Datenstrukturen & Algorithmen 25. Februar, 2015

Handout 1

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Allgemeines, Asymptotische Analyse ($\mathcal{O}$-, Ω-, Θ-Notation), Rekursionsgleichungen

For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. - Francis Sullivan

Keep programming every day for 2 years and you'll be a great programmer. Keep programming every day for 2 years and then take an algorithms course and you'll become a world class programmer. - Charles E. Leiserson MIT

Administratives und Vorlesungsinformationen

- Übungsabgabe: Mittwochs in der Übungsstunde oder bis spätestens Mittwoch 15 Uhr per E-Mail, keine Vorbesprechung
- ~~Testat~~Chocolatbedingung: alle bis auf 2 Übungen und Programmieraufgaben sinnvoll bearbeiten und abgeben
- Notizen, Code, Slides usw. jeweils auf http://dgraf.ch/da15
- Bei Fragen, Rückmeldungen, Übungsabgabe: grafdan@ethz.ch
- Inhalte der Übungsstunden:
  - Zusammenfassung und Vertiefung der Theorie aus der Vorlesung
  - Musterlösung der aktuellen Übungsserie (bei Bedarf auch Anmerkungen zur Korrektur der letzten Serie)
  - Keine Vorbesprechung der neuen Übungsserie
  - zusätzliche Aufgaben im selben Stil wie die Serien und die Prüfung
  - ab und zu: spezielle Programmieraufgaben, Rätsel und Brain Teaser

Zusätzliches Material

Literatur

Lest ein Buch für die Vorlesung. Die Themen die in diesem Kurs behandelt werden sind grundlegend und werden in den meisten Algorithmen-Büchern erörtert. Die Bücher sind meistens auch online oder in der Informatikbibliothek verfügbar.

- Algorithmen und Datenstrukturen. Spektrum Lehrbuch. Thomas Ottmann, Peter Widmayer. (kostenlos elektronisch verfügbar über das Moodle der Vorlesung)
- Introduction to Algorithms. MIT Press. Cormen et al.
- Algorithm Design, Kleinberg, Tardos
- 101 Uses for an Algorithms Book: http://www.youtube.com/watch?v=WbfovBef1r4

Online Material

- Skiena's Algorithms Lectures: http://www.cs.sunysb.edu/~algorith/video-lectures/ (http://goo.gl/Le163)
- MITOpenCourseware Lectures: Introduction to Algorithms.
- D&A Übungsslides 2007 und 2008 von Michael Belfrage: http://www.belfrage.net/eth/

Praktische Programmieraufgaben

Sehr interessant um das in D&A gelernte Wissen anzuwenden, sind die Programmieraufgaben von verschiedenen Online-Judges.

- http://www.spoj.pl/ (sehr guter Online Judge, der verschiedene Programmiersprachen unterstützt)
- http://uva.onlinejudge.org/
- http://www.uwp.edu/sws/usaco (aufsteigende Schwierigkeit, auch mit Theorie)
- http://www.topcoder.com/ (sehr gute Tutorials: hhttp://goo.gl/Qauf8o)

Bei all diesen Seiten kann man selber programmierte Lösungen zu genau spezifizierten Problemen einschicken. Der Judge testet danach, ob der implementierte Algorithmus auch das tut, was er sollte. Der Stil der Aufgaben ist sehr ähnlich zu den praktischen Aufgaben auf den Übungsblättern, allerdings in diversen Schwierigkeitsstufen.

Zusatzmaterial zur Vorlesung

In der ersten Vorlesung wurde das *Travelling Salesman Problem* angesprochen und wie sich dessen Lösbarkeit in den letzten 30 Jahren veränderte. Dieser spannende Artikel aus der Praxis beschreibt, wie der Paketservice UPS das Problem angeht und welche wirtschaftlichen Konsequenzen eine bessere Lösung des Problems haben kann: http://www.wsj.com/articles/at-ups-the-algorithm-is-the-driver-1424136536 (http://goo.gl/svvvbd).

Das in der zweiten Vorlesung diskutierte *Maximum Subarray Problem* ist auf meiner Homepage als praktische Aufgabe verfügbar. Damit kannst du selbst testen wie sich die in der Vorlesung behandelten Ansätze mit ihren asymptotischen Laufzeiten von $\mathcal{O}(n^3)$, $\mathcal{O}(n^2)$, $\mathcal{O}(n \log n)$ und $\mathcal{O}(n)$ in der Praxis bemerkbar machen.

Asymptotische Analyse

Die Asymptotische Analyse ($\rightsquigarrow$ Oh-Notation) ermöglicht das maschinenunabhängige Vergleichen der Effizienz von Algorithmen. Sie ist grundlegend für D&A und die meisten (weiterführenden) Informatikvorlesungen. Sie legt die Aufmerksamkeit auf "the big picture": wie skaliert ein Algorithmus mit der Grösse der Eingabe.

obere Schranke $f(n) \in \mathcal{O}(g(n)) :\leftrightarrow \exists c > 0, n_0 \in \mathbb{N} : \forall n \geqslant n_0 : 0 \leqslant f(n) \leqslant c * g(n)$

untere Schranke $f(n) \in \Omega(g(n)) :\leftrightarrow \exists c > 0, n_0 \in \mathbb{N} : \forall n \geqslant n_0 : 0 \leqslant f(n) \geqslant c * g(n)$

$$\begin{array}{lcl} f(n) \in \mathcal{O}(g(n)) & \sim & f \leqslant g \\ f(n) \in \Omega(g(n)) & \sim & f \geqslant g \\ f(n) \in \Theta(g(n)) & \sim & f = g \\ f(n) \in o(g(n)) & \sim & f < g \\ f(n) \in \omega(g(n)) & \sim & f > g \end{array}$$

Man sieht häufig die Notation $f(n) = \mathcal{O}(g(n))$. Gemeint ist aber stets $f(n) \in \mathcal{O}(g(n))$. Für das Vergleichen eignen sich die Implikationen:

$$0 \leqslant \lim_{n\to\infty} \tfrac{f(n)}{g(n)} < \infty \implies f(n) \in O(g(n))$$

$$0 < \lim_{n\to\infty} \tfrac{f(n)}{g(n)} \leqslant \infty \implies f(n) \in \Omega(g(n))$$

$$0 < \lim_{n\to\infty} \tfrac{f(n)}{g(n)} < \infty \implies f(n) \in \Theta(g(n))$$

Die O-Notation wird häufig auch Landau-Notation genannt, nach dem gleichnamigen deutschen Mathematiker, der diese Notation vor gut hundert Jahren bekannt gemacht hat. Das O stand dabei für *"in der Ordnung von"*.

Wachstungsordnungen

$n^n >> n! >> c^n >> n^3 >> n^2 >> n^{1+\varepsilon} >> n \log n >> n >> \sqrt{n} >> \log^2 n >> \log n >> \frac{\log n}{\log \log n} >> \log \log n >> \alpha(n) >> 1$

Bemerkungen (think about it!)

- $\sqrt{n}$ wächst schneller als $\log n$
- Bei exponentiellen Laufzeiten gilt: $\mathcal{O}(2^n) \subsetneq \mathcal{O}(3^n)$, usw., d.h. diese Klassen werden unterschieden. Dasselbe gilt auch für z.B. $2^{\sqrt{n}}$ und $3^{\sqrt{n}}$.
- $\mathcal{O}(\log(n^n)) = \mathcal{O}(n \log n) = \mathcal{O}(\log(n!))$, aber $\mathcal{O}(n!) \neq \mathcal{O}(n^n)$, da $n!$ langsamer wächst als n^n.
- $\log_c n, \forall c > 0$ gehören alle derselben Klasse an und zwar $\mathcal{O}(\log n)$.

Links zur asymptotischen Analyse

- (Source of All Lies) http://en.wikipedia.org/wiki/Big_O_notation (http://goo.gl/rZUT)
- O-Notation: http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf (http://goo.gl/Wnhly)
- MIT Lecture: http://goo.gl/Ze8Br

Rekursionsgleichungen

Der Ansatz ist hier folgender:

1. "Erraten" der Lösung
2. Verifikation und Beweis durch Vollständige Induktion

Teleskopieren

Diese Technik wird gebraucht für Laufzeit- und Speicherplatzanalysen: Insbesondere bei *Divide-and-Conquer* Algorithmen. "Educated Guess"/Erraten der geschlossenen Form der Laufzeit. ($\rightsquigarrow$ beweise anschliessend mit Induktion)

Vorgehen:

- ca. 3 mal rekursiv einsetzen
- Pattern erkennen
- i-ten Schritt verallgemeinern
- Argument von T gleich 1 setzen, weil $T(1) = const.$ (Oft $i = \log_2(n)$ o. Ä.)

wichtige Formel: **geometrische Reihe**: $\sum_{k=0}^{n-1} q^k = \begin{cases} \frac{q^n - 1}{q-1} & q \neq 1 \\ n & q = 1 \end{cases}$

$\rightsquigarrow$ Beweis mit Induktion.

Vollständige Induktion ist ein sehr wichtiges Beweisverfahren. Wird euch immer wieder begegnen. Gliedert sich in:

- Verankerung
- Induktionsschritt
- Induktionshypothese

(Pseudo-)Code Analyse

Mögliche Strategie

- Genaues Analysieren des Codes
- Laufzeit als Formel (typischerweise mit Summenzeichen) schreiben (arithmetische Operationen zählen)
- Formel vereinfachen
- In Kategorie der $\mathcal{O}$-Notation einteilen (z.B. falls $f(n) > 0.1 * \log n$ und $f(n) < 4 * \log n$, dann gilt $f \in \Theta(\log n)$).
- $\rightsquigarrow$ Asymptotische Laufzeit

oder: Überschätzen/Unterschätzen

Bemerkungen:

- Alle arithmetischen Operationen haben Kosten 1
- Bei 'normal' verschachtelten Loops können diese Laufzeiten multipliziert werden
- Ist eine Loop A an eine Kondition gebunden ('if'), dann muss man sich überlegen, wie oft A total ausgeführt wird und multipliziert dies mit den Kosten der Loop A.

Beispiel 1

Betrachte die folgende allgemeine Schleife

from $i = 1$ **until** $i > n$
    $A(i)$
    $i = i + 1$

Die Laufzeit $T(n)$ des Codefragments hängt von der Laufzeit des Loop Bodys $A(i)$ ab

1. konstant: $T(A(i)) = c$

   Faustregel: immer von der tiefsten Verschachtelung her die Laufzeit bestimmen (von "innermost to outermost"). Wenn die Laufzeit des Loop Bodys unabhängig von der Laufvariablen ist (wie in diesem Fall), so kann man mit der Anzahl von Iterationen multiplizieren: Der Schleifenkörper wird n mal durchlaufen mit jeweils dem gleichen Aufwand, also

   $$T(n) = n \cdot c = \Theta(n)$$

2. $T(A(i)) = i$

   Ansonsten muss man über die Iterationen die Kosten summieren

   $$T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

   Unter anderem können hier folgende Summationsformeln wichtig sein

   - $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$
   - $\sum_{i=0}^{n-1} (2i + 1) = 1 + 3 + 5 + 7 + \ldots + (2n - 1) = n^2$
   - $\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2 = \frac{1}{6} n(n + 1)(2n + 1)$
   - $\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}$ (geometrische Reihe)

Übung: beweise obige Äquivalenzen!

3. $T(A(i)) = \frac{n}{2^i}$

   Mit obiger Summationsformel ist dies ebenfalls straight-forward

$$T(n) = \Theta\left(\sum_{i=1}^{n} \frac{n}{2^i}\right) = \Theta\left(n \sum_{i=1}^{n} \frac{1}{2^i}\right) = \Theta\left(n\left(1 - \frac{1}{2^n}\right)\right) = \Theta(n)$$

4. $T(A(i)) = \log i$

   Da $\log i < \log n$, ist kann man nach oben abschätzen $T(n) = \mathcal{O}(n \log n)$. Andererseits ist

$$T(n) = \log 1 + \ldots + \log \frac{n}{2} + \ldots + \log n \geqslant \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - 1) = \Omega(n \log n)$$

   Und daher ist $T(n) = \Theta(n \log n)$

Beispiel 2

Betrachte die folgende modifizierte allgemeine Schleife

from $i = 1$ **until** $i > n$
    $A(i)$
    $i = 2 \cdot i$

Die Laufzeit $T(n)$ des Codefragments hängt wiederum von der Laufzeit des Loop Bodys $A(i)$ ab

1. konstant: $T(A(i)) = c$

   Da die Laufzeit des Loop Bodys unabhängig von der Laufvariablen ist, kann man mit der Anzahl von Iterationen multiplizieren: Der Schleifenkörper wird $\log_2 n + 1$ mal durchlaufen: es gilt die folgende Invariante: in der k-ten Iteration ist $i = 2^k$. $k = 0, \ldots, \log_2 n$.

   Da wir bei jeder Iteration konstanten Aufwand haben ist die Gesamtlaufzeit

$$T(n) = (\log_2 n + 1) \cdot c = \Theta(\log n)$$

2. $T(A(i)) = i$

   Über die Iterationen die Kosten summieren: die Laufvariable i nimmt die Werte $2^0, 2^1, 2^2, \ldots, 2^{\log_2 n}$ an

$$T(n) = \sum_{k=0}^{\log_2 n} 2^k = \sum_{k=0}^{\log_2 n} 2^k = \frac{2^{\log_2 n + 1} - 1}{2 - 1} = \Theta(n)$$

3. $T(A(i)) = n - i$

$$T(n) = \sum_{k=0}^{\log_2 n} n - 2^k = \sum_{k=0}^{\log_2 n} (n - 2^k) = \sum_{k=0}^{\log_2 n} n - \left(\sum_{k=0}^{\log_2 n} 2^k\right)$$

$$= n(\log_2 n + 1) - \frac{2^{\log_2 n + 1} - 1}{2 - 1} = n \log_2 n - n + 1 = \Theta\left(n \log n\right)$$

4. $T(A(i)) = \log i$

$$T(n) = \sum_{k=0}^{\log_2 n} k = \frac{\log_2 n \cdot (\log_2 n + 1)}{2} = \Theta(\log^2 n)$$

Links

- Belfrage Slides (Fragestunde) http://www.belfrage.net/eth/da/pdf/fragesession.pdf (http://goo.gl/OvG5o)

Zusatzübungen

1. Geben Sie für die untenstehenden Funktionen eine Reihenfolge an, so dass Folgendes gilt: Wenn Funktion f links von Funktion g steht, so gilt $f \in \mathcal{O}(g)$.

$$\begin{array}{ccc} \sqrt{n} & n & 2^n \\ n \log n & n - n^3 + 7n^5 & n^2 + \log n \\ n^2 & n^3 & \log n \\ n^{\frac{1}{3}} + \log n & \log^2 n & n! \\ \ln n & \frac{n}{\log n} & \log \log n \\ (\frac{1}{3})^n & (\frac{3}{2})^n & 6 \end{array}$$

2. Finde eine geschlosse Form und beweise diese mit vollständiger Induktion (Annahme: n ist eine 2er Potenz)

$$T(n) = \begin{cases} 2T(n/2) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

3. Bestimme die asymptotische Laufzeit von folgenden Code-Fragmenten:

```
from i := 1 until i> n loop
        from j := 1 until j>i loop
                j := j+1
        end
        i := i*2
end
```

```
from i := 1 until i> n loop
        from j := i until j>n loop
                j := j+1
        end
        i := i*2
end
```

```
from i := 1 until i> n loop
        from j := 1 until j>i loop
                j := j*2
        end
        i := i+1
end
```

```
from i := 1 until i> n loop
        from j := i until j>n loop
                j := j*2
        end
        i := i+1
end
```

Brain Teaser: 100 Tiger und ein Schaf befinden sich auf einer magischen Insel die leider nur aus Grasslandschaft besteht. Tiger können sich zwar von Grass ernähren möchten aber dennoch lieber das Schaf verspeisen.

Nehme folgendes an:

1. Zu jeder Zeit kann jeweils nur *ein* Tiger ein Schaf fressen
2. Verspeist ein Tiger ein Schaf wird er selber zum Schaf
3. Alle Tiger sind äusserst schlau, absolut rational und sie wollen überleben

Wird das Schaf gefressen?