Rekursion und Backtracking

Lernziele:

- Sie kennen das Prinzp von Rekursion und können es an Beispielen anwenden
- Sie kennen das Prinzp des Backtracking und können es an einem Beispiel aufzeigen

1 Rekursion

Rekursion ist ein zentrales Konzept in der Computerwissenschaft, um komplexere Probleme in einfach-überschaubare Problemgebiete zu unterteilen. Eine Definition von Rekursion ist:

Ein Problem wird mit einer Funktion in ein Teil-Problem zerlegt, das wiederum mit der gleichen Funktion weiter zerlegt wird, bis der Rest des Problems gelöst ist.

Somit enthält eine Rekursion endlose Selbst-Aufrufe, und benötigt eine Abbruch-bedingung, damit es nicht zu einer Endlos-Schlaufe kommt.

Die Rekursion teilt somit ein Problem auf (um es somit überschaubarer, d.h. lösbarer, zu machen). Eine Rekursion kann aber auch in einer Datenstruktur abgebildet werden.

1.1 Rekursion in einer Datenstruktur

Wir haben die Rekursion bereits kennengelernt, und zwar als wir eine Datenstruktur brauchten, um verkettete Listen und Stapel zu konstruieren:

```
public class Node{

            //these are private
            private Object item;
            private Node next;

            //constructor
            public Node (Object value){
                  next = null;
                  item = value;
            }
...
```

Die Klasse *Node* besitzt als Eigenschaft wiederum sich selber. Sie verwendet rekursiv die eigene Klasse um auf das nächste Element zu verweisen.

1.2 Rekursion um ein Problem zu lösen

Es gibt unzählige Beispiele für Probleme, die man am einfachsten mittels einer Rekursion löst. Es seien hier nur ein paar Beispiele herausgegriffen:

Ein bekanntes mathematisches Problem ist die Berechnung der Fakultät, hier in Pseudo-Code:

```
berechne(n)
{
  wenn n <= 1
       dann return 1
       sonst return ( n * berechne(n-1) )
}
```

Auch die Fibonacci-Reihe (1,1,2,3,5,8,…) kann mittels Rekursion berechnet werden:

```
public int calculate(int number) {

        if((number == 0) || (number == 1)){
                return number;
        } else {
        return calculate(number -1) + calculate(number-2);
        }
}
```

Übrigens ist obige Methode *calculate* ein Beispiel einer **Mehrfach-Rekursion**, da die Methode gleich mehrmals (hier zweimal) aufgerufen wird.

Rekursion findet sich auch im Bearbeiten einer Baum-Struktur, um die einzelnen Knoten durchzugehen. Wir können z.Bsp. das DOM für XML verwenden und dabei die Kinder in der XML-Struktur rekursiv aufrufen:

```
public void parseRecursive(Node node){

            //check if children exist:
            if (node.hasChildNodes()){

                  NodeList children = node.getChildNodes();
                  //go through children:
                  for (int i=0; i < children.getLength(); i++ ){
                        //get child:
                        Node childNode = children.item(i);
                        System.out.println("Node = " + childNode.getNodeName());
                        parseRecursive(childNode);
                  }

            } else {
                  String nodeName = node.getNodeName();
                  System.out.println("Node = " + nodeName);
            }
      }
```

1.3 Rekursion vs Iteration

Jede Rekursion kann auch in eine Iteration umgewandelt werden. Die Rekursion ist meistens die elegantere Lösung, jedoch mit dem Nachteil, dass sie langsamer ist. Es kommt auf die Performance an, ob es mehr Sinn macht, ein Problem iterativ (also mit einer Schleife) oder mittels einer Rekursion zu lösen.
Gewisse Programmiersprachen verbieten explizit die Iteration und bevorzugen die Rekursion (z.Bsp. LISP oder Prolog).

Beispiel:

Wir wollen Zahlen von n1 bis n2 ausgeben, wobei n1 <= n2 sein soll.
Als Iteration würden wir es so programmieren:

```
public static void printSeries(int n1, int n2){

            for (int i = n1; i < n2; i++){
                  System.out.print(i + ",");
            }
            System.out.print(n2);

}
```

Wir können diese Iteration auch als Rekursion programmieren:

```
public static void printSeries(int n1, int n2){

            //stop recursion:
            if (n1 == n2){
                  System.out.print(n2);
            } else {
                  System.out.print(n1 + ",");
                  printSeries(n1 + 1, n2);  //call method again
            }
}
```

Bei der Rekursion muss die Abbruchbedingung sichergestellt sein, damit es nicht zu unendlichen Aufrufen der Methode kommt.

2 Aufgaben

2.1 Rekursives Durchgehen aller Files oder Elemente

Erstellen Sie eine Klasse, welche mittels der *File* Klasse alle Verzeichnisse auflistet und dabei rekursiv jedes Verzeichnis durchgeht. Implementieren Sie eine Benutzerschnittstelle, so dass der Benutzer nach einem bestimmten File suchen kann.

Alternative: Sie können dasselbe Prinzip auch auf eine XML oder JSON-Struktur anwenden und diese nach einem bestimmten Element durchsuchen.

2.2 Binäre-Suche

Implementieren Sie eine binäre Suche, welche eine Zahl sucht. Dabei wird bei der binären Suche der definierte Suchraum (Anfangspunkt und Endpunkt) jeweils halbiert:

guess = (upperborder + lowerborder) / 2

Mit jedem Schritt wird der Suchraum verkleinert. In Pseudo-code:

if (guess == number)
➔ *Okay, found it!*
else if (guess > number)
number is lower, so we change our upperborder: upperborder = guess -1
else if (guess < number)
number is higher, so we change our lowerborder: lowerborder = guess + 1

Verwenden Sie dabei die Rekursion, um das wiederholte Suchen nach der Zahl durchzuführen.
Geben Sie aus, wieviele Schritte die Suche benötigt.

➔ Zeigen Sie Ihre Lösung der Lehrperson.

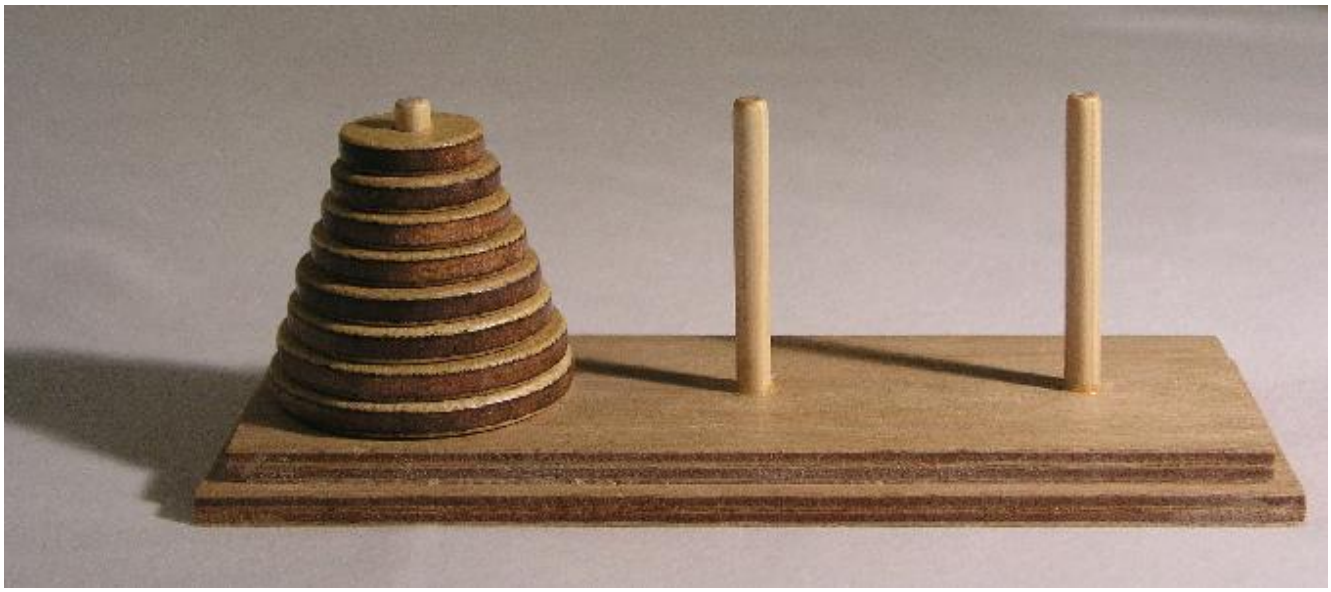
2.3 Für Fortgeschrittene: Die Türme von Hanoi

Es handelt sich hier um ein klassisches Denksport-Problem:
Gegeben sind 3 Türme. Auf einem Turm sind n Platten gelegt.
Die Platten haben unterschiedliche Grössen, mit der grössten Platte zuunterst und der kleinsten Platte zuoberst.

Aufgabe: Die Platten sollen vom 1.Turm zum 3.Turm transportiert werden. Dabei soll das in wenigen Schritten erfolgen.

Regeln:

1. Es darf nur eine Platte aufs Mal transportiert werden (von irgendeinem Turm zu irgendeinem Turm).
2. Grössere Platten dürfen NICHT auf Kleinere gelegt werden.



Implementieren Sie eine Klasse, welches dieses Problem löst. Die Anzahl Platten soll vom Benutzer bestimmt werden.

Beispiel: Transportiere 2 Platten (a und b) von Turm 1 nach Turm 3:

Wir benötigen 3 Schritte, um diese beiden Platten zu bewegen:

Schritt 1: bewege Platte a nach Turm 2
Schritt 2: bewege Platte b nach Turm 3
Schritt 3: bewege Platte a nach Turm 3

Die rekursive Lösung:

Nehmen wir an, dass Platte a aus mehreren Platten besteht.
Somit wären die Schritte wie oben, nur dass Schritt 1 und 3 eine Sammlung von Platten bewegt.
Schritt 1: bewege (n – 1) Platten von Turm 1 nach Turm 2 (n=totale Anzahl Platten)
Schritt 2: bewege die letzte Platte nach Turm 3
Schritt 3: bewege (n – 1) Platten von Turm 2 nach Turm 3

Schritte 1 und 3 sind rekursive Aufrufe derselben Methode. Dieser Methoden-Aufruf sieht wie folgt aus:

```
move(n - 1, start, end, middle);  //Schritt 1

move(n - 1, middle, start, end);  //Schritt 3
```

a) Implementieren Sie eine Klasse mit der besprochenen Lösung. Ihre Klasse soll die entsprechenden Schritte ausgeben.

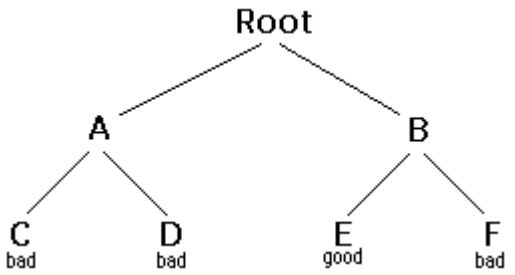
b) Vergleichen Sie die Anzahl Platten mit der Anzahl Schritte. Wie nimmt der Aufwand zu?

Versuchen Sie es selber, bevor Sie nach online-Lösungen suchen! ☺

3 Backtracking

Backtracking ist eine Form von Rekursion, jedoch werden hier verschiedene Lösungswege für ein Problem durchsucht.
Wenn man beim Suchen der Lösung merkt, dass dies der falsche Weg ist, wird der Weg bis zur letzten Entscheidung zurückgegangen und es wird nach einer neuen Lösung gesucht.

Am einfachsten stellt man sich das Suchen nach einer Lösung in Form eines Baumes vor:



1) Man beginnt bei *Root* und hat A oder B zur Auswahl. Man wählt A.
2) Bei A haben wir C oder D zur Auswahl. Man wählt C.
3) C ist schlecht. Gehen wir zurück zu A.
4) Bei A haben wir C schon probiert. Versuchen wir D.
5) D ist schlecht. Gehen wir zurück zu A.
6) Bei A haben wir keine weitere Lösungswege. Gehen wir zurück zu *Root*.
7) Bei *Root* haben wir A schon probiert. Versuchen wir B.
8) Bei B haben E und F zur Auswahl. Versuchen wir E.
9) E ist gut. Wir haben einen Weg gefunden.

Der Baum ist eine abstrakte Struktur, um Lösungswege zu veranschaulichen. Bei den meisten Problemen haben wir keine Baumstruktur, sondern müssen mit anderen Datenstrukturen (z.Bsp. Arrays) arbeiten.

In Pseudo-code sieht ein Backtracking-Algorithmus wie folgt aus:

```
boolean solve(Node n) {
    if n is a leaf node {
        if the leaf is a goal node, return true
        else return false
    } else {
        for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
    }
}
```

Wichtig ist, dass die Backtracking-Funktion als boolean umgesetzt wird. Somit wissen wir, wenn ein bestimmter Knoten (*Node*) *true* ist, dass wir einen gültigen Lösungsweg haben.

Der Algorithmus besteht aus 2 Teilen:

Das Ende eines Lösungsweges, wo wir prüfen, ob wir erfolgreich sind:

```
if the leaf is a goal node, return true
        else return false
```

Und der rekursive Teil, wo wir solange in „Unter-Wegen" Suchen (mit Rekursion) bis eine erfolgreiche Lösung gefunden wurde oder gar keine:

```
for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
```

Quelle und für weitere Beispiele zum Thema Backtracking: http://www.cis.upenn.edu/~matuszek/cit594-2012/Pages/backtracking.html (letzter Aufruf Januar 2018)