LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung
Wintersemester 2018/19

Kapitel 11: Rekursion

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 9. Januar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Überblick und Ziele

- Das Prinzip der Rekursion und rekursiver Berechnungen verstehen.
- Implementierung rekursiver Methoden in Java
- Verschiedene Formen der Rekursion
- Quicksort als rekursive Methode zum Sortieren eines Arrays

Rekursive Algorithmen und Methoden

- Begriffsherkunft: lateinisch recurrere „zurücklaufen“

Definition (rekursiver Algorithmus)

Ein Algorithmus ist **rekursiv**, wenn in seiner (endlichen) Beschreibung derselbe Algorithmus wieder aufgerufen wird.

- Ein rekursiver Algorithmus ist daher selbstbezüglich definiert
- In Java können rekursiver Algorithmen durch **rekursive Methoden** implementiert werden.

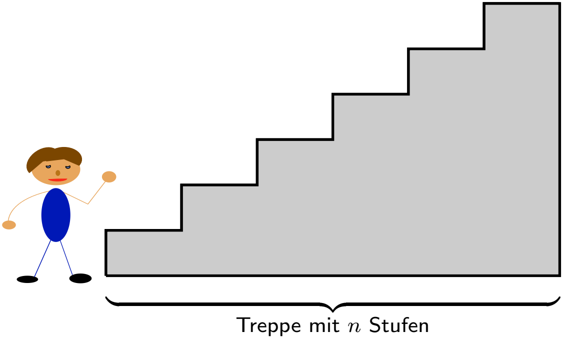
Definition (rekursive Methode)

Eine Methode ist rekursiv, wenn in ihrem Rumpf (Anweisungsteil) die Methode selbst wieder aufgerufen wird.

Beispiel für einen rekursiven Algorithmus

Treppe mit n Stufen hochsteigen:

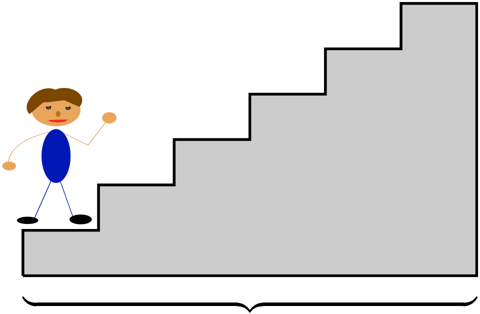
- Wenn $n = 0$, dann fertig, ansonsten:
    - Steige die erste Stufe hoch
    - Treppe mit $n - 1$ Stufen hochsteigen.



Beispiel für einen rekursiven Algorithmus

Treppe mit n Stufen hochsteigen:

- Wenn $n = 0$, dann fertig, ansonsten:
  - Steige die erste Stufe hoch
  - Treppe mit $n - 1$ Stufen hochsteigen.

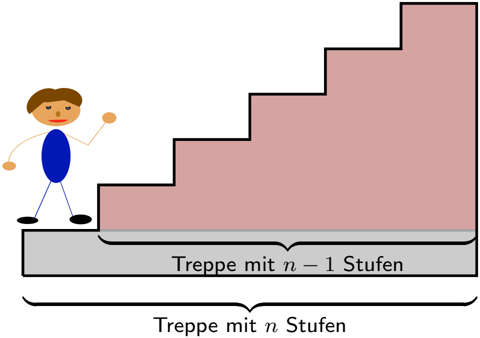


Treppe mit n Stufen

Beispiel für einen rekursiven Algorithmus

Treppe mit n Stufen hochsteigen:

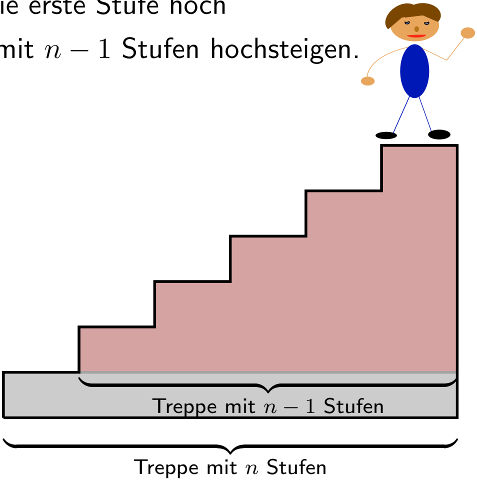
- Wenn $n = 0$, dann fertig, ansonsten:
    - Steige die erste Stufe hoch
    - Treppe mit $n-1$ Stufen hochsteigen.



Beispiel für einen rekursiven Algorithmus

Treppe mit n Stufen hochsteigen:

- Wenn $n = 0$, dann fertig, ansonsten:
  - Steige die erste Stufe hoch
  - Treppe mit $n-1$ Stufen hochsteigen.



Allgemeines Prinzip der Rekursion

- **Basisfall**: Das ist der einfache Fall, für den man das Ergebnis sofort weiß
  (z.B. 0 Stufen)

Allgemeines Prinzip der Rekursion

- **Basisfall**: Das ist der einfache Fall, für den man das Ergebnis sofort weiß
  (z.B. 0 Stufen)
- **Rekursiver Aufruf**:
  - Mache das Problem etwas kleiner, indem ein kleiner Teil gelöst wird.
  - Für das etwas kleinere Restproblem mache den rekursiven Aufruf (die Rekursion „kümmert“ sich um die Lösung)

  (z.B. eine Stufe hochsteigen, den Rest der Treppe rekursiv hochsteigen)

Allgemeines Prinzip der Rekursion

- **Basisfall**: Das ist der einfache Fall, für den man das Ergebnis sofort weiß
  (z.B. 0 Stufen)
- **Rekursiver Aufruf**:
  - Mache das Problem etwas kleiner, indem ein kleiner Teil gelöst wird.
  - Für das etwas kleinere Restproblem mache den rekursiven Aufruf (die Rekursion „kümmert“ sich um die Lösung)

  (z.B. eine Stufe hochsteigen, den Rest der Treppe rekursiv hochsteigen)

Wichtig dabei: Das Problem muss echt kleiner werden und der Basisfall muss irgendwann erreicht werden, anderenfalls **terminiert das Programm nicht**.

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! =$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! = 5 \cdot 4!$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3!$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2!$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1!$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{array}$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0!$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0 \end{aligned}$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! =$
$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1$

Einfache Beispiele

Die Fakultät einer Zahl $n \in \mathbb{N}$ ist definiert durch

- $0! = 1$ und
- $n! = n \cdot (n-1) \cdots 2 \cdot 1$ für alle $n \in \mathbb{N}$ mit $n > 0$

Z.B. ist $5! = 120$, denn $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

Rekursive Definition der Fakultät:

$$
\begin{array}{rcl}
0! & = & 1 \\
n! & = & n \cdot ((n-1)!) \text{ für alle } n \in \mathbb{N} \text{ mit } n > 0
\end{array}
$$

Z.B. $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! =$
$5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = \ldots = 120$

Rekursive Berechnung der Fakultät in Java

```
public static int fac(int n) {
  if (n == 0) {return 1;}     // Basisfall
  else {return n *            // selbst gel"oster Teil
               fac(n-1);      // rekursiver Aufruf
       }
}
```

Auswertung rekursiver Methodenaufrufe

Wir betrachten als Beispiel:

```
int k = fac(3);
```

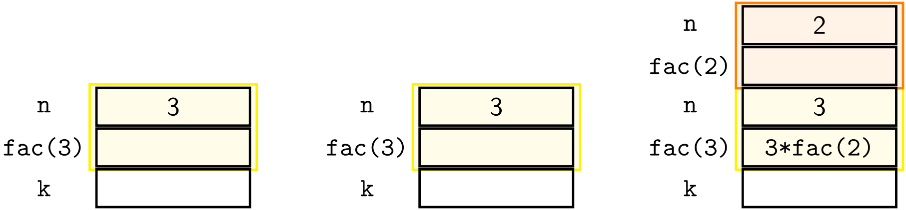
Im ersten Schritt wird auf dem Stack ein Speicherplatz für die Variable k angelegt:

`k`	

Beim Methodenaufruf wird neben Variablen für die aktuellen Parameter **auch eine Variable für das Ergebnis angelegt.**

`n`	3
`fac(3)`	
`k`	

Illustration des Stackaufbaus

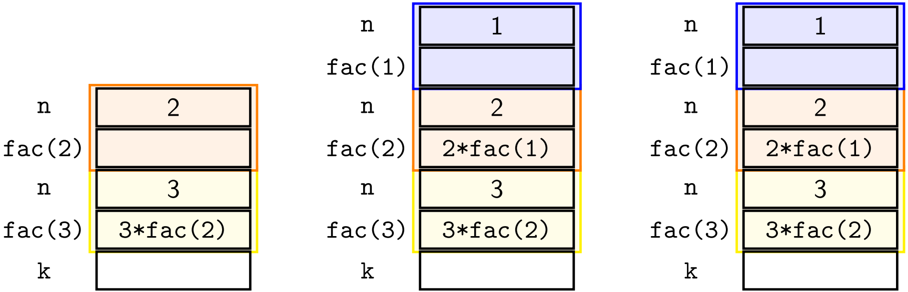



```
k = fac(3)
```

```
if (n == 0) {return 1;}
else {return n*fac(n-1);}
```

```
3*fac(2)
```

Illustration des Stackaufbaus




```
if (n == 0) {...}
else {return n*fac(n-1)}
```

```
2*fac(1)
```

```
if (n == 0) {...}
else {return n*fac(n-1);}
```

Illustration des Stackaufbaus

n	0	0	0
fac(0)			1
n	1	1	1
fac(1)	1*fac(0)	1*fac(0)	1*fac(0)
n	2	2	2
fac(2)	2*fac(1)	2*fac(1)	2*fac(1)
n	3	3	3
fac(3)	3*fac(2)	3*fac(2)	3*fac(2)
k			

1*fac(0)

```
if (n == 0) {return 1;}
else {return n*fac(n-1);}
```

return 1

Illustration des Stackabbaus

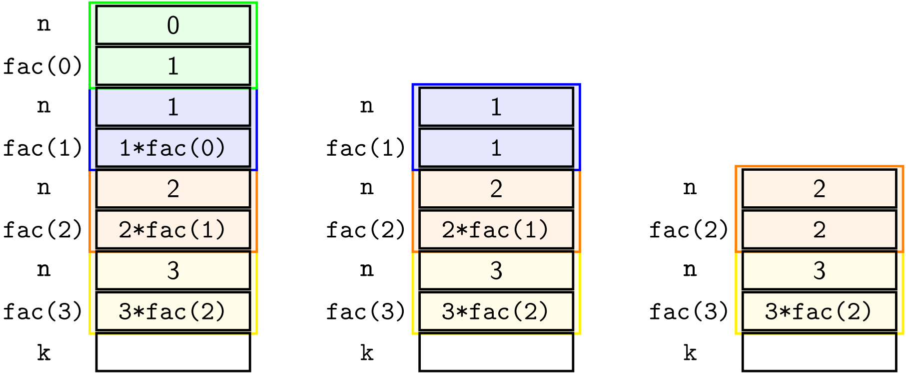
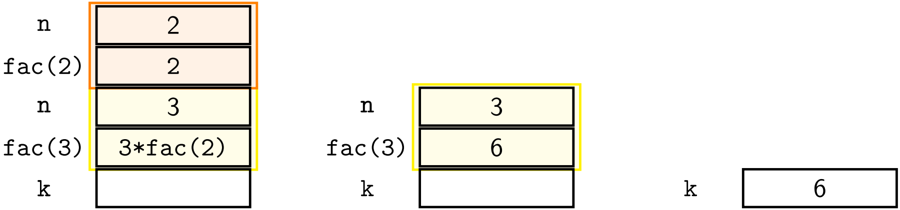


Illustration des Stackabbaus



Terminierung

Der Aufruf einer rekursiven Methode **terminiert**, wenn nach endlich vielen rekursiven Aufrufen ein Abbruchfall erreicht wird.
Beispiele:

```
public static int nonterm1(int n) {
  return n*nonterm1(n-1);
}
```

Aufruf von `nonterm(10)` terminiert nicht, da kein Abbruchfall erreicht wird (in Java erhalten wir einen `StackOverflowError`)

Terminierung

Der Aufruf einer rekursiven Methode **terminiert**, wenn nach endlich vielen rekursiven Aufrufen ein Abbruchfall erreicht wird.
Beispiele:

```
  public static int nonterm1(int n) {
   return n*nonterm1(n-1);
 }
```

Aufruf von `nonterm(10)` terminiert nicht, da kein Abbruchfall erreicht wird (in Java erhalten wir einen `StackOverflowError`)

```
public static int fac(int n) {
  if (n == 0) {return 1;}    // Basisfall
  else {return n *           // selbst gel"oster Teil
               fac(n-1);     // rekursiver Aufruf
       }
}
```

`fac(x)` terminiert für $x \geq 0$, **aber nicht** für $x < 0$!

Beispiele

```
public static int nonterm2(int n) {
  if (n == 0) {return 0;}
  else {return nonterm2(n-2);}
}
```

terminiert für gerade positive Zahlen, aber nicht für ungerade oder negative Zahlen.

Beispiele

```
public static int nonterm2(int n) {
  if (n == 0) {return 0;}
  else {return nonterm2(n-2);}
}
```

terminiert für gerade positive Zahlen, aber nicht für ungerade oder negative Zahlen.

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

Beispiele

```
 public static int nonterm2(int n) {
  if (n == 0) {return 0;}
  else {return nonterm2(n-2);}
}
```

terminiert für gerade positive Zahlen, aber nicht für ungerade oder negative Zahlen.

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

Bis heute ist nicht bewiesen, ob diese Funktion für jede positive natürliche Zahl terminiert (siehe Collatz-Vermutung)

Rekursion und Iteration (1)

Zu jedem rekursiven Algorithmus gibt es einen semantisch äquivalenten iterativen Algorithmus, d.h. einen Algorithmus mit Wiederholungsanweisungen, der dasselbe Problem löst.

Beispiel: Fakultät iterativ:

```
static int facIterativ(int n) {
  int result = 1;
  while (n != 0) {
     result = result * n;
     n--;
  }
  return result;
}
```

Vorteil des iterativen Algorithmus: Der Stack wächst nicht linear, sondern benötigt nur zwei Speicherplätze (für `result` und `n`).

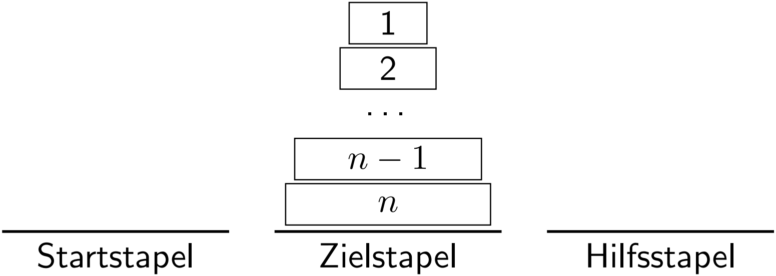
Rekursion und Iteration (2)

- Rekursive Algorithmen sind häufig **eleganter und übersichtlicher** als iterative Lösungen.
- Gute Compiler können aus rekursiven Programmen auch effizienten Code erzeugen; trotzdem sind iterative Programme **meist schneller** als rekursive.
- Für manche Problemstellungen kann es **wesentlich einfacher** sein, einen rekursiven Algorithmus anzugeben als einen iterativen.

Rekursion: Türme von Hanoi

Startsituation

Rekursion: Türme von Hanoi



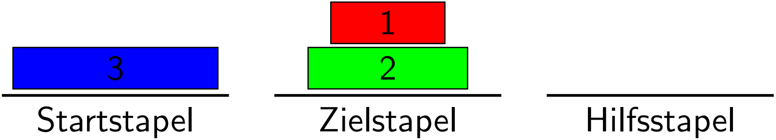
Zielsituation

Beispiel $n = 3$

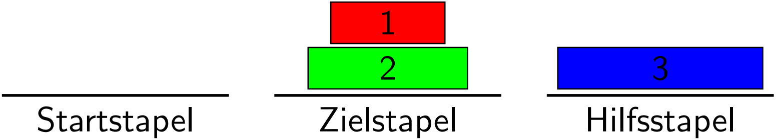
Beispiel $n = 3$

Beispiel $n = 3$

Beispiel $n = 3$



Beispiel $n = 3$



Beispiel $n = 3$

Beispiel $n = 3$

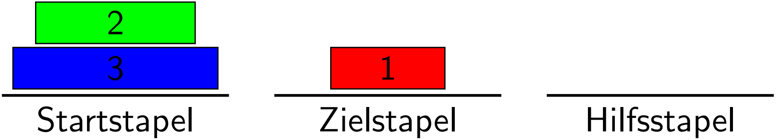
Beispiel $n = 3$

keine korrekte Lösung

Beispiel $n = 3$

zurück zum Anfang

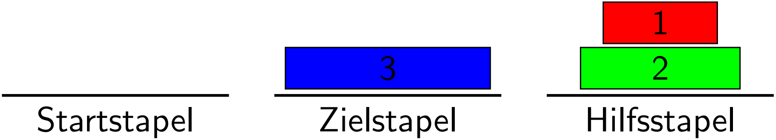
Beispiel $n = 3$



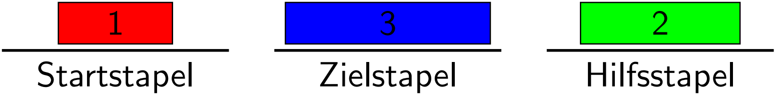
Beispiel $n = 3$

Beispiel $n = 3$

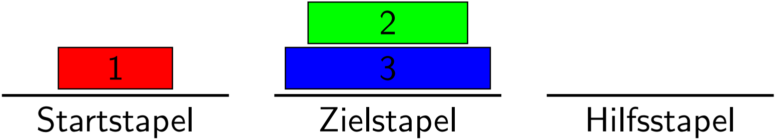
Beispiel $n = 3$



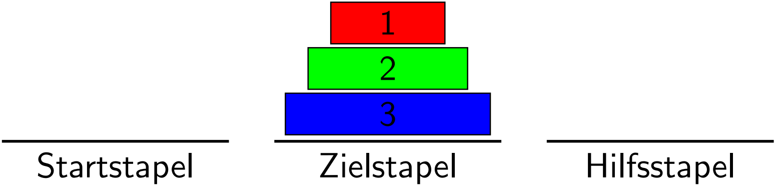
Beispiel $n = 3$



Beispiel $n = 3$



Beispiel $n = 3$



Korrekte Lösung

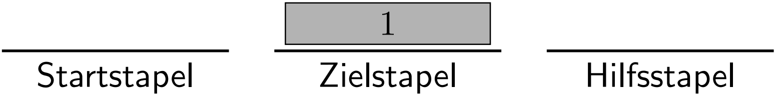
Lösen durch Rekursion: Rekursionanfang

$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang

$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang



$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionsschritt

Lösen durch Rekursion: Rekursionsschritt

1. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Hilfsstapel

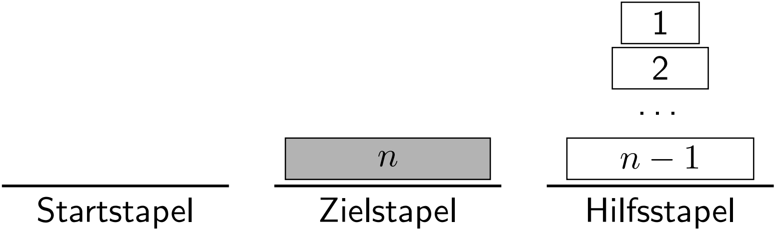
Lösen durch Rekursion: Rekursionsschritt

1. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Hilfsstapel

Lösen durch Rekursion: Rekursionsschritt

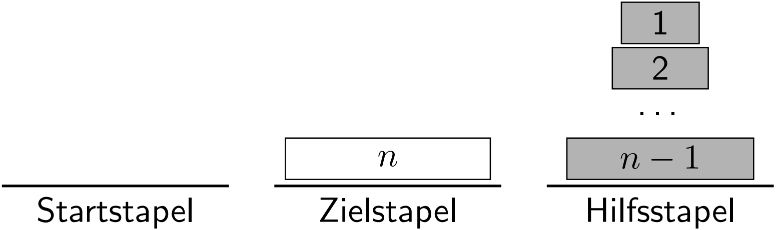
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



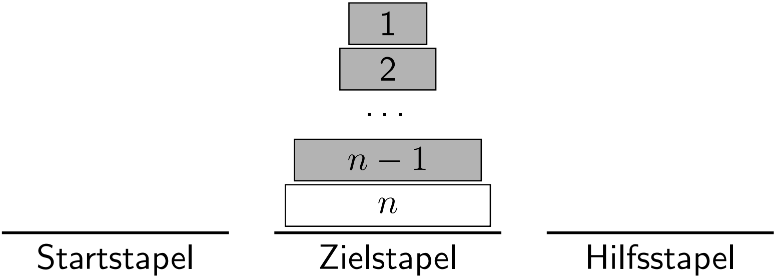
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Zielstapel

Pseudo-Algorithmus

verschiebe(n,start,ziel,hilf)

1. Wenn $n > 1$, dann verschiebe(n-1,start,hilf,ziel)
2. Schiebe Scheibe n von start auf ziel
3. Wenn $n > 1$, dann verschiebe(n-1,hilf,ziel,start)

- Rekursionanfang ist bei $n = 1$: keine rekursiven Aufrufe
- Beachte: Zwei rekursive Aufrufe pro Rekursionsschritt
- Java-Programme: In der Übung

Fibonacci-Zahlen: Rekursiv

- Rekursive Definition der Fibonacci-Zahlen:

$$
\begin{array}{lcl}
\mathit{fib}(0) & = & 1 \\
\mathit{fib}(1) & = & 1 \\
\mathit{fib}(n) & = & \mathit{fib}(n-2) + \mathit{fib}(n-1) \text{ für alle } n \in \mathbb{N} \text{ mit } n \geq 2
\end{array}
$$

Fibonacci-Zahlen: Rekursiv

- Rekursive Definition der Fibonacci-Zahlen:

$$
\begin{array}{rcl}
\mathit{fib}(0) & = & 1 \\
\mathit{fib}(1) & = & 1 \\
\mathit{fib}(n) & = & \mathit{fib}(n-2) + \mathit{fib}(n-1) \text{ für alle } n \in \mathbb{N} \text{ mit } n \geq 2
\end{array}
$$

- Java-Implementierung als rekursive Methode:

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

Beispiel: Kaninchen

- Im Jahr 0 wird 1 Kaninchenpaar geboren.
- Im Jahr 1 hat dieses Paar ein neues Paar geboren.
- In jedem Jahr $n \geq 2$ haben die ein- und zweijährigen Paare jeweils ein neues Paar geboren.

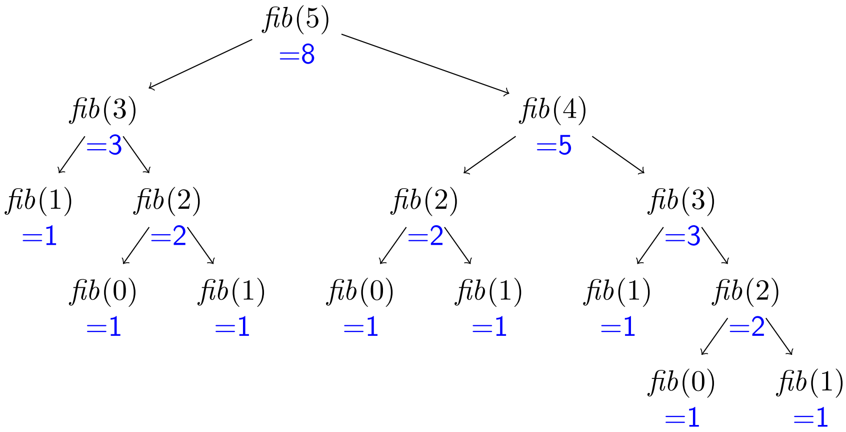
Anzahl der im Jahr n neu geborenen Kaninchenpaare: ?

Beispiel: Kaninchen

- Im Jahr 0 wird 1 Kaninchenpaar geboren.
- Im Jahr 1 hat dieses Paar ein neues Paar geboren.
- In jedem Jahr $n \geq 2$ haben die ein- und zweijährigen Paare jeweils ein neues Paar geboren.

Anzahl der im Jahr n neu geborenen Kaninchenpaare: $fib(n)$

Kaskade rekursiver Aufrufe



Die Zeitkomplexität der rekursiven Fibonacci-Funktion ist exponentiell, d.h. in $\mathcal{O}(2^n)$.
Grund: n-Schritte in die Tiefe, in jedem Schritt wird die Anzahl der rekursiven Aufrufe ungefähr verdoppelt.

Fibonacci-Zahlen iterativ berechnen

Idee: Berechne von $\mathit{fib}(0)$ und $\mathit{fib}(1)$ beginnend aufsteigend:

```
public static int fibIterativ(int n) {
 int fibIMinus2 = 1;   // fib(0) = 1
 int fibIMinus1 = 1;   // fib(1) = 1
 int fibI = 1;
 for (int i=2; i <= n; i++) {
   // fib(i) = fib(i-2) + fib(i-1)
   fibI = fibIMinus1 + fibIMinus2;
   // Verschiebe i um 1:
   fibIMinus2 = fibIMinus1; // fib(i-2) wird fib(i-1)
   fibIMinus1 = fibI;      // fib(i-1) wird fib(i)
 }
 return fibI;
}
```

Die Zeitkomplexität ist linear, d.h. in $\mathcal{O}(n)$ (da die `for`-Schleife $n-1$ mal durchlaufen wird und jeder Schleifendurchlauf konstante Zeit benötigt.
Die Speicherplatzkomplexität ist konstant, d.h. in $\mathcal{O}(1)$, da nur konstant viele Variablen verwendet werden.

Rekursionsformen

- **Lineare Rekursion**: In jedem Zweig der Fallunterscheidung kommt höchstens ein rekursiver Aufruf vor, z.B. Fakultätsfunktion `fac`.

Rekursionsformen

- **Lineare Rekursion**: In jedem Zweig der Fallunterscheidung kommt höchstens ein rekursiver Aufruf vor, z.B. Fakultätsfunktion `fac`.
- **Baumrekursion** (Kaskadenartige Rekursion): Meherere rekursive Aufrufe stehen nebeneinandern und sind durch Operationen verknüpft, z.B. Fibonacci-Zahlen `fib`

Rekursionsformen

- **Lineare Rekursion**: In jedem Zweig der Fallunterscheidung kommt höchstens ein rekursiver Aufruf vor, z.B. Fakultätsfunktion `fac`.
- **Baumrekursion** (Kaskadenartige Rekursion): Meherere rekursive Aufrufe stehen nebeneinandern und sind durch Operationen verknüpft, z.B. Fibonacci-Zahlen `fib`
- **Verschachtelte Rekursion**: Rekursive Aufrufe kommen in den Parametern von rekursiven Aufrufen vor, z.B. Ackermann-Funktion.

Die Ackermann-Funktion

```
public static int ack(int n, int m) {
 if (n == 0) {return m+1;}
 else if (m == 0)  {return ack (n-1,1);}
      else {return ack(n-1, ack(n,m-1));}
}
```

verschachtelte Rekursion

- Die Ackermann-Funktion wächst extrem schnell
- Sie ist das klassische Beispiel für eine berechenbare, terminierende Funktion, die nicht primitiv-rekursiv ist
(erfunden 1926 von Ackermann)
- Beispiele: $$\begin{array}{lcl} ack(4,0) & = & 13 \\ ack(4,1) & = & 65533 \\ ack(4,2) & = & 2^{65536}-3 \\ ack(4,4) & > & \text{Anzahl der Atome im Universum} \end{array}$$

Quicksort

Quicksort ist ein schneller (vergleichsbasierter) Sortieralgorithmus (entwickelt von Tony Hoare, 1962).

Ideen:

- Falls das zu sortierende Array mindestens 2 Elemente hat:
  1. Wähle irgendein Element aus dem Array als **Pivot** („Dreh- und Angelpunkt“), z.B. das erste Element.
  2. **Partitioniere** das Array in einen linken und einen rechten Teil, so dass
     - alle Elemente im linken Teil kleiner-gleich als das Pivot sind,
     - alle Elemente im rechten Teil größer als das Pivot sind.
  3. Wende Quicksort (rekursiv) auf beide Teilarrays an.
- Der Quicksort folgt einen ähnlichen Lösungsansatz wie die binäre Suche. Diesen Ansatz nennt man „Divide-and-Conquer“ („Teile und beherrsche“)

Quicksort: Beispiel

65	43	92	26	75	13

Quicksort: Beispiel

65	43	92	26	75	13

Wähle Pivot, z.B. 65

Quicksort: Beispiel

65	43	92	26	75	13

Wähle Pivot, z.B. 65

65	43	92	26	75	13

Quicksort: Beispiel

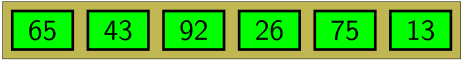
65	43	92	26	75	13

Wähle Pivot, z.B. 65

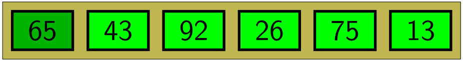
65	43	92	26	75	13

Partitioniere anhand des Pivots

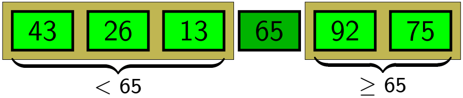
Quicksort: Beispiel



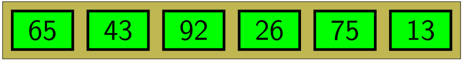
Wähle Pivot, z.B. 65



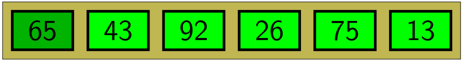
Partitioniere anhand des Pivots



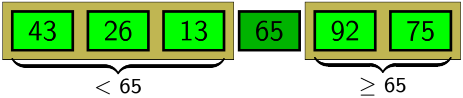
Quicksort: Beispiel



Wähle Pivot, z.B. 65



Partitioniere anhand des Pivots



Sortiere beide Teilarrays rekursiv mit Quicksort

Quicksort: Beispiel

65	43	92	26	75	13

Wähle Pivot, z.B. 65

65	43	92	26	75	13

Partitioniere anhand des Pivots

$$\underbrace{\boxed{43}\ \boxed{26}\ \boxed{13}}_{< 65}\ \boxed{65}\ \underbrace{\boxed{92}\ \boxed{75}}_{\geq 65}$$

Sortiere beide Teilarrays rekursiv mit Quicksort

13	26	43	65	75	92

Implementierung in Java

```
public static void quicksort(int[] arr) {
  // sortiere alle Elemente des Arrays:
  qsort(arr, 0, arr.length-1);
}

public static void qsort(int[] arr, int left, int right) {
 // sortiere den Tellbereich von left bis right in arr
 if (left < right) { // mehr als ein Element zu sortieren
   // w"ahle erstes Element als Pivot
   int pivot = arr[left];
   // partitioniere anhand des Pivot-Elements
   int pivotIndex = partition(arr,left,right,pivot);
   // sortiere linken und rechten Teil rekursiv
   qsort(arr,left,pivotIndex-1);
   qsort(arr,pivotIndex+1,right);
 }
}
```

Es fehlt noch die Methode: partition

Einfache Variante von partition

Idee:

- `partition(int[] arr,left,right,pivot)` partitioniert das Array `arr` im Bereich `left` bis `right` anhand des Pivots `pivot` und liefert den Index des Pivotelements.
- Benutze Kopie `copy` des Teilbereichs
- Durchlaufe `copy` dreimal um die Werte in `arr[left..right]` zu überschreiben:
  1. Schreibe die Werte kleiner als das Pivot
  2. Schreibe die Werte gleich zum Pivot
  3. Schreibe die Werte größer als das Pivot
- Dabei muss der Rückgabewert für den Index auf das Pivotelement entsprechend verwaltet werden.

Einfache Variante von partition

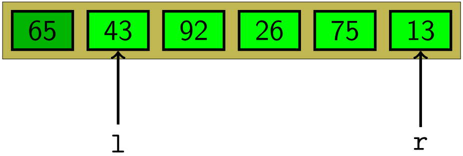
```java
public static int partition(int[] arr, int left, int right, int pivot) {
  int[] copy = new int[right-left+1];
   // erstelle Kopie des zu sortierenden Teils
  for (int i=0; i < copy.length; i++) {copy[i] = arr[left+i];}
  int pivotIndex = left-1;
  int writePos = left;
  // Schreibe linken Teil
  for (int i=0; i < copy.length; i++) {if (copy[i] < pivot) {
                                         arr[writePos] = copy[i];
                                         pivotIndex++; writePos++;}
  }
  // Schreibe alle Elemente gleich zum Pivot
  for (int i=0; i < copy.length; i++) {if (copy[i] == pivot) {
                                         arr[writePos] = copy[i];
                                         pivotIndex++; writePos++;}
  }
  // Schreibe rechten Teil
  for (int i=0; i < copy.length; i++) {if (copy[i] > pivot) {
                                         arr[writePos] = copy[i];
                                         writePos++;}
  }
  return pivotIndex;
 }
```

Speicherplatzkomplexität

Da `partition` eine Kopie des Arrays im Speicher hält, benötigt dieser Quicksort für ein Array der Länge n, $\mathcal{O}(n)$ (zusätzlichen) Speicherplatz.

Wir betrachten daher eine optimierte Variante.

Partitonieren ohne zusätzlichen Platzbedarf (1)

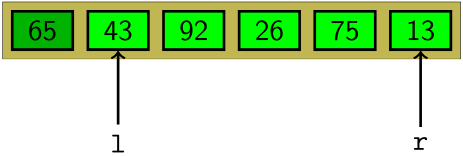


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

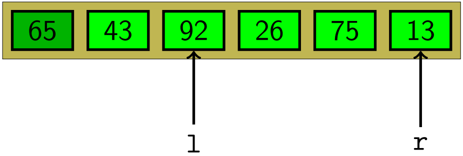


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

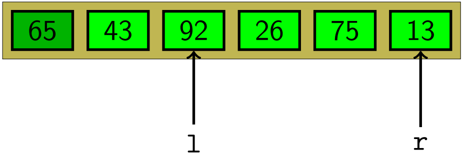


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

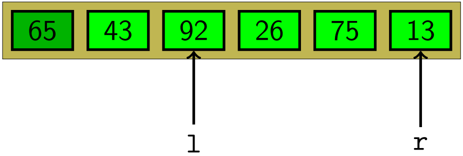


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

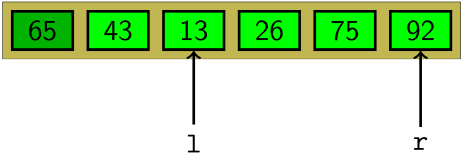


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

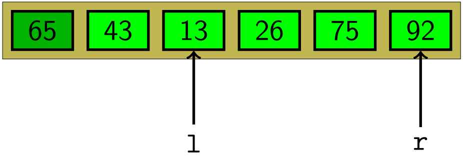


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

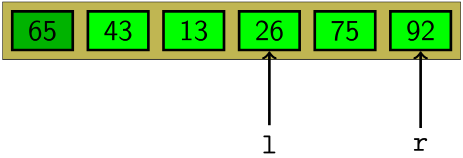


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

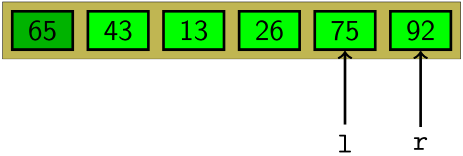


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

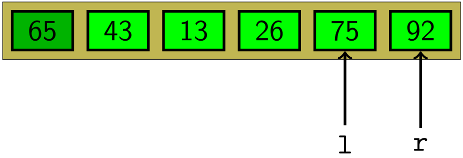


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

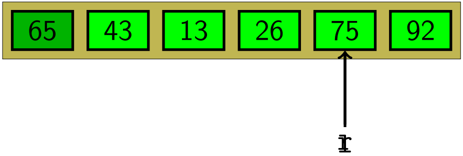


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

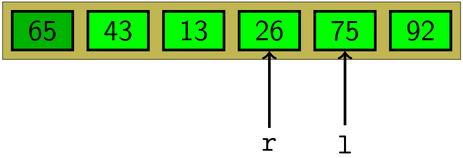


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

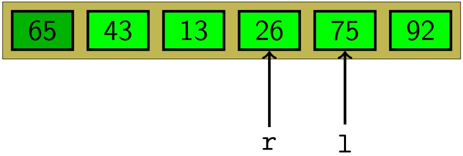


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

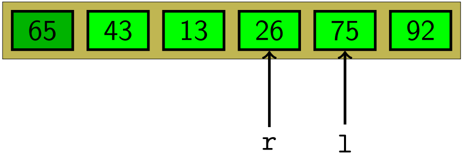


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)

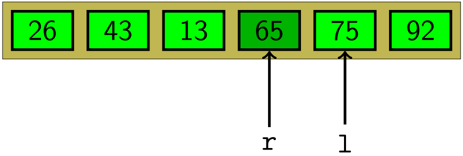


Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitonieren ohne zusätzlichen Platzbedarf (1)



Idee:

- wenn `l` und `r` sich noch nicht gekreuzt haben:
  1. Schiebe `l` solange nach rechts, bis eine Zahl größer als das Pivot gefunden wird.
  2. Schiebe `r` solange nach links, bis eine Zahl kleiner als das Pivot gefunden wird.
  3. Wenn sich dabei `l` und `r` nicht gekreuzt haben, vertausche die Einträge und mache weiter mit 1.

  Vertausche das Pivot mir `r`

Partitionieren ohne zusätzlichen Platzbedarf (2)

```
public static void swap(int[] arr, int l, int r) {
  int tmp = arr[l];
  arr[l] = arr[r];
  arr[r] = tmp;
}
public static int partition(int[] arr, int left, int right, int pivot) {
  // in-place partition, geht davon aus, dass pivot sich an arr[left] befindet
  int l = left+1; // fange links neben dem Pivot an
  int r = right; // fange rechts ganz rechts an
  boolean proceed = true; // vertausche weiter?
  while (proceed) {
   while (l <= right && arr[l] < pivot) {l++;} // schiebe l nach links bis ein
        zu gro"sses Element gefunden
   while (r >= left && arr[r] > pivot) {r--;} // schiebe r nach rechts bis ein
        zu kleines Element gefunden
   if (l < r) { swap(arr,l,r); // vertausche arr[l] und arr[r]
               l++; r--;     // schiebe l nach links und r nach rechts
    }
    else {proceed = false;} //stoppe
  }
  // setze Pivot an die richtige Position
  swap(arr,left,r); // r ist das erste zu kleine Element von rechts
  return r;
}
```

Platzbedarf der optimierten Variante

- Es werden neben der Eingabe nur konstant viele lokale Variablen verwendet.
- Aber: Die rekursiven Aufrufe werden auf dem Stack abgelegt.
- Daher: Platzbedarf ist abhängig von der maximalen Rekursionstiefe!

Komplexität von Quicksort (1)

Sei n die Länge des Eingabearrays.

- Der Zeitbedarf zum Partitionieren eines Teilarrays mit m Einträgen ist in allen Fällen in $\mathcal{O}(m)$, da `l` und `r` stets um mindestens 1 erhöht bzw. um 1 erniedrigt werden, und insgesamt weniger als $\texttt{r}-\texttt{l} < \texttt{m}$ solche Veränderungen möglich sind.
- Alle Partitionierungen in gleicher Rekursionstiefe (d.h. nach k-maligem Aufruf von `qsort`) benötigen in der Summe daher Zeit in $\mathcal{O}(n)$.
- Zur Laufzeitabschätzung müssen wir daher wissen, **wie oft** partitioniert werden muss.

Komplexität von Quicksort (2)

- Im besten Fall halbiert das Partitionieren jedesmal, d.h. die Elemente werden gleichmäßig in den linken und rechten Teil verteilt. Dann müssen wir nicht öfter als $(\log_2 n) + 1$ mal partitionieren. Daher ist die **best-case** Laufzeitkomplexität von Quicksort in $O(n\ \log\ n)$. Entsprechend ist die Platzkomplexität im **best-case** $\mathcal{O}(\log\ n)$ für die rekursiven Aufrufe auf dem Stack.

Komplexität von Quicksort (2)

- Im besten Fall halbiert das Partitionieren jedesmal, d.h. die Elemente werden gleichmäßig in den linken und rechten Teil verteilt. Dann müssen wir nicht öfter als $(\log_2 n) + 1$ mal partitionieren. Daher ist die **best-case** Laufzeitkomplexität von Quicksort in $O(n \ \log \ n)$. Entsprechend ist die Platzkomplexität im **best-case** $\mathcal{O}(\log \ n)$ für die rekursiven Aufrufe auf dem Stack.
- Im schlechtesten Fall ist eine Partition stets leer, und die andere enthält alle Elementen außer dem Pivot. Dann müssen wir $n-1$-mal partitionieren. Daher ist die **worst-case** Laufzeitkomplexität von Quicksort in $\mathcal{O}(n^2)$ und die **worst-case** Platzkomplexität in $\mathcal{O}(n)$.

Komplexität von Quicksort (2)

- Im besten Fall halbiert das Partitionieren jedesmal, d.h. die Elemente werden gleichmäßig in den linken und rechten Teil verteilt. Dann müssen wir nicht öfter als $(\log_2 n) + 1$ mal partitionieren. Daher ist die **best-case** Laufzeitkomplexität von Quicksort in $O(n \ \log \ n)$. Entsprechend ist die Platzkomplexität im **best-case** $\mathcal{O}(\log \ n)$ für die rekursiven Aufrufe auf dem Stack.
- Im schlechtesten Fall ist eine Partition stets leer, und die andere enthält alle Elementen außer dem Pivot. Dann müssen wir $n - 1$-mal partitionieren. Daher ist die **worst-case** Laufzeitkomplexität von Quicksort in $\mathcal{O}(n^2)$ und die **worst-case** Platzkomplexität in $\mathcal{O}(n)$.
- Man kann zeigen, dass im Durchschnitt immer noch $\mathcal{O}(\log \ n)$ rekursive Aufrufe ausreichen, daher ist die **average-case** Laufzeitkomplexität von Quicksort in $\mathcal{O}(n \ \log \ n)$ und die Platzkomplexität im Mittel in $\mathcal{O}(\log n)$.

Verbesserungen des Quicksort-Algorithmus

Praktische Verbesserungen

- Wenn Arrays kurz werden (z.B. 10 Elemente), verwende einfachen Sortieralgorithmus (z.B. Selection Sort)

Verbesserungen des Quicksort-Algorithmus

Praktische Verbesserungen

- Wenn Arrays kurz werden (z.B. 10 Elemente), verwende einfachen Sortieralgorithmus (z.B. Selection Sort)
- Bestimme Pivotelement durch Ziehen von 3 Elementen:
  - erstes Element
  - mittleres Element
  - letztes Element

  Wähle Pivot als Median der 3 Elemente.

Verbesserungen des Quicksort-Algorithmus

Praktische Verbesserungen

- Wenn Arrays kurz werden (z.B. 10 Elemente), verwende einfachen Sortieralgorithmus (z.B. Selection Sort)
- Bestimme Pivotelement durch Ziehen von 3 Elementen:
  - erstes Element
  - mittleres Element
  - letztes Element

  Wähle Pivot als Median der 3 Elemente.
- Teile das Array in 3 Teile: $<$ als Pivot, $=$ Pivot, $>$ Pivot.
  Der mittlere Teil wird nicht mehr im rekursiven Aufruf berücksichtigt.
  Starke Beschleunigung bei vielen gleichen Elementen.

Bemerkungen: Sortieren

- Man kann nachweisen, dass jeder **vergleichsbasierte** Sortieralgorithmus im worst-case log-linear ist.
- Es gibt Sortierverfahren, die auch im worst-case dies erreichen (z.B. Merge-Sort)
- Für nicht-vergleichsbasierte Sortierverfahren (z.B. von Ganzzahlen fester Länge) sind auch lineare Verfahren bekannt.

Zusammenfassung

- Prinzip der Rekursion: Basisfall, Rekursiver Aufruf
- Auf Terminierung achten!
- Rekursionsformen: lineare Rekursion, Baumrekursion, Verschachtelte Rekursion
- Iterativ vs. Rekursion
- Beispiele (Türme von Hanoi, fac, fib, ackermann)
- Quicksort als rekursives und schnelles Sortierverfahren