Tafelübung 05
Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Übersicht

Organisatorisches

Parameter-Übergabe: Semantik

Vollständige Induktion
- Einführung
- Beispiel
- Weitere Induktionsarten
- Totale Korrektheit

Dynamische Programmierung
- Motivation: Fibonacci-Zahlen
- Idee und Funktionsweise
- Umsetzung in einem Beispiel
- Bottom-Up-Berechnung

Backtracking
- Motivation
- Grundlagen

Hinweise zu den Aufgaben
- Induktionsbeweis hokuspokus

Organisatorisches

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Hausaufgaben und Rechnerübungen

Auslastung der Rechnerübungen

- verschiedene Rechnerübungen sind bislang unterschiedlich stark besucht
  $\rightsquigarrow$ Übersicht: `https://www2.cs.fau.de/teaching/WS2016/AuD/uebungen/AuD_RUe_Auslastung.png`
  $\rightsquigarrow$ oder im FSI-Forum (jeweils unregelmäßig aktualisiert)
- wenn möglich: besucht Rechnerübungen, die nicht überfüllt sind

Richtiger Zeitpunkt für den Beginn der Hausaufgaben

- so früh wie möglich!
- nicht erst am Freitagvormittag

$\rightsquigarrow$ Fragen wie „Ich hab die Angabe nicht verstanden“ sollten nicht erst in der Freitags-Rechnerübung auftreten

Parameter-Übergabe: Semantik

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Formale und tatsächliche Parameter

Formaler Parameter (*formal parameter*)

Parameter-Variable, die bei der Funktions-Deklaration angegeben wird
$\rightsquigarrow$ „benannter Speicherplatz“

Tatsächlicher Parameter (Argument, *actual parameter*)

Bei einem konkreten Aufruf übergebener Wert
$\rightsquigarrow$ wird in einem formalen Parameter gespeichert

Beispiel

```
void foo(int a) { /* a ist formaler Parameter */ }

// ...

foo(13); // 13 ist tatsächlicher Parameter für a
```

Einschub: primitive und zusammengesetzte Datentypen

- bereits bekannt:
  - primitive Datentypen sind z.B. `int, double, long, boolean`
  - zusammengesetzte Datentypen sind z.B. `int[], double[][], String`
  - Variablen mit primitiven Datentypen und zusammengesetzten Datentypen verhalten sich manchmal unterschiedlich

  $\rightsquigarrow$ z.B. `NullPointerExceptions` nur bei zusammengesetzten Datentypen
- weiterer Unterschied: Speicherort
  - Variablen mit primitivem Datentyp: direkt in der Methodenschachtel
  - Variablen mit zusammengesetztem Datentyp: an anderer Stelle

  $\rightsquigarrow$ in der Methodenschachtel wird lediglich auf diese Stelle verwiesen

  $\rightsquigarrow$ diesen Verweis nennt man auch Referenz

Call-by-Value vs. Call-By-Reference

Call-by-Value

Der tatsächliche Parameter wird beim Aufruf der Funktion ausgewertet und eine Kopie des Ergebnisses wird an die aufgerufene Funktion übergeben.

Call-by-Reference

Der formale Parameter ist ein Alias für die übergebene Variable (also den tatsächlichen Parameter), d.h. beide bezeichnen dieselbe Variable. Dazu wird keine Kopie des Arguments übergeben, sondern eine Referenz auf die Argumentvariable.

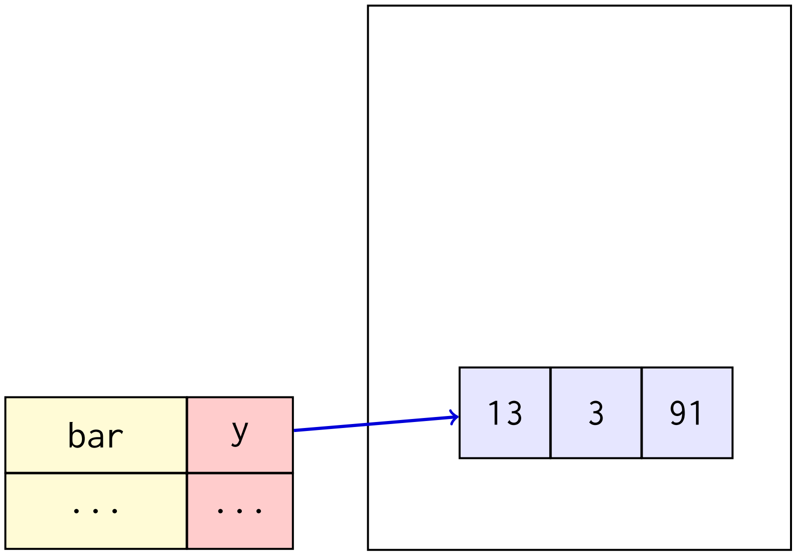
Achtung!

Java kennt nur Call-by-Value, auch Referenzen werden Call-by-Value übergeben! (Manchmal bezeichnet man dies auch als *Reference-by-Value*.)

Call-by-Value: Beispiel

Wie es tatsächlich ist...

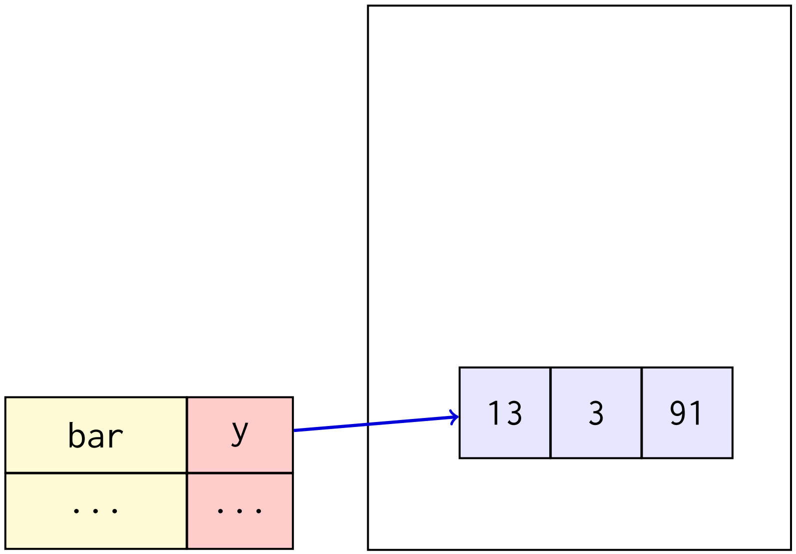
```java
public void foo(int[] x) {
  x[2] = 14;
  x = new int[3];
}

public void bar() {
  int[] y = {13, 3, 91};
  foo(y);
}
```



Call-by-Value: Beispiel

Wie es tatsächlich ist...

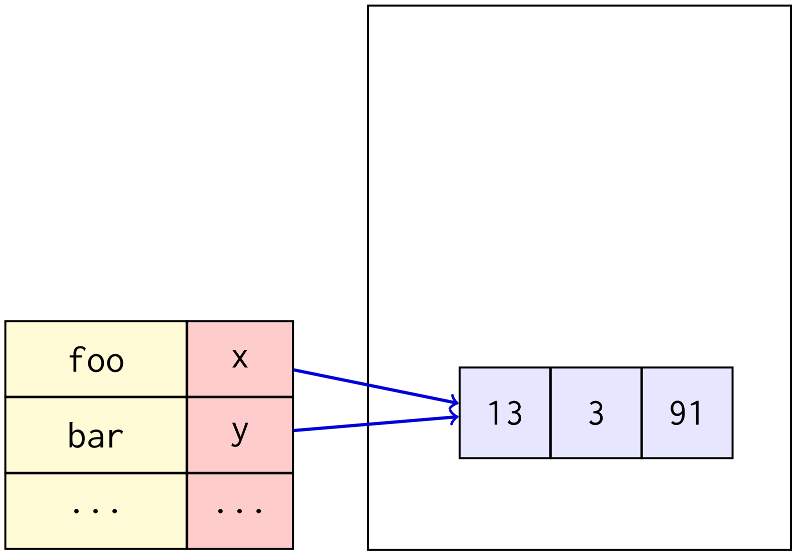
```
public void foo(int[] x) {
   x[2] = 14;
   x = new int[3];
}

public void bar() {
   int[] y = {13, 3, 91};
   foo(y);
}
```



Call-by-Value: Beispiel

Wie es tatsächlich ist...

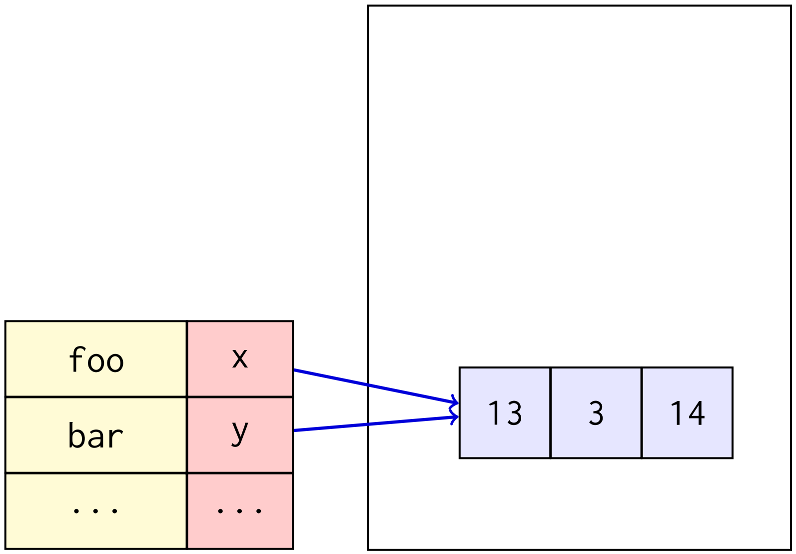
```
public void foo(int[] x) {
    x[2] = 14;
    x = new int[3];
}

public void bar() {
    int[] y = {13, 3, 91};
    foo(y);
}
```



Call-by-Value: Beispiel

Wie es tatsächlich ist...

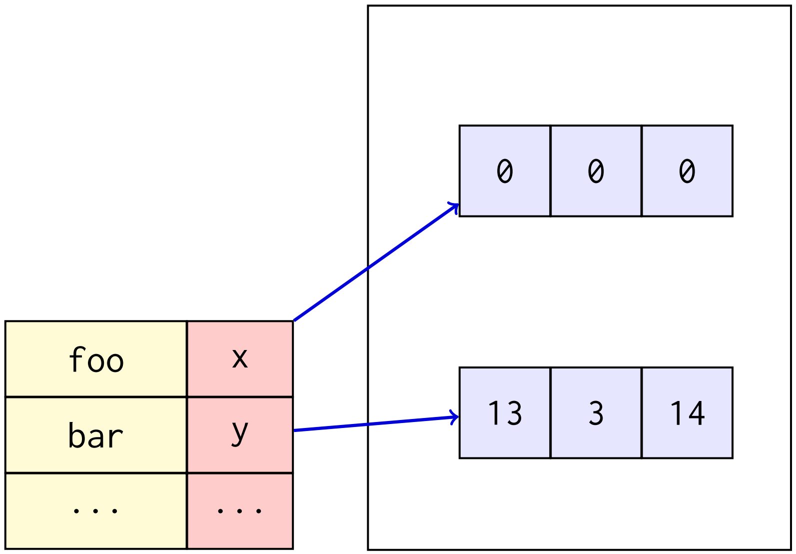
```java
public void foo(int[] x) {
  x[2] = 14;
  x = new int[3];
}

public void bar() {
  int[] y = {13, 3, 91};
  foo(y);
}
```



Call-by-Value: Beispiel

Wie es tatsächlich ist...

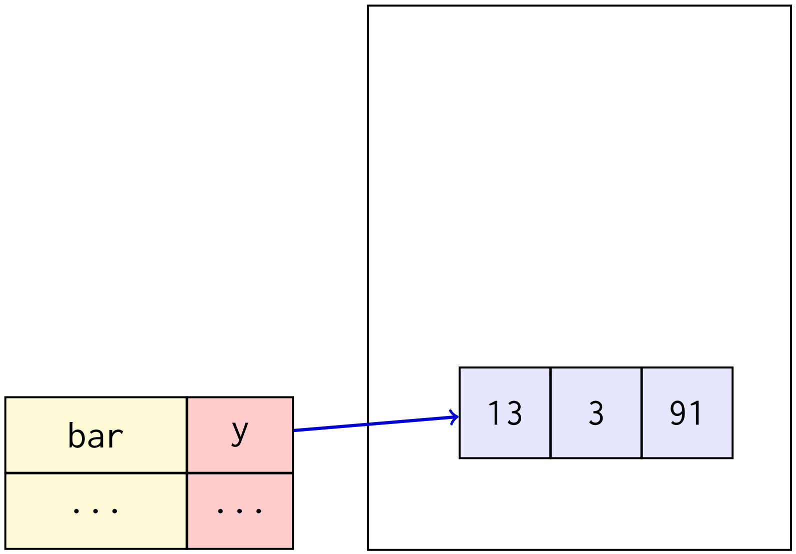
```
public void foo(int[] x) {
    x[2] = 14;
    x = new int[3];
}

public void bar() {
    int[] y = {13, 3, 91};
    foo(y);
}
```



Call-by-Reference: Beispiel

Wie es mit CbR wäre...

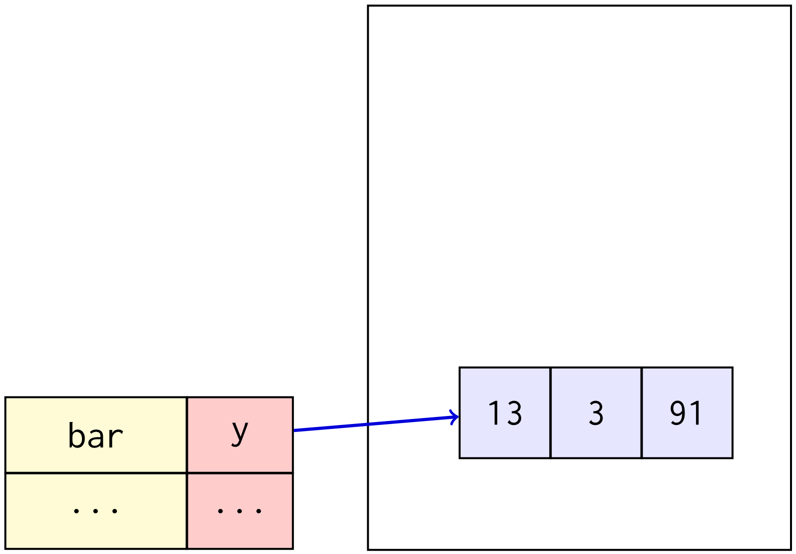
```
public void foo(int[] x) {
    x[2] = 14;
    x = new int[3];
}

public void bar() {
    int[] y = {13, 3, 91};
    foo(y);
}
```



Call-by-Reference: Beispiel

Wie es mit CbR wäre...

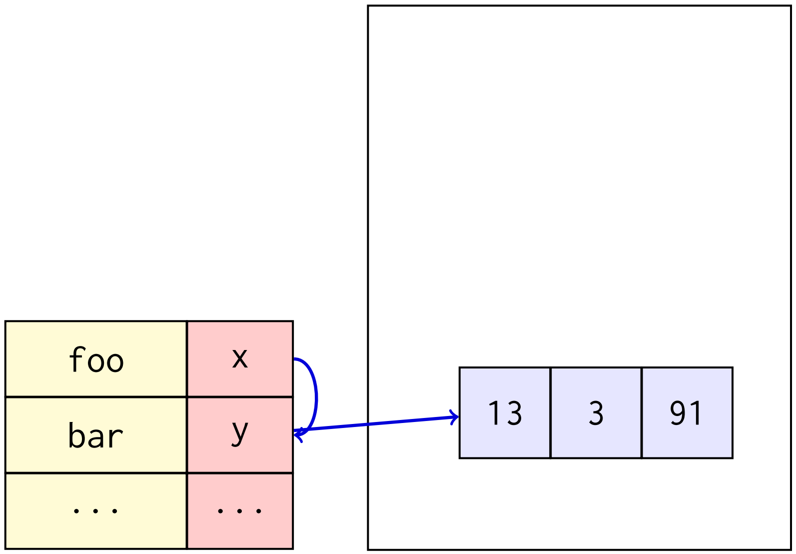
```
public void foo(int[] x) {
  x[2] = 14;
  x = new int[3];
}

public void bar() {
  int[] y = {13, 3, 91};
  foo(y);
}
```



Call-by-Reference: Beispiel

Wie es mit CbR wäre...

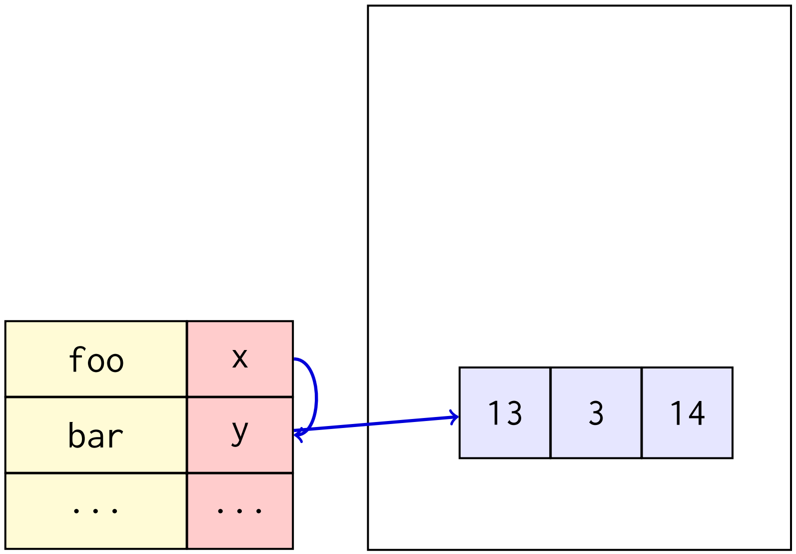
```
public void foo(int[] x) {
    x[2] = 14;
    x = new int[3];
}

public void bar() {
    int[] y = {13, 3, 91};
    foo(y);
}
```



Call-by-Reference: Beispiel

Wie es mit CbR wäre...

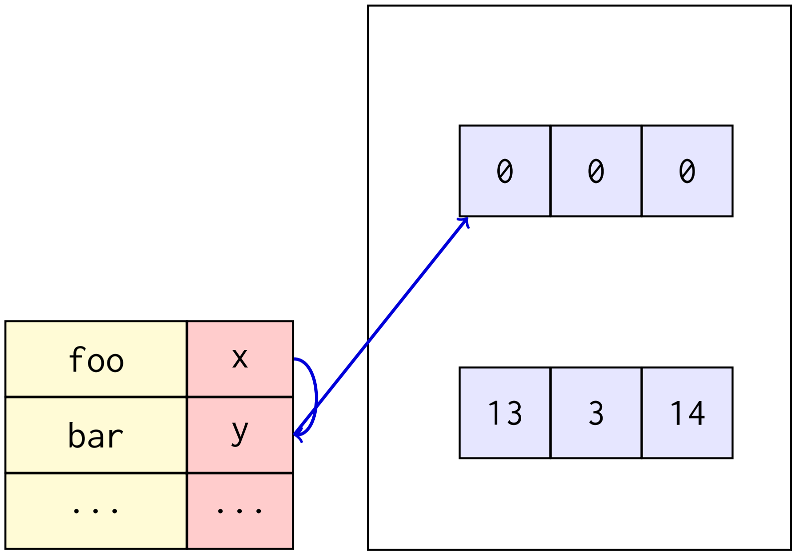
```
public void foo(int[] x) {
   x[2] = 14;
   x = new int[3];
}

public void bar() {
   int[] y = {13, 3, 91};
   foo(y);
}
```



Call-by-Reference: Beispiel

Wie es mit CbR wäre...

```
public void foo(int[] x) {
  x[2] = 14;
  x = new int[3];
}

public void bar() {
  int[] y = {13, 3, 91};
  foo(y);
}
```



Vollständige Induktion

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Beweis mittels Vollständiger Induktion

Vollständige Induktion

Das Beweisprinzip der Vollständigen Induktion besagt, dass eine Aussage $A(n)$ für alle $n \geq m$; $n, m \in \mathbb{N}$ korrekt ist, wenn

- $A(m)$ erfüllt ist (Induktionsanfang) und
- aus der Korrektheit von $A(n)$ (Induktionsvoraussetzung) für jedes $n \geq m$ die Korrektheit von $A(n+1)$ gefolgert werden kann (Induktionsschritt).

Da der Beweis für ein konkretes Element m im Induktionsanfang erbracht wurde und weiterhin im Induktionsschritt gezeigt wurde, dass aus $A(n)$ für ein beliebiges $n \geq m$ auch $A(n' = n + 1)$ folgt, ist die Aussage für alle $n \geq m$ gültig. Es gilt also $A(n)\ \forall n \geq m$.

Allgemeines Vorgehen

Allgemeines Vorgehen

- Induktionsanfang (IA) beweisen:
  - zeigen, dass die Aussage für das „erste“ Element gilt
- Induktionsvoraussetzung (IV) formulieren:
  - Annahme, dass die Aussage für ein beliebiges, aber festes n gilt
- Induktionsschritt (IS) beweisen:
  - zeigen, dass unter der IV die Aussage auch für $n + 1$ gilt

Hinweis

Man kann in der Induktionsvoraussetzung auch die Korrektheit für $n - 1$ annehmen, und im Induktionsschritt dann die Korrektheit für n folgern.

Vollständige Induktion $\leftrightarrow$ Rekursion

- vollständige Induktion $\leadsto$ Korrektheitsbeweis von Funktionen

$\leadsto$ berechnet die Funktion das gewünschte Ergebnis?

- insbesondere schön bei rekursiven Funktionen:
  - Induktionsanfang $\mapsto$ Basisfall
  - Induktionsvoraussetzung $\mapsto$ Annahme, dass rekursiver Aufruf korrekt ist
  - Induktionsschritt $\mapsto$ Korrektheit der Funktion

Wichtig

Der Zusammenhang zwischen zu beweisender Aussage und Programmcode muss deutlich werden. Andernfalls wird durch die Induktion nicht bewiesen, dass das Programm dasselbe berechnet wie die mathematische Formel.

Beispiel

Beispiel

Gegeben sei die folgende Funktion:

```
static long sum(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sum(n-1);
    }
    return sum;
}
```

Zeigen Sie, dass folgende Aussage gilt:

$$\forall n \in \mathbb{N}, n \geq 0: \;\; \texttt{sum(n)} \;\equiv\; \sum_{i=0}^{n} i \;=:\; S_n$$

Einschub: Summenzeichen(I)

Teil der Formel aus Beispiel

$$\sum_{i=0}^{n} i =: \; S_n$$

Aber was ist das für eine seltsame „Zickzacklinie“?

Summenzeichen

$\sum$ nennt man auch Summenzeichen.

Die Schreibweise

$$\sum_{i=a}^{b} i$$

bedeutet, dass alle Zahlen $i \in \mathbb{Z}$, die im Intervall $[a, b]$ liegen, aufsummiert werden. Das Intervall $[a, b]$ ist dabei ein geschlossenes Intervall, d.h. sowohl a als auch b sind darin enthalten.

Einschub: Summenzeichen (II)

Beispiel

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15$$

Umsetzung in *iterativen* Java-Code

Schreiben Sie eine *iterative* Methode `sumIterative`, die für gegebene Parameter *a* und *b* die folgende Summe berechnet:

$$\sum_{i=a}^{b} i$$

```java
static long sumIterative(int a, int b) {
    long sum = 0;
    for (int i = a; i <= b; i++) {
        sum = sum + i;
    }
}
```

Zurück zum Beispiel...

Beispiel

Gegeben sei die folgende Funktion:

```
static long sum(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sum(n-1);
    }
    return sum;
}
```

Zeigen Sie, dass folgende Aussage gilt:

$$\forall n \in \mathbb{N}, n \geq 0: \ \ \texttt{sum(n)} \ \equiv \ \sum_{i=0}^{n} i \ =: \ S_n$$

Vorgehen

- im Induktionsanfang zeigen, dass die Formel im Basisfall ($n = 0$) erfüllt ist
- in der Induktionsvoraussetzung annehmen, dass die Formel für $n - 1$ (und damit den rekursiven Aufruf) korrekt ist
- im Induktionsschritt zeigen, dass aus der Korrektheit für $n - 1$ auch die Korrektheit für n folgt

Induktionsanfang (Basisfall, $n = 0$):

$$\texttt{sum(n)} \stackrel{n=0}{\equiv} \texttt{sum(0)} \stackrel{if-then}{\equiv} 0 \equiv \sum_{i=0}^{0} i \equiv S_0$$

Induktionsvoraussetzung (IV, $n-1$):

$$\texttt{sum(n-1)} \equiv S_{n-1} \equiv \sum_{i=0}^{n-1} i$$

Induktionsschritt ($n-1 \rightsquigarrow n$):

$$\texttt{sum(n)} \stackrel{if-else}{\equiv} \texttt{n + sum(n-1)} \stackrel{IV}{\equiv} n + \sum_{i=0}^{n-1} i \equiv \sum_{i=0}^{n} i \equiv S_n \qquad \blacksquare$$

Induktion mit mehreren Induktionsanfängen

Beispiel

Gegeben sei die folgende Funktion:

```
static long lf(int n) { // returns the left factorial !n
    if      (n <= 0) return 0;
    else if (n == 1) return 1;
    else             return n * lf(n - 1) - (n - 1) * lf(n - 2);
}
```

Zeigen Sie, dass folgende Aussage gilt:

$$\forall n \geq 0 : \ \texttt{lf(n)} \equiv \sum_{k=0}^{n-1} k!$$

Wichtig

Die Methode `lf` besitzt zwei Basisfälle (für $n = 0$ und $n = 1$) und beim rekursiven Aufruf wird n um 2 verringert. Darum benötigt der Induktionsbeweis zwei Induktionsanfänge und zwei Induktionsvoraussetzungen!

Induktionsanfang:

IA_0 $(n = 0)$**:**

$$\sum_{k=0}^{0-1} k! = 0 \equiv 0 = \texttt{lf(0)}$$

IA_1 $(n = 1)$**:**

$$\sum_{k=0}^{1-1} k! = 0! = 1 \equiv 1 = \texttt{lf(1)}$$

Induktionsvoraussetzung(en):

IV_{n-1} $(n - 1)$**:**

$$\texttt{lf(n-1)} \equiv \sum_{k=0}^{(n-1)-1} k! = \sum_{k=0}^{n-2} k!$$

IV_n (n)**:**

$$\texttt{lf(n)} \equiv \sum_{k=0}^{n-1} k!$$

Induktionsschluss:

$$\texttt{lf(n+1)} = (n+1) \cdot \texttt{lf((n+1) - 1)} - ((n+1) - 1) \cdot \texttt{lf((n+1) - 2)} =$$

$$= (n+1) \cdot \texttt{lf(n)} - n \cdot \texttt{lf(n - 1)} \overset{IV_{n-1}, IV_n}{\equiv}$$

$$\equiv (n+1) \cdot \sum_{k=0}^{n-1} k! - n \cdot \sum_{k=0}^{n-2} k! =$$

$$= n \cdot \sum_{k=0}^{n-1} k! + \sum_{k=0}^{n-1} k! - n \cdot \sum_{k=0}^{n-2} k! =$$

$$= n \cdot (n-1)! + n \cdot \sum_{k=0}^{n-2} k! + \sum_{k=0}^{n-1} k! - n \cdot \sum_{k=0}^{n-2} k! =$$

$$= n! + \sum_{k=0}^{n-1} k! = \sum_{k=0}^{(n+1)-1} k! \qquad \blacksquare$$

Induktion über zwei Variablen (I)

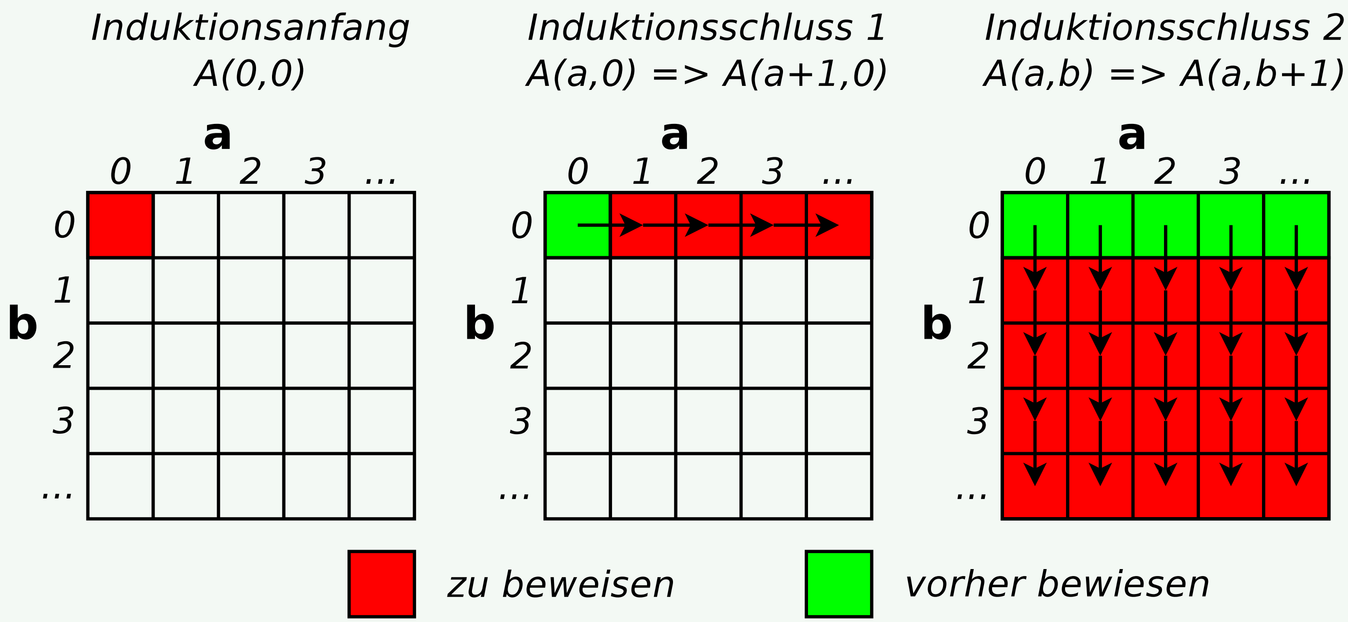
Induktion über zwei Variablen

Ist eine Funktion mit zwei Parametern gegeben, kann man auch für diese Funktion einen Induktionsbeweis führen:

- Induktionsanfang: setze beide Variablen auf 0
- Induktionsschritt$_1$:
  - halte zweite Variable fest bei 0
  - Induktion über erste Variable
- Induktionsschritt$_2$:
  - halte erste Variablen fest bei Wert *x* („beliebig, aber fest“)
  - Induktion über zweite Variable

Induktion über zwei Variablen (II)

Nochmal grafisch:



Weitere Induktionsarten

- Induktion mit n Basisfällen

$\leadsto$ für jeden Basisfall eigenen Induktionsanfang

- Induktion über n Variablen

$\leadsto$ Induktionsanfang: alle Variablen auf 0 setzen

$\leadsto$ n Induktionsschritte, analog zum Fall „2 Variablen"

- Induktion mit $n/2$ beim rekursiven Aufruf (statt $n - 1$)

$\leadsto$ je einen Induktionsanfang für geraden und ungeraden Basisfall (meist $n = 0$ und $n = 1$)

$\leadsto$ Induktionsschritt meist mit Fallunterscheidung, ob n gerade oder ungerade

- beliebig kombinierbar...
- ...und noch viele weitere Induktionsarten

$\leadsto$ diverse Theorie-Lehrveranstaltungen

Totale Korrektheit

- Terminierung gehört zur (totalen) Korrektheit einer rekursiven Methode dazu
  - Rekursion muss nach endlich vielen Schritten fertig sein

$\leadsto$ gesucht wird eine Terminierungsfunktion $T(n)$ mit folgenden Eigenschaften:
  - Werte von $T(n)$ sind ganzzahlig
  - die Folge der $T(n)$ ist streng monoton fallend
  - $T(n)$ ist nach unten beschränkt (meist: ≥ 0)

Tipp

In der Regel ist T eine Funktion über den Methodenargumenten. Aus dem Erreichen der unteren Schranke muss die Terminierung der Rekursion folgen.

Beispiel

Beispiel für eine Terminierungsfunktion

```
static long sum(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sum(n-1);
    }
}
```

$$\forall n \in \mathbb{N}, n \geq 0 : \ \texttt{sum(n)} \equiv \sum_{i=0}^{n} i =: S_n$$

Eine passende Terminierungsfunktion ist z.B.: $T(n) = n$

- Werte sind ganzzahlig (`int` bzw. $n \in \mathbb{N}$),
- streng monoton fallend (rekursiver Aufruf mit $n - 1$) und
- nach unten beschränkt (Basisfall `n==0` bzw. $n \geq 0$)

Dynamische Programmierung

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation: Fibonacci-Zahlen

Fibonacci-Folge

Die Fibonacci-Folge ist eine unendliche Folge von natürlichen Zahlen:

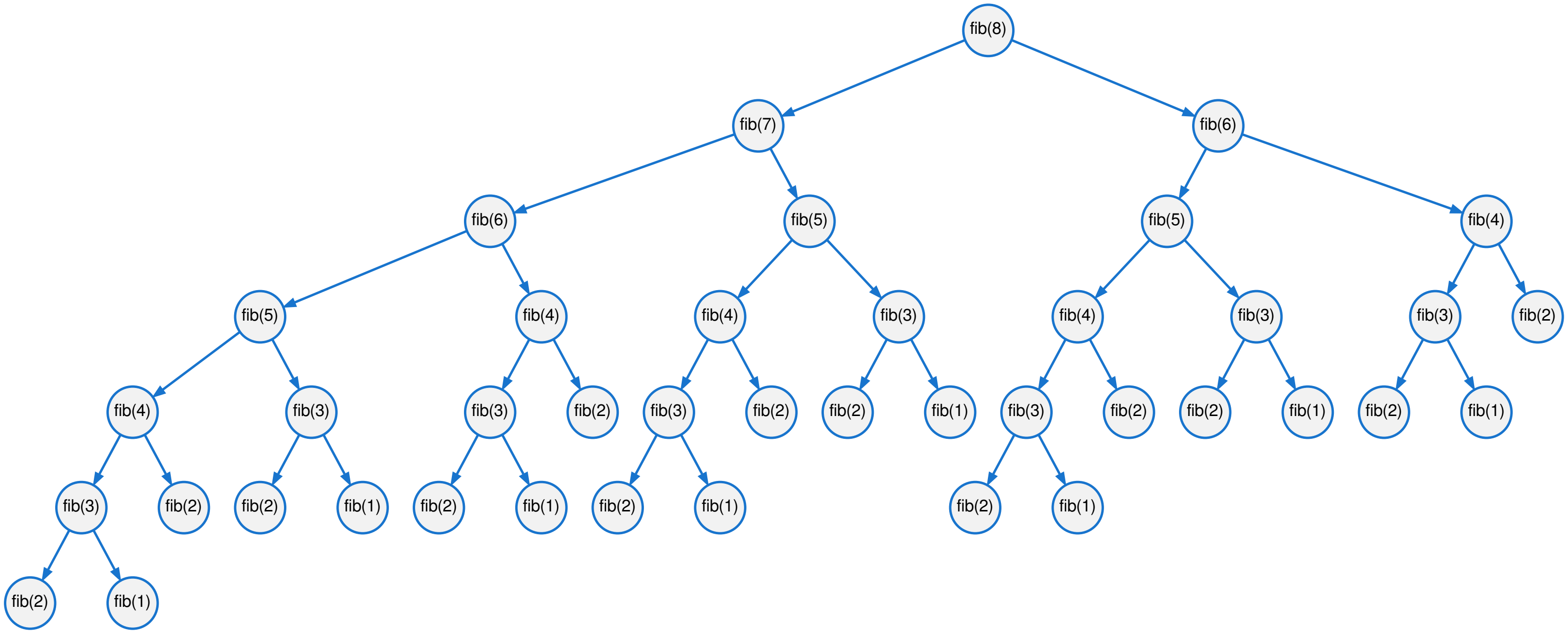
$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Die zwei ersten Fibbonacci-Zahlen sind beide 1, jede andere Fibbonacci-Zahl berechnet sich aus der Summe ihrer beiden Vorgänger.

Fibonacci-Zahlen in Java

```java
public static long fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Motivation: Fibonacci-Zahlen

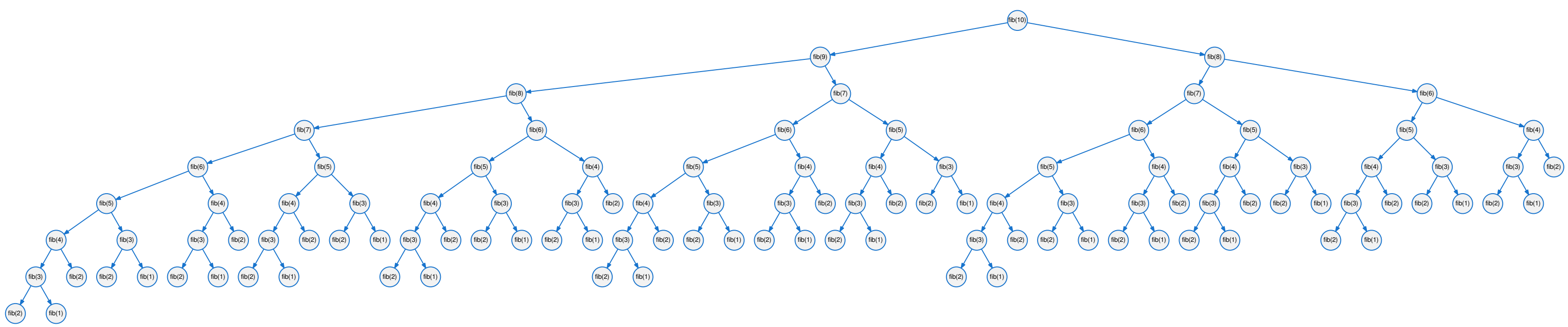


Beobachtung

Die gleichen, unveränderlichen Werte werden mehrfach berechnet.

Motivation: Fibonacci-Zahlen

- die gleichen, unveränderlichen Werte werden mehrfach berechnet
  - und zwar nicht nur konstant ein oder zwei Mal
  - sondern in Abhängigkeit von n
- für `n = 10` schaut der Aufrufbaum schon so aus:

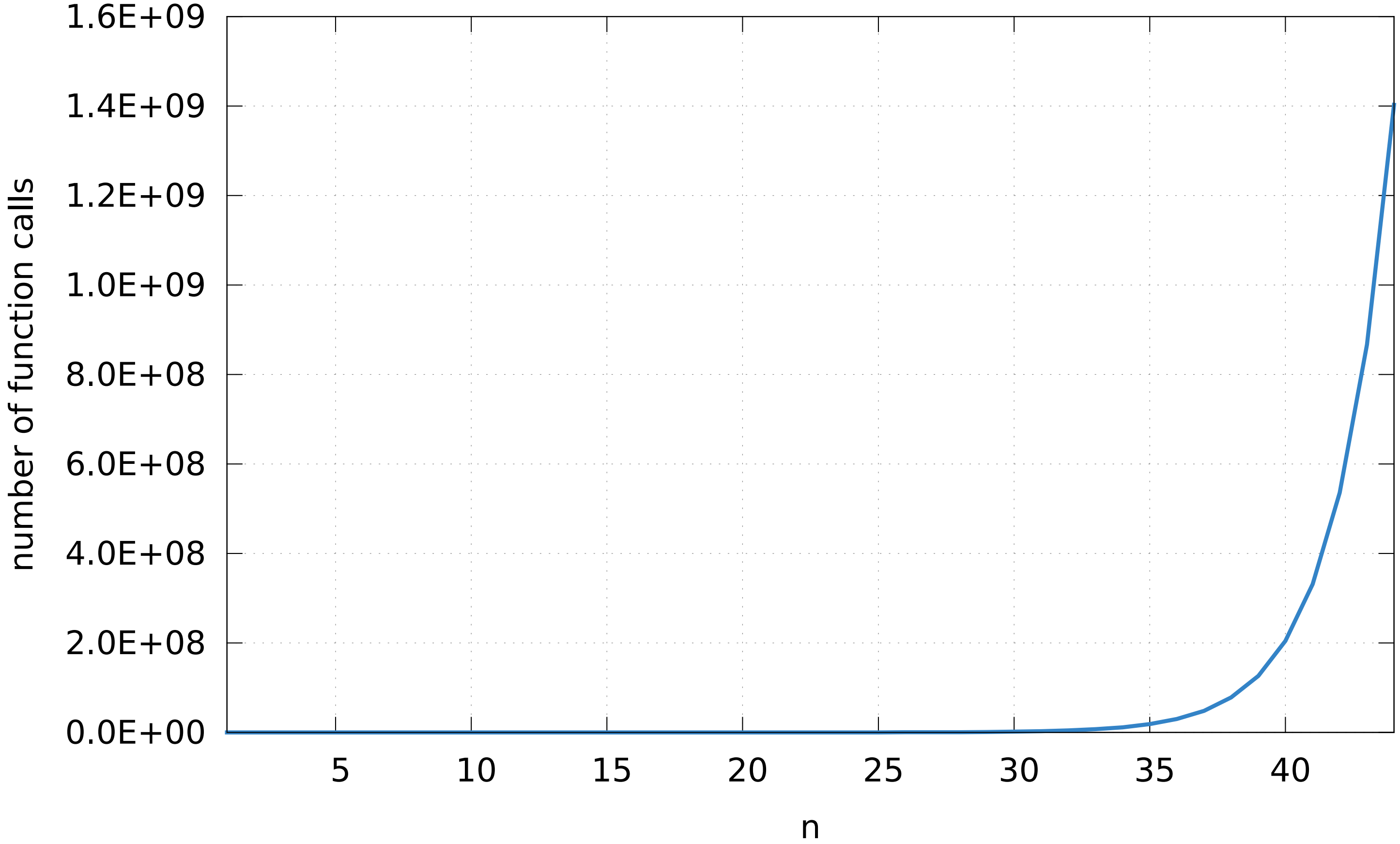


Beobachtung

Die Anzahl der Funktionsaufrufe scheint für steigendes n förmlich zu explodieren.
Stichwort: kaskadenförmige Rekursion!

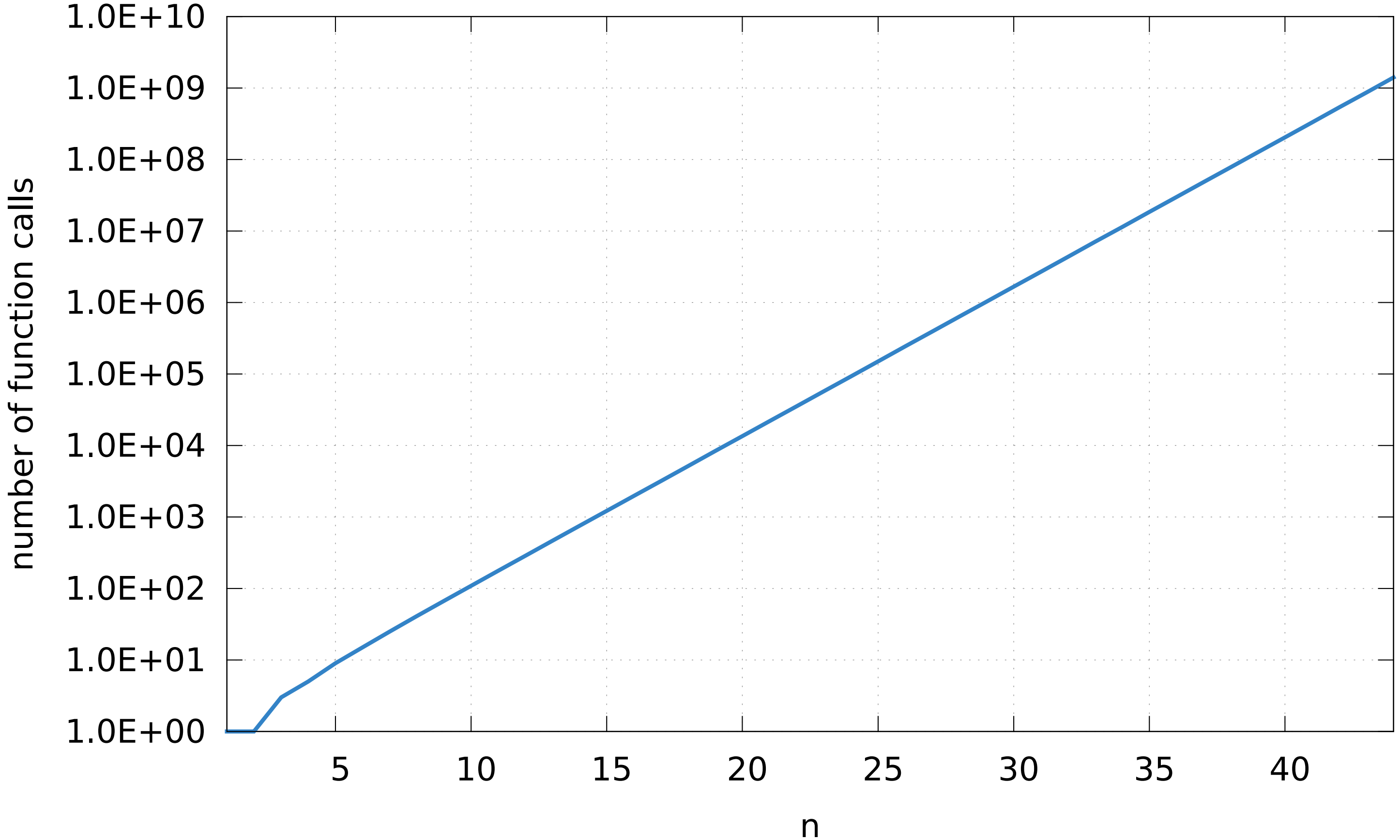
Fibonacci-Zahlen: Naive Implementierung

Anzahl der Funktionsaufrufe



Fibonacci-Zahlen: Naive Implementierung

Anzahl der Funktionsaufrufe



Dynamische Programmierung (*DP*)

- gegeben sei ein komplexes (Optimierungs-)Problem mit f. Eigenschaften:
  - Problem lässt sich in einfachere Teilprobleme aufteilen
  - Teilprobleme „überlappen“ sich

  $\leadsto$ die gleichen Teilprobleme werden mehrfach bearbeitet

$\leadsto$ Idee der Dynamischen Programmierung:

- jedes Teilproblem höchstens einmal bearbeiten
- Lösungen der Teilprobleme zwischenspeichern und wiederverwenden

Dynamische Programmierung: Idee

Memoization

Einmal berechnete Teillösungen werden in einer „Tabelle“ zwischengespeichert.

Lookup

Bevor die Lösung eines Teilproblems aufwändig berechnet wird, wird zuerst in der Tabelle nachgeschaut, ob für dieses Teilproblem bereits eine Lösung berechnet wurde. Falls ja, wird diese Lösung verwendet.

(Kleine) Probleme des DP-Ansatzes

- Parameterraum muss bekannt und beschränkt sein
- in der Tabelle muss eine Unterscheidung von berechneten und noch nicht berechneten Werten möglich sein

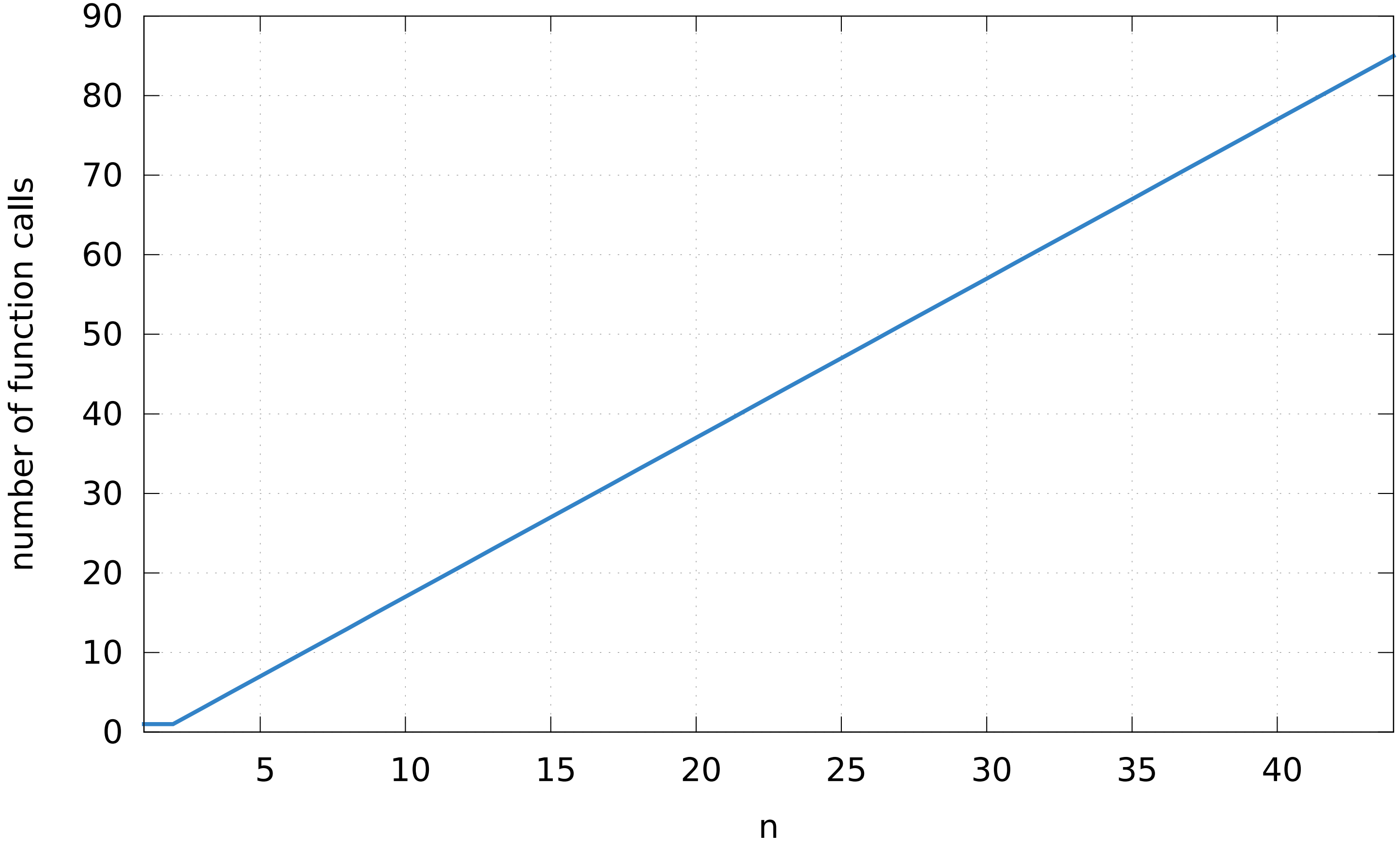
Fibonacci-Zahlen: Implementierung mit DP

Fibonacci mit Memoization

```java
public static long fibDP(int n) {
    long[] table = new long[n+1]; // Tabelle für die Memoization
    return fibHelper(n, table);
}

public static long fibDPHelper(int n, long[] table) {
    // Lookup
    if (table[n] != 0) {
        return table[n];
    }

    long result;
    if (n == 1 || n == 2) {
        result = 1;
    } else {
        result = fibDPHelper(n-1, table) + fibDPHelper(n-2, table);
    }
    // Memoization
    table[n] = result;

    return result;
}
```

Fibonacci-Zahlen: Implementierung mit DP

Anzahl der Funktionsaufrufe



Zielkonflikt: Laufzeit $\leftrightarrow$ Speicherplatz

Problem...

Tabelle kann je nach „Definitionsbereich“ der Parameter sehr groß werden, insbesondere bei mehreren Parametern (Kreuzprodukt!).

Aber...

Lösungen mit DP im Allgemeinen wesentlich schneller als „naive“ Lösungen.

Deswegen...

Häufig wird ein höherer Speicherverbrauch zugunsten einer besseren Laufzeit gerne in Kauf genommen ☺

Von der Rekursion zur Memoization

Vorgehen

- rekursive Version implementieren
- Tabelle zum Zwischenspeichern der Werte erzeugen und initialisieren
- rekursive Version um *Lookup*/*Memoization* erweitern
  - Tabelle als zusätzliche Eingabe
  - zu berechnenden Wert in der Tabelle nachschlagen ($\rightsquigarrow$ Lookup)
  - falls bereits berechnet: Wert direkt zurückgeben
  - ansonsten:
    - Wert rekursiv berechnen
    - Wert in der Tabelle zwischenspeichern ($\rightsquigarrow$ Memoization)
    - Wert zurückgeben

Beispiel: Binomialkoeffizienten

Berechnung des Binomialkoeffizienten

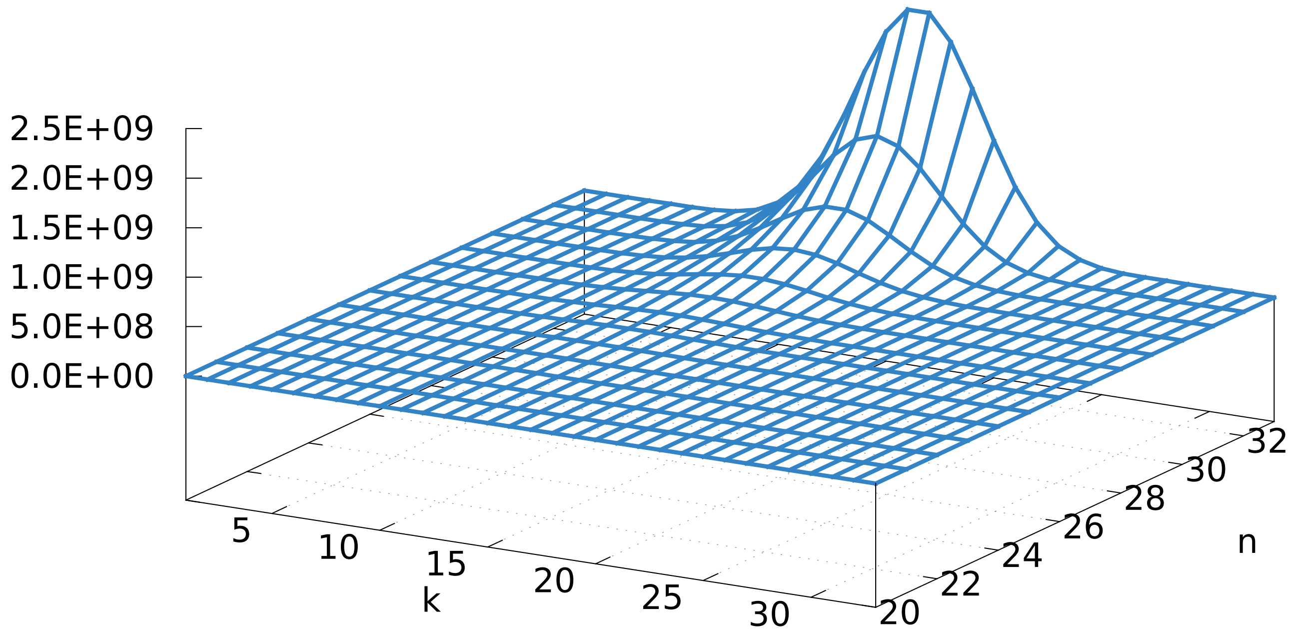
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1$$

Naive Implementierung (ohne Memoization)

```java
public static long binomBad(int n, int k) {
    if (k == 0 || n == k) {
        return 1;
    }
    return binomBad(n-1, k-1) + binomBad(n-1, k);
}
```

Binomialkoeffizienten: Naive Implementierung

Anzahl der Funktionsaufrufe



Binomialkoeffizienten: Implementierung mit DP

Bessere Implementierung (mit Memoization)

```java
public static long binomNice(int n, int k) {
    long[][] table = new long[n+1][];
    for (int i = 0; i < table.length; i++) {
        table[i] = new long[i+1];
    }
    return binomNiceHelper(n, k, table);
}

public static long binomNiceHelper(int n, int k, long[][] table) {
    // Lookup
    if (table[n][k] != 0) {
        return table[n][k];
    }

    long result;

    if (k == 0 || k == n) {
        result = 1;
    } else {
        result = binomNiceHelper(n-1, k-1, table) + binomNiceHelper(n-1, k, table);
    }
    // Memoization
    table[n][k] = result;

    return result;
}
```

Binomialkoeffizienten: Verbesserungsvorschlag (I)

Bisher

- Tabelle wird lokal für einen Aufruf von `binomNice()` angelegt
- für erneuten Aufruf muss Tabelle erneut gefüllt werden

Idee

- Tabelle global speichern und für alle Aufrufe verwenden
- Werte müssen nur noch einmal überhaupt berechnet werden
  - für zweiten Aufruf nur noch konstante Zeit (*Lookup*) notwendig
- Nachteil: „Wertebereich“ muss festgelegt werden

Binomialkoeffizienten: Verbesserungsvorschlag (II)

Noch bessere Implementierung

```
private static final int MAX_VALUE = 1000;
private static long[][] table = null;

static { // wird ausgefuehrt, bevor die Klasse das erste Mal betreten wird
    table = new long[MAX_VALUE+1][];
    for (int i = 0; i < table.length; i++) {
        table[i] = new long[i+1];
}   }

public static long binomEvenBetter(int n, int k) {
    // Alternative: if (table == null) { /* code of static block */ }
    if (n > MAX_VALUE) { return -1; }
    return binomEvenBetterHelper(n, k);
}

private static long binomEvenBetterHelper(int n, int k) {
    if (table[n][k] != 0) { return table[n][k]; } // Lookup

    long result;
    if (k == 0 || k == n) { result = 1; }
    else result = { binomEvenBetterHelper(n-1, k-1) + binomEvenBetterHelper(n-1, k); }

    table[n][k] = result; // Memoization

    return result;
}
```

Vergleich der drei Varianten

Kleines Testprogramm

```
(34 choose 13) with different solutions:
BAD: Took 11106 ms, result: 927983760
NICE: Took 0 ms, result: 927983760
EVEN BETTER: Took 0 ms, result: 927983760

(912 choose 512) with even better solution:
ROUND 1: Took 12 ms
ROUND 2: Took 0 ms
```

Bottom-Up-Berechnung

Bisher

Berechnungen top-down mittels rekursiver Aufrufe

- $fib(n) = fib(n-1) + fib(n-2)$
- $bin(n,k) = bin(n-1,k-1) + bin(n-1,k)$

Jetzt

Berechnungen bottom-up, beginnend bei den „Basisfällen“

Dazu...

Memoization-Tabelle in bestimmter Reihenfolge durchlaufen, sodass Werte immer schon zur Verfügung stehen, wenn diese gebraucht werden

Fibonacci *bottom-up*

Erkenntnis

- für den *n*-ten Werts werden die beiden „Vorgänger“ in der Tabelle benötigt

$\leadsto$ Tabelle „von links nach rechts“ durchlaufen

(Naive) Bottom-Up-Implementierung der Fibonacci-Folge

```java
public static int fibBottomUp(int n) {
  int[] table = new int[n + 1];
  table[0] = 0;
  table[1] = 1;
  for (int i = 2; i <= n; ++i) {
    // table[i-2] und table[i-1] wurden zuvor bereits berechnet
    // und haben deshalb bereits den richtigen Wert
    table[i] = table[i - 2] + table[i - 1];
  }
  return table[n];
}
```

Backtracking

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Motivation

Schatzsuche

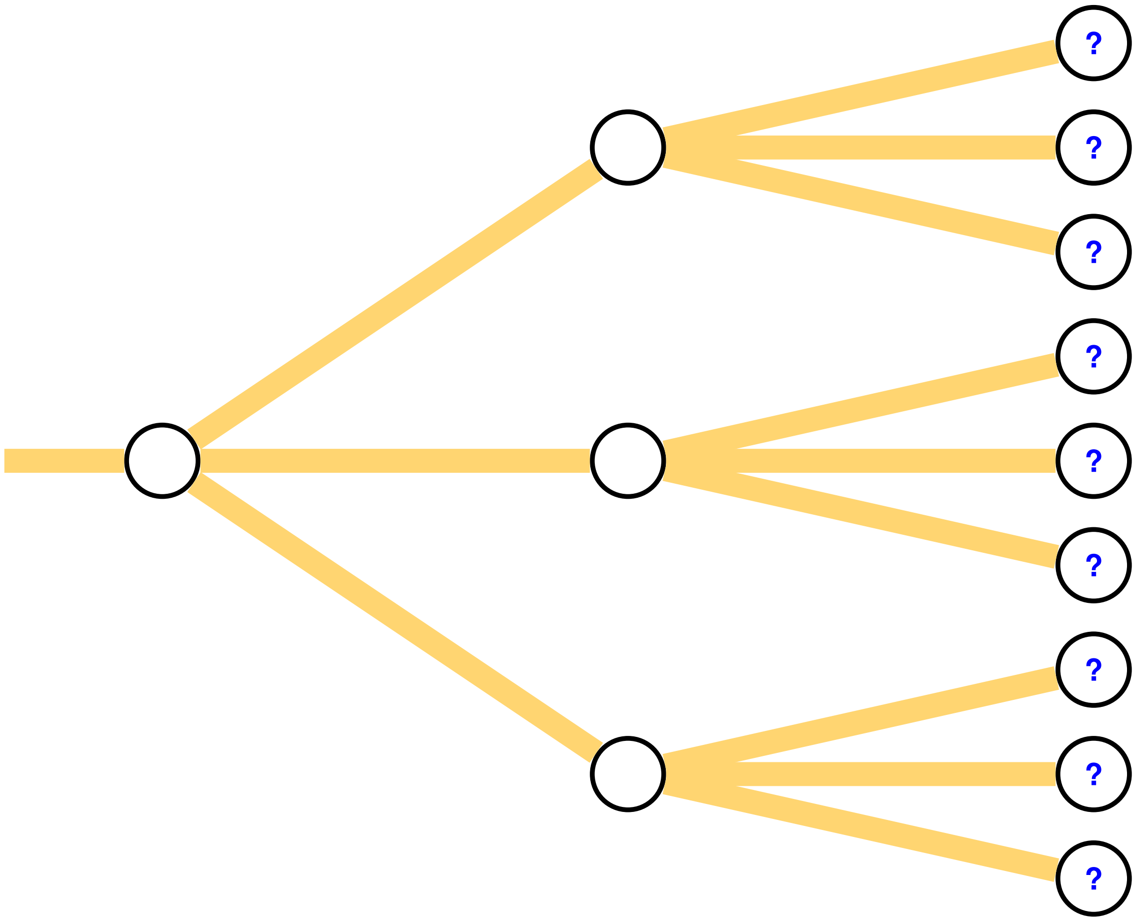
In einem Labyrinth ist ein Schatz versteckt. Da Alice und Bob bereits mit ihrer AuD-Hausaufgabe für diese Woche fertig sind, beschließen sie, ihn zu suchen.

Mögliche Vorgehensweisen

- planlos/zufällig abbiegen und hoffen, den Schatz irgendwann zu finden
- sich aufteilen und parallel an verschiedenen Stellen suchen
  $\leadsto$ *Parallele und Funktionale Programmierung*
- strukturiert alle möglichen Pfade durchprobieren
- ...

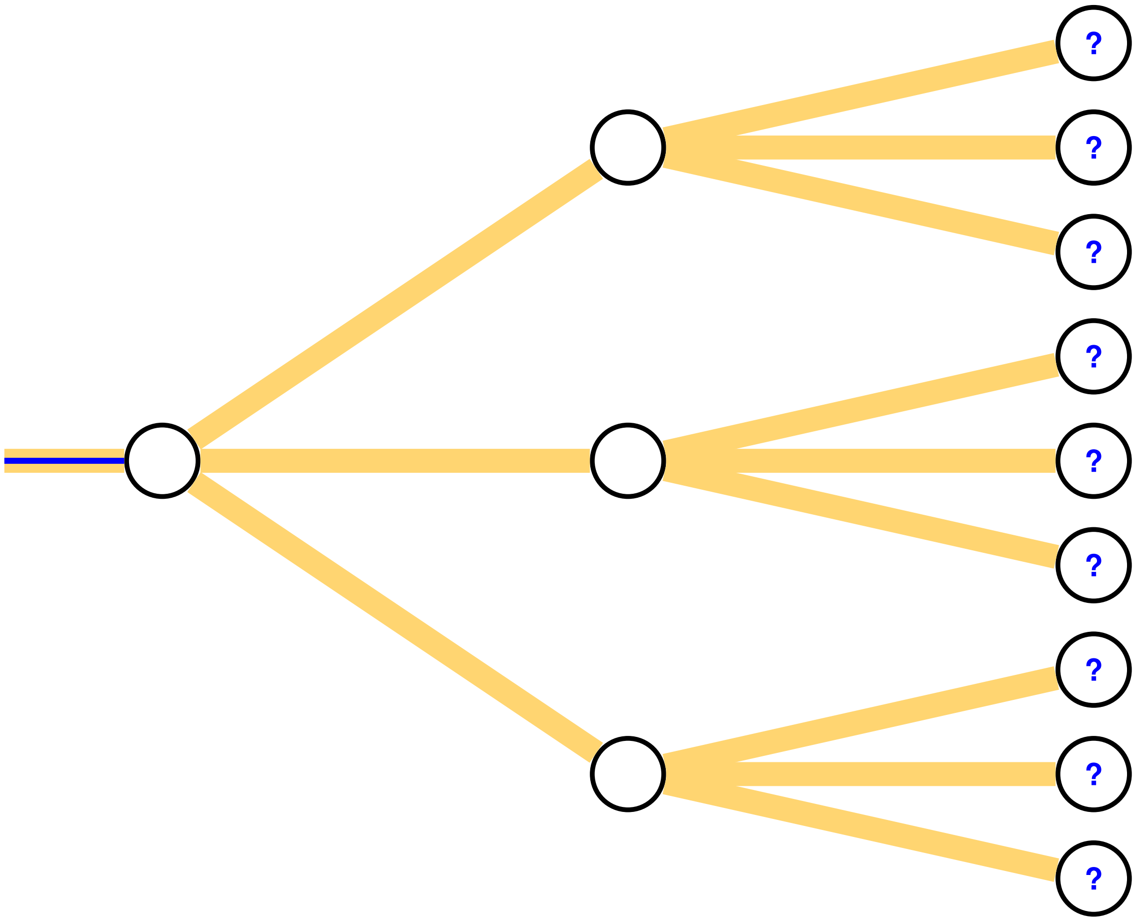
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



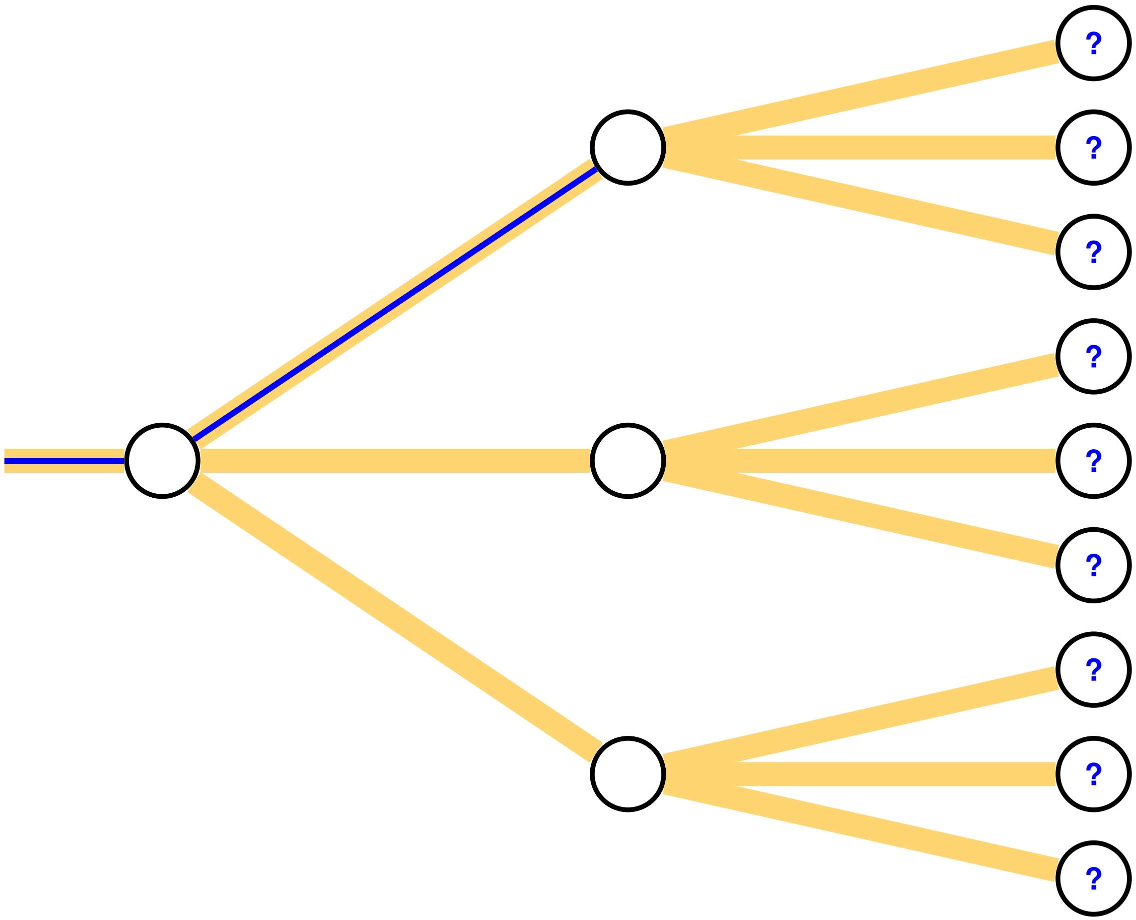
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



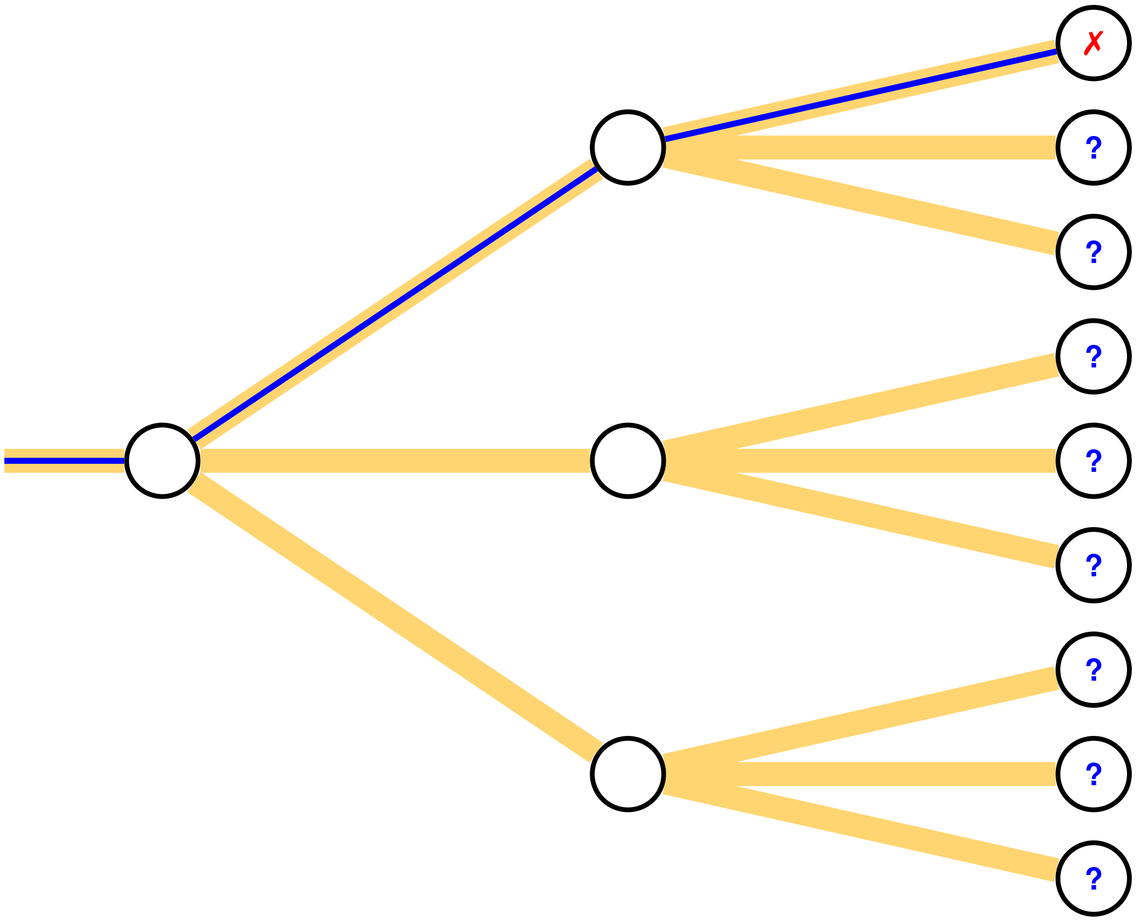
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



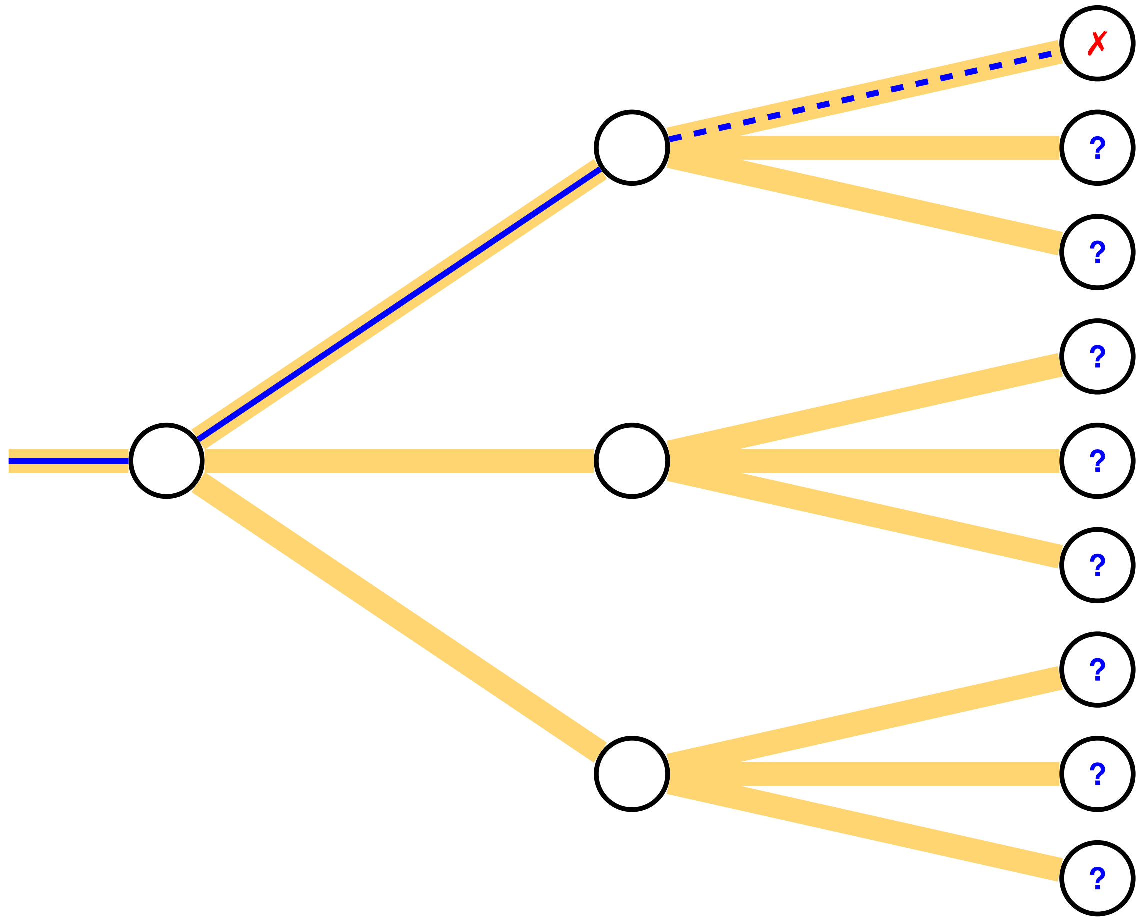
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



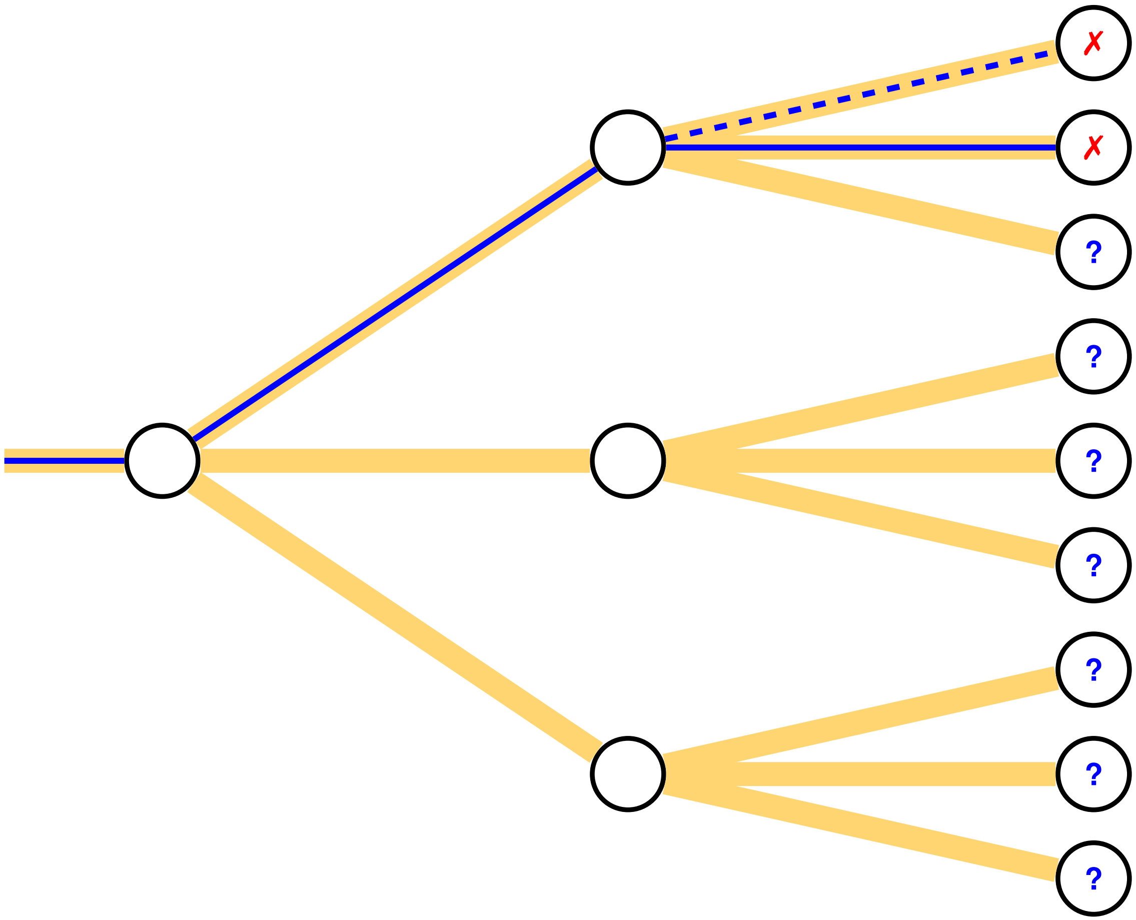
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



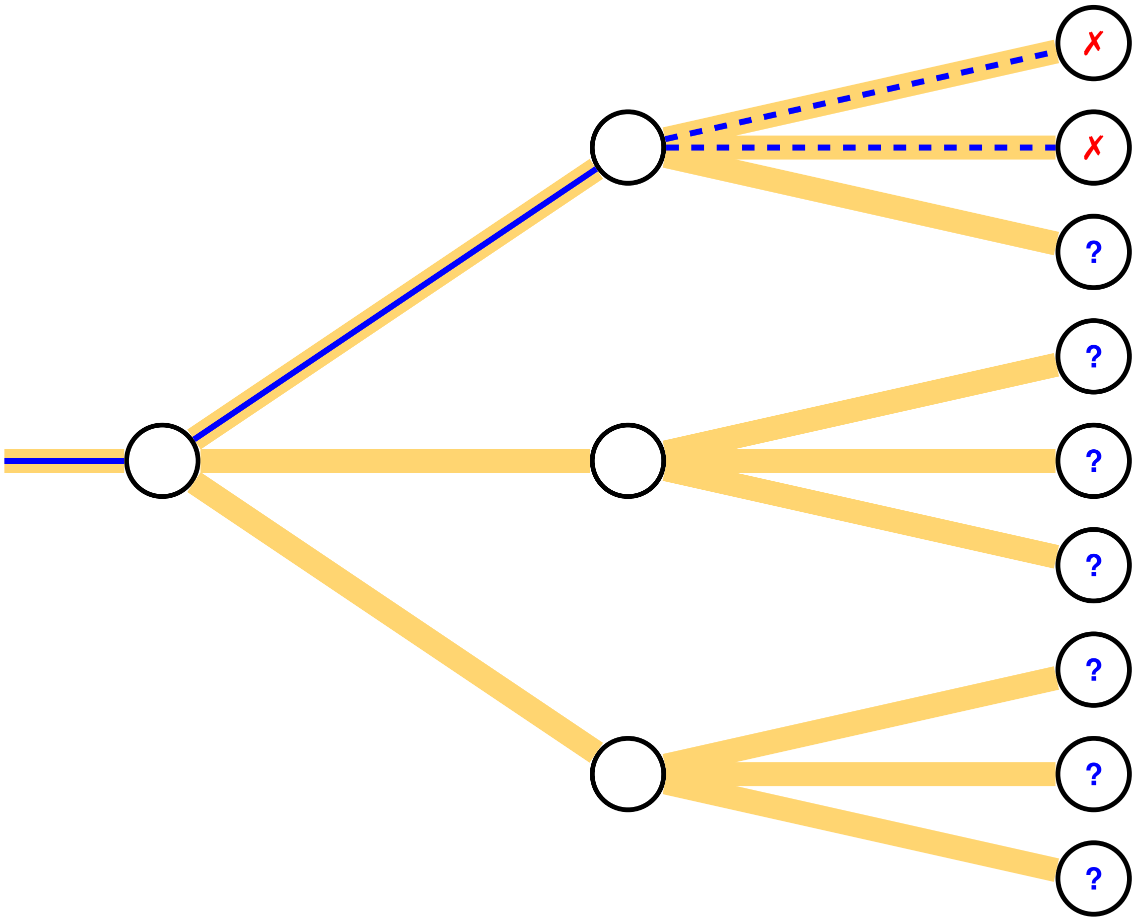
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



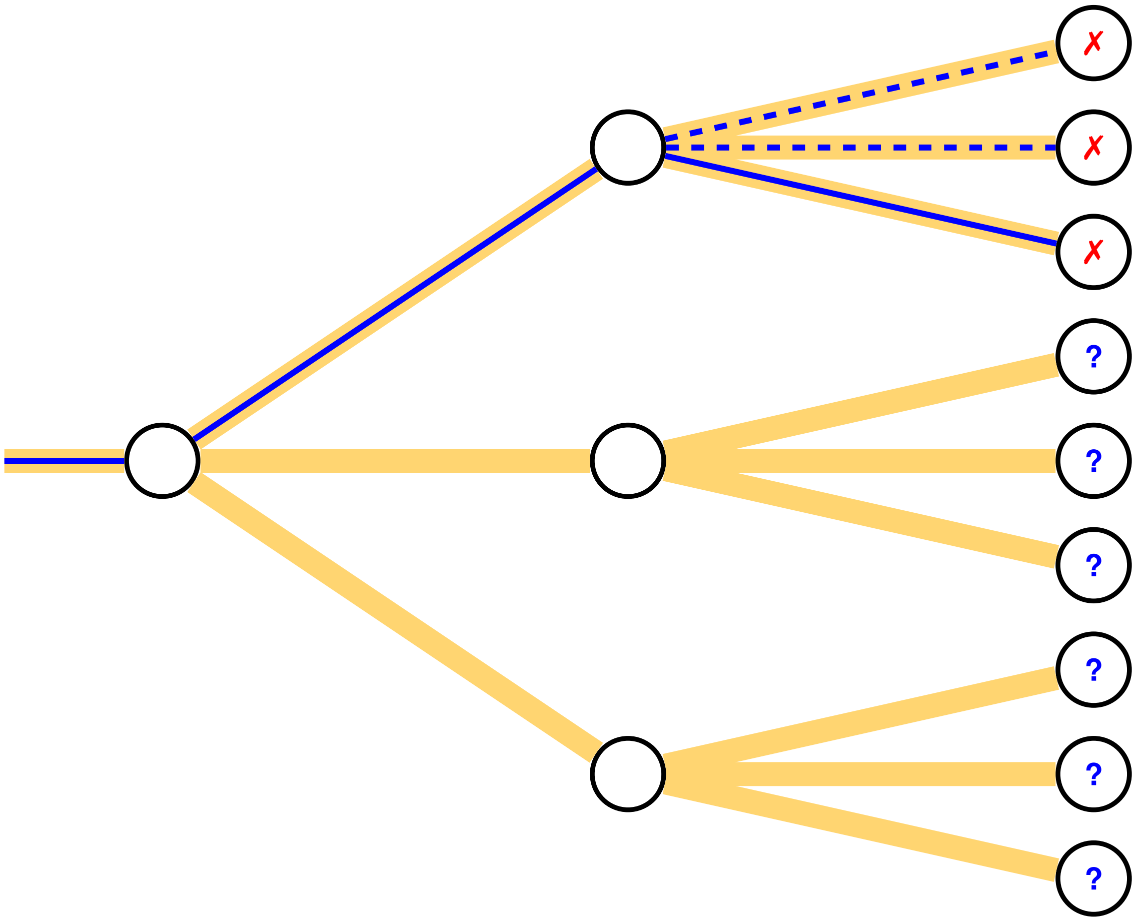
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



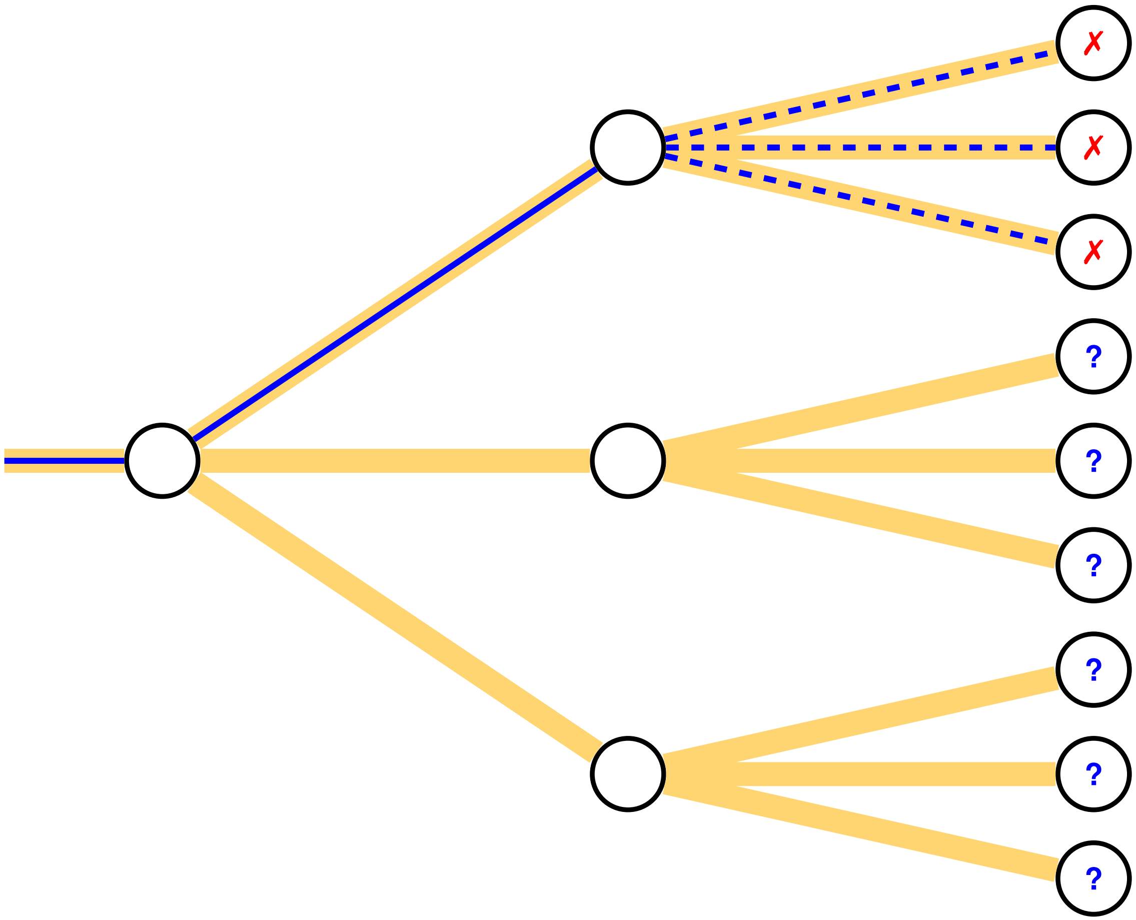
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



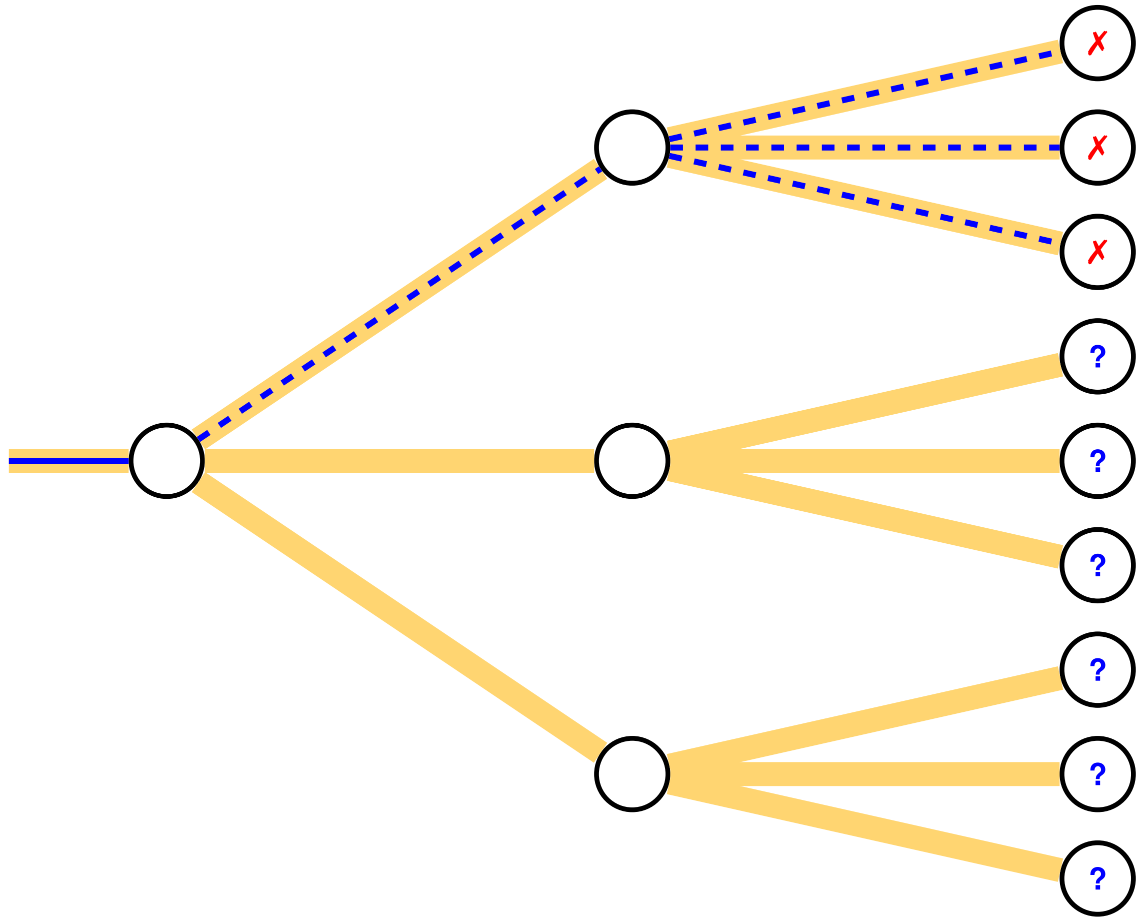
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



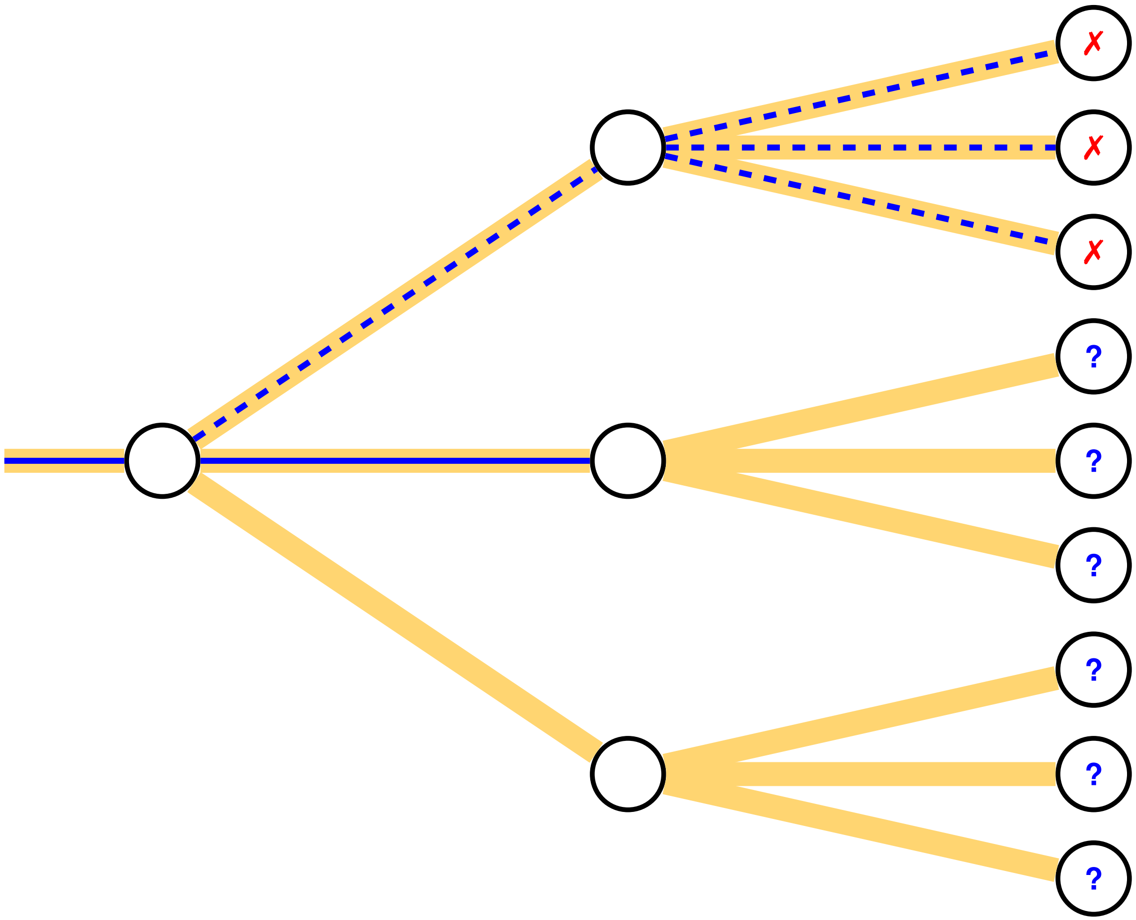
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



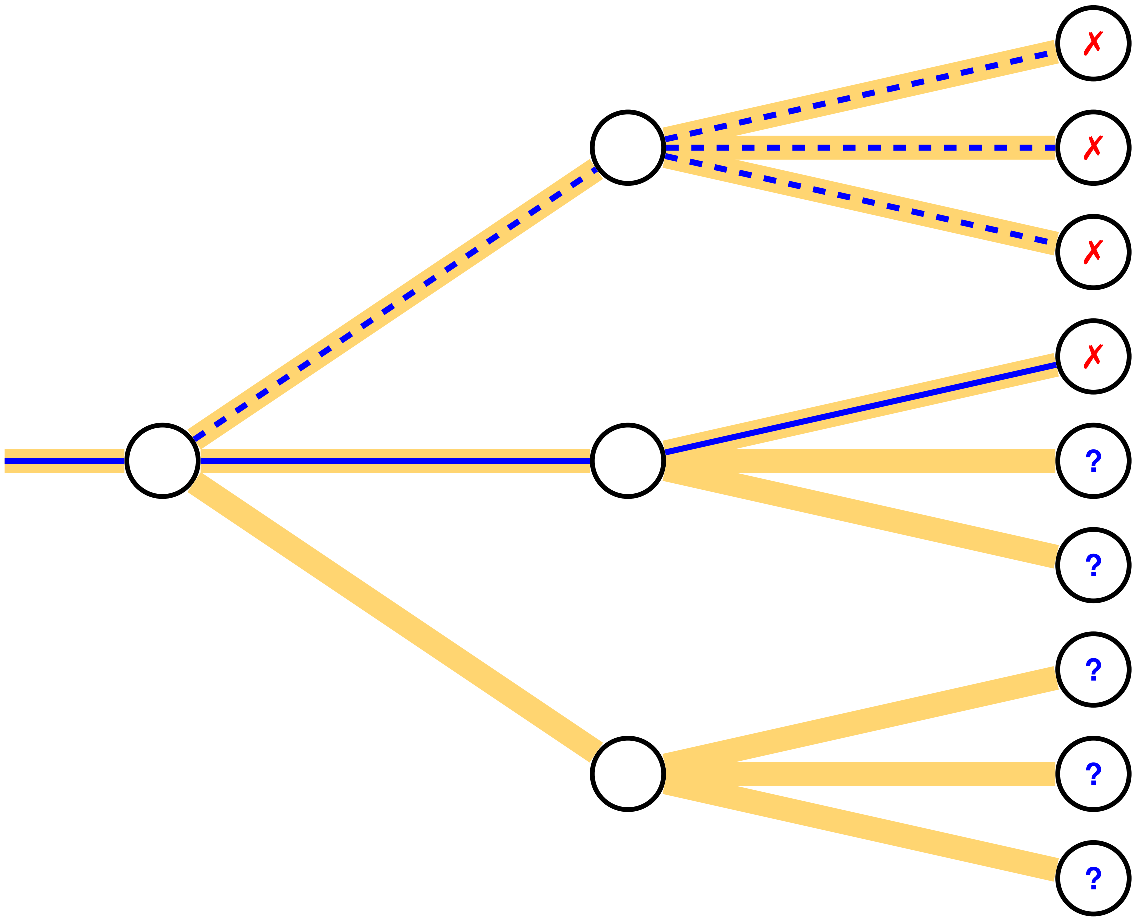
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



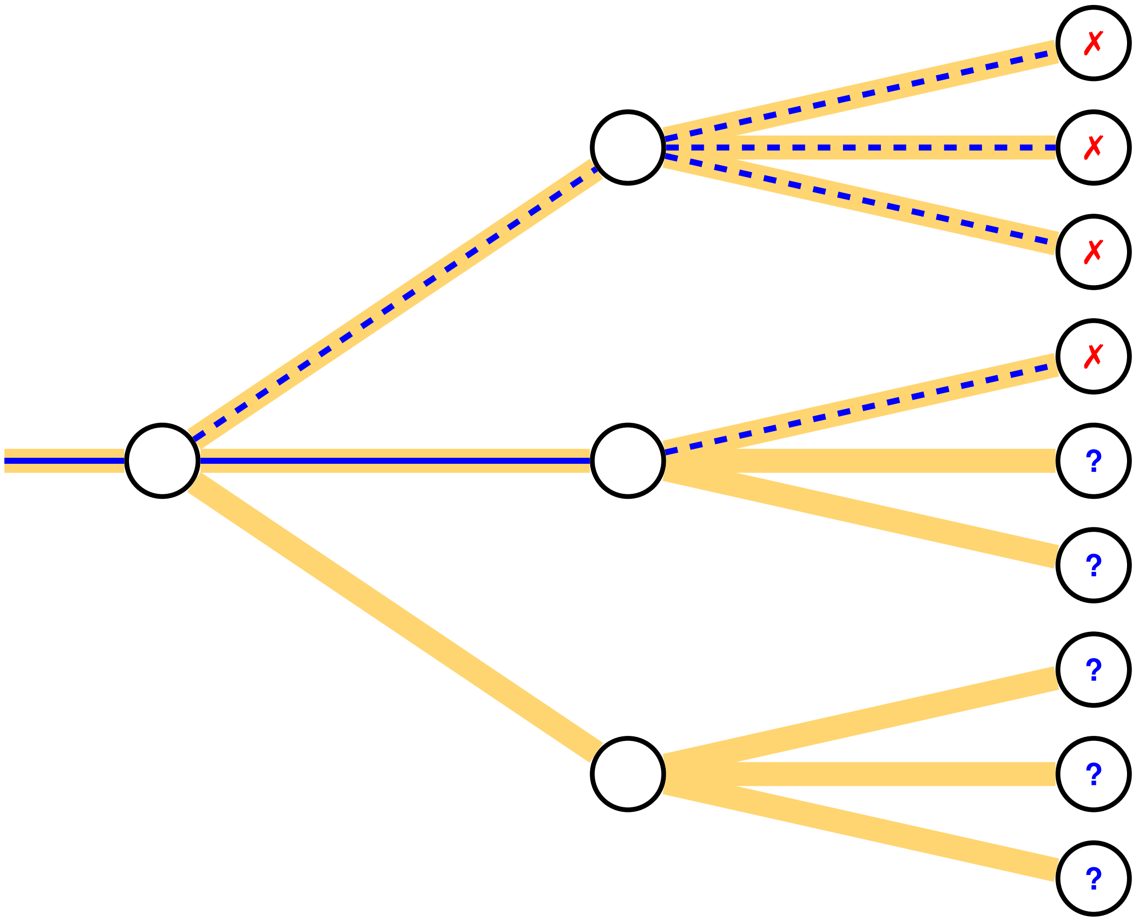
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



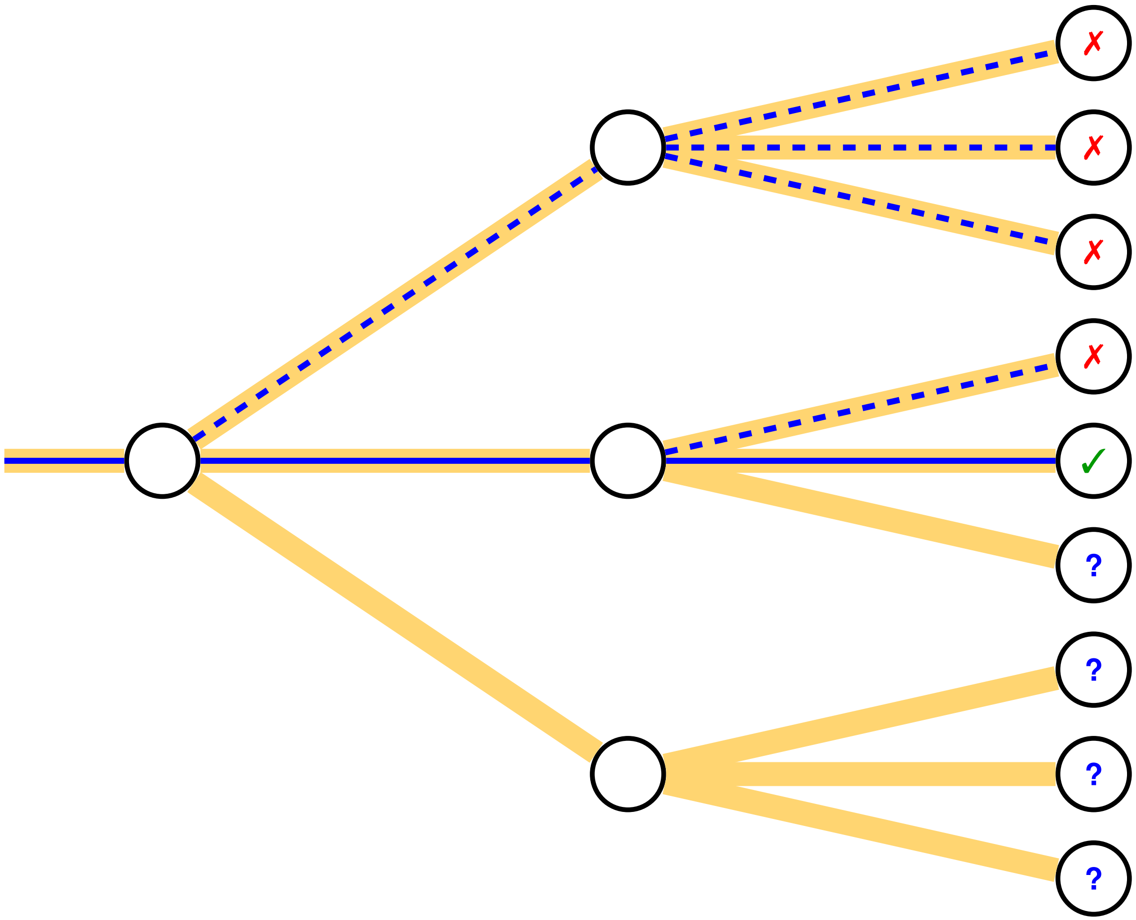
Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



Schatzsuche

Strukturiert alle möglichen Pfade durchprobieren



Backtracking

- Backtracking (*Rücksetzverfahren*):
  - Problemlösungsverfahren
  - systematisches Durchsuchen des Suchraums
  - dabei Anwendung von *trial-and-error* (*Versuch-Und-Irrtum*)
- falls für eine „Entscheidung“ mehrere Möglichkeiten existieren:
  - alle Möglichkeiten rekursiv durchprobieren
  - falls eine Möglichkeiten zum Erfolg führt: gut ☺
    - Suche kann i.A. abgebrochen werden
  - andernfalls:
    - Entscheidung war wohl falsch...
    - Entscheidung „rückgängig machen“

$\leadsto$ Backtracking findet garantiert eine Lösung, so sie denn existiert

Schema

- Backtracking läuft fast immer nach demselben Schema ab
- wenn man sich passende Methoden definiert, kann man für die Implementierung oft dasselbe Grundgerüst verwenden
- erforderliche Methoden sind:
  - `isFinal()` überprüft, ob eine Lösung gefunden wurde

    $\rightsquigarrow$ z.B.: ist ein Sudoku vollständig gefüllt?
  - `getExtensions()` gibt alle möglichen Erweiterungen zurück

    $\rightsquigarrow$ z.B.: alle erlaubte Zahlen für das aktuelle Feld
  - `apply()` verändert den aktuellen Zustand

    $\rightsquigarrow$ z.B.: schreibe die aktuelle Zahl ins aktuelle Feld
  - `revert()` stellt den vorherigen Zustand wieder her

    $\rightsquigarrow$ z.B.: lösche die letzte Zahl aus dem aktuellen Feld

Backtracking: Grundgerüst

Backtracking: Grundgerüst

```
static int[][] backtrack(int[][] state) { // z.B. Sudoku
    if (isFinal(state)) { // z.B. Sudoku komplett gefüllt
        return state;
    } else {
        // z.B. erlaubte Zahlen für aktuelles Feld
        int[] candidates = getExtensions(state);
        for (int i = 0; i < candidates.length; i++) {
            int c = candidates[i];
            state = apply(state, c); // z.B. Zahl setzen
            if (backtrack(state) != null) { // Rekursion
                return state;
            }
            state = revert(state, c); // z.B. Zahl löschen
        }
        return null; // keine Lösung gefunden -> Schritt zurück
    }
}
```

Hinweise zu den Aufgaben

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Induktionsbeweis hokuspokus

Hinweis 1

$$3^2 = 9$$

(vereinfacht die Rechnung ungemein ☺)

Hinweis 2

- richtige Ansätze geben auch schon Punkte
- wenn ein Fehler in der Rechnung ist, kann es trotzdem Folgefehlerpunkte geben

Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2
Programmiersysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT