Kapitel 6: Effiziente Algorithmen

Begriffsklärungen

Algorithmus ist `effizient`, wenn seine Ausführung möglichst wenig Aufwand verursacht.

Aufwand: Rechenzeit, Speicherplatz, Anzahl bestimmter Elementaroperationen.

Die *Komplexität* eines Algorithmus gibt den Aufwand als Funktion der Eingabe oder deren *Größe* an.

Bei Angabe der Komplexität als Funktion der Eingabegröße unterscheidet man

- *Best case:* Aufwand bei am günstigsten gewählter Eingabe zu fester Größe
- *Worst case:* Aufwand bei am ungünstigsten gewählter Eingabe zu fester Größe
- *Average case:* Erwartungswert des Aufwands bei Eingaben einer festen Größe bzgl. einer bestimmten Verteilung.

Beispiel: `insertel`

```
let rec insertel = function
   (a, []) -> [a]
 | (a, h::t) -> if a <= h then a::h::t else h :: insertel(a,t)
```

Anzahl der Auswertungsschritte von `insertel(a,l)` als Funktion von $n =$ `length(l)`.

- Best case: 4 (Element `a` wird am Anfang eingefügt)
- Worst case: $3n$ (Element `a` wird am Ende eingefügt)
- Average case: $1.5 \cdot n$ (Element `a` wird "in der Mitte" eingefügt)

Größenordnungen

Oft interessiert man sich nur für die *Größenordnung* der Komplexität: konstant, linear, quadratisch, exponentiell, ...

Man gibt diese mit der O-Notation an:

Sei $f : \mathbb{N} \to \mathbb{N}$:

$$O(f) = \{g \mid \exists N.\exists c > 0.\forall n \geq N.g(n) \leq c \cdot f(n)\}$$

Insbesondere: $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ existiert $: g \in O(f)$.

Beispiele

- $2n^2 = O(n^2)$
- $n^{1000} = O(2^n)$
- $\log(n) = O(n)$
- $1000/n = O(1)$
- Best case Laufzeit von `insertel` $= O(1)$
- Average case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `inssort` $= O(n^2)$

Notationen

Man verwendet oft folgende Abkürzungen:

- $f(n)$ statt f, z.B.: $2n^2 \in O(n^2)$ statt $\mathbf{function}(n)2n^2 \in O(\mathbf{function}(n)n^2)$,
- $g = O(f)$ statt $g \in O(f)$, z.B.: $n^2 = O(2^n)$,
- $f + O(g)$ statt $\{h \mid \exists u \in O(g).h = f + u\}$, z.B.: $\sum_{i=1}^{n} i^k = \frac{1}{k+1} n^{k+1} + O(n^k)$.

Entwicklung effizienter Programme

Wie kommt man zu effizienten Programmen?

- Effiziente Algorithmen
- Effiziente Datenstrukturen
- Einbeziehung des Auswerteprozesses, z.B.: günstige Rekursionsschemata, Vermeidung von “append”.

Der dritte Punkt lässt sich relativ schematisch umsetzen und wird mehr und mehr durch Fortschritte in der Compilertechnik automatisiert.

Die ersten beiden Punkte erfordern Kreativität, Fachwissen und Erfahrung.

Beispiel für effizienten Algorithmus

Wir haben gesehen, dass Sortieren durch Einfügen quadratische Komplexität hat.

Wir betrachten jetzt ein rekursives Verfahren, welches sowohl praktisch, als auch theoretisch (asymptotisch) bessere Laufzeit aufweist.

Sortieren durch Mischen

Um eine Liste der Länge n zu sortieren:

- Teile man die Liste in zwei gleichgrosse Hälften
- Sortiere jede Hälfte durch rekursiven Aufruf (bei Länge ≤ 1 ist natürlich nichts zu tun)
- Füge die beiden sortierten Hälften “im Reißverschlussverfahren” zusammen.

In OCAML

```
let rec split = function
    [] -> [],[]
  | [a] -> ([a],[])
  |  a::b::l -> let u,v = split l in a::u,b::v

let rec merge = function
    ([],l) -> l
  | (l,[]) -> l
  | ((a::t as x), (b::u as y)) ->
      if a <= b then a::merge(t,y) else b :: merge(x,u)

let rec mergesort = function
      [] -> []
  |   [a] -> [a]
  |   l -> let u,v = split l in
           let u1,v1 = mergesort u , mergesort v in
           merge(u1, v1)
```

Empirische Laufzeitbestimmung

Die Funktion

```
Unix.gettimeofday : unit -> float
```

liefert die Zeit in Sekunden, die zwischen 1.1.1970 und dem Aufruf verstrichen ist.

Die Funktion

```
Random.int : int -> int
```

liefert einen “zufälligen” Integer im Bereich $0 \ldots n$.

Profiler

```
let profile f s =
  let before = Unix.gettimeofday() in
  let _ = f() in
  let after = Unix.gettimeofday() in
  print_string (s ^ ": " ^ string_of_float (after-.before) ^
                          " Sekunden.\n")
```

Der Aufruf `profile f s` wertet `f` aus und gibt die dafür verbrauchte Zeit zusammen mit dem String `s` aus.

Damit können wir die Laufzeit von `inssort` und `mergesort` abhängig von der Eingabe empirisch bestimmen.

Zusammenfassung

ab ca $n = 20$ ist mergesort effizienter,

bei $n = 5000$ läuft mergesort 100 Mal so schnell wie inssort.

Analytische Bestimmung der Laufzeit:

Sei $T(n)$ die Laufzeit von mergesort: Es gilt

$$T(n) = 2.T(n/2) + O(n)$$

Lösung dieser Gleichung:

$$T(n) = O(n \cdot \log n)$$