Rekursive Programmierung

Prof. Dr. Christian Becker

Universität Stuttgart, Institut für Parallele und Verteilte Systeme

15. Mai 2025

Inhalt dieser Vorlesung

Literatur

Eric S. Roberts and Julie Zelenski
Programming Abstractions in C++
2014, Pearson, ISBN 978-0133454840

Eric S. Roberts
Thinking Recursively
1986, J. Wiley, ISBN 978-0-471-81652-2

Inhalt dieser Vorlesung

Einführung in Rekursive Programmierung

Thinking Recursively: Invarianten, Preconditions, Postconditions

Ausblick und Zusammenfassung

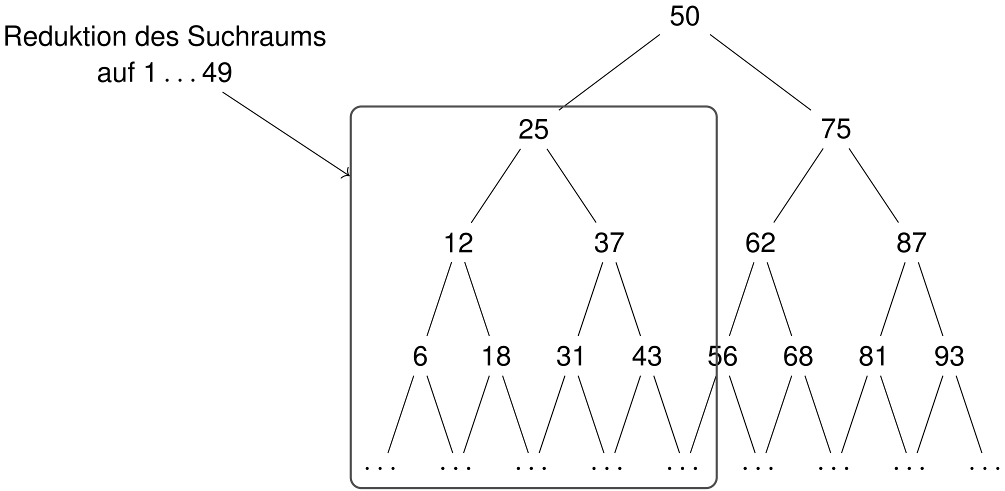
Einführendes Beispiel

Guess a Number (2 Spieler):

1. Spieler 1 wählt eine zufällige Zahl n zwischen 1 und 100
2. Spieler 2 rät eine Zahl k
   - $k < n$: Spieler 1 antwortet mit “zu klein”
   - $k > n$: Spieler 1 antwortet mit “zu groß”
3. Wiederhole Schritt 2 bis die die gewählte Zahl gefunden wurde

Was ist die optimale Spielstrategie für Spieler 2?

Einführendes Beispiel



Einführendes Beispiel

Wie viele Schritte werden maximal benötigt?

- $N \in \{1\}$: 1 Schritt
- $N \in \{1, 2, 3\}$: 2 Schritte
- $N \in \{1, \ldots, 7\}$: 3 Schritte
- $\ldots$
- $N \in \{1, \ldots, 127\}$: 7 Schritte
- Allgemein: $\lceil log_2(N + 1) \rceil$ Schritte

Rekursion

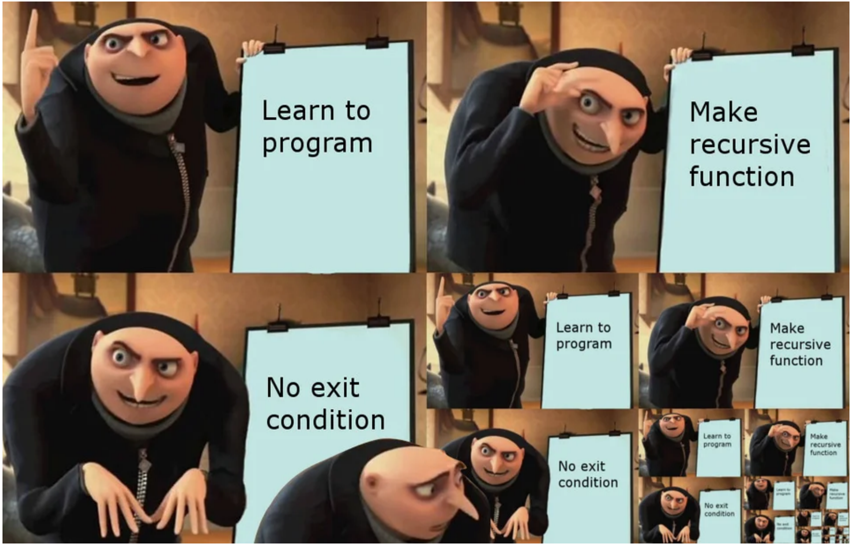
Definition (Rekursion)

Rekursion nutzt die Lösung von mehreren kleineren Probleminstanzen, um das gesamte Problem zu lösen.

Rekursion erlaubt oftmals eine elegante und kompakte Darstellung!
Beispiel: Fibonacci Reihe

- Anfangswerte:
  - $Fib(0) = 0$
  - $Fib(1) = 1$
- Für $N > 1$:
  - $Fib(N) = Fib(N-2) + Fib(N-1)$

Common Pitfalls...



Abbruchbedingungen

Falsch (Endlose Rekursion)

```java
int fib(int n) {
  return fib(n-1) + fib(n-2);
}
```

Java

Richtig (Abbruchbedingung $n \in \{0, 1\}$)

```java
int fib(int n) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Java

Jede rekursive Funktion benötigt eine oder mehrere Abbruchbedingungen!

Rekursionsschritt

Frage: Terminiert die folgende Funktion für alle Eingaben *n*?

```java
int collatz(int n) {
  if (n == 1) {
    return true;
  } else if (n % 2 == 0) {
    return collatz(n/2);
  } else {
    return collatz(3*n+1);
  }
}
```

Java

Rekursionsschritt

Frage: Terminiert die folgende Funktion für alle Eingaben *n*?

```java
int collatz(int n) {
  if (n == 1) {
    return true;
  } else if (n % 2 == 0) {
    return collatz(n/2);
  } else {
    return collatz(3*n+1);
  }
}
```

Java

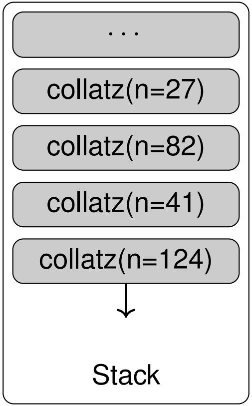
Jeder rekursive Funktionsaufruf sollte eine “einfachere” Probleminstanz lösen!

* Tatsächlich ist bekannt dass die Reihe bis zu Werten von 2.95×10^{20} konvergiert.

Randnotiz: Tail Recursion

```java
int collatz(int n) {
  if (n == 1) {
    return true;
  } else if (n % 2 == 0) {
    return collatz(n/2);
  } else {
    return collatz(3*n+1);
  }
}
```

Java



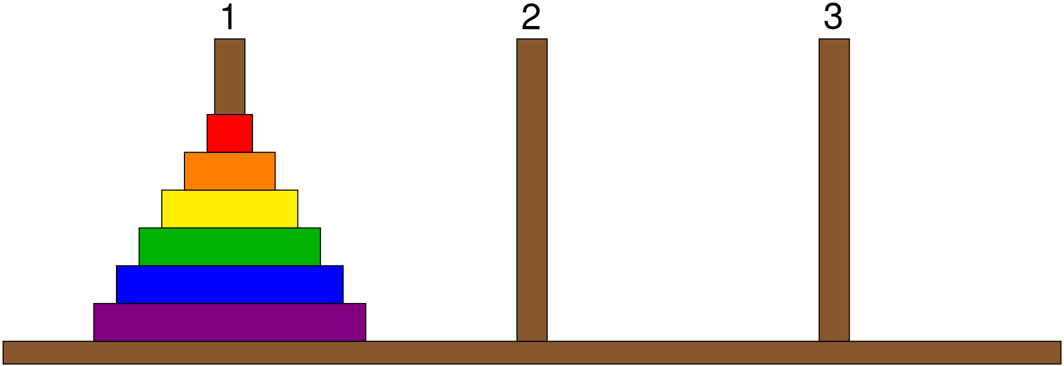
Quizfrage

1. Werden bei dem rekursiven Funktionsaufruf für $n = 124$ noch die vorherigen Werte benötigt $(41, 82, 27, \ldots)$?
2. Was wäre eine Alternative die weniger Platz auf dem Stack benötigt?

Inhalt dieser Vorlesung

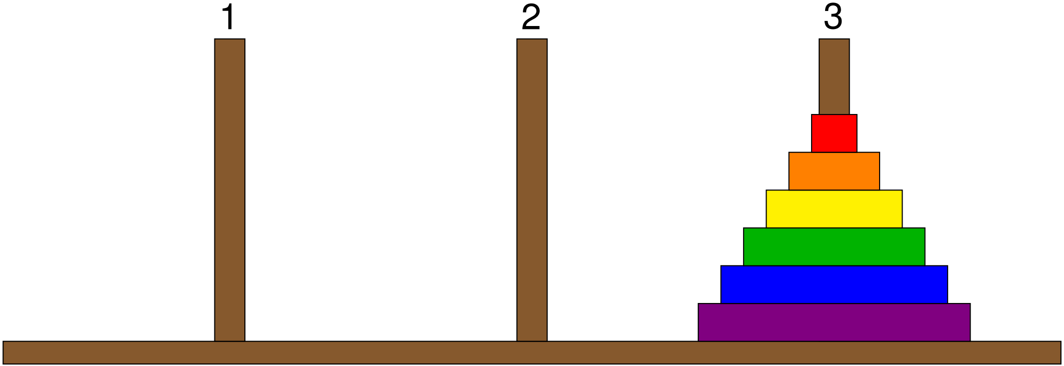
Towers of Hanoi

Start: N gelochte Scheiben sind an dem linken Stab der Größe nach angeordnet.



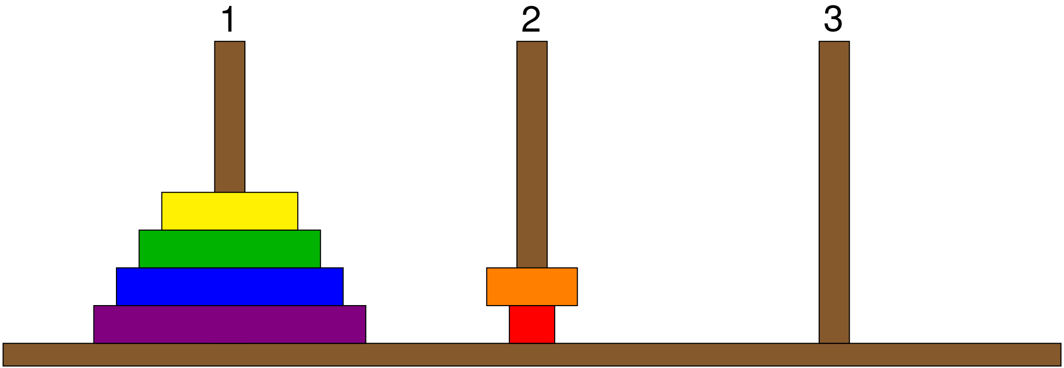
Towers of Hanoi

Ziel: Die Scheiben werden alle auf den rechten Stab bewegt.



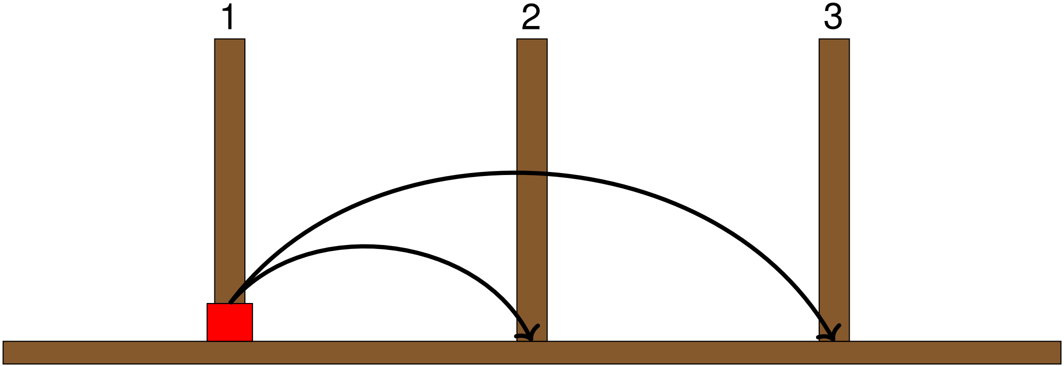
Towers of Hanoi

Einschränkung: Größere Scheiben dürfen nicht auf kleinere Scheiben gelegt werden!



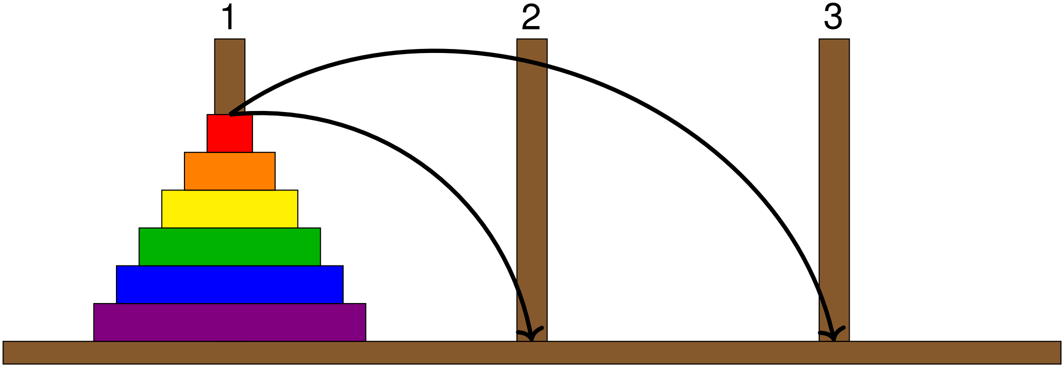
Thinking Recursively: Rekursionsbeginn

Start Simple: Für $N = 1$ kann die Scheibe ohne Einschränkungen bewegt werden.



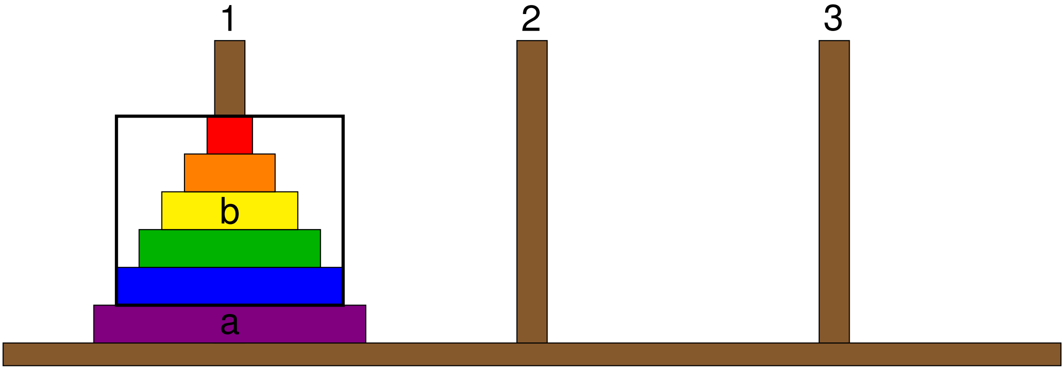
Thinking Recursively: Rekursionsbeginn

Start Simple: Für $N = 1$ kann die Scheibe ohne Einschränkungen bewegt werden.
And Expand: Selbes gilt für die rote Scheibe für beliebiges N!



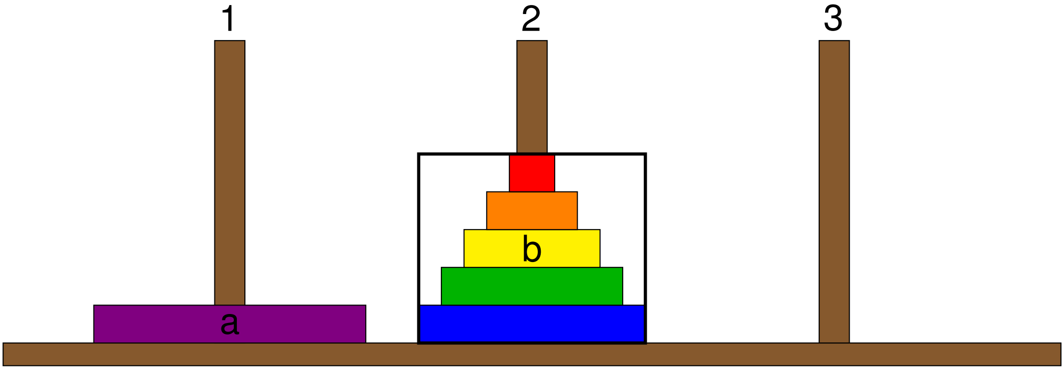
Thinking Recursively: Rekursionsschritt

Reduktion: Angenommen wir können das Problem für $N - 1$ lösen.



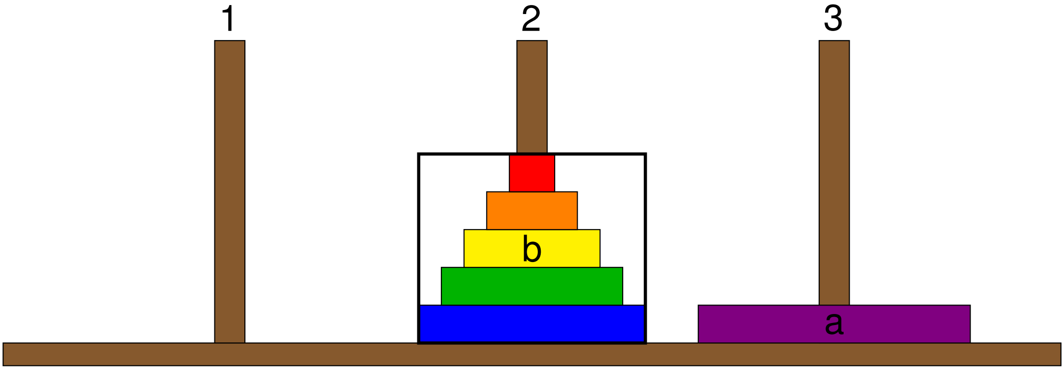
Thinking Recursively: Rekursionsschritt

Reduktion: Angenommen wir können das Problem für $N-1$ lösen.
Dann: b $\rightarrow$ 2;



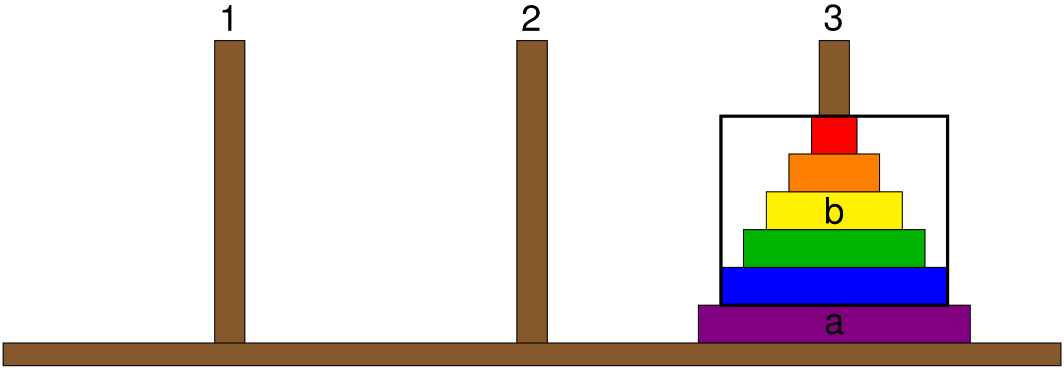
Thinking Recursively: Rekursionsschritt

Reduktion: Angenommen wir können das Problem für $N - 1$ lösen.
Dann: b $\rightarrow$ 2; a $\rightarrow$ 3



Thinking Recursively: Rekursionsschritt

Reduktion: Angenommen wir können das Problem für $N - 1$ lösen.
Dann: b $\rightarrow$ 2; a $\rightarrow$ 3; b $\rightarrow$ 3



Towers of Hanoi: Pseudocode

```java
void move(int disk, Pole source, Pole target) { ... }

void hanoi(int disks, Pole source, Pole helper, Pole target) {
  if (disks == 1) {
    // move last disk to target
    move(disks - 1, source, target);
  } else {
    // move smaller disks from source to helper pole
    hanoi(disks - 1, source, target, helper);
    // move largest disk to target
    move(disks - 1, source, target);
    // move remaining disks from helper to target
    hanoi(disks - 1, helper, source, target);
  }
}
```

Java

Zentrale Hilfsmittel für Korrektheitsbeweise

Definiere für jede (rekursive) Methode

Vorbedingungen:
Welche Eigenschaften müssen vor der Ausführung der Methode erfüllt sein?

Invarianten:
Unter der Annahme dass die Vorbedingungen erfüllt sind, welche Eigenschaften gelten innerhalb der Methode (bspw. bei jedem rekursiven Aufruf oder in jeder Schleifeniteration?).

Nachbedingungen:
Unter der Annahme dass die Vorbedingungen erfüllt sind, welche Eigenschaften gelten nach dem Ausführen der Methode?

Zentrale Hilfsmittel für Korrektheitsbeweise

Quizfrage

Definiere sinnvolle Vorbedingungen, Invarianten und Nachbedingungen für die Methoden *move* und *hanoi* aus dem vorherigen Beispiel.

Zentrale Hilfsmittel für Korrektheitsbeweise

Quizfrage

Definiere sinnvolle Vorbedingungen, Invarianten und Nachbedingungen für die Methoden *move* und *hanoi* aus dem vorherigen Beispiel.

Hier eine mögliche Formulierung (nicht zwangsweise die einzige Möglichkeit):

Vorbedingungen (move):

1. Wohldefinierte Parameter ($0 \leq \texttt{disk} < N$ und $\texttt{source.id}, \texttt{target.id} \in \{1, 2, 3\}$)
2. `disk` ist die oberste Scheibe an Pole `source`

Nachbedingungen (move):

1. `disk` ist die oberste Scheibe an Pole `target`
2. Es wurde keine andere Scheibe bewegt

Zentrale Hilfsmittel für Korrektheitsbeweise

Quizfrage

Definiere sinnvolle Vorbedingungen, Invarianten und Nachbedingungen für die Methoden *move* und *hanoi* aus dem vorherigen Beispiel.

Vorbedingungen (hanoi):

1. Wohldefinierte Parameter ($0 \leq \texttt{disk} < N$ und $\{\texttt{source.id}, \texttt{helper.id}, \texttt{target.id}\} = \{1, 2, 3\}$)
2. Alle Scheiben $0, \ldots, \texttt{disks} - 1$ liegen (der Größe nach sortiert) an Pole `source`

Nachbedingungen (hanoi):

1. Alle Scheiben $0, \ldots, \texttt{disks} - 1$ liegen (der Größe nach sortiert) an Pole `target`
2. Die anderen Scheiben $\texttt{disk}, \ldots, N - 1$ wurden nicht bewegt

Zentrale Hilfsmittel für Korrektheitsbeweise

Quizfrage

Definiere sinnvolle Vorbedingungen, Invarianten und Nachbedingungen für die Methoden *move* und *hanoi* aus dem vorherigen Beispiel.

Invarianten können benutzt werden um die Nachbedingungen zu zeigen.

Invarianten (hanoi):

1. Parameter bleiben wohldefiniert (u.A. relevant für Vorbedingungen von `move`, womit Nachbedingung 1 gezeigt werden kann)
2. Größere Scheiben werden niemals auf kleinere Scheiben gelegt (Spielregeln)
3. `disk` wird ausschließlich verkleinert (relevant für Nachbedingung 2)

Inhalt dieser Vorlesung

Rekursion

Definition (Rekursion)

Rekursion nutzt die Lösung von mehreren kleineren Probleminstanzen, um das gesamte Problem zu lösen.

Rekursive Programmierung findet viele Anwendungen, beispielsweise für

- das Traversieren von Datenstrukturen (oftmals Baumstrukturen)
- das Sortieren von Listen (bspw. Mergesort, Quicksort, Heapsort)

Es sollte jedoch sorgfältig angewandt werden

- Kann die Terminierung sichergestellt werden?
- Was sind die Vorbedingungen, Invarianten und Nachbedingungen?