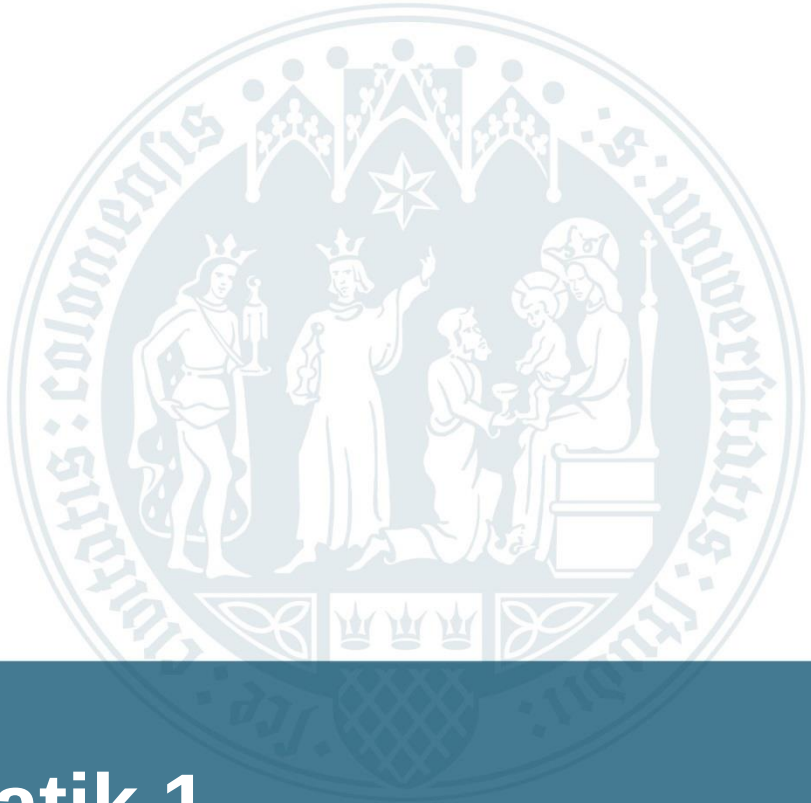
Universität zu Köln



Grundzüge der Informatik 1

Vorlesung 9

Überblick

Überblick

- Wiederholung
  - Matrix Multiplikation (verbesserter Algorithmus)
  - Auflösen von Laufzeitrekursionen
- Dynamische Programmierung
  - Laufzeit rekursive Berechnung der Fibonacci-Zahlen
  - Verbesserung durch Speichern der Lösungen
  - Iterative Lösung
  - Prinzip der dynamischen Programmierung
  - Ein erstes einfaches Beispiel

Teile & Herrsche Prinzip - Matrixmultiplikation

Matrixmultiplikation

- $\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$

Trick wie bei Integer Multiplikation

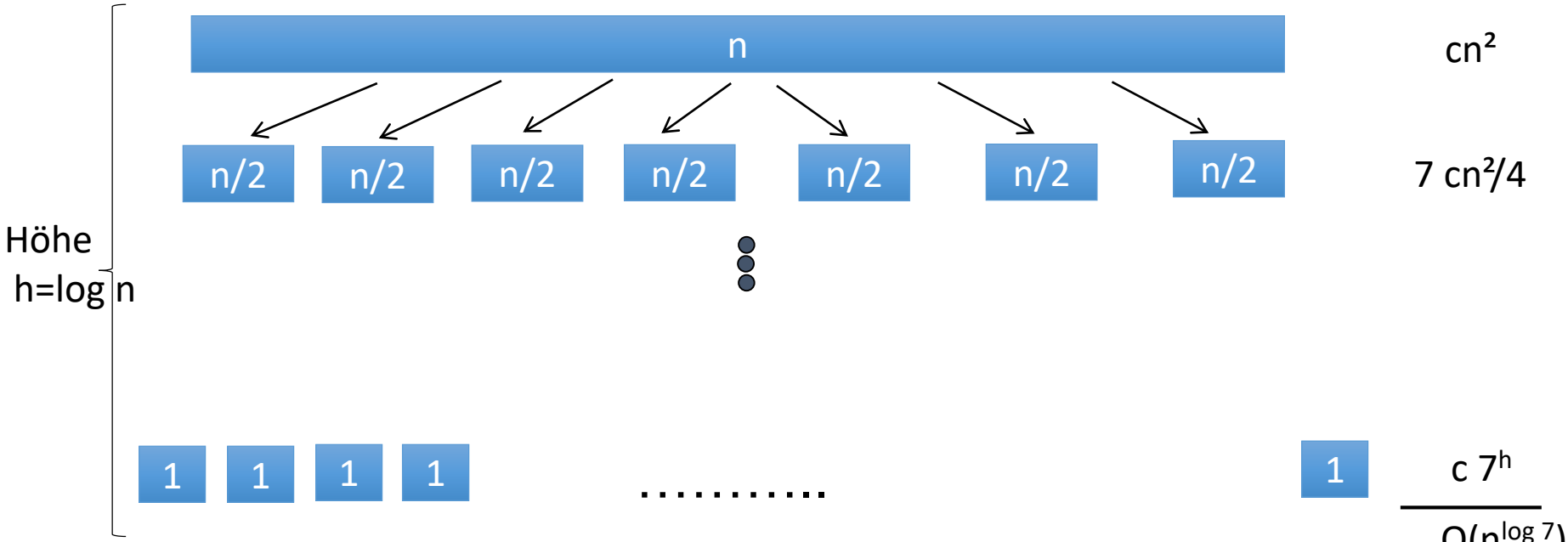
- $P_1 = A \cdot (F-H)$
- $P_2 = (A+B) \cdot H$
- $P_3 = (C+D) \cdot E$
- $P_4 = D \cdot (G-E)$
- $P_5 = (A+D) \cdot (E+H)$
- $P_6 = (B-D) \cdot (G+H)$
- $P_7 = (A-C) \cdot (E+F)$

$AE + BG = P_4 + P_5 + P_6 - P_2$
$AF + BH = P_1 + P_2$
$CE + DG = P_3 + P_4$
$AF + BH = P_1 + P_5 - P_3 - P_7$

- Nur 7 Multiplikationen!!!!!

Teile & Herrsche Prinzip - Matrixmultiplikation

- Auflösen von $T(n) \leq 7\,T(n/2) + cn^2$ (Intuition)
- $T(1) = c$



Teile & Herrsche Prinzip - Matrixmultiplikation

Satz 8.1 (Algorithmus von Strassen)

- Zwei $n \times n$ Matrizen können mit Hilfe des Teile & Herrsche Verfahrens in $O(n^{2.81})$ Zeit multipliziert werden.

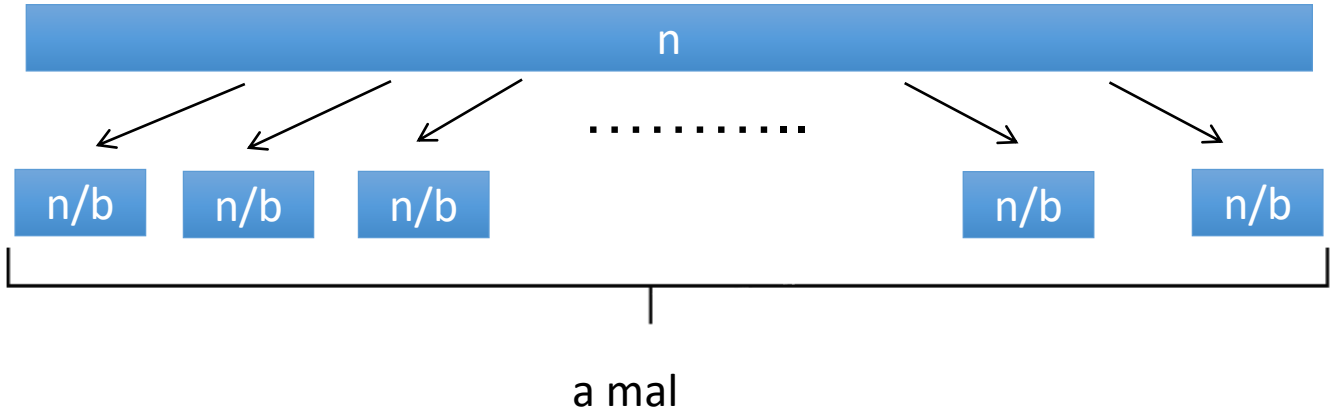
Beweis

- Laufzeit und Korrektheit können leicht per Induktion gezeigt werden.

Teile & Herrsche Prinzip
- Auflösen von Rekursionsgleichungen

Rekursionsgleichung:

- Sei T(n) = a T(n/b) + f(n) und T(1) = 1



Teile & Herrsche Prinzip
- Auflösen von Rekursionen

Satz 8.2

- Seien a ≥ 1 und b ≥ 2 ganzzahlige Konstanten und f: $\mathbb{N}_0 \to \mathbb{N}_0$.
- Sei $T(n) \leq \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \text{, für } n > 1 \\ f(1) & \text{, für } n = 1 \end{cases}$

- Dann gilt:

  (1) wenn f(n) = a f(n/b), dann ist T(n) = O(f(n) log n)

  (2) wenn f(n) ≥ γ a f(n/b) für eine Konstante γ>1, dann gilt T(n) = O(f(n))

  (3) wenn f(n) ≤ γ a f(n/b) für eine Konstante 0<γ<1, dann gilt T(n) = O($a^{\log_b n}$)

Dynamische Programmierung

```
Fib2(n)
1.  if n=1 then return 1
2.  if n=2 then return 1
3.  return Fib2(n-1) + Fib2(n-2)
```

Fibonacci Zahlen

- Fib(1)=1
- Fib(2)=1
- Fib(n) = Fib(n-1) + Fib(n-2)

Dynamische Programmierung

```
Fib2(n)
1.  if n=1 then return 1
2.  if n=2 then return 1
3.  return Fib2(n-1) + Fib2(n-2)
```

Lemma 9.1

- Fib2(n) hat eine Laufzeit von $\Omega(1.6^n)$.

Beweis

- Sei T(n) die Laufzeit von Fib2(n)
- Wir zeigen per Induktion, dass $T(n) \geq 1.6^n$
- Induktionsanfang:
- $T(1) = 2 > 1.6 = 1.6^1$
- $T(2) = 3 > 2.56 = 1.6^2$

Dynamische Programmierung

```
Fib2(n)
1. if n=1 then return 1
2. if n=2 then return 1
3. return Fib2(n-1) + Fib2(n-2)
```

Lemma 9.1

- Fib2(n) hat eine Laufzeit von $\Omega(1.6^n)$.

Beweis

- Induktionsannahme:
- Die Laufzeit von Fib2(m) ist mindestens 1.6^m für alle m<n.
- Induktionsschluss:
- Sei n>2. Es gilt $T(n) \geq T(n-1) + T(n-2) \geq 1.6^{n-1} + 1.6^{n-2}$

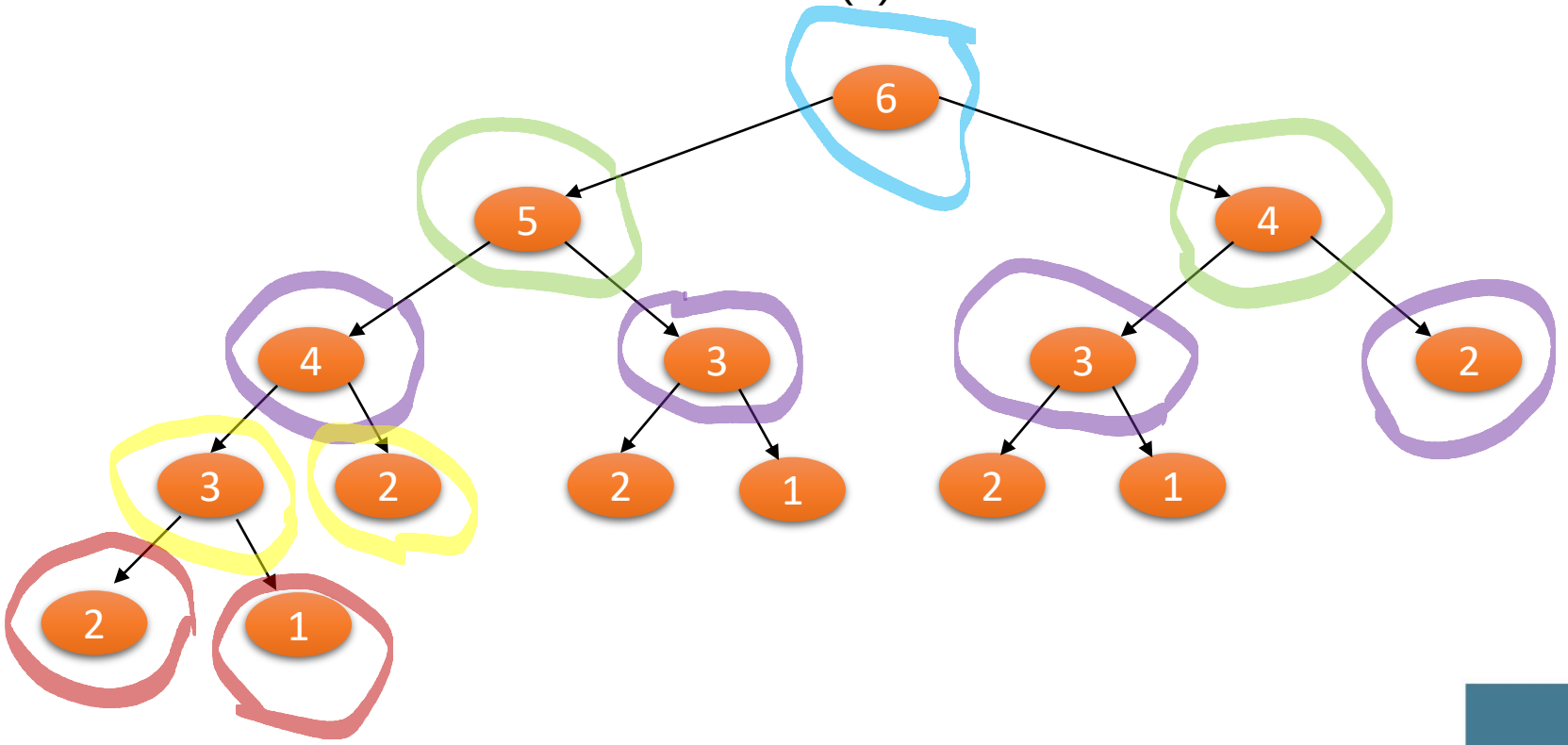
$$= (1.6 + 1) \cdot 1.6^{n-2} = 2.6 \cdot 1.6^{n-2}$$

$$\geq 1.6^2 \cdot 1.6^{n-2} = 1.6^n$$

Dynamische Programmierung

Warum ist die Laufzeit so schlecht?

- Betrachte Rekursionsbaum von Fib2(6)

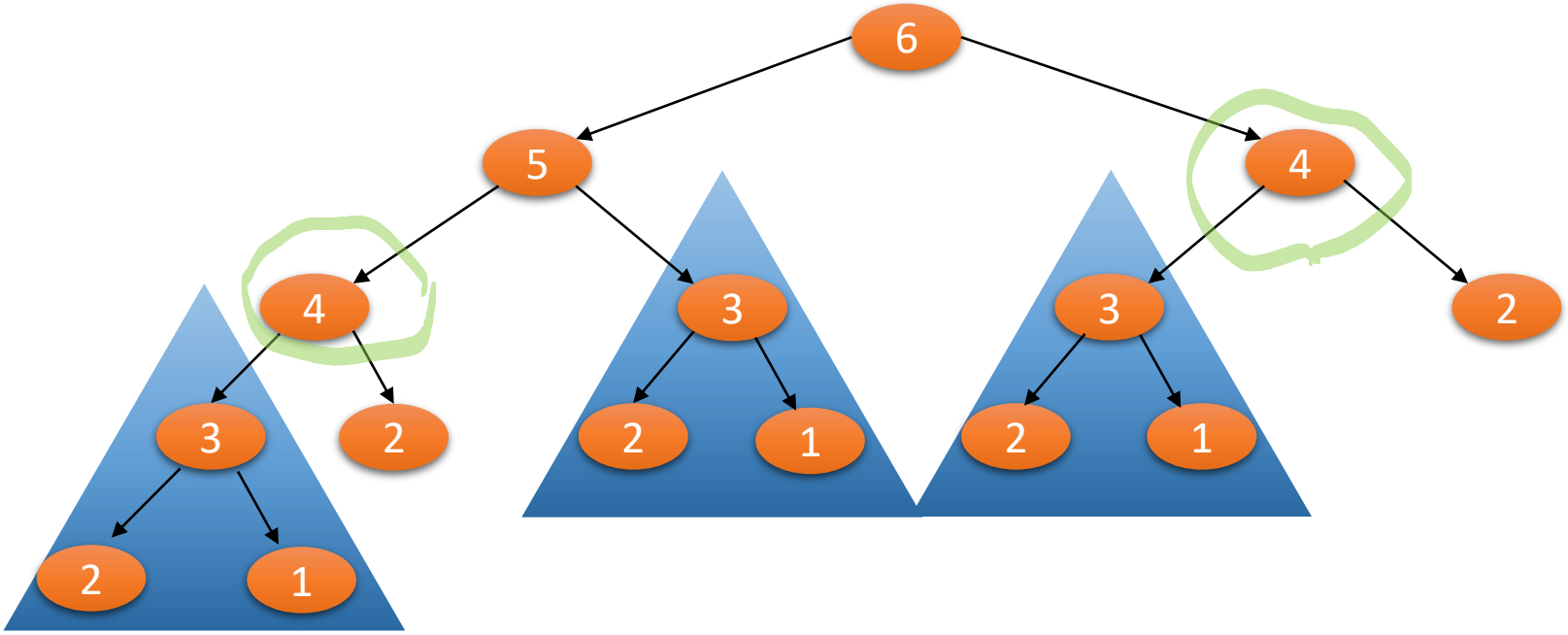


Dynamische Programmierung

Warum ist die Laufzeit so schlecht?

- Betrachte Rekursionsbaum von Fib2(6)

Bei der Berechnung von Fib2(6) wird Fib2(3) dreimal aufgerufen!

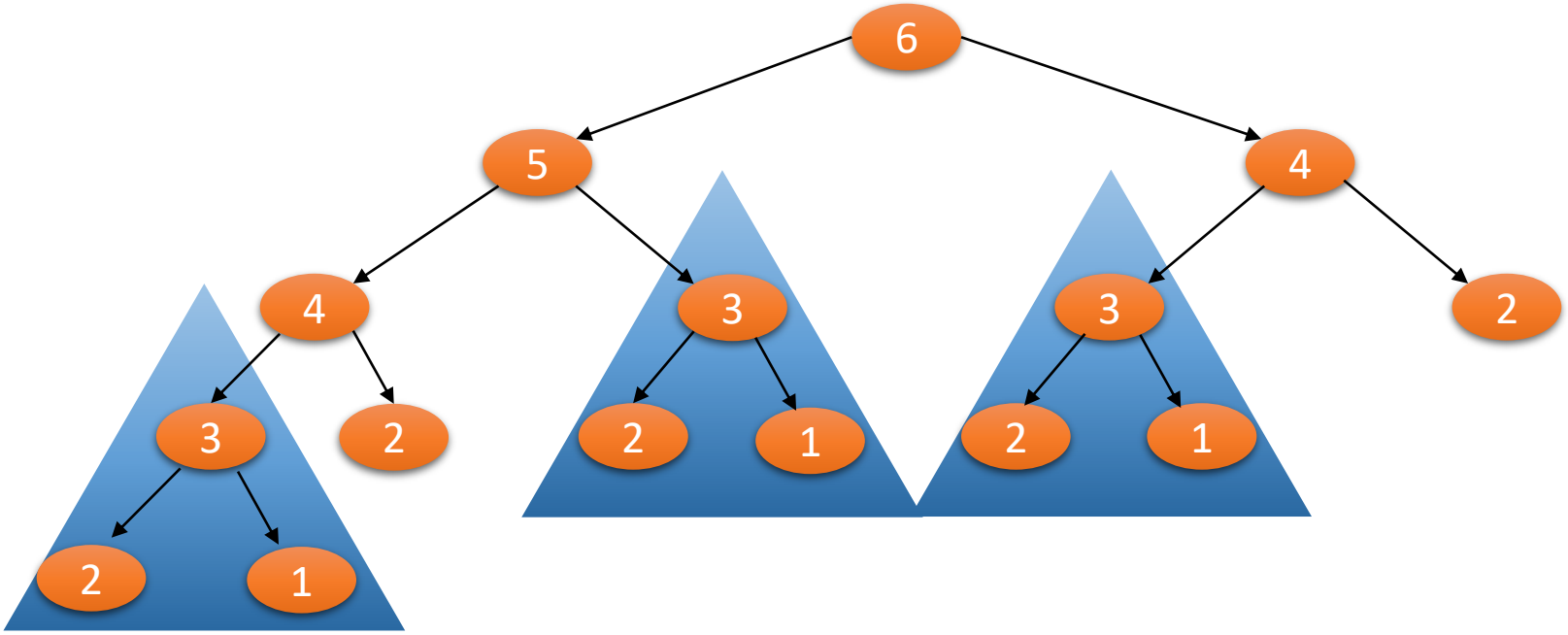


Dynamische Programmierung

Warum ist die Laufzeit so schlecht?

- Betrachte Rekursionsbaum von Fib2(6)

Es wird also dieselbe Rechnung dreimal ausgeführt!

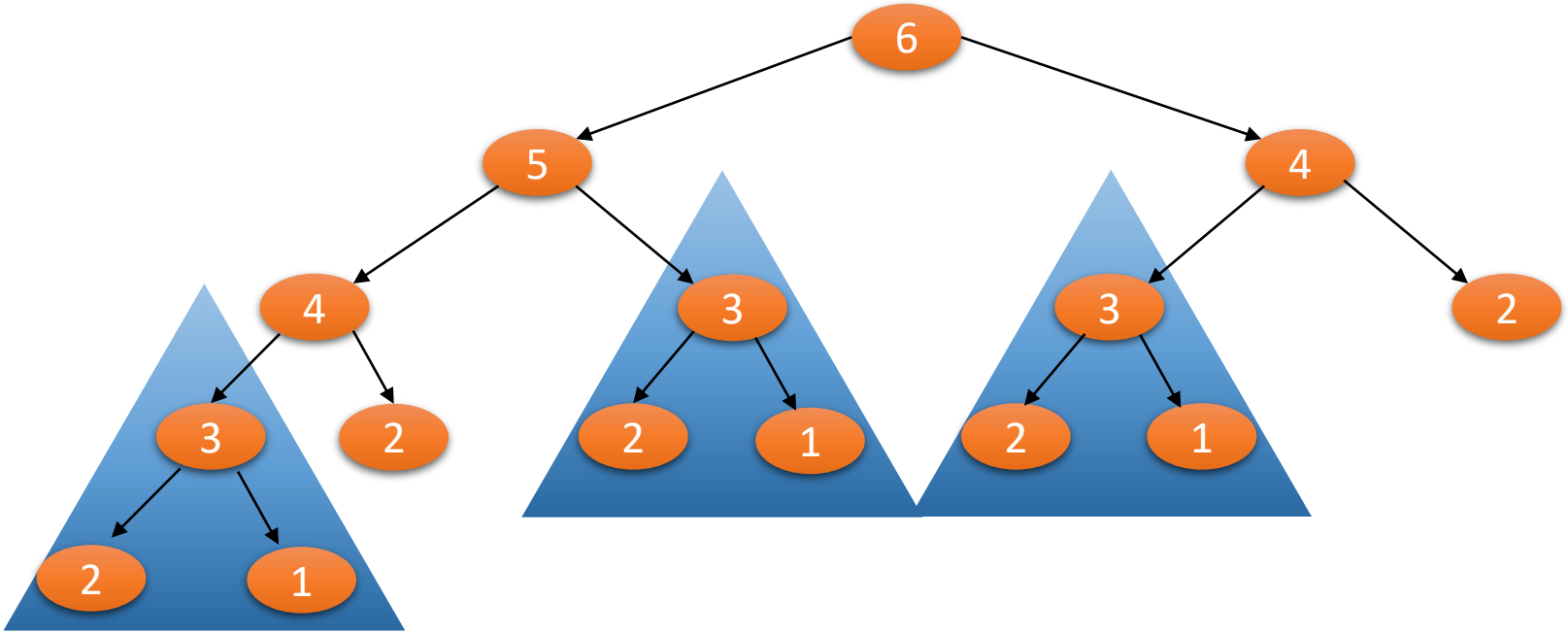


Dynamische Programmierung

Warum ist die Laufzeit so schlecht?

- Betrachte Rekursionsbaum von Fib2(6)

Bei großem n passiert dies sehr häufig!



Dynamische Programmierung

Zwischenspeichern von Rechenergebnisse

- Idee: Wir speichern die Ergebnisse, die wir bereits kennen

Dynamische Programmierung - Zwischenspeicherung

```
Fib3-Init(n)
1.  F = new array [1…n]
2.  for i=1 to n do
3.      F[i]=0
4.  F[1] = 1
5.  F[2] = 1
6.  return Fib3(F,n)
```

```
Fib3(F,n)
1.  if F[n] > 0 then return F[n]
2.  else
3.     F[n] = Fib3(F,n-1) + Fib3(F,n-2)
4.     return F[n]
```

Dynamische Programmierung - Zwischenspeicherung

```
Fib3-Init(n)
1.   F = new array [1…n]
2.   for i=1 to n do
3.      F[i]=0
4.   F[1] = 1
5.   F[2] = 1
6.   return Fib3(F,n)
```

O(n) O(n) O(1) 1 + T(n)

```
Fib3(F,n)
1.   if F[n] > 0 then return F[n]
2.   else
3.      F[n] = Fib3(F,n-1) + Fib3(F,n-2)
4.      return F[n]
```

T(n): Laufzeit von Fib3(F,n)

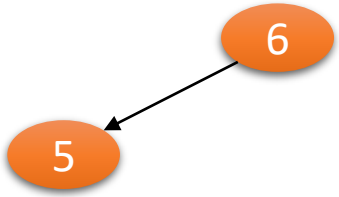
Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung



Beispiel

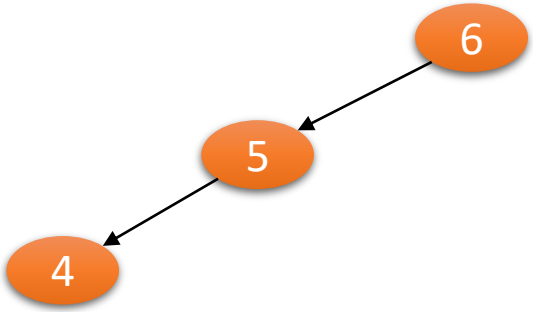
- Fib3(F,6) (nach Fib3-Init)

i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

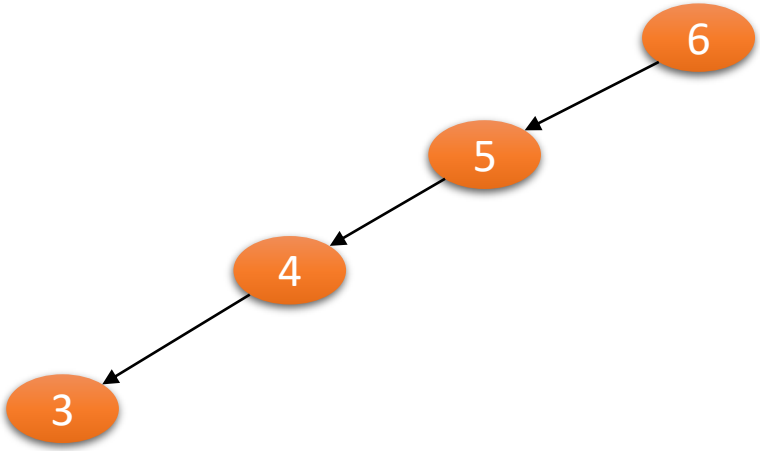


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

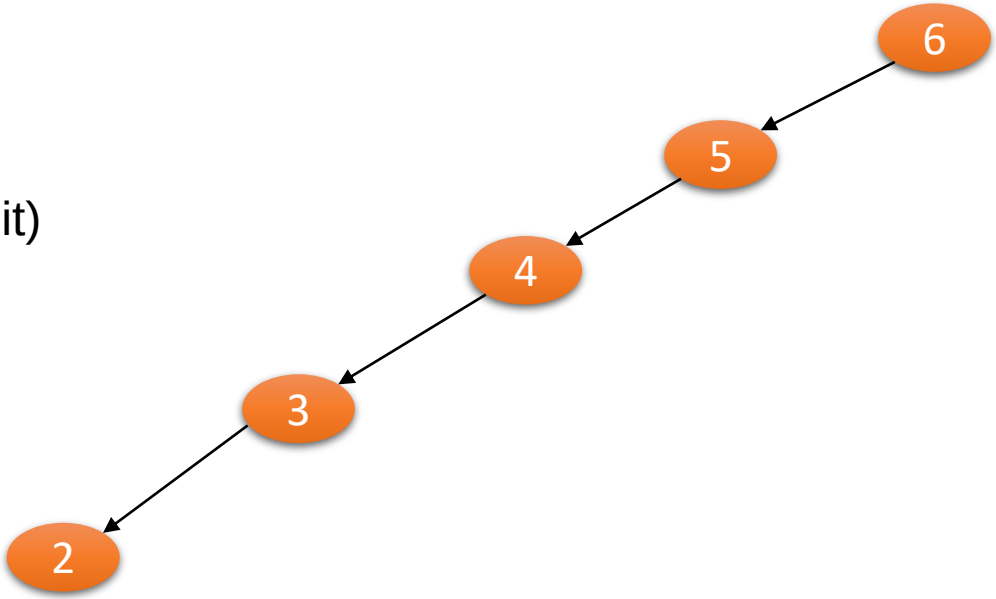


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

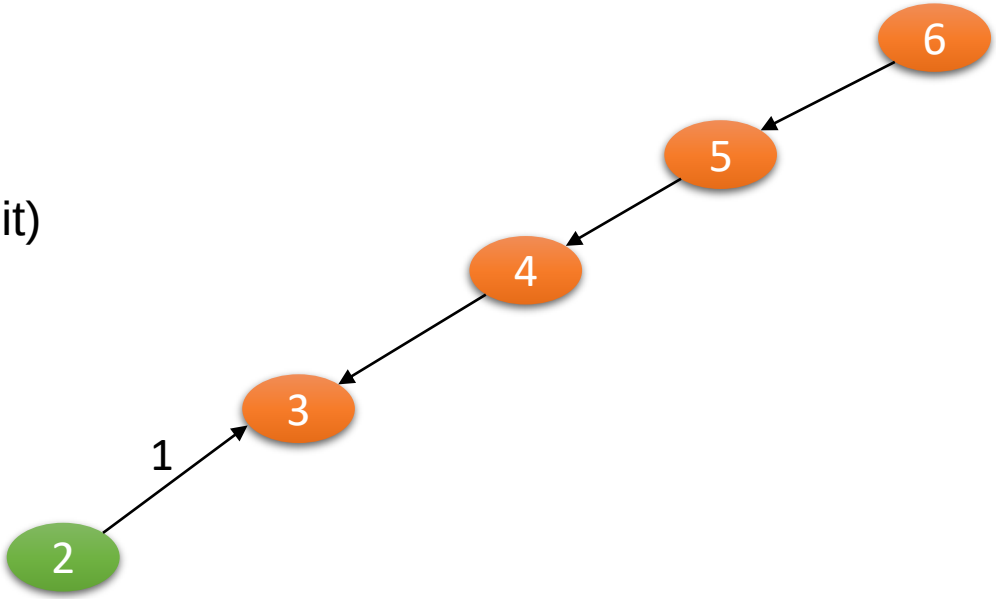


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

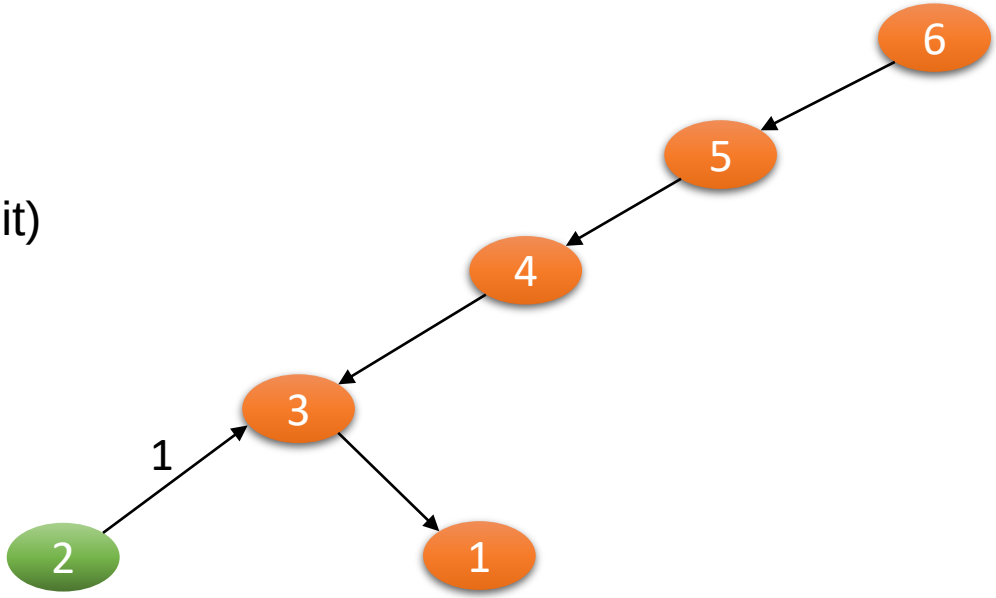


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

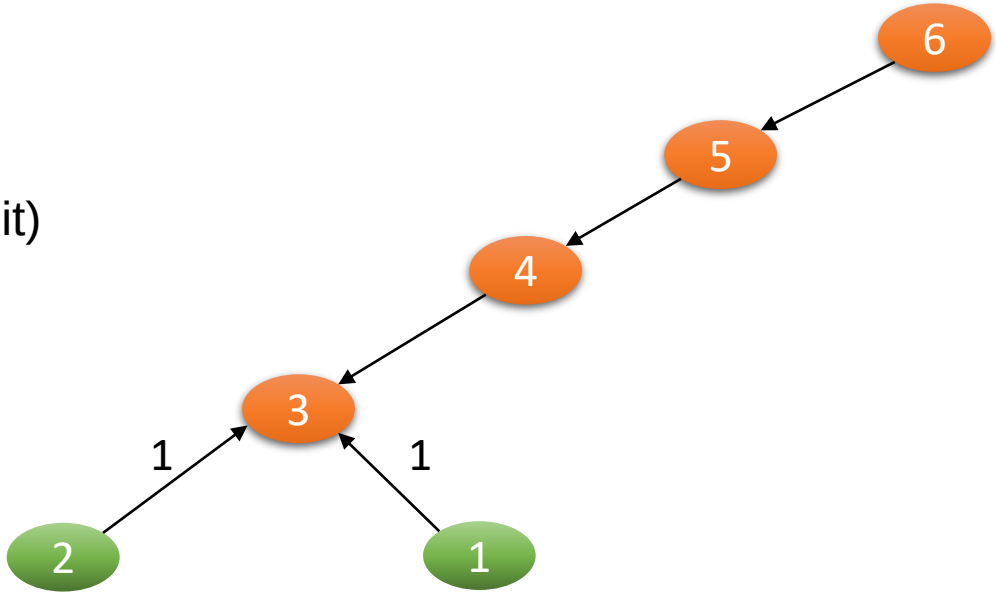


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

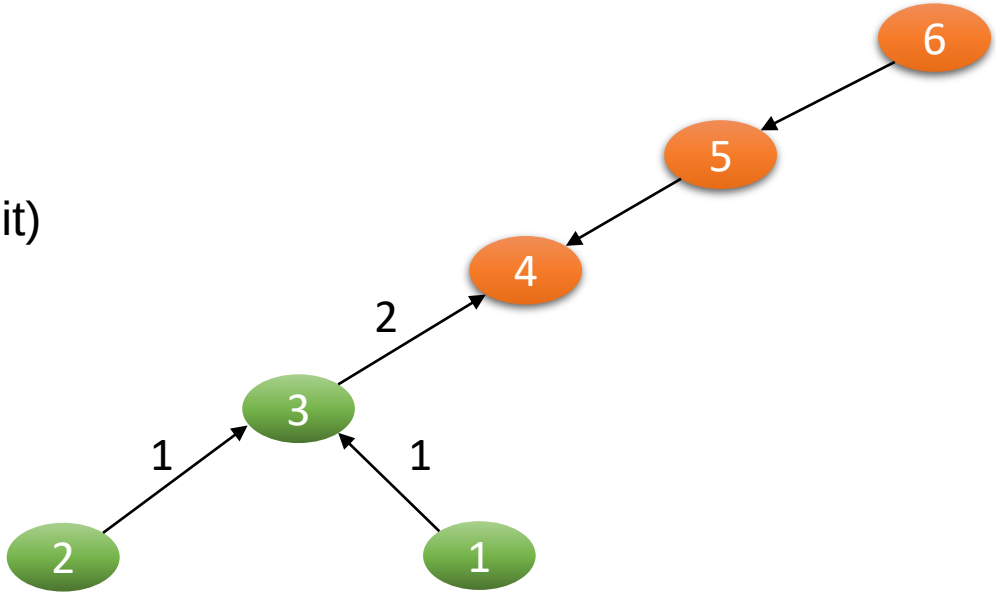


i	1	2	3	4	5	6
F[i]	1	1	0	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

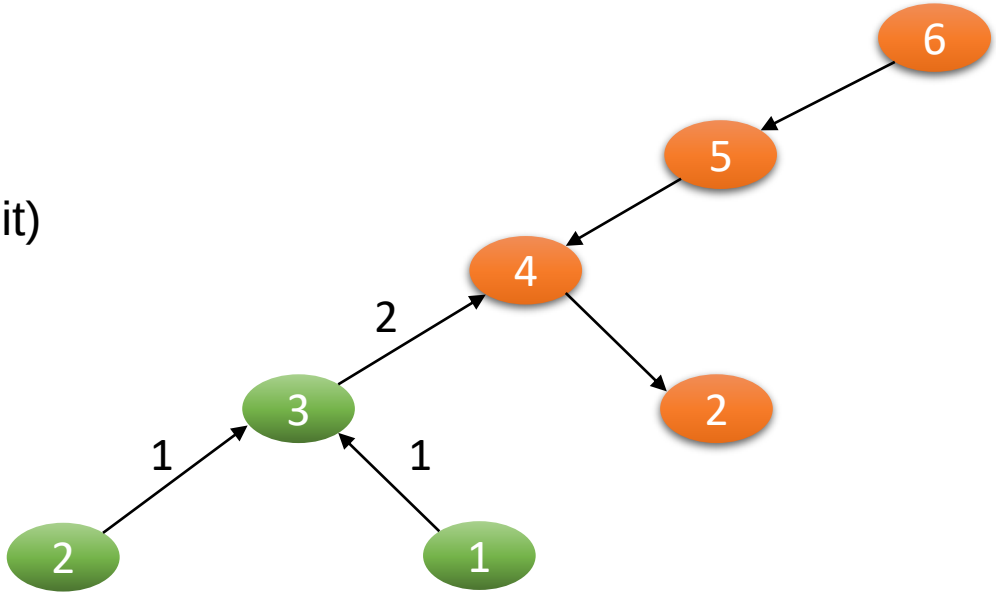


i	1	2	3	4	5	6
F[i]	1	1	2	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

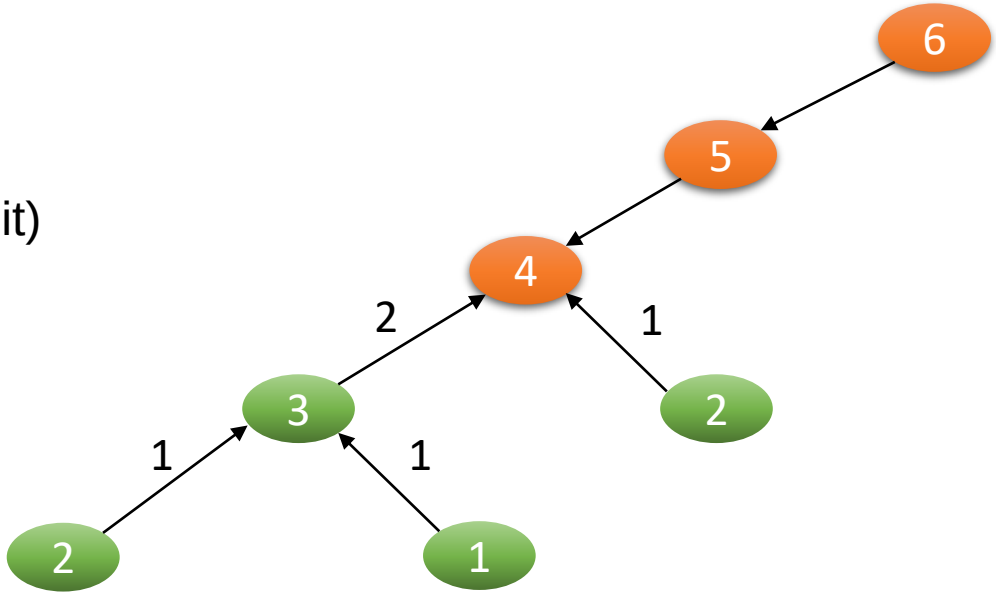


i	1	2	3	4	5	6
F[i]	1	1	2	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

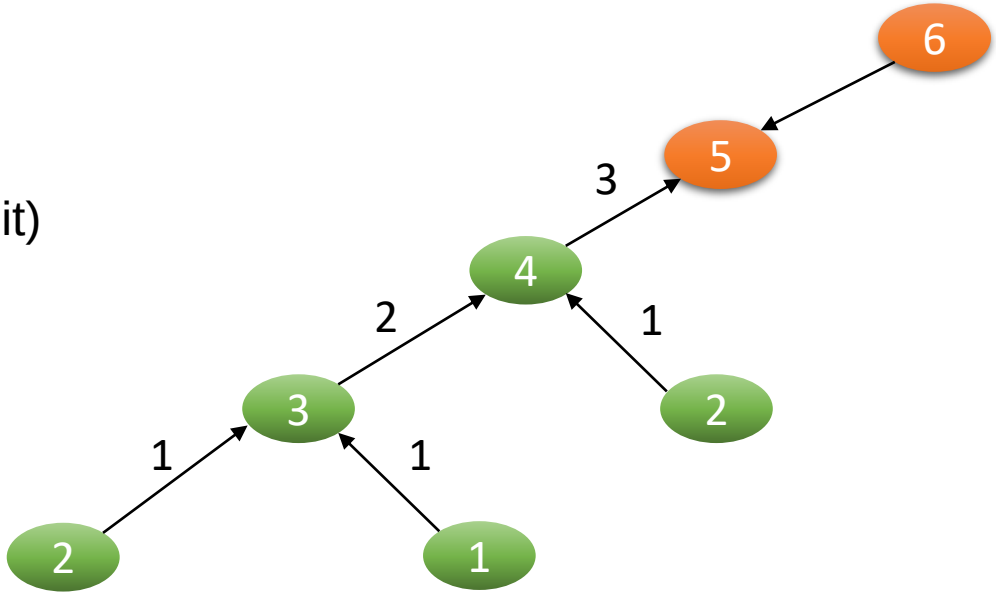


i	1	2	3	4	5	6
F[i]	1	1	2	0	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

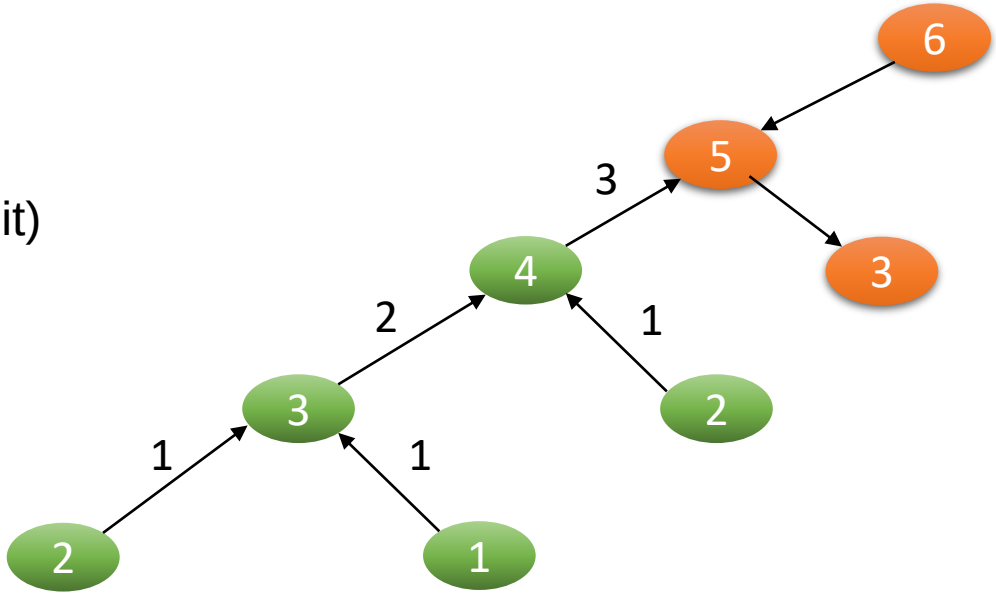


i	1	2	3	4	5	6
F[i]	1	1	2	3	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

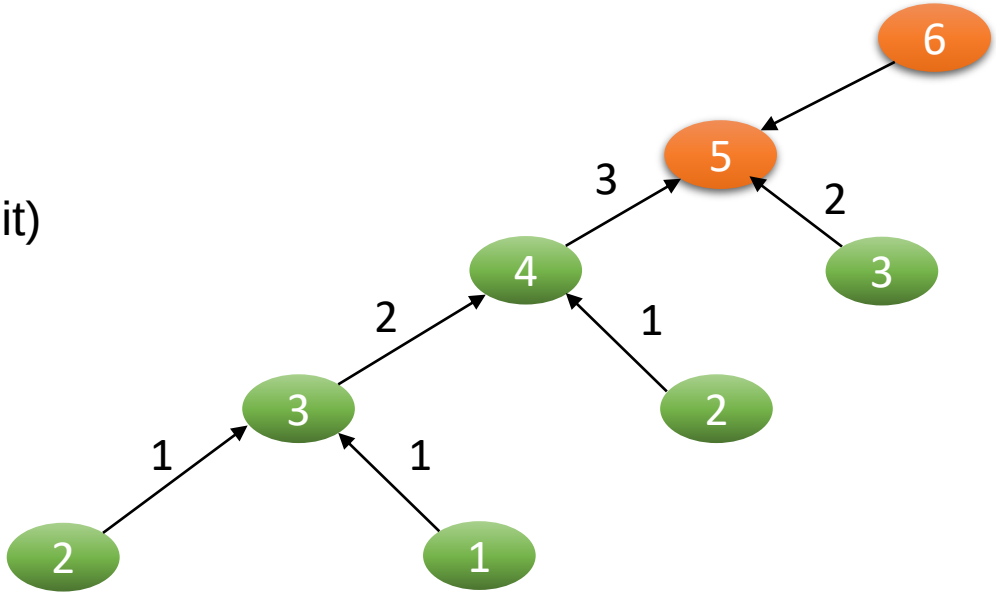


i	1	2	3	4	5	6
F[i]	1	1	2	3	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

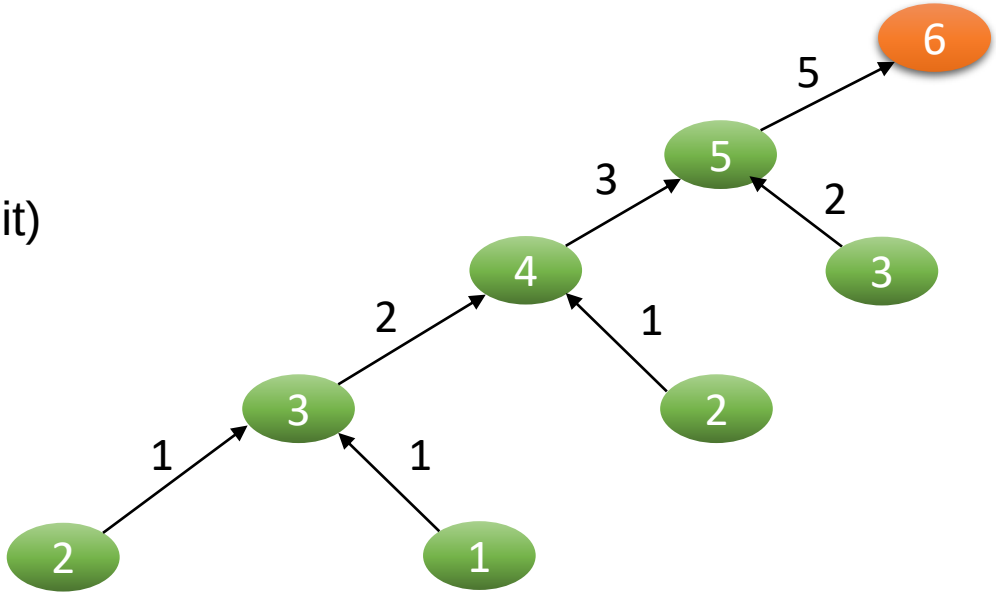


i	1	2	3	4	5	6
F[i]	1	1	2	3	0	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

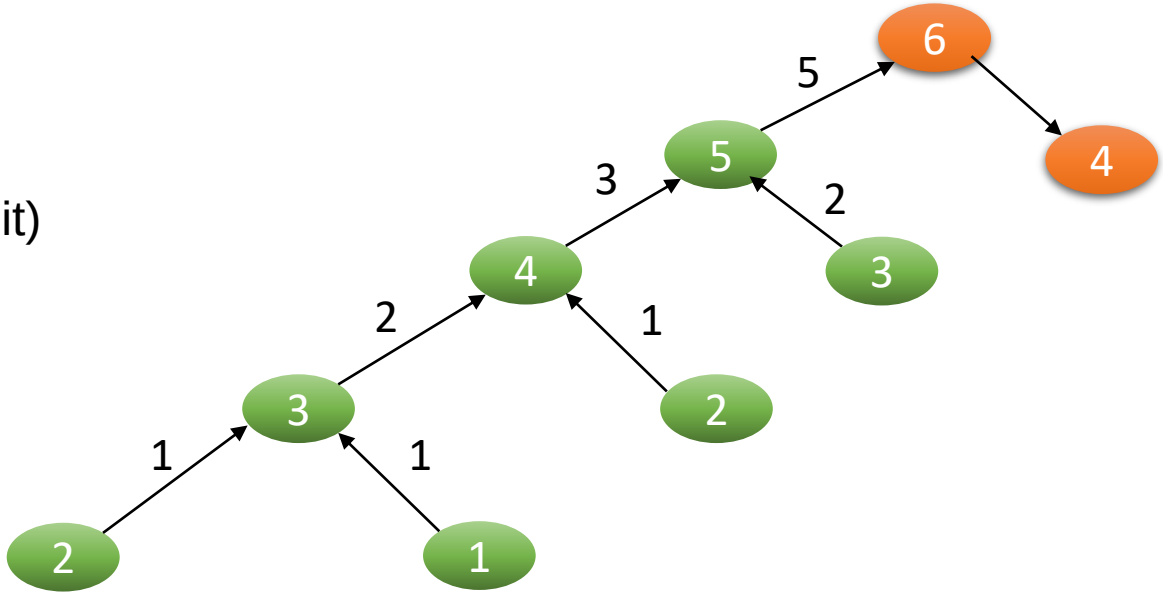


i	1	2	3	4	5	6
F[i]	1	1	2	3	5	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

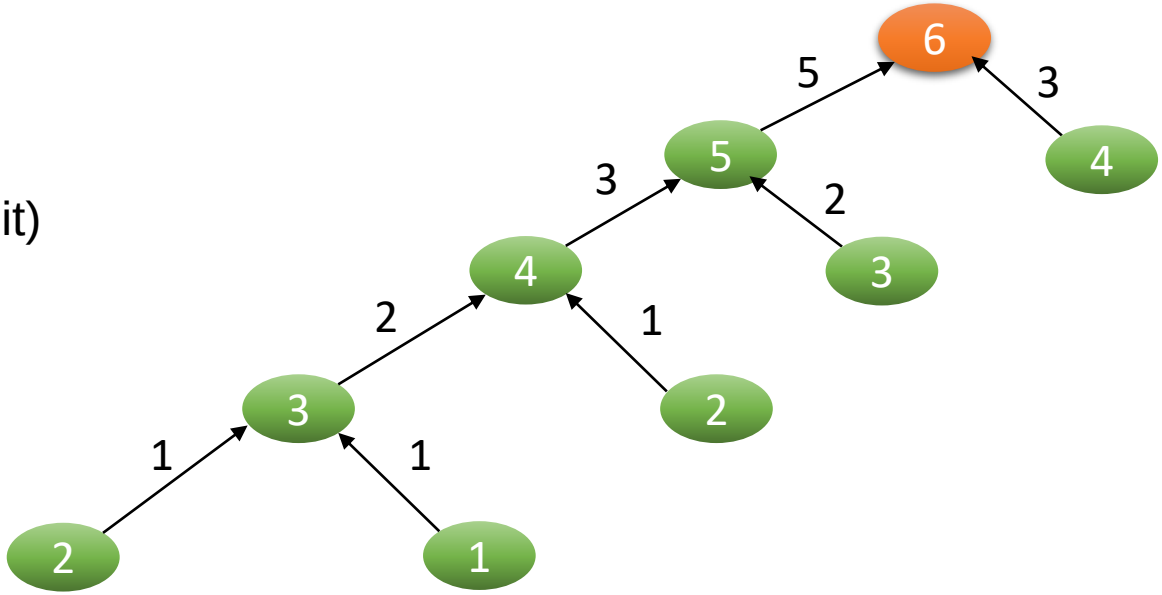


i	1	2	3	4	5	6
F[i]	1	1	2	3	5	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)

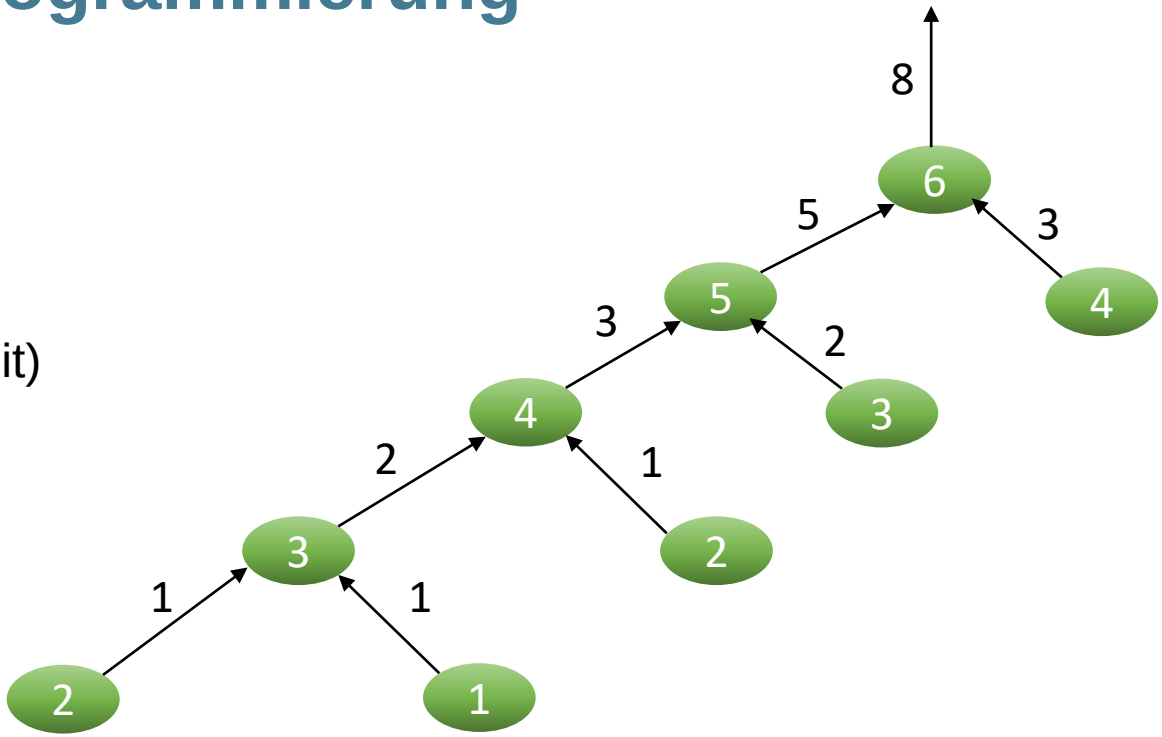


i	1	2	3	4	5	6
F[i]	1	1	2	3	5	0

Dynamische Programmierung

Beispiel

- Fib3(F,6) (nach Fib3-Init)



i	1	2	3	4	5	6
F[i]	1	1	2	3	5	8

Dynamische Programmierung

```
Fib3(F,n)
1.  if F[n] > 0 then return F[n]
2.  else F[n] = Fib3(F,n-1) + Fib3(F,n-2)
```

Behauptung 9.2

- Für jedes m>0 gilt: Fib3(F,m) wird maximal zweimal aufgerufen.

Beweis

- Fib3(F,m) wird entweder von Fib3(F,m+1) oder Fib3(F,m+2) aufgerufen
- Die rekursiven Aufrufe in Fib3(F,m+1) und Fib3(F,m+2) werden nur einmal ausgeführt, da bei einem weiteren Aufruf F[m+1] (bzw. F[m+2]) ungleich 0 ist und somit in Zeile 1 zurückgesprungen wird
- Damit folgt die Behauptung

Dynamische Programmierung

```
Fib3(F,n)
1.  if F[n] > 0 then return F[n]
2.  else F[n] = Fib3(F,n-1) + Fib3(F,n-2)
```

Lemma 9.3

- Die Laufzeit von Fib3(F,n) (inklusive Initialisierung) ist O(n).

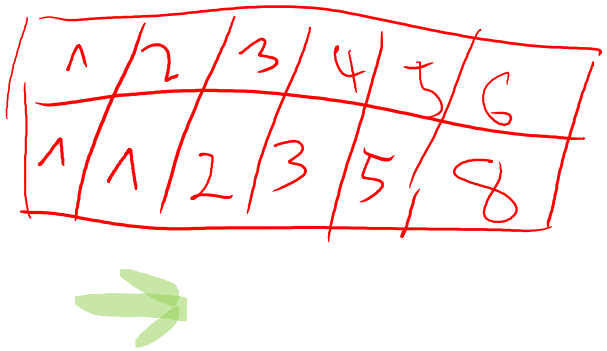
Beweis

- Wir haben bereits gesehen, dass die Initialisierung in O(n) Zeit geht
- Fib3(F,m) wird nur für Werte m aus dem Bereich {1,…,n} aufgerufen
- Ein Aufruf von Fib3(F,m) ohne die rekursiven Aufrufe benötigt O(1) Zeit
- Aufgrund der Behauptung 9.2 ist damit die Gesamtlaufzeit O(n)

Dynamische Programmierung

Beobachtung

- Die Tabelle wird bottom-up ausgefüllt



Vereinfachter Code

```
Fib1(n)
1.  F = new array[1…n]
2.  F[1] = 1
3.  F[2] = 1
4.  for i=3 to n do
5.     F[i] = F[i-1] + F[i-2]
6.  return F[n]
```

Dynamische Programmierung

Beobachtung

- Die Tabelle wird bottom-up ausgefüllt

Vereinfachter Code

```
Fib1(n)
1.   F = new array[1…n]
2.   F[1] = 1
3.   F[2] = 1
4.   for i=3 to n do
5.      F[i] = F[i-1] + F[i-2]
6.   return F[n]
```

O(n)

Dynamische Programmierung

Dynamische Programmierung

- Beschreibe optimale Lösung einer gegebenen Instanz durch optimale Lösungen „kleinerer“ Instanzen (hier kleinere Fibonacci-Zahlen)
- Beschreibe Rekursionsabbruch
- Löse die Rekursion „bottom-up“ durch schrittweises Ausfüllen einer Tabelle der benötigten Teillösungen

Wann verbessert der Ansatz die Laufzeit?

- Die Anzahl unterschiedlicher Funktionsaufrufe (Größe der Tabelle) ist klein
- Bei einer „normalen Ausführung“ des rekursiven Algorithmus ist mit vielen Mehrfachausführungen zu rechnen

Dynamische Programmierung

Aufgabe

- Entwickeln Sie einen Algorithmus für die Maximumssuche, der auf dynamischer Programmierung basiert. Der Algorithmus soll das maximale Element (nicht den Index) zurückgeben.
- Formulieren Sie dazu zunächst das Problem rekursiv
- Sie können für das Problem keine Laufzeitverbesserung erwarten!

Dynamische Programmierung

Ein einfaches Beispiel

- Maximum von n Zahlen
- Eingabe: Array A der Größe n
- Ausgabe: Wert des Maximums der Zahlen in A

Ziel

- Dynamische Programmierung anhand dieses Beispiels durchspielen
- Natürlich keine Laufzeitverbesserung zu erwarten

Dynamische Programmierung

Entwicklung der Rekursionsgleichung

- Eingabe besteht aus n Elementen
- Idee: Ordne die Elemente von 1 bis n (das Eingabefeld A gibt z.B. eine solche Ordnung)
- Drücke optimale Lösung für die ersten i Elemente als Funktion der optimalen Lösung ersten i-1 Elemente aus

Dynamische Programmierung

Entwicklung der Rekursionsgleichung

- Eingabe besteht aus n Elementen
- Idee: Ordne die Elemente von 1 bis n (das Eingabefeld A gibt z.B. eine solche Ordnung)
- Drücke optimale Lösung für die ersten i Elemente als Funktion der optimalen Lösung ersten i-1 Elemente aus

Beispiel

- Sei Max(i) = $\max_{1 \leq j \leq i} \{A[j]\}$
- Dann gilt Max(1) = A[1] (Rekursionsabbruch)
- Max(i) = max {Max(i-1), A[i])}

Dynamische Programmierung

MaxSucheDP(A,n)

1. Max = **new array** [1...n]
2. Max[1] = A[1]
3. **for** i=2 **to** n **do**
4. Max[i] = max{Max[i-1], A[i]}
5. **return** Max[n]

Dynamische Programmierung

Satz 9.4

- Algorithmus MaxSucheDP(A,n) berechnet ein maximales Element in einem Feld A[1..n].

Beweis

- Schleifeninvariante: Max[i-1] ist ein maximales Element aus A[1..i-1]
- Induktionsanfang (i=2): Vor dem ersten Schleifeneintritt ist Max[1] = A[1] und enthält das maximale Element aus A[1..1]
- Induktionsannahme: Max[i-1] ist ein maximales Element aus A[1..i-1] und $i \leq n$
- Induktionsschluss: In der Schleife wird Max[i] auf max{Max[i-1], A[i]} gesetzt
- Laut Induktionsannahme ist Max[i-1] ein maximales Element aus A[1..i-1]
- Damit ist Max[i] ein maximales Element aus A[1..i]
- Der Algorithmus gibt Max[n] zurück, was aufgrund der Schleifeninvariante korrekt ist

Dynamische Programmierung

MaxSucheDP(A,n)

1. Max = **new array** [1...n]
2. Max[1] = A[1]
3. **for** i=2 **to** n **do**
4. Max[i] = max{Max[i-1], A[i]}
5. **return** Max[n]

Wie kann man aus Max[1..n] den Index eines größten Elements von A herausfinden?

- Immer wenn sich der Wert von Max ändert, dann ändert sich auch der Index

Dynamische Programmierung

MaxIndexSucheDP(A,n)

```
1.  Max   = new array [1...n]
2.  Index = new array [1...n]
3.  Max[1] = A[1]
4.  Index[1] = 1
5.  for i=2 to n do
6.     Max[i] = max{Max[i-1], A[i]}
7.     if Max[i] = Max[i-1] then
8.        Index[i] = Index[i-1]
9.     else
10.       Index[i] = i
11. return Index[n]
```

Dynamische Programmierung

Was lernen wir aus der Maximumsberechnung?

- Wenn wir es mit Mengen zu tun haben, können wir eine Ordnung der Elemente einführen und die Rekursion durch Zurückführen der optimalen Lösung für i Elemente auf die Lösung für i-1 Elemente erhalten
- Benötigt wird dabei der Wert der optimalen Lösung für i-1 Elemente
- Die Lösung selbst (der Index des Maximums) kann nachher aus der Tabelle rekonstruiert werden

Dynamische Programmierung

Zusammenfassung

- Laufzeit rekursive Berechnung der Fibonacci-Zahlen
- Verbesserung durch Speichern der Lösungen
- Iterative Lösung
- Prinzip der dynamischen Programmierung
- Ein erstes einfaches Beispiel

Referenzen

- T. Cormen, C. Leisserson, R. Rivest, C. Stein. Introduction to Algorithms. The MIT press. Second edition, 2001.