



Informatik I

Prof. Dr. Christian Pape

Kapitel 13

Optimierung von Implementierungen



Optimierung

- Optimierung
 - Algorithmus bleibt im Kern erhalten
 - Keine Verbesserung innerhalb des O-Kalkül (sonst wird es ein anderer Algorithmus)
 - Verbesserung des konstanten Faktors: 10% schneller werden
 - Für welchen Fall optimieren (schlechtester, bester, durchschnittlicher)?
- Optimierungsmöglichkeiten (unvollständig)
 1. Befehle weglassen
 2. Wiederholte Berechnungen speichern
 3. Befehle ersetzen durch schnellere Befehle
 4. Teile in (schnelleren) Programmiersprache schreiben
 5. Compiler optimieren lassen



Optimierung

1. Befehle weglassen
2. Wiederholte Berechnungen speichern
3. Befehle ersetzen durch schnellere Befehle



Befehle weglassen

- Welche Befehle weglassen?
 - Sehr oft ausgeführte (sonst kein Einspareffekt)
 - Befehle, die lediglich Randbedingungen prüfen (können durch Vorverarbeitung ggf. eliminiert werden)
 - (unnötige Befehle sollten sowieso weggelassen werden)

Beispiel Sortieren durch direktes Einfügen

Optimieren für schlechtesten Fall

```

public void direktesEinfuegen(int a[]) {
    for (int i = 1; i < a.length; i++) {
        int t = a[i];
        for (int j = i; j > 0 && a[j-1] > t; j--) {
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}

```

Ist nur true, wenn t derzeit kleinste Element ist

Im schlimmsten Fall etwa $O(n^2)$ oft ausgeführt



Befehle weglassen

- Wie kann $j > 0$ eingespart werden?
 - Falls $a[0]$ das Minimum ist, dann wird $a[j-1] > t$ spätestens bei $j = 1$ wahr
- Vorverarbeitung
 - Ein Durchlauf, um das Minimum zu suchen
 - Minimum an Stelle 0 verschieben

```
public void direktesEinfuegen(int a[]) {
    // Minimum an Stelle 0 verschieben
    for (int i = 1; i < a.length; i++) {
        int t = a[i];
        for (int j = i; a[j-1] > t; j--) {
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}
```

Zeit Original-Version
 + $O(n)$ Zeit für Minimumsuche
 - $O(n^2)$ weniger $j < 0$

Zeit neue Version

Egal wie viel Zeit die Minimumsuche kostet, Wenn n hinreichend groß ist, Dann ist neue Version zeitlich schneller



Befehle weglassen

- Optimierte Version mit **Wächterelement** `a[0]`
 - Auch anwendbar bei Linearen Listen, siehe Info 2.
 - Gesamtaufwand natürlich unverändert $O(n^2)$

```
public void direktesEinfuegen(int a[]) {
    int minIndex = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[i] < a[minIndex]) {
            minIndex = i;
        }
    }
    tausche(a, i, minIndex);
    for (int i=1; i<a.length; i++) {
        int t = a[i];
        for (int j = i; a[j-1] > t; j--) {
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}
```

Messung bei 1 000 Elemente.

Etwa 5-10% schneller
als Originalversion

Bei 10 000 durch $O(n^2)$
nicht mehr praktikabel
(unabhängig von Optimierung)

Geschwindigkeitsvorteil **unabhängig**
von Programmiersprache

Eigentlich:
Algorithmus (nicht Implementierung)
wurde optimiert



Befehle weglassen

- Befehle weglassen durchführen, falls
 - Aufwand für Weglassen asymptotisch höher ist als Zusatzaufwand für Vorverarbeitung
 - Also: $O(n^2)$ Schritte durch $O(n)$ Vorverarbeitungsschritte einsparen
 - Resultat immer nachmessen: Einsparung oft für Problemgröße gar nicht messbar
 - Resultierende Implementierung immer noch lesbar und verständlich; ansonsten nur bei extremer Verbesserung
- Was weglassen?
 - Häufig durchlaufende Teile auf Randbedingungen und Spezialfälle untersuchen
- Bitte nicht in der Klausur optimieren, es sei denn, es ist in der Aufgabe verlangt



Optimierung

1. Befehle weglassen
2. **Wiederholte Berechnungen speichern**
3. Befehle ersetzen durch schnellere Befehle

Wiederholte Berechnungen speichern

- Wiederholte Berechnungen speichern
 - Berechnungen kosten Zeit
 - Optimierungspotential, falls Zeit für Speichern wiederholte Berechnung aufwiegt

Beispiel Sortieren durch direktes Einfügen

Optimieren für schlechtesten Fall

```
public void direktesEinfuegen(int a[]) {
    for (int i = 1; i < a.length; i++) {
        int t = a[i];
        for (int j = i; j > 0 && a[j-1] > t; j--) {
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}
```

Wiederholte Berechnungen speichern

1. Versuch

- Lokale Variable jm hinzufügen
- Berechnung und Zuweisung in Bedingung

Messung bei 1 000 Elemente.
Keine Verbesserung messbar,
eher etwas schlechter

Vermutlich Zuweisung etwa zwei mal so
teuer wie Berechnung $j-1$ bei Java

Beispiel Sortieren durch direktes Einfügen

Optimieren für schlechtesten Fall

```
public void direktesEinfuegen(int a[]) {
    for (int i=1; i < a.length; i++) {
        int t = a[i];
        for (int j = i, jm; j > 0 && a[jm=j-1] > t; j = jm) {
            a[j] = a[jm];
            a[jm] = t;
        }
    }
}
```

Wiederholte Berechnungen speichern

2. Versuch, in $j-1$ in j „zischenspeichern“

- $j--$ ganz Einsparen, insgesamt weniger $j-1 / j+1$ Berechnungen
- Berechnung $j--$ vorziehen: $a[j-1]$ durch $a[--j]$ ersetzen
- „Fehler“ in Rest des Quelltextes „korrigieren“, indem j durch $j+1$ ersetzt und „ausgerechnet“ wird

Beispiel Sortieren durch direktes Einfügen

Optimieren für schlechtesten Fall

```
public void direktesEinfuegen(int a[]) {
    for (int i=1; i < a.length; i++) {
        int t = a[i];
        for (int j = i; j > 0 && a[--j] > t; ) {
            a[j+1] = a[j]; // war a[j] = a[j-1]
            a[j] = t;      // war a[j-1] = t
        }
    }
}
```

Zwei $j-1/j+1$ echt eingespart

Resultat aber nur geringfügig schneller

Messung bei 1 000 Zahlen:
Ca. 1-3%



Befehle weglassen

- **Resultat**
 - Schwerer zu verstehen
 - Optimierung hängt schon sehr stark von Programmiersprache / Prozessor ab
 - In diesem Beispiel: keine echte Verbesserung
- **Obige Optimierungsform nicht machen**



Optimierung

1. Befehle weglassen
2. Wiederholte Berechnungen speichern
3. **Befehle ersetzen durch schnellere Befehle**

Befehle ersetzen durch schnellere Befehle

- Gleichartige aber schnellere Befehle verwenden:
 - `i++` nicht schneller als `i = i + 1` in Java
 - Ganzzahlige Arithmetik statt Gleitkommazahlen
 - `int` statt `byte`, `short`, `long`
 - Rechtsshift bei positiven Zahlen statt Division durch 2,4,8,16 (`x >> 1`)

Beispiel Binärsuche

```
public boolean binaersuchen(int a[], int z) {
    int links = 0, rechts = a.length-1, m;
    while (links <= rechts) {
        m = (links + rechts) / 2;
        if ( a[m] > z ) {
            rechts = m - 1;
        } else if (a[m] < z){
            links = m + 1;
        } else {
            return true;
        }
    }
    return false;
}
```

Division durch 2 kostet viele Taktzyklen (Division mehrere Zyklen, meist nur eine Verarbeitungseinheit)

`>> 1` kostet weniger als ein Taktzyklus (Ein Maschinenbefehl, mehrere Schieberegister in Prozessor)

Erhoffte Ersparnis:
Mehrere Taktzyklen pro Durchlauf.

Befehle ersetzen durch schnellere Befehle

- Vorsicht bei Verschiebeoperationen
Vorzeichenbit wird mit nach rechts geschoben
-1 >> 1 ist nicht 0
-2 >> 1 ist eine positive Zahl
- links + rechts ist aber **(fast)** immer größer, gleich 0

Beispiel Binärsuche

```
public boolean binaersuchen(int a[], int z) {
    int links = 0, rechts = a.length-1, m;
    while (links <= rechts) {
        m = (links + rechts) >> 1;
        if ( a[m] > z ) {
            rechts = m - 1;
        } else if ( a[m] < z ){
            links = m + 1;
        } else {
            return true;
        }
    }
    return false;
}
```

ca. 30% schneller bei kleinen *und* großen Feldern.

Implementierung entspricht ungefähr der von Sun
Arrays.binarySearch()

Enthält noch einen Fehler!

Was passiert, wenn links und rechts nah an Integer.MAX_VALUE ist?

Befehle ersetzen durch schnellere Befehle

- Aufruf

```
a = new int[Integer.MAX_VALUE];
a[a.length-1] = 1; // Rest ist 0
binaersuchen(a, 1);
```

- links + rechts wird irgendwann negativ!
- Es wird false zurückgegeben

- In der Praxis (derzeit) aber

a = new int[Integer.MAX_VALUE] gibt
OutOfMemoryError

Mehr als 8 GByte Hauptspeicher
benötigt

- Trotzdem ein Bug. Auch bei Mergesort!
- Korrekte Lösung?

```
public boolean binaersuchen(int a[], int z) {
    int links = 0, rechts = a.length-1, m;
    while (links <= rechts) {
        // rechts - links >= 0
        m = ((rechts - links) >> 1) + links;
        if ( a[m] > z ) {
            rechts = m - 1;
        } else if ( a[m] < z){
            links = m + 1;
        } else {
            return true;
        }
    }
    return false;
}
```

Mathematik:

$$\begin{aligned} & (\text{links} + \text{rechts}) / 2 \\ = & (\text{rechts} - \text{links}) / 2 \\ & + \text{links} \end{aligned}$$



Fazit

- Optimierungen
 - Nur Optimieren, wenn es keine besseren Algorithmen gibt ($O(n)$ Algorithmus einem $O(n^*n)$ vorziehen)
 - Nur in Betracht ziehen, wenn es nötig ist.
 - Erst Optimieren, wenn Algorithmus korrekt implementiert und ausreichend getestet ist.
 - Optimieren durch Vorverarbeitung (Algorithmus nicht Implementierung optimieren) anstreben
 - Optimierung verwerfen, wenn keine wesentliche zeitliche Verbesserung erreicht
 - Nur Optimieren, wenn Ergebnis noch lesbar und verständlich ist