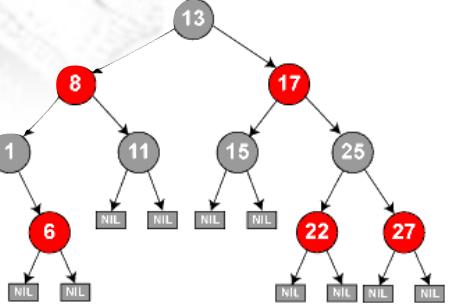
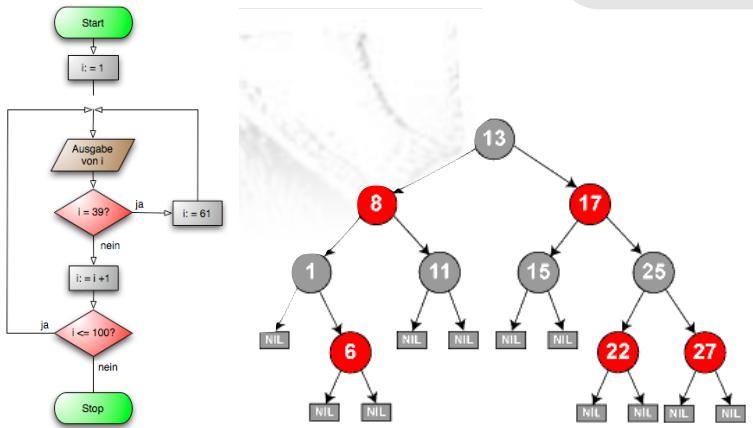




Algorithmen und Datenstrukturen



Version 1.26 vom 09. Mai 2021

III. Entwurfsmethoden

3. Entwurfsmethoden

- 3.1. Teile und Herrsche
- 3.2. Backtracking
- 3.3. Memorization
- 3.4. Dynamische Programmierung
- 3.5. Greedy-Algorithmen

Einleitung

- **Man kann keine starren Regeln zum Entwurf eines Algorithmus angeben**
Kein Algorithmus zum erstellen eines Algorithmus!
- **Für bestimmte Problemstellungen haben sich jedoch einige Entwurfsprinzipien bewährt.**
- **Vier sollen an dieser Stelle kurz vorgestellt werden:**
 - Teile und Herrsche
 - Backtracking
 - Memoization
 - Dynamische Programmierung
 - Greedy-Algorithmen

III. Entwurfsmethoden

3. Entwurfsmethoden

3.1. Teile und Herrsche

- 3.2. Backtracking
- 3.3. Memoization
- 3.4. Dynamische Programmierung
- 3.5. Greedy-Algorithmen

Teile und Herrsche

- **Auch bekannt als *divide and conquer* bzw. *divide et impera***

Kann fast auf jedes Problem angewendet werden, dass sich in kleinere, unabhängige Teilprobleme zerlegen lässt

Teile und Herrsche

- **Auch bekannt als *divide and conquer* bzw. *divide et impera***

Kann fast auf jedes Problem angewendet werden, dass sich in kleinere, unabhängige Teilprobleme zerlegen lässt

- **Drei Schritte:**

- **Divide:** Zerlege das Problem in (kleinere, unabhängige) Teilprobleme bis es elementar bearbeitbar wird ⇒ Conquer-Schritt
- **Conquer:** Löse elementare Probleme
- **Merge:** Setze Lösung der Teilprobleme zur Gesamtlösung zusammen

Teile und Herrsche

- **Auch bekannt als *divide and conquer* bzw. *divide et impera***

Kann fast auf jedes Problem angewendet werden, dass sich in kleinere, unabhängige Teilprobleme zerlegen lässt

- **Drei Schritte:**

- **Divide:** Zerlege das Problem in (kleinere, unabhängige) Teilprobleme bis es elementar bearbeitbar wird ⇒ Conquer-Schritt
- **Conquer:** Löse elementare Probleme
- **Merge:** Setze Lösung der Teilprobleme zur Gesamtlösung zusammen

- **Oftmals rekursive Implementierung**

- z.B. Baumdurchlauf (gut), Fibonacci-Zahlen (schlecht)
- Leistungsanalyse über Rekursionsgleichungen
Abschätzung mit Master-Theorem

Teile und Herrsche

- **Auch bekannt als *divide and conquer* bzw. *divide et impera***

Kann fast auf jedes Problem angewendet werden, dass sich in kleinere, unabhängige Teilprobleme zerlegen lässt

- **Drei Schritte:**

- **Divide:** Zerlege das Problem in (kleinere, unabhängige) Teilprobleme bis es elementar bearbeitbar wird ⇒ Conquer-Schritt
- **Conquer:** Löse elementare Probleme
- **Merge:** Setze Lösung der Teilprobleme zur Gesamtlösung zusammen

- **Oftmals rekursive Implementierung**

- z.B. Baumdurchlauf (gut), Fibonacci-Zahlen (schlecht)
- Leistungsanalyse über Rekursionsgleichungen
Abschätzung mit Master-Theorem

- (trivial) **parallelisierbar**

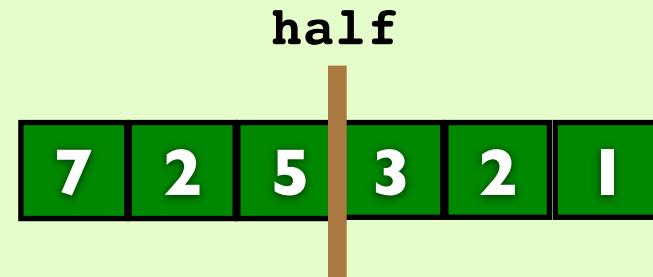
Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

7	2	5	3	2	1
---	---	---	---	---	---

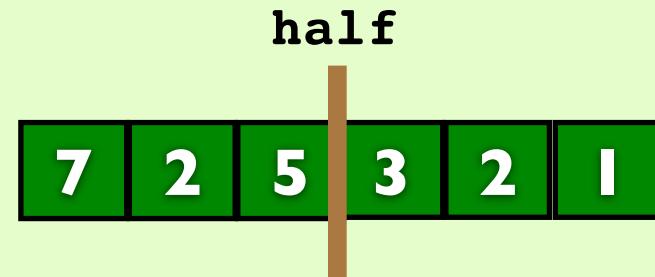
Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



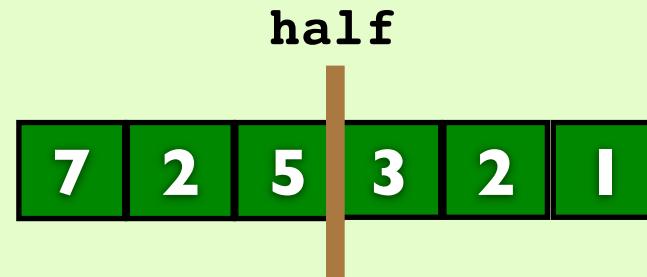
Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



half



7

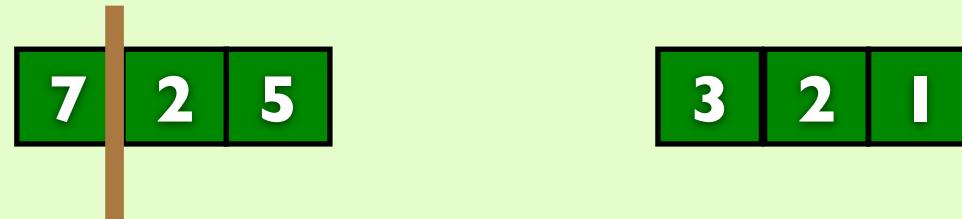
Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



half

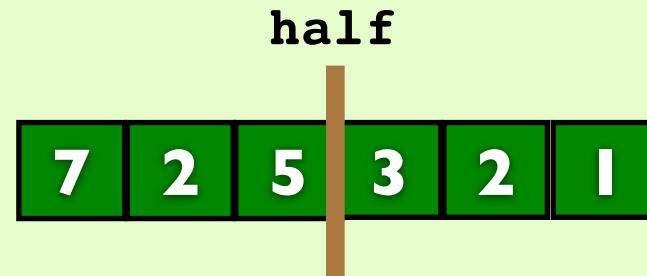


7

2 5

Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



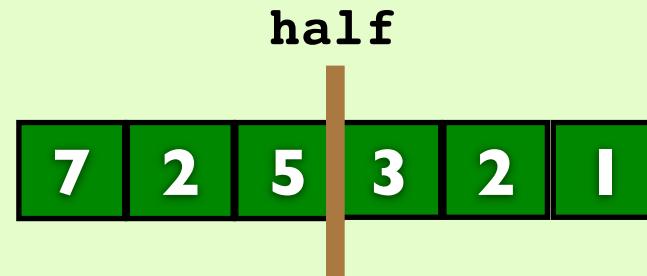
(7,7)

7

2 5

Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



(7,7)

7

(5,2)

2 5

Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



(7,2) half



(7,7)



(5,2)



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

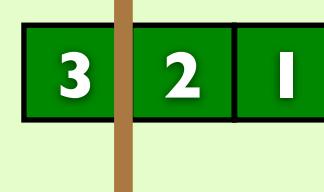
half



(7,2) half



half



(7,7)



(5,2)



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



(7,2) half



half



(7,7)



(5,2)



3



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



(7,2) half



half



(7,7)



(5,2)



(3,3)



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



(7,2) half



half



(7,7)



(5,2)



(3,3)



(2,1)



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I

half



(7,2) half



(3,1) half



(7,7)



(5,2)



(3,3)

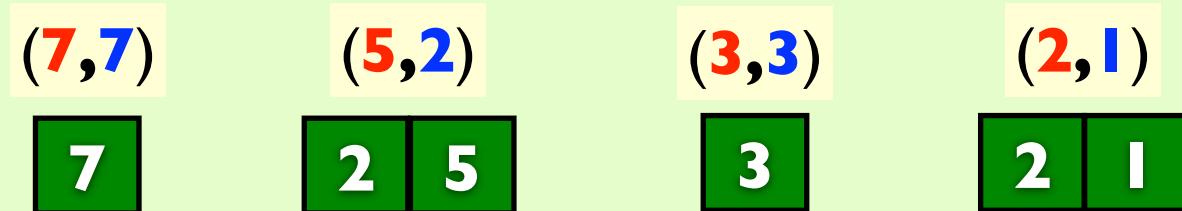
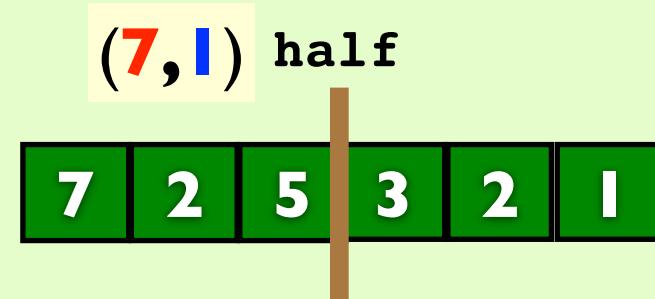


(2,1)



Beispiel: minmax

B Simultane Bestimmung von Maximum und Minimum I



Beispiel: minmax in C++

B Simultane Bestimmung von Maximum und Minimum

Beispiel: minmax in C++

B Simultane Bestimmung von Maximum und Minimum

```
pair<int,int> minmax(const vector<int>& v, int start, int end) {  
    if (end-start<=2) { // Basisfall: nur zwei Elemente (conquer)  
        return pair<int,int>(  
            max(v[start],v[end-1]),  
            min(v[start],v[end-1]) );  
    }  
    else { // Teile Vektor in zwei Hälften auf (divide) ...  
        int half = start + ((end-start) >> 1);  
        // bestimme max und min der Hälften  
        pair<int,int> mm1 = minmax(v, half, end);  
        pair<int,int> mm2 = minmax(v, start, half);  
        // Setzte Gesamtergebnis zusammen (merge)  
        return pair<int,int>(  
            max(mm1.first,mm2.first),  
            min(mm1.second,mm2.second) );  
    }  
}
```

Beispiel: minmax - Laufzeitanalyse

B Simultane Bestimmung von Maximum und Minimum

Laufzeitanalyse: Sei $n=2^N$ - für die Zahl der Vergleiche $T(n)$ ergibt sich:

Beispiel: minmax - Laufzeitanalyse

B Simultane Bestimmung von Maximum und Minimum

Laufzeitanalyse: Sei $n=2^N$ - für die Zahl der Vergleiche $T(n)$ ergibt sich:

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Beispiel: minmax - Laufzeitanalyse

B Simultane Bestimmung von Maximum und Minimum

Laufzeitanalyse: Sei $n=2^N$ - für die Zahl der Vergleiche $T(n)$ ergibt sich:

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Erläuterung:

$n = 2$: ein Vergleich für min/max zweier Zahlen

$n > 2$: zwei Aufrufe von minmax mit $\frac{n}{2}$ -elementigen Mengen und zwei zusätzliche Vergleiche (Maxima und Minima der Teilergebnisse)

Beispiel: minmax - Laufzeitanalyse

B Simultane Bestimmung von Maximum und Minimum

Laufzeitanalyse: Sei $n=2^N$ - für die Zahl der Vergleiche $T(n)$ ergibt sich:

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Erläuterung:

$n = 2$: ein Vergleich für min/max zweier Zahlen

$n > 2$: zwei Aufrufe von minmax mit $\frac{n}{2}$ -elementigen Mengen und zwei zusätzliche Vergleiche (Maxima und Minima der Teilergebnisse)

Behauptung (ohne Beweis):

$$T(n) = \frac{3}{2}n - 2$$

Zum Vergleich: Algorithmus, der Maximum und Minimum **separat** berechnet benötigt

2 · (n-1) Vergleiche

III. Entwurfsmethoden

3. Entwurfsmethoden

3.1. Teile und Herrsche

3.2. Backtracking

3.3. Memoization

3.4. Dynamische Programmierung

3.5. Greedy-Algorithmen

Backtracking: Einleitung

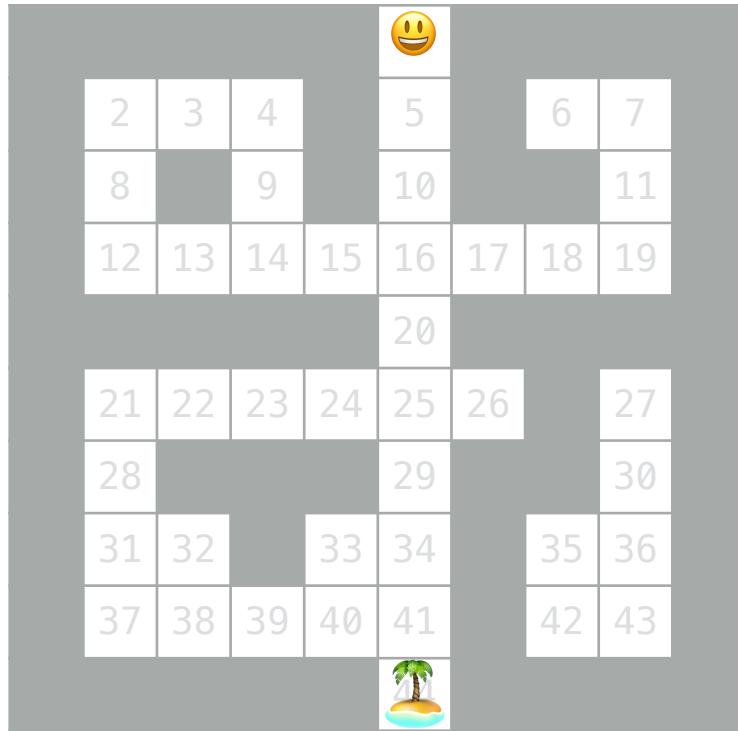
- **Situation:** Mehrere Alternativen sind in einem bestimmten Schritt eines Algorithmus möglich
 - Entscheidungsbäume (siehe NEA)
 - Beispiel: vier Richtungen bei Wegsuche im Labyrinth

Backtracking: Einleitung

- **Situation:** Mehrere Alternativen sind in einem bestimmten Schritt eines Algorithmus möglich
 - Entscheidungsbäume (siehe NEA)
 - Beispiel: vier Richtungen bei Wegsuche im Labyrinth
- **Lösung mit Backtracking:**
 - Tiefensuche im Entscheidungsbaum
 - Wähle eine der Alternativen und verfolge Lösungsberechnung weiter
 - Findet man so eine Lösung des Problems, ist man fertig
 - Ansonsten geht man einen Schritt zurück und verfolgt eine bisher noch nicht untersuchte Alternative
 - Wurden alle Alternativen erfolglos probiert, geht man wieder einen Schritt zurück
 - Kommt man so zum Ausgangspunkt zurück und hat keine weitere Alternative zur Wahl, ist das Problem nicht lösbar

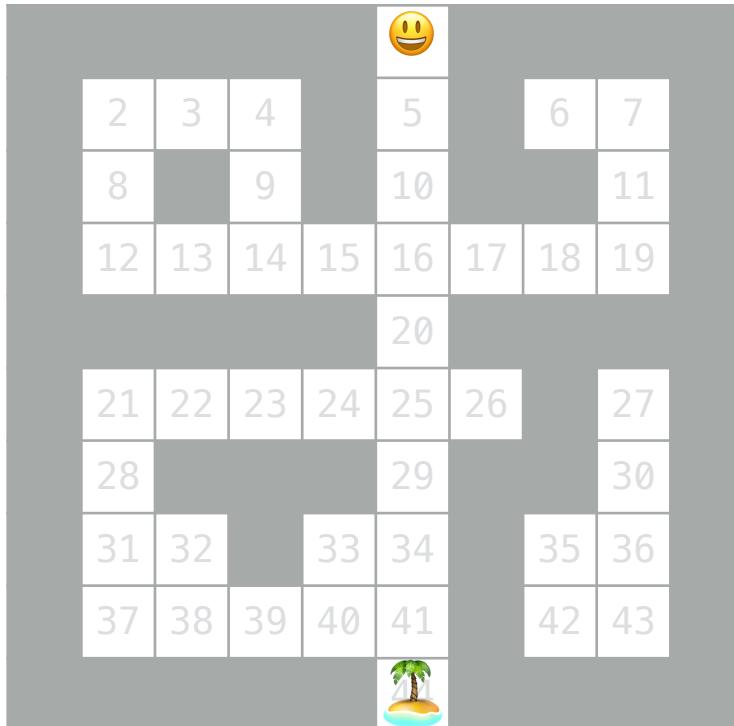
Beispiel: Wegsuche im Labyrinth

- **Aufgabe:** gibt es einen Weg zur Palme?
- **Lösungsversuch:** **Tiefensuche** im „Entscheidungsbaum“



Beispiel: Wegsuche im Labyrinth

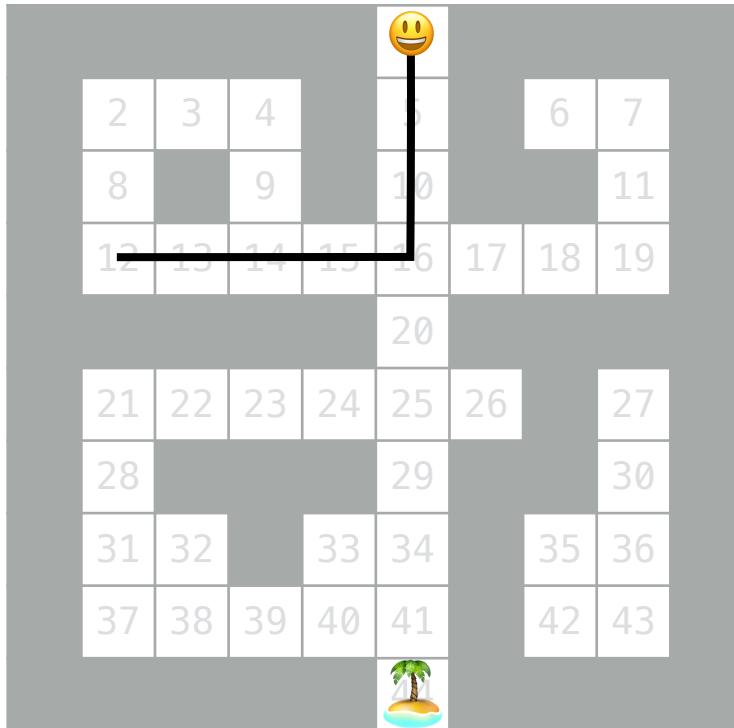
- **Aufgabe:** gibt es einen Weg zur Palme?
- **Lösungsversuch:** **Tiefensuche** im „Entscheidungsbaum“



```
solve(pos): boolean
// Position nicht Wand und
// auf Spielfeld?
if ! pos.legal
    return false
else if pos == target
    return true
else
    return
        solve(pos.left) or
        solve(pos.down) or
        solve(pos.up) or
        solve(pos.right)
```

Beispiel: Wegsuche im Labyrinth

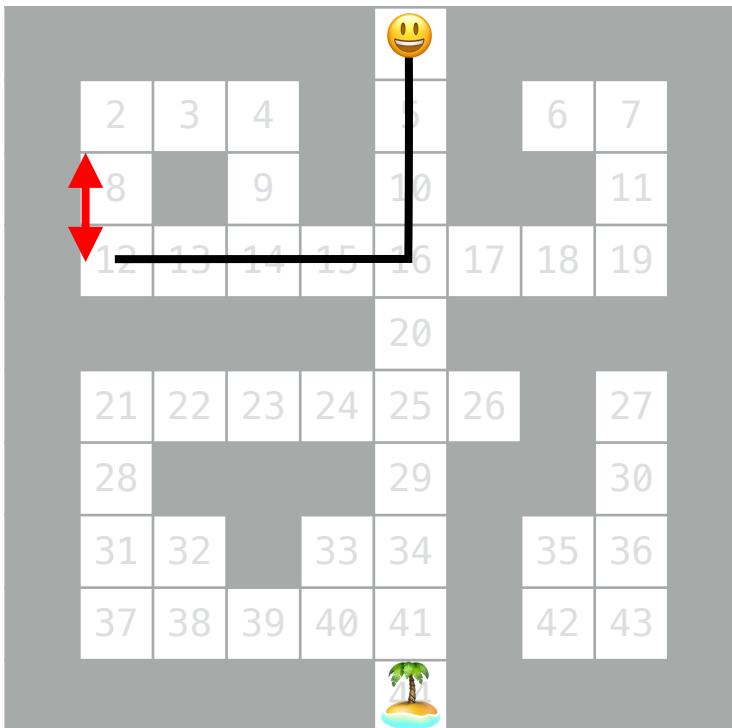
- **Aufgabe:** gibt es einen Weg zur Palme?
- **Lösungsversuch:** **Tiefensuche** im „Entscheidungsbaum“



```
solve(pos): boolean
// Position nicht Wand und
// auf Spielfeld?
if ! pos.legal
    return false
else if pos == target
    return true
else
    return
        solve(pos.left) or
        solve(pos.down) or
        solve(pos.up) or
        solve(pos.right)
```

Beispiel: Wegsuche im Labyrinth

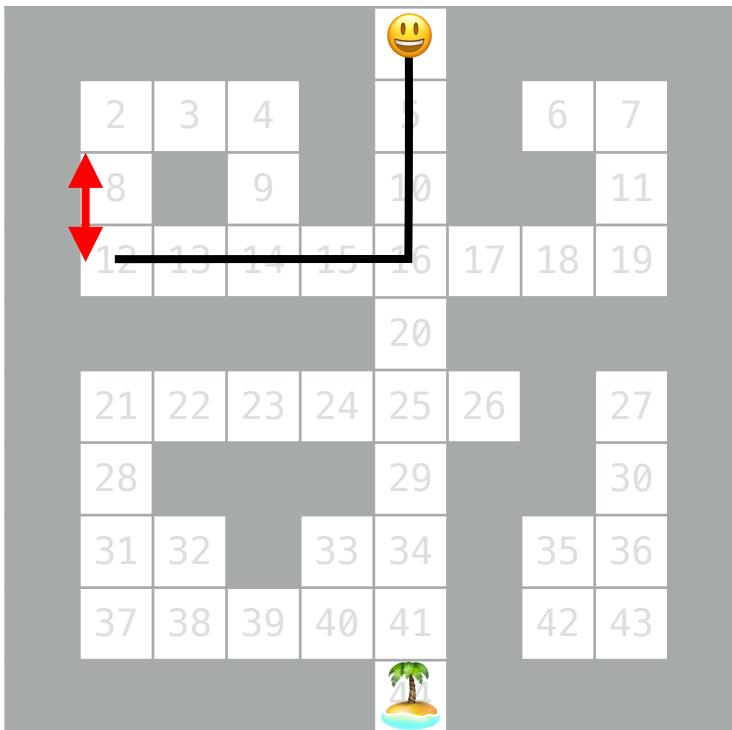
- **Aufgabe:** gibt es einen Weg zur Palme?
- **Lösungsversuch:** **Tiefensuche** im „Entscheidungsbaum“



```
solve(pos): boolean
// Position nicht Wand und
// auf Spielfeld?
if ! pos.legal
    return false
else if pos == target
    return true
else
    return
        solve(pos.left) or
        solve(pos.down) or
        solve(pos.up) or
        solve(pos.right)
```

Beispiel: Wegsuche im Labyrinth

- **Aufgabe:** gibt es einen Weg zur Palme?
- **Lösungsversuch:** **Tiefensuche** im „Entscheidungsbaum“

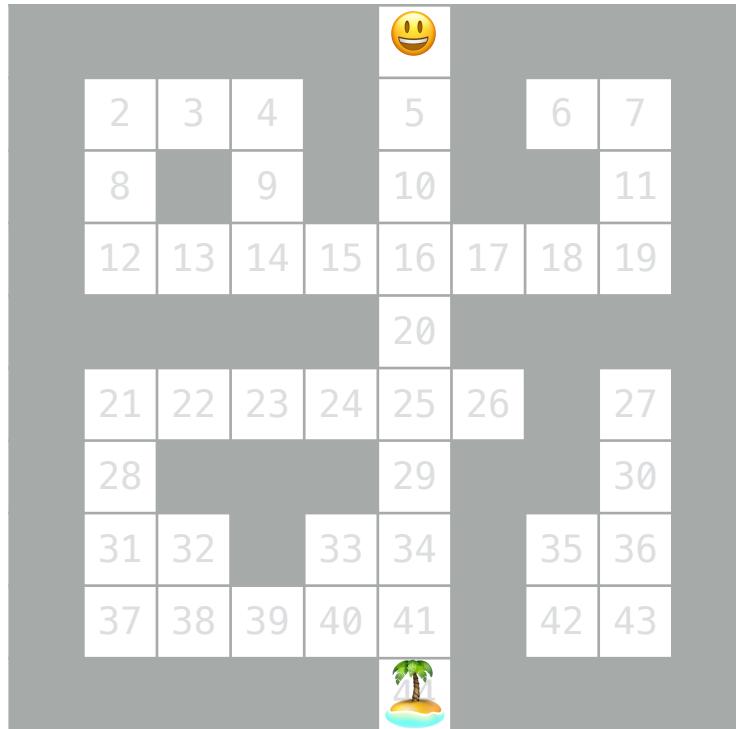


Bei Erreichen von Feld 12
beginnt eine **Endlosschleife**:
 $8 \leftrightarrow 12$

```
solve(pos): boolean
// Position nicht Wand und
// auf Spielfeld?
if ! pos.legal
    return false
else if pos == target
    return true
else
    return
        solve(pos.left) or
        solve(pos.down) or
        solve(pos.up) or
        solve(pos.right)
```

Motivation: Wegsuche im Labyrinth

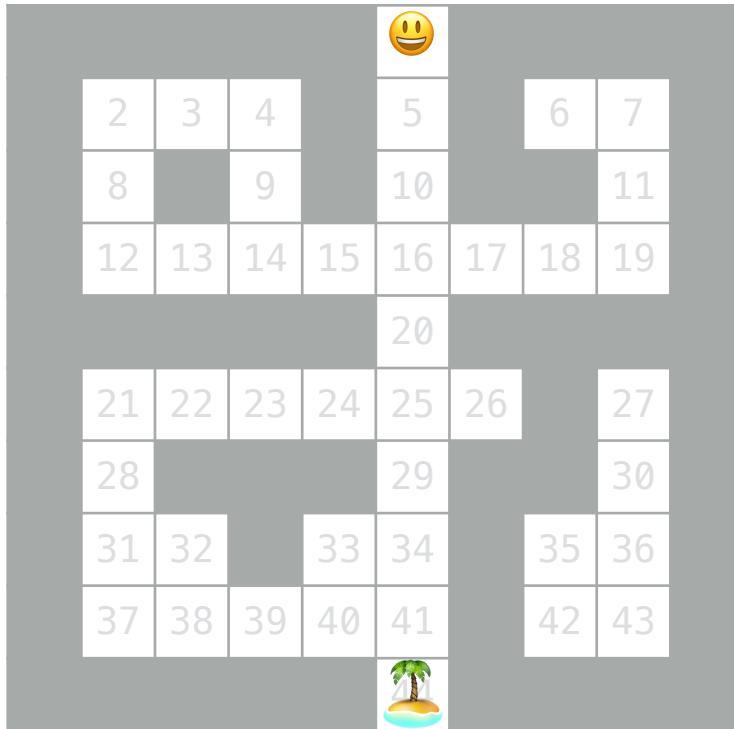
- **Tiefensuche im „Entscheidungsbaum“ mit „Brotkrumen“, um Zyklen zu vermeiden**



```
solve(pos): boolean
    // Position nicht Wand, auf Spielfeld und
    // ohne Brotkrumen?
    if ! pos.legal || breadCrumb[pos]
        return false
    else if pos == target
        return true
    else {
        breadCrumb[pos] = yes
        result =
            solve(pos.left) or
            solve(pos.down) or
            solve(pos.up) or
            solve(pos.right)
        breadCrumb[pos] = no
        return result
    }
```

Motivation: Wegsuche im Labyrinth

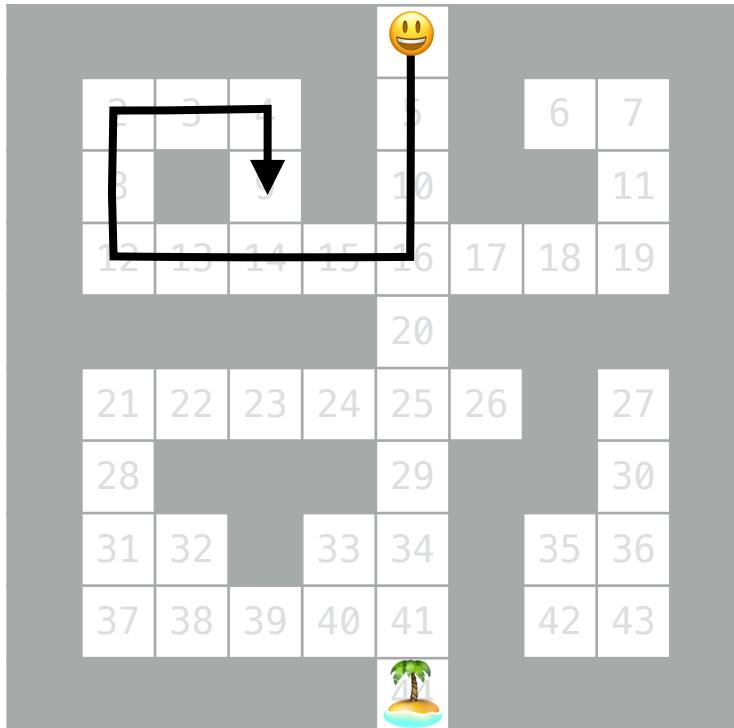
- **Tiefensuche im „Entscheidungsbaum“ mit „Brotkrumen“, um Zyklen zu vermeiden**



```
solve(pos): boolean
  // Position nicht Wand, auf Spielfeld und
  // ohne Brotkrumen?
  if ! pos.legal || breadCrumb[pos]
    return false
  else if pos == target
    return true
  else {
    breadCrumb[pos] = yes
    result =
      solve(pos.left) or
      solve(pos.down) or
      solve(pos.up) or
      solve(pos.right)
    breadCrumb[pos] = no
    return result
  }
```

Motivation: Wegsuche im Labyrinth

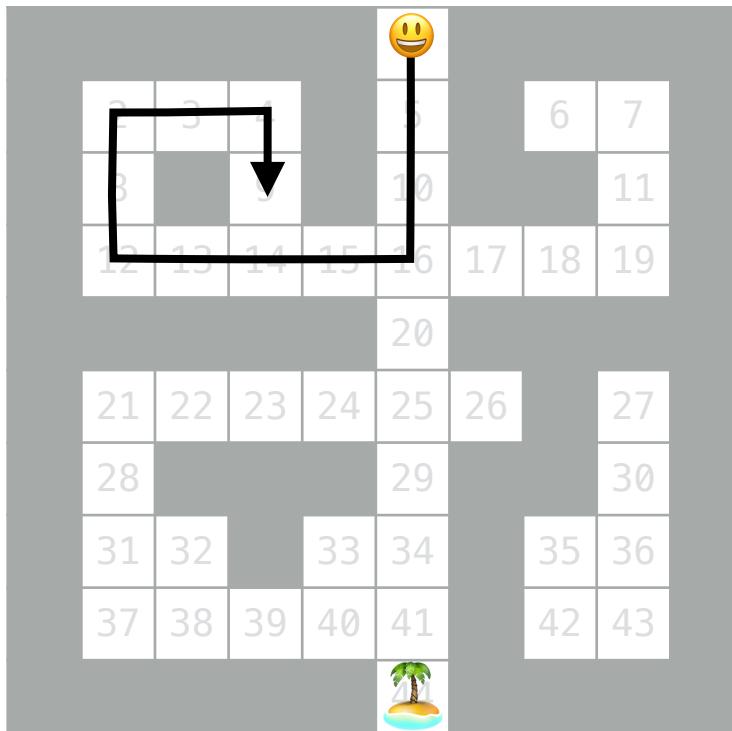
- **Tiefensuche im „Entscheidungsbaum“ mit „Brotkrumen“, um Zyklen zu vermeiden**



```
solve(pos): boolean
// Position nicht Wand, auf Spielfeld und
// ohne Brotkrumen?
if ! pos.legal || breadCrumb[pos]
    return false
else if pos == target
    return true
else {
    breadCrumb[pos] = yes
    result =
        solve(pos.left) or
        solve(pos.down) or
        solve(pos.up) or
        solve(pos.right)
    breadCrumb[pos] = no
    return result
}
```

Motivation: Wegsuche im Labyrinth

- **Tiefensuche im „Entscheidungsbaum“ mit „Brotkrumen“, um Zyklen zu vermeiden**



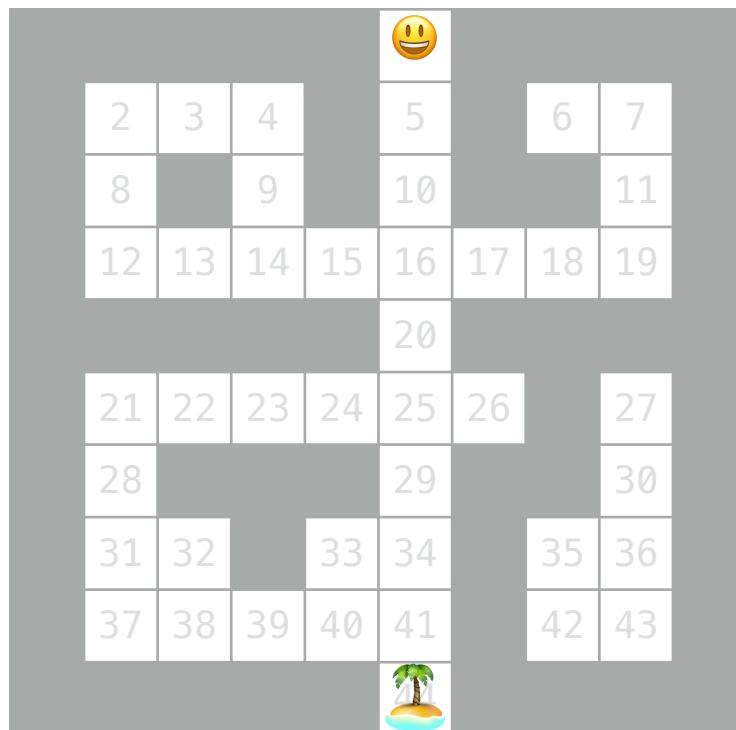
Keine Endlosschleife
keine „im Kreis laufen“

```
solve(pos): boolean
    // Position nicht Wand, auf Spielfeld und
    // ohne Brotkrumen?
    if ! pos.legal || breadCrumb[pos]
        return false
    else if pos == target
        return true
    else {
        breadCrumb[pos] = yes
        result =
            solve(pos.left) or
            solve(pos.down) or
            solve(pos.up) or
            solve(pos.right)
        breadCrumb[pos] = no
        return result
    }
```

Beispiel: Kürzeste Wege im Labyrinth

- **Aufgabe:** Wie lang ist ein kürzester Weg?

Es kann mehrere kürzeste Pfade geben

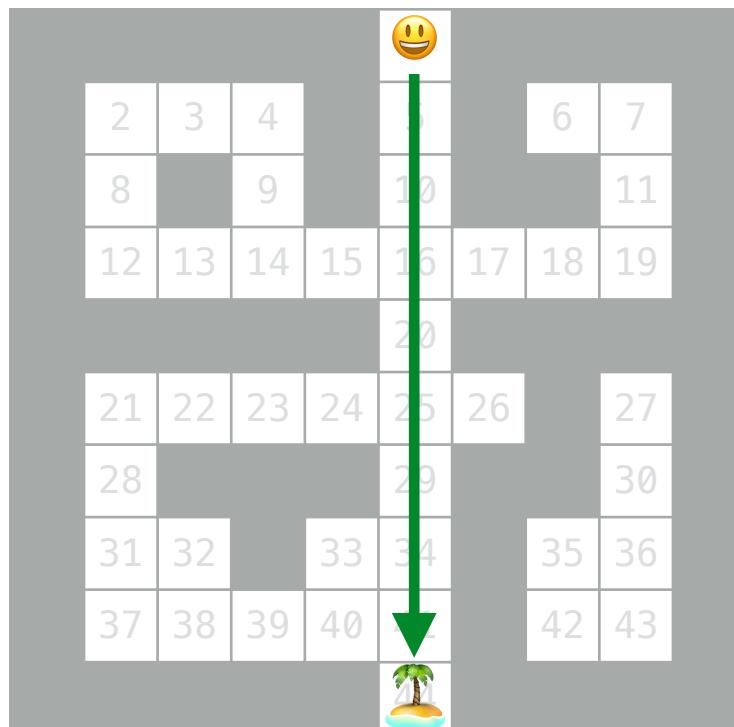


```
solve(pos, current): int?  
    if ! pos.legal || breadCrumb[pos]  
        return null  
    else if pos == target  
        return current  
    else  
        min = null;  
        breadCrumb[pos] = yes  
        for d in [down, left, up, right] {  
            result = solve(pos.d, current+1)  
            if (result) {  
                If !min || result < min  
                    min = result  
            }  
        }  
        breadCrumb[pos] = no  
    return min
```

Beispiel: Kürzeste Wege im Labyrinth

- **Aufgabe:** Wie lang ist ein kürzester Weg?

Es kann mehrere kürzeste Pfade geben



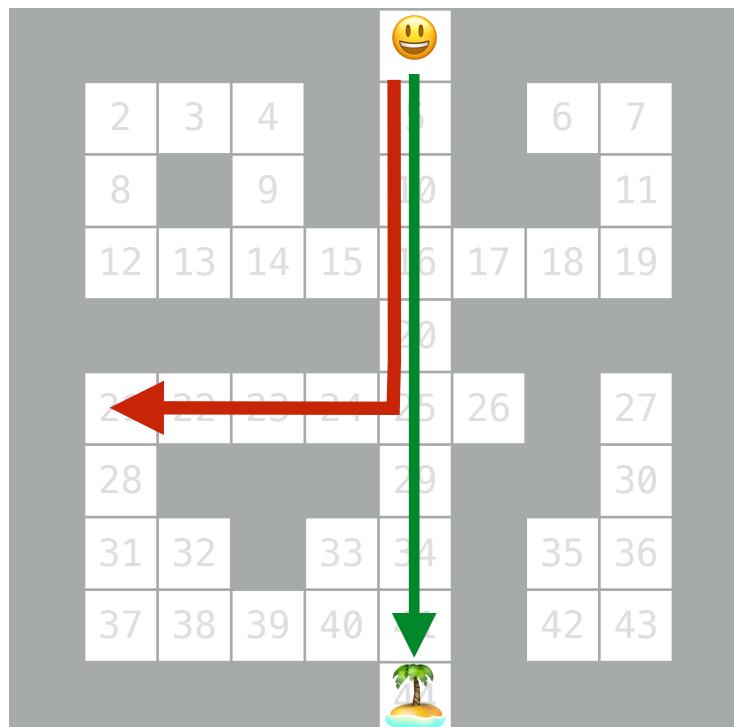
Kürzester Pfad: 9 Schritte

```
solve(pos, current): int?  
    if ! pos.legal || breadCrumb[pos]  
        return null  
    else if pos == target  
        return current  
    else  
        min = null;  
        breadCrumb[pos] = yes  
        for d in [down, left, up, right] {  
            result = solve(pos.d, current+1)  
            if (result) {  
                If !min || result < min  
                    min = result  
            }  
        }  
        breadCrumb[pos] = no  
    return min
```

Beispiel: Kürzeste Wege im Labyrinth

- **Aufgabe:** Wie lang ist ein kürzester Weg?

Es kann mehrere kürzeste Pfade geben



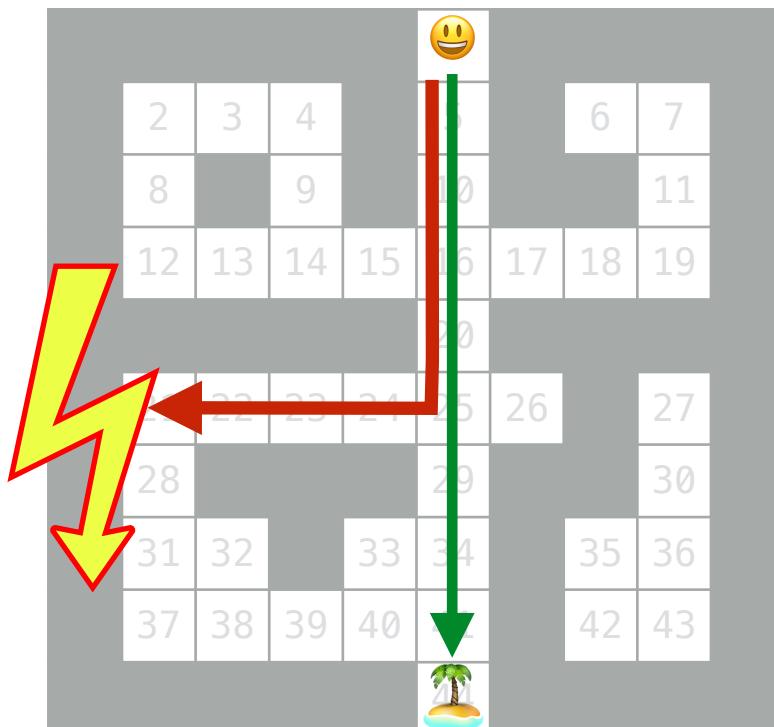
Kürzester Pfad: 9 Schritte

```
solve(pos, current): int?  
    if ! pos.legal || breadCrumb[pos]  
        return null  
    else if pos == target  
        return current  
    else  
        min = null;  
        breadCrumb[pos] = yes  
        for d in [down, left, up, right] {  
            result = solve(pos.d, current+1)  
            if (result) {  
                If !min || result < min  
                    min = result  
            }  
        }  
        breadCrumb[pos] = no  
    return min
```

Beispiel: Kürzeste Wege im Labyrinth

- **Aufgabe:** Wie lang ist ein kürzester Weg?

Es kann mehrere kürzeste Pfade geben



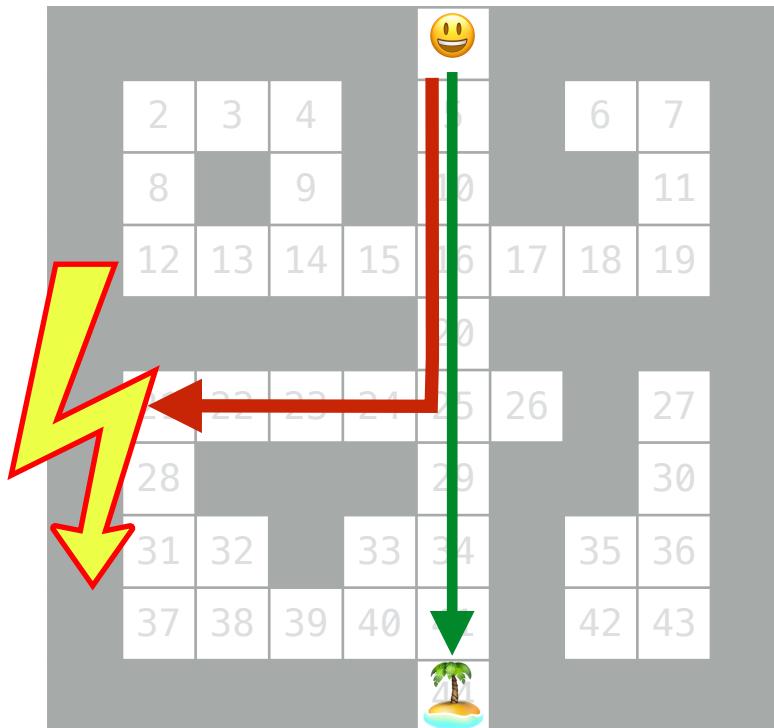
Kürzester Pfad: 9 Schritte

```
solve(pos, current): int?  
    if ! pos.legal || breadCrumb[pos]  
        return null  
    else if pos == target  
        return current  
    else  
        min = null;  
        breadCrumb[pos] = yes  
        for d in [down, left, up, right] {  
            result = solve(pos.d, current+1)  
            if (result) {  
                If !min || result < min  
                    min = result  
            }  
        }  
        breadCrumb[pos] = no  
    return min
```

Beispiel: Kürzeste Wege im Labyrinth

- **Aufgabe:** Wie lang ist ein kürzester Weg?

Es kann mehrere kürzeste Pfade geben



Kürzester Pfad: 9 Schritte

```
solve(pos, current): int?  
    if ! pos.legal || breadCrumb[pos]  
        return null  
    else if pos == target  
        return current  
    else  
        min = null;  
        breadCrumb[pos] = yes  
        for d in [down, left, up, right] {  
            result = solve(pos.d, current+1)  
            if (result) {  
                If !min || result < min  
                    min = result  
            }  
        }  
        breadCrumb[pos] = no  
    return min
```

Optimierung: ist Pfad länger als der aktuell kürzeste Pfad,
kann Suche beendet werden - man spricht von **branch and bound**

Backtracking: Beispiel

B 8-Damen-Problem

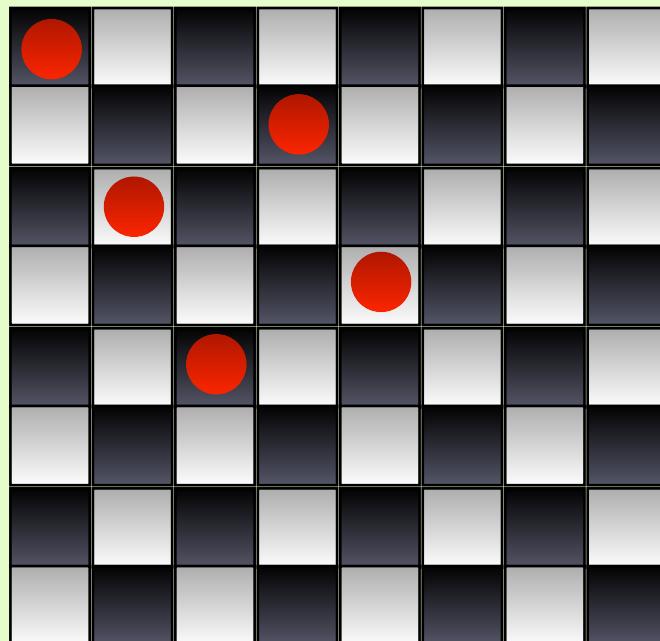
Aufgabe: Acht Damen so auf dem Schachbrett positionieren, dass sie sich nicht gegenseitig schlagen können; d.h. in keiner Diagonalen, Senkrechten oder Waagerechten darf mehr als eine Dame stehen.

Backtracking: Beispiel

B 8-Damen-Problem

Aufgabe: Acht Damen so auf dem Schachbrett positionieren, dass sie sich nicht gegenseitig schlagen können; d.h. in keiner Diagonalen, Senkrechten oder Waagerechten darf mehr als eine Dame stehen.

Eine Sackgasse

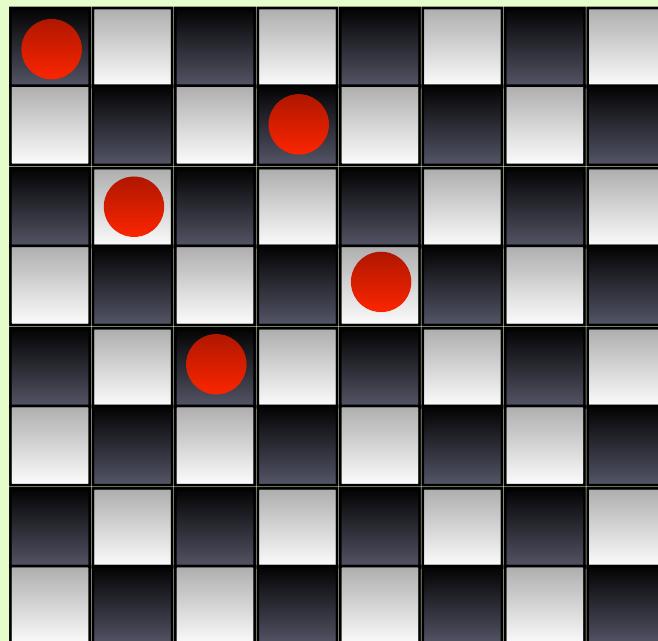


Backtracking: Beispiel

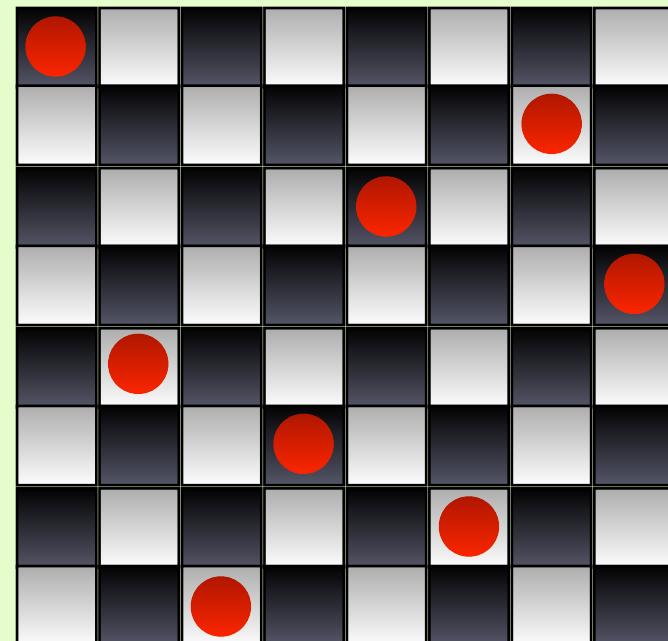
B 8-Damen-Problem

Aufgabe: Acht Damen so auf dem Schachbrett positionieren, dass sie sich nicht gegenseitig schlagen können; d.h. in keiner Diagonalen, Senkrechten oder Waagerechten darf mehr als eine Dame stehen.

Eine Sackgasse



Eine Lösung



Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, $(i+1)$ -te Dame genauso behandeln

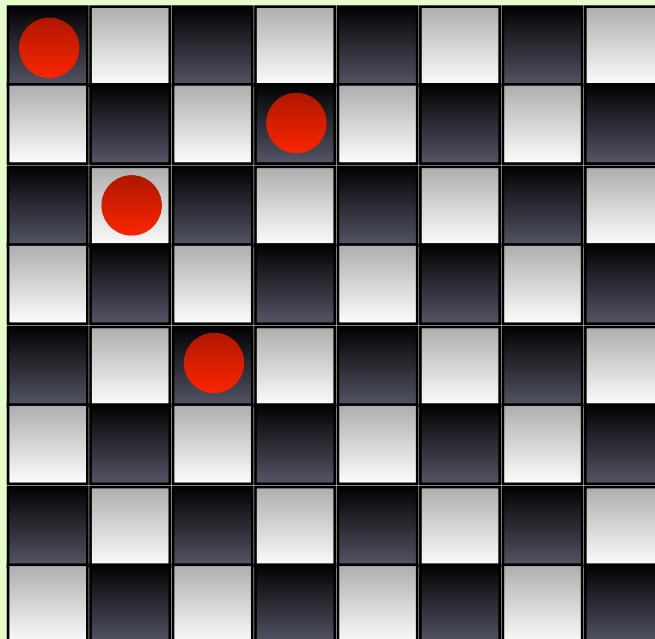
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, ($i+1$)-te Dame genauso behandeln



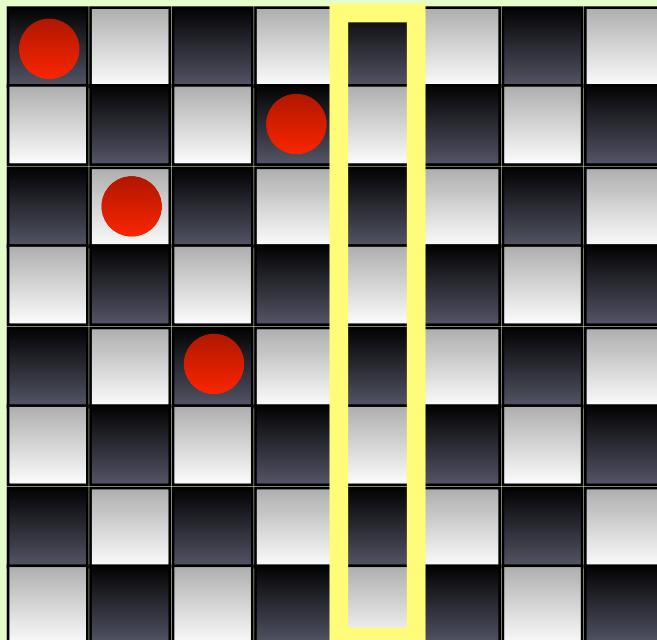
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, $(i+1)$ -te Dame genauso behandeln



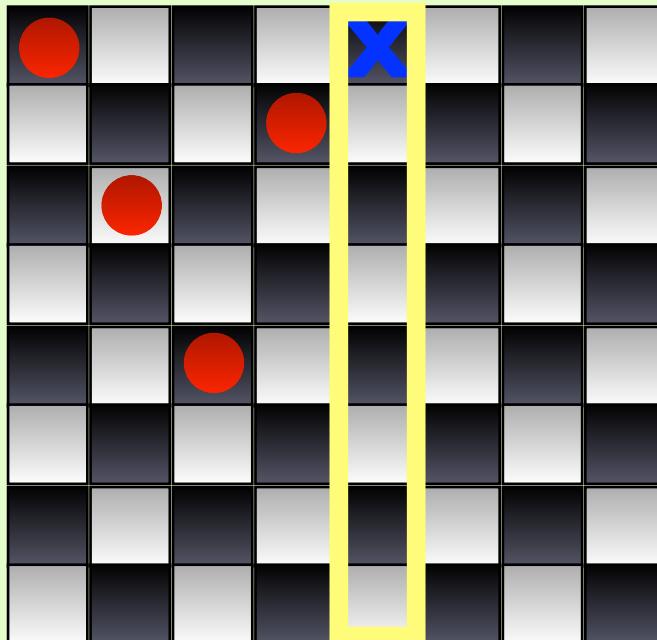
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, ($i+1$)-te Dame genauso behandeln



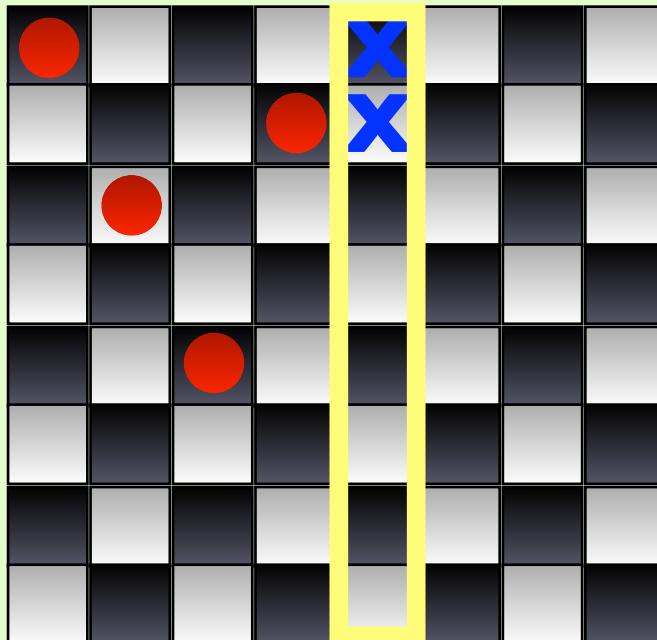
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, ($i+1$)-te Dame genauso behandeln



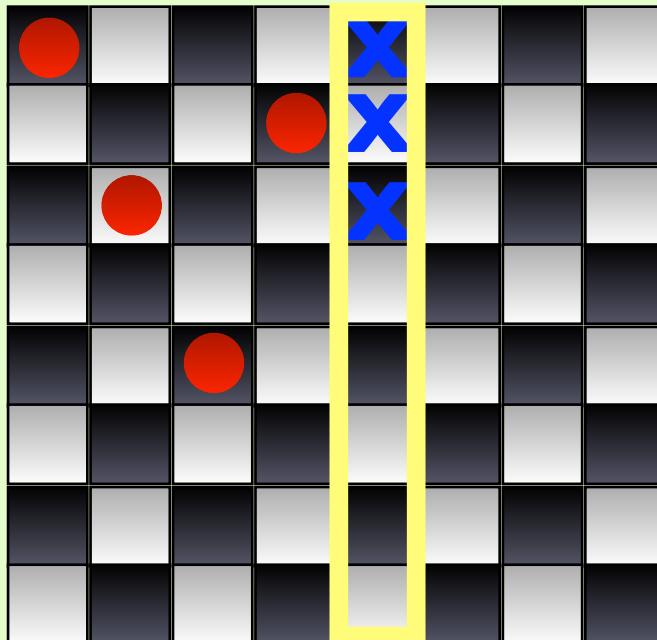
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, ($i+1$)-te Dame genauso behandeln



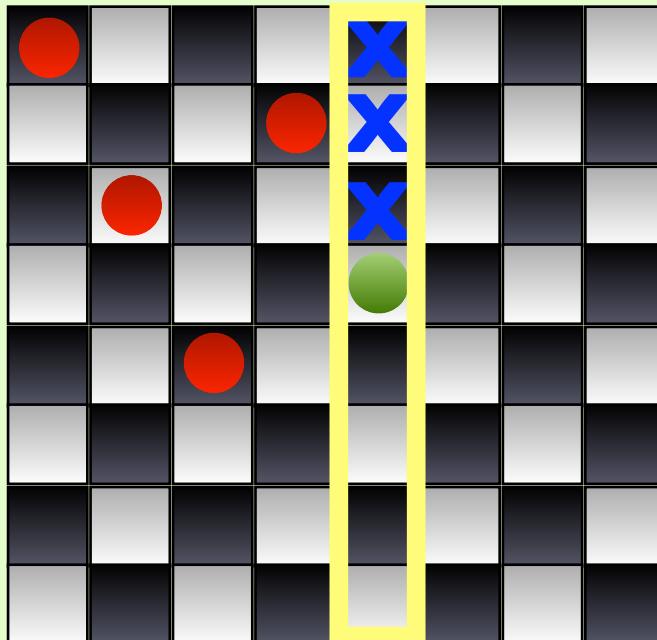
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, $(i+1)$ -te Dame genauso behandeln



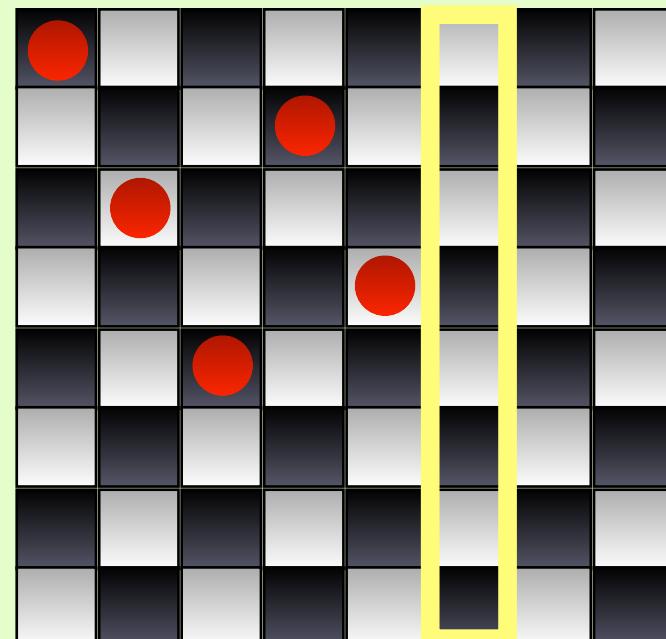
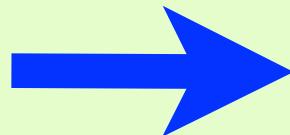
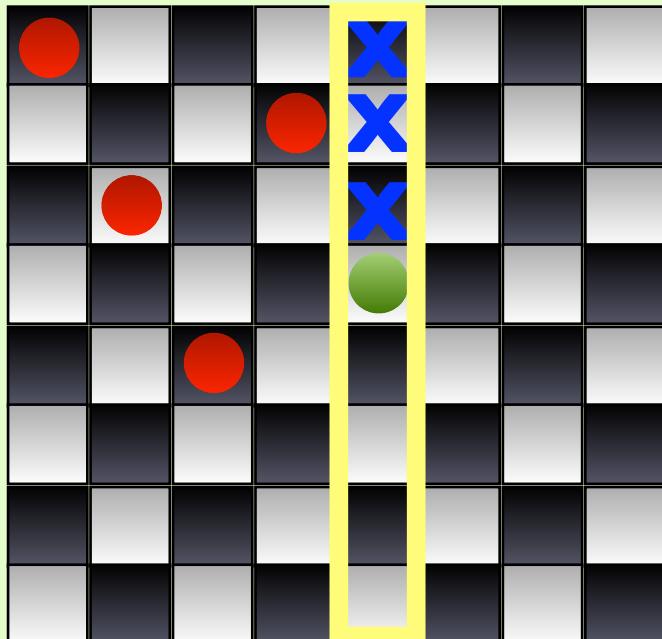
Backtracking: Beispiel

B 8-Damen-Problem

Lösungsansatz

Zerlege Problem in acht Schritte.

- in Schritt i : Eine Dame in der i -ten Spalte unter Berücksichtigung der Damen in den Spalten j ($1 \leq j < i$) platzieren.
Dazu: Platzieren in Zeile 1-8 probieren - erste Zeile wählen, in der die Dame nicht bedroht ist.
- Wurde eine sichere Position gefunden, ($i+1$)-te Dame genauso behandeln



Backtracking: Beispiel

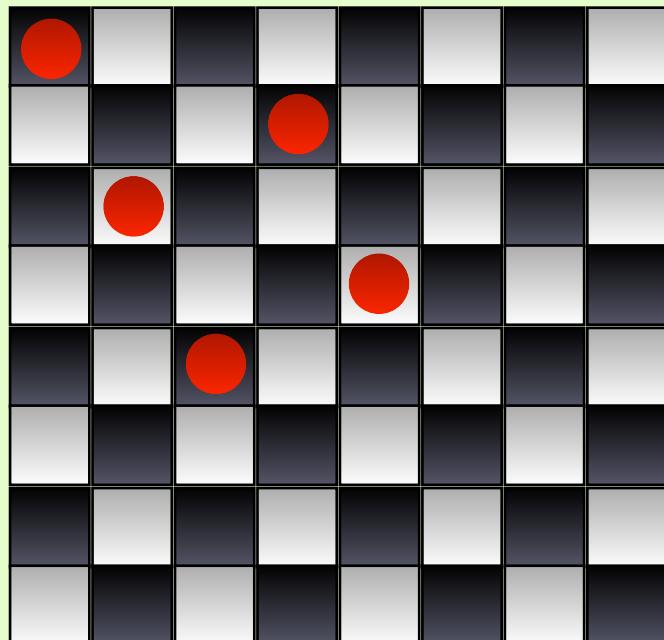
B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv

Backtracking: Beispiel

B 8-Damen-Problem

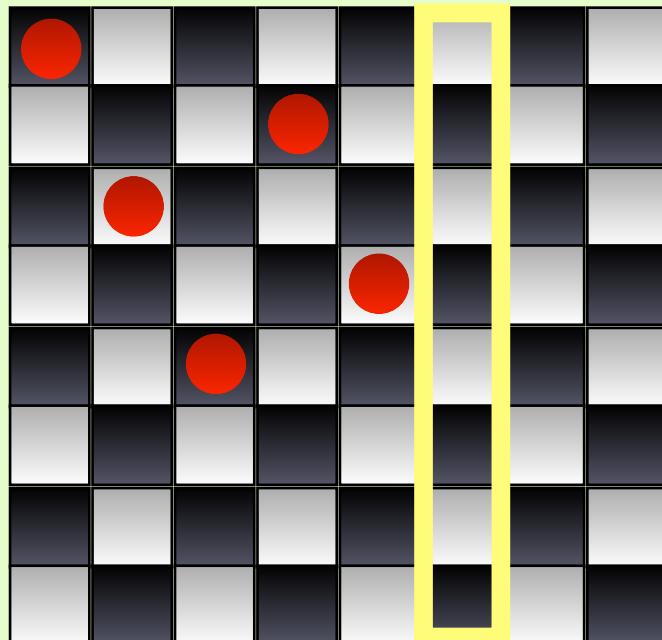
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

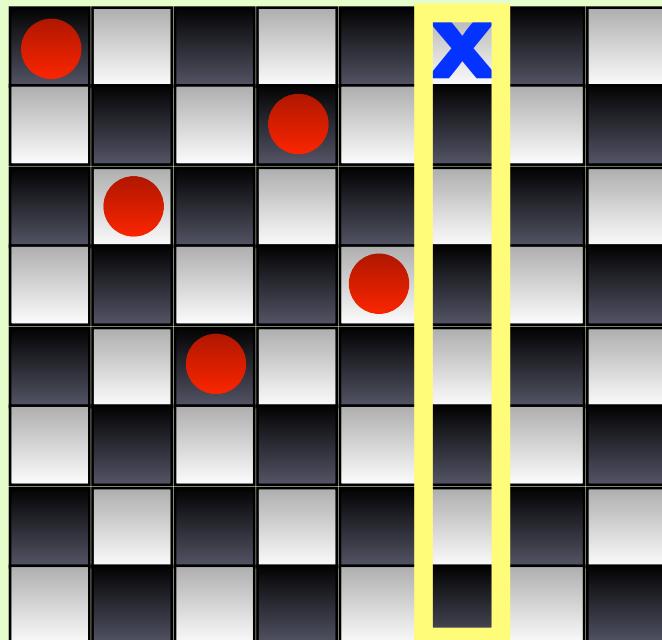
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

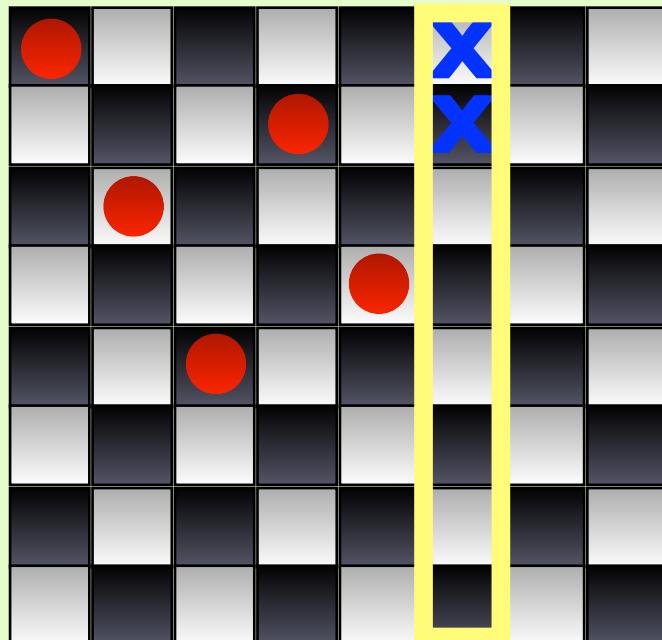
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

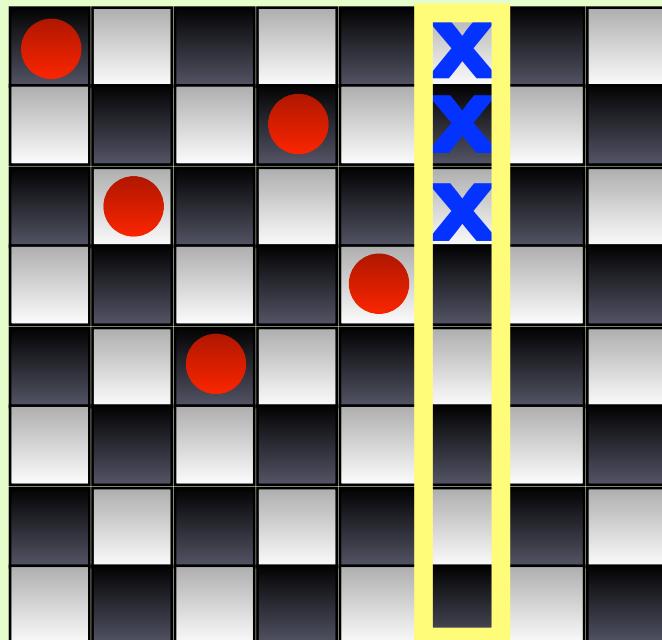
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

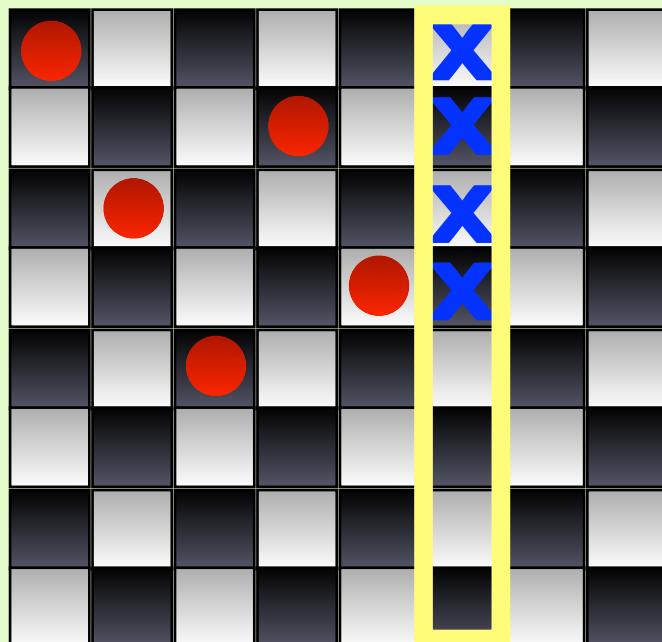
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

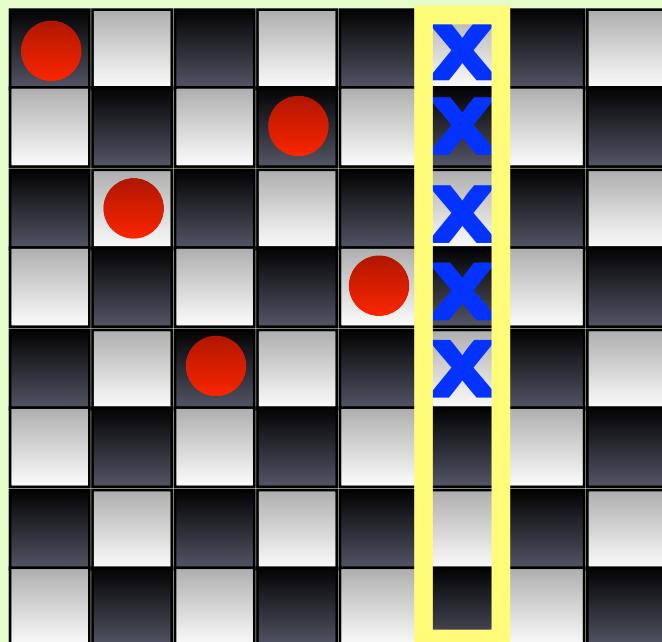
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

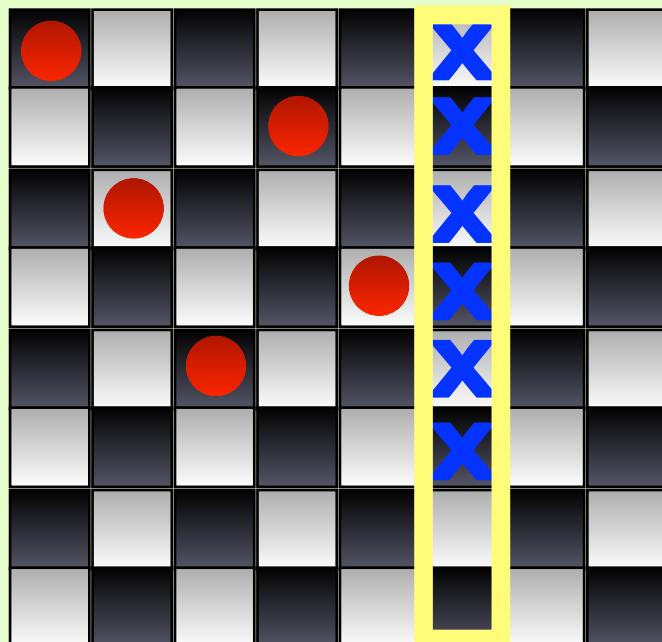
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

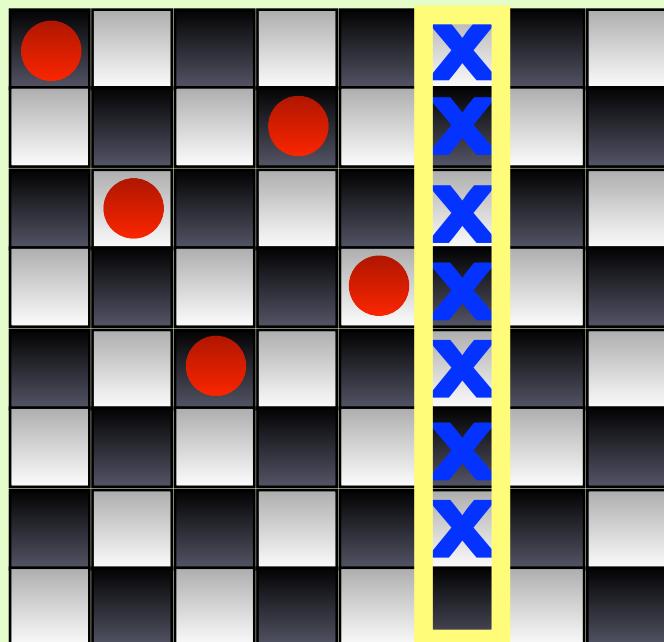
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

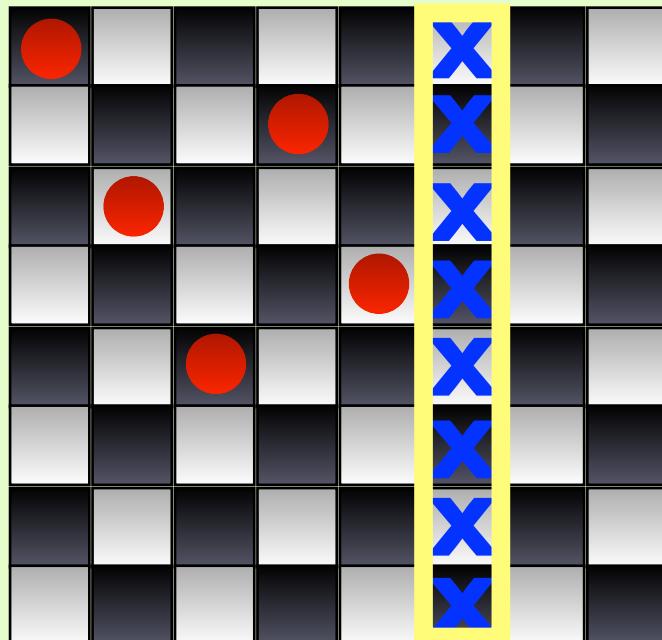
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtracking: Beispiel

B 8-Damen-Problem

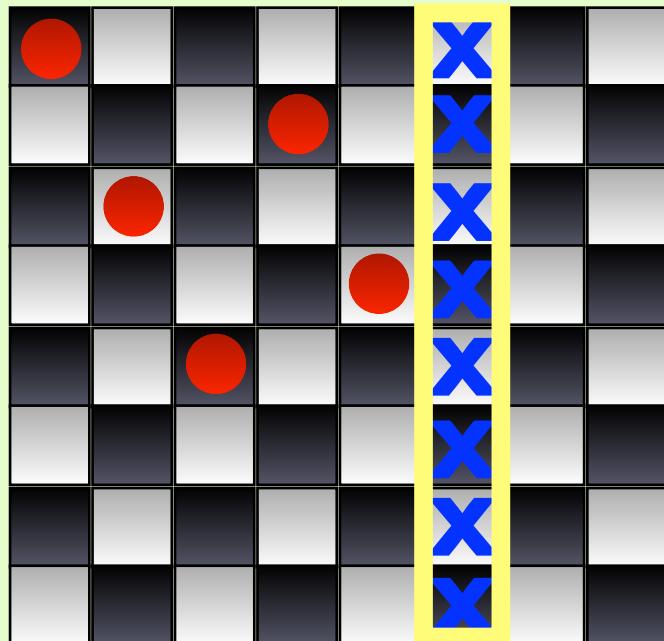
- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



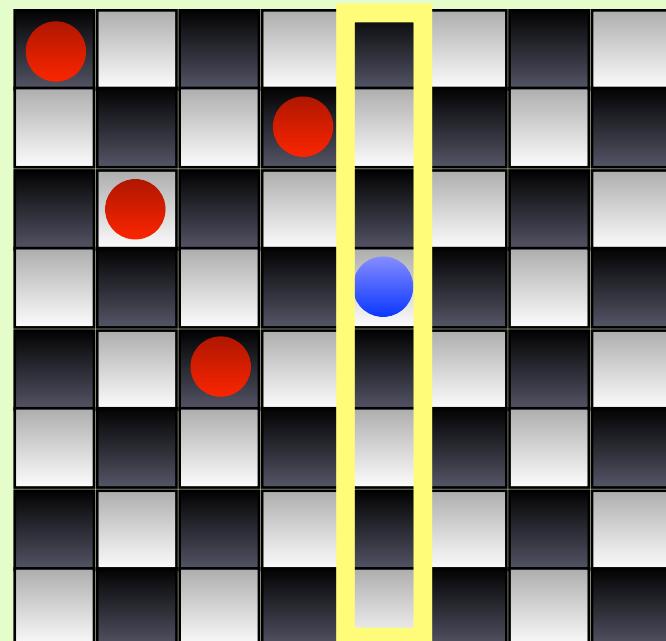
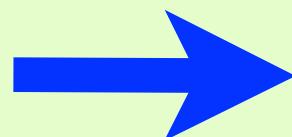
Backtracking: Beispiel

B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



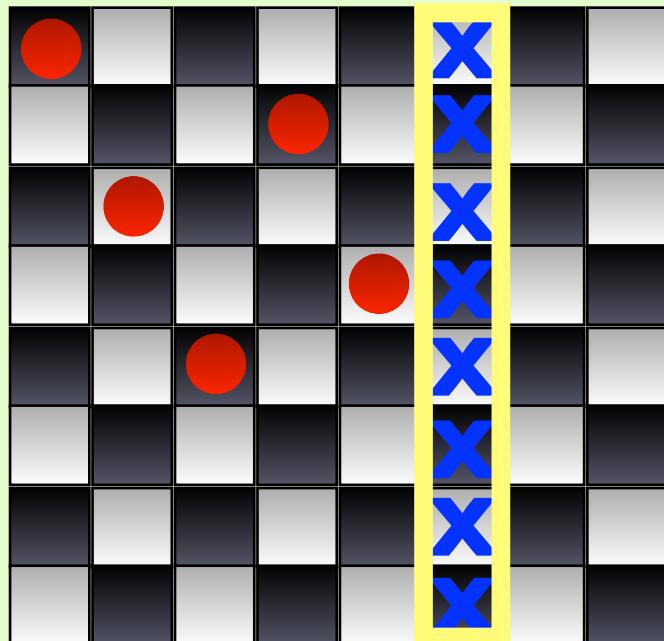
Backtrack ...



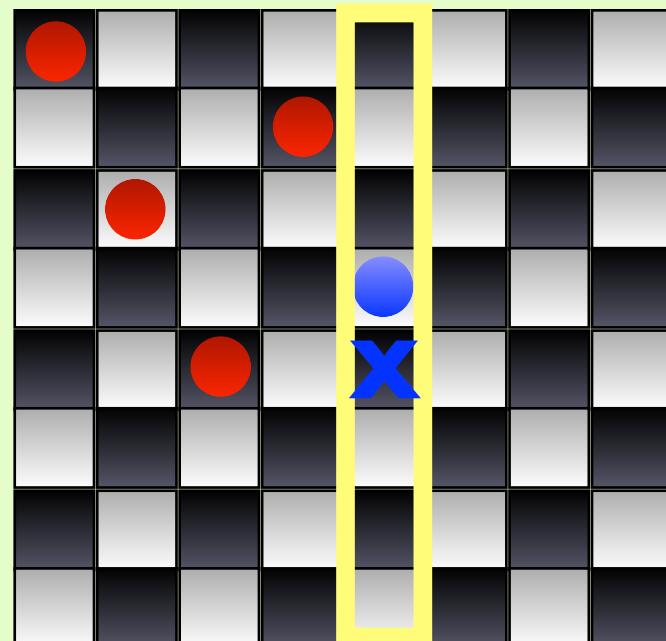
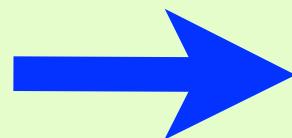
Backtracking: Beispiel

B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



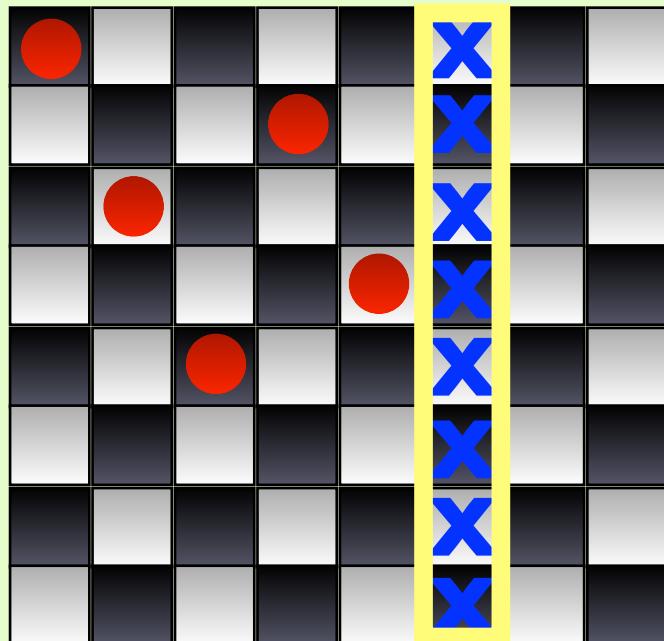
Backtrack ...



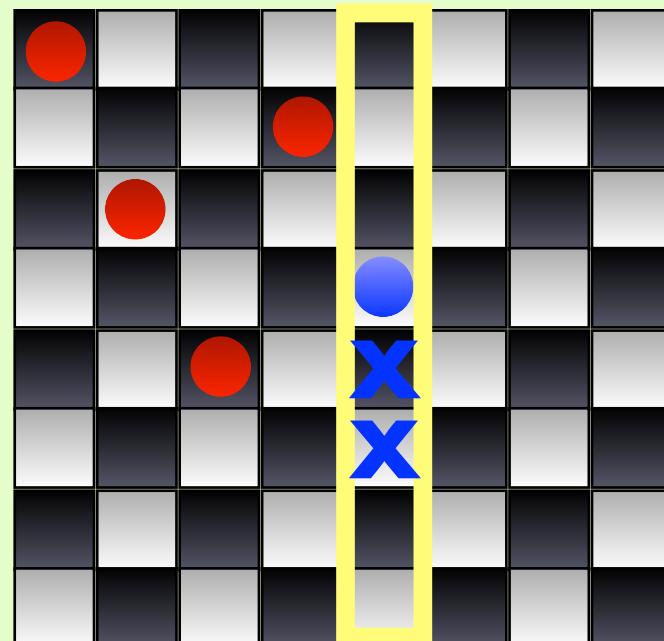
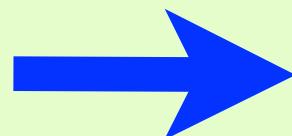
Backtracking: Beispiel

B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



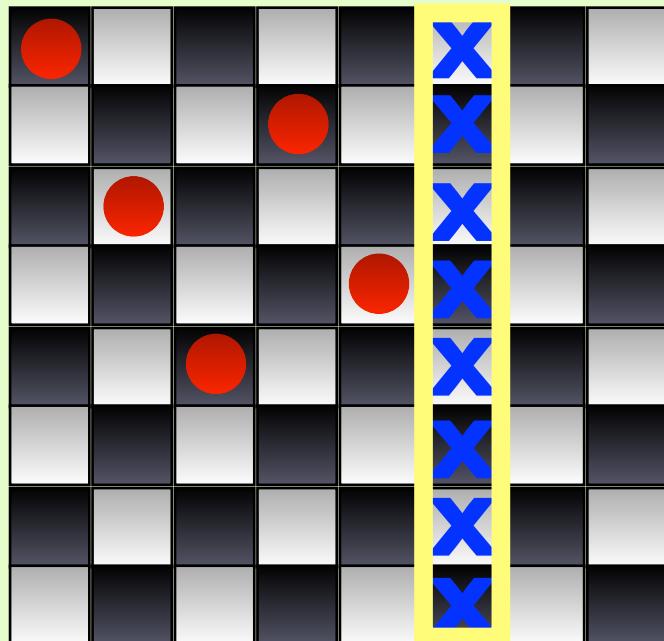
Backtrack ...



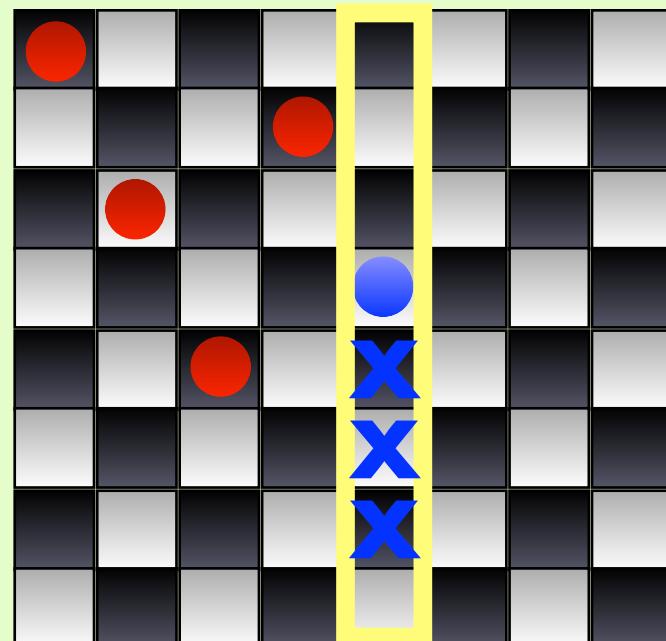
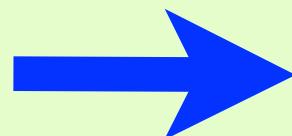
Backtracking: Beispiel

B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



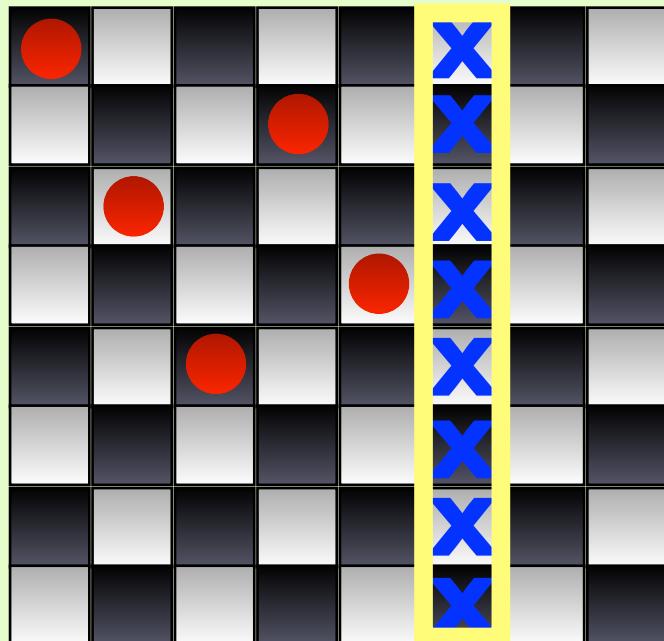
Backtrack ...



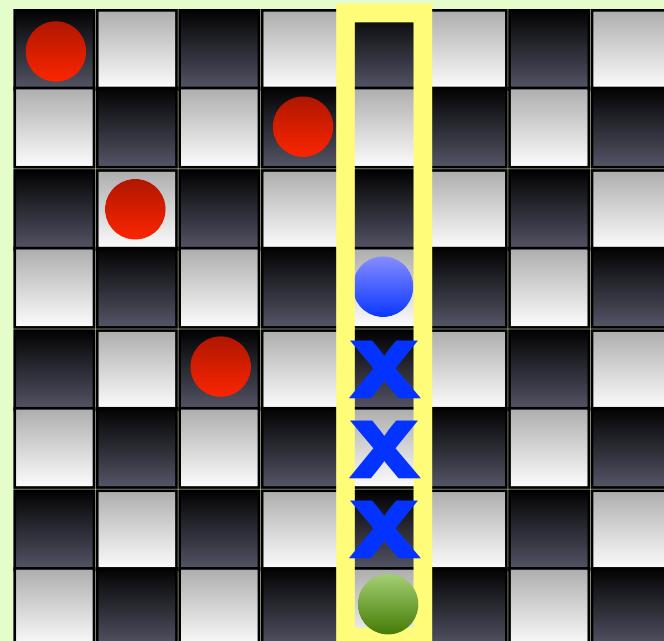
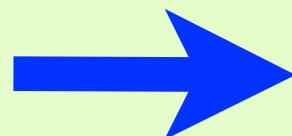
Backtracking: Beispiel

B 8-Damen-Problem

- Falls sichere Platzierung in keiner Zeile möglich, einen Schritt zurück gehen (Backtrack) und eine alternative Position für die $(i-1)$ -te Dame finden
- Verfahren funktioniert rekursiv



Backtrack ...



Backtracking: Beispiel

B 8-Damen-Problem - Implementierung in JAVA I

```
class Damen {  
  
    /* Zeilenposition der 8 Damen */  
    static int Position[] = new int[8];  
  
    /* Prüft, ob Feld Spalte=n,Zeile=y bedroht ist */  
    static boolean feldBedroht(int n,int y) {  
        /* Alle bereits gesetzten Damen testen */  
        for (int i=0 ; i<n ; ++i) {  
            if (Position[i] == y)  
                return true; /* Zeile gleich? */  
            int xdistance = n-i; /* Spaltenabstand der Figuren */  
            if ((y == Position[i] - xdistance) ||  
                (y == Position[i] + xdistance))  
                return true; /* Diagonale? */  
        }  
        return false;  
    }  
}
```

Backtracking: Beispiel

B 8-Damen-Problem - Implementierung in JAVA 2

```
/* Berechnung einer Lösung:          */
/* Setzen der n-ten Dame in die n-te Spalte */
static boolean setzeDame(int n) {
    /* Alle Damen gesetzt, dann sind wir fertig      */
    if (n>=8) return true;
    /* Versuche Dame in einer der Zeilen zu setzen */
    for (int zeile=0 ; zeile<8 ; zeile++) {
        /* Nur wenn Feld nicht bedroht .... */
        if (! feldBedroht(n,zeile) ) {
            /* ... Feld merken und */
            Position[n] = zeile;
            /* prüfen, ob Setzen der nächsten Dame zum */
            /* Ziel führt. Wenn ja, wurde eine Lösung */
            /* gefunden und wir können aufhören      */
            if (setzeDame(n+1)) return true;
        } /* ansonsten: nächste Zeile probieren */
    } /* Beachte: der letzte Zug wird zurückgenommen !*/
    /* Sollten wir hier ankommen, wurde keine Lösung */
    /* gefunden                                         */
    return false;
}
```

Backtracking: Beispiel

B 8-Damen-Problem - Implementierung in JAVA 3

```
/* Berechnung aller Lösungen: */  
/* Setzen der n-ten Dame in die n-te Spalte. */  
/* Gefundene Lösungen werden in einer verketteten Liste */  
/* abgelegt */  
static void setzeDameAll(LinkedList<int[]> solutions, int n) {  
    /* Wurden 8 Damen erfolgreich gesetzt? */  
    if (n>=8) {  
        /* Dann haben wir eine neue Lösung gefunden */  
        solutions.add(Position.clone());  
        return;  
    }  
    /* Alle Zeilen durchprobieren ... */  
    for (int zeile=0 ; zeile<8 ; zeile++) {  
        /* Wenn Zug möglich ist, entwickeln wir von hier die */  
        /* Lösung weiter */  
        if (! feldBedroht(n,zeile)) {  
            Position[n] = zeile;  
            setzeDameAll(solutions,n+1);  
            /* Selbst wenn der Versuch erfolgreich war, fahren */  
            /* wir fort und testen die nächste Alternative */  
        }  
    }  
}
```

III. Entwurfsmethoden

3. Entwurfsmethoden

- 3.1. Teile und Herrsche
- 3.2. Backtracking
- 3.3. Memoization**
- 3.4. Dynamische Programmierung
- 3.5. Greedy-Algorithmen

Memoization - Die Idee

- **Memoization:** abgeleitet vom lat. **Memorandum:** das zu Erinnernde
Memorandum oft Memo genannt; Memoization als Umwandlung einer Funktion in ein Memo

Memoization - Die Idee

- **Memoization:** abgeleitet vom lat. **Memorandum:** das zu Erinnernde
Memorandum oft Memo genannt; Memoization als Umwandlung einer Funktion in ein Memo
- **Anwendbar z.B. bei rekursiven Funktionen**
 - → Rückgriff auf Funktionswerte aus früheren Aufrufen
 - Folge: mehrfache Berechnung von Funktionswerten

Beispiel: Fibonacci-Folge:

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

Memoization - Die Idee

- **Memoization:** abgeleitet vom lat. **Memorandum:** das zu Erinnernde
Memorandum oft Memo genannt; Memoization als Umwandlung einer Funktion in ein Memo
- **Anwendbar z.B. bei rekursiven Funktionen**
 - → Rückgriff auf Funktionswerte aus früheren Aufrufen
 - Folge: mehrfache Berechnung von Funktionswerten
Beispiel: Fibonacci-Folge:
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$
- **Lösung:**
 - Teil- bzw. Zwischenergebnisse in Tabelle merken

Memoization - Beispiel

B Memoized Fibonacci

```
/* Die rekursive Struktur bleibt erhalten, doppelte Berechnungen
 * werden jedoch vermieden. */
unsigned int fibMemoRec(unsigned int* memory,
                        unsigned int n,unsigned int l) {
    if (l>n) return memory[n];
    memory[l] = memory[l-1] + memory[l-2]; // fib(l)=fib(l-1)+fib(l-2)
    fibMemoRec(memory, n, l+1);           // Tail-Call!
}

unsigned int fibMemoized(int n) {
    unsigned int* memory = new unsigned int(n);
    memory[0]=0; // "Gedächtnis" initialisieren
    memory[1]=1;
    memory[2]=1;
    return fibMemoRec( memory, n, 3);
}
```

Memoization - Beispiel

B Memoized Fibonacci

```
/* Die rekursive Struktur bleibt erhalten, doppelte Berechnungen
 * werden jedoch vermieden. */
unsigned int fibMemoRec(unsigned int* memory,
                        unsigned int n,unsigned int l) {
    if (l>n) return memory[n];
    memory[l] = memory[l-1] + memory[l-2]; // fib(l)=fib(l-1)+fib(l-2)
    fibMemoRec(memory, n, l+1);           // Tail-Call!
}

unsigned int fibMemoized(int n) {
    unsigned int* memory = new unsigned int(n);
    memory[0]=0; // "Gedächtnis" initialisieren
    memory[1]=1;
    memory[2]=1;
    return fibMemoRec( memory, n, 3);
}
```

Komplexität: $O(n)$

III. Entwurfsmethoden

3. Entwurfsmethoden

- 3.1. Teile und Herrsche
- 3.2. Backtracking
- 3.3. Memoization
- 3.4. Dynamische Programmierung**
- 3.5. Greedy-Algorithmen

Dynamische Programmierung: Einleitung

- **Optimierungsaufgabe:** Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen

Dynamische Programmierung: Einleitung

- **Optimierungsaufgabe:** Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen
- **Grundprinzip:**
 - finde optimale Lösung für „kleine“ Elementarprobleme
 - konstruiere sukzessive „größere“ Lösungen bis zur Gesamtlösung

Dynamische Programmierung: Einleitung

- **Optimierungsaufgabe:** Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen
- **Grundprinzip:**
 - finde optimale Lösung für „kleine“ Elementarprobleme
 - konstruiere sukzessive „größere“ Lösungen bis zur Gesamtlösung
- **Die Strategie besagt:**
 - Teilprobleme bearbeiten
 - Teilergebnisse in Tabellen eintragen (wie bei *Memoization*)
 - Zusammensetzen der Gesamtlösung (*bottom up*)

Dynamische Programmierung: Einleitung

- **Optimierungsaufgabe:** Optimiere eine Zielfunktion unter Beachtung von Nebenbedingungen
- **Grundprinzip:**
 - finde optimale Lösung für „kleine“ Elementarprobleme
 - konstruiere sukzessive „größere“ Lösungen bis zur Gesamtlösung
- **Die Strategie besagt:**
 - Teilprobleme bearbeiten
 - Teilergebnisse in Tabellen eintragen (wie bei *Memoization*)
 - Zusammensetzen der Gesamtlösung (*bottom up*)
- **Anwendungsbedingungen**
 - optimale Lösung enthält optimale Teillösungen
 - *Divide and Conquer* nicht anwendbar: **überlappende Teillösungen**

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 1

Seien $A \in \mathbb{R}^{l \times m}$ und $B \in \mathbb{R}^{m \times n}$ zwei reellwertige Matrizen. Für die Elemente der Produktmatrix $C = A \cdot B$ mit $C \in \mathbb{R}^{l \times n}$ gilt:

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

wobei $1 \leq i \leq l$ und $1 \leq j \leq n$.

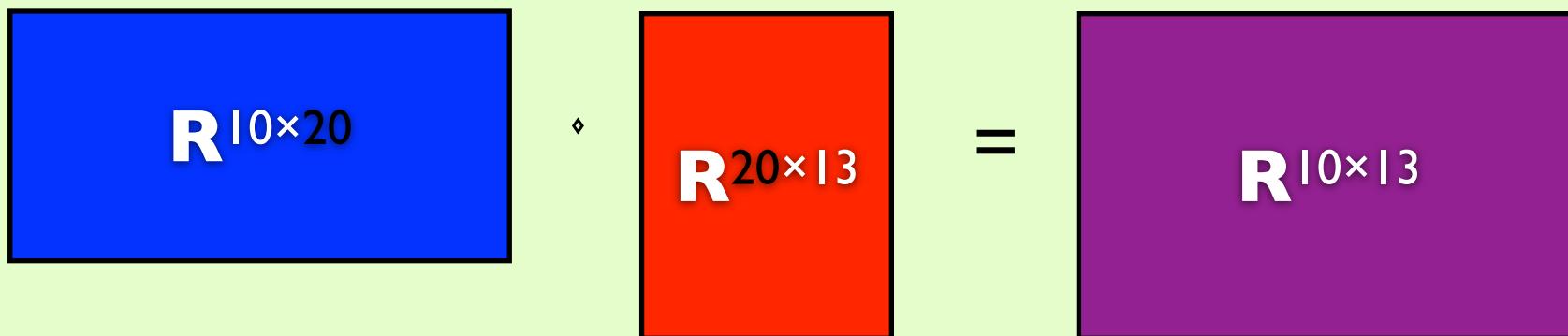
Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 1

Seien $A \in \mathbb{R}^{l \times m}$ und $B \in \mathbb{R}^{m \times n}$ zwei reellwertige Matrizen. Für die Elemente der Produktmatrix $C = A \cdot B$ mit $C \in \mathbb{R}^{l \times n}$ gilt:

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

wobei $1 \leq i \leq l$ und $1 \leq j \leq n$.



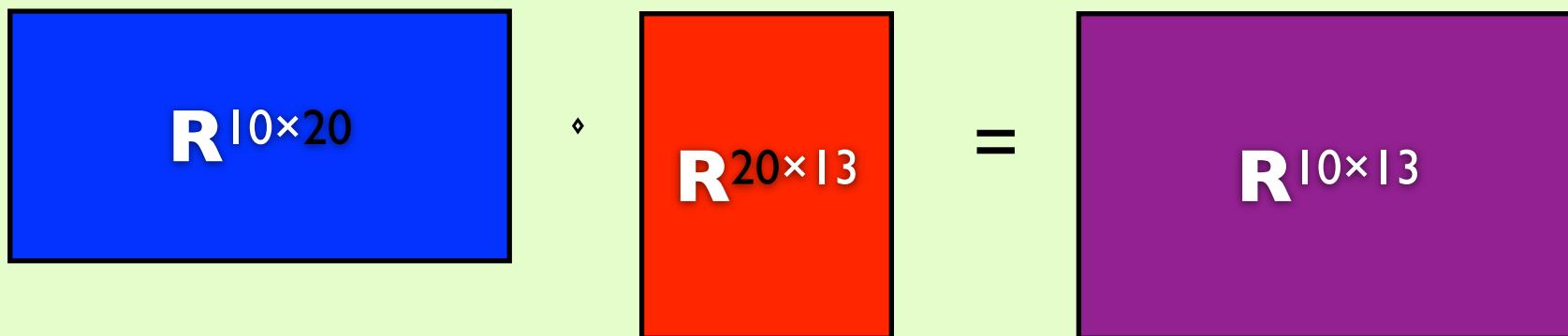
Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 1

Seien $A \in \mathbb{R}^{l \times m}$ und $B \in \mathbb{R}^{m \times n}$ zwei reellwertige Matrizen. Für die Elemente der Produktmatrix $C = A \cdot B$ mit $C \in \mathbb{R}^{l \times n}$ gilt:

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

wobei $1 \leq i \leq l$ und $1 \leq j \leq n$.

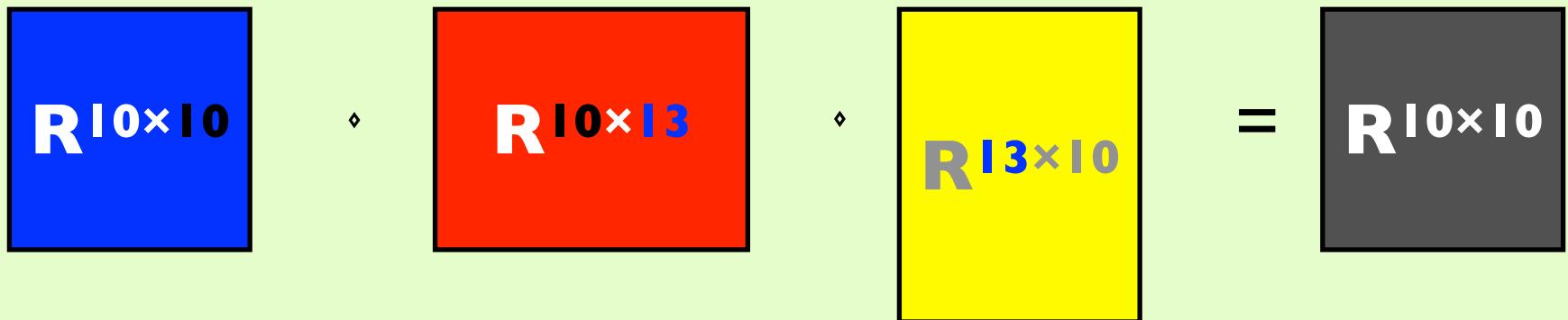


Die Berechnung des Produkts erfordert $l \cdot n \cdot m$ skalare **Multiplikationen**.

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

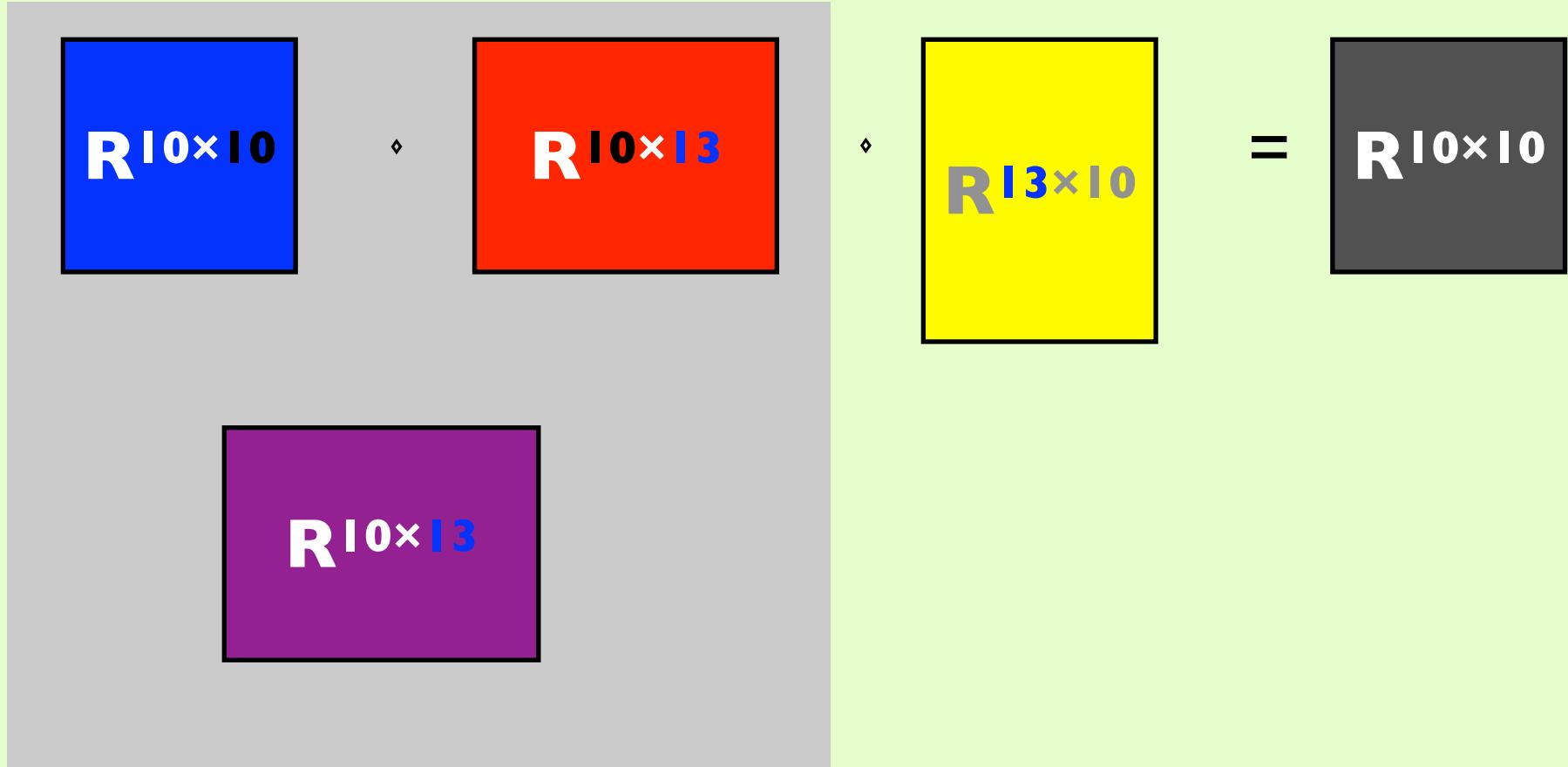
Die Matrixmultiplikation ist **assoziativ**.



Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

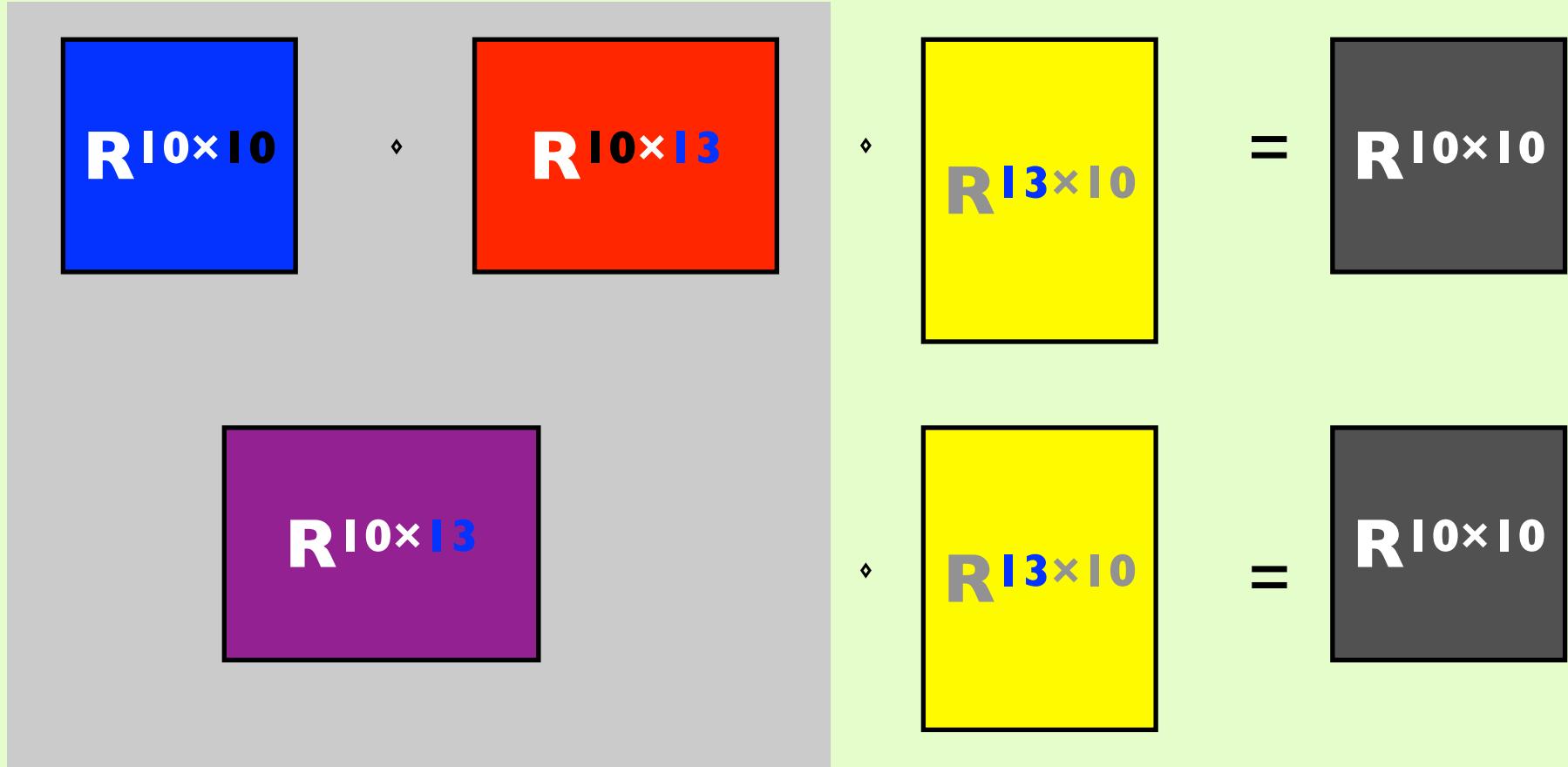
Die Matrixmultiplikation ist **assoziativ**.



Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

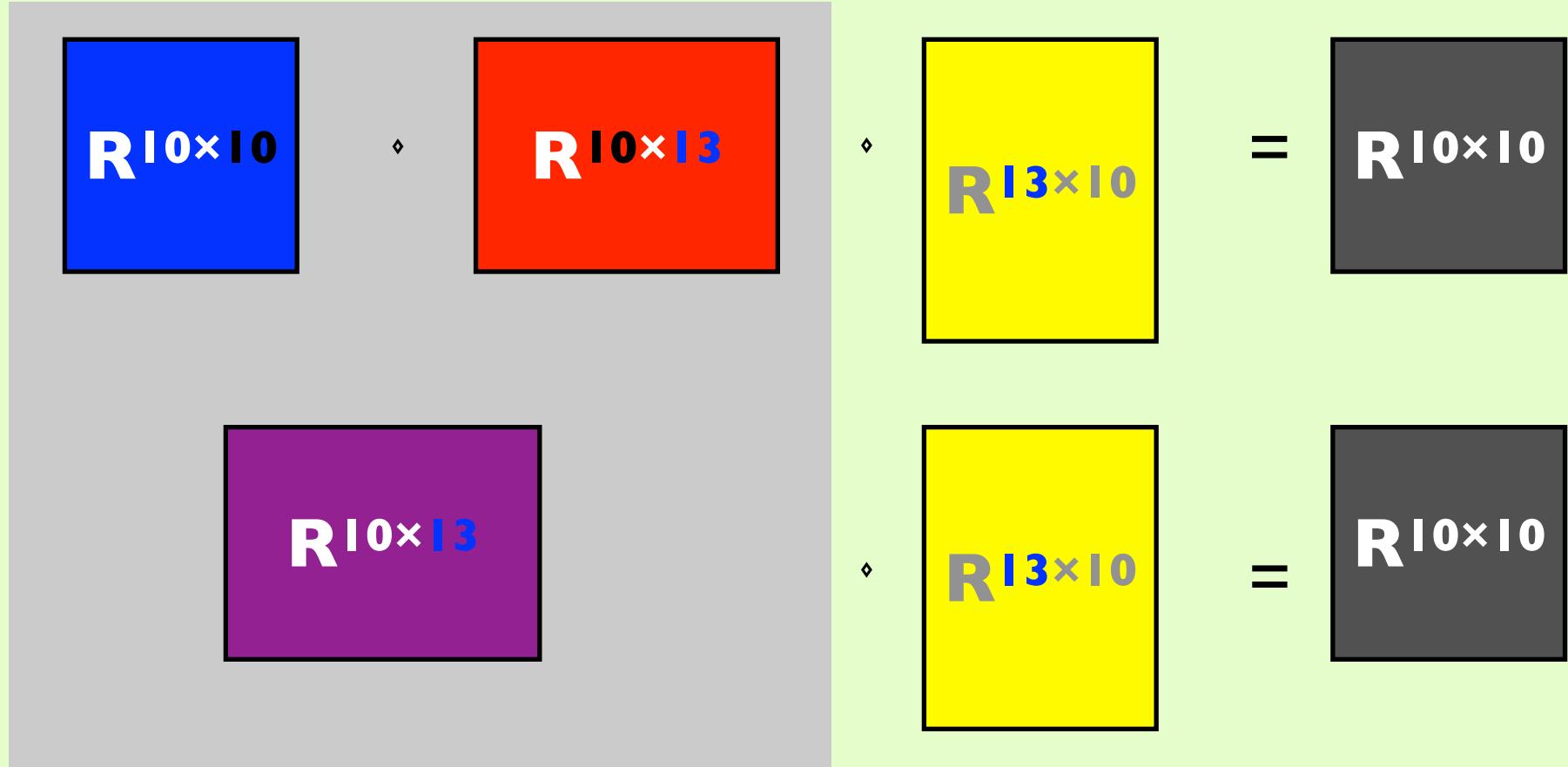
Die Matrixmultiplikation ist **assoziativ**.



Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

Die Matrixmultiplikation ist **assoziativ**.

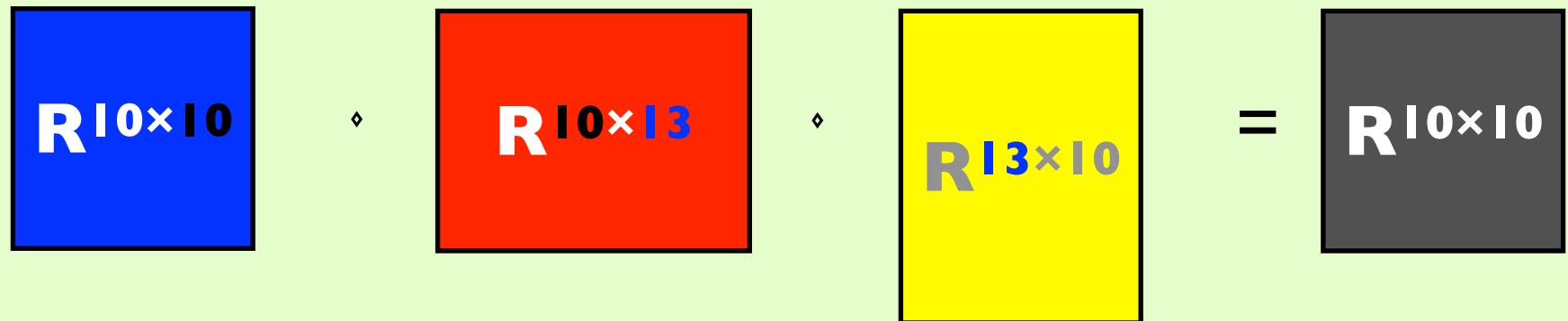


Insgesamt $(10 \cdot 10 \cdot 13) + (10 \cdot 13 \cdot 10) = 2.600$ Multiplikationen

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

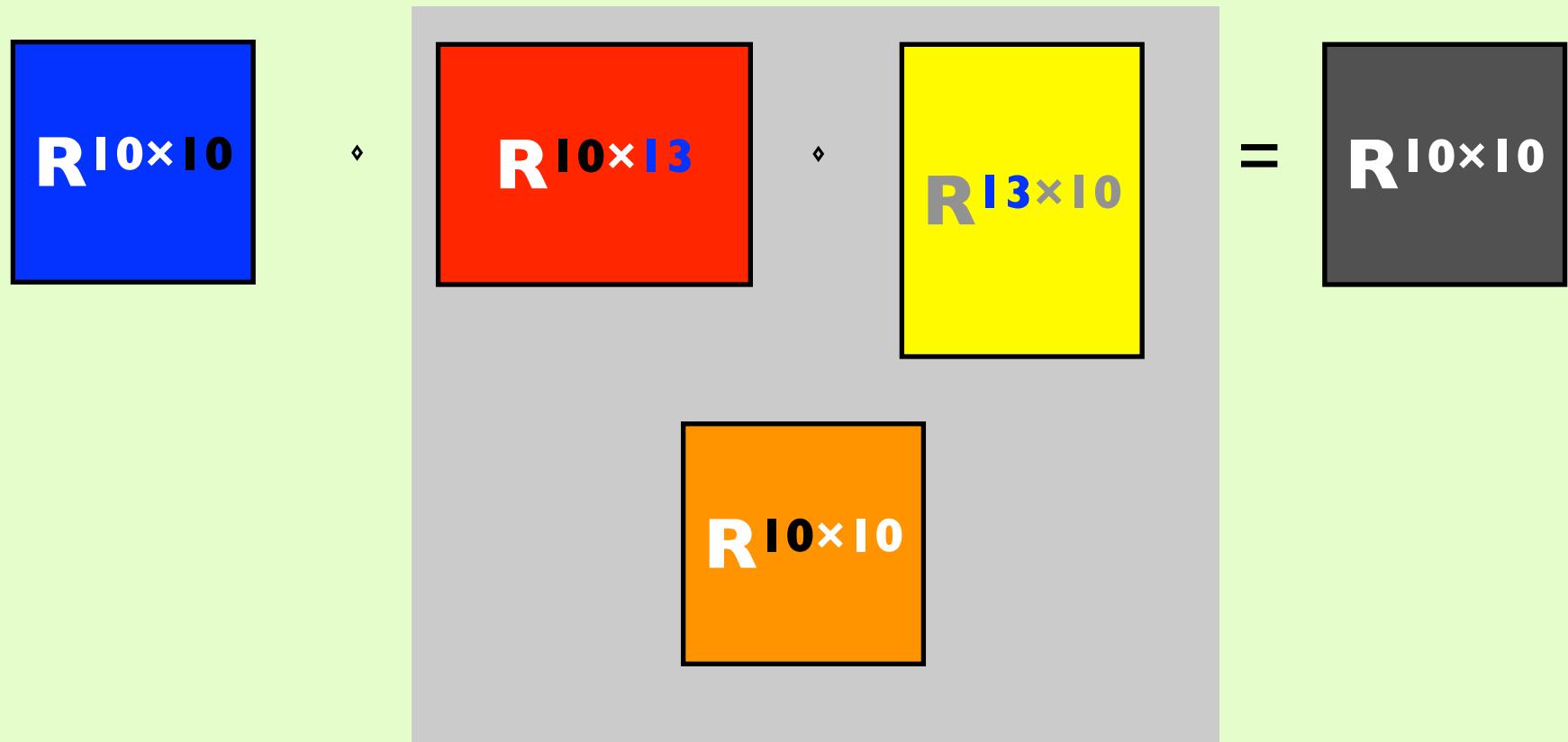
Die Matrixmultiplikation ist **assoziativ**.



Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

Die Matrixmultiplikation ist **assoziativ**.



Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

Die Matrixmultiplikation ist **assoziativ**.

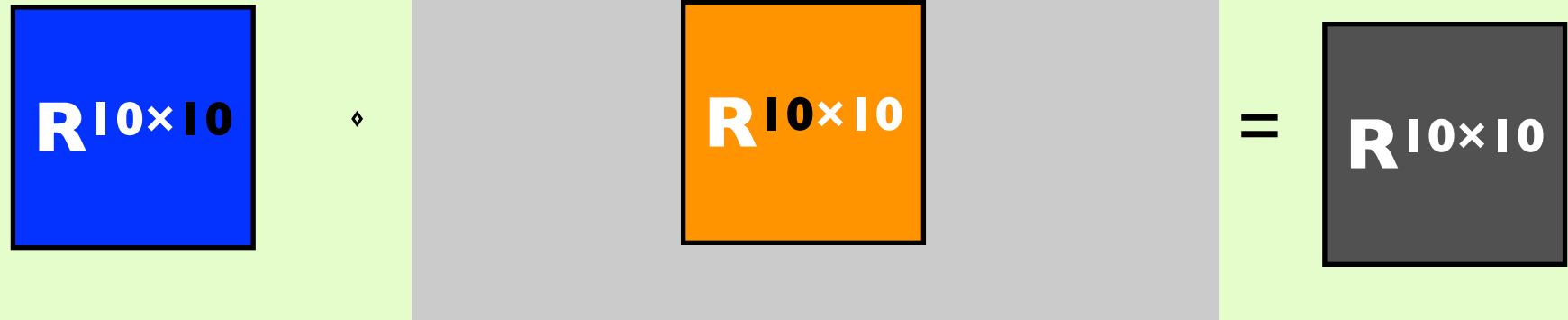
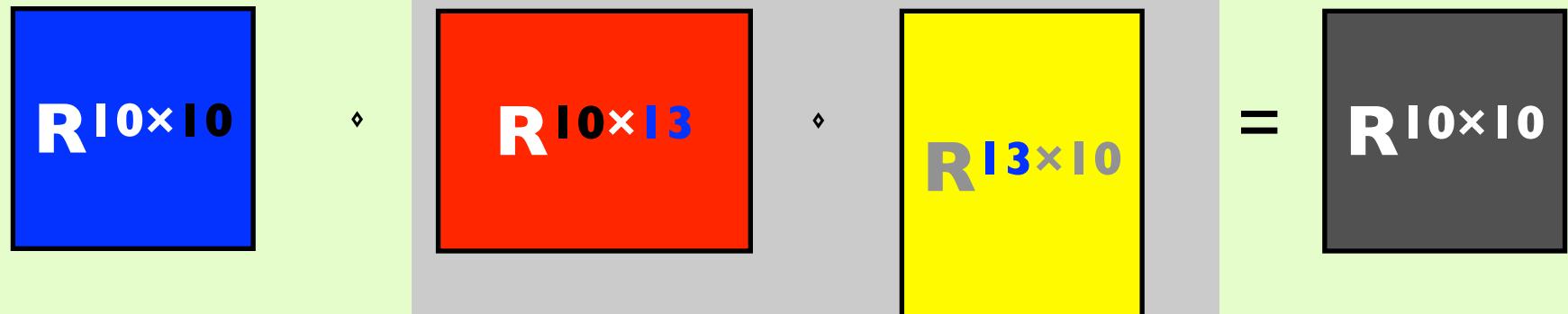
$$\begin{matrix} \text{R}^{10 \times 10} & \diamond & \text{R}^{10 \times 13} & \diamond & \text{R}^{13 \times 10} \\ & & & & = \\ & & & & \text{R}^{10 \times 10} \end{matrix}$$

$$\begin{matrix} \text{R}^{10 \times 10} & \diamond & \text{R}^{10 \times 10} \\ & & = \\ & & \text{R}^{10 \times 10} \end{matrix}$$

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Motivation 2

Die Matrixmultiplikation ist **assoziativ**.



Insgesamt $(10 \cdot 10 \cdot 13) + (10 \cdot 10 \cdot 10) = 2.300$ Multiplikationen

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Assoziativität und Performance

Seien $M_1 \in \mathbb{R}^{10 \times 20}$, $M_2 \in \mathbb{R}^{20 \times 50}$, $M_3 \in \mathbb{R}^{50 \times 1}$ und $M_4 \in \mathbb{R}^{1 \times 100}$ Matrizen.

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Assoziativitat und Performance

Seien $M_1 \in \mathbb{R}^{10 \times 20}$, $M_2 \in \mathbb{R}^{20 \times 50}$, $M_3 \in \mathbb{R}^{50 \times 1}$ und $M_4 \in \mathbb{R}^{1 \times 100}$ Matrizen.

$$1. \quad M_1 \cdot \underbrace{\left(M_2 \cdot \underbrace{\left(M_3 \cdot M_4 \right)}_{[50,1,100]} \right)}_{[20,50,100]} \quad \begin{array}{r} 5000 \text{ Operationen} \\ + 100000 \text{ Operationen} \\ + 20000 \text{ Operationen} \\ \hline 125000 \text{ Operationen} \end{array}$$

[10,20,100]

$$2. \quad \underbrace{\left(M_1 \cdot \underbrace{\left(M_2 \cdot M_3 \right)}_{[20,50,1]} \right) \cdot M_4}_{[10,20,1]} \quad \begin{array}{r} 1000 \text{ Operationen} \\ + 200 \text{ Operationen} \\ + 1000 \text{ Operationen} \\ \hline 2200 \text{ Operationen} \end{array}$$

[10,1,100]

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Assoziativität und Performance

Seien $M_1 \in \mathbb{R}^{10 \times 20}$, $M_2 \in \mathbb{R}^{20 \times 50}$, $M_3 \in \mathbb{R}^{50 \times 1}$ und $M_4 \in \mathbb{R}^{1 \times 100}$ Matrizen.

$$1. \quad M_1 \cdot \underbrace{(M_2 \cdot \underbrace{\underbrace{(M_3 \cdot M_4)}_{[50,1,100]}}_{[20,50,100]})}_{[10,20,100]} \quad \begin{array}{r} 5000 \text{ Operationen} \\ + 100000 \text{ Operationen} \\ + 20000 \text{ Operationen} \\ \hline 125000 \text{ Operationen} \end{array}$$

$$2. \quad (M_1 \cdot \underbrace{\underbrace{(M_2 \cdot M_3)}_{[20,50,1]}}_{[10,20,1]}) \cdot M_4 \quad \begin{array}{r} 1000 \text{ Operationen} \\ + 200 \text{ Operationen} \\ + 1000 \text{ Operationen} \\ \hline 2200 \text{ Operationen} \end{array}$$

Aufgabe: finde günstigste Klammerung
für Kette M_1, \dots, M_n

Matrixkettenprodukt - *Exhaustive Search*

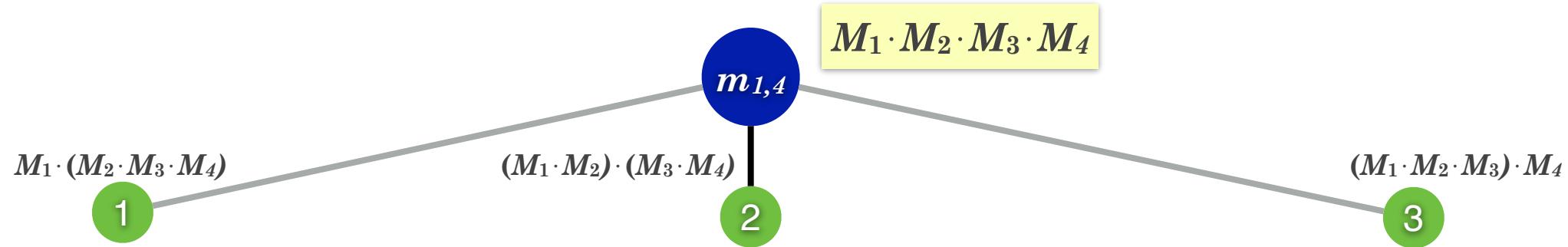
- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt

$m_{1,4}$

$M_1 \cdot M_2 \cdot M_3 \cdot M_4$

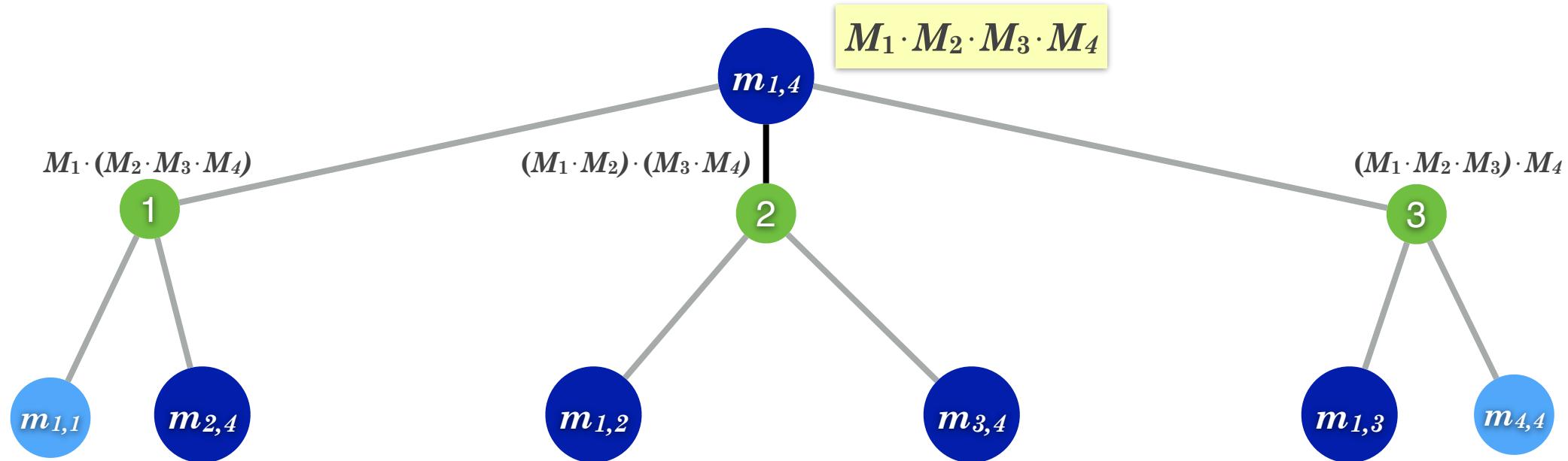
Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

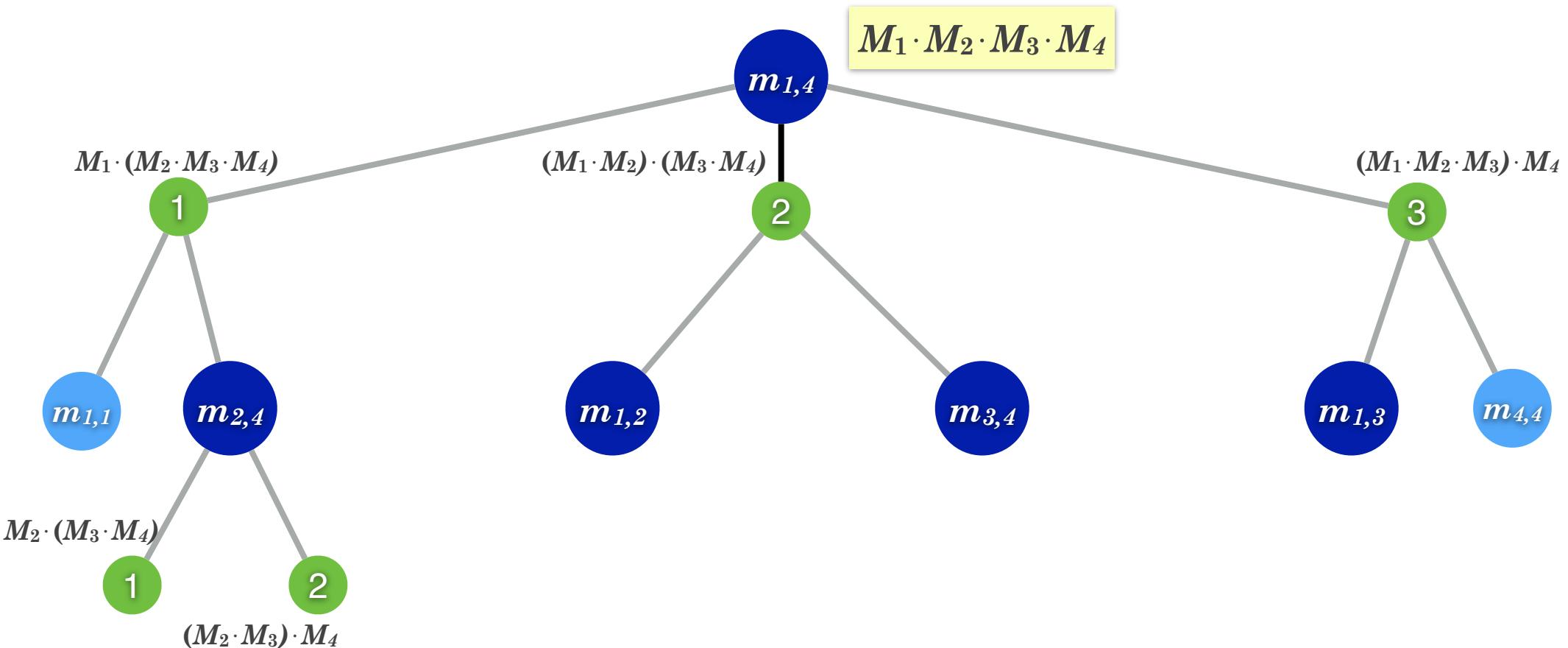
- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus

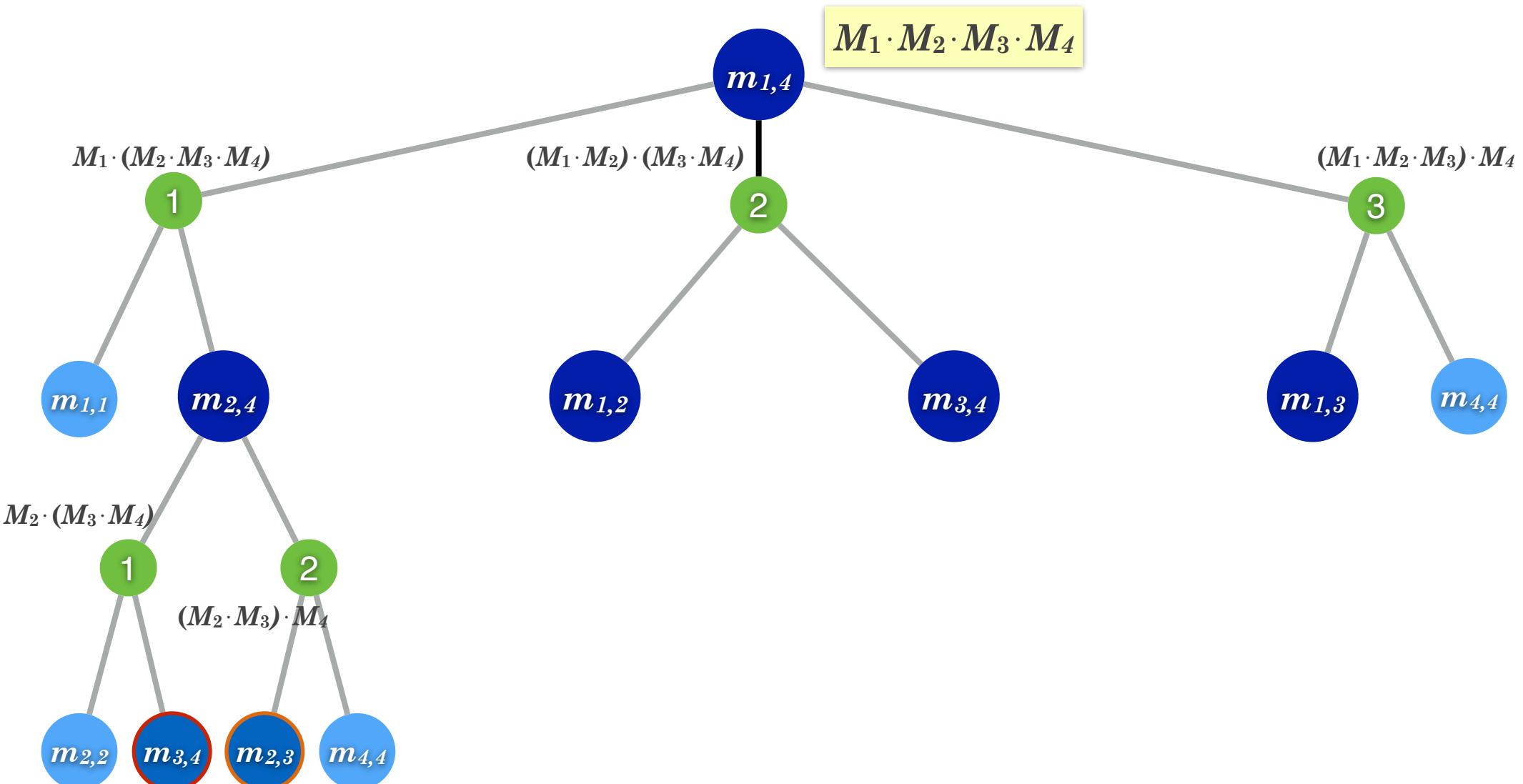
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus

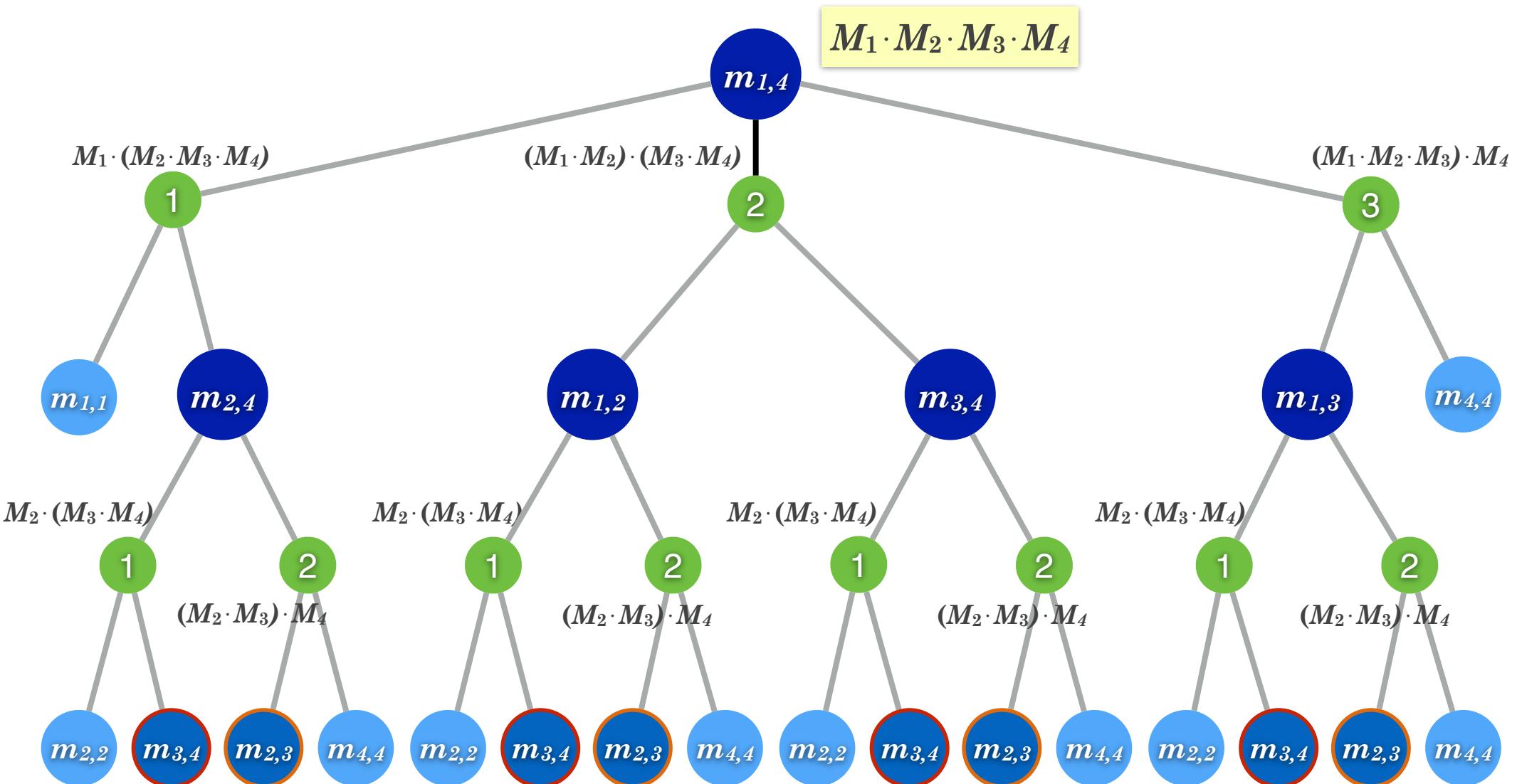
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus

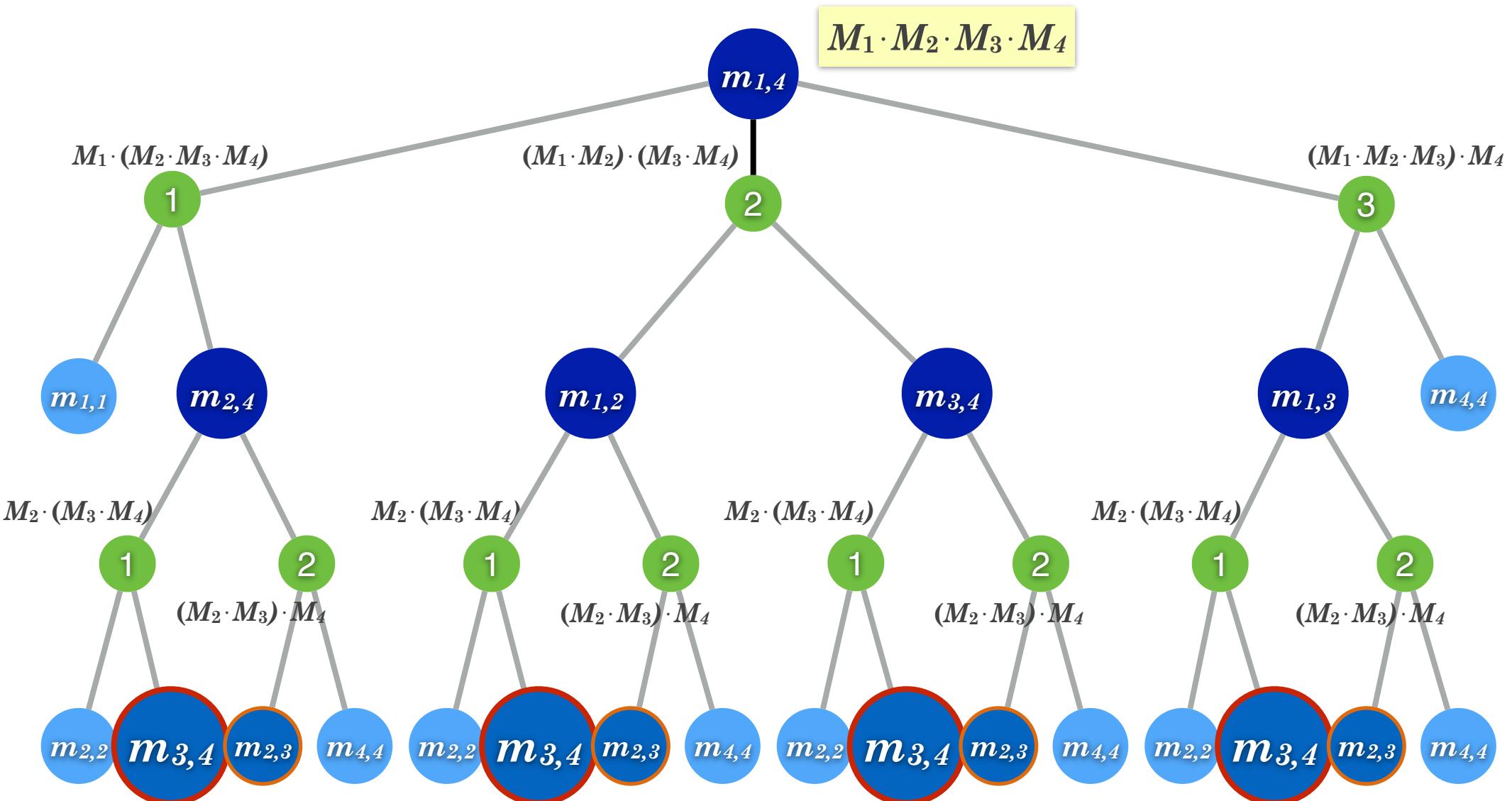
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus

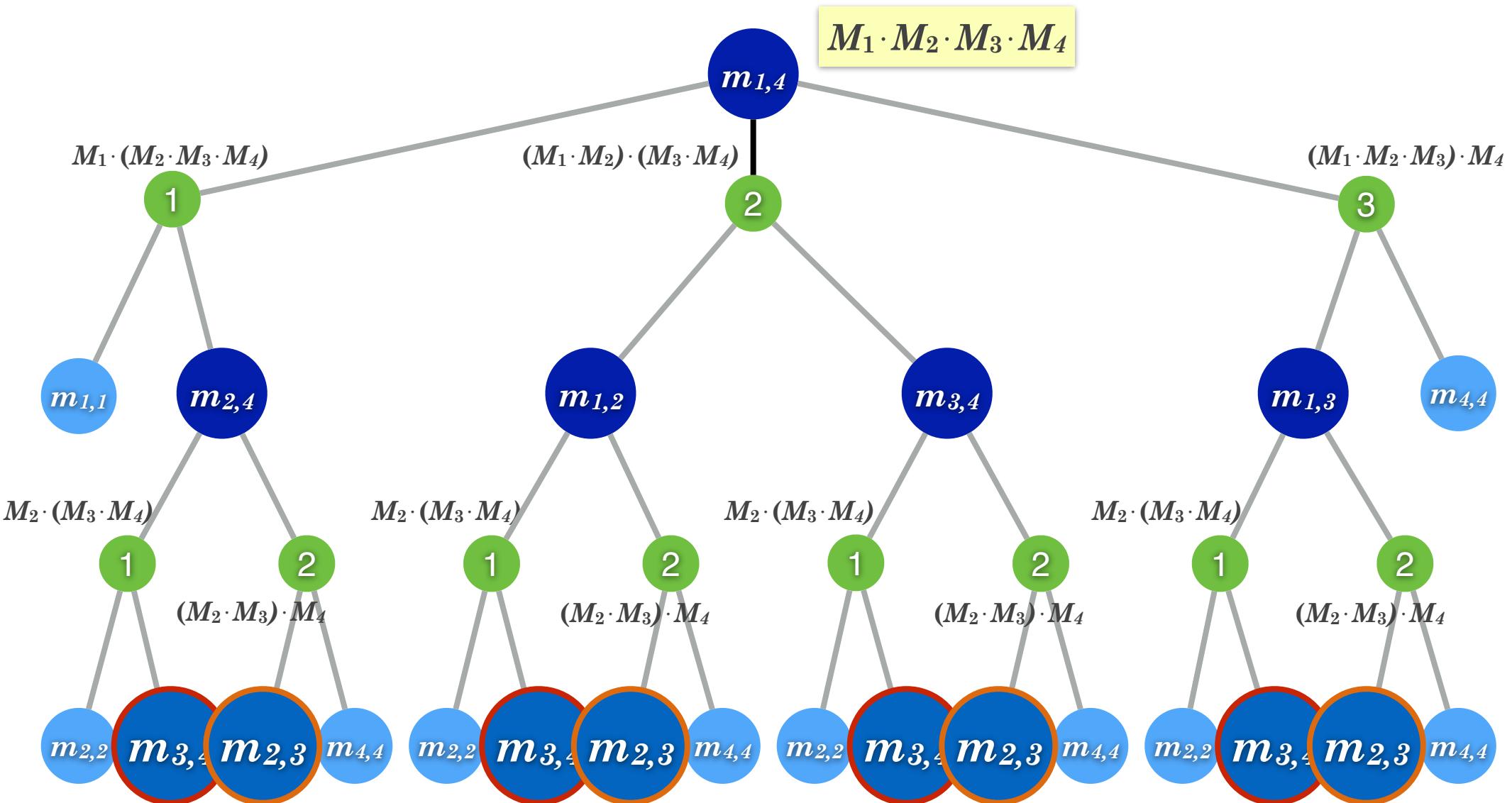
Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus

Exhaustive Search: erschöpfende Suche; **Brute Force**-Ansatz: mit roher Gewalt



Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus
(*Exhaustive Search* - erschöpfende Suche)

Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus (*Exhaustive Search* - erschöpfende Suche)
- **Aufwand:** $T(n)$ - Anzahl der Klammerungen für n Matrizen
 - **n=2:** $M_1 \cdot M_2$
Offenbar ist $T(2)=1$ - es gibt nur eine Möglichkeit zu klammern

Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus (*Exhaustive Search* - erschöpfende Suche)
- **Aufwand:** $T(n)$ - Anzahl der Klammerungen für n Matrizen
 - **n=2:** $M_1 \cdot M_2$
Offenbar ist $T(2)=1$ - es gibt nur eine Möglichkeit zu klammern
 - **n>2** $M_1 \cdot M_2 \cdot \dots \cdot M_{n-1} \cdot M_n$

Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus (***Exhaustive Search*** - erschöpfende Suche)
- **Aufwand:** $T(n)$ - Anzahl der Klammerungen für n Matrizen
 - **n=2:** $M_1 \cdot M_2$
Offenbar ist $T(2)=1$ - es gibt nur eine Möglichkeit zu klammern
 - **n>2** $M_1 \cdot M_2 \cdot \dots \cdot M_{n-1} \cdot M_n$
Betrachte (**nur**) 2 Möglichkeiten für **äusserste** Klammer:
 $(M_1 \cdot M_2 \cdot \dots \cdot M_{n-1}) \cdot M_n$ und $M_1 \cdot (M_2 \cdot \dots \cdot M_{n-1} \cdot M_n)$

Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus (***Exhaustive Search*** - erschöpfende Suche)
- **Aufwand:** $T(n)$ - Anzahl der Klammerungen für n Matrizen
 - **$n=2$:** $M_1 \cdot M_2$
Offenbar ist $T(2)=1$ - es gibt nur eine Möglichkeit zu klammern
 - **$n>2$** $M_1 \cdot M_2 \cdot \dots \cdot M_{n-1} \cdot M_n$
Betrachte (**nur**) 2 Möglichkeiten für **äusserste** Klammer:
 $(M_1 \cdot M_2 \cdot \dots \cdot M_{n-1}) \cdot M_n$ und $M_1 \cdot (M_2 \cdot \dots \cdot M_{n-1} \cdot M_n)$

Offenbar ist $T(n) \geq T(n-1)$ (* mögliche Klammerungen $M_1 - M_{n-1}$ *)
+ $T(n-1)$ (* mögliche Klammerungen $M_2 - M_n$ *)

$$\geq 2 \cdot T(n-1)$$

(ohne Beweis): es ist $T(n) \geq 2^n$

Matrixkettenprodukt - *Exhaustive Search*

- **Erster Ansatz:** Probiere alle möglichen Klammerungen aus (***Exhaustive Search*** - erschöpfende Suche)
- **Aufwand:** $T(n)$ - Anzahl der Klammerungen für n Matrizen
 - **n=2:** $M_1 \cdot M_2$
Offenbar ist $T(2)=1$ - es gibt nur eine Möglichkeit zu klammern
 - **n>2** $M_1 \cdot M_2 \cdot \dots \cdot M_{n-1} \cdot M_n$
Betrachte (**nur**) 2 Möglichkeiten für **äusserste** Klammer:

$$(M_1 \cdot M_2 \cdot \dots \cdot M_{n-1}) \cdot M_n \quad \text{und} \quad M_1 \cdot (M_2 \cdot \dots \cdot M_{n-1} \cdot M_n)$$

$$\begin{aligned}\text{Offenbar ist } T(n) &\geq T(n-1) \text{ (* mögliche Klammerungen } M_1 \text{ - } M_{n-1} *) \\ &\quad + T(n-1) \text{ (* mögliche Klammerungen } M_2 \text{ - } M_n *) \\ &\geq 2 \cdot T(n-1)\end{aligned}$$

(ohne Beweis): es ist $T(n) \geq 2^n$

Komplexität: $\Omega(2^n)$

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt
- Gegeben sei das Kettenprodukt

$$M_1 \cdot \dots \cdot M_n \boxed{M_i \in \mathbb{R}^{r_{i-1} \times r_i}}$$

von Matrizen

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt
- Gegeben sei das Kettenprodukt
$$M_1 \cdot \dots \cdot M_n \quad M_i \in \mathbb{R}^{r_{i-1} \times r_i}$$
von Matrizen
- Eine Lösung beschreibt die Reihenfolge, in der die $n-1$ Multiplikationen auszuführen sind; z.B. als Vektor (k_1, \dots, k_{n-1}) mit $k_i \in \{1, \dots, n\}$

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt
- Gegeben sei das Kettenprodukt

$$M_1 \cdot \dots \cdot M_n \boxed{M_i \in \mathbb{R}^{r_{i-1} \times r_i}}$$

von Matrizen

- Eine Lösung beschreibt die Reihenfolge, in der die $n-1$ Multiplikationen auszuführen sind; z.B. als Vektor (k_1, \dots, k_{n-1}) mit $k_i \in \{1, \dots, n\}$
- Die **äußerste** Klammerung einer **optimalen Lösung** teilt das Kettenprodukt an der Stelle $k=k_{n-1}$ in zwei Teile (die letzte durchzuführende Multiplikation ist die k -te):

$$(M_1 \cdot \dots \cdot M_k) \cdot (M_{k+1} \cdot \dots \cdot M_n)$$

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt
- Gegeben sei das Kettenprodukt

$$M_1 \cdot \dots \cdot M_n \boxed{M_i \in \mathbb{R}^{r_{i-1} \times r_i}}$$

von Matrizen

- Eine Lösung beschreibt die Reihenfolge, in der die $n-1$ Multiplikationen auszuführen sind; z.B. als Vektor (k_1, \dots, k_{n-1}) mit $k_i \in \{1, \dots, n\}$
- Die **äußerste** Klammerung einer **optimalen Lösung** teilt das Kettenprodukt an der Stelle $k=k_{n-1}$ in zwei Teile (die letzte durchzuführende Multiplikation ist die k -te):

$$(M_1 \cdot \dots \cdot M_k) \cdot (M_{k+1} \cdot \dots \cdot M_n)$$

- Offenbar kann diese Lösung nur dann optimal sein, wenn die Klammerung für die kürzeren Teilketten $M_1 \cdot \dots \cdot M_k$ und $M_{k+1} \cdot \dots \cdot M_n$ optimal gewählt wurde!

Matrixkettenprodukt - Dynamische Programmierung

- **1. Schritt:** Charakterisiere eine optimale Lösung
 - **Ziel:** Charakterisiere **optimale Lösung** so, dass sie sich aus optimalen Lösungen von kleineren Problemen zusammensetzen lässt
- Gegeben sei das Kettenprodukt

$$M_1 \cdot \dots \cdot M_n \boxed{M_i \in \mathbb{R}^{r_{i-1} \times r_i}}$$

von Matrizen

- Eine Lösung beschreibt die Reihenfolge, in der die $n-1$ Multiplikationen auszuführen sind; z.B. als Vektor (k_1, \dots, k_{n-1}) mit $k_i \in \{1, \dots, n\}$
- Die **äußerste** Klammerung einer **optimalen Lösung** teilt das Kettenprodukt an der Stelle $k=k_{n-1}$ in zwei Teile (die letzte durchzuführende Multiplikation ist die k -te):

$$(M_1 \cdot \dots \cdot M_k) \cdot (M_{k+1} \cdot \dots \cdot M_n)$$

- Offenbar kann diese Lösung nur dann optimal sein, wenn die Klammerung für die kürzeren Teilketten $M_1 \cdot \dots \cdot M_k$ und $M_{k+1} \cdot \dots \cdot M_n$ optimal gewählt wurde!
- **Idee:** kennen wir die günstigste Klammerung für die kürzeren Ketten, können wir daraus die günstigste Klammerung für die längere Kette konstruieren
(wie, erarbeiten wir in Schritt 2 ...)

Matrixkettenprodukt - Dynamische Programmierung

- **2. Schritt:** Rekursive Lösung

- **Ziel:** Definiere Lösung des großen Problems unter Zuhilfenahme der Lösungen der kleineren Probleme.

Matrixkettenprodukt - Dynamische Programmierung

• 2. Schritt: Rekursive Lösung

- **Ziel:** Definiere Lösung des großen Problems unter Zuhilfenahme der Lösungen der kleineren Probleme.
- Hier interessieren uns die minimalen Kosten, seien also $m[i,j]$ die minimalen Kosten zur Multiplikation der Kette

$$M_i \cdot \dots \cdot M_j$$

Memoization



Matrixkettenprodukt - Dynamische Programmierung

- **2. Schritt:** Rekursive Lösung

- **Ziel:** Definiere Lösung des großen Problems unter Zuhilfenahme der Lösungen der kleineren Probleme.

- Hier interessieren uns die minimalen Kosten, seien also $m[i,j]$ die minimalen Kosten zur Multiplikation der Kette

$$M_i \cdot \dots \cdot M_j$$

Memoization

- Betrachten wir wieder eine **optimale Lösung:**

$$(M_i \cdot \dots \cdot M_k) \quad \cdot \quad (M_{k+1} \cdot \dots \cdot M_j)$$

Matrixkettenprodukt - Dynamische Programmierung

• 2. Schritt: Rekursive Lösung

- **Ziel:** Definiere Lösung des großen Problems unter Zuhilfenahme der Lösungen der kleineren Probleme.
- Hier interessieren uns die minimalen Kosten, seien also $m[i,j]$ die minimalen Kosten zur Multiplikation der Kette

$$M_i \cdot \dots \cdot M_j$$

Memoization

- Betrachten wir wieder eine **optimale Lösung**:

$$(M_i \cdot \dots \cdot M_k) \quad \cdot \quad (M_{k+1} \cdot \dots \cdot M_j)$$

- und nehmen wir an, dass wir die minimalen Kosten für die **kürzeren Ketten** bereits kennen. Dann können die minimalen Kosten $m[i,j]$ für die längere Kette wie folgt berechnet werden:

$$\underbrace{(M_i \cdot M_{i+1} \cdot \dots \cdot M_k)}_{\in \mathbb{R}^{r_{i-1} \times r_k}}$$

$$\underbrace{(M_{k+1} \cdot \dots \cdot M_j)}_{\in \mathbb{R}^{r_k \times r_j}}$$

$$M_i \in \mathbb{R}^{r_{i-1} \times r_i}$$

$$m[i, k]$$

+

$$m[k + 1, j]$$

+

$\underbrace{r_{i-1} \cdot r_k \cdot r_j}_{\text{Multiplikation der Teilergebnisse}}$ Operationen

Matrixkettenprodukt - Dynamische Programmierung

- **3. Schritt:** Berechnung der optimalen Lösung

- **Ziel:** Ableitung eines Algorithmus aus Schritt 2

Es ergibt sich folgende Rekursionsgleichung:

$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & j > i \end{cases}$$

Beobachtung: Offenbar benötigt man für die Berechnung einer Kette der Länge l nur die Ergebnisse der kürzeren Produkte.

Matrixkettenprodukt - Dynamische Programmierung

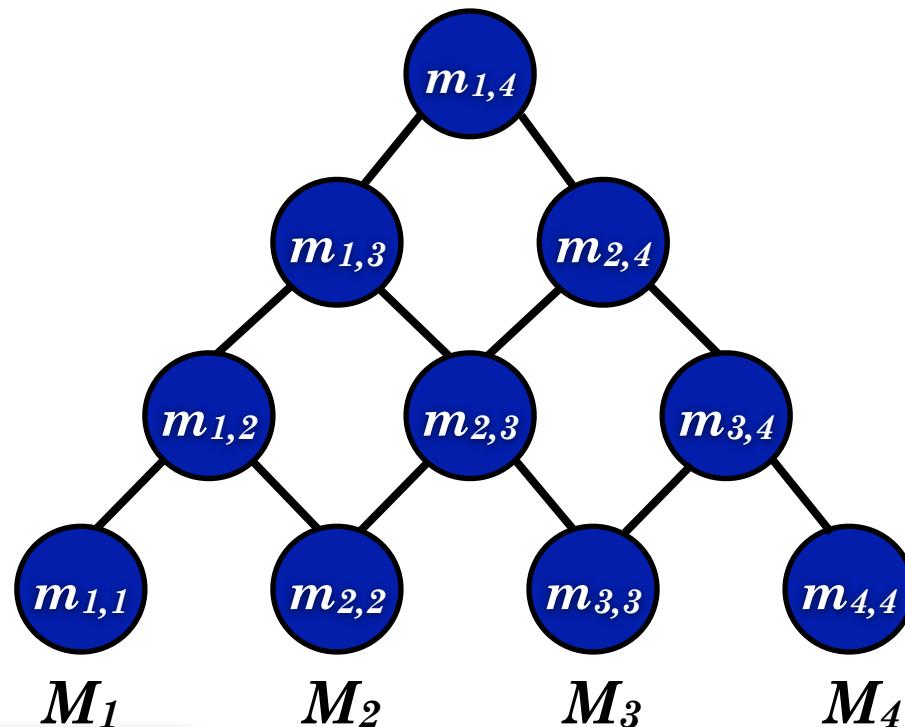
- **3. Schritt:** Berechnung der optimalen Lösung

- **Ziel:** Ableitung eines Algorithmus aus Schritt 2

Es ergibt sich folgende Rekursionsgleichung:

$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & j > i \end{cases}$$

Beobachtung: Offenbar benötigt man für die Berechnung einer Kette der Länge l nur die Ergebnisse der kürzeren Produkte.



Matrixkettenprodukt - Dynamische Programmierung

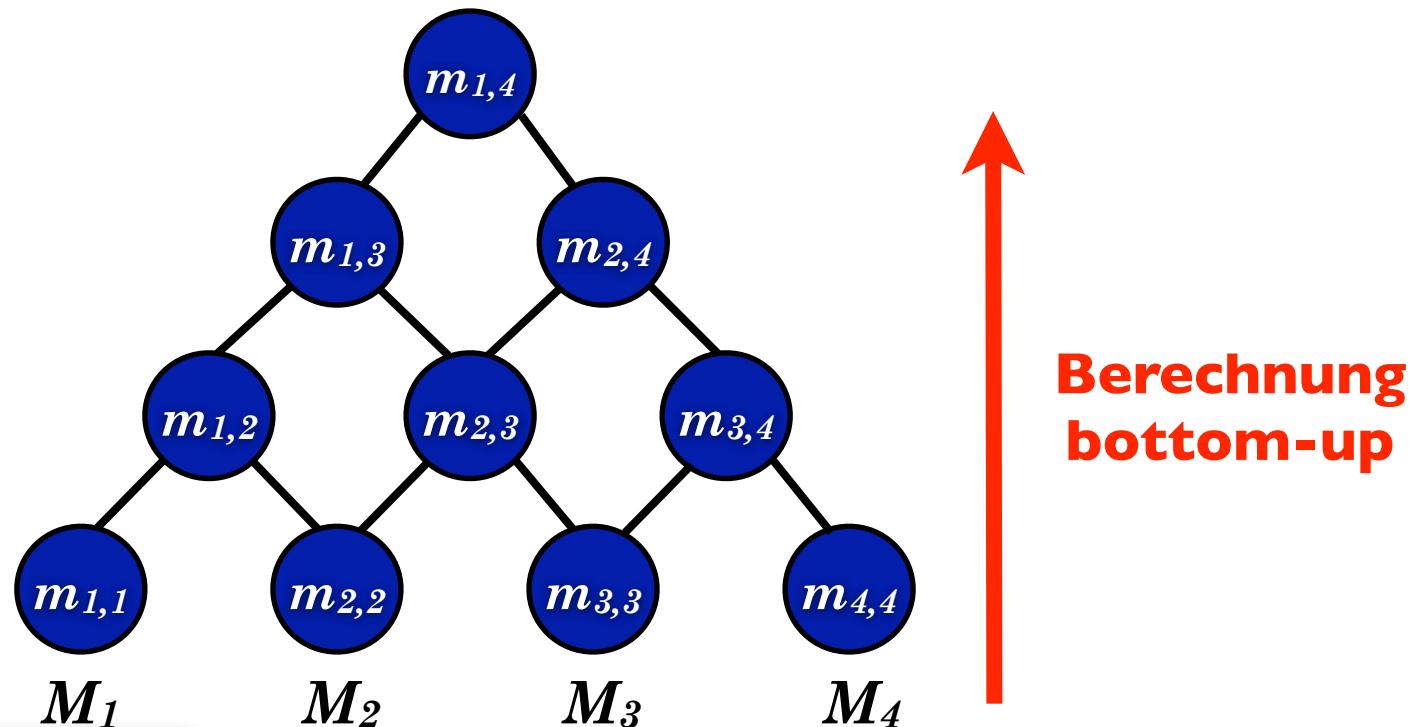
- **3. Schritt:** Berechnung der optimalen Lösung

- **Ziel:** Ableitung eines Algorithmus aus Schritt 2

Es ergibt sich folgende Rekursionsgleichung:

$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Beobachtung: Offenbar benötigt man für die Berechnung einer Kette der Länge l nur die Ergebnisse der kürzeren Produkte.



Matrixkettenprodukt - Dynamische Programmierung

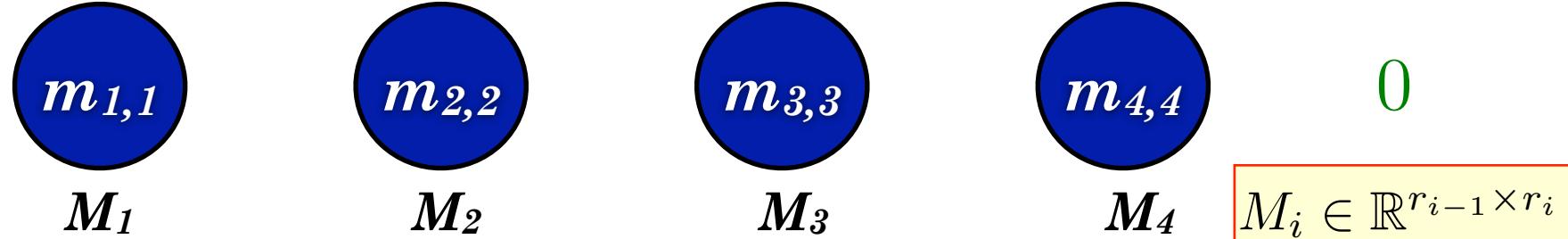
$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung



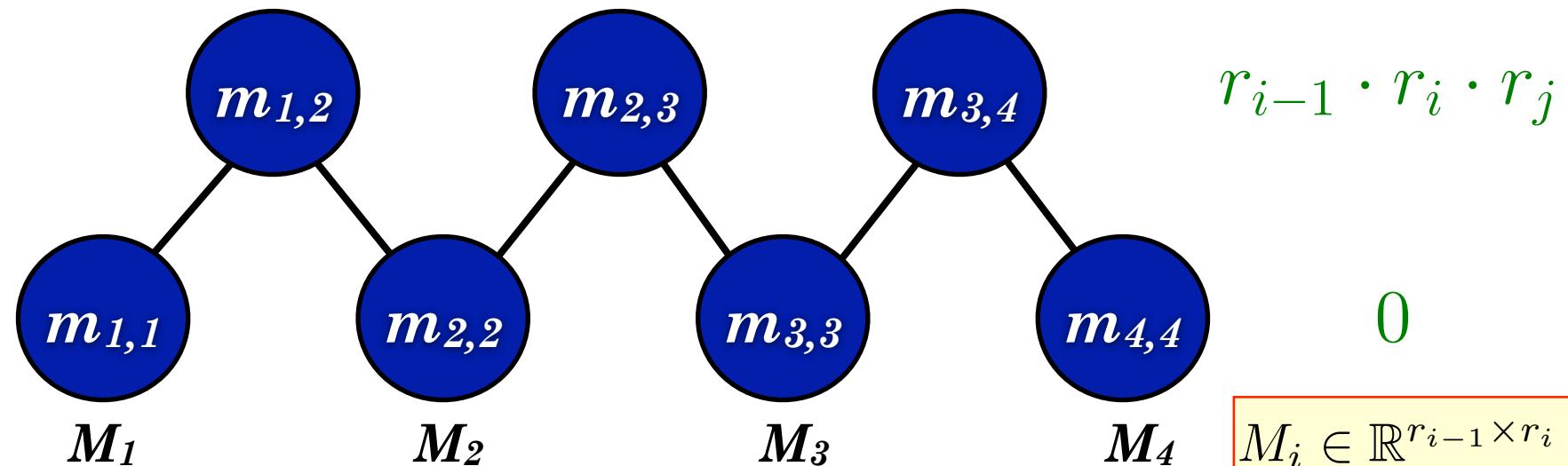
$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

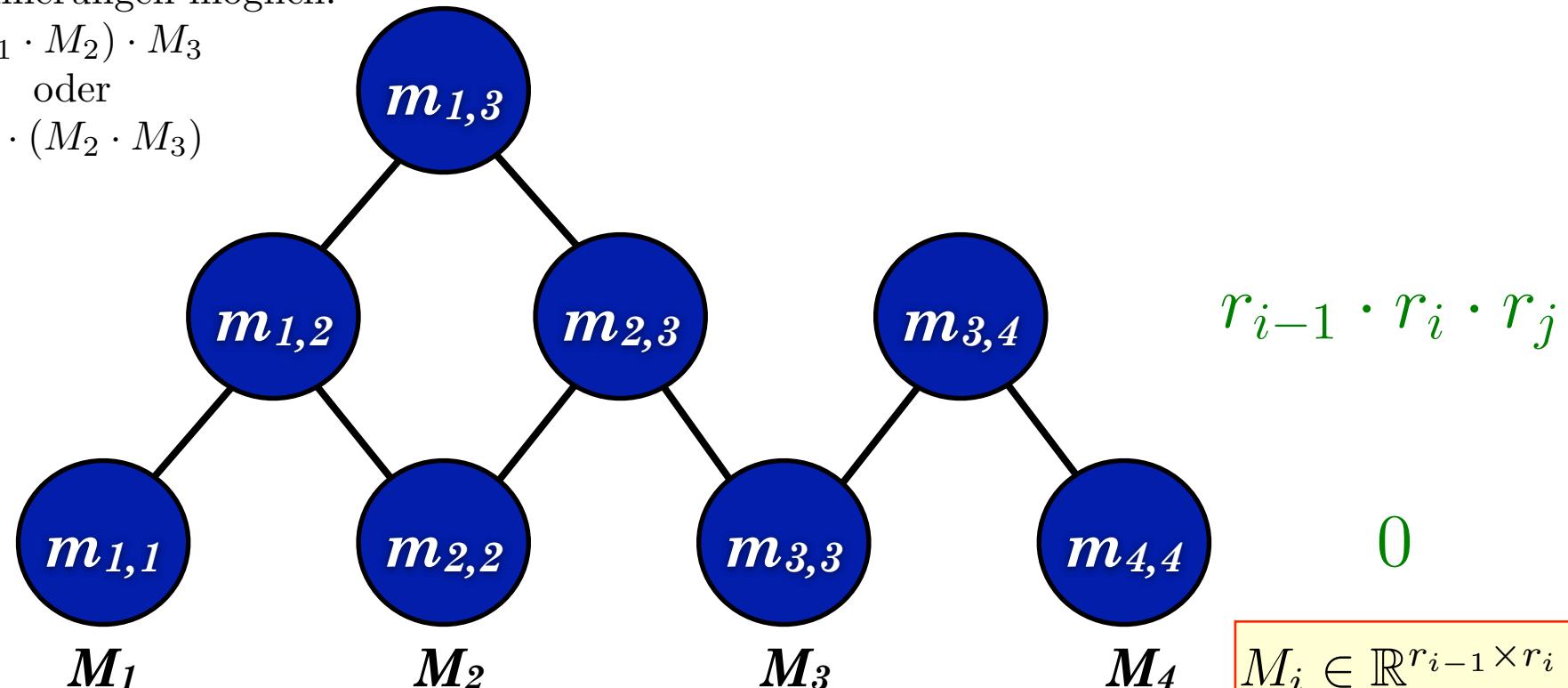
Matrixkettenprodukt - Dynamische Programmierung

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

$$M_1 \cdot (M_2 \cdot M_3)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

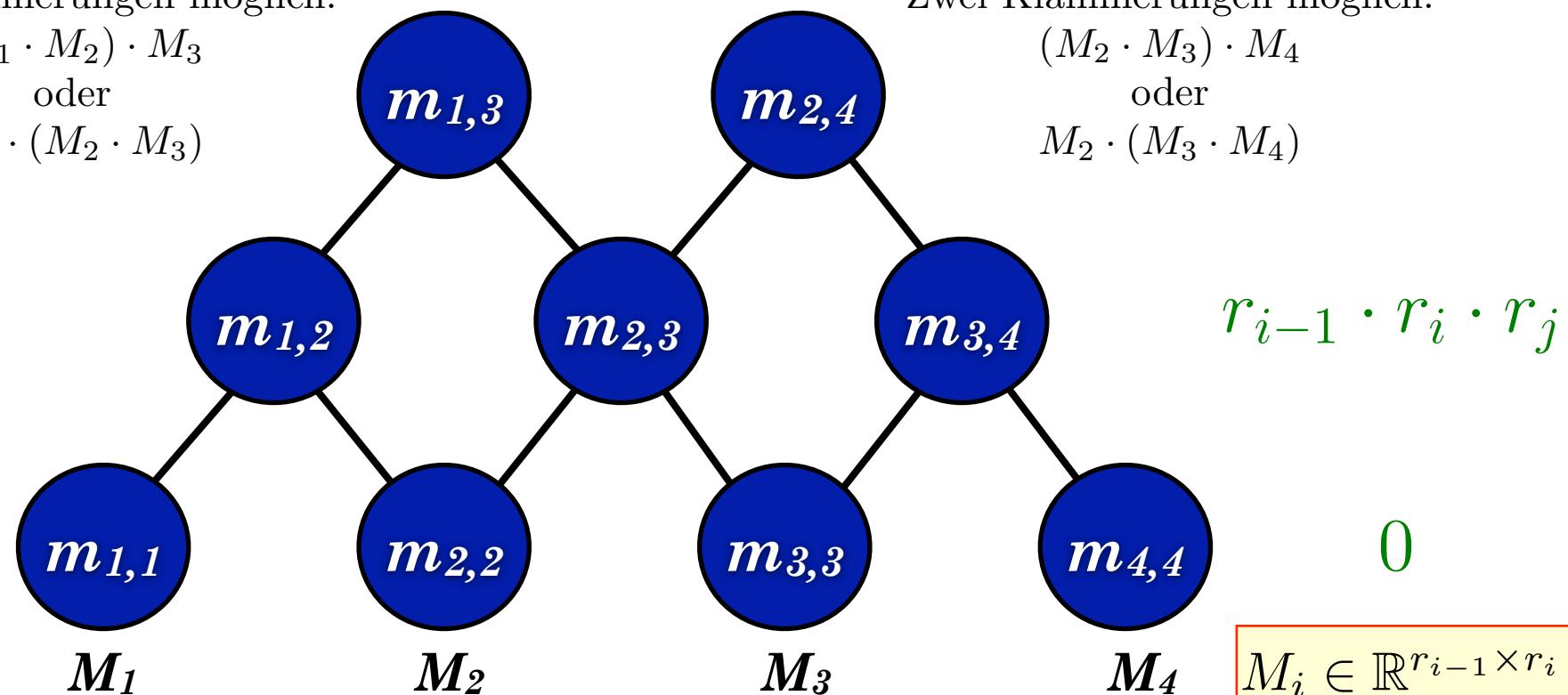
$$M_1 \cdot (M_2 \cdot M_3)$$

Zwei Klammerungen möglich:

$$(M_2 \cdot M_3) \cdot M_4$$

oder

$$M_2 \cdot (M_3 \cdot M_4)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung

Drei Möglichkeiten, die Kette in zwei Teile zu teilen:

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

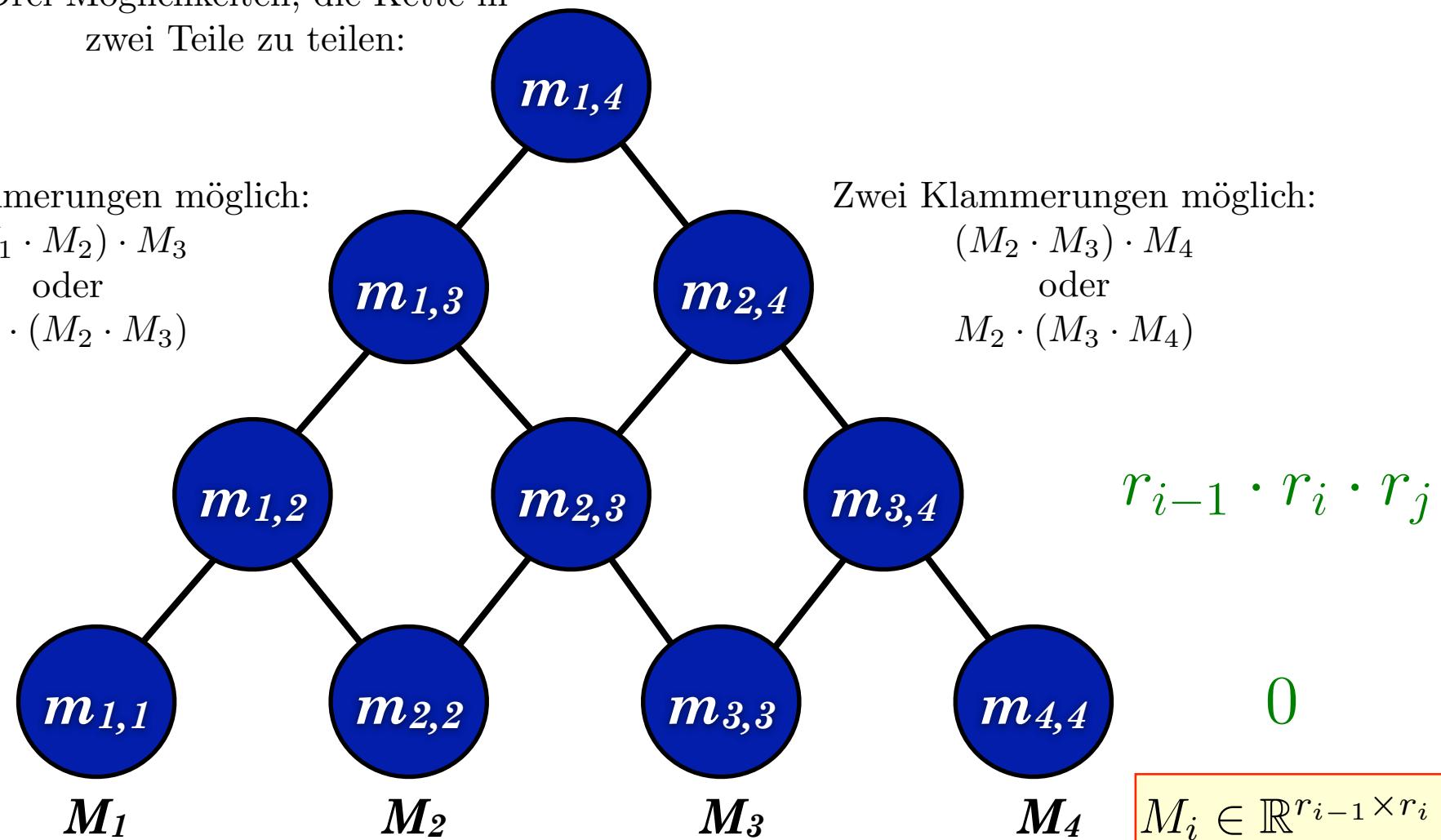
$$M_1 \cdot (M_2 \cdot M_3)$$

Zwei Klammerungen möglich:

$$(M_2 \cdot M_3) \cdot M_4$$

oder

$$M_2 \cdot (M_3 \cdot M_4)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung

Drei Möglichkeiten, die Kette in zwei Teile zu teilen:

$$1) \ M_1 \cdot (M_2 \cdot M_3 \cdot M_4) \quad k=1$$

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

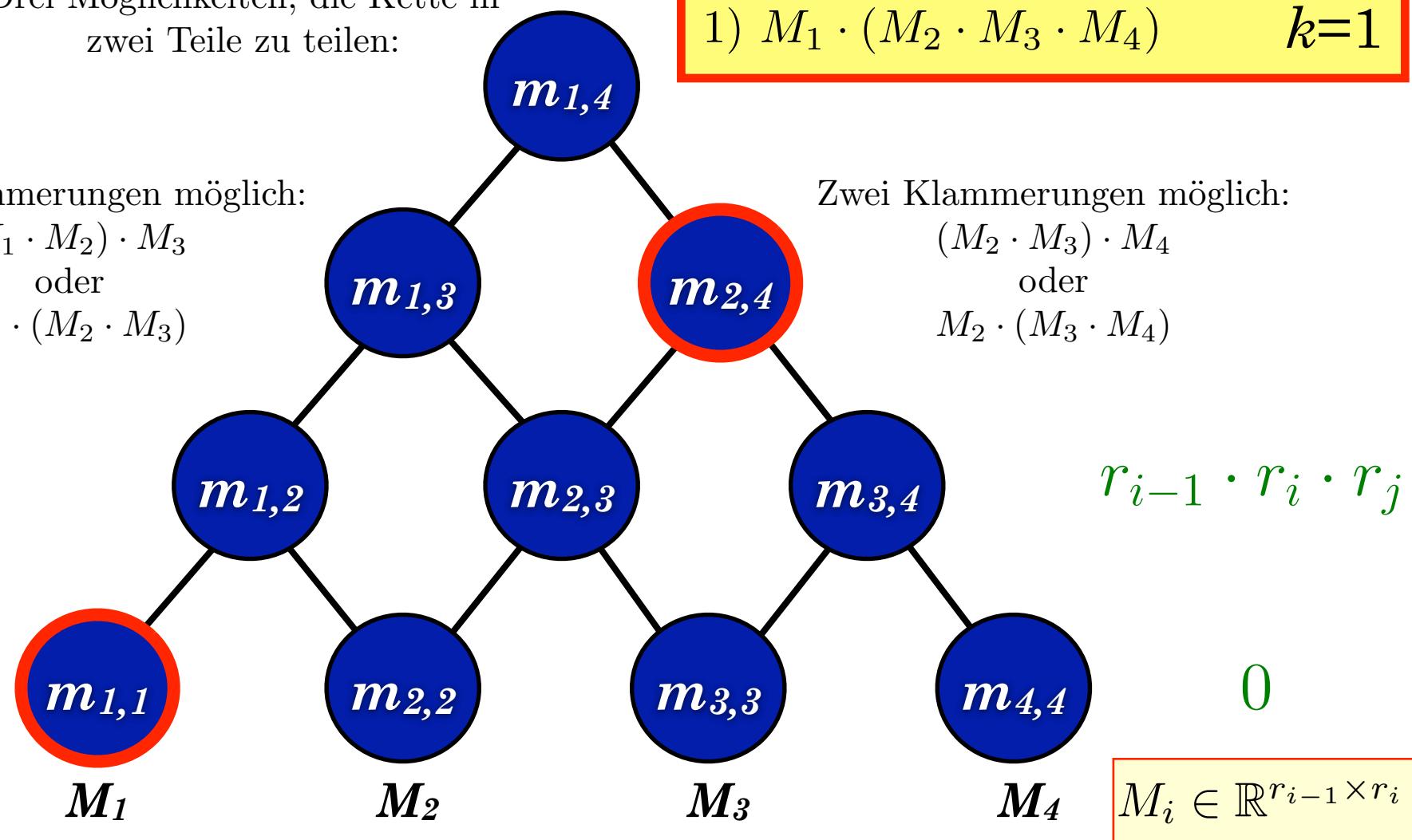
$$M_1 \cdot (M_2 \cdot M_3)$$

Zwei Klammerungen möglich:

$$(M_2 \cdot M_3) \cdot M_4$$

oder

$$M_2 \cdot (M_3 \cdot M_4)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung

Drei Möglichkeiten, die Kette in zwei Teile zu teilen:

$$2) (M_1 \cdot M_2) \cdot (M_3 \cdot M_4) \quad k=2$$

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

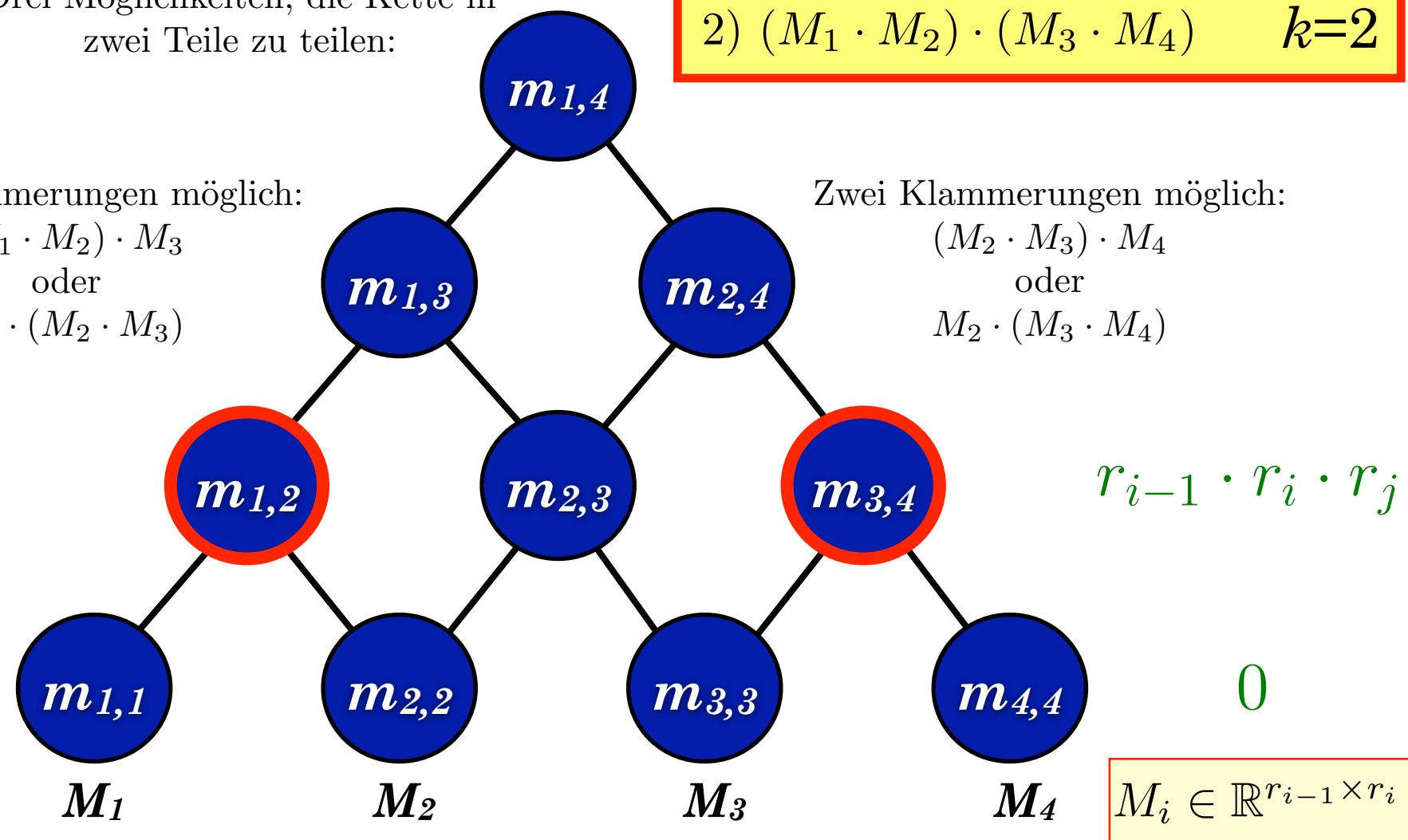
$$M_1 \cdot (M_2 \cdot M_3)$$

Zwei Klammerungen möglich:

$$(M_2 \cdot M_3) \cdot M_4$$

oder

$$M_2 \cdot (M_3 \cdot M_4)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Matrixkettenprodukt - Dynamische Programmierung

Drei Möglichkeiten, die Kette in zwei Teile zu teilen:

$$3) (M_1 \cdot M_2 \cdot M_3) \cdot M_4 \quad k=3$$

Zwei Klammerungen möglich:

$$(M_1 \cdot M_2) \cdot M_3$$

oder

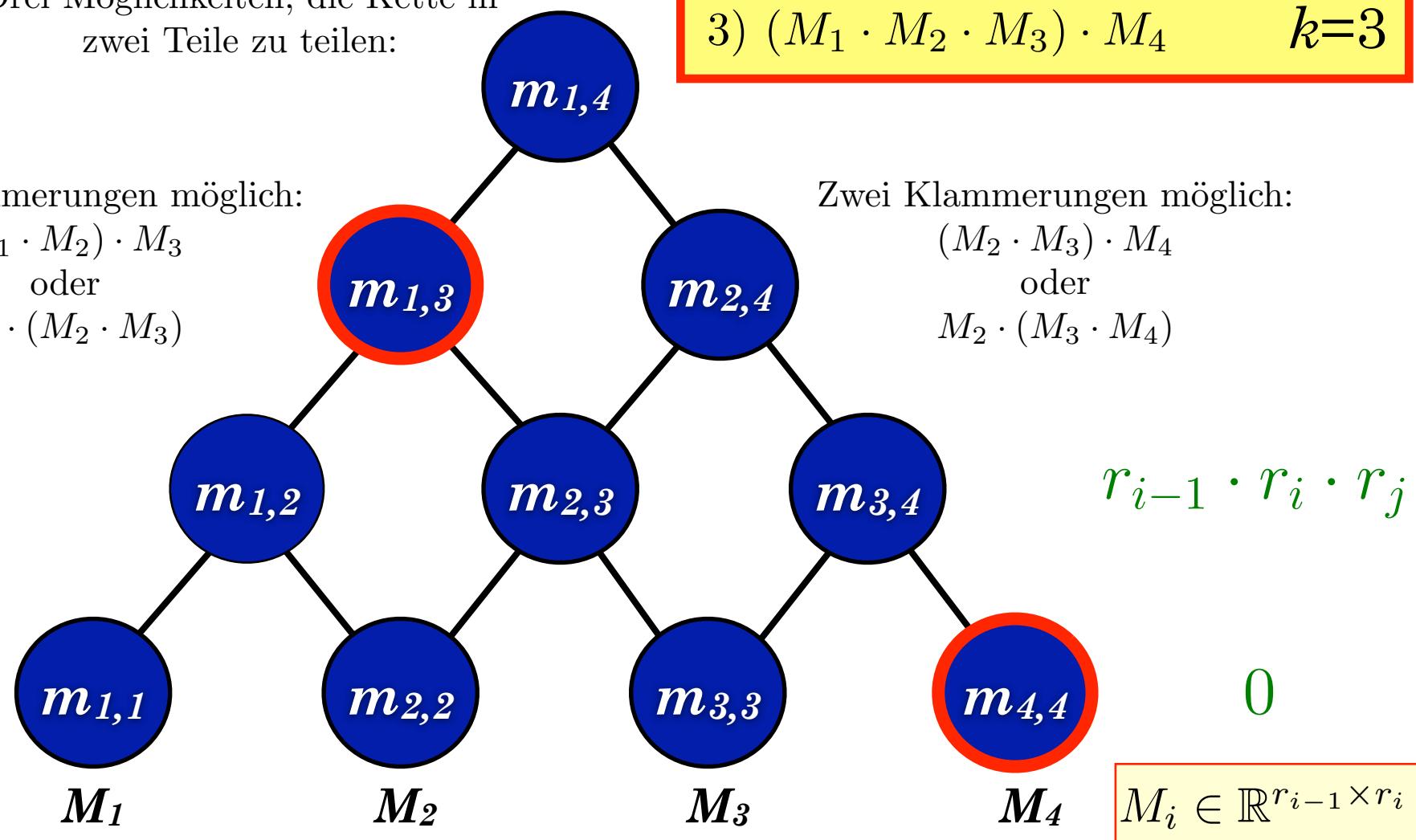
$$M_1 \cdot (M_2 \cdot M_3)$$

Zwei Klammerungen möglich:

$$(M_2 \cdot M_3) \cdot M_4$$

oder

$$M_2 \cdot (M_3 \cdot M_4)$$



$$m[i, j] = \begin{cases} 0 & \text{für } j = i \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + r_{i-1} \cdot r_k \cdot r_j\} & \text{für } j > i \end{cases}$$

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Implementierung in C++

```
void MatrixProduct(vector<int> r, int** m, int** s) {  
    int N = r.size();  
    for (int i=0 ; i<N ; ++i)  
        m[i][i]=0; // Diagonale der Kosten mit 0 belegen  
    for (int l=1 ; l<N ; ++l)          // Länge der Kette  
        for (int i=0 ; i<N-l ; ++i) { // Startposition der Kette  
            int j = i+l;           // Endposition der Kette  
            int minCost = -1, splitIndex;  
            for (int k=i ; k<j ; ++k) {  
                int cost = m[i][k] + m[k+1][j] + r[i-1] * r[k] * r[j];  
                if ((cost < minCost) || (minCost== -1)) {  
                    minCost = cost ; splitIndex = k;  
                }  
            }  
            m[i][j] = minCost;  
            s[i][j] = splitIndex;  
        }  
}
```

Dynamische Programmierung - Beispiel

B Matrix-Kettenprodukt - Implementierung in C++

```
void MatrixProduct(vector<int> r, int** m, int** s) {  
    int N = r.size();  
    for (int i=0 ; i<N ; ++i)  
        m[i][i]=0; // Diagonale der Kosten mit 0 belegen  
    for (int l=1 ; l<N ; ++l)          // Länge der Kette  
        for (int i=0 ; i<N-l ; ++i) { // Startposition der Kette  
            int j = i+l;           // Endposition der Kette  
            int minCost = -1, splitIndex;  
            for (int k=i ; k<j ; ++k) {  
                int cost = m[i][k] + m[k+1][j] + r[i-1] * r[k] * r[j];  
                if ((cost < minCost) || (minCost== -1)) {  
                    minCost = cost ; splitIndex = k;  
                }  
            }  
            m[i][j] = minCost;  
            s[i][j] = splitIndex;  
        }  
}
```

Komplexität: $O(n^3)$

III. Entwurfsmethoden

3. Entwurfsmethoden

- 3.1. Teile und Herrsche
- 3.2. Backtracking
- 3.3. Memoization
- 3.4. Dynamische Programmierung
- 3.5. Greedy-Algorithmen**

Ein Klassiker: Münzwechsel

Ein Klassiker: Münzwechsel

- **Acht Euro-Münzen** in unseren Währungssystem:



Ein Klassiker: Münzwechsel

- **Acht Euro-Münzen** in unseren Währungssystem:



- **Aufgabe:**

**Minimale Anzahl Münzen um
€ 5,42 auszuzahlen?**

Ein Klassiker: Münzwechsel

- **Acht Euro-Münzen** in unseren Währungssystem:



- **Aufgabe:**

**Minimale Anzahl Münzen um
€ 5,42 auszuzahlen?**

- **Lösung:**



Greedy-Algorithmus

Greedy-Algorithmus

- **Greedy-Strategie:**
 - Wähle immer die größte noch passende Münze

Greedy-Algorithmus

- **Greedy-Strategie:**

- Wähle immer die größte noch passende Münze

```
public static double EuroWerte[] = {1,2,5,10,20,50,100,200};  
  
public static int[] WechselGeld(double betrag, double muenzWerte[] ) {  
    int[] result = new int[ muenzWerte.length ];  
    double r = betrag * 100;  
    while (r>0) {  
        int biggestIdx = 0;  
        // Gierig sein: größte noch passende Münze suchen  
        while ((biggestIdx<muenzWerte.length) &&  
               (muenzWerte[biggestIdx]<=r)) biggestIdx++;  
        if (biggestIdx>0) --biggestIdx;  
        result[biggestIdx]++;  
        r = r - muenzWerte[biggestIdx];  
    }  
    return result;  
}
```

Greedy-Algorithmus

- **Greedy-Strategie:**

- Wähle immer die größte noch passende Münze

```
public static double EuroWerte[] = {1,2,5,10,20,50,100,200};  
  
public static int[] WechselGeld(double betrag, double muenzWerte[] ) {  
    int[] result = new int[ muenzWerte.length ];  
    double r = betrag * 100;  
    while (r>0) {  
        int biggestIdx = 0;  
        // Gierig sein: größte noch passende Münze suchen  
        while ((biggestIdx<muenzWerte.length) &&  
               (muenzWerte[biggestIdx]<=r)) biggestIdx++;  
        if (biggestIdx>0) --biggestIdx;  
        result[biggestIdx]++;  
        r = r - muenzWerte[biggestIdx];  
    }  
    return result;  
}
```

- **Wirklich korrekt?**
- **Funktioniert das für beliebige Münzsätze?**

Korrektheit

- **Zunächst formalisieren wir die Aufgabe:**

D Münzwechselproblem in der €-Zone

Sei $\mathcal{C} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ der €-Münzsatz mit Cent-Werten $v(c_1) = 1, v(c_2) = 2, v(c_3) = 5, v(c_4) = 10, v(c_5) = 20, v(c_6) = 50, v(c_7) = 100$ und $v(c_8) = 200$.

Eine Münzfolge ist eine Multimenge $m : \mathcal{C} \rightarrow \mathbb{N}_0$ (m gibt an, wie oft eine Münze in der Münzfolge auftaucht). Der Wert einer Münzfolge m ist

$$v(m) = \sum_{i=1}^8 m(i) \cdot v(c_i)$$

und sie enthält

$$n(m) = \sum_{i=1}^8 m(i)$$

viele Münzen. Die Menge aller Münzfolgen bezeichnen wir mit \mathcal{M} .

Sei $v \in \mathbb{N}$ ein beliebiger Geldbetrag in Cent. Dann besteht das **Münzwechselproblem** im Auffinden einer Münzfolge m , so dass $v(m) = v$ und $n(m)$ minimal ist; formal

$$v(m) = v \quad \wedge \quad n(m) = \min_{m' \in \mathcal{M}} \{n(m') \mid v(m') = v\}$$

Charakterisierung der optimalen Lösung - 1

Charakterisierung der optimalen Lösung - 1

- **In einer optimalen Lösung können Münzen nicht beliebig oft vorkommen**

Charakterisierung der optimalen Lösung - 1

- In einer optimalen Lösung können Münzen nicht beliebig oft vorkommen
- Beispiele:
 - zwei 1-Cent-Stücke wird man in der optimalen Lösung nicht finden - sie könnten durch ein 2-Cent-Stück ersetzt werden:



Charakterisierung der optimalen Lösung - 1

- In einer optimalen Lösung können Münzen nicht beliebig oft vorkommen
- Beispiele:
 - zwei 1-Cent-Stücke wird man in der optimalen Lösung nicht finden - sie könnten durch ein 2-Cent-Stück ersetzt werden:



- zwei 2-Cent-Münzen wird man nur vorfinden, wenn die Lösung keine 1-Cent-Münze enthält, sonst könnte man die drei Münzen durch eine 5-Cent-Münze ersetzen und erhielte eine bessere Lösung:



Charakterisierung der optimalen Lösung - 2

- Insgesamt erfüllt eine optimale Lösung folgende Eigenschaften:

k	v(c_k)	Bedingung
1	1	$m(c_1) \leq 1$
2	2	$m(c_2)+m(c_1) \leq 2$
3	5	$m(c_3) \leq 1$
4	10	$m(c_4) \leq 1$
5	20	$m(c_5)+m(c_4) \leq 2$
6	50	$m(c_6) \leq 1$
7	100	$m(c_7) \leq 1$
8	200	-

Charakterisierung der optimalen Lösung - 2

- Insgesamt erfüllt eine optimale Lösung folgende Eigenschaften:

k	v(c_k)	Bedingung
1	1	$m(c_1) \leq 1$
2	2	$m(c_2) + m(c_1) \leq 2$
3	5	$m(c_3) \leq 1$
4	10	$m(c_4) \leq 1$
5	20	$m(c_5) + m(c_4) \leq 2$
6	50	$m(c_6) \leq 1$
7	100	$m(c_7) \leq 1$
8	200	-

- **Konsequenz:** Eine optimale Lösung die nur Münzen c_1, \dots, c_j enthält ($j < 8$), kann einen bestimmten Maximalbetrag \max_{j+1} nicht übersteigen!

Charakterisierung der optimalen Lösung - 3

- Zusammenfassend erhalten wir:

k	v(c_k)	Bedingung	max_k
1	1	$m(c_1) \leq 1$	0
2	2	$m(c_2) + m(c_1) \leq 2$	1
3	5	$m(c_3) \leq 1$	4
4	10	$m(c_4) \leq 1$	9
5	20	$m(c_5) + m(c_4) \leq 2$	19
6	50	$m(c_6) \leq 1$	49
7	100	$m(c_7) \leq 1$	99
8	200	-	199

Charakterisierung der optimalen Lösung - 3

- Zusammenfassend erhalten wir:

k	v(c_k)	Bedingung	max_k
1	1	$m(c_1) \leq 1$	0
2	2	$m(c_2) + m(c_1) \leq 2$	1
3	5	$m(c_3) \leq 1$	4
4	10	$m(c_4) \leq 1$	9
5	20	$m(c_5) + m(c_4) \leq 2$	19
6	50	$m(c_6) \leq 1$	49
7	100	$m(c_7) \leq 1$	99
8	200	-	199

Beachte:
max_k<v(c_k)

Charakterisierung der optimalen Lösung - 3

- Zusammenfassend erhalten wir:

k	v(c_k)	Bedingung	max_k
1	1	$m(c_1) \leq 1$	0
2	2	$m(c_2) + m(c_1) \leq 2$	1
3	5	$m(c_3) \leq 1$	4
4	10	$m(c_4) \leq 1$	9
5	20	$m(c_5) + m(c_4) \leq 2$	19
6	50	$m(c_6) \leq 1$	49
7	100	$m(c_7) \leq 1$	99
8	200	-	199

Beachte:
max_k<v(c_k)

- Damit haben wir alle Voraussetzungen zusammen, um die Korrektheit des Greedy-Algorithmus zu beweisen ...

S Optimalität des Greedy-Algorithmus für €-Münzwechsel

Seien $v \in \mathbb{N}$, m eine optimale Lösung des Münzwechselproblems zu v und g eine Lösung des Greedy-Algorithmus für v . Dann gilt $m=g$.

...

S Optimalität des Greedy-Algorithmus für €-Münzwechsel

Seien $v \in \mathbb{N}$, m eine optimale Lösung des Münzwechselproblems zu v und g eine Lösung des Greedy-Algorithmus für v . Dann gilt $m=g$.

Beweis: Wir beweisen per Widerspruch. Angenommen $m \neq g$. Dann muss es einen größten Index $j \in [1, 8]$ geben, so dass

$$m(j) \neq g(j) \quad \wedge \quad \forall i \in (j, 8] : m(i) = g(i)$$

und

$$\sum_{i=1}^j m(i) \cdot v(c_i) = \sum_{i=1}^j g(i) \cdot v(c_i)$$

Offenbar können wir $g(j) < m(j)$ ausschließen, denn der Greedy-Algorithmus wählt immer die betragsmäßig größte Münze, die noch passt - sei also $g(j) > m(j)$.

...

Beweis (*Fortsetzung*):

Aus unseren Beobachtungen über optimale Lösungen wissen wir einerseits:

$$\sum_{i=1}^{j-1} m(i) \cdot v(c_i) < v(c_j)$$

(Anmerkung: man vergewissere sich, dass $\max_j < v(c_j)$ gilt); andererseits folgt aus $g(j) > m(j)$:

$$g(j) \cdot v(c_j) - m(j) \cdot v(c_j) \geq v(c_j)$$

woraus sich insgesamt $v(m) \neq v(g)$ ergibt (Anmerkung: $v(c_j)$ kann die optimale Lösung unter Verwendung der Münzen $c_j - 1, \dots, c_1$ nicht kompensieren).

Das kann aber nicht sein, da m eine optimale Lösung für v war. Folglich muss unsere Annahme $m \neq g$ falsch gewesen sein!

Grenzen der Greedy-Methode - 1

- Funktioniert der Algorithmus auch für Briefmarken?



Grenzen der Greedy-Methode - 1

- Funktioniert der Algorithmus auch für Briefmarken?



- Für 80 Pfennig findet der Greedy-Algorithmus **keine Lösung!**

Grenzen der Greedy-Methode - 2

- **Greedy liefert nicht für jeden Münzsatz die optimale Lösung**

Grenzen der Greedy-Methode - 2

- **Greedy liefert nicht für jeden Münzsatz die optimale Lösung**
- **Betrachte Satz aus 3 Münzen c_1, c_2 und c_3 mit**
 - $v(c_1)=1$
 - $v(c_2)=k$
 - $v(c_3)=2 \cdot v(c_2) + 1$

Grenzen der Greedy-Methode - 2

- Greedy liefert nicht für jeden Münzsatz die optimale Lösung
- Betrachte Satz aus 3 Münzen c_1, c_2 und c_3 mit
 - $v(c_1)=1$
 - $v(c_2)=k$
 - $v(c_3)=2 \cdot v(c_2) + 1$
- Für den Betrag $n=3 \cdot v(c_2)$ kommt man offenbar mit drei Münzen aus c_2 aus; Greedy liefert k Münzen:
 - $1 \cdot c_3$ - Restbetrag: $v(c_2)-1$
 - $(k-1) \cdot c_1$ - Restbetrag: 0

Grenzen der Greedy-Methode - 2

- Greedy liefert nicht für jeden Münzsatz die optimale Lösung
- Betrachte Satz aus 3 Münzen c_1, c_2 und c_3 mit
 - $v(c_1)=1$
 - $v(c_2)=k$
 - $v(c_3)=2 \cdot v(c_2) + 1$
- Für den Betrag $n=3 \cdot v(c_2)$ kommt man offenbar mit drei Münzen aus c_2 aus; Greedy liefert k Münzen:
 - $1 \cdot c_3$ - Restbetrag: $v(c_2)-1$
 - $(k-1) \cdot c_1$ - Restbetrag: 0
- Folglich ist Greedy-Algorithmus hier **beliebig schlecht!**

Grenzen der Greedy-Methode - 2

- Greedy liefert nicht für jeden Münzsatz die optimale Lösung
- Betrachte Satz aus 3 Münzen c_1, c_2 und c_3 mit
 - $v(c_1)=1$
 - $v(c_2)=k$
 - $v(c_3)=2 \cdot v(c_2) + 1$
- Für den Betrag $n=3 \cdot v(c_2)$ kommt man offenbar mit drei Münzen aus c_2 aus; Greedy liefert k Münzen:
 - $1 \cdot c_3$ - Restbetrag: $v(c_2)-1$
 - $(k-1) \cdot c_1$ - Restbetrag: 0
- Folglich ist Greedy-Algorithmus hier **beliebig schlecht!**

Wir kommen später auf Greedy-Algorithmen zurück
... um besser zu verstehen, wann der Ansatz funktioniert