Algorithmen

Michael Kaufmann

19/10/2021 – 2. Vorlesung
Pseudocode - O-Notation - Laufzeitanalyse

Organisatorisches

Vorlesungstermine: Mo 10.30 – 11.45 und Di 8.15-9.45
Tutorien: ab dieser Woche
Helpdesk: Mo 12-14

Klausur: letzte Vorlesungswoche 8.2.22 ???
Nachklausur: letzte Woche vor SoSe Beginn
Teilnahme nur an Nachklausur möglich (dann keine 2. Chance!)
Übungsblätter: 50 Prozent der Punkte gefordert, Boni !

Anmeldung für Tutorien (am besten heute !!!)

Organisation: https://moodle.zdv.uni-tuebingen.de/

Gliederung

I. Einführung
- Motivation und Notationen
- Laufzeitanalyse, Rekursionen

II. Grundlegende Datenstrukturen
III. Graphenalgorithmen
IV. Sortieren
V. Suchen
VI. Generische algorithmische Methoden
VII. Algorithmen auf Zeichenketten

Erstes Beispiel aus Dasgupta: Fibonacci Zahlen

$$F(n) = \left\{ \begin{array}{ll} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F(n-1) + F(n-2), & \text{falls } n \geq 2 \end{array} \right\}$$

Ergibt die Folge 0,1,1,2,3,5,8,13,21, ...

Algorithmus 1: nicht-rekursive einfache Schleife

Algorithm: FibLoop(n)

Allokiere F als ein Array der Länge $n+1$;

$F[0] \leftarrow 0$;

$F[1] \leftarrow 1$;

for $i \in \{2, \ldots, n\}$ **do**

  $F[i] \leftarrow F[i-1] + F[i-2]$;

end

return $F[n]$

Beachte: Das ist Pseudo-Code, keine PS.
Kurz, lesbar, verständlich, nicht zu detailliert !

Algorithmus 2: Rekursion

Algorithm: FibRekursive(n)

if $n = 0$ **then**

  return *0*

end

if $n = 1$ **then**

  return *1*

end

return $FibRekursive(n-1) + FibRekursive(n-2)$

Laufzeiten der Fibonacci Algorithmen:
Zählen die 'Elementaroperationen'

FibLoop(n)

- Alloziere ein Array der Länge n, entweder 1 OP (Pointer auf den Anfang des Arrays, oder $<= n$ OPs (Initialisiere mit Nullen)
- 2 Elementaroperationen für 2 Zuweisungen.
- In jedem Schleifendurchlauf addieren wir 2 Zahlen.
  Da $2^{n/2} \leq F(n) \leq 2^n$ für $n \geq 6$, ist $F(n)$ etwa n Bits lang.
- Schleife wird $(n-1)$- mal durchlaufen.

Anzahl der Elementaroperationen: $1 + 2 + (n-1) \cdot n \approx n^2$

Beachte: Add. 'normaler' Zahlen gehen in $O(1)$
'lange' Zahlen in O(ihrer Länge)

Laufzeiten der Fibonacci Algorithmen: Zählen die 'Elementaroperationen'

FibRecursive(n)

- Anfangs jeweils 1 Elementaroperation
- Dann eine Addition und die Zeit für die rekursiven Aufrufe.
- Also $T(n) = 4 + T(addition) + T(n-1) + T(n-2) \geq 4 + T(n-1) + T(n-2)$
- Vergleich mit Definition von F_n zeigt, dass $T(n) \geq F_n$.
- Wegen $F_n \geq 2^{n/2}$ für $n \geq 6$, wir erhalten $T(n) \geq 2^{n/2}$.

Laufzeiten der Fibonacci Algorithmen: Vergleich

- Laufzeit von FibRecursive ist exponentiell in n. Also ist die Laufzeit sogar für mittelgroße n zu groß für alle denkbare Computer.
- Laufzeit von FibLoop ist quadratisch in n. Okay für relativ große n, aber nicht toll. Immerhin 'polynomiell'.
- Geht es besser als mit FibLoop ? Überlege !

Pseudocode

Formulieren der Algorithmen in Pseudocode

- Brauchen kompakte, informelle high-level Beschreibung der Funktionsweise
- Ähnlich wie Programmiersprache, aber verständlich für Menschen, weniger für Maschinen

Konventionen:

- Benutzen Standardkonstrukte wie IF, FOR, WHILE
- keine allzu feste Syntax
- oftmals high-level Anweisungen: Alloziere / Sortiere ein Array
- Oft unterschiedliche Syntax, nicht so wichtig hier. Hauptsache lesbar.
- Schreibe einfachen, verständlichen Code, keine Tricks
- Benutze Einrückungen
- latex mit ‘usepackage algorithm2e‘

Beispiel: Suchen in einem Array A nach einem Element a

Algorithm: NaiveSuche(A, a)

$it \leftarrow 0$; foundit $\leftarrow$ false;

while *(NOT foundit) AND* $(it < length(A))$ **do**

  $it \leftarrow it + 1$;

  if $A[it] = a$ **then**

    foundit = true; position = it

  end

end

if *foundit* **then**

  return *position*

end

else

  return *nicht gefunden*

end

Wie man algorithmische Qualität misst: O-Notation

O-Notation

– Betrachten Laufzeiten, abhängig von der 'Größe' des Problems'
– Problemgröße ist Länge der Eingabe
– Interessant sind große Probleminstanzen, kleine gehen eh schnell

$\Rightarrow$ betrachten Funktionenwachstum (i.e. Laufzeitverhalten)

– Genauer: Interessieren uns nur für Größenordnung:
– Also statt Laufzeit $3 \cdot n^3 + 50 \cdot n^2 + 10000$ betrachte n^3 (keine Konstanten, nur größter Term)

$\Rightarrow$ O-Notation

O-Notation

$f \in \mathrm{O}(g)$ bedeutet 'f ist von Ordnung höchstens g':

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) \leq cg(n)$$

$$\Leftrightarrow \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

$f \in \mathrm{o}(g)$ bedeutet 'f ist von Ordnung echt kleiner als g':

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) < cg(n)$$

$$\Leftrightarrow \limsup_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

O-Notation

$f \in \Omega(g)$ bedeutet 'f ist von Ordnung mindestens g':

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) \geq cg(n) > 0$$
$$\Leftrightarrow \limsup_{n\to\infty} \frac{f(n)}{g(n)} > 0$$
$$\Leftrightarrow g \in \mathrm{O}(f)$$

$f \in \omega(g)$ bedeutet 'f ist von Ordnung echt größer als g':

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) \geq cg(n) > 0$$
$$\Leftrightarrow \limsup_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$
$$\Leftrightarrow g \in \mathrm{o}(f)$$

$f \in \Theta(g)$ bedeutet 'f ist von selber Ordnung wie g'

$$\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$
$$\Leftrightarrow 0 < \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$
$$\Leftrightarrow f \in \mathrm{O}(g) \text{ und } f \in \Omega(g)$$

Beispiele:

- Betrachte $f(n) = n^2 + 3n + 7$.
  $f \in \mathrm{O}(n^8)$; $f \in \mathrm{o}(n^8)$; $f \in \omega(\log n)$; $f \in \Theta(n^2)$
- Betrachte $f(n) = n^2 + 10n + 5$.
  $f \in \mathrm{O}(n^2)$; aber $f \notin \mathrm{o}(n^2)$; $f \in \Omega(n^2)$; aber $f \notin \omega(n^2)$

Spezialfälle:

- Ist f Polynom mit Grad d , so ist $f \in \Theta(n^d)$. Außerdem $f \in o(exp(n))$ sowie $f \in \omega(\log n)$.
- Bei Logarithmen ist Basis egal: $\log_a \in \Theta(\log_b(n))$.
- Funktionen, die durch eine Konstante $b > 0$ beschränkt, wobei b unabhängig von n ist, ist $O(1)$.

Rechenregeln:

- $f \in O(g_1 + g_2)$ und $g_1 \in O(g_2) \Rightarrow f \in O(g_2)$.
- $f_1 \in O(g_1)$ und $f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$
- $f \in g_1 \cdot O(g_2) \Rightarrow f \in O(g_1 g_2)$
- Für Konstante $c > 0$ gilt: $O(cg)$ ist gleich $O(g)$.
- $f \in O(g_1), g_1 \in O(g_2) \Rightarrow f \in O(g_2)$
- Oft schreibt man $f = O(g)$, wenn man $f \in O(g)$ meint.

Laufzeitanalyse

Keine Laufzeitmessung !
Da abhängig von benutztem Rechner und jeweiliger Eingabe

Benutzen 'RAM' Modell und zählen 'Elementaroperationen'
Random Access Machine (RAM): sequentielles Abarbeiten, uniformer Speicherzugriff

Elementaroperationen:

- Laden und Schreiben eines Bits vom und auf den Speicher
- Elementare Arithmetik in konstanter Zeit ($O(1)$)
- Zahlen in 64-Bit Arithmetik ebenfalls in $O(1)$
- Bei langen Zahlen Arithmetik nicht mehr elementar !!

Beispiele: Mitüberlegen ! In O-Notation !

- Was kostet Addition zweier standard Integer? Zweier Zahlen der Länge n ?
- Was kostet Multiplikation zweier standard Integer? Zweier Zahlen der Längen n und m ?
- Lies n Strings der Länge 10 ein
- Was kostet Operation $a + b \cdot c$, falls a aus n^2 Bits besteht, b aus $\log n$ Bits, c aus n Bits

Wie ist nun die Laufzeit eines Algorithmus ?

Wie schwierig ist ein 'Problem' ?

Probleme und Instanzen

Problem: Abstrakte Frage, die wir lösen wollen. Menge von Eingaben plus die gewünschten Ausgaben

Verschiedene Typen von Problemen: Entscheidungsprobleme, Optimierungsprobleme, Suchprobleme, etc

Formales gibt es in 'Theoretischer Informatik' (4. Semester)

Instanz: Konkreter Fall eines Problems, z.b. explizites Array, das zu sortieren ist.

Wie ist die Laufzeit eines Algorithmus?
$\rightarrow$ Instanzen haben verschiedene Laufzeiten
$\rightarrow$ Worst Case / Average Case ?

Worst Case Laufzeit

Was ist die längste Laufzeit für den Algorithmus, maximiert über alle möglichen Instanzen ?

- Sei $\mathcal{I}_n$ die Menge aller Instanzen der Länge n
- Sei $T(I_n)$ die Laufzeit des Algorithmus an Instanz I_n.
- Dann ist die Worst Case Laufzeit für Eingaben der Länge n definiert als:
$$T_{wc}(n) := \max\{T(I_n) | I_n \in \mathcal{I}_n\}$$

Beispiel: NaiveSuche(A, a).
Schaue nach:
Beachte, im Worst Case müssen wir n while-Schleifen durchlaufen. Sollte sowas wie $2 + n \cdot 7 + 2 \in O(n)$ rauskommen

Untere Schranken / obere Schranken im Worst Case

– Für eine untere Schranke T_{lower}, finde eine Instanz I_n mit Laufzeit mindestens T_{lower}. Dann gilt: $T_{wc}(n) \geq T_{lower}$

– Für die obere Schranke T_{upper}, muss gelten, dass die Laufzeit für alle Instanzen höchstens T_{upper} ist. Dann gilt: $T_{wc}(n) \leq T_{upper}$

Diskussion:

– Worst Case Szenarios sind wichtig in sicherheitskritischen Systemen.
– Oft ist das Laufzeitverhalten viel besser, und das Worst Case Verhalten tritt nur in ganz seltenen Fällen auf.

$\rightarrow$ Mittlere Laufzeit (Average Case)

Average Case Laufzeit

Was ist die mittlere Laufzeit für den Algorithmus, gemittelt über alle möglichen Instanzen ?

- Sei $\mathcal{I}_n$ der Raum aller Instanzen der Länge n
- Sei $T(I_n)$ die Laufzeit des Algorithmus an Instanz I_n.
- Dann ist die Average Case Laufzeit für Eingaben der Länge n definiert als:

$$T_{ac}(n) := \frac{1}{\mathcal{I}_n} \cdot \sum_{I_n \in \mathcal{I}_n} T(I_n)$$

Bemerkung:

- Oft ist es kritisch, anzunehmen, dass alle Instanzen gleichwahrscheinlich sind. Manche kommen öfters vor als andere.
- Hier meist Worst Case Szenarien.

Nützliches zum Rechnen:
Logarithmen, Fakultätsfunktion,...

Summen

$\sum_{i=1}^{n} i = n(n+1)/2$

$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ geometr. Reihe

$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ für $|x| < 1$

$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$ für $|x| < 1$ warum ? Ableiten !!

$\sum_{i=1}^{n} 1/i = 1 + 1/2 + 1/3 + ... = \ln n + O(1)$ n-te harmon. Zahl

Logarithmen-Regeln

$\log(a \cdot b) = \log a + \log b$

$\log(a^b) = b \log a$

$\log_a b = \frac{\log b}{\log a}$

$b^{\log_b a} = a$

Überlege: Was ist $2^{\log_4 n}$?

Überlege: Wieviel Stellen im Dezimalsystem hat eine 64-Bit Zahl maximal ?

$n!$

$$n! = n \cdot (n-1) \cdot (n-2) \ldots . 3 \cdot 2 \cdot 1$$

$$n! = \sqrt{2\pi n} \cdot (n/e)^n \cdot (1 + o(1))$$ Stirling-Approx.

einfacher: $(n/2)^{n/2} \leq n! \leq n^n$