Rekursion

CoMa-Übung X

TU Berlin

14.01.2013

Themen der Übung

Themen heute

- Evaluation
- Assertions
- Einlesen von Dateien
- Queues und Breitensuche
- Rekursion
  - Wegrekonstruktion
  - Tiefensuche
  - Backtracking

Evaluation

- Diese Woche bekommt ihr Evaluationsbögen zur CoMa.
- Schreibt, was euch gut gefällt & was euch nicht gefällt!
- Beides ist wichtig.

Assertions

Idee

- Ein/Ausschaltbare Tests im Code einbauen.
- Keine Laufzeitverluste wenn ausgeschaltet.
- Hilfe beim Bugfixing wenn eingeschaltet.

Im Code

- `assert condition : "Fehlermeldung";`
- `condition` ist ein boolscher Ausdruck.
- `"Fehlermeldung"` ist ein `String`, der zusammen mit einem Stack-Trace ausgegeben wird, wenn die Bedingung `false` ist.

Ein- und Ausschalten

- `java -ea ...` zum Einschalten, `java ...` zum Ausschalten.
- Erfordert kein Neukompilieren.

Dateien einlesen – Sudokus

Datei mit gegebenem Datei-Namen einlesen

```
1  public static LinkedList<Sudoku> readSudokusFromFile(
       String filename) throws ParseException {
2    File file = new File(filename);
3    if (!file.exists() || !file.isFile()) {
4      throw new IllegalArgumentException("Es wurde keine
           Datei des Namens " + filename + " gefunden!");
5    }
6    BufferedReader reader = null;
7    LinkedList<Sudoku> result = new LinkedList<Sudoku>();
8    ...
9    return result;
10 }
```

- `File` ist Java-Klasse für Dateien.
- `File(String)` erzeugt ein Datei-Objekt für einen Dateipfad.
- `exists()` und `isFile()` testen, ob wirklich eine Datei existiert.

Dateien einlesen – Sudokus (2)

Datei mit gegebenem Datei-Namen einlesen

```
try {
  reader = new BufferedReader(new FileReader(file));
  String line;
  while ((line = reader.readLine()) != null) {
    ...
  }
}
```

- `FileReader` ist eine Klasse zum Text-Dateien einlesen.
- `BufferedReader` erlaubt, schnell zeilenweise mit anderen Readern zu arbeiten.
- `reader.readLine()` liest die nächste Zeile und gibt sie zurück.
- Ist keine Zeile mehr da, wird `null` zurückgegeben.
- `line` enthält in der Schleife die aktuelle Zeile.

Dateien einlesen – Sudokus (3)

Datei mit gegebenem Datei-Namen einlesen

```
} catch (IOException ex) {
  ex.printStackTrace();
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException ex1) {
      ex1.printStackTrace();
    }
  }
}
```

- `Reader` sollten nach Gebrauch wieder geschlossen werden.
- `close()` macht das.
- `close()` kann eine IO-Exception werfen und braucht daher selbst einen `try-catch`-Block.

Dateien einlesen – Sudokus (4)

Datei mit gegebenem Datei-Namen einlesen

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.text.ParseException;
import java.util.LinkedList;
```

- Die Klassen aus den Beispielen müssen importiert werden.

Queues

Stacks

- Stacks sind Listen, die Elemente hinten einfügen und entnehmen.
- Das letzte Element eines Stacks wird zuerst entnommen.
- LIFO – Last-in-First-out.

Queues

- Queues sind Listen, die Elemente hinten einfügen und **vorne** entnehmen.
- Das erste Element eines Stacks wird zuerst entnommen.
- FIFO – First-in-First-out.

Queues

- Queues und Stacks unterschieden sich nur darin, wo eingefügt und entnommen wird.
- Beides wichtige Datenstrukturen für Algorithmen.

Breitensuche

- Wichtige Anwendung von Queues.
- Traversiert einen Graphen (wie Tiefensuche).
- Berechnet kürzeste Wege in ungewichteten Graphen.

Ideen

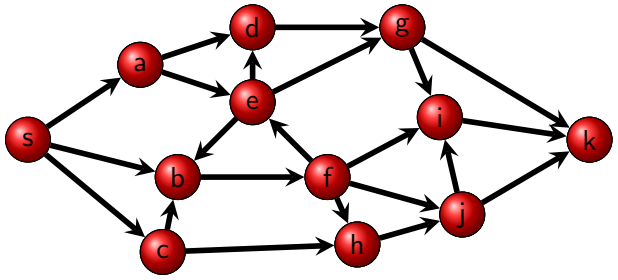
- Beginne bei Startknoten
- Benutze Adjazenzlisten, um neue Knoten zu finden
- Markiere Knoten:
  - Weiß für noch nicht besuchte Knoten
  - Orange für besuchte Knoten, von denen wir weitersuchen können
  - Rot für besuchte Knoten, von denen wir nicht weitersuchen können
- Benutze zwei Queues, um Knoten der aktuellen Phase und Knoten der nächsten Phase zu merken
- Nummeriere die Knoten nach der Phase, in der sie bearbeitet werden ($\rightarrow$ Distanz)

Breitensuche

Basis-Algorithmus mit Startknoten s

- Färbe alle Knoten weiß s a b ...
- Erzeuge zwei leere Queues
- Füge s zur zweiten Queue hinzu
- Solange die zweite Queue nicht leer ist:
  - Verschiebe die Knoten der zweiten Queue in die erste Queue; die nächste Phase beginnt
  - Solange die erste Queue nicht leer ist: Betrachte den ersten Knoten v der ersten Queue
    - Färbe den Knoten rot $\rightarrow$ und gib ihm die aktuelle Phasennummer
    - Weiße Nachbarn werden orange $\rightarrow$ und zur zweiten Queue hinzugefügt (Adjazenzliste durchlaufen)

Breitensuche



1. Queue: k j j h
2. Queue: k i j h

Rekursion

- Kurze Auffrischung durch Fibonacci-Zahlen.
- Effiziente und ineffiziente rekursive Algorithmen.

Fibonacci-Zahlen sind definiert durch:

- $\text{Fib}(0) := 0$, $\text{Fib}(1) := 1$.
- $\text{Fib}(n+1) := \text{Fib}(n) + \text{Fib}(n-1)$ für $n \in \mathbb{N}, n > 1$.

Intuitive rekursive Implementierung:

```
1 public int fibonacciSlow(int n) {
2   switch (n) {
3     case 0: return 0;
4     case 1: return 1;
5     default: return fibonacciSlow(n-1) + fibonacciSlow(n-2);
6   }
7 }
```

Exponentiell viele Methoden-Aufrufe zur Berechnung von $\text{Fib}(n)$.

Fibonacci rekursiv (und effizient)

- Problem: Viele Fibonacci-Zahlen werden mehrfach berechnet.
- Lösung: Zahlen systematisch berechnen.

Rekursive Implementierung

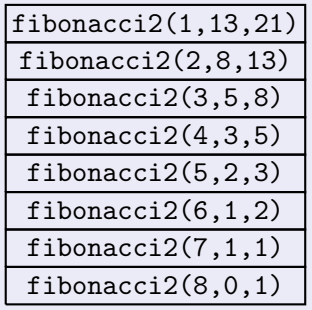
```java
public int fibonacciFast(int n) {
  switch (n) {
    case 0: return 0;
    case 1: return 1;
    default: return fibonacci(n,0,1);
  }
}

public int fibonacci(int n, int zahl1, int zahl2) {
  switch (n) {
    case 0: return zahl1;
    case 1: return zahl2;
    default: return fibonacci(n-1,zahl2,zahl1+zahl2);
  }
}
```

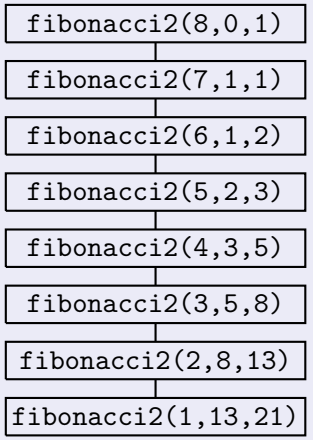
Linear viele Methoden-Aufrufe zur Berechnung von Fib(n).

Fibonacci – Rekursionsbaum & -stack

Rekursionsstack



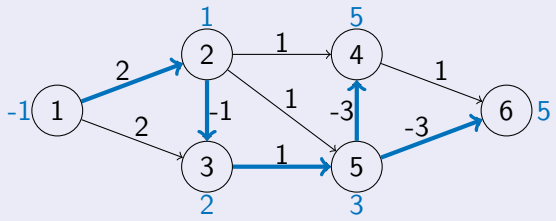
Rekursionsbaum



- Rekursionsbaum ist entartet zur Liste → Iteratives Vorgehen besser (weniger Overhead)

Tree-Matrizen

Wiederholung:



- Ein kürzester 1-6-Weg lässt sich schrittweise aus der Matrix ablesen: $1 \to 2 \to 3 \to 5 \to 6$
- `ShortestPath(1,6)` = `ShortestPath(1,5)` + (5,6)
- `ShortestPath(1,5)` = `ShortestPath(1,3)` + (3,5)
- `ShortestPath(1,3)` = `ShortestPath(1,2)` + (2,3)
- `ShortestPath(1,2)` = (1,2)

Tiefensuche rekursiv

- Tiefensuche kennt ihr bereits – Graph-Traversierung mittels eines Stacks.
- Stack kann durch Rekursionsstack ersetzt werden → rekursive Variante.

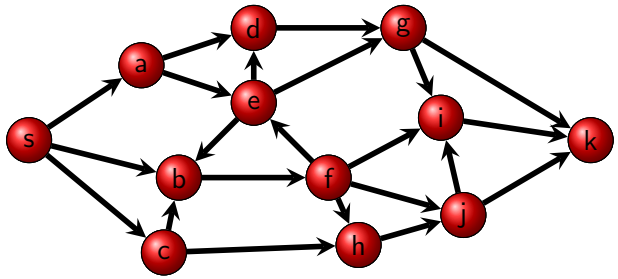
Ideen

- Beginne bei Startknoten
- Benutze Adjazenzlisten, um neue Knoten zu finden
- Markiere Knoten:
  - Weiß für noch nicht besuchte Knoten
  - Orange für besuchte Knoten, von denen wir weitersuchen können
  - Rot für besuchte Knoten, von denen wir nicht weitersuchen können
- **Kein Stack mehr nötig** → Rekursionsstack wird jetzt benutzt
- Nummeriere die Knoten in der Reihenfolge, in der sie gefunden werden

Tiefensuche rekursiv

- Färbe alle Knoten weiß (s) (a) (b) ...
- Rufe Tiefensuche-Methode für den Startknoten s auf
- Tiefensuche-Methode für einen Knoten v:
  - Färbe v orange (v) → (v)
  - Rufe für jeden weißen Nachbarn (w) die Tiefensuche-Methode auf
  - Färbe v rot (v) → (v)

Tiefensuche rekursiv – Beispiel



j
h
f
g
d
a
s

Backtracking

- Wichtiger Problemlösungsansatz
- Basiert auf Trial-and-Error / Versuch-und-Irrtum
- "Probier was aus, wenns nicht klappt, lass es"
- Systematisch angewendet führt das zu Lösungen
- Gut rekursiv zu implementieren (ist eine Art Tiefensuche)

Beispiel

- 4x4-Sudoku:

	1		
			2
		4	
3			

2	1	3	4
4	3	1	2
1	2	4	3
3	4	2	1

Backtracking

- Suche ein leeres Kästchen aus, teste einen möglichen Wert, und fahre rekursiv fort
- Entsteht ein Widerspruch, nimm die letzte Entscheidung zurück
- Ist alles widerspruchsfrei ausgefüllt → fertig

Beispiel

- 4x4-Sudoku:

a_{11}	1		
			2
		4	
3			

Backtracking

Beispiel

- 4x4-Sudoku:

a_{11}	1		
			2
		4	
3			

- 4 Möglichkeiten für das Kästchen oben links:

1	1		
			2
		4	
3			

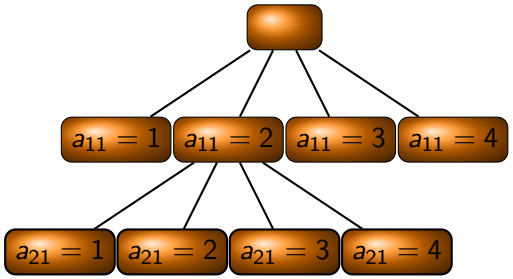
2	1		
			2
		4	
3			

3	1		
			2
		4	
3			

4	1		
			2
		4	
3			

- 1 und 3 ergeben sofort einen Widerspruch
- 4 ergibt nach einigen weiteren Rekursionschritten einen Widerspruch ($\rightarrow$ 2 muss Lösung sein)

Backtracking – Darstellung als Graph



- Graph enthält alle "Ausprobierwege"
- 4^{16} Blätter – alle 4x4-Matrizen mit Einträgen aus $1, \ldots, 4$
- Fast alle davon kein Sudoku $\rightarrow$ finde Sudoku-Blatt, vermeide Sackgassen (Heuristiken, ...)
- Nutzt den Pseudocode von der HP, wenn ihr ein Gerüst sucht.