Algorithmen und Datenstrukturen

Rekursion

Prof. Justus Piater, Ph.D.

23. März 2022

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

1 Fakultät

> Viele Probleme lassen sich als Erweiterung einer kleineren Version desselben Problems formulieren. Eine solche Formulierung nennt man ***rekursiv***. Dieses Prinzip im Allgemeinen und die rekursive Lösung von Problemen im Speziellen bezeichnet man als ***Rekursion***.

Fakultät: iterative Version [Slide 1]

$$n! = \prod_{i=1}^{n} i$$

```
Algorithm factorial(n):
  Require: Integer n ≥ 0.
  Ensure: Return n!.
  f ← 1
  for i ← 2 to n do
    f ← fi
  return f
```

Schauen wir uns zunächst eine einfache Funktion in einer nicht-rekursiven Formulierung an. n-Fakultät können wir als das Produkt der ganzen Zahlen von 1 bis n formulieren. Diese Definition lässt sich leicht in einen iterativen Algorithmus zur Berechnung von n-Fakultät übersetzen: Wir beginnen mit dem Wert 1, und multiplizieren dort alle folgenden ganzen Zahlen bis einschließlich n nacheinander hinein.

Fakultät: rekursive Version [Slide 2]

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{Rekursionsanfang} \\ n(n-1)! & \text{otherwise} \quad \text{Rekursionsschritt} \end{cases}$$

Beispiel

$4! = 4 \cdot 3! = 4 \cdot (3 \cdot 2 \cdot 1)$

```
Algorithm factorial(n):
  Require: integer n ≥ 0.
  Ensure: Return n!.
  if n = 0 then
    return 1                    // Rekursionsanfang
  else
    return n×factorial(n − 1)   // Rekursionsschritt
```

Betrachten wir nun diese ***rekursive*** Definition der Fakultät. Hier definieren wir n-Fakultät als n mal $n-1$-Fakultät. Die Rekursion dieser Definition besteht darin, dass die Fakultät durch die Fakultät definiert wird.

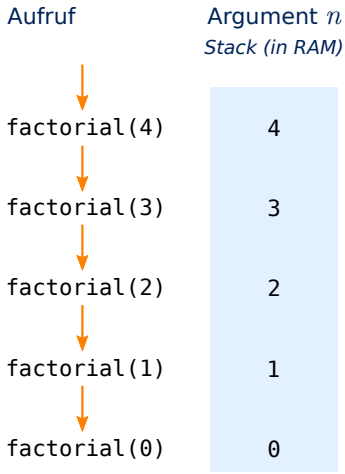
Das funktioniert natürlich nur, wenn die Rekursion irgendwo endet. Dies ist hier für $n = 0$ der Fall: 0-Fakultät wird gesondert behandelt und ist als der Wert 1 definiert anstatt rekursiv als $0 \cdot (0-1)$-Fakultät.

Dies ist der sogenannte ***Rekursionsanfang***. Den anderen Fall $n(n-1)!$ bezeichnet man als den ***Rekursionsschritt***. Jede Rekursion beinhaltet einen Rekursionsanfang und einen Rekursionsschritt.

Unsere rekursive Definition der Fakultät lässt sich wiederum unmittelbar in einen rekursiven Algorithmus zur Berechnung der Fakultät von n übersetzen. Die Rekursion dieses Algorithmus besteht darin, dass er sich selbst aufruft.

Aufrufbaum: `factorial(4)` [Slide 3]

```
Algorithm factorial(n):
  if n = 0 then
    return 1
  else
    return n×factorial(n − 1)
```



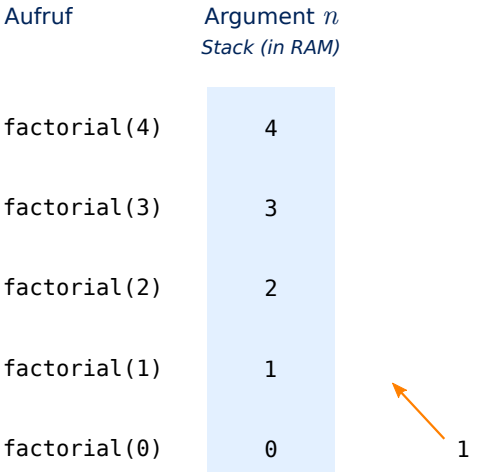
Wird `factorial(4)` aufgerufen, ist $n = 4$ und nicht 0; daher ruft sich die Funktion `factorial` im `else`-Zweig selbst auf, mit dem Argument $n - 1 = 3$. Wir haben nun zwei aktive Aufrufe dieser Funktion, von denen jeder einen eigenen Speicherbereich für sein Argument besitzt.

Im Speicherbereich des ersten Aufrufs von `factorial` liegt die Variable n mit dem Wert 4, und im Speicherbereich des zweiten Aufrufs von `factorial` liegt die Variable n mit dem Wert 3. Der erste Aufruf wartet nun, bis der zweite Aufruf endet und seinen Rückgabewert liefert, um diesen mit seinem Wert von n zu multiplizieren und seinerseits an seinen Aufrufer zurückzugeben.

Der zweite Aufruf von `factorial` verhält sich genauso wie der erste: Er ruft seinerseits rekursiv `factorial` auf, mit dem Argument $n - 1 = 2$. Diese rekursiven Aufrufe setzen sich fort, bis schließlich beim 5. Aufruf $n = 0$ und damit der Rekursionsanfang erreicht ist.

Aufrufbaum: `factorial(4)` [Slide 4]

```
Algorithm factorial(n):
  if n = 0 then
    return 1
  else
    return n×factorial(n − 1)
```

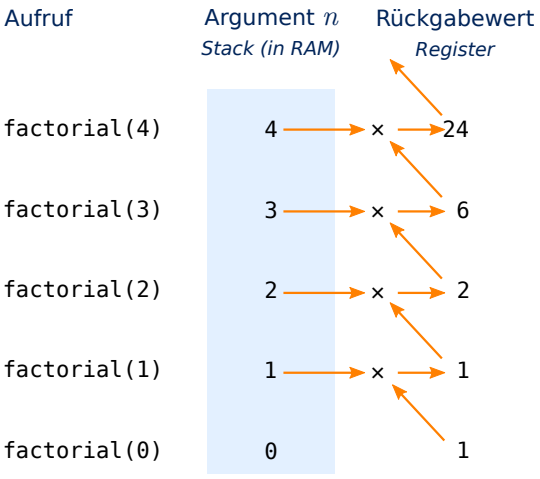


Spätestens hier wird deutlich, dass die Bezeichnung *Rekursionsanfang* vielleicht nicht die glücklichste ist: Er bildet das *Ende* der Berechnung und nicht den Anfang. Der Begriff Rekursionsanfang soll ausdrücken, dass er die *Basis* bildet, auf der die Rekursion ruht.

- An dieser Stelle liefert die Funktion unmittelbar den Wert 1 zurück.

Aufrufbaum: `factorial(4)` [Slide 5]

```
Algorithm factorial(n):
  if n = 0 then
    return 1
  else
    return n×factorial(n − 1)
```

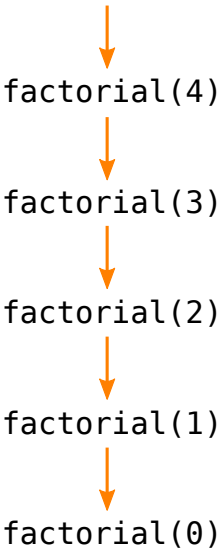


> Dieser Rückgabewert wird nun vom 4. Aufruf, der bis jetzt darauf gewartet hat, mit seinem Wert von n multipliziert, nämlich 1, und das Ergebnis an seinen Aufrufer zurückgeliefert, den 3. Aufruf.
>
> Auf diese Weise kehren auch der 3., 2. und schließlich der 1. Aufruf zurück, der uns das Endergebnis von `factorial(4)` liefert, nämlich 24.

Aufrufbaum und asymptotische Laufzeit [Slide 6]

***Aufrufbaum*:** Aufgerufene Funktionen sind die Kindknoten der aufrufenden Funktion.

```
Algorithm factorial(n):
  if n = 0 then
    return 1
  else
    return n×factorial(n − 1)
```



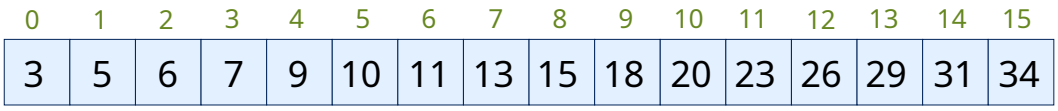
Asymptotische Laufzeit von `factorial()`: $O(n)$, da genau n rekursive Aufrufe stattfinden und die Laufzeiten aller anderen Anweisungen konstant sind.

Für rekursive Aufrufe liefert uns unser voriges Kapitel zur asymptotischen Laufzeitanalyse keine Hilfsmittel. Für `factorial()`, wie häufig, gibt uns der Aufrufbaum entscheidende Hinweise auf das Laufzeitverhalten.
Ein breit einsetzbares Werkzeug für die asymptotische Laufzeitanalyse rekursiver Algorithmen ist das sogenannte ***Master-Theorem***, das jedoch über den Umfang dieses Kurses hinausgeht.

> Die Abfolge der Funktionsaufrufe eines Algorithmus oder Programms bilden eine Baumstruktur: Aufgerufene Funktionen bilden die Kindknoten der aufrufenden Funktion.
>
> Insbesondere bei rekursiven Algorithmen stellt der Aufrufbaum ein wichtiges Hilfsmittel zur Analyse des Laufzeitverhaltens dar. Bei unserem `factorial`-Algorithmus ruft sich die `factorial`-Funktion genau einmal rekursiv auf; der Aufrufbaum bildet also eine lineare Liste.
>
> Wie lang ist diese Liste? Wir sehen im Algorithmus, dass sich der Wert von n bei jedem rekursiven Aufruf um 1 vermindert. Da $n = 0$ den Rekursionsanfang bildet, finden also genau n rekursive Aufrufe statt, und die Gesamtzahl der Aufrufe inklusive des ersten ist genau $n + 1$, also $O(n)$.
>
> Da alle anderen Anweisungen des Algorithmus $O(1)$ sind, also in von n unabhängiger Zeit ablaufen, ist die Gesamtlaufzeit von `factorial(n)` folglich $O(n)$.

2 Binärsuche

Suche in einer sortierten Sequenz [Slide 7]



- Ist die Sequenz *sortiert* und *indizierbar*, dann kann ***Binärsuche*** *(bisection search)* angewendet werden.

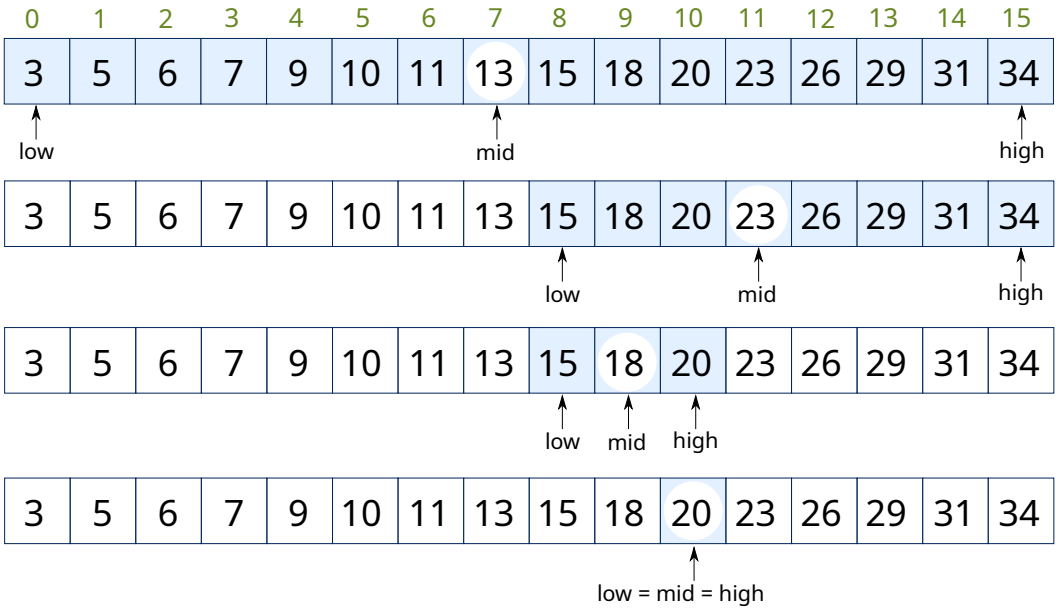
  Ein besserer Begriff wäre *Halbierungssuche*, denn es handelt sich um Suche mittels des (als solches bekannten) *Halbierungsverfahrens*. Der Begriff Halbierungssuche ist allerdings leider nicht gebräuchlich.

  Andernfalls muss man i.d.R. auf ***sequenzielle Suche*** zurückgreifen.

Binärsuche: Rekursive Suche in einer Hälfte der Sequenz.

> Ein typisches Beispiel für die Anwendung von Rekursion ist die Suche nach einem bestimmten Element in einer sortierten Sequenz. Die Ordnung der Sequenz können wir dahingehend nutzen, dasss wir einen Großteil der Elemente bei der Suche überhaupt nicht betrachten müssen und trotzdem garantiert das gesuchte Element finden bzw. die Abwesenheit dieses Elements feststellen können. Dies setzt voraus, dass wir auf jedes Element der Sequenz in konstanter Zeit per Index zugreifen können.

Binärsuche: Beispiel [Slide 8]



> Die Idee ist ganz einfach: Wir vergleichen das gesuchte Element mit dem Element in der Mitte der Sequenz. Sind beide identisch, dann haben wir es gefunden. Ist das gesuchte Element kleiner, suchen wir es rekursiv in der linken Hälfte der Sequenz; ist es größer, dann suchen wir es rekursiv in der rechten Hälfte.
>
> Falls das gesuchte Element in der Sequenz vorhanden ist, werden wir es auf diese Weise garantiert finden. Falls das Element nicht vorhanden ist, stellen wir dies fest, wenn die durchsuchte Teilsequenz leer ist.

Dieser Algorithmus ist im Deutschen als Binärsuche bekannt. Er englische Begriff *bisection search* ist viel treffender, denn er beschreibt in einem Wort, wie der Algorithmus funktioniert: In jedem Schritt zerschneidet er die Sequenz in zwei Hälften. Das aus dem Lateinischen stammende Verb *to bisect* bedeutet wörtlich, etwas in zwei Teile zu schneiden. Danach sucht der Algorithmus rekursiv in einer der beiden Hälften weiter.

Binärsuche [Slide 9]

```
Algorithm bisectionSearch(S, k, l, h):
  Require: An ordered vector S storing at least h entries,
           key k sought, and integers l (low) and h (high).
  Ensure: Return true iff k exists between l and h.
  if l > h then
    return false
  m ← ⌊(l + h)/2⌋
  e ← S[m]
  if k = e then
    return true
  else if k < e then
    return bisectionSearch(S, k, l, m − 1)
  else
    return bisectionSearch(S, k, m + 1, h)
```

Ist unserer Algorithmus `bisectionSearch` korrekt? Das heißt, findet er garantiert das gesuchte Element, falls es existiert, und stellt er garantiert seine Abwesenheit fest, falls es nicht existiert?

Zunächst stellen wir fest, dass der Algorithmus in endlicher Zeit terminiert. ll, h und m sind ganzzahlige Indizes. m liegt abgerundet in der Mitte zwischen l und h. Beide rekursiven Aufrufe stellen sicher, dass die durchsuchte Teilsequenz echt kürzer wird. Damit ist sie irgendwann leer, was durch den Rekursionsanfang aufgefangen wird.

Wegen der Sortierung muss sich das gesuchte Element immer in der durchsuchten Teilsequenz befinden, falls es existiert. Die einzige Möglichkeit, es zu übersehen, wäre, dass es nicht getestet wird und sich beim nächsten rekursiven Aufruf außerhalb der neuen Teilsequenz befindet. Das ist jedoch ausgeschlossen: Das Element m wird mit dem gesuchten verglichen, und das Element daneben gehört zur anschließend durchsuchten Teilsequenz.

Somit haben wir die Korrektheit des `bisectionSearch`-Algorithmus nachgewiesen.

Binärsuche: Laufzeitanalyse [Slide 10]

```
Algorithm bisectionSearch(S, k, l, h):
  if l > h then
    return false
  m ← ⌊(l + h)/2⌋
  e ← S[m]
  if k = e then
    return true
  else if k < e then
    return bisectionSearch
             (S, k, l, m − 1)
  else
    return bisectionSearch
             (S, k, m + 1, h)
```

- Alle Anweisungen sind $O(1)$, bis auf die rekursiven Aufrufe. Die Laufzeit ist also proportional zur Anzahl der rekursiven Aufrufe.
- Bei jedem rekursiven Aufruf wird die Anzahl der verbleibenden Kandidaten halbiert. Nach i Aufrufen verbleiben also noch maximal $n/2^i$ Kandidaten.
- Die Anzahl der rekursiven Aufrufe ist also maximal gleich der kleinsten ganzen Zahl m mit $n/2^m < 1$, also $\log n < m$, also $m = \lfloor \log n \rfloor + 1$.

  Sobald kein Kandidat mehr verbleibt, ist $n/2^m < 1$, und die Rekursion endet, d.h., m wird nicht weiter erhöht.

Die Laufzeit von `bisectionSearch()` ist also $O(\log n)$.

> Analysieren wir nun das asymptotische Laufzeitverhalten unseres `bisectionSearch`-Algorithmus.
>
> Alle Anweisungen laufen in konstanter Zeit, bis auf die rekursiven Aufrufe. Die Laufzeit ist also proportional zur Anzahl der rekursiven Aufrufe.
>
> Um diese zu ermitteln, stellen wir uns wieder die Frage, wie der rekursive Aufrufbaum aussieht. Die Funktion ruft sich maximal einmal rekursiv auf. Das heißt, unser Aufrufbaum ist wieder eine lineare Liste, und die Gesmatlaufzeit ist proportional zur Länge dieser Liste, also zur Rekursionstiefe. Was ist also die Rekursionstiefe?
>
> Bei jedem rekursiven Aufruf wird die Anzahl der ursprünglich n Kandidaten halbiert. Nach i Aufrufen verbleiben also noch maximal $n/2^i$ Kandidaten. Sobald diese Zahl $n/2^i$ unter 1 fällt, ist die Teilsequenz leer, und die Rekursion endet. Dies ist der Fall, wenn $n < 2^i$ ist, also $\log n < i$. Die Rekursionstiefe ist also logarithmisch in n, der Länge der Eingabesequenz. Damit ist die Laufzeit von `bisectionSearch` $O(\log n)$.
>
> Wie eingangs erwähnt ist diese besser-als-lineare Laufzeit erreichbar, weil wir dank der Sortierung viele Elemente der Sequenz bei der Suche komplett übersprigen können.

3 Verzeichnisbaum

Verzeichnisbaum [Slide 11]

```python
from pathlib import Path

def fstree(file, depth):
    print(' ' * depth * 2 + file.name)
    if file.is_dir():
        for f in file.iterdir():
            fstree(f, depth + 1)

fstree(Path('root'), 0)
```

```
root
  afile
  adir
    subdir
      deepf
    file
  bdir
    fileA
    fileB
```

Anmerkung

Dies ist ein Beispiel für ***mehrfache Rekursion*** *(multiple recursion)*: `fstree()` ruft sich selbst mehrfach auf.

> Sie sind alle mit der hierarchischen Verzeichnisstruktur Ihres Rechners vertraut. Diese Struktur lässt sich als Baumstruktur darstellen: Die Verzeichnisse und Dateien, die sich in einem gegebenen Verzeichnis befinden, sind dessen Kinder in der Baumstruktur. Diese Baumstruktur lässt sich durch eingerückte Zeilen darstellen; auch dies kennen Sie sicher von Ihrem Rechner, in grafisch aufgehübschter Form.
>
> Im Beispiel rechts enthält das Verzeichnis `root` die Datei `afile` und die Verzeichnisse `adir` and `bdir`, das Verzeichnis `adir` enthält das Verzeichnis `subdir` und die Datei `file`, usw.
>
> Diese eingerückte Darstellung wurde durch den links gezeigten Python-Code erzeugt. Falls Ihnen die Programmiersprache Python nicht geläufig ist, macht das überhaupt nichts. Ignorieren Sie einfach die erste Zeile mit der `from-import`-Deklaration und lesen Sie den Rest wie Pseudocode.
>
> Die Funktion `fstree`, kurz für das englische *file system tree*, empfängt zwei Argumente, den aktuellen Verzeichniseintrag und die aktuelle Verschachtelungstiefe, und gibt den Verzeichniseintrag entsprechend eingerückt aus. Dann testet sie mit der Methode `is_dir`, ob es sich beim aktuellen Verzeichniseintrag um ein Verzeichnis handelt, englisch *directory*. Falls ja, iteriert sie über alle darin enthaltenen Verzeichniseinträge und ruft für jeden sich selbst rekursiv auf.
>
> Damit handelt es sich um ***mehrfache Rekursion***: Ein gegebener Aufruf von `fstree` ruft sich selbst *mehrmals* rekursiv auf.

Speicherbedarf des Verzeichnisbaums [Slide 12]

```python
from pathlib import Path

def fstree(file, depth):
    size = file.stat().st_size
    print(' ' * depth * 2 +
          f"{file.name} ({size} Bytes)")
    if file.is_dir():
        for f in file.iterdir():
            size += fstree(f, depth + 1)
    return size

total = fstree(Path('root'), 0)
print(f"Total: {total} Bytes")
```

```
root (26 Bytes)
  afile (15 Bytes)
  adir (20 Bytes)
    subdir (16 Bytes)
      deepf (13 Bytes)
    file (9 Bytes)
  bdir (20 Bytes)
    fileA (1 Bytes)
    fileB (1 Bytes)
Total: 121 Bytes
```

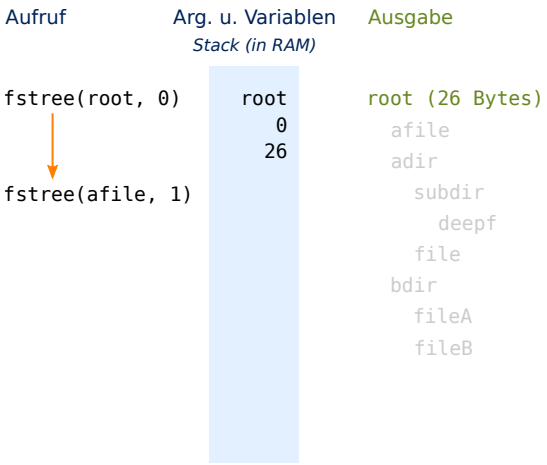
> Um die Verarbeitung von Rückgabewerten bei einer Mehrfachrekursion zu illustrieren, haben wir hier unsere Funktion `fstree` um die Berechnung des Speicherbedarfs der Dateien und Verzeichnisse erweitert. Für den aktuellen Verzeichniseintrag wird der Speicherbedarf ermittelt, in der Variablen `size` gespeichert, und zusammen mit dem Verzeichniseintrag ausgegeben. Falls es sich um ein Verzeichnis handelt, wird rekursiv der Speicherbedarf der darin enthaltenen Verzeichniseinträge ermittelt und auf `size` aufaddiert. Damit enthält `size` die Gesamtgröße des aktuellen Verzeichnisunterbaums, und wird an den Aufrufer zurückgegeben.

Traversierung des Verzeichnisbaums [Slide 13]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



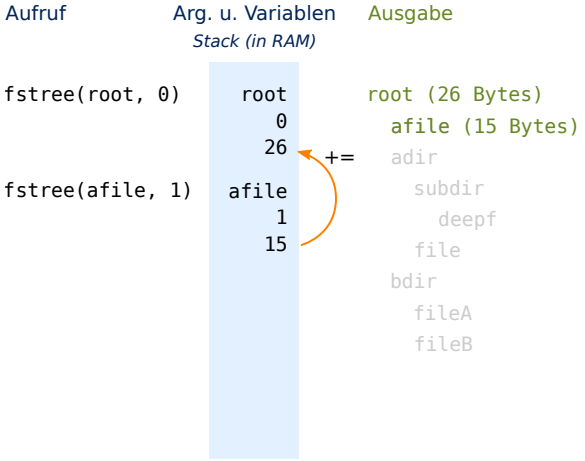
> Schauen wir uns an, wie `fstree` den gezeigten Verzeichnisbaum abläuft.
>
> Zu Beginn wird `fstree` mit dem Wurzelverzeichnis namens `root` und der Verschachtelungstiefe 0 aufgerufen. Diese beiden Werte werden im lokalen Speicherbereich dieses Aufrufs abgelegt, zusammen mit dem Wert 26 der Variablen `s`, hier verkürzt für `size`.
>
> Da es sich bei `root` um ein Verzeichnis handelt, ruft sich `fstree` anschließend mit dem ersten Verzeichniseintrag in `root` auf, nämlich `afile`, unter Angabe einer um 1 erhöhten Verschachtelungstiefe.

Traversierung des Verzeichnisbaums [Slide 14]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



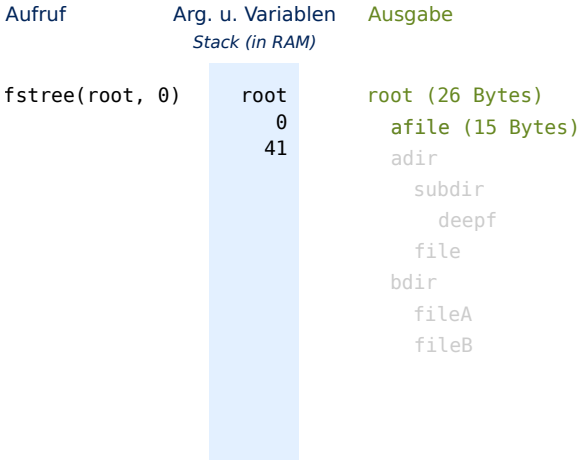
> `fstree(afile, 1)` gibt Dateiname und Dateigröße aus. Da es sich nicht um ein Verzeichnis handelt, liefert es nun die Dateigröße von `afile` zurück.

Traversierung des Verzeichnisbaums [Slide 15]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



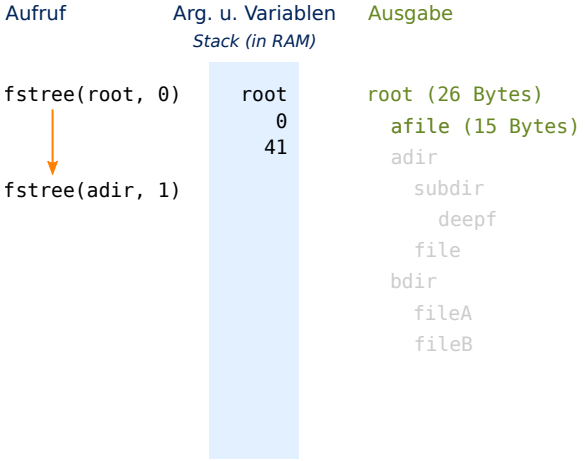
> Der Aufrufer, `fstree(root, 0)`, empfängt diesen Rückgabewert, und addiert ihn zu `s`. Damit ist der erste Eintrag im Verzeichnis `root` abgearbeitet.

Traversierung des Verzeichnisbaums [Slide 16]

```
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



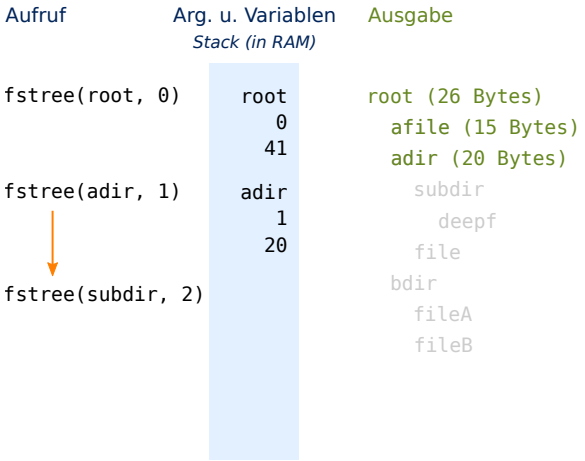
- Es folgt der Aufruf für den zweiten Eintrag, `adir`.

Traversierung des Verzeichnisbaums [Slide 17]

```
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



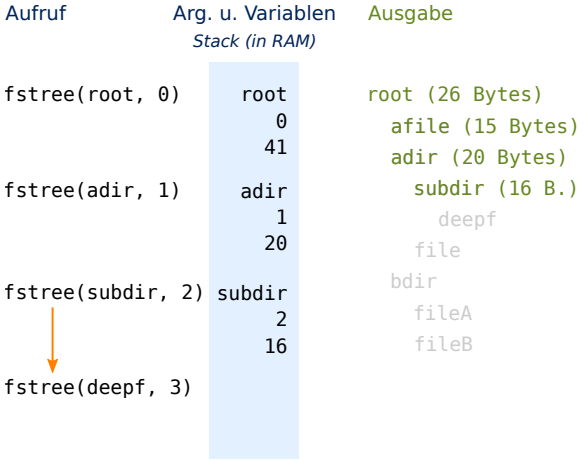
> Der Aufruf `fstree(adir, 1)` gibt wiederum den Verzeichnisnamen und seine Größe aus. Da es sich bei `adir` um ein Verzeichnis handelt, wird `fstree` nun rekursiv mit dem ersten Eintrag dieses Verzeichnisses augerufen.

Traversierung des Verzeichnisbaums [Slide 18]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



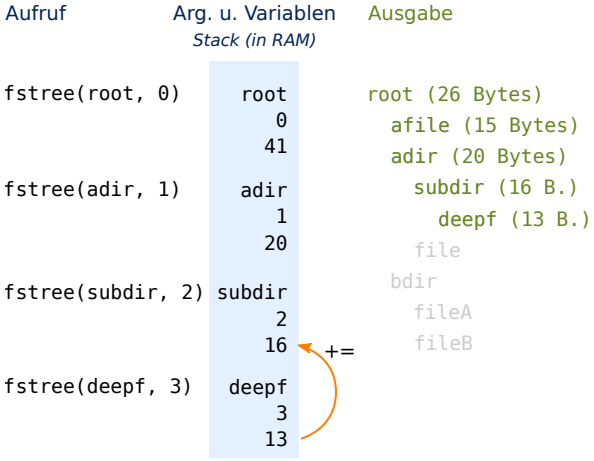
> Dieser Eintrag, `subdir`, ist ebenfalls ein Verzeichnis. Daher folgt ein weiterer rekursiver Aufruf, `fstree(deepf, 3)`.

Traversierung des Verzeichnisbaums [Slide 19]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



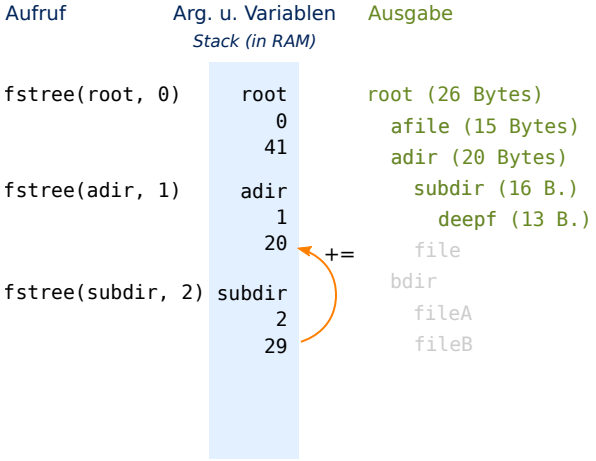
> Da es sich bei `deepf` um eine Datei handelt und nicht um ein Verzeichnis, kehrt dieser Aufruf nun zurück, mit der Dateigröße als Rückgabewert.

Traversierung des Verzeichnisbaums [Slide 20]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



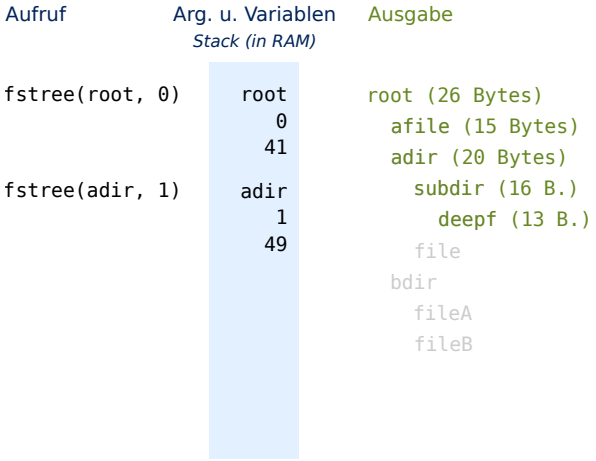
> Der Aufrufer, `fstree(subdir, 2)`, addiert diesen Rückgabewert zur Größe seines Verzeichnisunterbaums hinzu. Da das Verzeichnis `subdir` neben `deepf` keine weiteren Einträge enthält, kehrt `fstree(subdir, 2)` nun ebenfalls zurück.

Traversierung des Verzeichnisbaums [Slide 21]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



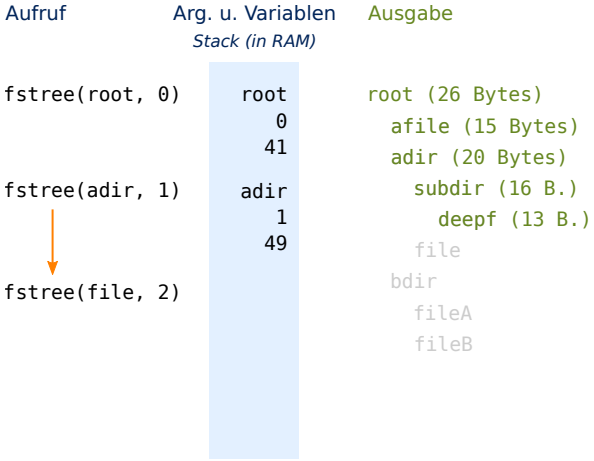
> Der Aufrufer, `fstree(adir, 1)`, addiert den Rückgabewert von `fstree(subdir, 2)` zur Größe seines Verzeichnisunterbaums dazu.

Traversierung des Verzeichnisbaums [Slide 22]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



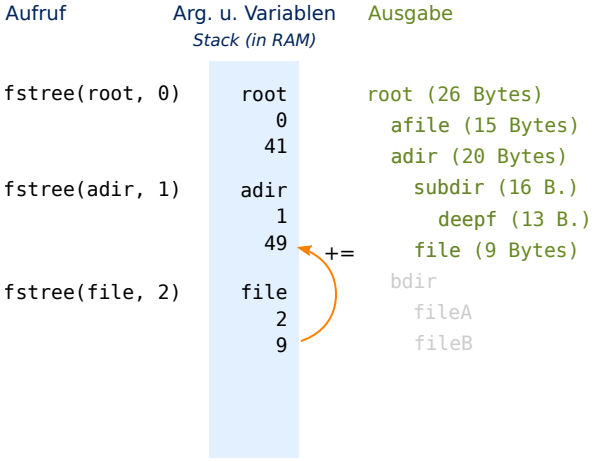
Anschließend ruft er sich ein zweites Mal rekursiv auf, diesmal mit dem zweiten Verzeichniseintrag von `adir`, nämlich `file`.

Traversierung des Verzeichnisbaums [Slide 23]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



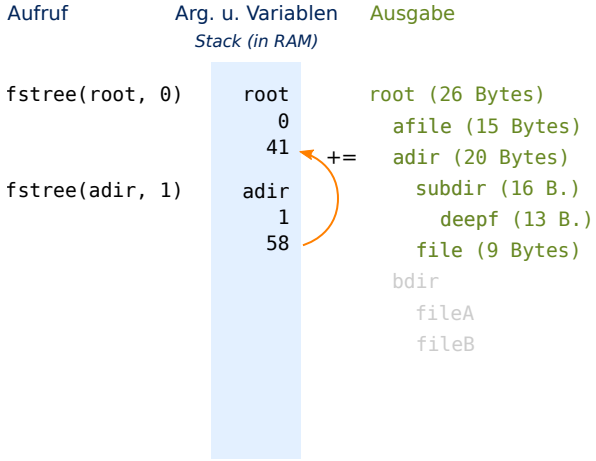
Dieser Aufruf gibt die Dateigröße von `file` zurück.

Traversierung des Verzeichnisbaums [Slide 24]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



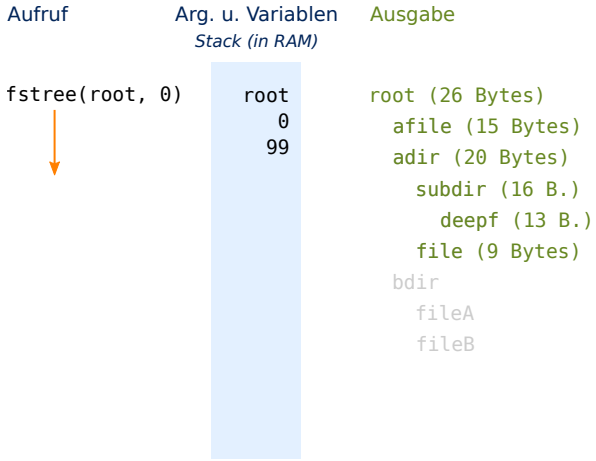
> `fstree(adir, 1)` hat nun seine sämtlichen Verzeichniseinträge abgearbeitet und kehrt nun seinerseits zurück.

Traversierung des Verzeichnisbaums [Slide 25]

```python
from pathlib import Path

def fstree(file, depth):
  s = file.stat().st_size
  print("...")
  if file.is_dir():
    for f in file.iterdir():
      s += fstree(f, depth+1)
  return s

size = fstree(Path('root'), 0)
print(f"Total: {size} Bytes")
```



> `fstree(root, 0)` addiert den Rückgabewert von `fstree(adir, 1)` zur Größe seines Verzeichnisunterbaums hinzu.
>
> Es folgt der nächste rekursive Aufruf, nämlich `fstree(bdir, 1)`. Wir brechen allerdings an dieser Stelle ab, denn es dürfte klar geworden sein, wie die rekursive Aufrufstruktur von `fstree` der rekursiven Struktur des Verzeichnisbaums folgt und wie die Rückgabewerte der einzelnen Aufrufe verarbeitet werden.

`fstree()`: Laufzeitanalyse (1) [Slide 26]

```
def fstree(file, depth):
    size = file.stat().st_size
    print(' ' * depth * 2 +
          f"{file.name} ({size} Bytes)")
    if file.is_dir():
        for f in file.iterdir():
            size += fstree(f, depth + 1)
    return size
```

Was ist die Laufzeit in Abhängigkeit von n, der Anzahl der Verzeichniseinträge (Verzeichnisse + Dateien)?

- Bei jedem Aufruf von `fstree()` kann die `for`-Schleife maximal $n-1$ Mal iterieren.
- Die Aufruftiefe ist maximal n.
- `fstree()` ist also $O(n^n)$.

Formal korrekt, aber nicht *eng*.

> Wie können wir das asymptotische Laufzeitverhalten von `fstree` als Funktion von n bestimmen, der Gesamtzahl der Dateien und Verzeichnisse?
>
> Gehen wir zunächst davon aus, dass alle Anweisungen bis auf `for` und den rekursiven Aufruf in konstanter Zeit ablaufen.
>
> Als ersten Versuch stellen wir nun fest, dass in einem gegebenen Aufruf die `for`-Schleife nicht mehr als $n-1$ Mal iterieren kann. Zweitens ist klar, dass die Aufruftiefe n nicht übersteigen kann.
>
> Gehen wir in beiden Fällen vom Maximalwert aus, dann enthält jeder rekursive Aufruf n Iterationen, bis zu einer Rekursionstiefe von n. Damit haben wir eine Gesamtlaufzeit von $O(n^n)$.
>
> Diese Analyse ist zwar formal korrekt, aber sie überschätzt die maximale Gesamtlaufzeit gewaltig. Da wir insgesamt nur n Verzeichniseinträge haben, kann die maximale Iterationszahl und die maximale Rekursionstiefe nicht gleichzeitig erreicht werden.

`fstree()`: Laufzeitanalyse (2) [Slide 27]

Was ist die Laufzeit in Abhängigkeit von n, der Anzahl der Verzeichniseinträge (Verzeichnisse + Dateien)?

```
root
  afile
  adir
    subdir
      deepf
    file
  bdir
    fileA
    fileB
```

- `fstree()` wird für jedes Element des Verzeichnisbaums exakt einmal (rekursiv) aufgerufen.
- Dies gilt *über alle rekursiven Aufrufe hinweg*. Nicht *alle* Schleifen können $O(n)$ Iterationen durchlaufen, sondern die *Gesamtzahl aller* Iterationen ist $O(n)$.

  Der Aufrufbaum ist zum Dateibaum isomorph!

  Der Aufrufbaum zeigt alle Aufrufe von (Kind-)Funktionen durch (Eltern-)Funktionen. Er muss hier nicht extra gezeichnet werden, denn er entspricht exakt der eingerückten Darstellung rechts.
- `fstree()` ist also $O(n)$.

Anmerkung

Dies ist ein Beispiel für ***amortisierte Laufzeitanalyse*** (siehe später):

Eine Analyse *über mehrere Iterationen bzw. rekursive Aufrufe hinweg* ermöglicht die Bestimmung *engerer Schranken* als eine Analyse pro Iteration bzw. Aufruf.

> Für eine enge und relevante Laufzeitanalyse kommt uns wiederum der Aufrufbaum zu Hilfe. Die Struktur der Aufrufe folgt exakt der Struktur des Verzeichnisbaums. Mit anderen Worten, der Aufrufbaum und der Verzeichnisbaum sind zueinander isomorph. Jedem Verzeichniseintrag entspricht genau ein Aufruf von `fstree`. Damit ist die Gesamtlaufzeit $O(n)$.

`fstree()`: Laufzeitanalyse (3) [Slide 28]

Ist die Laufzeit dieser Anweisung wirklich konstant:

```
print(' ' * depth * 2 + f"{file.name} ({size} Bytes)")
```

- Wie könnte Python `' ' * depth` implementieren?
- Wie lange dauert die Ausgabe einer Zeichenkette auf dem Terminal?

Die Laufzeit von `fstree()` ist $O(nd)$ für eine maximale Verschachtelungstiefe von d.

> Bisher haben wir angenommen, dass alle Anweisungen bis auf `for` und den rekursiven Aufruf in konstanter Zeit ablaufen. Wir müssen allerdings davon ausgehen, dass die `print`-Anweisung für die Ausgabe der Einrückung eine Zeit proportional zu `depth` benötigt. Da diese Ausgabe bei *jedem* rekursiven Aufruf anfällt, multipliziert sich die Laufzeitkomplexität dieser Ausgabe mit der Anzahl der rekursiven Aufrufe. Daher ist die Laufzeitkomplexität von `fstree` insgesamt $O(nd)$, wenn die maximale Verschachtelungstiefe d beträgt.

> Schlimmstenfalls besteht unsere Verzeichnisstruktur aus n ineinander verschachtelten Verzeichnissen. Damit hat `depth` beim ersten rekursiven Aufruf den Wert 1, beim zweiten den Wert 2, usw. bis n. Die Gesamtdauer der `print`-Anweisungen ist also proportional zur Summe der Zahlen von 1 bis n, also $O(n^2)$.

Quiz [Slide 29]

```
from pathlib import Path

def fstree(file, depth):
    print(' ' * depth * 2 + file.name)
    if file.is_dir():
        for f in file.iterdir():
            fstree(f, depth + 1)

fstree(Path('root'), 0)
```

Worin besteht hier der Rekursionsanfang?

A: Mehrfache Rekursion benötigt keinen Rekursionsanfang.

B: Der Aufruf `fstree(Path('root'), 0)` ist der Rekursionsanfang.

C: Der abwesende `else`-Zweig ist der Rekursionsanfang.

D: weiß nicht

4 Kombinationen

Kombinationen [Slide 30]

Alle Zeichenketten der Länge n über einem Alphabet mit k Zeichen:

$n = 2$, $k = 3$

```
AA
AB
AC
BA
BB
BC
CA
CB
CC
```

$n = 3$, $k = 2$

```
AAA
AAB
ABA
ABB
BAA
BAB
BBA
BBB
```

Kombinationen [Slide 31]

Idee: An *jedes Zeichen* des Alphabets hänge *alle Kombinationen* von $n-1$ Zeichen an.

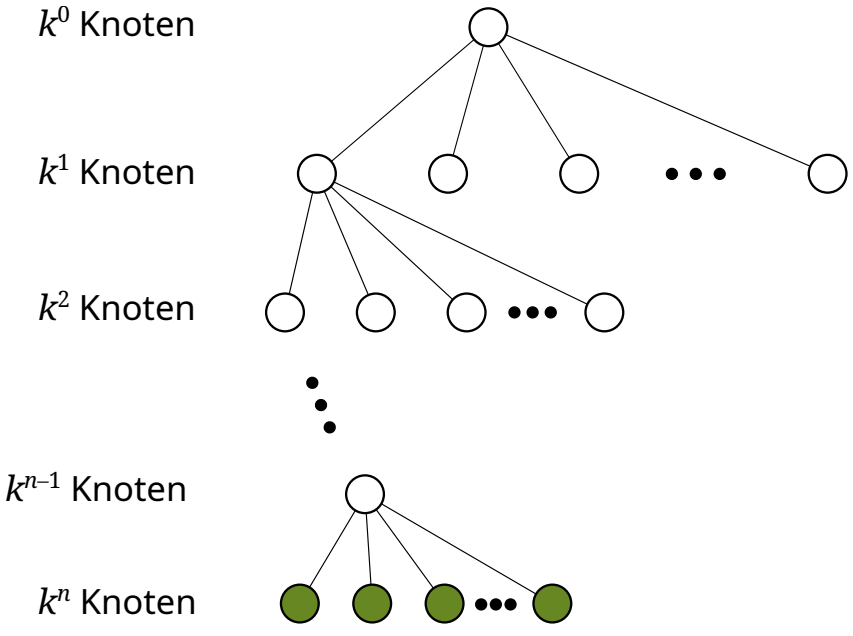
```
Algorithm combinations(p, n):
  Require: String head p (initially empty);
           length n of tail to generate.
  Ensure:  Displays a list of all length−n strings
           under a k−letter alphabet A.
  if n > 0 then
    for i ← 1 to k do
      combinations(p + A[i], n − 1)
  else
    output p
```

Laufzeitanalyse:

- Bei jedem Funktionsaufruf wird die Schleife k Mal durchlaufen.
- Jeder Schleifendurchlauf erzeugt einen rekursiven Aufruf.
- Die rekursive Aufruftiefe ist n.
- Der Aufrufbaum hat also k^n Blätter, plus – da die Zahl der Knoten hier einer geometrischen Reihe entspricht – insgesamt $\frac{k^n-1}{k-1}$ interne Knoten.
- Jedes Blatt gibt einen String der Länge n aus.
- Benötigen alle anderen Anweisungen konstante Zeit, ist die Gesamtlaufzeit also $O(nk^n)$.

Die *Mindestlaufzeit* ergibt sich aus der Ausgabe eines Rechtecks von k^n Zeilen und n Spalten.

Kombinationen: Aufrufbaum [Slide 32]



Endliche geometrische Reihe [Slide 33]

$$
\begin{aligned}
\sum_{i=0}^{n} k^i &= k^0 + k^1 + \cdots + k^n \\
(1-k)\sum_{i=0}^{n} k^i &= (1-k)\left(k^0 + k^1 + \cdots + k^n\right) \\
&= k^0 + k^1 + \cdots + k^n - k^1 - k^2 - \cdots - k^{n+1} \\
&= k^0 - k^{n+1} \\
\sum_{i=0}^{n} k^i &= \frac{k^0 - k^{n+1}}{1-k} \\
&= \frac{k^{n+1} - 1}{k-1}
\end{aligned}
$$

Anmerkung

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

… wie jedem Informatiker durch das binäre Zahlensystem bekannt sein dürfte.

5 Eigenschaften

Konzepte [Slide 34]

- ***Rekursionsanfang***

  Ohne Rekursionsanfang wäre die Rekursion unendlich.

- ***Rekursionsschritt***

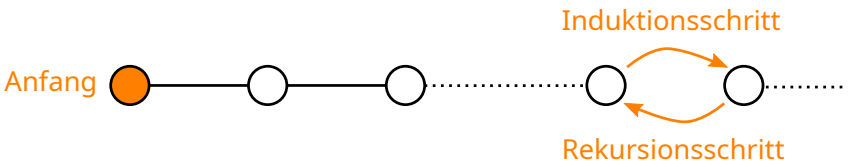
  Auch Sequenzen mehrerer Rekursionsschritte sind möglich (*mehrfache Rekursion*).

Anmerkung

Jeder (rekursive) Aufruf verfügt über seine *eigenen* Parameter und lokalen Variablen.

Rekursion und Induktion [Slide 35]

Beide beruhen auf den gleichen Konzepten, verfahren jedoch in entgegengesetzte Richtungen:



	Anfang	**Schritt**	**Sinn**
Induktion	Von hier geht alles aus.	Zum nächsten Fall, vom *Anfang* weg.	Einfach zu beweisen.
Rekursion	Hierauf wird alles zurückgeführt.	Zum vorigen Fall, zum *Anfang* hin.	Einfach zu berechnen.

Iteration → Rekursion [Slide 36]

Jede Iteration kann als ***Endrekursion*** *(tail recursion)* ausgedrückt werden: Die aufrufende Funktion kehrt unmittelbar nach Rückkehr des rekursiven Aufrufs zurück; ein allfälliger Rückgabewert wird nicht weiter verarbeitet.

```
Algorithm factorial(n):
  Require: Integer n ≥ 0.
  Ensure:  Return n!.
  f ← 1
  for i ← 2 to n do
    f ← fi
  return f
```

```
Algorithm factorial(n):
  Require: Integer n ≥ 0.
  Ensure:  Return n!.
  return factorial-rec(n, 2, 1)

Algorithm factorial-rec(n, i, f):
  Require: Integer n, f ≥ 0;
           integer counter 1 ≤ i ≤ n + 1.
  Ensure:  Return i!.
  if i > n then
    return f
  else
    return factorial-rec(n, i + 1, fi)
```

> ***Endrekursion*** bedeutet, dass ein rekursiver Aufruf das letzte ist, was in einer Funktion passiert. Falls dieser Aufruf einen Rückgabewert liefert, wird dieser unmittelbar zurückgegeben.
>
> Jede Iteration lässt sich als Endrekursion ausdrücken. Jede Iteration der Schleife entspricht einem rekursiven Aufruf, wobei der Zähler, seine obere Schranke, und das Ergebnis der Berechnung als Argumente übergeben werden.
>
> Der tiefste rekursive Aufruf berechnet also das Ergebnis der letzten Iteration. Dieses Ergebnis wird dann als Rückgabewert von allen Aufrufen bis zum ersten Aufruf unverändert durchgereicht.
>
> Sehen wir hier links noch einmal unsere iterative `factorial`-Funktion. Die Laufvariable i läuft von 1 bis n, und das Ergebnis der Berechnung jeder Iteration wird in f gespeichert.
>
> Rechts unten sehen wir die entsprechende Formulierung als Endrekursion. Die Funktion `factorial-rec` übernimmt die drei Variablen n, i und f als Parameter. Als erstes wird die Schleifenbedingung getestet: Falls diese nicht mehr wahr ist, ist das Ergebnis in f bereits vollständig berechnet und wird sofort zurückgegeben.
>
> Andernfalls findet der nächste Berechnungsschritt statt, indem das Teilergebnis f mit der Laufvariablen i multipliziert und die Laufvariable inkrementiert wird. Die Ergebnisse werden als Argumente dem nächsten rekursiven Aufruf übergeben und werden somit dessen Parametern f und i zugewiesen.
>
> Nun benötigen wir lediglich noch eine Funktion, die die rekursive Prozedur mit den korrekten Anfangsparametern startet. Die Funktion `factorial` oben rechts übergibt die Anfangswerte $i = 2$ und $f = 1$, analog zum iterativen Algorithmus links.

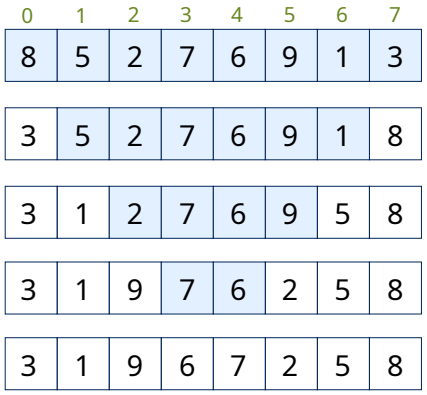
Rekursion → Iteration [Slide 37]

Jede Rekursion kann als Iteration ausgedrückt werden; unmittelbar (und automatisierbar) bei Endrekursion.

Andernfalls verwendet man explizit einen oder mehrere *Stapel (stack)*.

```
Algorithm reverseArray(A, i, j):
  Require: Array A and indices i, j ≥ 0.
  Ensure:  Reversal of elements in A
           from i to j.
  if i < j then
    swap A[i] and A[j]
    reverseArray(A, i + 1, j − 1)

reverseArray(A, 0, 7)
```



> Umgekehrt kann jede Rekursion als Iteration ausgedrückt werden. Im Allgemeinen benötigt man hierfür Hilfsdatenstrukturen; dies wäre z.B. bei unserem `fstree`-Algorithmus der Fall.
>
> Endrekursion kann jedoch unmittelbar und automatisierbar als Iteration ausgedrückt werden, indem der umgekehrte Weg gegangen wird wie zuvor beschrieben. Jede Rekursion hat einen Anfang; dieser wird zur Abbruchbedingung der Schleife. Werte, die sich von einem rekursiven Aufruf zum nächsten ändern, ändern sich entsprechend von einer Schleifeniteration zur nächsten.
>
> In unserem Beispiel invertiert `reverseArray` das Array A, indem es das erste und das letze Element vertauscht und danach sich selbst auf dem verbleibenden Unterarray zwischen diesen beiden Elementen aufruft. Es ist eine einfache Übung, diesen rekursiven Algorithmus in einen iterativen zu überführen.

Formen von Rekursion [Slide 38]

- ***lineare*** Rekursion *(linear recursion)*

  Maximal ein rekursiver Aufruf (s.o.).

- ***mehrfache*** Rekursion *(multiple recursion)*

  Mehrere rekursive Aufrufe (s.o.).

- ***End***rekursion *(tail recursion)*

  Nichts folgt nach dem rekursiven Aufruf (s.o.).

- ***wechselseitige*** Rekursion *(mutual recursion)*

  Mehrere Funktionen rufen sich derart gegenseitig auf, dass von mindestens einer davon mehrere Instanzen gleichzeitig (auf dem Aufrufstapel) aktiv sind (s.u.).

Wechselseitige Rekursion [Slide 39]

```
Algorithm isEven(n):
  Require: Integer n ≥ 0.
  Ensure:  Return true if n even;
           false if n odd.
  if n = 0 then
    return true
  else
    return isOdd(n − 1)
```

```
Algorithm isOdd(n):
  Require: Integer n ≥ 0.
  Ensure:  Return true if n odd;
           false if n even.
  if n = 0 then
    return false
  else
    return isEven(n − 1)
```

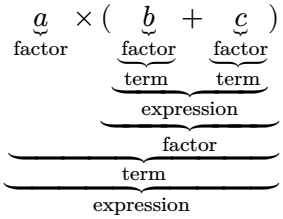
Selbstverständlich würde niemand ernsthaft einen Test auf Geradzahligkeit auf diese Weise implementieren, die in mehrfacher Hinsicht sehr ineffizient ist.

Wechselseitige Rekursion: Praktisches Beispiel [Slide 40]

Rekursiver Abstieg:

- Jedem *Nichtterminalsymbol* (linke Seite jeder *Produktionsregel* der Grammatik) entspricht eine Methode des *Parsers*.
- Diese Methoden rufen sich entsprechend der Produktionsregeln (rechte Seite) gegenseitig auf.

expression → term { + term }
term → factor { × factor }
factor → *ident* | (expression)



Ineffiziente Rekursion: `unique3()` [Slide 41]

```
Algorithm unique3(S, l, h):
  Require: Sequence S, indizes l (low) and h (high).
  Ensure: Return true iff all elements of the subsequence
          S[l, ..., h] are unique.
  if (l ≥ h) return true
  if (not unique3(S, l, h − 1)) return false
  if (not unique3(S, l + 1, h)) return false
  return S[l] ≠ S[h]
```

- Jeder Aufruf löst zwei weitere Aufrufe aus,
- bis zu einer Rekursionstiefe von $n-1$.
- Es finden also $1+2+4+8+\cdots+2^{n-1}=2^n-1$ Aufrufe statt.
- Da alle anderen Anweisungen in konstanter Zeit ablaufen, ist `unique3()` $O(2^n)$.

Ineffiziente Rekursion: `Fibonacci()` [Slide 42]

$F_0=0$, $F_1=1$, $F_k=F_{k-2}+F_{k-1}$

```
Algorithm Fibonacci(k):
  Require: Integer k ≥ 0.
  Ensure: Return the k−th Fibonacci number.
  if k ≤ 1 then
    return k
  return Fibonacci(k − 2) + Fibonacci(k − 1)
```

Die Laufzeitanalyse ist ähnlich wie bei `unique3()`.

Warnung

Rekursive Definitionen können ineffizienten Algorithmen entsprechen!

Fibonacci: effiziente, lineare Rekursion [Slide 43]

```
Algorithm Fibonacci(k):
  Require: Integer k ≥ 0.
  Ensure: Return the pair of Fibonacci numbers (F_k, F_{k−1}).
  if k ≤ 1 then
    return (k, 0)
  else
    (i, j) ←Fibonacci(k − 1)
    return (i + j, i)
```

$O(n)$

6 Zusammenfassung

Zusammenfassung [Slide 44]

- Für viele Probleme existieren rekursive Algorithmen, die deutlich simpler sind als nicht-rekursive Algorithmen.
- *Einfach* (für den Leser des Codes) impliziert jedoch nicht immer *effizient* (Bedarf an Runtime-Ressourcen):
  - Die Laufzeit kann explodieren.
  - Jeder rekursive Aufruf verbraucht Speicherplatz auf dem Aufrufstapel.
  - Funktionsaufrufe sind aufwändiger als Schleifensprünge.
- Rekursion und Induktion beruhen auf denselben Konzepten (*Anfang* und *Schritt*), aber verfahren in entgegengesetzte Richtungen.
- Zur Laufzeitanalyse eines rekursiven Algorithmus ist der Aufrufbaum oft sehr nützlich.
- Jede Iteration kann (einfach) als Endrekursion ausgedrückt werden, und jede Rekursion kann (ggf. unter expliziter Verwendung von Stapeln) als Iteration ausgedrückt werden.