Algorithmen und Datenstrukturen

A5. Laufzeitanalyse: Einführung und Selectionsort

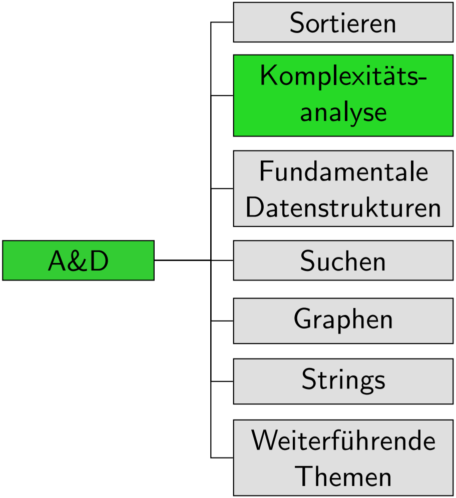
Marcel Lüthi and Gabriele Röger

Universität Basel

10. März 2021

A5.1 Laufzeitanalyse Allgemein

Inhalt dieser Veranstaltung



Exakte Laufzeitanalyse unrealistisch

- Wäre schön: Formel, die für konkrete Eingabe berechnet, wie lange das Programm läuft.
- exakte Laufzeitprognose schwierig, da zu viele Einflüsse:
  - Geschwindigkeit und Architektur des Computers
  - Programmiersprache
  - Compilerversion
  - aktuelle Auslastung (was sonst noch läuft)
  - Cacheverhalten

Wir können und wollen das nicht alles in die Formel aufnehmen.

Laufzeitanalyse: Vereinfachung 1

Zähle Anzahl der Operationen statt die Zeit zu messen!

Was ist eine Operation?

- Idealerweise: eine Zeile Maschinencode oder – noch präziser – ein Prozessorzyklus
- Stattdessen: Anweisungen, die konstante Zeit benötigen
  - konstante Zeit: Laufzeit unabhängig von Eingabe
  - ignoriere Laufzeitunterschiede verschiedener Anweisungen
  - z.B. Addition, Zuweisung, Verzweigung, Funktionsaufruf
  - grob: Operation = eine Zeile Code
  - aber: auch beachten, was dahinter steht
    z.B. Schritte innerhalb einer aufgerufenen Funktion

Wichtig: Laufzeit ungefähr proportional zu Anzahl Operationen

Laufzeitanalyse: Vereinfachung 2

Schätze ab statt genau zu zählen!

- Meistens Abschätzung nach oben („obere Schranke“)
  Wie viele Schritte braucht das Programm höchstens?
- Manchmal auch Abschätzung nach unten („untere Schranke“)
  Wie viele Schritte werden mindestens ausgeführt?

„Laufzeit“ für Abschätzung der Anzahl ausgeführter Operationen

Laufzeitanalyse: Vereinfachung 3

Abschätzung nur abhängig von Eingabegrösse

- $T(n)$: Laufzeit bei Eingabe der Grösse n
- Bei adaptiven Verfahren unterscheiden wir
  - Beste Laufzeit (best case)
    Laufzeit bei günstigstmöglicher Eingabe
  - Schlechteste Laufzeit (worst case)
    Laufzeit bei schlechtestmöglicher Eingabe
  - Mittlere Laufzeit (average case)
    Durchschnitt der Laufzeit über alle Eingaben der Grösse n

Kostenmodelle

Auch: Analyse mit Kostenmodell

- Identifiziere grundlegende Operationen der Algorithmenklasse
  z.B. für vergleichsbasierte Sortierverfahren
  - Vergleich von Schlüsselpaaren
  - Tausch zweier Elemente oder Bewegung eines Elementes
- Schätze Anzahl dieser Operationen ab.

Beispiel aus C++-Referenz

function template

std::sort

<algorithm>

default (1)	`template <class RandomAccessIterator>` `void sort (RandomAccessIterator first, RandomAccessIterator last);`
custom (2)	`template <class RandomAccessIterator, class Compare>` `void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);`

Sort elements in range

Sorts the elements in the range `[first,last)` into ascending order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

Complexity

On average, linearithmic in the distance between *first* and *last*: Performs approximately $N*\log_2(N)$ (where *N* is this distance) comparisons of elements, and up to that many element swaps (or moves).

http://www.cplusplus.com/reference/algorithm/sort/

A5.2 Beispiel: Selectionsort

Selectionsort: Algorithmus

```python
def selection_sort(array):
    n = len(array)
    for i in range(n - 1):  # i = 0, ..., n-2
        # find index of minimum element at positions i, ..., n-1
        min_index = i
        for j in range(i + 1, n):  # j = i+1, ..., n-1
            if array[j] < array[min_index]:
                min_index = j
        # swap element at position i with minimum element
        array[i], array[min_index] = array[min_index], array[i]
```

Selectionsort mit Kostenmodell

```python
def selection_sort(array):
    n = len(array)
    for i in range(n - 1):  # i = 0, ..., n-2
        # find index of minimum element at positions i, ..., n-1
        min_index = i
        for j in range(i + 1, n):  # j = i+1, ..., n-1
            if array[j] < array[min_index]:
                min_index = j
        # swap element at position i with minimum element
        array[i], array[min_index] = array[min_index], array[i]
```

→ n-1 mal Tausch zweier Elemente („linear“)
→ 0.5(n-1)n Schlüsselvergleiche („quadratisch“)

Selectionsort: Analyse I

Wir zeigen: $T(n) \leq c' \cdot n^2$ für $n \geq 1$ und irgendeine Konstante c'

- Äussere Schleife (3-10) und innere Schleife (6-8)
- Anzahl Operationen für jede Iteration der äusseren Schleife:
  - Konstante a für Anzahl Operationen in Zeilen 7 und 8
  - Konstante b für Anzahl Operationen in Zeilen 5 und 10

i	# Operationen
0	$a(n-1)+b$
1	$a(n-2)+b$
	$\ldots$
n-2	$a \cdot 1 + b$

- Insgesamt: $T(n) = \sum_{i=0}^{n-2}(a(n-(i+1))+b)$

Selectionsort: Analyse II

$$
\begin{aligned}
T(n) &= \sum\nolimits_{i=0}^{n-2}(a(n-(i+1))+b)\\
&= \sum\nolimits_{i=1}^{n-1}(a(n-i)+b)\\
&= a\sum\nolimits_{i=1}^{n-1}(n-i)+b(n-1)\\
&= 0.5a(n-1)n+b(n-1)\\
&\leq 0.5an^2+b(n-1)\\
&\leq 0.5an^2+b(n-1)n\\
&\leq 0.5an^2+bn^2\\
&= (0.5a+b)n^2
\end{aligned}
$$

$\Rightarrow$ mit $c' = (0.5a+b)$ gilt für $n \geq 1$, dass $T(n) \leq c' \cdot n^2$

Selectionsort: Analyse III

Zu grosszügig abgeschätzt?

Wir zeigen für $n \geq 2$: $T(n) \geq c \cdot n^2$ für irgendeine Konstante c

$$\begin{aligned} T(n) &= \cdots = 0.5a(n-1)n + b(n-1) \\ &\geq 0.5a(n-1)n \\ &\geq 0.25an^2 \qquad (n-1 \geq 0.5n \text{ für } n \geq 2) \end{aligned}$$

$\Rightarrow$ mit $c = 0.25a$ gilt für $n \geq 2$, dass $T(n) \geq c \cdot n^2$

Theorem

Selectionsort hat quadratische Laufzeit, d.h. es gibt Konstanten $c > 0, c' > 0, n_0 > 0$, so dass für $n \geq n_0$: $cn^2 \leq T(n) \leq c'n^2$.

Selectionsort: Analyse IV

Quadratische Laufzeit:
doppelt so grosse Eingabe, ca. viermal so lange Laufzeit

Was bedeutet das in der Praxis?

- Annahme: $c = 1$, eine Operation dauert im Schnitt 10^{-8} Sek.
- Bei 1 Tsd. Elementen warten wir
  $10^{-8} \cdot (10^3)^2 = 10^{-8} \cdot 10^6 = 10^{-2} = 0.02$ Sekunden.
- Bei 10 Tsd. Elementen $10^{-8} \cdot (10^4)^2 =$ 1 Sekunde
- Bei 100 Tsd. Elementen $10^{-8} \cdot (10^5)^2 =$ 100 Sekunden
- Bei 1 Mio. Elementen $10^{-8} \cdot (10^6)^2$ Sekunden $=$ 2.77 Stunden
- Bei 1 Mrd. Elementen $10^{-8} \cdot (10^9)^2$ Sekunden $=$ 317 Jahre
  1 Mrd. Zahlen bei 4 Bytes/Zahl sind „nur“ 4 GB.

Quadratische Laufzeit problematisch für grosse Eingaben

A5.3 Zusammenfassung

Zusammenfassung

- Bei der Laufzeitanalyse schätzen wir die Anzahl der ausgeführten Operationen ab.
  - Wir zählen nicht exakt.
  - Wir ignorieren, wie lange eine Operation tatsächlich dauert.
  - Hauptsache: Laufzeit ungefähr proportional zu Anzahl Operationen.
- Selectionsort hat quadratische Laufzeit und benötigt linear viele Vertauschungen und quadratisch viele Schlüsselvergleiche.