Einführung in die Informatik

Recursion

Wolfram Burgard
Cyrill Stachniss

Einführung (1)

- Geschirr nach großer Party muss gespült werden.
- Sie laufen unvorsichtigerweise an der Küche vorbei und jemand sagt Ihnen: *Erledigen Sie den Abwasch*.
- Was tun Sie?
- Sie sind **faul** und deshalb
  - spülen Sie ein einziges Teil und
  - suchen dann die nächste Person, um ihr/ihm zu sagen, dass sie den Abwasch erledigen soll.
- Vielleicht ist die Person, der Sie die Aufgabe übergeben haben genauso faul und verhält sich nach dem gleichen Muster wie Sie und ebenso alle nachfolgenden.

Einführung (2)

- Diese Methode ist sehr angenehm, denn so muss keiner mehr als ein Teil spülen.
- Der Letzte muss gar nichts mehr machen, er sieht nur ein leeres Spülbecken.
- Demnach können wir den faulen (lazy) Ansatz von **Erledige den Abwasch** folgendermaßen definieren:
  - Wenn das Spülbecken leer ist, ist nichts zu tun.
  - Wenn es nicht leer ist, dann
    - spüle ein Teil und
    - finde die nächste Person und sage ihr/ihm: „**Erledige den Abwasch**.“

Bemerkungen

- Der faule Ansatz enthält einen **Aufruf zu sich selbst**.
- Um sicherzugehen, dass dieser Ansatz nicht zu einem endlosen Weiterleiten führt, muss **jede Person einen Teil** der Arbeit tun.
- Weil der Job somit für die nächste Person ein wenig kleiner wird, ist er **irgendwann ganz erledigt** und die Kette, dem Nächsten zu sagen, den Rest zu tun, kann gebrochen werden.
- Eine Prozedur dieser Art, die einen Teil der Aufgabe selbst löst und dann den Rest erledigt, indem sie sich selbst aufruft, wird **rekursive Prozedur** genannt.
- Rekursion ist ein **einfaches** und **mächtiges** Prinzip, mit dem viele **schwierige** Probleme gehandhabt werden können.

Beispiel: Potenzierung mithilfe von Rekursion

- Um das Prinzip näher zu verstehen, beginnen wir mit der Potenzierung, ein einfaches Problem, welches wir bereits mit Iteration gelöst haben und für das Rekursion nicht zwingend notwendig ist.
- Wir suchen eine Funktion, die zwei Integer-Parameter x und y übergeben bekommt und einen Integer-Wert, nämlich x^y, zurückgibt.
- Der Prototyp einer solchen Methode ist also

```
private int power(int x, int y)
```

- Die Definition der Potenzierung ist

$$x^y = 1 * \underbrace{x * \ldots * x}_{y \text{ mal}}$$

Potenzierung mithilfe von Rekursion (1)

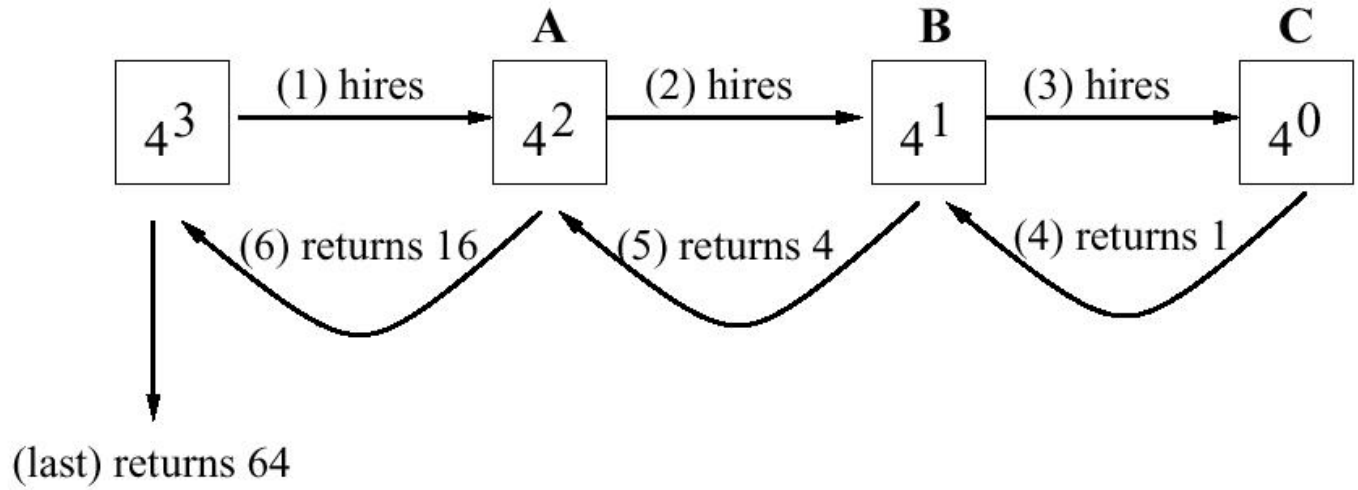
- Um dieses Problem mithilfe von Rekursion zu lösen, stellen wir uns vor, wir müssten die Rechnung von Hand durchführen.
- Wenn y ziemlich groß ist, erscheint die Berechnung von x^y aufwendig zu sein.
- Also nehmen wir den faulen Ansatz und lassen jemand anderen einen Teil der Arbeit tun.
- Wenn wir einen Assistenten hätten, der uns x^{y-1} berechnet, müssten wir das Ergebnis des Assistenten nur noch mit x multiplizieren.
- Natürlich kann der Assistent ebenfalls einen Assistenten haben, der x^{y-2} berechnet usw.

Potenzierung mithilfe von Rekursion (2)

- Wie läuft diese Prozedur also genau ab?
- Wenn wir davon ausgehen, dass der Assistent prüft, ob er den einfachen Fall von x^0 zu berechnen hat, in welchem Fall das Ergebnis 1 ist, kann die Prozedur **Berechne x hoch y** folgendermaßen angegeben werden:
  - Wenn $y = 0$ ist, sind keine Multiplikationen durchzuführen und das Ergebnis ist 1.
  - Wenn $y > 0$ ist, dann
    - Sage einem Assistenten: „**Berechne x hoch y - 1**.“
    - Das Ergebnis ist x-mal das Ergebnis des Assistenten.

Potenzierung mithilfe von Rekursion (3)

- Die Prozedur ist **rekursiv**, weil sie einen Aufruf zu sich selbst enthält und sie ist **gültig**, weil sie
  - einem Assistenten ein **kleineres** Problem zu lösen gibt und weil
  - sie einen **Test** zur Verfügung stellt, um zu prüfen, ob es überhaupt noch Arbeit zu erledigen gibt.G



Potenzierung mithilfe von Rekursion: Java-Code

```
static int power(int x, int y)
    int assistantResult;
    if (y==0)
            // nothing to do, result is 1
        return 1;
    else {
            // tell the assistant to compute x to the (y-1) power
        assistantResult = power(x, y-1);
            // the result is x times the assistant's result
        return x * assistantResult;
    }
}
```

Worauf muss beim Definieren einer rekursiven Methode geachtet werden?

- **Der rekursive Aufruf:** Die Argumente des rekursiven Aufrufs müssen eine Aufgabe darstellen, die **einfacher** zu lösen ist als die Aufgabe, die dem Aufrufer übergeben wurde.
- **Terminierung:** Bei jedem Aufruf einer rekursiven Methode muss geprüft werden, ob Aufgabe ohne erneute Rekursion gelöst werden kann.
  - Der Terminierungs-Code muss **vor** rekursivem Aufruf stehen!
  - Andernfalls würde **nie** Terminierung erreicht und immer wieder der
    rekursive Aufruf gestartet werden.

Wie designed man eine rekursive Methode ?

1. Wie kann das gegebene Problem verkleinert/vereinfacht werden?
   - Dies erledigt der **rekursive Aufruf**.
   - Oft kann Faulheit hierbei eine Inspiration sein.
2. Wann ist das Problem klein/einfach genug, dass es direkt gelöst werden kann?
   - Dies erledigt der **Terminierungscode**.
   - Ebenfalls kann hierbei Faulheit eine Inspiration sein. Frage: Was ist der einfachste Fall des zu lösenden Problems?

Einlesen von Daten, um eine Sammlung von `Employee`-Objekten zu erzeugen

Prototyp der Methode:

```
private void getEmployees(BufferedReader br, Vector v)
```

1. Wir sind faul und lesen immer nur **ein** Objekt, das wir `v` hinzufügen:

   ```
   Employee e = Employer.read(br);
   v.addElement(e);
   ```

   Dann überlassen wir jemand anderem (dem rekursiven Aufruf) **den Rest** des Einlesens (das Problem ist kleiner geworden, da ein Objekt weniger im `BufferedReader` ist).

   ```
   getEmployees(br, v);
   ```

2. Der **Test auf Terminierung** ist einfach: Wir können aufhören, wenn es keinen weiteren Input gibt, also die `Employee.read`-Methode `null` zurückliefert.

   ```
   if (e==null) return;
   ```

Rekursives Einlesen von Daten - Javacode

```
private void getEmployees (BufferedReader br, Vector v)
    Employee e = Employer.read(br);
        // termination code must come after the attempt to read
        // but BEFORE we add to v or make the recursive call
    if (e==null)
        return;
    v.addElement(e);
    getEmployees(br, v);
}
```

Es gibt zwei Rekursionsmuster

- Wir lösen ein Problem, indem wir **einen kleinen Teil selbst** erledigen und **den rekursiven Aufruf den Rest** machen lassen (Beispiele: Abwasch, Einlesen).

```
method (problem) {
    if (problem is very easy)
        solve it and return;
    solve part of the problem, leaving a smaller problem;
    method (smaller problem);
}
```

- Ein **einfacheres Problem wird rekursivem Aufruf übergeben** und **dessen Ergebnis wird benutzt**, um das ursprüngliche Problem zu lösen (Beispiel: `power`).

```
method (problem) {
    if (problem is very easy)
        return solution to easy problem;
    solution to smaller problem = method(smaller problem);
    solve problem, using solution to smaller problem;
    return solution;
}
```

Speicherverbrauch während des Aufrufs einer rekursiven Methode

Um den Speicherverbrauch einer Rekursionsmethode zu verstehen, schauen wir uns einmal an, was genau passiert:

1. Aufrufer kommt zu der Stelle des rekursiven Aufrufes.
2. Aufrufer übergibt Argumente an Aufgerufenen.
3. Aufgerufener reserviert Speicher für Parameter und lokale Variablen.
4. Programmcode des Aufgerufenen wird ausgeführt.
5. Aufgerufener gibt sein Ergebnis an Aufrufer zurück und gibt Speicher frei.
6. Aufrufer arbeitet seinen Programmcode weiter ab.

Innerhalb der Ausführung des Aufgerufenen kann es natürlich zu weiteren rekursiven Aufrufen kommen.

Activation Records

Das Programm muss sich also für **jeden** rekursiven Aufruf Informationen merken, d.h. es wird Speicher belegt für:

- **die Übergabeparameter**,
- **die lokalen Variablen** und
- **die Stelle im Programm**, bei der nach der Ausführung des rekursiven Aufrufes weitergemacht werden soll.

Ein solcher Block an Informationen heißt **Activation Record**.

Bei **jedem** neuen rekursiven Aufruf wird ein solcher Activation Record erstellt und beim Verlasen des rekursiven Aufrufs wieder gelöscht, d.h. der Speicher wird freigegeben.

Beispiel Activation Records (1)

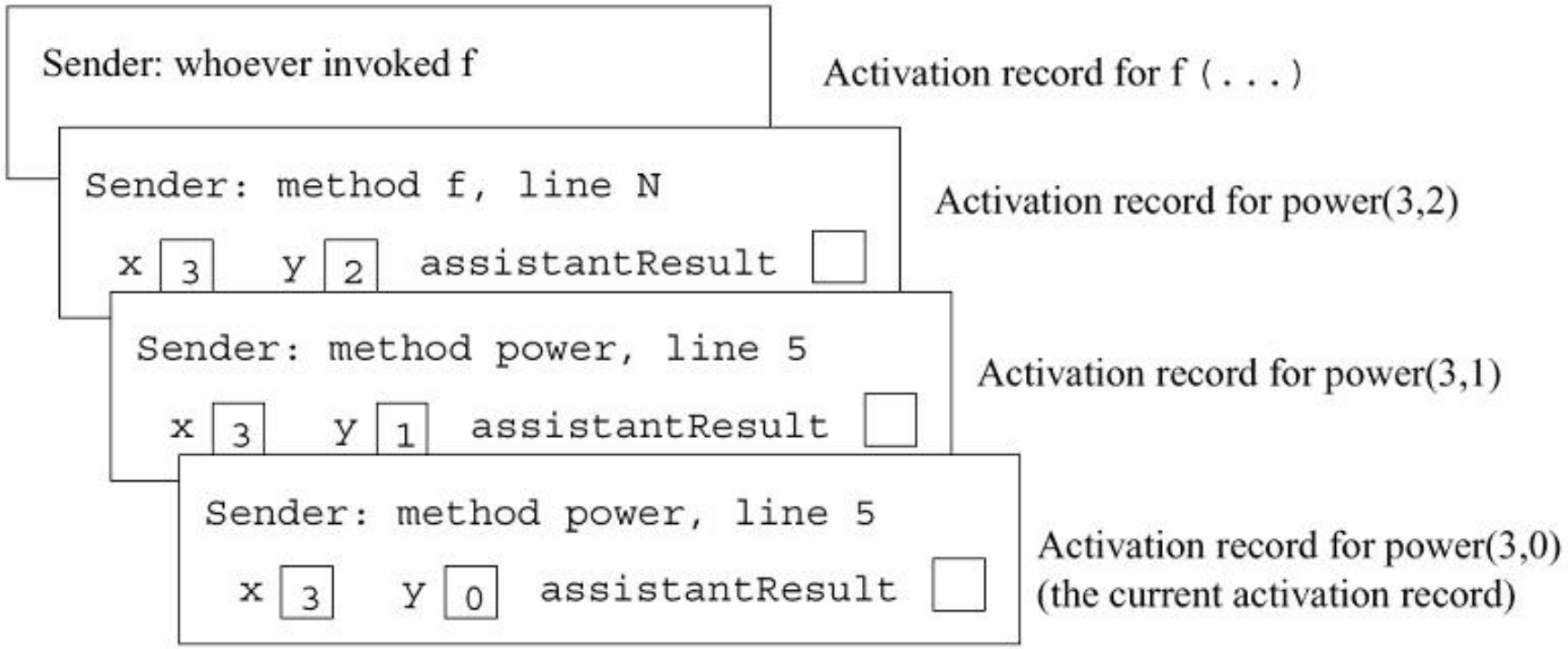
Wir befinden uns in einer Methode `f` und es kommt zum Aufruf der Funktion `power`:

```
...
x = power(3,2); // Zeile z
...
```

- Durch den Aufruf `power(3,2)` wird ein Activation Record (`x=3, y=2, assistantResult`, Rücksprung in Zeile `z`) erstellt und im Speicher gehalten, bis power(3,2) komplett abgearbeitet wurde.
- Da innerhalb von power(3,2) auch power(3,1) und power(3,0) aufgerufen werden, entstehen während der Ausführung auch noch die entsprechenden anderen beiden Activation Records.

Beispiel Activation Records (2)

Nach Aufruf von `power(3,2)` entstehen folgende Activation Records:

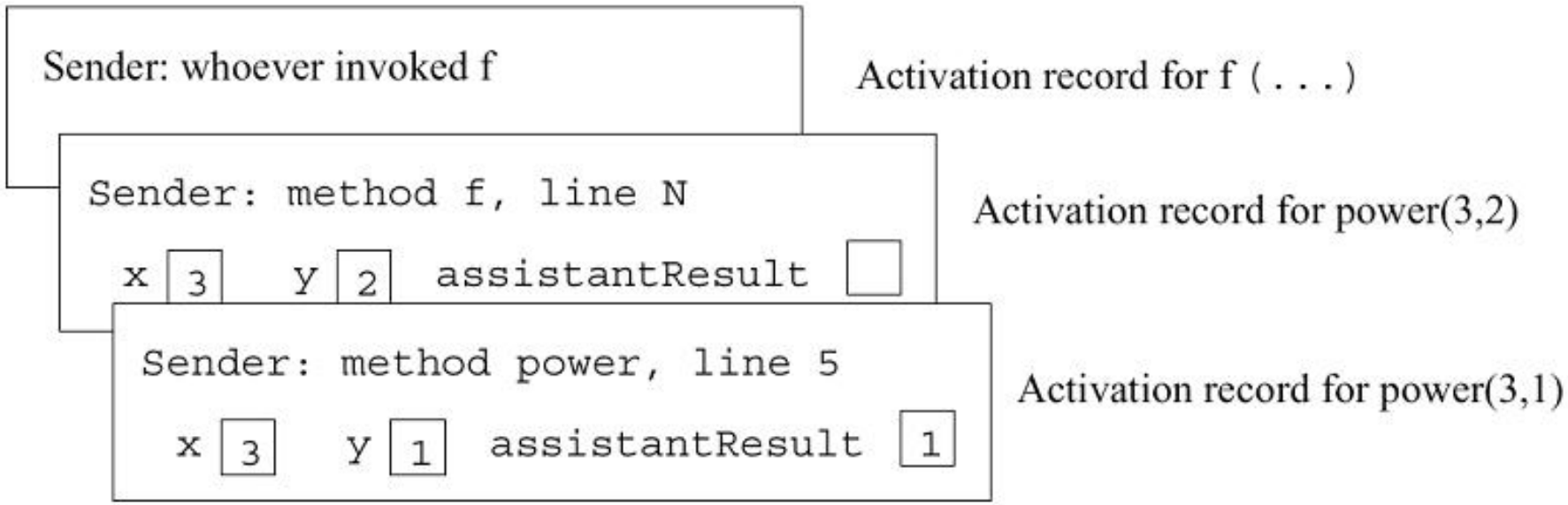


Was bewirkt das `return`-Statement?

- Es wertet den Rückgabewert aus.
- Es zerstört den aktuellen Activation Record.
- Es ersetzt den Aufruf der Methode durch den Rückgabewert.
- Es bewirkt, dass beim Sender mit der Ausführung seines Programmcodes fortgefahren wird.

Beispiel Activation Records (3)

Nachdem `power(3,0)` abgearbeitet wurde (Terminierung), gibt es folgende Activation Records:



Ausgabe der eingelesen Wörter in umgekehrter Reihenfolge

```
import java.io.*;

class ReverseInputRecursive {
    static void reverse(BufferedReader br) throws Exception {
        String s = br.readLine();
        if (s != null){
            reverse(br);
            System.out.println(s);
        }
    }

    public static void main(String arg[]) throws Exception {
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        reverse(br);
    }
}
```

Activation Records für `reverse`

```
static void reverse(BufferedReader br){
     String s = br.readLine();
     if (s != null){
         reverse(br);
         System.out.println(s);
     }
}
```

Speicherverbrauch - Zusammenfassung

- Bei Methoden mit vielen rekursiven Aufrufen entsteht eine **sehr große Menge an Activation Records**, die im Speicher gehalten werden müssen.
- Unter Umständen kommt es dadurch zu einem **Überlauf des Speichers**, d.h. es ist nicht genügend Speicher zum Verwalten der Activation Records vorhanden.
- Aus diesem Grund **limitiert** der zur Verfügung stehende Speicher die Tiefe einer Rekursion!

Rekursion und Iteration (1)

- Rekursion und Iteration basieren beide auf dem Verfahren der **Wiederholung bis hin zu einer Abbruchbedingung**.
- Eine Iteration, z. B. eine `while`-Schleife, kann auf einfache Weise **in eine Rekursion überführt** werden.
- Sei folgende allgemeine `while`-Schleife gegeben:

```
while(condition)
    body;
```

wobei:

- **`v1, ..., vN`** Variablen sind, die in `condition` und `body` auftreten und deren Werte **nach Terminierung der Schleife noch benötigt** werden sowie
- **`p1, ..., pN`** Variablen sind, die in `condition` und `body` auftreten und deren Werte **nach Terminierung der Schleife nicht mehr benötigt** werden.

Überführung der Iteration in eine Rekursion

1. Deklaration von `v1, ..., vN` als Instanzvariablen.
2. Definition einer rekursiven Methode für die `while`-Schleife:

```
private void recursiveWhile(p1, ..., pN) {
    if (!condition)
        return;
    body;
    recursiveWhile(p1, ..., pN);
}
```

3. Ersetzen der ursprünglichen `while`-Schleife durch folgenden Aufruf:

```
recursiveWhile(p1, ..., pN);
```

Berechnung der Summe der Elemente eines Arrays mit einer Iterative Version

```
class SomeClass{
    ...
    public int getSum(int[] x) {
        int n = x.length;
        int i =0;
        int sum = 0;
        while (i!=n) {
            sum += x[i];
            i++;
        }
        // sum == x[0] + x[1] + ... + x[n-1]
        return sum;
    }
    ...
}
```

Berechnung der Summe der Elemente eines Arrays – Überführung in rekursive Version

- Die Variablen `n`, `x` und `i` werden in der Schleife benutzt, aber anschließend nicht mehr benötigt. Also werden sie zu **Parametern der rekursiven Methode**.
- Die Variable `sum` wird nach Terminierung der `while`-Schleife gebraucht. Deshalb wird sie eine **Instanzvariable**.

Berechnung der Summe der Elemente eines Arrays – Eine rekursive Version

```
class SomeClass{
    ...
    public int getSum(int[] x) {
        int n = x.length;
        int i = 0;
        this.sum = 0;
        recComputeSum(x,i,n);                  // recursiveWhile(x,p1,p2)
        return this.sum;
    }
    private void recComputeSum(int[] x,        // recursiveWhile(x,p1,p2)
                               int i, int n) {
        if (!(i!=n))                           // if (!condition)
            return;                            // return;
        this.sum += x[i];                      // body
        i++;                                   // body
        recComputeSum(x, i, n);                // recursiveWhile(p1,p2)
    }
    ...
    private int sum;                           // instance variable
}
```

Wie sieht es mit der Umwandlung von Rekursion in Iteration aus?

- In bestimmten Fällen ist der andere Transformationsweg, also eine Rekursion in eine Iteration zu verwandeln, einfach.
- Prinzipiell ist dies dagegen nicht so einfach oder nur mit größerem Aufwand möglich.

Endrekursion

- In einer Schleife sind die Variablen des Schleifenkörpers nur ein einziges Mal vorhanden — bei jeder Zuweisung werden sie überschrieben.
- Dagegen werden in jedem rekursiven Aufruf mit dem Activation Record neue Variablen erzeugt.
- Ein Aufruf einer rekursiven Methode heißt **endrekursiv**, wenn er die letzte Anweisung der rekursiven Methode ist.
- Funktionen mit **einem einzigen** endrekursiven Aufruf lassen sich **leicht** in Schleifen umwandeln, weil man die lokalen Variablen einfach überschreiben kann, da sie nach der Rückkehr aus dem Methodenaufruf nicht mehr benötigt werden.

Eine endrekursive Version für die Funktion ggT

$$ggT(a,b) = \begin{cases} b & \text{, falls } a \bmod b = 0 \\ ggT(b, a \bmod b), & \text{sonst} \end{cases}$$

Vergleich Iteration/Rekursion

$$ggT(a,b) = \begin{cases} b & \text{, falls } a \bmod b = 0 \\ ggT(b, a \bmod b), & \text{sonst} \end{cases}$$

Iterative Variante:

```
static int ggt(int a, int b){
   int r;
   while (b != 0) {
      r = a % b;
      a = b;
      b = r;
   }
   return a;
}
```

Endrekursive Variante:

```
static int ggt(int a, int b){
   int r = a % b;
   if (r == 0)
      return b;
   return ggt(b, r);
}
```

- Die rekursive Version lehnt sich stärker an die mathematische Definition an.
- Sie ist aber langsamer als die iterative Variante.

Berechnung der Summe der Elemente eines Arrays: Eine endrekursive Version ohne Instanzvariable

```
class SomeClass{
    ...
    private int recComputeSum(int[] x, int i, int sum) {
        if (i == x.length)
            return sum;
        return recComputeSum(x, i+1, sum+x[i]);
    }
    public int getSum(int[] x) {
        return recComputeSum(x, 0, 0);
    }
    ...
}
```

- Anstelle der Instanzvariable verwendet man einen Parameter, in dem man die Zwischensumme **akkumuliert** .
- Ist die Rekursion beendet, gibt man den Wert des **Akkumulators** zurück.

Eine Anwendung

```
private int rCS(int[] x, int i, int sum) {
   if (i == x.length)
       return sum;
   return rCS(x, i+1, sum+x[i]);
}
```

x:

1	2	5	0	7

1. rCS(x, 0, 0)

Geschwindigkeit von Rekursion und Iteration

- Bei Rekursion und Iteration (mittels `while`-Schleifen) handelt es sich um **gleich mächtige Verfahren**. D.h. alle Probleme, die man mit einer Rekursion lösen kann, kann man auch mit einer `while`-Schleife lösen und umgekehrt.
- In den meisten Fällen sind auch beide Verfahren **gleich schnell**.
- Der Nachteil der Rekursion besteht darin, dass, sofern das System nicht entsprechende Optimierungen vornimmt, stets **Activation Records** angelegt werden.
- Dieser zusätzliche Speicherplatz fällt bei vielen iterativen Methoden nicht an (siehe z.B. `power`).
- Bei sehr **tiefen Rekursionen** kann es daher vorkommen, dass eine **iterative Lösung schneller** ist, da die Verwaltung der Activation Records entfällt.

Weshalb benutzt man Rekursionen?

- Mit Hilfe der Rekursion lassen sich einige Probleme sehr viel **eleganter** lösen als mit der Iteration.
- Die **Korrektheit** eines rekursiven Programmes lässt sich häufig wesentlich **einfacher zeigen** als die der iterativen Variante.
- Bei **rekursiven Lösungen** muss man nicht immer alle **Zwischenergebnisse** speichern, da diese automatisch **in den Activation Records abgelegt** werden. Die Verwaltung der zu verarbeitenden Objekte entfällt. Beispielsweise benötigt die rekursive Version des Umdrehens der Reihenfolge der Zeilen im Gegensatz zur iterativen Variante mittels `while`-Schleife kein `Vector`-Objekt.

Wechselseitig rekursive Methoden

Zwei Methoden p und q heißen **wechselseitig rekursiv**, wenn p die Methode q aufruft und q zu einem Aufruf von p führt.

```
class evenOdd {

    static boolean even (int i) {
        if (i == 0)
            return true;
        else
            return odd(i-1);
    }

    static boolean odd (int i) {
        if (i == 0)
            return false;
        else
            return even(i-1);
    }
}
```

Eine bekannte rekursive Funktion: Die Ackermann-Funktion

- Die **Ackermann-Funktion** spielt eine wichtige Bedeutung in der theoretischen Informatik, da sie außerordentlich schnell wächst.
- Die mathematische Definition der Ackermann-Funktion ist:

$$\mathrm{ack}(x,y) = \begin{cases} y+1 & \text{falls } x = 0 \\ \mathrm{ack}(x-1,1) & \text{falls } y = 0 \\ \mathrm{ack}(x-1,\mathrm{ack}(x,y-1)) & \text{sonst} \end{cases}$$

Implementierung der Ackermann-Funktion

```
class ProgramAck {
    static int ack(int x, int y){
        if (x == 0)
            return y+1;
        else if (y == 0)
            return ack(x-1, 1);
        else
            return ack(x-1, ack(x,y-1));
    }

    public static void main(String arg[]) {
        for (int x = 0; x < 5; x++)
            for (int y = 0; y < 5; y++)
                System.out.println("Ack("+x+","+y+")="+ack(x,y));
    }
}
```

Die Werte der Ackermann-Funktion

	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	?	?	?

- Der Wert von `ack(4,2)` kann mit 19729 Dezimalziffern beschrieben werden.
- `ack(4,4)` ist größer $10^{10^{1019000}}$. (Die Anzahl der Elementarteilchen im bekannten Universum liegt etwa bei 10^{70}.)

Berechnung aller Permutationen eines `Vector`-Objektes

Ein weiteres Problem, dass sich nur schwer mit Iteration realisieren lässt, ist die Berechnung aller Permutationen:

```
static void printPermutation(int n, Vector v){
    if (n >= v.size())
        System.out.println(v);
    else{
        printPermutation(n+1, v);
        for (int i = n+1; i < v.size(); i++){
            swap(v, n, i);
            printPermutation(n+1, v);
            swap(v, n, i);
        }
    }
}
```

Beispielanwendung


```
printPermutation(0, v)
```

Zusammenfassung

- **Rekursion** ist ein mächtiges Verfahren zur Definition von Methoden.
- Grundidee der Rekursion ist die **Reduktion eines gegebenen Problems auf ein einfacheres Problem**.
- Bei der Rekursion werden **Activation Records** angelegt, um die notwendigen Informationen (lokale Variablen etc.) zu speichern.
- Dadurch werden **rekursive Varianten manchmal kompakter als iterative Methoden**, allerdings **kostet die Verwaltung der Activation Records Zeit**.