1. Übung zum Kurs
Algorithmische Mathematik II

Averkov, Zeile, Peters
Email: clemens.zeile@ovgu.de, benjamin.peters@ovgu.de

Fakultät für Mathematik
Otto-von-Guericke-Universität Magdeburg
Sommersemester 2016
13. April 2016

Inhalt I

Organisatorisches
- Scheinkriterien
- Sonstiges und Fragen

Unterschiede zwischen C und C++

Präsenzaufgaben

Programmiergrundlagen (Backup)
- Variablen, Datentypen, Operationen
- Prozeduren
- Kontrollstrukturen
- Rekursion

Scheinkriterien

Teil 1:

- voraussichtlich **10** Übungsblätter,
- in denen insgesamt **50 %** der Punkte zu erreichen sind.
- Die Übungsblätter sollen in **2er** Gruppen abgegeben werden.
- Üblicherweise gibt es 4 Aufgaben: eine Programmieraufgabe, drei Beweisaufgaben.
- Alle Aufgaben sind abzugeben. Von den Beweisaufgaben sind bestimmte Aufgaben als **Votieraufgabe** gekennzeichnet. Um für Votieraufgaben Punkte zu erhalten, muss jeder in der Lage sein diese auch **vorzurechnen**.
- Für die **Programmieraufgabe(n)** gelten die Hinweise aus dem ersten Semester, Abgaben **an clemens.zeile@ovgu.de**

Teil 2:

- **Ende Mai** wird eine Aufgabe für ein **Programmierprojekt** bekanntgegeben.
- Dieses soll in **3er** Gruppen bearbeitet werden und
- **Anfang Juli** abgegeben werden.
- Um den Schein zu erhalten (für die Prüfung zugelassen zu werden) muss das Programmierprojekt erfolgreich präsentiert werden.

Sonstiges und Fragen

- Übungsaufgaben werden immer 1 Woche nach Abgabe besprochen
- Donnerstag-Übung am **5. Mai 2016** fällt aus (Christi Himmelfahrt) und wird nachgeholt
- Wann wollt ihr mit der Übung beginnen?
- Andere Fragen?

C und C++

- C++ (erstmals 1983) ist Nachfolger bzw. eine Erweiterung von C, daher häufig „C/C++“
- C++ ist zu C **abwärtskompatibel**, womit sich (fast alle) C-Programme auch mit einem C++-Compiler übersetzen lassen.
- Wichtigste Erweiterung **Objektorientierte Programmierung** (OOP) - auch mit C möglich, doch C++ hilft dem Entwickler dabei.
- Die Erweiterungen, die in C++ existieren, werden mit den gleichen Konstrukten gesteuert (Entscheidungen treffen, Programmteile in Funktionen zusammenfassen und Wiederholungen durch Schleifen), die bereits C eingeführt hat.
- Mit **Standardbibliothek** deutlich komfortabler: Strings anstatt Char, bool als Datentyp (0 oder 1) und viele weitere Optionen
- Dateiendungen: .h und .c für C-Programme, .h/.hpp und .cpp für C++
- Referenzen und Zeiger in C++, nur Zeiger in C

Ausgabe

Dieses C-Programm schreibt „Hallo Welt“ auf den Bildschirm und lässt sich also auf C und C++ gleichermaßen übersetzen, es ist also ein C/C++-Programm.

```
#include <stdio.h>

int main( void )
{
  printf( "Hallo Welt\n" );
  return 0;
}
```

Folgendes Programm ist ein reines C++-Programm, das ebenfalls „Hallo Welt“ auf den Bildschirm schreibt:

```
#include <iostream>

int main( void )
{
  std::cout << "Hallo Welt" << std::endl;
  return 0;
}
```

Ein C-Compiler würde dieses Programm nicht übersetzen können.

OOP: Klassen und Objekte

C++ kennt Klassen und Strukturen, Hauptmerkmale

- Objekte werden durch Klassen beschrieben, sind vom gleichen Typ (z.B. Klasse „Auto", Objekte: verschiedene Modelle)
- Schlüsselwort *class*
- **Attribute** (auch Eigenschaften, Variablen) und **Methoden** (Funktionen) bestimmen Objekt einer Klasse.
- **public**: Zugriff auch außerhalb der Klasse, **private**: nur innerhalb Klasse

OOP: Klassen und Objekte, Beispiel

```
class rennwagen
{
    int Kmh;

  public:
    rennwagen() : Kmh(0)
    {
    }

    void beschleunigen(int kmh)
    {
      Kmh += kmh;
    }

    void bremsen()
    {
      Kmh -= 50;
      if (Kmh < 0)
      {
        Kmh = 0;
      }
    }
};
```

OOP: Klassen und Objekte, Beispiel

```
#include <iostream>
#include "rennwagen.h"

int main()
{
  rennwagen MeinRennwagen;

  MeinRennwagen.beschleunigen(250);
  std::cout << "Rennwagen auf 250 km/h beschleunigt" << std::endl;
  MeinRennwagen.bremsen();
  MeinRennwagen.bremsen();
  std::cout << "Rennwagen auf 150 km/h abgebremst" << std::endl;
  MeinRennwagen.beschleunigen(50);
  std::cout << "Rennwagen auf 200 km/h beschleunigt" << std::endl;
  MeinRennwagen.bremsen();
  MeinRennwagen.bremsen();
  MeinRennwagen.bremsen();
  MeinRennwagen.bremsen();
  std::cout << "Rennwagen angehalten" << std::endl;
}
```

Strukturen

Eine Struktur ist nun ein benutzerdefinierter Datentyp - ein Datentyp, den Sie selbst erstellen

```
#include <string>

struct adresse
{
  std::string Anrede;
  std::string Vorname;
  std::string Nachname;
  std::string Strasse;
  int Hausnummer;
  int Postleitzahl;
  std::string Ort;
  std::string Land;
};

int main()
{
  adresse MeineAdresse;

  MeineAdresse.Anrede = "Herr";
  MeineAdresse.Vorname = "Boris";
  MeineAdresse.Nachname = "Schaeling";
  MeineAdresse.Strasse = "Schlossallee";
  MeineAdresse.Hausnummer = 1;
  MeineAdresse.Postleitzahl = 12345;
  MeineAdresse.Ort = "Entenhausen";
  MeineAdresse.Land = "Deutschland";
}
```

Aufgabe 1

Schreiben Sie ein Programm, das Vorname, Nachname und Alter in eine Struktur (person) einliest. Hinweis: Nutzen Sie zum Einlesen folgende Funktion aus der Standardbibliothek:

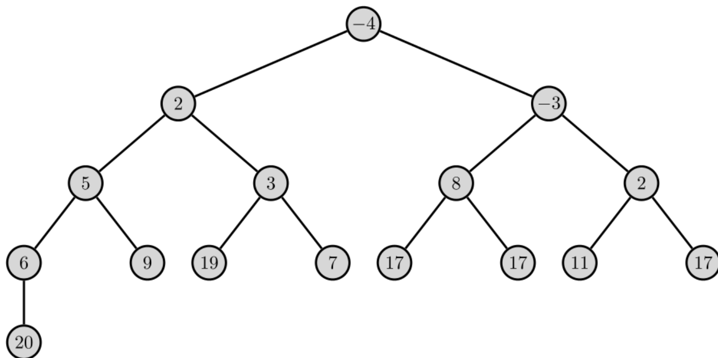
$std :: cin >> variablenname;$

Zudem soll das Programm diese eingelesene Informationen der person-Struktur in einem Satz ausgeben.

1. Schreiben Sie zunächst alles in eine Datei (.cpp)
2. Splitten Sie dann Struktur, Funktionen (.hpp) und main (.cpp in einzelne Dateien auf

Exkurs: Heaps

- Heap (englisch wörtlich: Haufen oder Halde) auf Bäumen basierende abstrakte Datenstruktur
- Objekte oder Elemente können abgelegt und aus diesem wieder entnommen werden
- Elementen ist Schlüssel zugeordnet, der die Priorität der Elemente festlegt
- Verwendung finden Heaps vor allem dort, wo es darauf ankommt, schnell ein Element mit höchster Priorität aus dem Heap zu entnehmen (HIFO-Prinzip), beispielsweise bei Vorrangwarteschlangen.



Aufgabe 2

Erstellen Sie eine Struktur **myHeap** mit den Attributen **data** (ganzzahliges Array), **maxSize** und **size**. Diese soll durch eine Funktion **createHeap**(int maxSize) initialisiert werden. Erstellen Sie eine Funktion **insertHeap** (myHeap (*)heap, int value), die heap mit value befüllt. Lassen Sie sich heap durch eine Funktion **printHeap** ausgeben. Hinweis: Das Attribut data soll als Zeiger initialisiert werden.

Programmiergrundlagen

- Variablen
- Prozeduren
- Parameterübergabe
- Kontrollstrukturen
- Rekursion

Definition 4.1

Imperative Programmierung ist ein Programmierparadigma, in welchem ein Programm als Folge von Anweisungen (Befehlen) dargestellt wird, die im Speicher aufbewahrte Rechengrößen verändern.

Bemerkung

Assemblersprachen und viele moderne höhere Programmiersprachen sind imperativ.

Variable

Definition 4.2

Variable ist ein Behälter für Rechnungsgrößen (*Werte*) die im Verlauf des Rechenprozesses auftreten. Variable hat einen Namen und eine bestimmte Adresse im Speicher. Der Wert der Variable kann sich im Verlauf der Berechnung ändern. Der *Typ* (*Datentyp*) einer Variable bestimmt die Menge der möglichen Werte, welche die Variable annehmen kann.

Variable (Fortsetzung)

Variable:

- Variablenname ist Bezeichner für ein Objekt.
- Objekt kann durch die Verwendung des Variablennamens verändert und ausgegeben werden.
- Objekt belegt einen bestimmten Bereich im Speicher (vgl. [**?**, § 4.9.6])
- Jede gegebene Variable darf nur innerhalb eines bestimmten *Gültigkeitsbereiches* (engl. scope of the variable) benutzt werden.

Deklaration von Variablen

Definition 4.3

Deklaration einer Variable ist eine Anweisung, die einen Variablennamen in das Programm einführt.

Definition 4.4

Definition einer Variable ist eine Anweisung, die eine Variable in das Programm einführt.

Einfache Datentypen

Definition 4.5

Ein Datentyp ist *einfach*, falls er nicht in andere Datentypen zerlegbar ist.

z.B.:

- *Numerische Datentypen* (ganze Zahlen, Gleitkommazahlen)
- Zeichen (engl. character) ist druckbares Symbol oder ein Kontrollzeichen (Tabulation, Neue Zeile usw.)
- *Boolescher Typ* ist der Typ, dessen Variablen zwei mögliche Werte annehmen können (die Werte Wahr oder Falsch).
- Zeiger

Zeiger

- Ein *Zeiger* ist eine Variable, deren Wert Adresse eines Objektes ist.
- Zeiger ermöglichen Vernetzung von Objekten.
- Zeiger ermöglichen dynamische Speicherbelegung.
- Zeiger sind nicht in allen höheren Programmiersprachen verfügbar.
- Zeiger ist ein low-level-Konzept (sie existieren in Assemblersprachen)

Zuweisung

Definition 4.6

Zuweisung (Bezeichung $:=$) ist Operation der Form

$$\text{BEZEICHNER} := \text{AUSDRUCK},$$

wobei BEZEICHNER der Name einer Variable ist und AUSDRUCK ein Ausdruck, der einen Wert zurückgibt. Nach der Ausführung der Zuweisung enthält die Variable den Rückgabewert des Ausdruckes.

Definition 4.7

Ein *Ausdruck* ist ein Konstrukt (in einer gegeben Sprache), welches ausgewertet werden kann.

Definition 4.8

In allgemeiner Situation steht der BEZEICHNER für einen sogenannten *lvalue* (einen Ausdruck, der zu einem Objekt im Speicher ausgewertet wird).

Zuweisung: Beispiel

Listing 1: zuweisungen_beispiel.py

```
1  x=12
2  y=2
3  temp=x
4  x=y
5  y=temp
6  x=x+1
7  y=2*y+x
```

Zeile	*x*	*y*	*temp*
1	12	?	?
2	12	2	?
3	12	2	12
4	2	2	12
5	2	12	12
6	3	12	12
7	3	27	12

Bemerkungen zu Zuweisungen

- Andere Bezeichnungen für Zuweisung: $=$ (C++ und viele weitere Sprachen; inkompatibel mit mathematischen Texten, wo $=$ als Gleichheitzeichen benutzt wird), $\leftarrow$ (theoretische Bücher)

Definition 4.9

Eine Variable V heißt *Struktur* (*Record*) mit *Attributen* $\mathrm{ATTR}_1, \ldots, \mathrm{ATTR}_n$ ($n \in \mathbb{N}$) falls V als Ansammlung von Variablen mit den Namen $\mathrm{ATTR}_1[V], \ldots, \mathrm{ATTR}_n[V]$ aufgefasst werden kann. Die vorigen Variablen heißen Attribute von V.

Bemerkung

In modernen Programmiersprachen sind die Bezeichnungen $V.\mathrm{ATTR}_1, \ldots, V.\mathrm{ATTR}_n$ typisch.

Strukturen: Beispiel

Wir führen eine Variable p ein, die einen Punkt auf der Ebene darstellt, der durch die x- und y-Koordinaten sowie durch seine Farbe gegeben wird.

Pseudocode 4.1 Beispiel zu Strukturen

require: p - Variable mit numerischen Attributen $x[p]$, $y[p]$ und $\text{Farbe}[p] \in \{\text{Weiss}, \text{Schwarz}\}$.

1: $x[p] := 20$
2: $y[p] := 7$
3: $\text{Farbe}[p] := \text{Weiss}$
4: $\text{temp} := x[p]$
5: $x[p] := -y[p]$
6: $y[p] := \text{temp}$

Zusammengesetzte Datentypen: Array ($=$ Feld $\approx$ Liste)

Definition 4.10

Ein *Array* (*Feld*, *Liste*) A ist ein nummerierte Ansammlung von n Variablen, die man durch $A[1]$,...,$A[n]$ bezeichnet. Dabei heißt n die *Länge* von A und $A[1], \ldots, A[n]$ die *Komponenten* von A. Wir werden die Länge durch das Attribut $\text{LÄNGE}[A]$ führen.

Beispiel 4.11

1: $A := [1, 3, 5, 7, 6]$
2: $A[4] := A[1] + A[2] + A[4]$
3: $A[A[2]] := A[5] - A[1]$

Bemerkung

Leere Arrays (Bezeichnung: []) sind möglich.

Bemerkung

$A[i\,..\,j]$ bezeichnet *Teilarray* von A mit Komponenenten $A[i], \ldots, A[j]$. Für $i > j$ gilt $A[i\,..\,j] = []$.

Benutzung von zusammengesetzten Datentypen: Beispiel

- Liste der Hochschulen – ist Array mit Komponenten vom Typ Hochschule
- Hochschule – Struktur mit Attributen:
  - Name – Zeichenkette
  - Standort – Zeichenkette
  - Art der Hochschule – (Fachhochschule/Universität)
  - Wann gegründet – Datum
  - Die Mitarbeiter – Array mit Komponenten vom Typ Mitarbeiter
  - Die Studierenden – Array mit Komponenten vom Typ Studierender
- Studierender – Struktur mit Attributen:
  - Name – Zeichenkette
  - Geburtsdatum – Datum
  - Matrikelnummer – ???
  - Aktuelles Semester – ganze Zahl
  - usw.
- Mitarbeiter – Struktur mit Attributen:
  - Name
  - Geburtsdatum
  - Art des Vertrages
  - usw.

Prozedurale Programmierung

Definition 4.12

Prozedurale Programmierung ist ein Programmierparadigma, in welchem ein Programm als Ansamlung von Prozeduren dargestellt wird (jede Prozedur ist für Lösung eines Teilproblems zuständig).

Prozedur (≈ Unterprogramm ≈ Routine ≈ Funktion)

Pseudocode 4.2 $m = \textsc{hölderMittel}(a, b, p)$

require: a, b, p sind reelle Variablen mit $a, b, p > 0$
ensure m ist das Höldermittel von a, b bzgl. p

1: $m := \frac{1}{2}(a^p + b^p)$
2: $m := m^{1/p}$

- HÖLDERMITTEL - Name der Funktion/Prozedur
- a, b, p sind *formale Eingabeparameter* der Funktion HÖLDERMITTEL
- m ist der *Ausgabeparameter* der Funktion HÖLDERMITTEL
- Die Zeile $m = \textsc{hölderMittel}(a, b, p)$ heißt das *Interface* der Funktion HÖLDERMITTEL
- Hilfsvariablen, die innerhalb einer Funktion eingeführt und benutzt werden, heißen *lokale Variablen* dieser Funktion.

Formale und tatsächliche Parameter

Pseudocode 4.3 $q = \textsc{quadratischesMittel}(x, y)$

require: $x, y > 0$
ensure q ist das quadratische Mittel von x und y
1: $q := \textsc{hölderMittel}(x, y, 2)$

- Die Funktion hölderMittel wird während der Ausführung der Funktion quadratischesMittel aufgerufen. Die Parameter $x, y, 2$, für welche die Funktion hölderMittel benutzt wird, heißen *tatsächliche Parameter* beim Aufruf von hölderMittel

Parameterübergabe

Pseudocode 4.4 VERTAUSCHEN(a, b)

require: a, b – Referenzen auf reelle Variablen, *temp* ist eine reelle Variable innerhalb der Prozedur umtausch
ensure Die Werte von a und b werden vertauscht.

1: $temp := a$
2: $a := b$
3: $b := temp$

Pseudocode 4.5

1: $x := 3, y := 5, z := 7$
2: VERTAUSCHEN(x, y)
3: VERTAUSCHEN(y, z)

- In diesem Beisipel möchten wir, dass sich die Werte von x und y nach der Ausführung der Prozedur VERTAUSCHEN verändern. Deswegen definieren wir Parameter a und b von VERTAUSCHEN als Referenzen. Während der Ausfürhung von VERTAUSCHEN sind die Parameter a bzw. b “Zweitnamen” für x bzw. y. Solche Parameterübergabe heißt *call by reference* (*Übergabe durch die Referenz*). Die Parameterübergabe in HÖLDERMITTEL ist *call by value* (*Überegabe durch den Wert*).
- *temp* ist lokaler Parameter der Funktion VERTAUSCHEN

Vereinbarung zur Parameterübergabe

- Vereinbarung: im Folgenden werden nicht zusammengesetzte Variablen durch den Wert übergeben und zusammengesetzte (Arrays oder Variablen, die Attributen besitzten) durch die Referenz.

Beispiel zu geschachtelten Aufrufen

Funktion A wird aufgerufen (Parameter der Funktion werden generiert)
  In der Zeile X der Funktion A wird die Funktion C aufgerufen
    Die Funktion C startet; ihre Parameter werden generiert
      In der Zeile Y der Funktion C wird die Funktion D aufgerufen
        Die Funktion D startet; ihre Parameter werden generiert
        Die Funktion D terminiert; ihre Parameter werden gelöscht
      Die Funktion C setzt von der Zeile Y fort
    Die Funktion C terminiert; ihre Parameter werden gelöscht
  Die Funktion A wird von der Zeile X fortgesetzt
Funktion A terminiert; ihre Parameter werden gelöscht

Strukturierte Programmierung

Definition 4.13

Strukturierte Programmierung ist ein Programmierparadigma, in welchem ein Programm folgende Kontrollstrukturen benutzt:

- Auswahl (= Verzweigung)
- Wiederholung (= Schleifen = Iterative Kontrollstrukturen)

Kontrollstrukturen: Verzweigung (= Falls-Anweisungen)

Pseudocode 4.6 $m = \textsc{Abstimmung}(a, b, c)$

require: drei Personen stimmen ab, um eine Entscheidung zu treffen; a, b, c sind boolesche Variablen, welche darstellen, ob enstprechende Person dafür oder dagegen ist.

ensure m ist das Resultat der Abstimmung; $m = \textsc{Wahr}$, falls die Mehrheit dafür ist, und $m = \textsc{Falsch}$ sonst.

```
1: if a und b oder a und c oder b und c :
2:     m := WAHR
3: else:
4:     m := FALSCH
5: end
```

- Der Sonst-Block ist nicht obligatorisch.

Kontrollstrukturen: geschachtelte Falls-Anweisungen

Pseudocode 4.7 $p = \textsc{relativePosition}(x, a, b)$

ensure x, a, b sind reelle Werte und $a \leq b$.

require: $p := -1$ falls x links von dem Segment $[a, b]$ liegt, $p = 1$ falls x rechts von dem Segment $[a, b]$ liegt und $p = 0$ falls $x \in [a, b]$

```
1: if x < a :
2:     p := −1
3: else if x ≤ b :
4:     p := 0
5: else:
6:     p := 1
7: end
```

Solange-Schleife

Pseudocode 4.8 $s = \textsc{Summe}(A)$

require: A ist Array mit numerischen Werten
ensure s ist die Summe aller Komponenten von A

1: $i := 1, s := 0$
2: **while** $i \leq \textsc{Länge}[A]$ **:**
3: $\quad s := s + A[i]$
4: $\quad i := i + 1$
5: **end**

Für-Schleife

Pseudocode 4.9 $s = \textsc{Summe}(A)$

1: $s := 0$
2: **for** $i = 1, \ldots, \textsc{Länge}[A]$ **:**
3: $\quad s := s + A[i]$
4: **end**

Weitere Schleifen und Kontrollstrukturen

- Repeat-Until, Do-While, Loop...
- Goto
- return

Definition 4.14

Rekursion ist ein oder mehrere Aufrufe der Prozedur innerhalb von sich selbst.

Rekursion: Einfachstes Beispiel - Fakultät ausrechnen

Pseudocode 4.10 $f = \textsc{Fakultät}(n)$

require: $n \in \mathbb{N}_0$.
ensure $f = n!$.

```
1: if n ≤ 1 :
2:     f := 1
3: else:
4:     f := n · Fakultät(n − 1)
5: end
```

- Entartete Fälle werden abgedeckt.
- Mit jedem weiteren rekursiven Aufruf wird der Eingabeparameter einfacher.
- Wie kann man zeigen, dass die Funktion Fakultät terminiert?

Ausführung von Fakultät

Pseudocode 4.11 Verwendung von Fakultät

1: Fakultät(3) in die Standartausgabe schreiben

- Rekursionsbaum für die Ausführung von Fakultät(3):
- Fakultät(3) $\longrightarrow$ Fakultät(2) $\longrightarrow$ Fakultät(1)

Details der Ausführung von Fakultät(3)

- Fakultät startet (erste Ausführung). Parameter n, f werden generiert; $n := 3$.
- Fakultät startet (zweite Ausführung). (Parameter n, f werden generiert; $n := 2$)
- Fakultät startet (dritte Ausführung) Parameter n, f werden generiert; $n := 1$
- Fakultät terminiert (dritte Ausführung). Wert 1 wird zurückgegeben. Parameter n, f werden gelöscht.
- Fakultät terminiert (zweite Ausführung). Wert 2 wird zurückgegeben. Parameter n, f werden gelöscht.
- Fakultät terminiert (erste Ausführung). Wert 6 wird zurückgegeben. Parameter n, f werden gelöscht.

Bemerkungen zur rekursiven Version der Fakultät

- Für die Ausfürhung von FAKULTÄT(n) brauchen wir $O(n)$ Einheiten im Speicher (ohne es direkt im Programm zu sehen)

Rekursion: Praktisches Beispiel - schnelles Potenzieren; nichtrekursive Version

Pseudocode 4.12 $p = \textsc{PotenzNichtRekursiv}(a, n)$

require: $a > 0, n \in \mathbb{N}_0$
ensure $p = a^n$

```
1: p := 1
2: while n > 0 :
3:     p := p · a
4:     n := n − 1
5: end
```

- Anzahl der Arithmetischen Operationen $\Theta(n^k)$ mit $k =$??.

Rekursion: Praktisches Beispiel - schnelles Potenzieren

Pseudocode 4.13 $p = \textsc{Potenz}(a, n)$

require: $a > 0$, $n \in \mathbb{N}_0$.
ensure $p = a^n$.

1: **if** $n = 0$ **:**
2: $\quad p := 1$
3: **else if** n ist gerade **:**
4: $\quad \text{temp} := \textsc{Potenz}(a, n/2)$
5: $\quad p := \text{temp} \cdot \text{temp}$
6: **else:**
7: $\quad \text{temp} := \textsc{Potenz}(a, \lfloor n/2 \rfloor)$
8: $\quad p := a \cdot \text{temp} \cdot \text{temp}$
9: **end**

- Schneller als direkte iterative Lösung (leichtes Beispiel: n ist Potenz von 2; aber auch für andere Instanzen)
- Iterative variante ist möglich.
- Rekursive Variante ist naheliegend und einfach.

Rekursion: größter gemeinsamer Teiler

Pseudocode 4.14 $g = \text{GGT}(a, b)$

require: $a, b \in \mathbb{Z}$.
ensure g ist der größte gemeinsamer Teiler von a, b.

```
1: a := |a|,  b := |b|
2: if a > b :
3:     VERTAUSCHEN(a, b)
4: end
5: if a = 0 :
6:     g := b
7: else:
8:     g := GGT(a, b − a)
9: end
```

- Sehr ineffizient.
- Wir setzen $\text{GGT}(0, 0) = 0$.

Rekursion: Fibonacci-Zahlen

Definition 4.15

Die Folge der *Fibonacci-Zahlen*: $F_0 = 0$, $F_1 = 1$, und $F_i = F_{i-1} + F_{i-2}$ für $i \geq 2$ mit $i \in \mathbb{N}$.

Pseudocode 4.15 $f = \textsc{Fibonacci}(i)$

require: $i \in \mathbb{N}_0$.
ensure f ist die i-te Fibonacci Zahl.

```
1: if i ≤ 1 :
2:     f := i
3: else:
4:     f := FIBONACCI(i − 1) + FIBONACCI(i − 2)
5: end
```

- Rekursionsbaum für der Ausführung von $\textsc{Fibonacci}(4)$ enthält mehrfache Aufrufe von $\textsc{Fibonacci}(2)$, $\textsc{Fibonacci}(1)$ und $\textsc{Fibonacci}(0)$.
- Daher: nicht effizient.

Rekursion: die Türme von Hanoi

- Gegeben sind drei Stäbe (S= Start, Z = Ziel, H - Hilfstab).
- Auf dem Stab S befinden sich $n \in \mathbb{N}$ gelochte Scheiben verschiedener Größe die von oben nach unten nach der Größe aufsteigend sortiert sind.
- Die obersten Scheiben können von einem Stab verlegt werden (falls dabei die Scheiben auf jedem Stab sortiert bleiben)
- Ziel: Alle Stäbe von S nach Z zu verlegen.

Rekursive lösung für "Türme von Hanoi"

Pseudocode 4.16 LöseTHanoi(n, S, Z, H)

require: $n \in \mathbb{N}$, S, Z, H haben verschiedene Werte.
ensure Die Lösung der Aufgabe "Türme von Hanoi" wird ausgegeben.

```
1: if n = 1 :
2:     Ausgeben: Verlege die Scheibe der Größe n von S nach Z
3: else:
4:     LöseTHanoi(n − 1, S, H, Z)
5:     Ausgeben: Verlege die Scheibe der Größe n von S nach Z
6:     LöseTHanoi(n − 1, H, Z, S)
7: end
```

Weitere Bemerkungen zur Rekursion

- Indirekte rekursive Aufrufe sind möglich (z.B: FUNKTION1 $\longrightarrow$ FUNKTION2 $\longrightarrow$ FUNKTION3 $\longrightarrow$ FUNKTION1).
- Ausgangsbedingung (die den entarteten Fall behandelt) ist entscheidend für die Terminierung.
- *Rekursionstiefe* ist die maximale Anzahl der geschachtelten rekursiven Aufrufe.
- Rekursion ist eine Alternative zu den Schleifen.
- Rechenprobleme, die man mit Arrays und iterativen Strukturen, löst, kann man mit Rekursion ohne Arrays und ohne iterative Strukturen lösen (die Effizienz kann ein Problem sein).
- Manche rekursive Algorithmen haben naheliegende nichtrekursive Analoga (z.B.: schnelles Potenzieren)
- Manche Compiler können erkennen, wo die Rekursion durch Iteration ersetzt werden kann, und generieren dabei entsprechenden iterativen ausführbaren Code.
- Rekursion ist ein natürliches Mittel für manche wichtige Klassen der Probleme (Formale Sprachen, Lexikalische Scanner, Suchalgorithmen auf Graphen)