Algorithmen und Datenstrukturen

Prof. Dr. Thomas Wiemann - FB AI

Hochschule Fulda
University of Applied Sciences

Verifikation

- Durch Testen kann nachgewiesen werden, dass sich ein Programm für endlich viele Eingaben korrekt verhält.
- Durch eine Verifikation kann nachgewiesen werden, dass sich ein Programm für alle Eingaben korrekt verhält
- Bei der Zusicherungsmethode sind zwischen den Statements so genannte Zusicherungen eingestreut, die eine Aussage darstellen über die momentane Beziehung zwischen den Variablen
- Kommentare zwischen den Anweisungen

Beispiel:

```
// i > 0 ∧ z = i²

i = i – 1;

// i ≥ 0 ∧ z = (i + 1)²
```

Zusicherungsmethode - Grundlagen

- Bei Schleifen wird die Zusicherung P, die vor Eintritt und vor Austritt gilt, die ***Schleifeninvariante*** genannt

```
// P
while Q {
    // P ∧ Q
    ...
    ...
    // P
}
// P ∧ ¬Q
```

- Das geschickte Bestimmen von P ist in der Regel der Knackpunkt bei einer formalen Verifikation mit dieser Methode
- Mit dieser Methode kann man partielle Korrektheit nachweisen

Totale Korrektheit = Partielle Korrektheit + Terminierung

Zusicherungsmethode - Beispiel

```
int n, x, y, z;
do n = IO.readInt(); while (n < 0);

x = 0; y = 1; z = 1;

while (z <= n) {

    x = x + 1;          1 2 3 4 ...

    y = y + 2;          1 3 5 7 ...

    z = z + y;          1 4 9 16 ...

}

IO.println (x);
```

Zusicherungsmethode - Beispiel

```
int n, x, y, z;
do n = IO.readInt(); while (n < 0);
// n ≥ 0
x = 0; y = 1; z = 1;
// x² ≤ n ∧ z = (x + 1)² ∧ y = 2x + 1 = Schleifeninvariante P
while (z <= n) /* Q*/ {
    // x² ≤ n ∧ z = (x + 1)² ∧ y = 2x + 1 ∧ z ≤ n
    // (x + 1)² ≤ n ∧ z = (x + 1)² ∧ y = 2x + 1
    x = x + 1;
    // x² ≤ n ∧ z = x² ∧ y = 2x - 1
    y = y + 2;
    // x² ≤ n ∧ z = x² ∧ y = 2x + 1
    z = z + y;
    // x² ≤ n ∧ z = x² + 2x + 1 ∧ y = 2x + 1
    // x² ≤ n ∧ z = (x + 1)² ∧ y = 2x + 1 = Schleifeninvariante P
}
// x² ≤ n ∧ z = (x + 1)² ∧ y = 2x + 1 ∧ z > n = P ∧ ¬Q
// x² ≤ n < (x + 1)²
// x = ⌊√n⌋
IO.println (x);
// Ausgabe: ⌊√n⌋
```

Das Halteproblem

```
public class Fairy {
    public static boolean terminates (char[]s, char[]t) {
        // liefert true, falls das durch die Zeichenkette s dargestellte
        // Java-Programm bei den durch die Zeichenkette t dargestellten
        // Eingabedaten anhaelt;
        // liefert false, sonst
    }
}

import AlgoTools.IO;
public class CheckFairy {
    public static void main(String[] argv) {
        char[] s = IO.readChars();
        if (Fairy.terminates(s,s)) while (true);
    }
}
```

- Sei q der String, der in der Datei `CheckFairy.java` steht.
- Was passiert, wenn das Programm `CheckFairy.class` auf den String q angesetzt wird?

Das Halteproblem

- Sei *q* der String, der in der Datei `CheckFairy.java` steht.
- Was passiert, wenn das Programm `CheckFairy.class` auf den String *q* angesetzt wird?

```
java CheckFairy < CheckFairy.java
```

- 1. Fall: Hält an
  - ➡ `terminates(q,q) == false`
  - ➡ `CheckFairy` angesetzt auf `q` hält nicht an
- 2. Fall: Hält nicht an
  - ➡ `terminates(q,q) == true`
  - ➡ `CheckFairy` angesetzt auf `q` hält an

Also kann es die Methode `terminates` nicht geben!

Programmierparadigmen - Greedy

- Suche lokal die nächste plausibel aussehende Lösung
- Liefert in vielen Fälle gute Lösungen
- Übersichtliche, leicht zu verstehende Verfahren
- Oft aber nicht die perfekten
- Näherungslösung

Rekursion

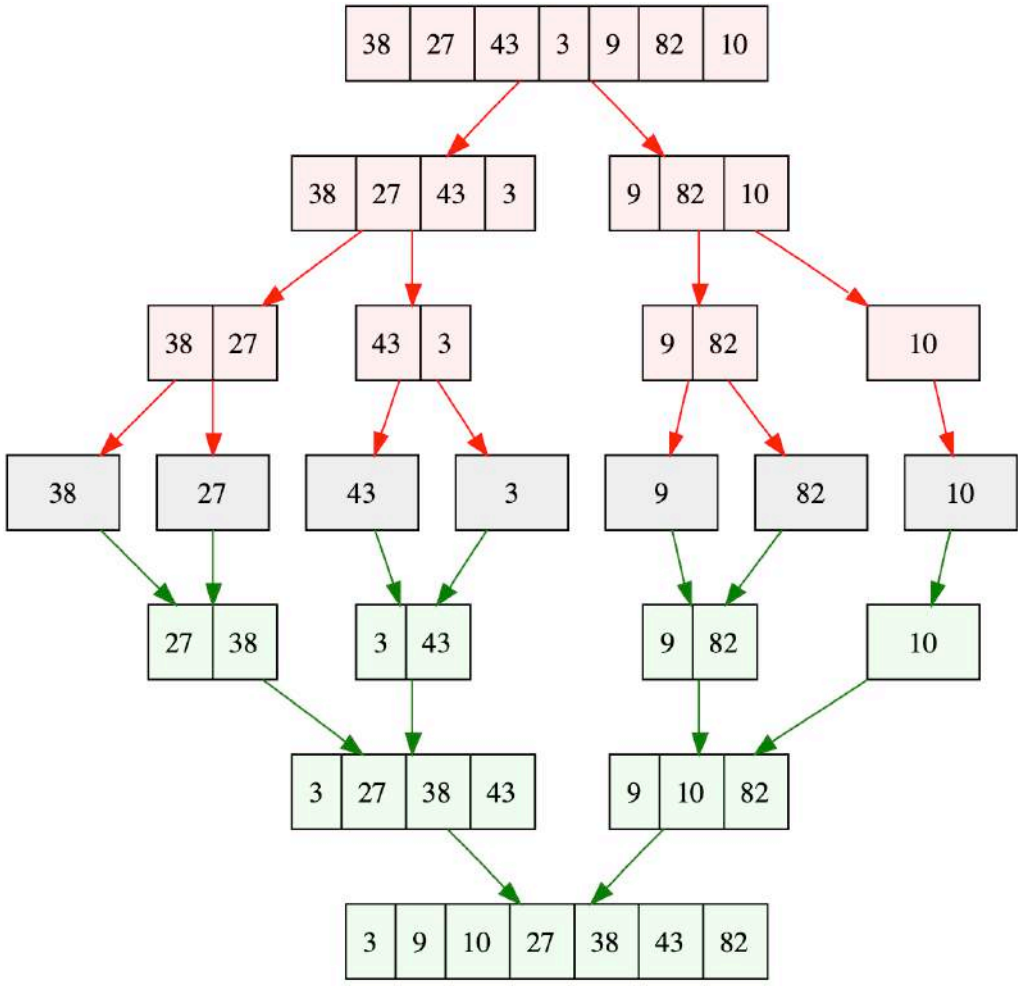
- Zerlege das Problem durch wiederholten Aufruf einer Funktion mit reduziertem Datensatz
- Benötigen Abbruchbedingung
- Beispiele: Fakultät, Fibonacci-Zahlen, …
- Gut Kombinierbar mit dem Divide-and-Conquer Prinzip

Programmierparadigmen - Divide- and Conquer Prinzip

- Programmierparadigma für den Entwurf effizienter Algorithmen
- Ansatz: Unterteile ein in seiner Gesamtheit zu schwierig erscheinendes Problem rekursiv in kleinere Unterprobleme
- Wenn diese beherrschbar sind, löse sie
- Rekonstruiere aus den Teillösungen die Gesamtlösung
- Anwendungen
  - Sortieralgorithmen
  - Parallel Reduction
  - Fast Fourier Transform (FFT)
  - Multiplikation großer Zahlen

Sortieralgorithmen: Mergesort (1)

- Beispiel: Mergesort



Mergesort (2)

```
private int[] mergeSort(int[] elements, int left, int right)
{
    // End of recursion reached?
    if (left == right) return new int[]{elements[left]};

    int middle = left + (right - left) / 2;
    int[] leftArray = mergeSort(elements, left, middle);
    int[] rightArray = mergeSort(elements, middle + 1, right);
    return merge(leftArray, rightArray);
}
```

- Wie implementiere ich den Merge?

Mergesort (3)

```
public int[] merge(int[] leftArray, int[] rightArray) {
    int leftLen = leftArray.length;
    int rightLen = rightArray.length;

    int[] target = new int[leftLen + rightLen];
    int targetPos = 0;
    int leftPos = 0;
    int rightPos = 0;

    // As long as both arrays contain elements...
    while (leftPos < leftLen && rightPos < rightLen) {
      // Which one is smaller?
      int leftValue = leftArray[leftPos];
      int rightValue = rightArray[rightPos];
      if (leftValue <= rightValue) {
        target[targetPos++] = leftValue;
        leftPos++;
      } else {
        target[targetPos++] = rightValue;
        rightPos++;
      }
    }
    // Copy the rest
    while (leftPos < leftLen) {
      target[targetPos++] = leftArray[leftPos++];
    }
    while (rightPos < rightLen) {
      target[targetPos++] = rightArray[rightPos++];
    }
    return target;
  }
```

Simple Merge-Strategie