Algorithmen und Datenstrukturen

A5. Laufzeitanalyse: Einführung, Selection- und Mergesort

Marcel Lüthi and Gabriele Röger

Universität Basel

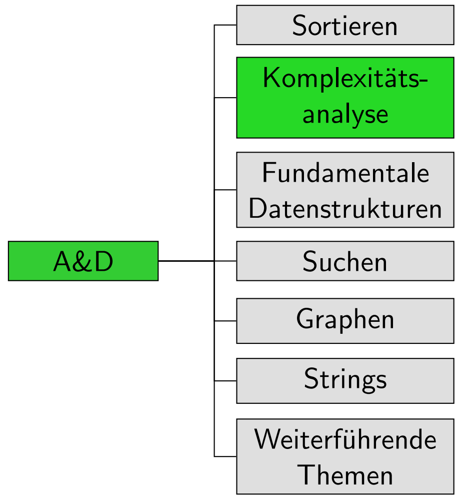
27. Februar 2019

Algorithmen und Datenstrukturen

27. Februar 2019 — A5. Laufzeitanalyse: Einführung, Selection- und Mergesort

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: 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	$a(n-2)+b$
	...
n-2	$a \cdot 1 + b$

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

Selectionsort: Analyse II

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

$\Rightarrow$ mit $c' = 0.5(a+b)$ 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

Selectionsort mit Kostenmodell

```
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“)

A5.3 Exkurs: Logarithmus

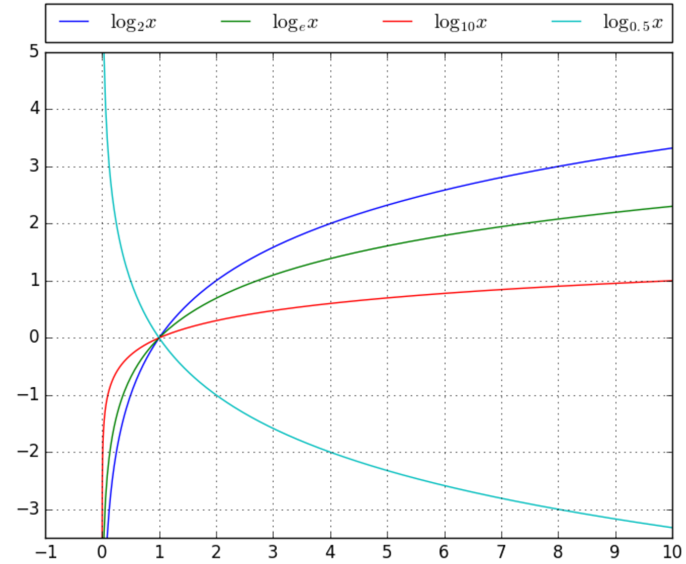
Logarithmus

- In der Analyse von Mergesort werden wir eine Logarithmusfunktion verwendet.
- Dies ist bei der Analyse von Laufzeiten oft der Fall.
- Der Logarithmus zur Basis b ist invers zur Exponentialfunktion mit Basis b, also

$$\log_b x = y \text{ gdw. } b^y = x.$$

- Beispiele: $\log_2 8 = 3$, da $2^3 = 8$
  Beispiele: $\log_3 81 = 4$, da $3^4 = 81$
- $\log_b a$ intuitiv (wenn das glatt aufgeht):
  „Wie oft muss man a durch b teilen bis man bei 1 ist?“

Logarithmus: Illustration



Rechenregeln Logarithmus

Die Rechenregeln ergeben sich direkt aus den Regeln $(a^x)^y = a^{xy} = (a^y)^x$ und $a^x a^y = a^{x+y}$:

Produktregel	$\log_b(xy) = \log_b x + \log_b y$
Potenzrechnung	$\log_b(x^r) = r \log_b x$
Basisumrechnung	$\log_b x = \log_a x / \log_a b$
Summenregel	$\log_b(x + y) = \log_b x + \log_b(1 + y/x)$

Logarithmus: Beispielrechnung

Bei der Algorithmenanalyse begegnet man öfters Ausdrücken der Form $a^{\log_b x}$. Wie bekommt man da den Logarithmus aus dem Exponenten?

Beispiel: $5^{\log_2 x}$

Wir verwenden $5 = 2^{\log_2 5}$.

$$\begin{aligned}
5^{\log_2 x} &= (2^{\log_2 5})^{\log_2 x} \\
&= 2^{\log_2 5 \log_2 x} \\
&= 2^{\log_2 x \log_2 5} \\
&= (2^{\log_2 x})^{\log_2 5} \\
&= x^{\log_2 5} \\
&\approx x^{2.32}
\end{aligned}$$

A5.4 Beispiel: Mergesort

Merge-Schritt

c_1 c_2 c_3

```python
def merge(array, tmp, lo, mid, hi):
    i = lo
    j = mid + 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        if j > hi or (i <= mid and array[i] <= array[j]):
            tmp[k] = array[i]
            i += 1
        else:
            tmp[k] = array[j]
            j += 1
    for k in range(lo, hi + 1):  # k = lo,...,hi
        array[k] = tmp[k]
```

Wir analysieren Laufzeit für $m := \mathrm{hi} - \mathrm{lo} + 1$

Merge-Schritt: Analyse

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\geq (c_2 + c_3) m \end{aligned}$$

Für $m \geq 1$:

$$\begin{aligned} T(m) &= c_1 + c_2 m + c_3 m \\ &\leq c_1 m + c_2 m + c_3 m \\ &= (c_1 + c_2 + c_3) m \end{aligned}$$

Theorem

Der Merge-Schritt hat lineare Laufzeit, d.h. es gibt Konstanten $c, c', n_0 > 0$, so dass für alle $n \geq n_0$: $cn \leq T(n) \leq c'n$.

Bottom-Up-Mergesort

```
1  def sort(array):
2      n = len(array)
3      tmp = list(array)
4      length = 1
5      while length < n:
6          lo = 0
7          while lo < n - length:
8              mid = lo + length - 1
9              hi = min(lo + 2 * length - 1, n - 1)
10             merge(array, tmp, lo, mid, hi)
11             lo += 2 * length
12         length *= 2
```

Wir verwenden für die Abschätzung:

- c_1 Zeilen 2–4
- c_2 Zeilen 6 und 12
- c_3 Zeilen 8,9,11

Annahme: merge benötigt c_4(hi-lo+1) Operationen.

Bottom-Up-Mergesort: Analyse I

Annahme: $n = 2^k$ für ein $k \in \mathbb{N}_{>0}$

Iterationen der äusseren Schleife (m für hi-lo+1):

- Iteration 1: $n/2$ mal innere Schleife mit Merge für $m = 2$
  $c_2 + n/2(c_3 + 2c_4) = c_2 + 0.5c_3 n + c_4 n$
- Iteration 2: $n/4$ mal innere Schleife mit Merge für $m = 4$
  $c_2 + n/4(c_3 + 4c_4) = c_2 + 0.25c_3 n + c_4 n$
- ...
- Äussere Schleife endet nach letzter Iteration ℓ.
- Iteration ℓ: 1 mal innere Schleife mit Merge für $m = n$
  $c_2 + n/n(c_3 + nc_4) = c_2 + c_3 + c_4 n$

Insgesamt $T(n) \leq c_1 + \ell(c_2 + c_3 n + c_4 n) \leq \ell(c_1 + c_2 + c_3 + c_4) n$

Bottom-Up-Mergesort: Analyse II

Wie gross ist ℓ?

- In Iteration i ist für den Merge-Schritt $m = 2^i$
- In Iteration ℓ hat Merge-Schritt $m = 2^\ell = n$
- Da $n = 2^k$ ist $\ell = k = \log_2 n$.

Mit $c := c_1 + c_2 + c_3 + c_4$ erhalten wir $T(n) \leq cn \log_2 n$.

Bottom-Up-Mergesort: Analyse III

Was, wenn n keine Zweierpotenz, also $2^{k-1} < n < 2^k$?

- Trotzdem k Iterationen der äusseren Schleife.
- Innere Schleife verwendet nicht mehr Operationen.
- $T(n) \leq cnk = cn(\lfloor \log_2 n \rfloor + 1) \leq 2cn \log_2 n$ (für $k > 2$)

Bottom-Up-Mergesort: Analyse IV

Ähnliche Abschätzung auch für untere Schranke möglich.
→ Übung

Theorem
Bottom-Up-Mergesort hat leicht überlineare Laufzeit, d.h. es gibt Konstanten $c, c', n_0 > 0$, so dass für alle $n \geq n_0$ gilt $cn \log_2 n \leq T(n) \leq c' n \log_2 n$.

Leicht überlineare Laufzeit

Leicht überlineare Laufzeit $n \log_2 n$:
→ doppelt so grosse Eingabe, etwas mehr als doppelt 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 $10^{-8} \cdot 10^3 \log_2(10^3) \approx$ 0.0001 Sekunden.
- Bei 10 Tsd. Elementen $\approx$ 0.0013 Sekunden
- Bei 100 Tsd. Elementen $\approx$ 0.017 Sekunden
- Bei 1 Mio. Elementen $\approx$ 0.2 Sekunden
- Bei 1 Mrd. Elementen $\approx$ 299 Sekunden

Laufzeit $n \log_2 n$ nicht viel schlechter als lineare Laufzeit

Mergesort mit Kostenmodell I

Schlüsselvergleiche

- Werden nur in `merge` durchgeführt.
- Mergen zweier Teilfolgen der Länge m und n benötigt bestenfalls $\min(n, m)$ und schlimmstenfalls $n + m - 1$ Vergleiche.
- Bei zwei etwa gleich langen Teilfolgen sind das linear viele Vergleiche, d.h. es gibt $c, c' > 0$, so dass Anzahl Vergleiche zwischen cn und $c'n$ liegt.

→ Anzahl der zum Sortieren einer Sequenz notwendigen Schlüsselvergleiche ist leicht überlinear in der Länge der Sequenz (analog zu Laufzeitanalyse).

Mergesort mit Kostenmodell II

Elementbewegungen

- Werden nur in `merge` durchgeführt.
- $2n$ Bewegungen für Sequenz der Länge n.
- Insgesamt für Mergesort leicht überlinear
  (analog zu Schlüsselvergleichen)

A5.5 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.
- Mergesort hat leicht überlineare Laufzeit, Schlüsselvergleiche und Elementbewegungen.