DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Algorithmen

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
28. März 2019

Outline

Inhalt

Inhalt

Begriffklärung

Etymologie

- Leitet sich ursprünglich vom persischen Astronomen „Muhammad Ibn-Musa al-Hwarizmi“ ab
  - Schrieb Bücher über das indische Zahlensystem (um 800 n. Chr.)
  - Im 12. Jh übersetzt ins lateinische
  - Dabei wurde der Namensbestandteil „al-Hwarizmi“ in „Algorismi“ lateinisiert
- Durch spätere Überlieferungen wurde der Begriff später als Zusammensetzung betrachtet aus...
  - Dem Namen „Algus-“...
  - und dem aus dem griechisch entlehnten „-rismus“ (Zahl)

Begriffsklärung

Was bedeutet das jetzt

Formale Definition

Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.

Oder auch

Ein Algorithmus ist eine domänenunabhängige Beschreibung einer Handlungsvorschrift zur Lösung eines Problems. Eine bestimmte Eingabe wird in eine bestimmte Ausgabe überführt.

Begriffsklärung

Also

- Ist also die Beschreibung eines Programmes oder einer Funktion
  - Unabhängig von der verwendeten Programmiersprache!
  - Source Code direkt ist also kein Algorithmus...
  - ...aber aus diesem lässt sich der verwendete Algorithmus ableiten und beschreiben
- Algorithmen können in verschiedenen Formen dargestellt werden (Mehr dazu im nächsten Kapitel)

Inhalt

Ziele

- Am Ende des Moduls könnt ihr...
  - Einen Algorithmus in eine Implementierung umsetzen
  - Aus einer Implementierung den Algorithmus ableiten
  - Die formalen Eigenschaften von Algorithmen kennen
  - Algorithmen anhand der kennengelernten Methoden zu analysieren

Inhalt

Inhalt

Eigenschaften von Algorithmen

Grundlegendes

- **Finitheit** - Ein Algorithmus lässt sich in endlch vielen Schritten eindeutig beschreiben
- **Ausführbarkeit** - Jeder Einzelschritt muss tatsächlich ausführbar sein
- **Platzkomplexität** - Ein Algorithmus benötigt zu jedem Zeitpunkt nur endlich viel Speicherplatz
- **Terminierung** - Der Algorithmus benötigt eine endliche Anzahl von Schritten zur Ausführung
- **Determiniertheit** - Der Algorithmus muss bei gleichen Rahmenbedingungen das gleiche Ergebnis liefern
- **Determinismus** - Der nächste Schritt des Algorithmus ist zu jedem Zeitpunkt genau definiert

Effizienz von Algorithmen

- Ergibt sich indirekt aus den Grundlegenden Eigenschaften
- Effizienz lässt sich über verschiedene Größen beschreiben:
  - Speicherverbrauch
  - Zeitverbrauch
- Die sind jedoch oft Implementierungs- und Rechnerabhägig
- Deshalb wird mit formalisierten Modellen gearbeitet
- ...Mehr dazu im Kapitel „Analyse“

Inhalt

Ein kleines Beispiel

Warum wir das überhaupt brauchen

```
1  float Q_rsqrt( float number ){
2    long i;
3    float x2, y;
4    const float threehalfs = 1.5F;
5
6    x2 = number * 0.5F;
7    y  = number;
8    i  = * ( long * ) &y;
9    i  = 0x5f3759df - ( i >> 1 );
10   y  = * ( float * ) &i;
11   y  = y * ( threehalfs - ( x2 * y * y ) );
12 //y  = y * ( threehalfs - ( x2 * y * y ) );
13   return y;
14 }
```

```
1  float Q_rsqrt( float number ) {
2    long i;
3    float x2, y;
4    const float threehalfs = 1.5F;
```

Warum wir das brauchen

- Source Code ist nicht immer verständlich
- Benötigt spezielles Wissen über die Sprache
- Nutzt ggf. Besonderheiten der Sprache aus
- Nutzt teilweise Workarounds (zum Beispiel aus Effizienzgründen)

Möglichkeiten der Darstellung

- Zur Definition von Algorithmen gibt es verschiedenste Möglichkeiten
- Mit ganz eigenen Vor- und Nachteilen
- Wir betrachten im Rahmen der Vorlesung:
  - Prosatext
  - Pseudocode
  - Struktogramme
  - Programmablaufplan (PAP)

Was beschreiben wir?

Unser Referenzalgorithmus

- Um die verschiedenen Elemente zu vergleichen, wollen wir mit allen den folgenden Algorithmus beschreiben:

Referenz

Für eine Zahl n (Wobei gilt: $n \in \mathbb{N}$), soll die Summe aller geraden Zahlen von 0 bis n berechnet werden.

Darstellung als Prosatext

Der simple Weg

- Simpelste Herangehensweise
- Man beschreibt in eigenen Worten, wie man vorgehen würde um die gegebene Problemstellung zu lösen
- **Achtung:** Unterscheiden zwischen Problemstellung und Lösungsbeschreibung!
- Auch in Prosaform sollten die Einzelschritte eindeutig beschrieben sein
- Nicht standardisiert $\rightarrow$ Beschreibung von Algorithmen inkonsistent

Prosabeschreibung

Für unseren Algorithmus

Addiere alle geraden Zahlen

Lese die Zahl `n` ein.
Anschließend setze die Zählvariable `i` sowie die Ergebnisvariable `res` auf `0`.
Wenn `i` gerade ist, addiere `i` auf die Ergebnisvariable. Erhöhe anschließend `i` um `1`. Wiederhole die letzten zwei Schritte bis `i` größer ist als `n`.
Gebe `res` aus

Darstellung als Pseudocode

Der Zwischenweg

- Mischung aus Prosa und tatsächlichem Code
- Orientiert sich an den in Programmiersprachen vorhandenen Strukturen (If-then-else, Schleifen...)
- Nutzt dabei aber leicht verständliche und programmiersprachenunabhängige Begriffe
- Wie Code in der Regel zeilenweise auf atomare Operationen beschränkt
- Keine formale Standardisierung, dadurch auch hier Inkonsistenzen möglich $\rightarrow$ Aber weniger als bei Prosabeschreibung

Pseudocode

Für unser Pseudoproblem

```
1 LESE n
2 SETZE res=0
3 FUER i=0 BIS n
4     WENN istGerade(i) DANN
5         res+=i
6     ENDE WENN
7 ENDE FUER
8 GEBE res AUS
```

Struktogramme

Der erste Standard

- Entwickelt durch *Nassi Shneidermann*
- Grafische Darstellung von Algorithmen
- Standardisiert nach **DIN 66261**
- Zerlegt den Algorithmus in elementare Grundstrukturen
- Die über die definierten Blöcke dargestellt werden
- Werden (lückenlos) von oben nach unten aneinander gereiht

Elemente von Struktogrammen

Anweisung

TODO: Abbildung Anweisungsblock

Elemente von Struktogrammen

Verzweigungen

TODO: Abbildung Verzweigungen

Elemente von Struktogrammen

Zählschleifen

TODO: Abbildung Zählschleife

Elemente von Struktogrammen

Schleifen

TODO: Abbildung Schleifen

Struktogram für unseren Algorithmus

TODO: Struktogram Algorithmus

Programmablaufplan

Der zweite Standard

- Bildet einen linearen Programmfluss aber
- Standardisiert nach **DIN 66001**
- Wie beim Struktogramm gibt es fest definierte Grundblöcke
- Diese werden hier jedoch über Pfeile verbunden

Elemente von Programmablaufplänen

Start, Stop, Anweisungsblock, Ein- und Ausgaben

TODO: Abbildung Anweisungsblock

Elemente von Programmablaufplänen

Verzweigungen

TODO: Abbildung Verzweigungen

Elemente von Programmablaufplänen

Zählschleifen

TODO: Abbildung Zählschleife

Elemente von Programmablaufplänen

Schleifen

TODO: Abbildung Schleifen

Programmablaufplan für unseren Algorithmus

TODO: PAP Algorithmus

Zusammenfassung

- Keine der dargestellten Formen ist optimal
- Verwendung kommt auf Anforderungen und persönliche Vorlieben an
- Keine der hier vorgestellten Methoden zur Abbildung komplexerer objektorientierter Zusammenhänge möglich
- Weitere Darstellungsformen:
  - Aktivitätsdiagramm
  - Petrinetze
  - Interaktionsdiagramme

Inhalt

Inhalt

Korrektheit von Algorithmen

Allgemeines

- Jeder Algorithmus sollte auch in allen Fällen das korrekte Ergebnis liefern...
- Klingt simpel, aber eindeutiger Beweis für alle Eingaben oft schwierig
- Testen an ausgewählten Beispielen **nicht** ausreichend
  - Jedoch verringern umfangreiche Tests natürlich das Risiko eines unentdeckten Fehler
- Korrektheit lässt sich im Grunde nur durch formalen Beweis zeigen
  - Wie zum Beispiel Induktionsbeweis
  - Diese sind häufig sehr umfangreich und komplex...
  - ...und deshalb auch nicht Teil der Vorlesung

Korrektheit von Algorithmen

Quelle:

„Program testing can be used to show the presence of bugs, but never to show their absence!.“

Edsger W. Dijkstra

Inhalt

Speicherkomplexität

Wie lässt sich diese messen?

- Wie schon erwähnt: Der verbrauchte Speicher ist Sprach- und Rechnerabhängig
- Mögliche Lösung über Definition von Referenzsprache und -system
- Messungen sind allerdings nicht repräsentativ
- Deswegen wird in der formalen Informatik mit dem *Random-Access-Machine*(RAM) Modell gearbeitet
  - Besteht im Grunde ausabzählbar unendlich vielen addressierbaren Speicherzellen
  - Für einen Algorithmus wird dann bestimmt, wie viele Speicherzellen genutzt werden müssen
  - Dies entspricht dann der Speicherkomplexität

Laufzeitkomplexität

Grundlegendes

- Gleiches Problem wie bei der Speicherkomplexität
- Deswegen hier ähnliches Modell:
  - Man bestimmt die Anzahl von „atomaren Operationen“ des Algorithmus
  - Diese Operationen sind vergleichbar mit Assembler-Befehlsrepertoire
- Beispiele für atomare Operationen:
  - Addition/Subtraktion/Multiplikation/Division zweier Zahlen
  - Lesen einer Variable von einer Speicheradresse
  - Schreiben einer Variable an eine bestimmte Adresse
  - Vergleich zweier Zahlen

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt