Computergrundlagen
Vom Problem zum Programm

Axel Arnold

Institut für Computerphysik
Universität Stuttgart

Wintersemester 2011/12

Problemanalyse

Was soll das Programm leisten?

Zum Beispiel

- eine Liste sortieren
- Echtzeitanalyse von Messwerten
- Computersimulation einer Polymerschmelze

Top–Down-Analyse

- schrittweises Zerlegen in Teilprobleme
- gibt Einsicht in die Komplexität
- Welche Teile gibt es schon? Bibliotheken, ...

Wer kann mir helfen?

- Wer hat Erfahrung mit ähnlichen Problemen?

Aufwandsabschätzung

- Lohnt sich das Programmieren überhaupt?

Problemanalyse

Gründe, ein Programm zu entwickeln:

- mit den vorhandenen Mitteln dauert es zu lange
  - automatisiertes Sortieren/Sammeln tausender Datensätze
  - Skript durch eine dediziertes Programm ersetzen
- das Programm kann wiederverwendet werden
  - löst ein generisches Problem (z.B. Statistik)
  - regelmäßige Aufgaben (z.B. Backup)
- Festschreiben einer Vorgehensweise
  - wenn sich Parameter nicht ändern dürfen (z.B. Backup)
  - um anderen ein Vorgehen vorzugeben (z.B. Abgabeskript)
- um komplexere Programme oder Algorithmen zu verstehen

Gründe, es zu lassen:

- das Programm wird viel zu langsam sein
- die Datenmengen werden zu groß
- das Programmieren dauert länger, als das Programm Zeit spart

Methodenwahl

Wahl der Rechenstrukturen (Algorithmen)

- Aufteilen in möglichst kleine sinnvolle Einheiten
- möglichst allgemein formulieren $\Longrightarrow$ wiederverwendbar
- Welche bekannten Algorithmen kann ich benutzen?

Wahl von Datentypen und -strukturen

- Ganz-, Fließkommazahlen, Strings, ...
- Listen, Tupel, Wörterbücher, Klassen
- Datenbanken

Wahl der Programmiersprache

- Skriptsprachen für Automatisierung und kleine Aufgaben
- Compilersprachen für sehr rechenaufwändige und oft wiederkehrende Aufgaben

Implementation

Programmieren

- modular programmieren: Teilprobleme in getrennten Funktionen, Modulen, Klassen
- Probleme von außen nach innen bearbeiten (Top–Down)
- Kommentare nicht für „Reservecode“ missbrauchen

Dokumentieren

- schon *während* des Programmierens
- Kommentare im Code und Anleitung/Hilfe

Testen

- möglichst einzelne Teile getrennt testen
- so früh wie möglich
- Tests müssen schnell ausführbar sein

Testen des Programms

Funktion des Programms

- *generische* Fälle prüfen, etwa Zufallszahlen
- Extremwerte: leere Eingabe, 0, Grundzustand, ...
- Beispiele mit bekannter Lösung
  – z.B. harmonischer Oszillator, ideales Gas, ...
- bekannte implizite Eigenschaften
  – z.B. Energieerhaltung, Normerhaltung der FFT, ...

Fehlerbehandlung

- alle Fehlermeldungen sollten in Tests ausgelöst werden
- Verhalten bei nicht erlaubten Eingaben überprüfen

Effizienz

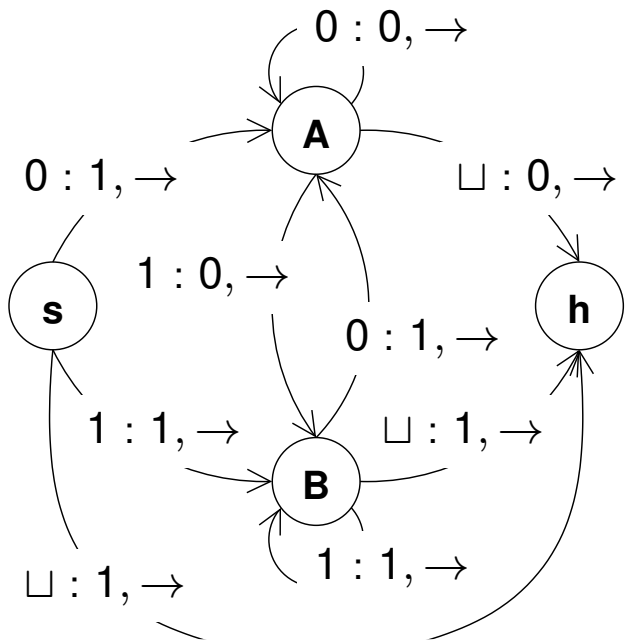
- Laufzeit des Programms bei generischer Eingabe
- Fest- und Hauptspeicherbedarf

Beispiel Universelle Turingmaschine

ein Programm, das die Ausgabe einer Turingmaschine berechnet

Eine *Turingmaschine* ist definiert durch

- Arbeitsalphabet Γ mit Leerzeichen $\sqcup \in \Gamma$
- Zustände Z
- Start- und Endzustände $s, h \in Z$
- Überführungsfunktion
  $\delta : Z \times \Gamma \to Z \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$



Problemanalyse

- Programm sinnvoll bei langen Eingaben / komplexem δ (z.B. fleißiger Biber)
- universelle Touringmaschine ist wiederverwertbar
- eine einzige Schleife genügt

Implementation

Wahl von Datentypen und -strukturen

- Alphabet: ganze Zahlen
- Band: Liste von Alphabetwerten, auch negative Indizes
- Position im Band: Listenindex, ganze Zahl
- Zustände: beliebige Zeichenketten
- Übergangsfunktion: Wörterbuch, ordnet Zustand, Wert-Paar einen neuen Zustand, Wert und eine Richtung zu
- Richtung: ± 1 oder 0, Änderung des Listenindex

Wahl der Programmiersprache

- Python hat effiziente Listen und Wörterbücher

Testen

- leere Eingabe, leere Übergangsfunktion nicht erlaubt
- einfache Touringmaschine (1 einfügen o.ä.)

Beispiel Sortieren

Sortieren einer Liste $A = a_o, \ldots, a_N$, so dass die sortierte Liste A' $a'_0 \leq a'_1 \leq \ldots \leq a'_N$ erfüllt

Problemanalyse

- sehr generisches Problem, effiziente Lösung wichtig
- viele bekannte Algorithmen:
  Bubblesort, Quicksort, Heapsort, ...
- zu allen gibt es gute Wikipedia-Artikel
- optimaler Algorithmus von der genauen Aufgabe abhängig

Detailfragen

- Was heißt „$\leq$“ bei beliebigen Datentypen?
- Was ist mit gleichen Einträgen?

Sortieralgorithmus: Bubblesort

Idee

- paarweises Sortieren, die größeren Werte steigen wie Blasen auf
- ein Durchlauf aller Elemente $\Longrightarrow$ größtes Element ganz oben
- m Durchläufe $\Longrightarrow$ m oberste Elemente einsortiert
- $\Longrightarrow$ nach spätestens N Durchläufen fertig
- fertig, sobald nichts mehr vertauscht wird

Effizienz

- im Schnitt $N/2$ Durchläufe mit $N/2$ Vergleichen
  $\Longrightarrow$ Laufzeit $\mathcal{O}(N^2)$
- Auch Worst Case $N-1+N-2+\ldots+1=\mathcal{O}(N^2)$
- Kein zusätzlicher Speicherbedarf

Sortieralgorithmus: Bubblesort

Implementation

```
N = len(liste)
for runde in range(N):
    getauscht = False
    for k in range(N - runde - 1):
        if liste[k] > liste[k+1]:
            liste[k], liste[k + 1] = liste[k+1], liste[k]
            getauscht = True
    if not getauscht: break
```

Test

- leere Liste
- Zufallslisten von ca. 10–100 Elementen
- benötigt noch ein Testprogramm drumherum
- Fehler, falls `liste` keine Liste ist

Sortieralgorithmus: Quicksort

Idee

- Teile und Herrsche (Divide & Conquer):
  Aufteilen in zwei kleinere Unterprobleme
- Liste ist fertig sortiert, falls $N \leq 1$
- wähle *Pivot*element p
- erzeuge Listen K und G der Elemente kleiner/größer als p
- sortiere die beiden Listen K und G
- Ergebnis ist die Liste $K \oplus \{p\} \oplus G$

Effizienz

- im Schnitt $\log_2 N$ Rekursionen, N Sortierungen pro Rekursion
  $\Longrightarrow$ Laufzeit $\mathcal{O}(N \log N)$
- Aber Worst Case $N - 1 + N - 2 + \ldots + 1 = \mathcal{O}(N^2)$
- Zusätzlicher Speicherbedarf mindestens $O(\log N)$

Sortieralgorithmus: Quicksort

Implementation

```
if len(liste) <= 1: return
pivot = liste[0]
kleiner  = []
groesser = []
for element in liste[1:]:
    if element < pivot:
        kleiner.append(element)
    else:
        groesser.append(element)
sortiere(kleiner)
sortiere(groesser)
liste[:] = kleiner + [pivot] + groesser
```

Test

analog Bubblesort