Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL05 – Rekursion

Inhalt

- Einführung
  - Fünf Beispiele
- Rekursionsbasis und -vorschrift
- Rekursiv programmieren
- Analyse rekursiver Programme
- Formen der Rekursion
  - Direkte vs. indirekte Rekursion
  - Divide and Conquer (Teile und Herrsche)
  - Backtracking
- Iterative Lösung oder Rekursion?
- Klassisches Beispiel: Türme von Hanoi

EINFÜHRUNG

Einführung

- **Rekursion** ist ein abstraktes Konzept, welches an vielen Stellen in der Informatik auftritt. Rekursion bedeutet „Rückführung auf sich selbst“.

- Viele Dinge sind rekursiv aufgebaut:
  - z.B. Verzeichnisbäume, Zahlenfolgen (Fakultät und viele andere), …

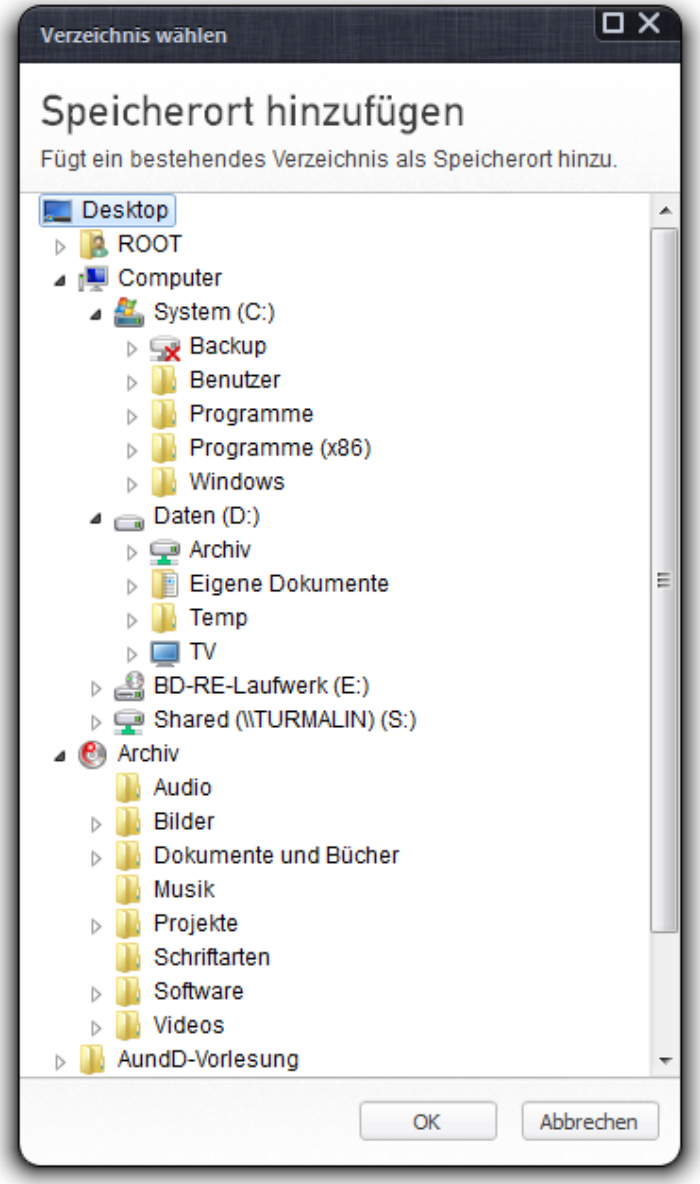
- Rekursive Algorithmen eignen sich besonders gut, um rekursiv strukturierte Probleme zu lösen:
  - z.B. Suche nach bestimmten Dateien im Verzeichnisbaum, Ermitteln einer bestimmten Zahl in einer rekursiven Folge, …

Nächste Woche lernen wir mit „Bäumen“ eine rekursiv aufgebaute Datenstruktur kennen, deren Operationen rekursiv programmiert werden.

Einführung

Beispiel 1: Verzeichnisbaum

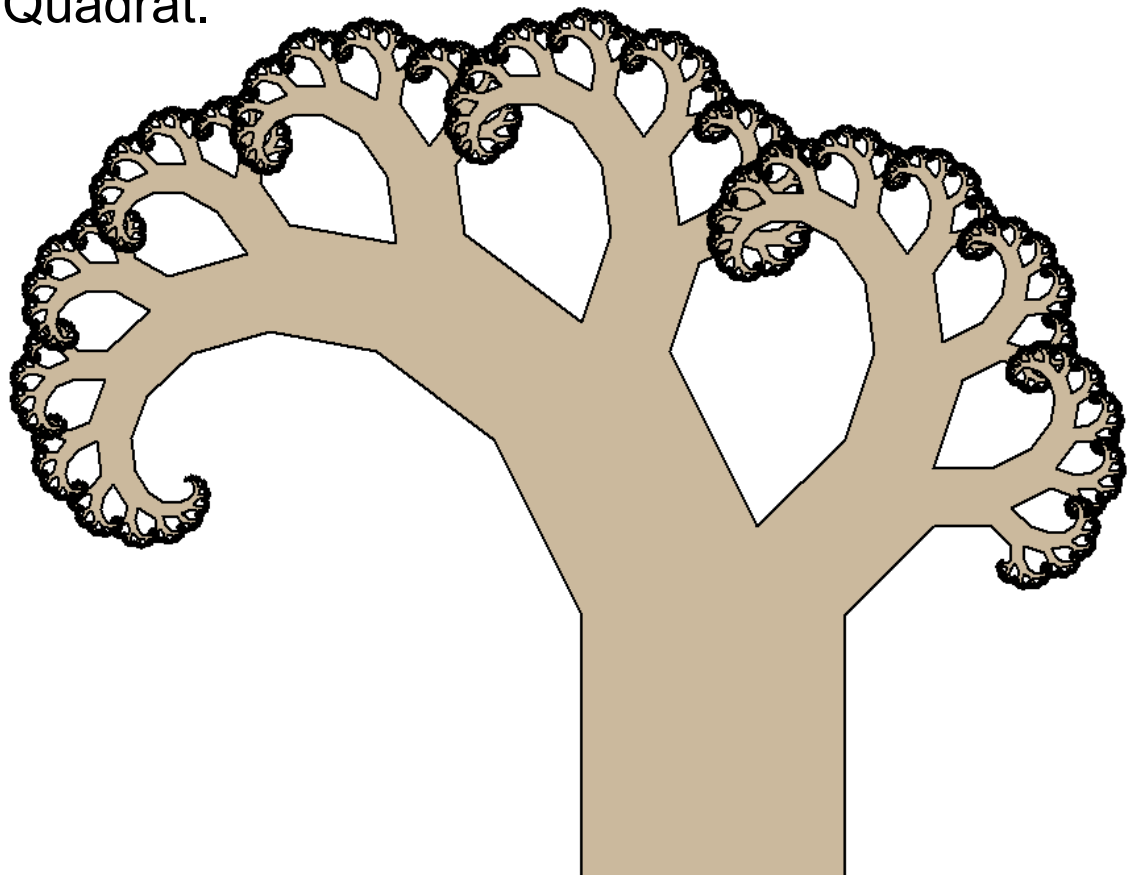
- Verzeichnisbäume sind rekursiv aufgebaut:
  - Es gibt ein Hauptverzeichnis (`Desktop` bei Windows, `/` bei Unix incl. MacOS)
  - Jedes Verzeichnis enthält Dateien und weitere Verzeichnisse, die gleich strukturiert sind (also wiederum Dateien und weitere Verzeichnisse enthalten).



Einführung

Beispiel 2: Pythagoras-Baum

- Rekursive Zeichenvorschrift:
  1. Zeichne ein einzelnes Quadrat.
  2. Errichte über dem Quadrat ein rechtwinkliges Dreieck.
  3. Ergänze die Schenkel des Dreiecks zu Quadraten.
  4. Fahre bei 2 fort für jedes Quadrat.



Einführung

Beispiel 3: Lexikon als Hypertext-Dokument

Hauptspeicher
Bezeichnung für den *Speicher*, der die gerade auszuführenden Programme und die benötigten Daten enthält. Der Hauptspeicher ist eine Komponente der *Zentraleinheit*.

Speicher
Funktionseinheit zum Aufbewahren von Daten.

Zentraleinheit
Unter Zentraleinheit versteht man den Rechnerkern, der *Befehle* interpretieren und ausführen kann.

Befehl
Ein Befehl ist die kleinste bezüglich einer *Programmier-sprache* nicht weiter zerlegbare Einheit.

Programmiersprache
Sprache zur Formulierung von Rechenvorschriften, d.h. *Algorithmen* und *Datenstrukturen*.

Einführung

Beispiel 4a: Rekursiv definierte Zahlenfolgen

- Gegeben: rekursiv definierte Zahlenfolge

  $a_0 = 2$

  $a_n = 2 * a_{n-1} + 1$ für $n>0$
- Gesucht: Wert von a_3
- Lösung:

$$a_3 = 2 * a_2 + 1$$

$$= 2 * (2 * a_1 + 1) + 1$$

$$= 2 * (2 * (2 * a_0 + 1) + 1) + 1$$

$$= 2 * (2 * (2 * 2 + 1) + 1) + 1$$

$$= 23$$

Einführung

Beispiel 4b: Fakultätsfunktion

- $n! = n * (n-1) * (n-2) * \ldots * 1$
- Rekursive Definition:
  $0! = 1$
  $n! = n * (n-1)!$ für alle $n>0$

Rekursionsbasis und -vorschrift

Rekursionsbasis und Rekursionsvorschrift

- Prinzip der Rekursion: ein Problem wird auf ein kleineres zurückgeführt, welches wiederum nach demselben Verfahren bearbeitet wird
- Eine rekursiv definiertes Problem wird durch eine Rekursionsrelation beschrieben. Sie setzt sich zusammen aus der **Rekursionsbasis** (einfaches Grundproblem, das sofort gelöst werden kann) und der **Rekursionsvorschrift**.
- Es muss gewährleistet sein, dass nach endlich vielen Anwendungen der Rekursionsvorschrift die Rekursionsbasis erreicht wird.

Rekursionsbasis und Rekursionsvorschrift

- Beispiel Verzeichnisbaum:
  - Rekursionsbasis: Verzeichnis ohne weitere Unterverzeichnisse
  - Rekursionsvorschrift: Verzeichnisse enthalten Unterverzeichnisse
- Beispiel Pythagoras-Baum:
  - Rekursionsbasis: Quadrat zeichnen
  - Rekursionsvorschrift: Dreiecks-Schenkel für neue Quadrate ergänzen
- Beispiel Lexikon:
  - Rekursionsbasis: Artikel ohne weiterführende Links
  - Rekursionsvorschrift: Artikel enthalten weiterführende Links
- Beispiel Zahlenfolge:
  - Rekursionsbasis: $a_0 = 2$
  - Rekursionsvorschrift: $a_n = 2 * a_{n-1} + 1$ für $n>0$
- Beispiel Fakultät:
  - Rekursionsbasis: $0! = 1$
  - Rekursionsvorschrift: $n! = n * (n-1)!$ für $n>0$

Rekursiv programmieren

Rekursiv programmieren

Beispiel: Fakultät

- Fakultät: $n! = 1 * 2 * 3 * ... * (n-1) * n$
  - Das entspricht der Folge
    $a_0 = 1, a_1 = 1, a_2 = 2, a_3 = 6, a_4 = 24, a_5 = 120, \ldots$
  - Rekursiv ausgedrückt:
    $a_0 = 0! = 1$ // Rekursionsbasis
    $a_n = n! = n * (n-1)! = n * a_{n-1}$ // Rekursionsvorschrift

- In Java programmiert:

```
static long fakRek(int n)
{
  // Die Fakultät ist nur für n>=0 definiert
  assert(n>=0);

  return (n<=1) ? 1 : n*fakRek(n-1);
}
```

Die Methode ruft sich selbst auf.

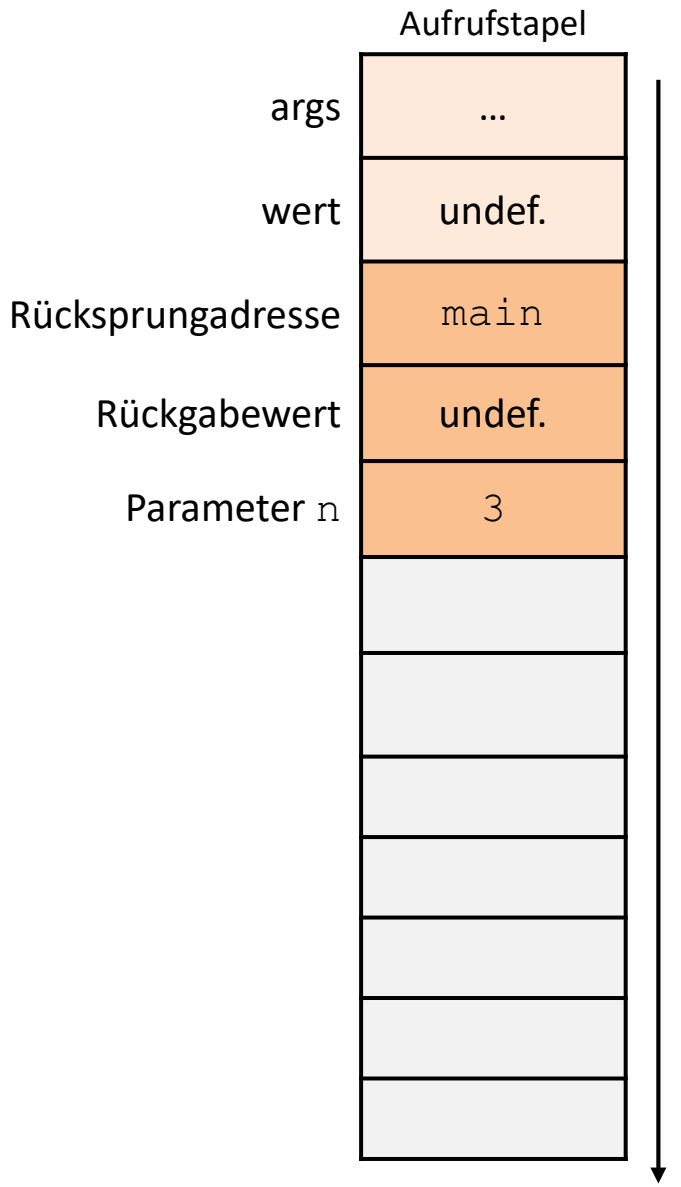
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



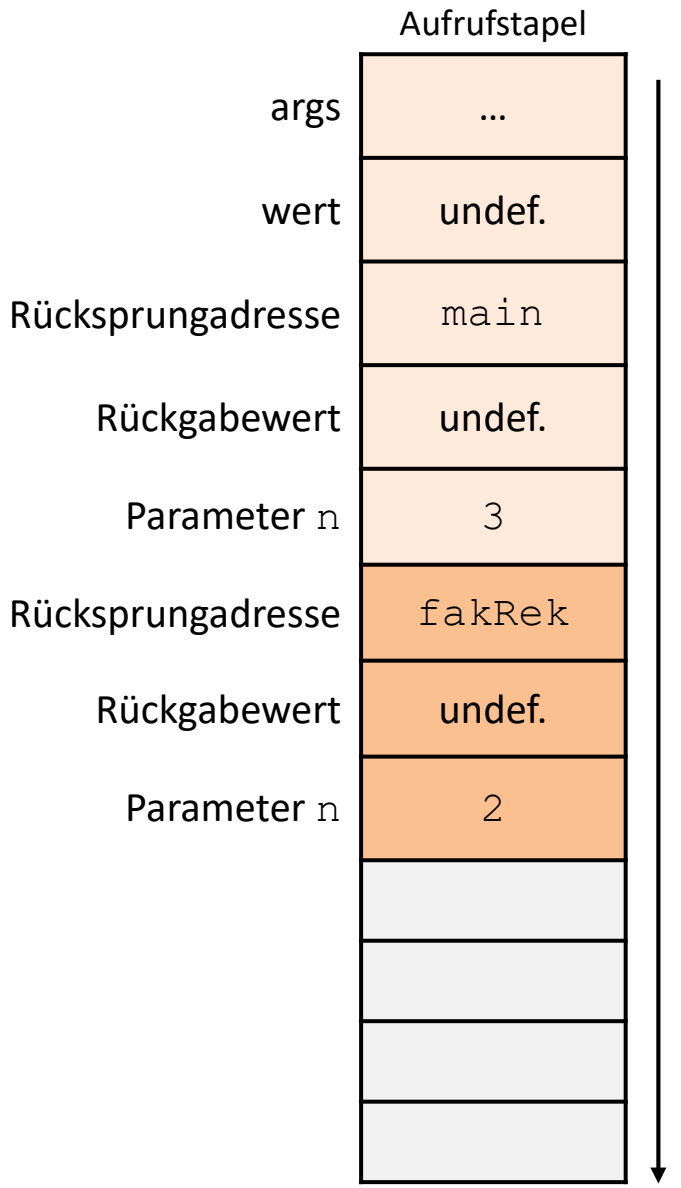
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



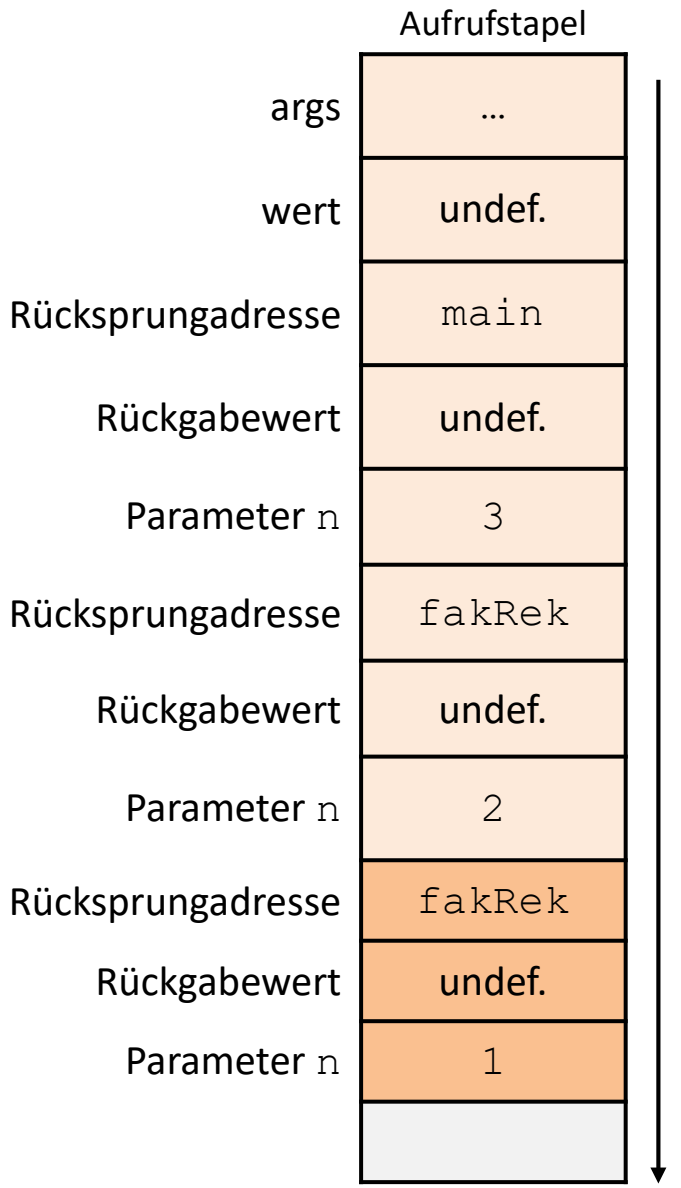
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



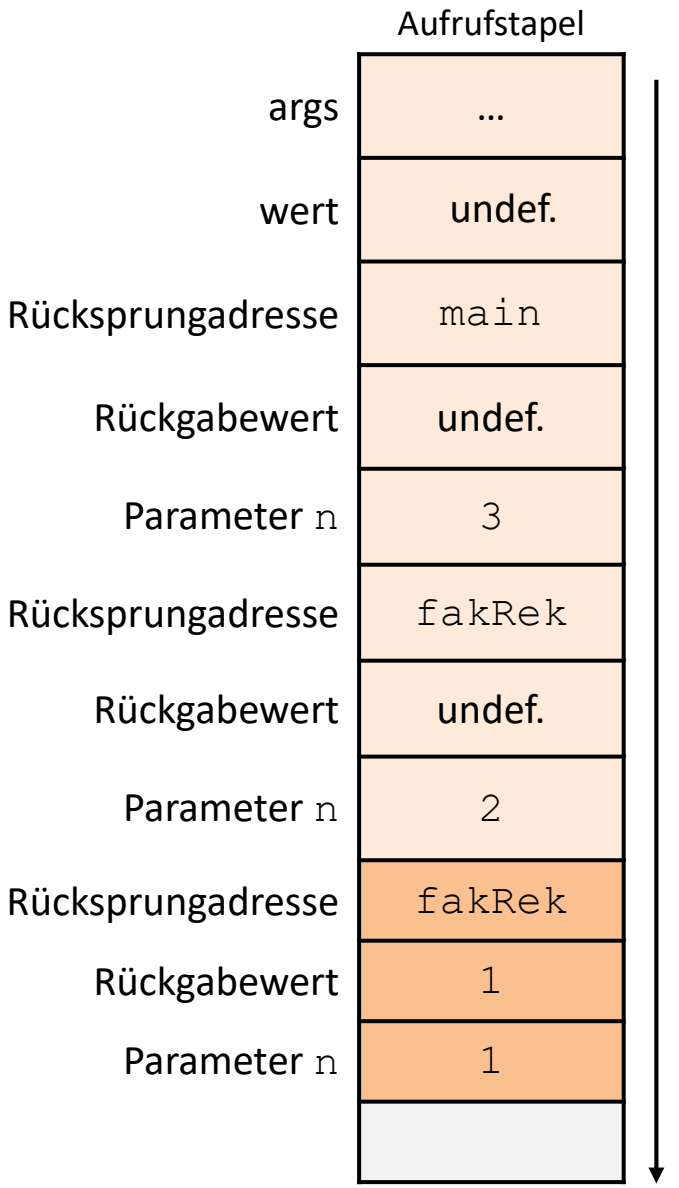
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



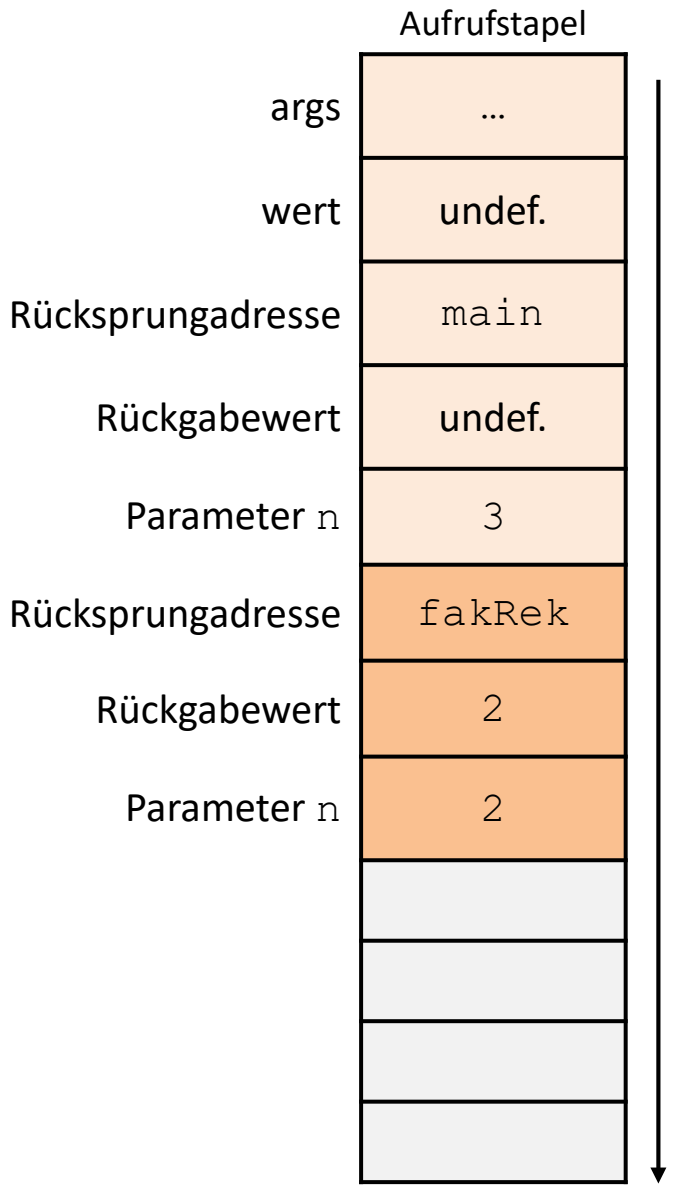
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



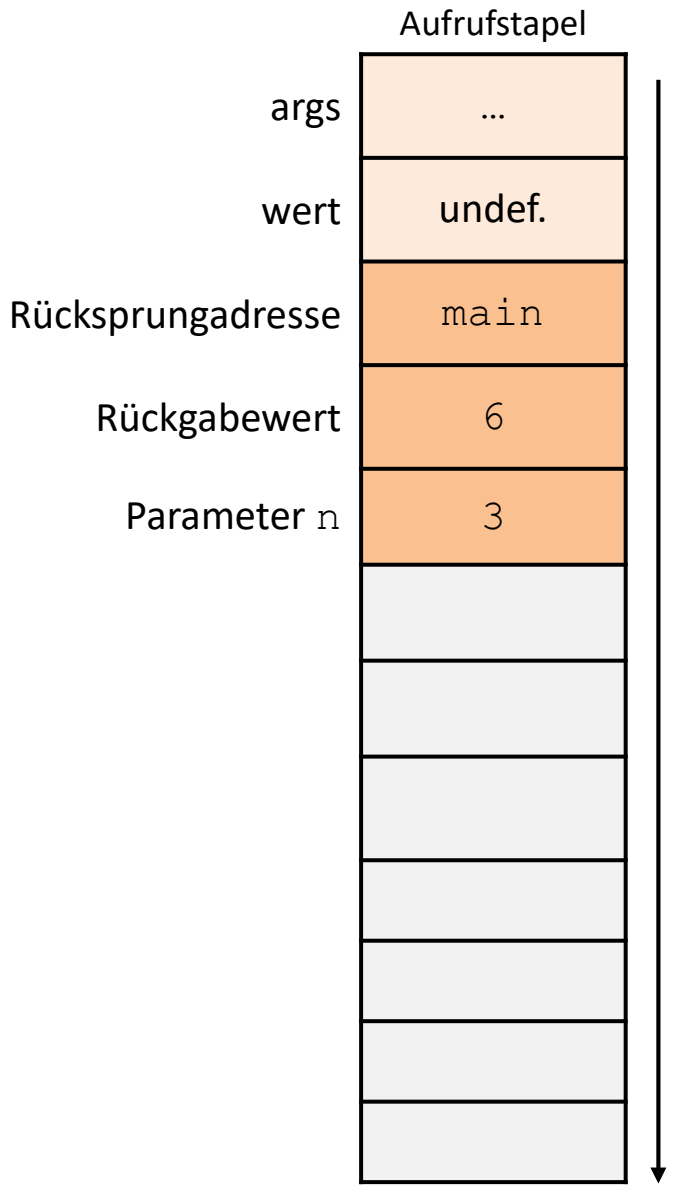
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



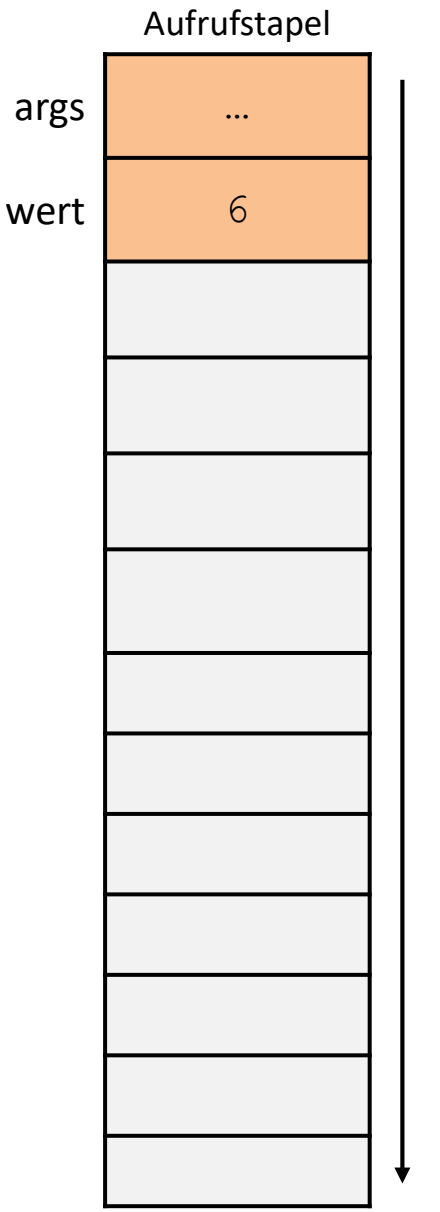
Rekursiv programmieren

Verhalten im Speicher

Situation: im Hauptprogramm wurde `fakRek(3)` aufgerufen.

```
public static void main(
    String[] args)
{
  long wert = fakRek(3);
  …
}

static long fakRek(int n)
{
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```



Rekursiv programmieren

Verhalten im Speicher

- Bei jedem Methodenaufruf wird auf dem Aufrufstapel (engl. call stack) Speicherplatz für die lokalen Variablen incl. Parameter sowie für die Verwaltung der Aufrufe belegt.
- Jede Rekursionsstufe braucht Speicherplatz auf dem Stapel. Dieser ist umso größer, je mehr lokale Variablen und Parameter die Methode besitzt.
  Gefahr: Stapelüberlauf (engl. stack overflow)!
- Deshalb bei der Programmierung:
  - möglichst wenige lokale Variablen und Parameter benutzen
  - für möglichst geringe Anzahl der gleichzeitig aktiven rekursiven Aufrufe (**Rekursionstiefe**) sorgen

Rekursiv programmieren

Dynamische Endlichkeit

- Aufrufe rekursiver Methoden müssen dynamisch endlich sein, d.h. irgendwann entstehen keine weiteren **Rekursionsstufen** (**Inkarnationen** der Methode) mehr.
- Zwei Bedingungen sorgen für dynamische Endlichkeit:
  - Jeder rekursive Aufruf muss die Berechnung bzw. das Problem vereinfachen bzw. verkleinern:
    `n*fakRek(n-1)`
  - Der einfachste Fall muss ohne weiteren rekursiven Aufruf behandelt werden. Es muss also immer eine Abbruchbedingung für die Rekursion geben: `(n<=1) ? 1 :` …

Analyse rekursiver Programme

Analyse rekursiver Programme

Was macht ein rekursives Programm?

- Man kann die Funktionsweise eines rekursiven Programms mittels eines Stapels nachvollziehen.
- Ist die rekursive Methode ohne Seiteneffekte, so kann man jeden Aufruf der Methode für sich betrachten.

```
static long exp2(int n)
{
  assert(n>=0);
  return (n==0) ? 1 : exp2(n-1) + exp2(n-1);
}
```

- Beispiel: Aufruf mit `n=3`
  - `exp2(3)` ist `exp2(2) + exp2(2)`
  - `exp2(2)` ist `exp2(1) + exp2(1)`
  - `exp2(1)` ist `exp2(0) + exp2(0)`
  - `exp2(0)` ist die Rekursionsbasis und hat den Rückgabewert 1

Analyse rekursiver Programme

Komplexität eines rekursiven Programms

```
static long exp2(int n)
{
  assert(n>=0);
  return (n==0) ? 1 : exp2(n-1) + exp2(n-1);
}
```

- Möglichkeit 1: Aufstellen einer rekursiven Gleichung
  - Für die Laufzeit $t(n)$ von `exp2` gilt: $t(n) \leq 2 * t(n\text{-}1) + c$ mit $c > 0$
  - Somit gilt insgesamt (Beweis z.B. per Induktion): $t(n) \leq c * (2^{n+1}\text{-}1)$

- Wodurch wird die exponentielle Laufzeit $O(2^n)$ verursacht?

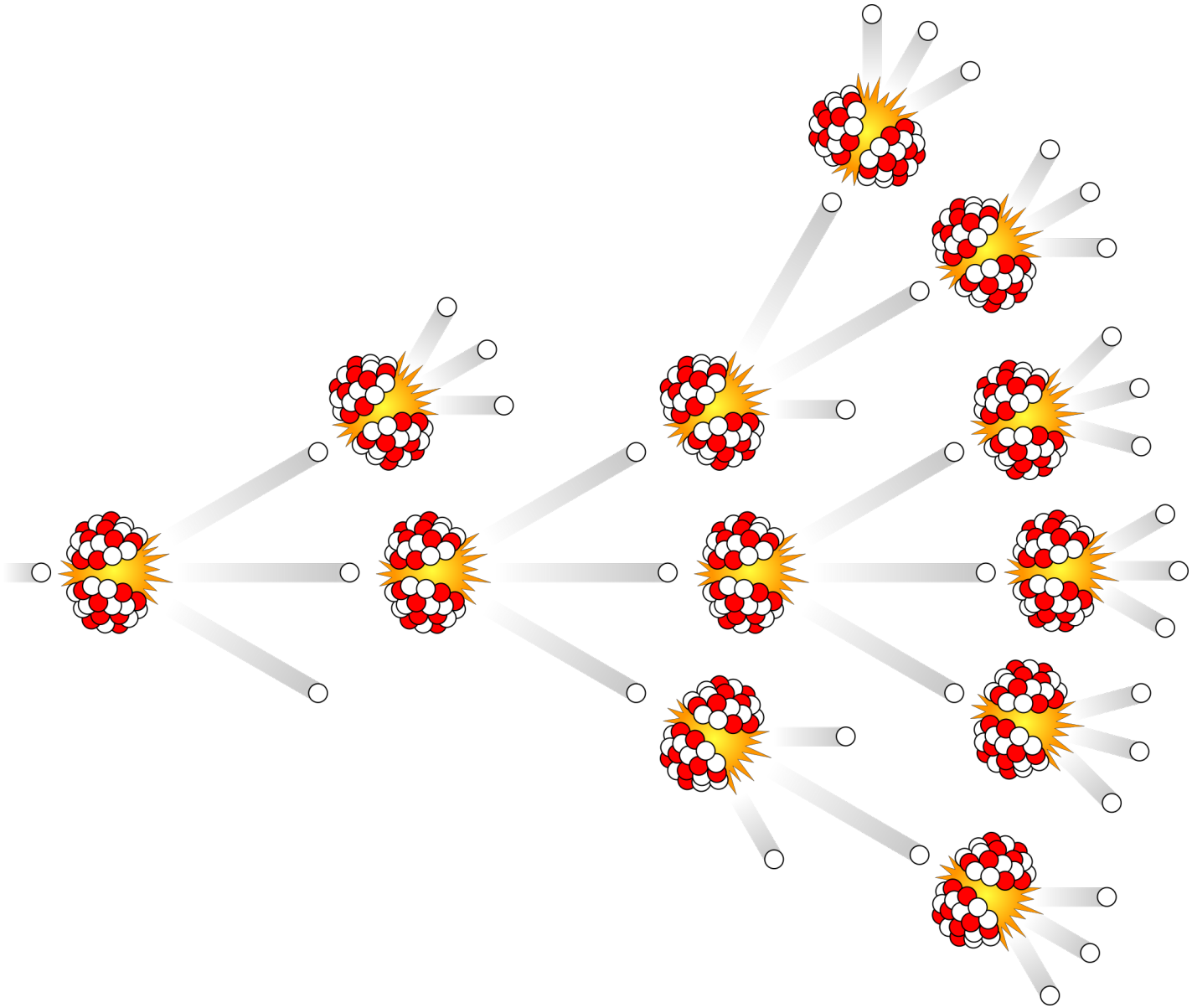
Analyse rekursiver Programme

Exkurs: Kettenreaktion bei der Spaltung von ^{235}U

- Atomkerne bestehen aus Protonen und Neutronen.
  - Die Anzahl der Protonen bestimmt das chemische Element (Uran: 92 Protonen).
  - Daneben enthält jeder Atomkern Neutronen.
- Atomkerne des Uran-Isotops ^{235}U (92 Protonen, 143 Neutronen) können durch Neutronen gespalten werden, so dass zwei kleinere Atome entstehen und Wärmeenergie frei wird.
- **Beim Zerfall entstehen 3 neue Neutronen**, die wiederum bis zu drei weitere ^{235}U-Kerne spalten können, diese bis zu 9 weitere, dann 27, 81, 243, … (exponentielle Kettenreaktion).

Analyse rekursiver Programme

Exkurs: Kettenreaktion bei der Spaltung von ^{235}U



Analyse rekursiver Programme

Exkurs: Kettenreaktion bei der Spaltung von ^{235}U

Bikini-Atoll (1946)
http://petapixel.com/2013/02/18/photos-from-the-worlds-first-underwater-nuclear-explosion/

Analyse rekursiver Programme

Komplexität eines rekursiven Programms

```
static long exp2(int n)
{
  assert(n>=0);
  return (n==0) ? 1 : exp2(n-1) + exp2(n-1);
}
```

- Bei `n>0` erzeugt **jede** Rekursionsstufe **zwei** weitere Inkarnationen mit `n-1` als Parameter:
  - Der **Aufrufbaum** (siehe Folien 35, 39ff, 59, 63ff und VL06) besteht also insgesamt aus exponentiell vielen Inkarnationen, die bearbeitet werden müssen. Zeitkomplexität: $O(2^n)$
  - Es sind aber zu jedem Zeitpunkt höchstens $n+1$ Rekursionsstufen gleichzeitig auf dem Aufrufstapel. Speicherkomplexität, Rekursionstiefe: $O(n)$

Analyse rekursiver Programme

Komplexität eines rekursiven Programms

```
static long fakRek(int n)
{
  assert(n>=0);
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```

- Jede Rekursionsstufe erzeugt höchstens eine weitere Inkarnation mit `n-1` als Parameter:
  - Zeitkomplexität: $O(n)$
  - Speicherkomplexität: $O(n)$
  - Rekursionstiefe: $O(n)$

Formen der Rekursion

Formen der Rekursion

Direkte vs indirekte Rekursion

- Das bisher angegebene Schema zeigt die **direkte Rekursion**, bei der sich eine Prozedur oder Funktion direkt selbst aufruft.
- Schwieriger zu überblicken ist dagegen der Fall, dass eine Prozedur / Funktion `A` eine andere Prozedur / Funktion `B` aufruft, die ihrerseits wiederum `A` aufruft.
  - Diese Fälle nennt man **indirekte Rekursion**.
  - Hiervon sind beliebig komplizierte Varianten möglich.
  - In dieser Veranstaltung beschäftigen wir uns daher nur mit direkter Rekursion.

Formen der Rekursion

Einfache vs. mehrfache Rekursion

- Bei vielen rekursiven Algorithmen wird auf jeder Rekursionsstufe nur **ein einziger Ausführungspfad** weiterverfolgt (jede Inkarnation erzeugt also höchstes eine weitere Inkarnation):
  $P(n) \rightarrow P(n\text{-}1) \rightarrow P(n\text{-}2) \rightarrow \ldots \rightarrow P(1)$

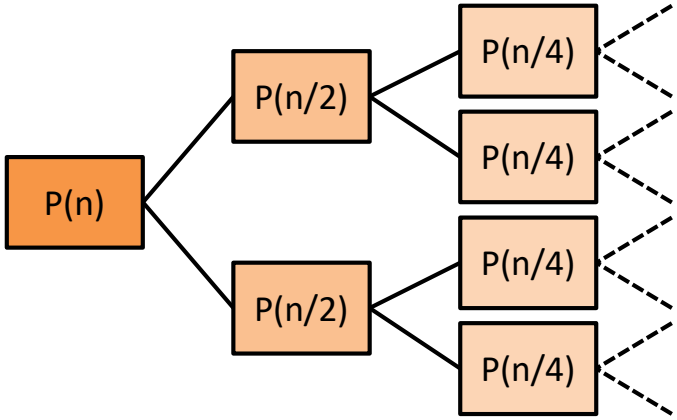
- Beispiel: Fakultät
  $n! = n * (n\text{-}1)! = n * (n\text{-}1) * (n\text{-}2)! = n * \ldots * 1$

- Andere Algorithmen erzeugen pro Rekursionsstufe **mehrere neue Inkarnationen**.

Formen der Rekursion

Divide and Conquer (Teile und Herrsche)

- **Divide & Conquer**-Verfahren zur Lösung eines Problems der Größe n arbeiten wie folgt:
  1. Divide: Teile das Problem der Größe n in (wenigstens) zwei annähernd gleich große Teilprobleme.
  2. Conquer: Löse die Teilprobleme unabhängig voneinander auf dieselbe Art (rekursiv). Wenn ein Teilproblem klein genug ist, löse es direkt.



  3. Combine: Kombiniere die Teillösungen zur Gesamtlösung.

Formen der Rekursion

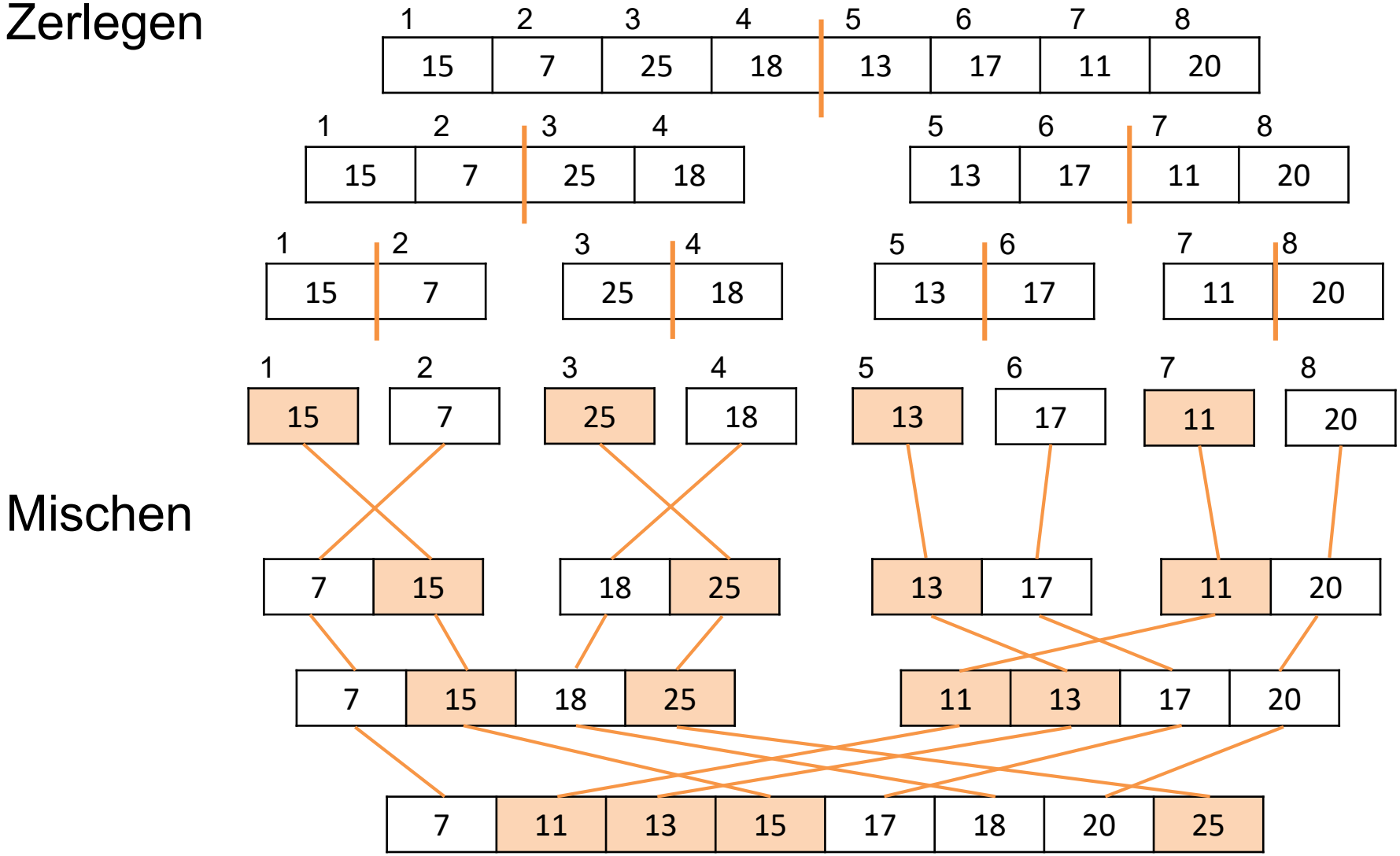
Divide and Conquer: Beispiel

Mergesort (dt. Sortieren durch Mischen)

- **Zerlege** das zu sortierende Feld in zwei (ungefähr) gleich große Teile:
  - Sortiere die Teilfelder erneut (Rekursion)
  - Wenn ein Teilfeld der Länge kleiner gleich eins auftritt, ist dieses bereits sortiert und die Rekursion bricht ab
- Bei der Rückkehr aus der Rekursion werden jeweils zwei sortierte Teilfelder zu einem sortierten Gesamtfeld **gemischt**.

Formen der Rekursion

Divide and Conquer: Beispiel



Weitere Details zu Mergesort und anderen Sortierverfahren folgen in späteren Vorlesungen.

Formen der Rekursion

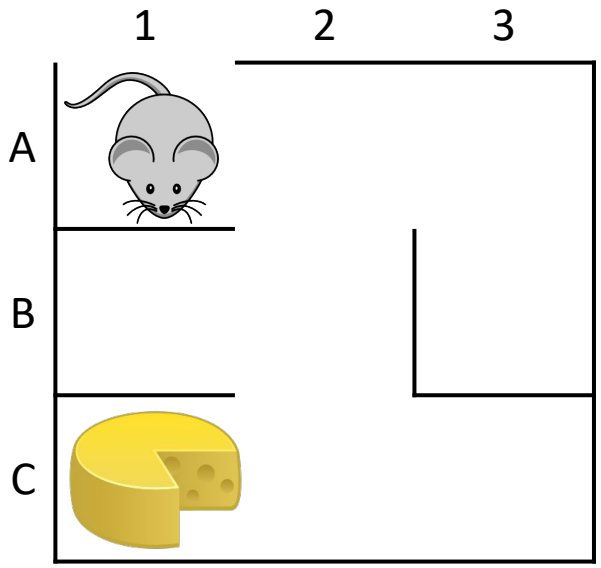
Backtracking

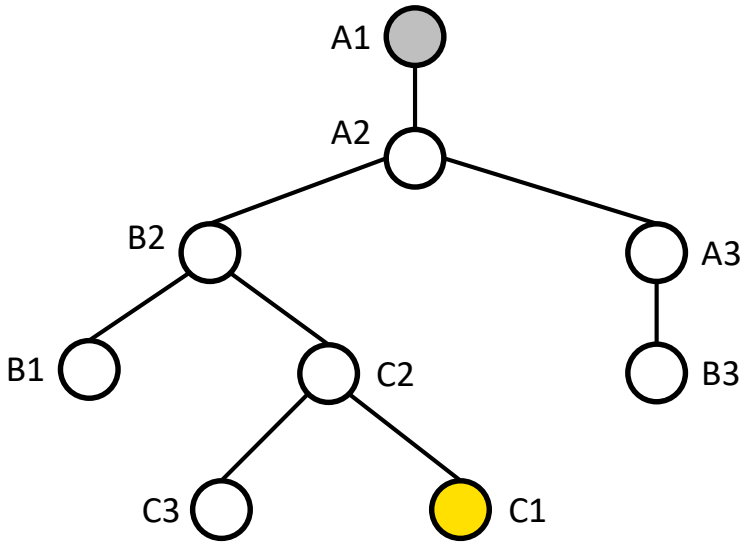
- Backtracking gestattet es, ein Problem durch systematisches Durchprobieren zu lösen.
- Backtracking-Verfahren versuchen systematisch, eine Teillösung schrittweise zu einer Gesamtlösung auszubauen.
  - Ist bei einer Teillösung kein weiterer Ausbau möglich (Sackgasse), dann werden die zuletzt durchgeführten Schritte so weit zurückgenommen (daher der Name) bis es Alternativschritte gibt
  - Mit dem Alternativschritt wird fortgesetzt
- Existiert eine Lösung, so wird sie gefunden, da der Lösungsraum systematisch abgesucht wird.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



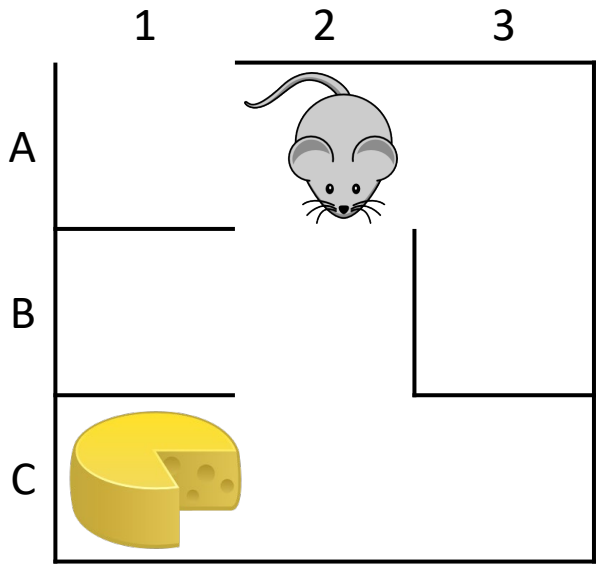


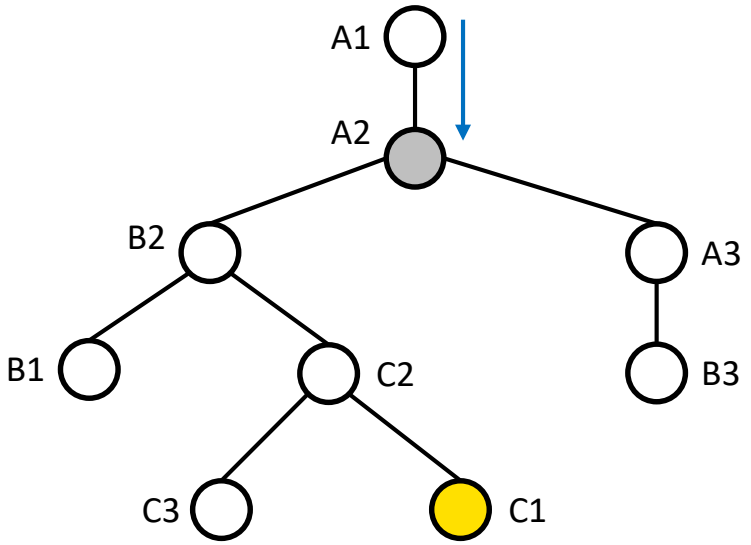
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



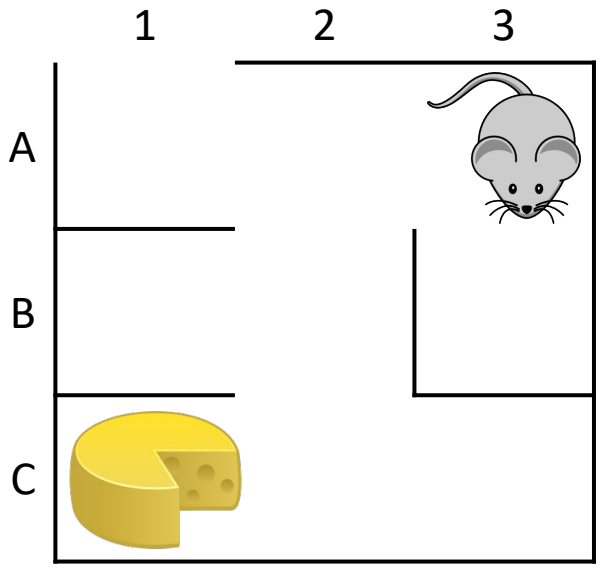


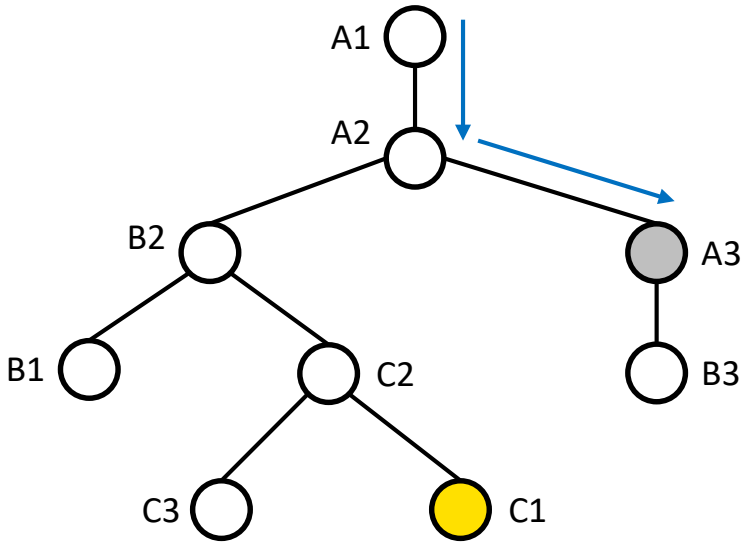
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



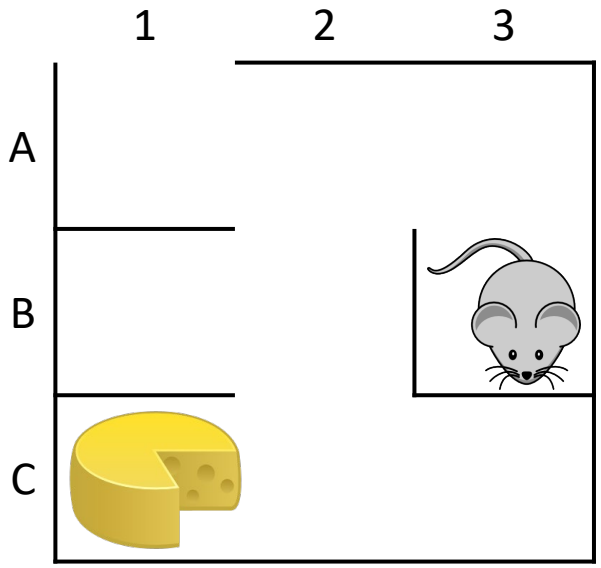


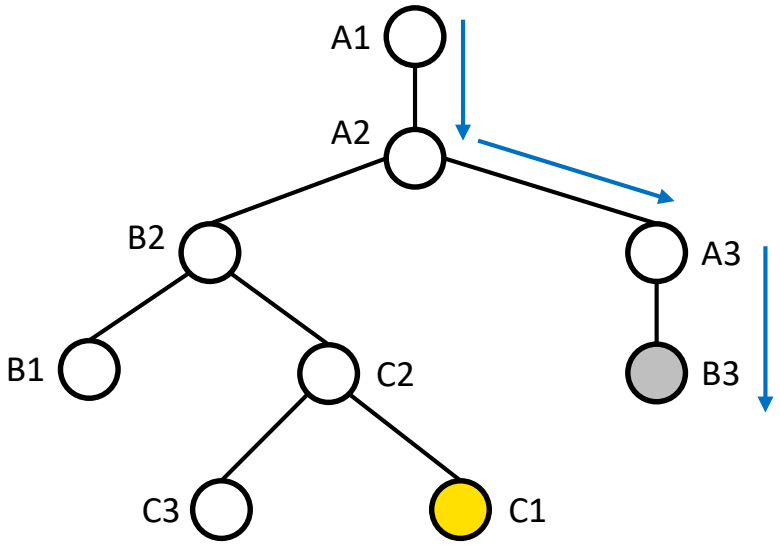
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



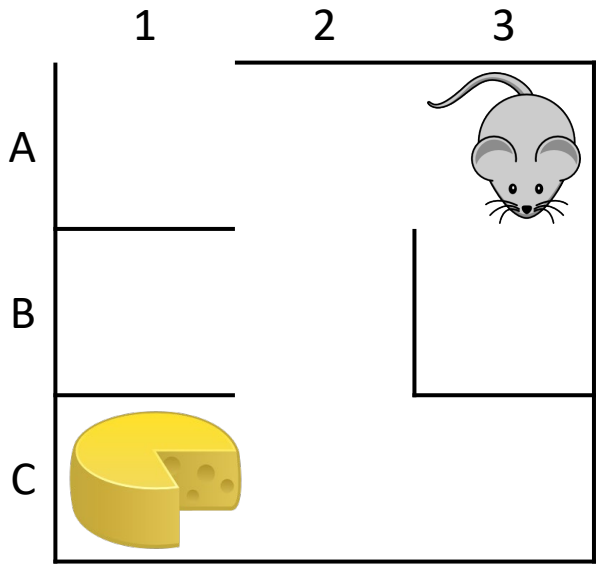


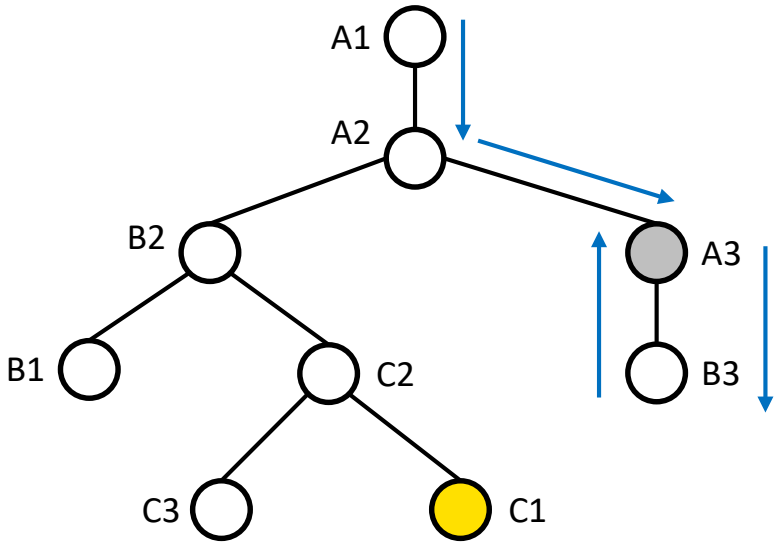
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



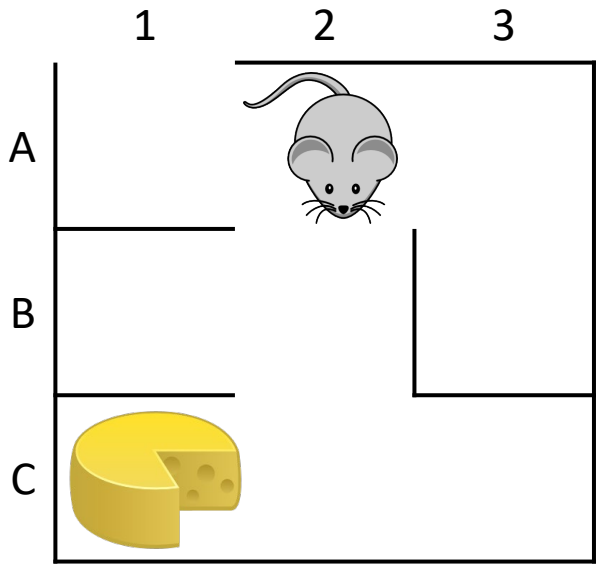


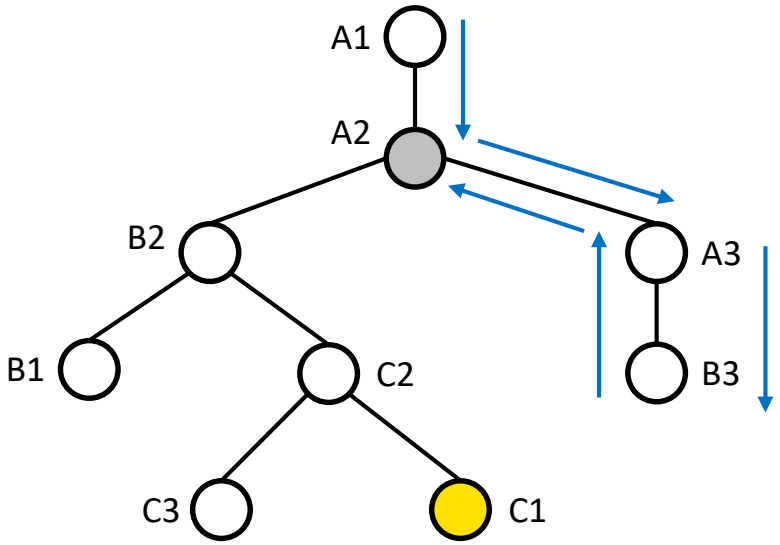
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



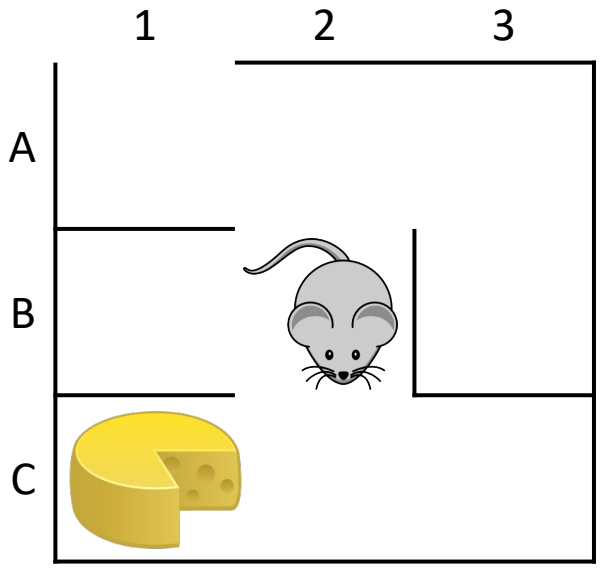


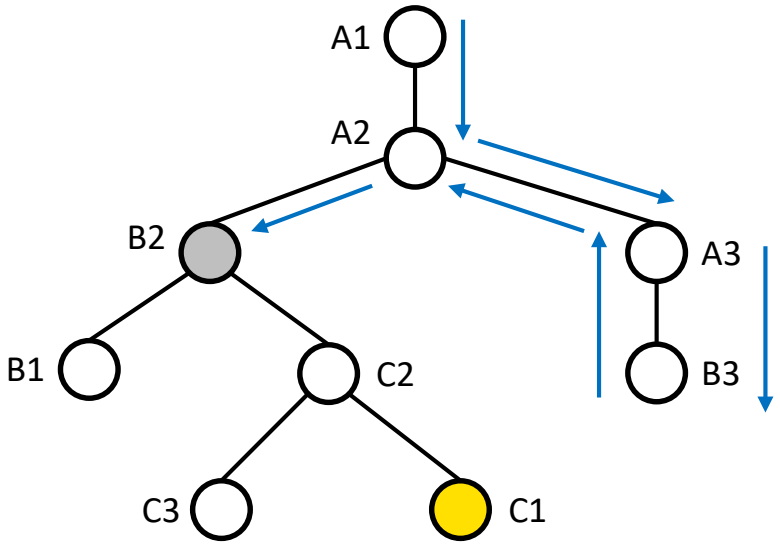
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



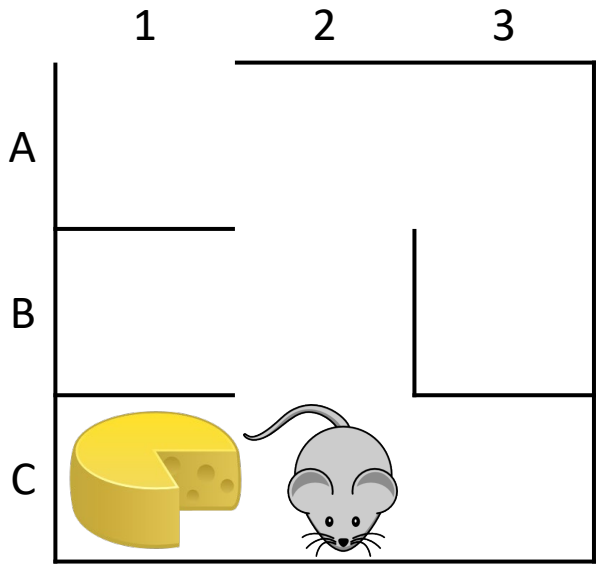


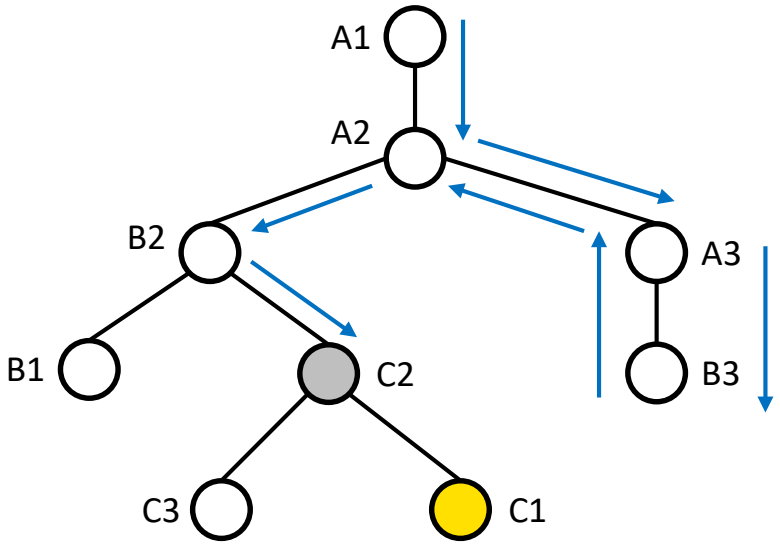
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?



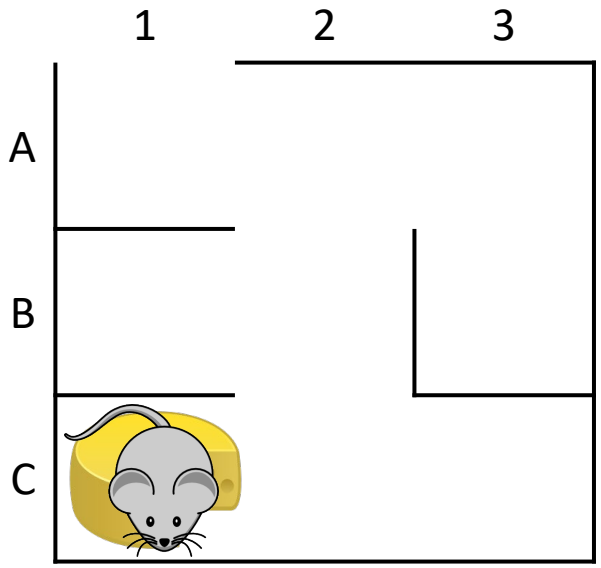


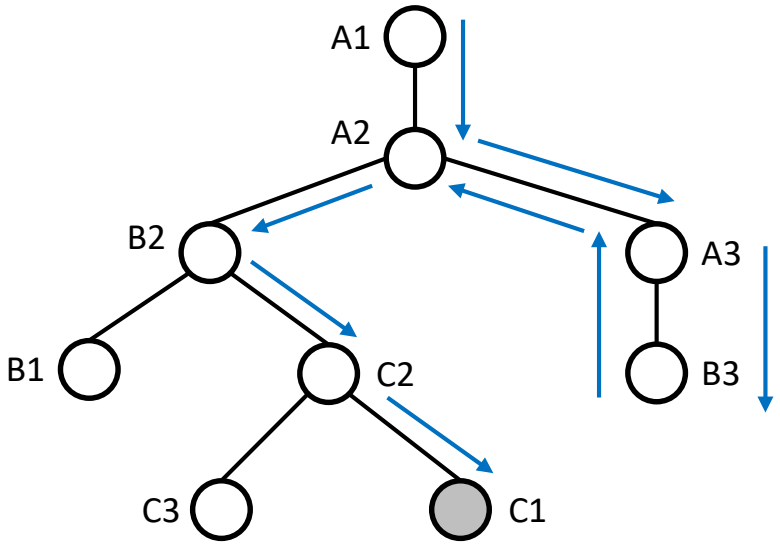
- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.

Formen der Rekursion

Backtracking: Beispiel

- Wie kommt die Maus zum Käse?





- Backtracking: bei der Suche gerät man in Sackgassen und geht zur nächsten unbearbeiteten Abzweigung zurück.
- Der Lösungsraum wird vollständig durchlaufen, so dass immer eine Lösung gefunden wird, sofern diese existiert.

Formen der Rekursion

Backtracking: Abstraktion

- Darstellung als Pseudocode

```
sucheLösung(teillösung)
  if teillösung komplett then
    gib teillösung aus
    beende programm
  else
    for alle möglichen Schritte s do
      erweitere teillösung um s
      sucheLösung(teillösung)
      nimm s zurück
    end for
    // Es wurde keine Lösung gefunden -> vorige Stufe zurücknehmen
end if
```

- Das Backtracking-Verfahren wird anfangs mit einer leeren Teillösung gestartet:

```
sucheLösung(leereTeillösung)
```

Iterative Lösung oder Rekursion?

Iterative Lösung oder Rekursion?

Grundsätzliches

- Fast jedes Problem lässt sich rekursiv begreifen und lösen:
  - z.B. Ausgabe einer verketteten Liste: Ausgabe des ersten Elements der Liste, dann rekursiver Aufruf für Rest-Liste

- Jedes Problem lässt sich iterativ lösen:
  - Falls ein rekursives Problem nicht auf einfache und offensichtliche Weise iterativ gelöst werden kann, muss notfalls die Rekursion durch einen eigenen Stack oder eine eigene Queue simuliert werden.

Iterative Lösung oder Rekursion?

Rekursiv definierte Probleme

- Bei rekursiv definierten Problemen ist eine rekursive Lösung naheliegend, sonst eine iterative Lösung.
  - Dies gilt insbesondere, wenn auf jeder Rekursionsstufe mehrere Pfade verfolgt werden müssen (Beispiele: erschöpfende Suche im Lexikon oder Verzeichnisbaum, Pythagoras-Baum, Türme von Hanoi).
    - Hier ist in der Regel die Aufrufhäufigkeit größer als die Rekursionstiefe!

- Wird immer nur höchstens ein Pfad weiterverfolgt, ist eine iterative Lösung sinnvoller (Beispiel: Suchbaum aus VL06).
  - Häufiges Muster ist dabei eine **Endrekursion** (Pseudocode):

```
RekursiveProc(…)
  Anweisungen
  if Bedingung then
    RekursiveProc(…)
  endif
```

Iterative Lösung oder Rekursion?

Iterative Lösung für Fakultät

- Rekursive Lösung für Fakultät:

```
static long fakRek(int n)
{
  assert(n>=0);
  return (n<=1) ? 1 : n*fakRek(n-1);
}
```

  – Es wird immer nur ein Ausführungspfad weiterverfolgt!

- Iterative Lösung für Fakultät:

```
static long fakIter(int n)
{
  long fak = 1;
  for(int a = 2; a <= n; a++)
    fak *= a;
  return fak;
}
```

  – Iterative Lösungen benötigen oft zusätzliche Variablen als Akkumulator.

Türme von Hanoi

Türme von Hanoi

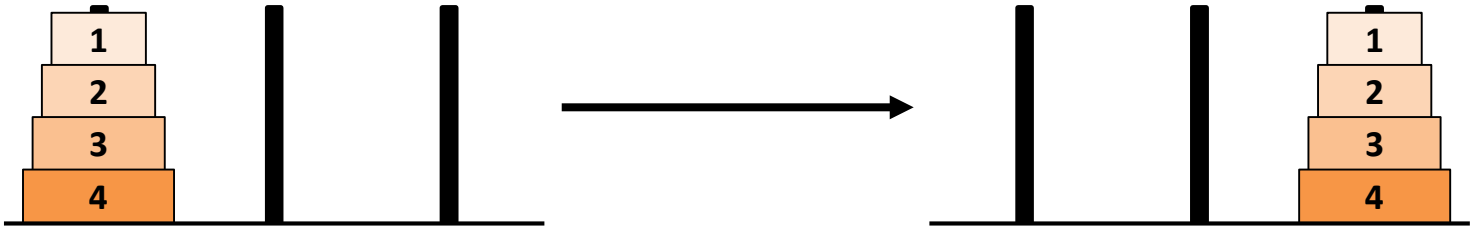
Problemstellung

- Klassisches Rekursions-Beispiel der Informatik:
  - Nach einer alten Legende standen einmal drei goldene Säulen in einem Tempel in Hanoi. Auf einer der Säulen befanden sich 100 Scheiben, jedes mal eine kleinere auf einer größeren Scheibe.
  - Ein alter Mönch bekam den Auftrag, den Scheibenturm von der ersten auf die dritte Säule zu versetzen, wobei er jeweils nur die oberste Scheibe von einem Turm nehmen und nie eine größere Scheibe auf eine Kleinere legen durfte.
  - Die Legende sagt vorher: wenn der Mönch seine Arbeit erledigt habe, wird das Ende der Welt kommen.

Türme von Hanoi

Problemstellung

- Lösungsidee?



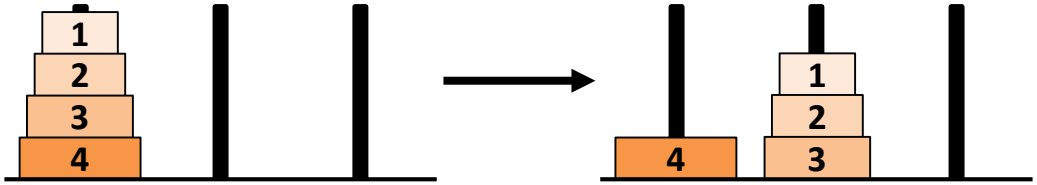
Erlaubt

Verboten

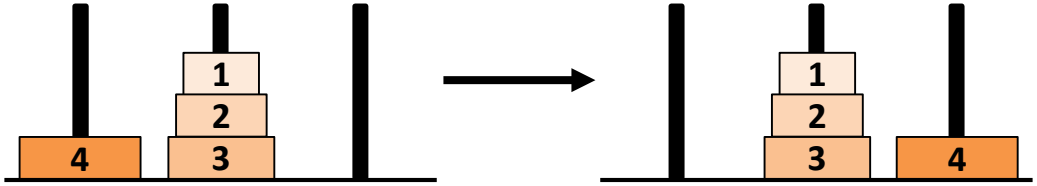
Türme von Hanoi

Rekursive Lösungsidee

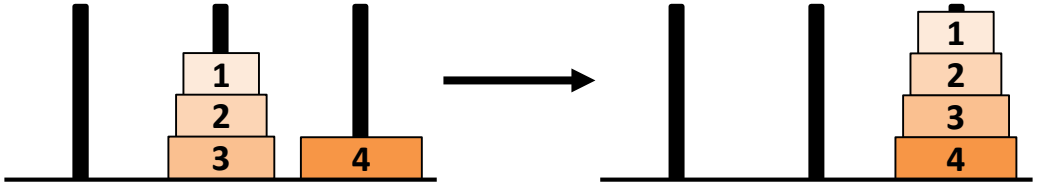
- Rekursionsbasis $P(1)$: für ein einzige Scheibe kann das Problem direkt gelöst werden.
- Rekursionsvorschrift $P(n) \rightarrow P(n\text{-}1)$:
  - Transportiere den Turm, bestehend aus den oberen $n\text{-}1$ Scheiben von Säule 1 nach Säule 2:



  - Transportiere die letzte, größte Scheibe von Säule 1 nach Säule 3:



  - Transportiere zum Schluss den Turm von $n\text{-}1$ Scheiben von Säule 2 nach Säule 3:



Türme von Hanoi

Algorithmus umgangssprachlich

- Besteht der Turm aus einer Scheibe:
  - Trage die Scheibe von der Ausgangssäule zur Zielsäule.
- Besteht der Turm aus mehr als einer Scheibe:
  - Transportiere einen Turm von $n-1$ Scheiben von der Ausgangssäule zur Hilfssäule.
  - Trage dann eine Scheibe von der Ausgangssäule zur Zielsäule.
  - Transportiere einen Turm von $n-1$ Scheiben von der Hilfssäule zur Zielsäule.
- **Anmerkung:** welche Säule jeweils Ausgangs-, Hilfs- und Zielsäule ist, ändert sich mit jedem Schritt!

Türme von Hanoi

Algorithmus in Pseudocode

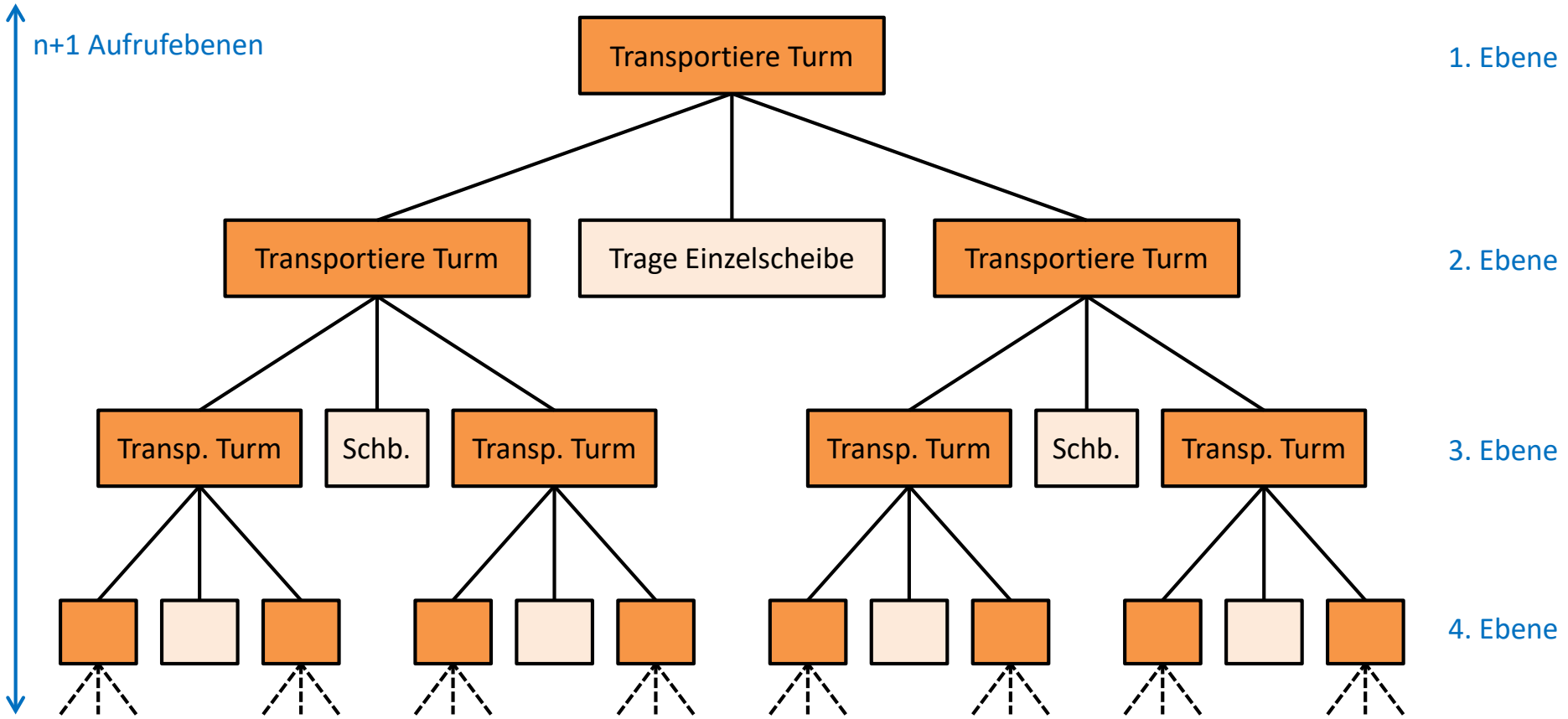
```
transportiereTurm(n, turm1, turm2, turm3)
  // Transportiere n Scheiben von turm1 nach turm3
  // mittels turm2
  if n >= 1 then
    // Rekursiver Aufruf
    if n > 1 then
      transportiereTurm(n-1, turm1, turm3, turm2)
    end if
    trageScheibe(turm1, turm3)
    // Rekursiver Aufruf
    if n > 1 then
      transportiereTurm(n-1, turm2, turm1, turm3)
    end if
  end if

trageScheibe(turm1, turm2)
  // Trage eine Scheibe von turm1 nach turm2
  nimmObersteScheibe(turm1)
  legeScheibeAuf(turm2)
```

Türme von Hanoi

Fragen der Informatiker

- Wie groß ist die Zeitkomplexität des Algorithmus?



Türme von Hanoi

Fragen der Informatiker

- Wie groß ist der Zeitaufwand des Algorithmus?
  - 1 Aufruf zum Transport der größten Scheibe
  - 2 Aufrufe zum Transport der zweitgrößten Scheibe
  - 4 Aufrufe zum Transport der drittgrößten Scheibe
  - 8 Aufrufe zum Transport der viertgrößten Scheibe
  - …
  - 2^{n-1} Aufrufe zum Transport der kleinsten Scheibe

$$\sum_{i=1}^{n} 2^{i-1} = \sum_{i=0}^{n-1} 2^{i} = 2^{n} - 1$$

- Der Zeitaufwand für „Türme von Hanoi“ ist also $O(2^n)$.

Türme von Hanoi

Fragen der Informatiker

- Was ist der maximal bearbeitbare Problemumfang?
  - Umfang eines Problems, der mit Algorithmen der Ordnung $O(2^n)$ in gegebener Zeit bearbeitet werden kann ($c = 10^{-6}s$)
  - Beispielrechnung für 1 Sekunde:
    $1s = 10^{-6}s * 2^n \Leftrightarrow 10^6 s = 2^n s \Leftrightarrow n = \log_2(1000000) \approx 19$
  - Maximal bearbeitbarer Problemumfang:
    - In 1 Sekunde: 19 Scheiben
    - In 1 Minute: 25 Scheiben
    - In 1 Stunde: 31 Scheiben
    - In 60 Stunden: 37 Scheiben

Türme von Hanoi

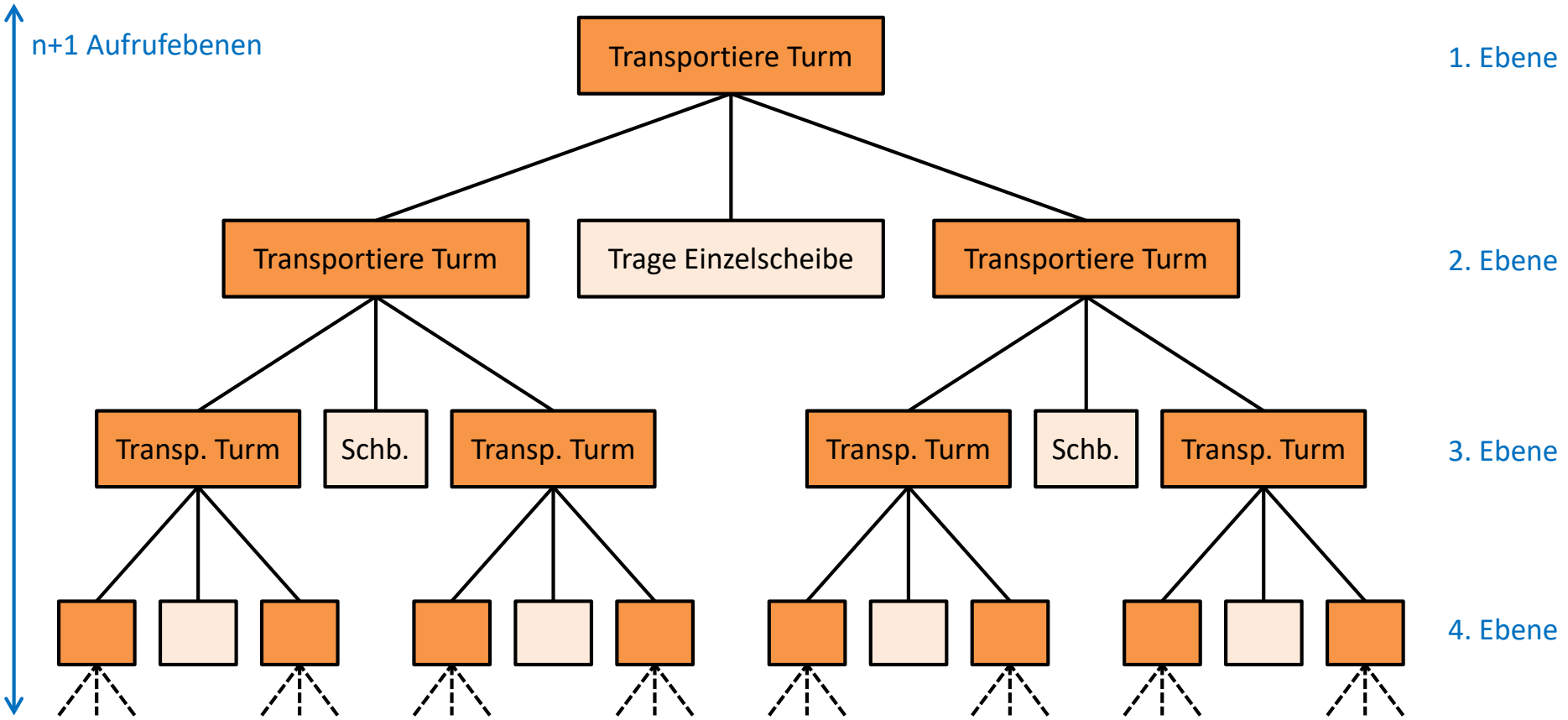
Hat die Legende Recht?

- Wie lange dauert es, bis die Arbeit an den Türmen von Hanoi beendet ist, wenn in jeder Sekunde eine Scheibe transportiert wird? Hat die Legende Recht?
  - Um 100 Scheiben von Säule 1 nach Säule 3 zu bringen, müssen 2^{100}-1 Scheiben transportiert werden. Daraus ergibt sich: $2^{100}\text{-}1 \approx 1{,}2676 * 10^{30}\text{s} \approx 4 * 10^{16}\ \text{Jahrmillionen}$
  - Die Prophezeiung der Legende kann also durchaus in Erfüllung gehen!

Türme von Hanoi

Zeitkomplexität

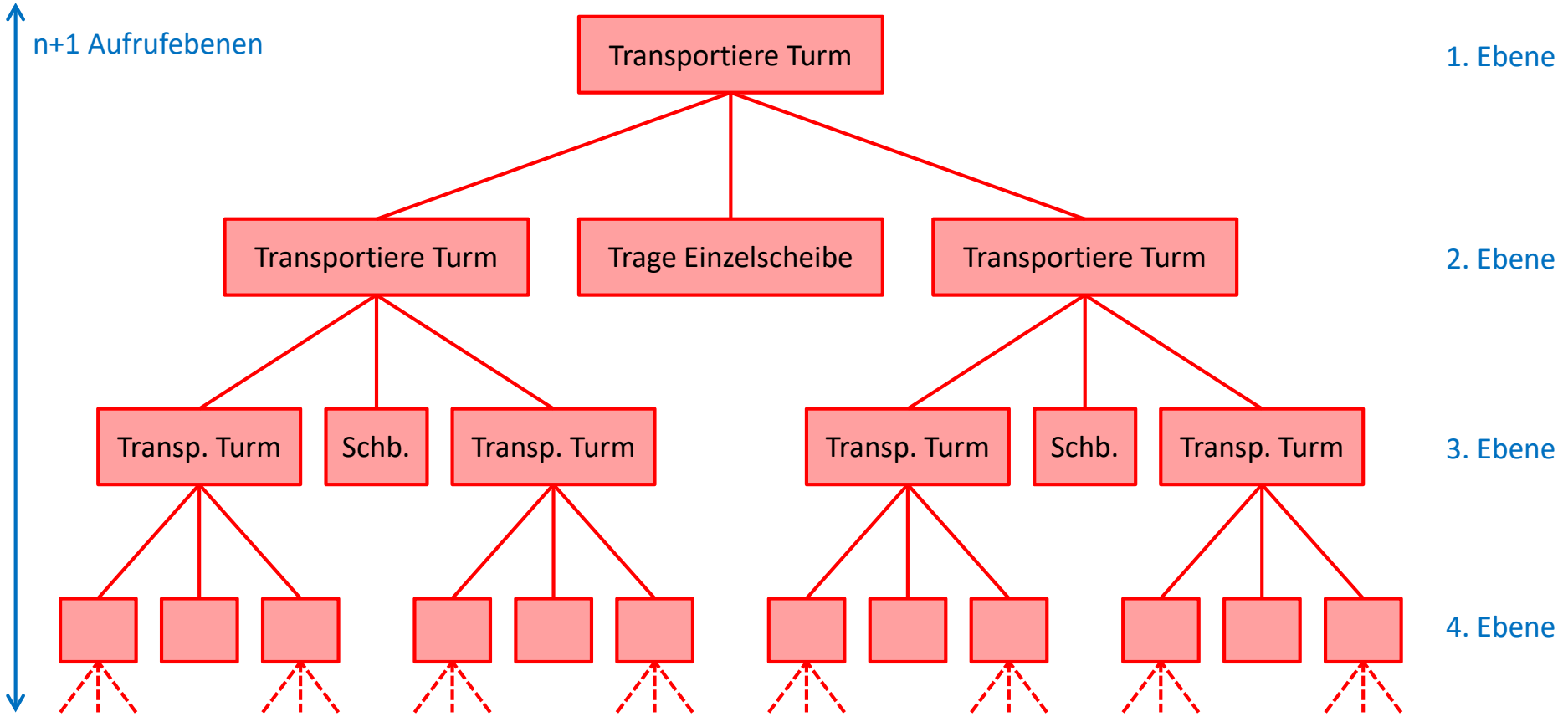
- Wie groß ist die Zeitkomplexität des Algorithmus?



Türme von Hanoi

Zeitkomplexität

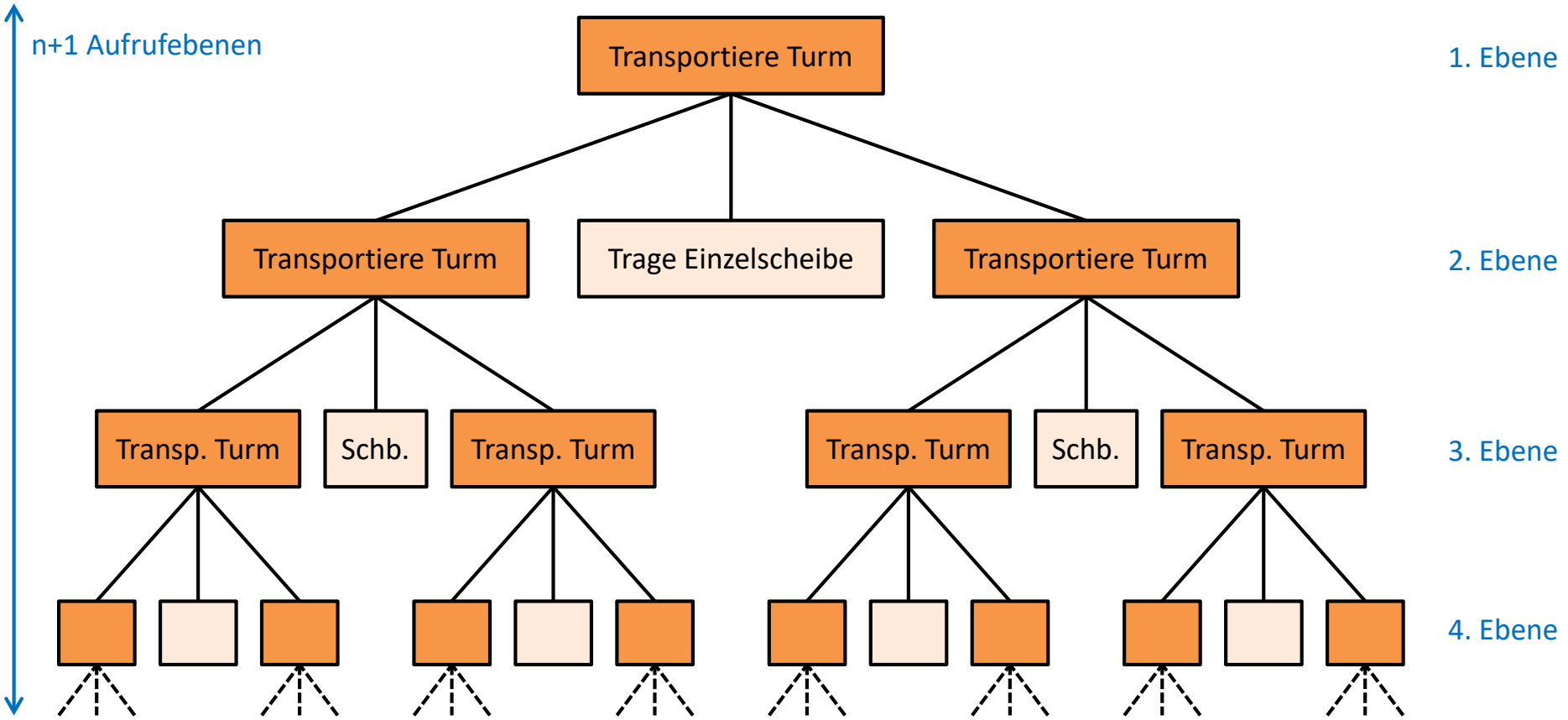
- Wie groß ist die Zeitkomplexität des Algorithmus?
  - $O(2^n)$, da **nacheinander** alle Operationen im Aufrufbaum (rot) ausgeführt werden müssen (Anzahl):



Türme von Hanoi

Speicherkomplexität

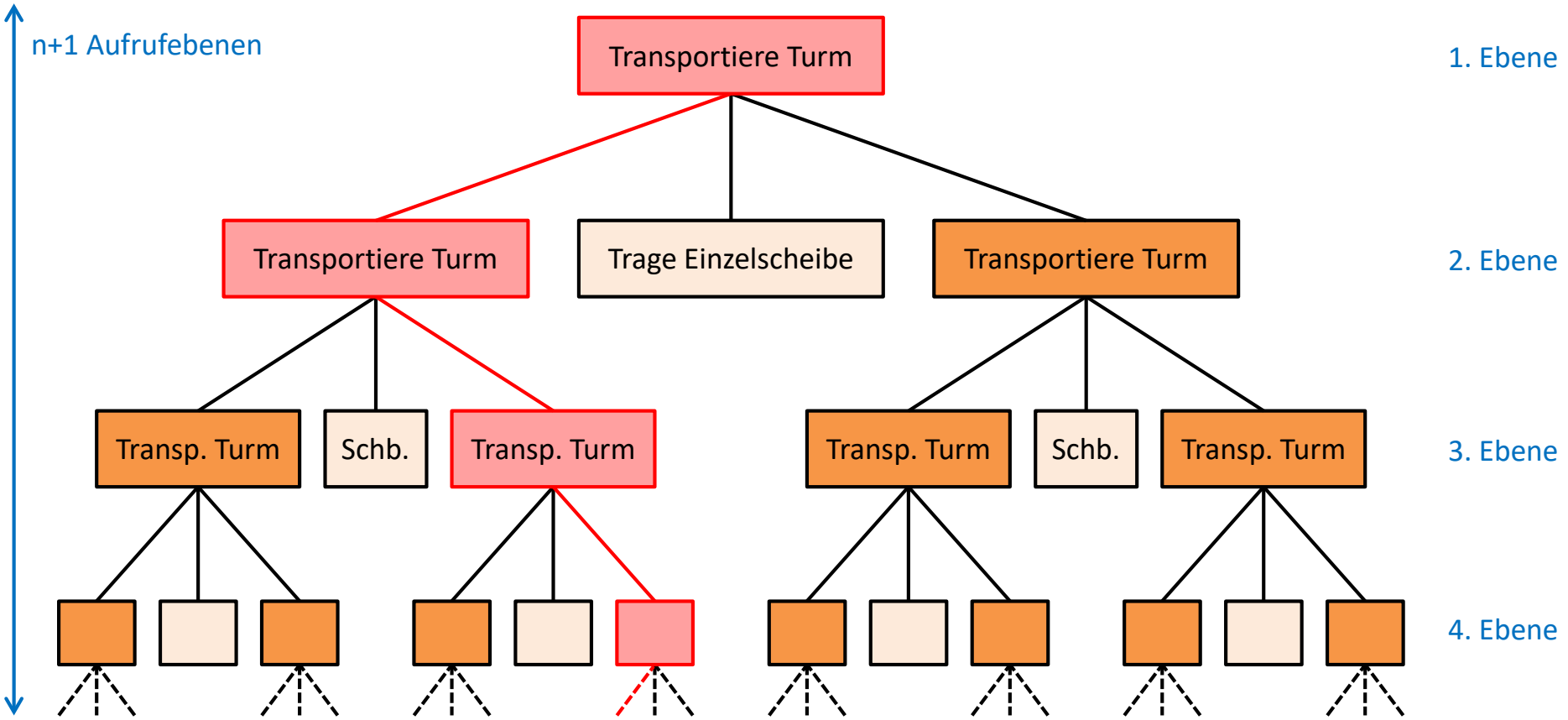
- Wie groß ist die Speicherkomplexität des Algorithmus?



Türme von Hanoi

Speicherkomplexität

- Wie groß ist die Speicherkomplexität des Algorithmus?
  - $O(n)$, da zu jedem Zeitpunkt höchstens $n+1$ Aufrufe (z.B. roter Pfad) **gleichzeitig** aktiv sind (maximale Rekursionstiefe):



Türme von Hanoi

Rekursionstiefe und Anzahl der Aufrufe

- In unserem Beispiel ($n=4$) sind also im Verlauf des Scheibentransportes maximal vier Methodenaufrufe für `transportiereTurm` und ein Aufruf von `trageScheibe` gleichzeitig aktiv, die Rekursionstiefe beträgt somit 4.
- Insgesamt müssen aber 15 Rekursionsstufen bearbeitet werden.

- Wie in fast allen sinnvollen Anwendungsfällen für Rekursion ist also die Zahl der zu bearbeitenden Aufrufe größer als die Rekursionstiefe (die Anzahl der **gleichzeitig aktiven** Aufrufe).

Lernziele

- Sie können das Prinzip der Rekursion an Beispielen erläutern
- Sie kennen die zentralen Elemente der Rekursion:
  - Rekursionsbasis
  - Rekursionsvorschrift
  - Dynamische Endlichkeit
- Sie können rekursive Definitionen nachvollziehen
- Sie können die Algorithmenmuster „Divide and Conquer“ und „Backtracking“ erläutern
- Sie können einfache rekursive Lösungen in iterative Lösungen umwandeln
- Sie können rekursive Algorithmen und Programme nachvollziehen und deren Zeit- und Speicherkomplexität bestimmen
- Sie können rekursive Methoden in Java implementieren

Literatur

- Quellen:
  - Balzert, H.: Grundlagen der Informatik
    (LE 16: Aufwand von Algorithmen)
  - Saake, G.: Algorithmen und Datenstukturen
    Kapitel 8.1.3: Problemreduzierung durch Rekursion
    Kapitel 8.3: Rekursion: Divide and Conquer
    Kapitel 8.4: Rekursion: Backtracking
    Kapitel 8.5: Dynamische Programmierung