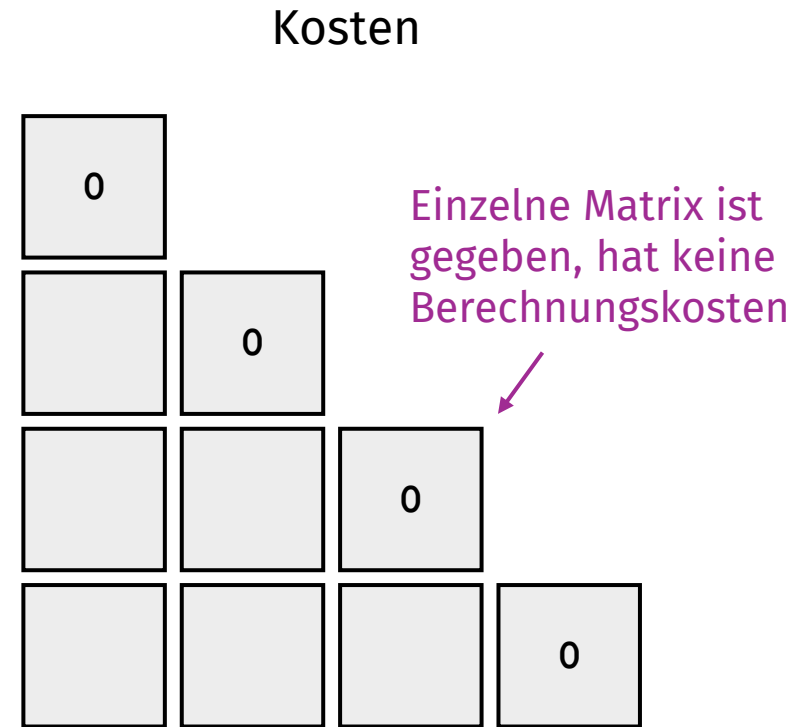
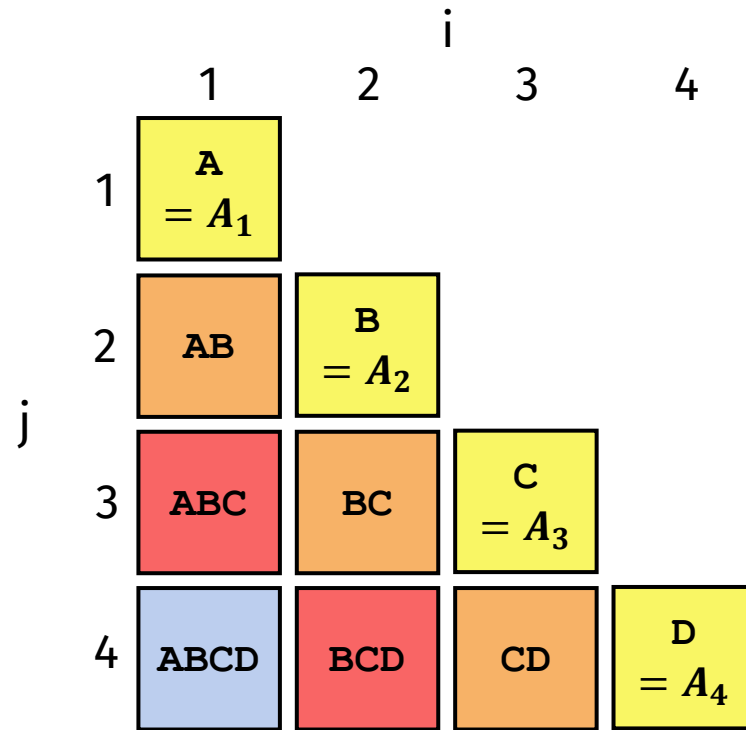
- Da $m[i, j]$ nur für $i < j$ definiert ist, müssen wir nur ein Dreieck berechnen!

DP-Tabelle

		i			
		1	2	3	4
j	1	A $= A_1$			
	2	AB	B $= A_2$		
	3	ABC	BC	C $= A_3$	
	4	ABCD	BCD	CD	D $= A_4$

- Da $m[i, j]$ nur für $i < j$ definiert ist, müssen wir nur ein Dreieck berechnen!
- Hier sind die Matrizen A_1, A_2, A_3, A_4 als A, B, C, D bezeichnet.

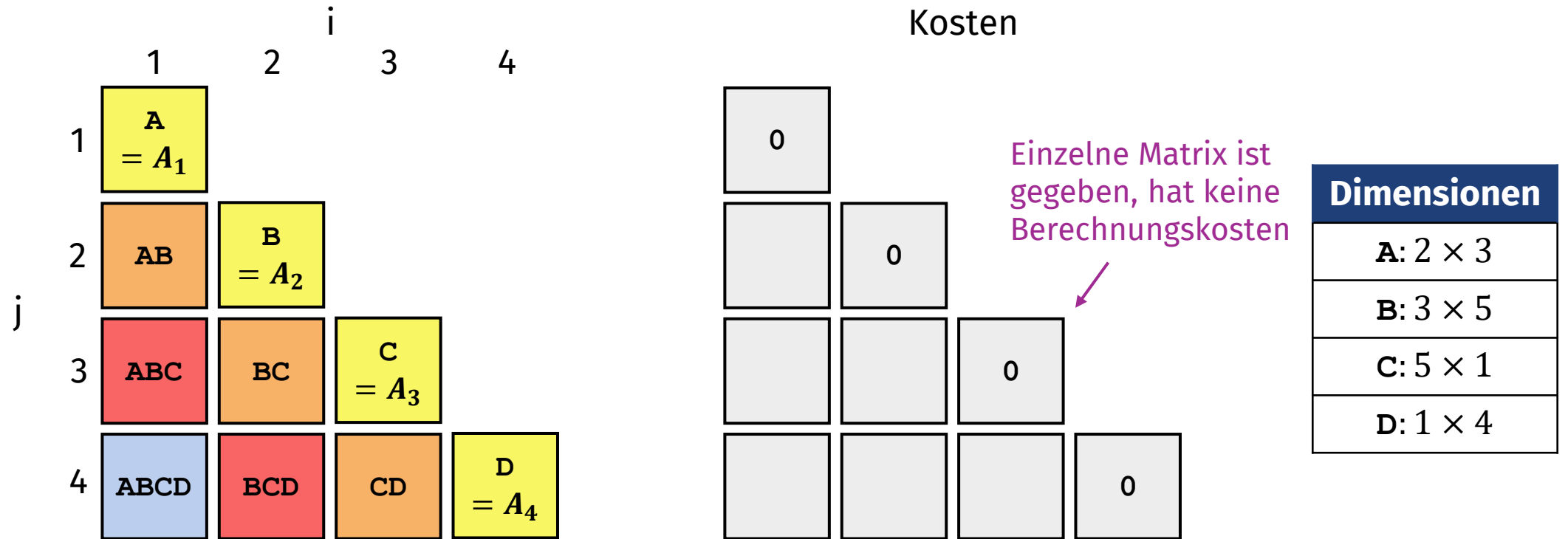
DP-Tabelle



Dimensionen
A: 2 × 3
B: 3 × 5
C: 5 × 1
D: 1 × 4

- Die finale Lösung für unser Problem können wir bei $m[1, n]$ finden

DP-Tabelle



- Die finale Lösung für unser Problem können wir bei $m[1, n]$ finden
- Die Basisfälle sind durch die einzelnen Matrizen mit Berechnungskosten 0 gegeben.

DP-Tabelle

		i			
		1	2	3	4
j	1	A = A_1			
	2	AB	B = A_2		
	3	ABC	BC	C = A_3	
	4	ABCD	BCD	CD	D = A_4

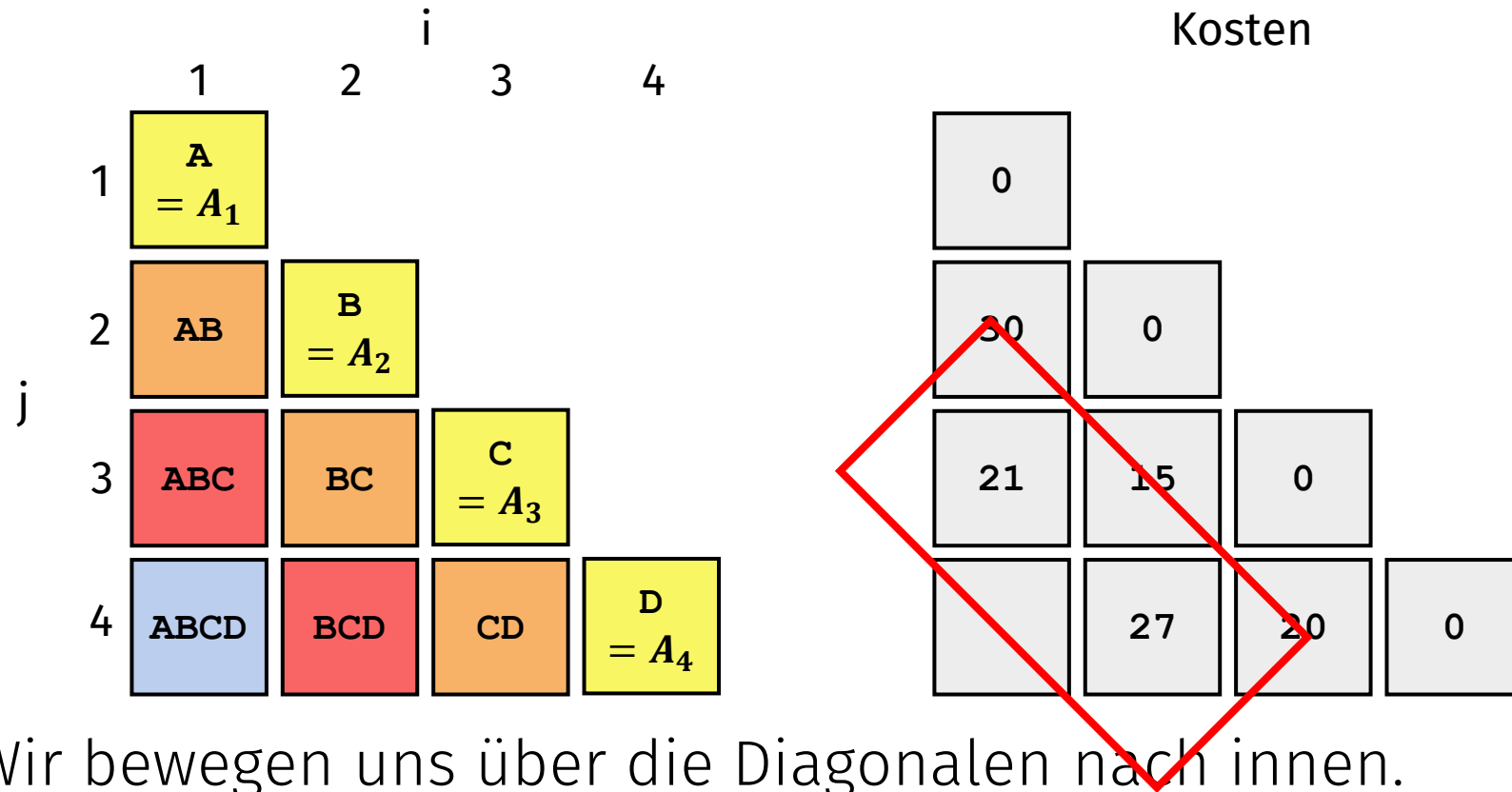
Kosten

	0			
	30	0		
		15	0	
			20	0

Dimensionen
A: 2×3
B: 3×5
C: 5×1
D: 1×4

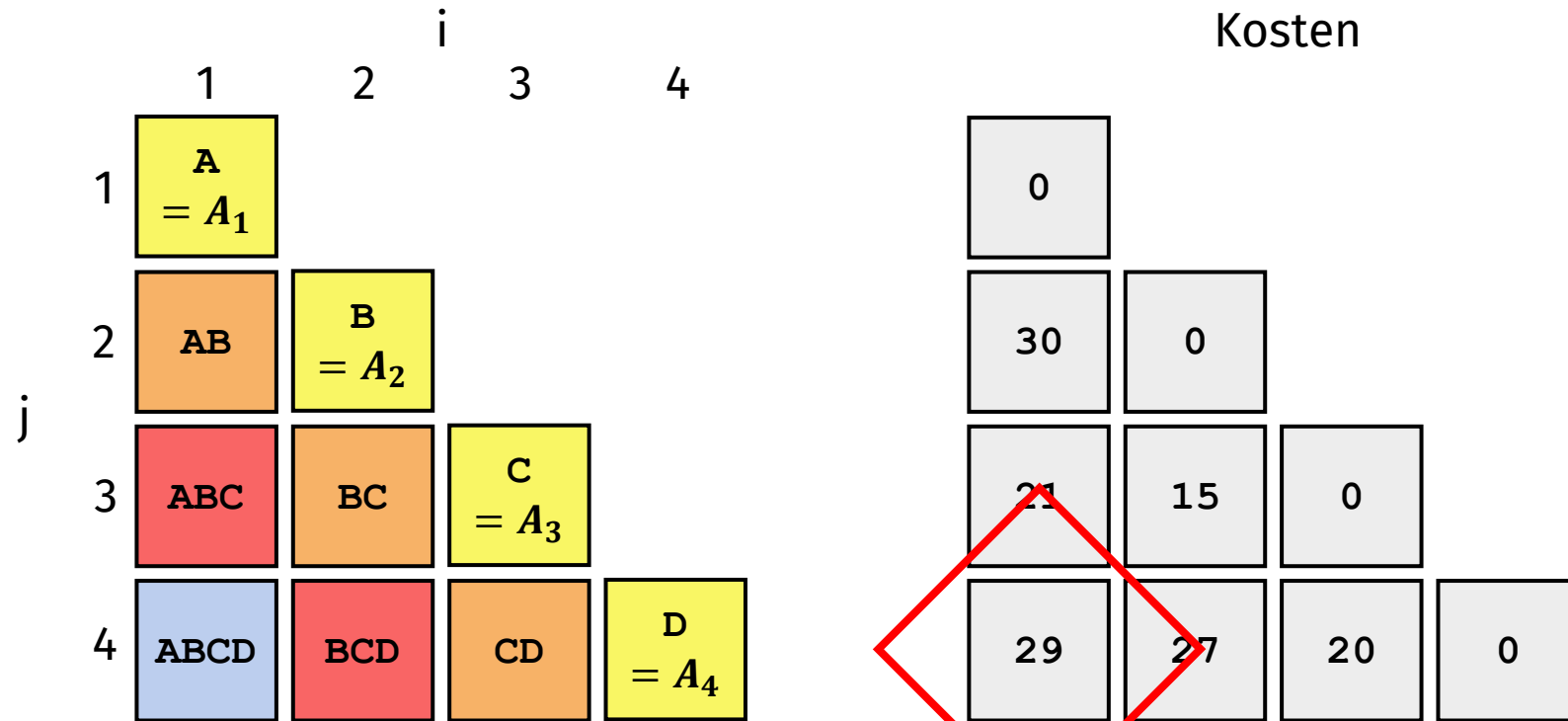
- Wir bewegen uns über die Diagonalen nach innen.
- AB kostet die Kosten von A , die Kosten von B , und Kombinationskosten

DP-Tabelle



- Wir bewegen uns über die Diagonalen nach innen.
- AB kostet die Kosten von A , die Kosten von B , und Kombinationskosten
- ABC können wir aus $A \cdot BC$ oder $AB \cdot C$ berechnen!

DP-Tabelle



Dimensionen
A: 2×3
B: 3×5
C: 5×1
D: 1×4

- Wir bewegen uns über die Diagonalen nach innen.
- AB kostet die Kosten von A , die Kosten von B , und Kombinationskosten.
- ABC können wir aus $A \cdot BC$ oder $AB \cdot C$ berechnen.
- $ABCD \dots$

DP-Tabelle

		i			
		1	2	3	4
j	1	A = A_1			
	2	AB	B = A_2		
	3	ABC	BC	C = A_3	
	4	ABCD	BCD	CD	D = A_4

	Kosten			
		0		
		30	0	
		21	15	0
		29	27	20
				0

Dimensionen
A: 2×3
B: 3×5
C: 5×1
D: 1×4

- Die finale Lösung steht bei $m[1, -1]$

Reihenfolge der Lösungen

$$\underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{A_{1:2}} \underbrace{[1 \quad 1 \quad 1]}_{A_{3:3}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{A_{3:3}} = \begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}_{A_{1:2}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{A_{3:3}}$$

- Um die Berechnungs-Reihenfolge zu finden, müssen wir uns bloss erinnern, wo wir die Subprobleme getrennt haben.

Reihenfolge der Lösungen

$$\underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{A_{1:2}} \underbrace{[1 \quad 1 \quad 1]}_{A_{3:3}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{A_{3:3}} = \begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}_{A_{1:2}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{A_{3:3}}$$

- Um die Berechnungs-Reihenfolge zu finden, müssen wir uns bloss erinnern, wo wir die Subprobleme getrennt haben.
- $k = 2$ sieht z.B. so aus.

Dynamic Programming in Python

```
def best_chain(dims):  
    # Erstelle Strukturen für Resultate  
    n = len(dims) - 1  
    s, m = [[None] * n for _ in range(0, n)], [[None] * n for _ in range(0, n)]  
  
    # Basisfälle füllen  
    for i in range(0, n):  
        m[i][i] = 0  
  
    # Dynamische Berechnung  
    for diag in range(1, n):  
        for i in range(0, n - diag):  
            j = i + diag  
            for k in range(i, j):  
                cost = m[i][k] + m[k+1][j] + dims[i]*dims[k+1]*dims[j+1]  
                if m[i][j] is None or cost < m[i][j]:  
                    m[i][j] = cost  
                    s[i][j] = k  
  
    return m[0][-1]
```

4. Wrap-Up

Wrap-Up: Dynamic Programming

Allgemeines Vorgehen bei Dynamic Programming:

1. Datenstruktur initialisieren

Wrap-Up: Dynamic Programming

Allgemeines Vorgehen bei Dynamic Programming:

1. Datenstruktur initialisieren
2. Basisfall implementieren

Wrap-Up: Dynamic Programming

Allgemeines Vorgehen bei Dynamic Programming:

1. Datenstruktur initialisieren
2. Basisfall implementieren
3. Datenstruktur ausfüllen

Wrap-Up: Dynamic Programming

Allgemeines Vorgehen bei Dynamic Programming:

1. Datenstruktur initialisieren
2. Basisfall implementieren
3. Datenstruktur ausfüllen
4. Resultat zurückgeben

Wrap-Up: Matrixmultiplikation

1. Datenstruktur initialisieren
 - 2D Matrix (Dreieck)

```
m = [[None] * n  
      for _ in range(0, n)],
```

Wrap-Up: Matrixmultiplikation

1. Datenstruktur initialisieren

- 2D Matrix (Dreieck)

```
m = [[None] * n  
      for _ in range(0, n)],
```

2. Basisfall implementieren

- Fall $i = j$

```
for i in range(0, n):  
    m[i][i] = 0
```

Wrap-Up: Matrixmultiplikation

1. Datenstruktur initialisieren

- 2D Matrix (Dreieck)

```
m = [[None] * n  
      for _ in range(0, n)],
```

2. Basisfall implementieren

- Fall $i = j$

```
for i in range(0, n):  
    m[i][i] = 0
```

3. Datenstruktur ausfüllen

- In Diagonalen nach innen

```
for diag in range(1, n):  
    for i in range(0, n - diag):  
        ...
```

Wrap-Up: Matrixmultiplikation

1. Datenstruktur initialisieren

- 2D Matrix (Dreieck)

```
m = [[None] * n  
      for _ in range(0, n)],
```

2. Basisfall implementieren

- Fall $i = j$

```
for i in range(0, n):  
    m[i][i] = 0
```

3. Datenstruktur ausfüllen

- In Diagonalen nach innen

```
for diag in range(1, n):  
    for i in range(0, n - diag):  
        ...
```

4. Resultate zurückgeben

```
return m[0][-1]
```


5. Hausaufgaben

Übung 8: Dynamic Programming I

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

Exercise 8: DP I

- Tribonacci
- Catalan-Zahlen
- Blöcke
- Aufgabenplanung v2.0

Abgabedatum: Montag 28.04.2025, 20:00 MEZ

KEINE HARDCODIERUNG