Tagesprogramm

Effizienz

Binäre Suche

Bubblesort

Wovon Effizienz abhängt

Hardware	unbedeutend
Compiler und Interpreter	unbedeutend
Implementierungsdetails	unbedeutend
Art der Aufgabe (= Problem)	bedeutend, nicht beeinflussbar
Datenmenge (= Problemgröße)	bedeutend, nicht beeinflussbar
Algorithmen und Datenstrukturen	bedeutend, beeinflussbar

Der entscheidende Faktor

Anzahl der Methodenaufrufe oder Datenzugriffe bei gleichen Daten

besonders schlimm:

Anzahl der Zugriffe pro Element kann mit Problemgröße steigen

→ Datenstrukturen und Algorithmen günstig wählen

Manchmal, oft, immer?

Algorithmen und Datenstrukturen können optimiert sein für

besten Fall:

sehr effizient nur wenn man Glück hat, meist ungünstig

durchschnittlichen Fall:

bei vielen Messungen effizient, aber Ausreißer möglich

schlechtesten Fall:

keine Ausreißer, aber durchschnittlich weniger effizient

Aufgabe: Welchen Fall betrachten?

Sollen wir eher auf den durchschnittlichen oder schlechtesten Fall achten?

A Durchschnittlicher Fall, weil meist effizienter.

B Schlechtester Fall, weil keine bösen Ausreißer.

C Beides ist gleich wichtig.

D Weder noch, weil solche Kleinigkeiten egal sind.

Termination

wird nie fertig = Extremfall von Ineffizienz

im Allgemeinen: Termination auch im schlechtesten Fall nötig

nicht durch Testen feststellbar, nur durch Programmanalyse

→ zeigen, dass
- jeder Schritt
- uns näher an Lösung bringt
- um konstanten Mindestbetrag (Multiplikationsfaktor zählt nicht)

Korrektheit vor Effizienz

Grundregeln der Programmoptimierung:

für Nichtexperten: „Lass es bleiben“

für Experten: „Warte damit noch“ (gilt dauerhaft)

kleine Optimierungen sind häufiger Grund für schwere Fehler

nicht von Grundregeln betroffen:

Auswahl geeigneter Algorithmen und Datenstrukteren

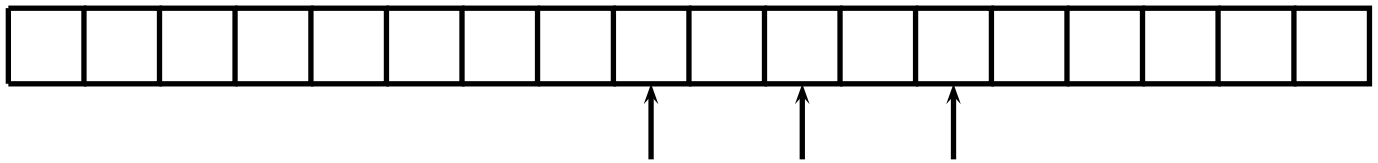
Aufgabe: Optimierungen und Fehler

Suchen Sie in Gruppen zu 2 bis 3 Personen eine Antwort auf folgende Frage:

Wie können kleine Optimierungen zu Fehlern führen?

Zeit: 2 Minuten

Binäre Suche



nötige Suchschritte:

$\approx$ ld(n) da Problemgröße in jedem Schritt halbiert

Termination:

ja da Intervall in jedem Schritt um mindestens 1 kleiner

Aufgabe: Welchen Fall betrachten?

Warum hat die binäre Suche ohne $+1$ bzw. -1 nicht terminiert?

A Weil nicht mehr genau in der Mitte geteilt wird.

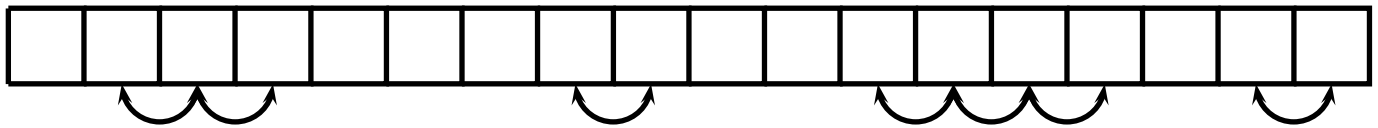
B Weil immer wieder dieselben Schritte durchgeführt werden.

C Weil wir die Abbruchbedingung nicht entsprechend angepasst haben.

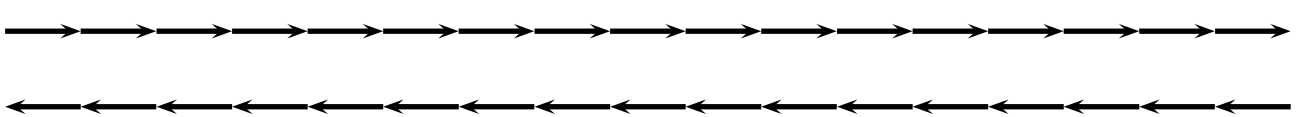
D Weil wir nicht lange genug auf die Termination gewartet haben.

Bubblesort

ein Arraydurchlauf (von links nach rechts):



Grenzfälle:



nötige Suchschritte:

$\approx n^2$ da gesamtes Array bis zu n Mal durchlaufen

Termination:

ja da nach n-tem Arraydurchlauf alle Positionen erreicht

Aufgabe: Bubblesort ohne boolesche Variable

Schreiben Sie Bubblesort so um, dass der Algorithmus nach einer bestimmten Zahl an Arraydurchläufen terminiert (egal ob ein Durchlauf etwas geändert hat).

```
private void sort(int[] a) {
    boolean done;
    do { done = true;
         for (int i = 1; i < a.length; i++) {
             if (a[i] < a[i-1]) {
                 int h = a[i];  a[i] = a[i-1];  a[i-1] = h;
                 done = false;
             }
         }
    } while (! done);
}
```

Welche dieser Varianten von Bubblesort (ursprüngliche oder veränderte) optimiert den durchschnittlichen Fall, welche den schlechtesten?