

Info II Klausurstoff

Andreas Zwinkau

July 30, 2005

Contents

1	Vorwarnung	3
1.1	Was drin ist	3
1.2	Was nicht drin ist	3
2	Prädikatenlogik	3
2.1	Negationsnormalform	3
2.2	Pränex Normalform	3
2.2.1	bereinigt	4
2.2.2	Pränex	4
2.3	Skolem Normalform	4
2.4	Klausel Normalform	4
2.5	Resolution	4
2.5.1	Vorgehen	5
2.5.2	erlaubte Operationen	5
2.6	allgemeinster Unifikator	5
3	Datentypen	6
3.1	ADT Abstrakte Daten Typen	6
3.1.1	FIFO (queue, Schlange)	6
3.1.2	FILO (Keller, Stack)	6
3.1.3	Liste	7
3.2	Hashing	7
3.2.1	Sondieren	7
3.2.2	Löschen eines Elements	8
3.3	Graphen	8
3.3.1	Speicherung	8
3.3.2	Suchalgorithmen	9
3.4	Bäume	9

3.4.1	Traversierung	9
3.4.2	Methoden	9
3.4.3	Rot-Schwarz-Bäume	10
4	Algorithmen	11
4.1	Dynamisches Programmieren	11
4.1.1	Wegfindung	12
4.2	Probabilistische Algorithmen	12
4.2.1	Las-Vegas Algorithmen	12
4.2.2	Monte-Carlo Algorithmen	12
4.2.3	Miller-Rabin-Test	12
4.2.4	Pollard Rho	13
4.3	Suchen von Zeichenketten	13
4.3.1	Naives Suchen	13
4.3.2	Knuth-Morris-Pratt	13
4.3.3	Boyer-Moore	14
4.4	Greedy-Algorithmen	14
4.5	Graphenalgorithmen	14
4.5.1	Warshall	14
4.5.2	Kruskal	15
4.5.3	Prim	15
4.6	Sortieralgorithmen	15
4.6.1	Mergesort	15
4.6.2	Insertsort	16
4.6.3	Quicksort	16
5	Algorithmenübersicht	16
6	Algorithmenanalyse	17
6.1	Überblick	17
6.2	O Notation	17
6.3	o Notation	17
6.4	Ω Omega Notation	17
6.5	ω omega Notation	18
6.6	Θ Theta Notation	18
6.7	Vom Programm zur Rekurrenz	18
6.8	Generierende Funktion	19
6.9	Mastertheorem	20
7	Haskell	20

8	Nachschatlag	20
8.1	Ehre wem Ehre gebührt	20
8.2	Changelog	21

1 Vorwarnung

Das ist eine Zusammenfassung für die Informatik 2 Klausur im Sommersemester 2005 an der Uni Karlsruhe. Die Vorlesung war von Prof. Calmet und Übungsleiter der Herr Eberhardt.

1.1 Was drin ist

Der ganze theoretische Kram wie Prädikatenlogik, Graphentheorie und all der Kram, von dem ich nicht weiß, wozu wir das genau lernen

Algorithmen zum Auswendiglernen, mit Übersichtstabelle für den Aufwand und einigen Implementierungen in Haskell, Python oder Pseudocode.

Jede Menge Fehler und vor allem sind die Ausdrücke oft nicht mathematisch korrekt. Ich habe immer mal wieder Definitionen weggelassen, der Übersichtlichkeit wegen. Man muss ja nicht immer hinschreiben, dass ϵ irgendeine positive Zahl ist.

1.2 Was nicht drin ist

Methoden, wie man die Algorithmen auf dem Papier berechnet. Dafür gibt es im Internet nette Illustrationen, die man nicht so schön und vor allem nur aufwendig in ein Dokument packen kann. Für Stringsearch und Co empfehle ich eine Google-suche.

Garantie, dass alles stimmt. Wenn du einen Fehler findest, darfst du ihn behalten oder mir (beza.le1@web.de) nen Verbesserungsvorschlag schicken.

2 Prädikatenlogik

2.1 Negationsnormalform

Alle vorkommenden Negationen stehen vor Atomen (Prädikaten), d.h. etwas wie $\neg(a \vee b)$ kommt nicht vor. Man kann diese Formel aber äquivalent umformen in $\neg a \wedge \neg b$.

2.2 Pränex Normalform

Normalerweise die erste Stufe in die man eine Formel umwandelt.

2.2.1 bereinigt

1. keine Variable kommt sowohl frei als auch gebunden vor

Wenn eine Variable frei und gebunden vorkommt, sind es eigentlich zwei Variablen. Deswegen wird die *gebundene* Variable ersetzt.

$$F = \forall x P(x) \wedge Q(x) \text{ wird zu } F[x/y] = \forall y P(y) \wedge Q(x)$$

2. hinter jedem Quantor steht eine andere Variable

Eine Variable kann doppelt frei bzw. doppelt gebunden vorkommen. Auch das soll vermieden werden.

$$G = \forall x P(x) \wedge \forall x Q(x) \text{ wird zu } G[x/y] = \forall y P(y) \wedge \forall x Q(x)$$

2.2.2 Pränex

bedeutet, dass alle Quantoren vorne stehen. Beispielsweise:

$$F = P(x) \wedge \forall y Q(y) \text{ wird zu } F' = \forall y P(x) \wedge Q(y)$$

2.3 Skolem Normalform

hat keinen Existenzquantor mehr. Die Idee ist, dass eine Existenzquantorvariable durch *all* die davorstehenden Allquantorvariablen festgelegt ist. Man spricht ja "Für alle x existiert ein y". Aus diesem Grund kann man y auch als Funktion $f(x)$ sehen. Kurz gesagt: $\forall x \exists y P(x, y)$ wird zu $\forall x P(x, f(x))$

2.4 Klausel Normalform

ist eine andere Schreibweise für eine Formel in konjunktiver Normalform¹.

KNF: $(a \vee b) \wedge (\neg a \vee b)$
Klauselform: $\{\{a, b\}, \{\neg a, b\}\}$

Die Klauselform wird vorallem für die Resolution verwendet.

2.5 Resolution

Die Frage ist, ob eine Formel überhaupt erfüllbar ist. Wenn man aus der Klauselform die leere Menge herleiten kann, bedeutet das, dass die Formel unerfüllbar ist. Kann man das nicht, sagt das gar nichts aus.

¹KNF: $(A \vee B \vee \dots) \wedge (\neg A \vee B \vee \dots) \dots$ Innen nur \vee , Dazwischen nur \wedge

Da man Elemente mehrfach verwenden darf, kann man theoretisch unendlich lange weiterprobieren. Deswegen gibt es auch keinen (garantiert terminierenden) Algorithmus, der eine Formel auf Unerfüllbarkeit testen kann.

Wenn man die Erfüllbarkeit einer Formel beweisen will, kann man per Resolution beweisen, dass die Negation unerfüllbar ist. Allerdings ist es aufwendig eine negierte KNF wieder zu einer KNF zu machen.

2.5.1 Vorgehen

- $F = ((Q(x) \vee \neg P(a)) \wedge \neg Q(x) \wedge P(y))$

Zur Vereinfachung nun die Skolem KNF als Klauselform schreiben.

- $\{\{Q(x), \neg P(a)\}, \{\neg Q(x)\}, \{P(y)\}\}$

Man kann die ersten Beiden vereinigen, da einmal Q und einmal $\neg Q$ vorkommt.

- $\{\{\neg P(a)\}, \{P(y)\}\}$

Nun kann man die Variable y auf a festlegen $[y/a]$, wieder vereinfachen und erhält dann die leere Menge

- $\{\}$

Das bedeutet die Formel ist unerfüllbar.

2.5.2 erlaubte Operationen

- Variable durch Konstante ersetzen
- Variable durch Funktion ersetzen
- Variable durch Variable ersetzen

2.6 allgemeinsten Unifikator

Wenn man zwei Formel "gleich macht", indem man substituiert nennt man das Unifikation. Beispiel:

$P(x, a)$ und $Q(f(y), z)$

Nun substituiert man $[x/f(y)]$ und $[z/a]$, dann sind P und Q unifiziert.

$P(f(y), a)$ und $Q(f(y), a)$

Man darf natürlich wieder nur die oben erwähnten erlaubten Operationen durchführen.

Der allgemeinste Unifikator bedeutet möglichst "wenig substituiert" oder formal μ is allgemeinsten Unifikator, wenn für alle Unifikatoren ν gilt, dass es einen Unifikator τ gibt, so dass $\nu = \tau \cdot \mu$ oder ganz stilistisch:

μ ist allgemeinsten Unifikator $\Leftrightarrow \forall \nu \exists \tau : \nu = \tau \cdot \mu$

3 Datentypen

3.1 ADT Abstrakte Daten Typen

3.1.1 FIFO (queue, Schlange)

First In - First Out

Methoden:

\perp	Konstruktor
insert	Einfügen eines Elements
tail	Entfernen des letzten (obersten) Elements
head	liefert das letzte (oberste) Element
length	Länge der Schlange

Axiome:

- $\text{head}(\text{insert}(x, \perp)) = x$
- $\text{head}(\text{insert}(x, S)) = \text{head}(S)$
- $\text{tail}(\text{insert}(x, \perp)) = \perp$
- $\text{tail}(\text{insert}(x, S)) = \text{insert}(x, \text{tail}(S))$
- $\text{length}(\perp) = 0$
- $\text{length}(\text{insert}(x, S)) = 1 + \text{length}(S)$

3.1.2 FILO (Keller, Stack)

First In - Last Out

Methoden:

\perp	Konstruktor
push	Einfügen eines Elements
pop	Entfernen des letzten (obersten) Elements
top	liefert das letzte (oberste) Element
length	Länge des Stacks

Axiome:

- $\text{top}(\text{push}(x, S)) = x$
- $\text{pop}(\text{push}(x, S)) = S$
- $\text{length}(\perp) = 0$
- $\text{length}(\text{push}(x, S)) = 1 + \text{length}(S)$

3.1.3 Liste

fast ein FIFO, aber keine Längenmethode, Man denke hierbei an eine verkettete Liste (linked list).

Methoden:

\perp	Konstruktor
cons	Einfügen eines Elements
head	liefert das letzte (obersten) Elements
tail	liefert den Rest ohne das letzte Element

Axiome:

- $\text{head}(\text{cons}(x, L)) = x$
- $\text{tail}(\text{cons}(x, L)) = L$

3.2 Hashing

Problem: Um einen Datensatz schneller zu finden, ordne ich ihm einen Index zu. Allerdings hat man normalerweise sehr viel weniger Daten als mögliche Varianten. Zum Beispiel hat mein Telefonbuch 200 Nummern, allerdings sind mit 6 Ziffern 10^6 Nummern möglich. Ich will aber nicht den Speicherplatz für 10^6 belegen.

Lösung: Hashfunktion. Man findet den Speicherplatz durch eine Funktion $h(x) = x \bmod 200$. So beschränkt man den Speicherplatz auf 200 Speicherplätze (0-199).

Folgeproblem: Was geschieht bei einer Kollision? $h(10) = 10 = h(210)$

1. An jedem Index ist eine Liste. So kann man an 10 und 210 an derselben Stelle ablegen
2. Suche deterministisch einen freien Platz. Beispielsweise einfach $+1 \bmod 200$. Es gibt verschiedene Verfahren für dieses "Sondieren".

3.2.1 Sondieren

Suchen eines freien Platzes

- Lineares Sondieren
 $+j \pmod{n}$
- Quadratisches Sondieren
 $+j^2 \pmod{n}$

- Doppeltes Hashen
 $+h(j) \pmod{n}$

Natürlich darf man noch Faktoren davorstellen etc.

3.2.2 Löschen eines Elements

Wenn man ein Element einfach entfernen würde, könnte beim Suchen vorzeitig aufgehört werden (*Oh, Feld leer, dann wäre das Gesuchte ja hier gewesen, also gibt es das Gesuchte nicht*)

Die Lösung ist ein zusätzliches Element "gelöscht" einzuführen.

3.3 Graphen

Wie speichert man einen Graphen im Computer?

3.3.1 Speicherung

	1	→	[2,5]
	2	→	[1,5,4,3]
<i>Adjazenzliste</i>	3	→	[2,4]
	4	→	[5,2,3]
	5	→	[1,2,4]

		1	2	3	4	5
<i>Adjazenzmatrix</i>	1	0	1	0	0	1
	2	1	0	1	1	1
	3	0	1	0	1	0
	4	0	1	1	0	1
	5	1	1	0	1	0

Speicherbedarf

- Adjazenzliste: $O(n^2)$ und $\Omega(n)$
- Adjazenzmatrix: $\Theta(n^2)$

Zugriff auf eine Kante

- Adjazenzliste: $O(n)$ und $\Omega(1)$
- Adjazenzmatrix: $\Theta(1)$

3.3.2 Suchalgorithmen

Es gibt die *Breitensuche* und die *Tiefensuche*. Kurz gesagt benutzt die Breitensuche einen FIFO und die Tiefensuche einen FILO.

- Breitensuche geht einen Graphen in Baumstruktur quasi Zeile für Zeile durch. Man ist aber natürlich nicht auf die Baumstruktur festgelegt.
- Tiefensuche geht einen Ast bis zum Ende und läuft dann zurück zur letzten Abzweigung. Ein Mensch würde so systematisch nach den Ausgang aus einem Labyrinth suchen.

3.4 Bäume

Wir schränken Bäume meist auf *binäre* Suchbäume ein, also Bäume die immer nur zwei Abzweigungen besitzen. Wenn man die Regel beachtet, das im linken Zweig nur Elemente sind die kleiner als die Wurzel und rechts nur größer als die Wurzel sind. Beschränkt sich die Suche auf $O(\log n)$.

3.4.1 Traversierung

Ausgabe eines ganzen Baums auf drei verschiedene Arten möglich.

In-Order	((links) wurzel (rechts))
Pre-Order	(wurzel (links) (rechts))
Post-Order	((links) (rechts) wurzel)

3.4.2 Methoden

- search
- minimum
- maximum
- successor
- predecessor
- insert
- delete

Löschen ist etwas kniffliger. Wenn wir den Knoten z löschen wollten, gibt es drei Fälle:

1. keine Kinder \rightarrow Parent auf Null zeigen lassen
2. ein Kind \rightarrow Parent auf Kind zeigen lassen
3. zwei Kinder \rightarrow finde einen Nachfolger y mit höchstens einem Kind; Ersetze z durch y ; Lösche den Nachfolger rekursiv.

Alle Operationen sind in $O(h)$, wobei h die Höhe des Baumes ist.

3.4.3 Rot-Schwarz-Bäume

Eine Methode um den Baum möglichst ausgeglichen zu halten, so dass er eine minimale Höhe hat.

Dazu bekommt jeder Knoten eine Farbe (Rot oder Schwarz) und die Regeln sind:

1. Jeder Knoten ist rot oder schwarz
2. Die Wurzel ist schwarz
3. Jedes Blatt ist schwarz
4. Wenn ein Knoten rot ist, dann sind beide Kinder schwarz
5. Von jedem Knoten zu allen Blättern darunter ist die Anzahl der schwarzen Blätter gleich. Insbesondere von der Wurzel.

Ein RSB mit n Knoten hat höchstens die Höhe $2 \log_2(n + 1)$

Durch die 5. Regel wird die Balance gefordert. Allerdings wird nun insert und delete aufwendiger $O(\log n)$, da der Baum möglicherweise durch Rotation neu ausbalanciert werden muss.

```

def rotate_left(Tree, x):
    """Tree holds the root node, x ist the node we want to shift left"""
    y = x.right
    ### changing the childs
    x.right = y.left
    y.left.parent = x
    ### changing the parents
    y.parent = x.parent
    if x.parent == Null:
        Tree = y
    elif x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y
    ### changing the x y relation
    y.left = x
    x.parent = y

...

```

4 Algorithmen

4.1 Dynamisches Programmieren

Zum finden *optimaler* Lösungen ähnlich dem divide'n'conquer. Die Idee ist, dass divide'n'conquer oft dieselbe Berechnung mehrmals ausführt, weil die Unterprobleme gleich oder ähnlich sind.

Einen dynamisches Programmieren Algorithmus findet man normalerweise in vier Schritten:

1. Wie sieht die optimale Lösung aus?
2. Definiere den Wert einer Lösung rekursiv
3. Berechne den Wert bottom-up
4. Verwende Zwischenergebnisse um die optimale Lösung zu finden

Typischerweise wendet man diese Methode auf Optimierungsprobleme an, wie zum Beispiel

4.1.1 Wegfindung

Man kann alle Punkte und Wege in einem Trellis (Tree-like-structure) darstellen und dann vereinfachen.

...

4.2 Probabilistische Algorithmen

4.2.1 Las-Vegas Algorithmen

liefern immer ein korrektes Ergebnis oder terminieren nicht.

Ein Beispiel wäre Random-Quicksort, wo das Pivotelement zufällig gewählt wird. Auch Pollard Rho

4.2.2 Monte-Carlo Algorithmen

kann für binäre Fragen verwendet werden (Ja oder Nein?) und *eine* der beiden Antworten ist *nicht sicher* korrekt.

Ein Beispiel dafür ist der Miller-Rabin-Test. Wenn dieser Test zurückgibt, die Zahl ist keine Primzahl, ist sie es sicher nicht. Wenn er aber Ja ausgibt, heißt das nur, dass es wahrscheinlich eine Primzahl ist.

Man kann Monte-Carlo Algorithmen wiederholen, um die Wahrscheinlichkeit einer richtigen Antwort zu erhöhen.

4.2.3 Miller-Rabin-Test

Testet ob Zahl n prim ist.

1. Finde s und d : $2^s * d = n - 1$
2. Wiederhole ein paar mal mal:
 - (a) Sei a irgendeine Zahl in $[0, n - 1]$
 - (b) Wenn $a^d \not\equiv 1 \pmod{n}$ und $a^{2^r d} \not\equiv -1 \pmod{n}$ für $r \in [0, s - 1]$ dann ist n nicht prim, Abbruch!
3. Falls wir hier angekommen sind ist n vermutlich prim

Miller-Rabin ist ein Monte-Carlo Algorithmus

4.2.4 Pollard Rho

Versucht einen Primfaktor zu finden. Leider terminiert Pollard nicht unbedingt.

```
def pollard(n):
    i = 1
    x = random.randrange(0,n)
    y = x
    k = 2
    while True:
        i = i + 1
        x = x**2 % n
        d = ggT(y-x, n)
        if d != 1 and d != n:
            return d
        if i == k:
            y = x
        else:
            k = 2*k
```

4.3 Suchen von Zeichenketten

Nette Beispiele:

- “example” in “Here is a simple example.”
- “ananas” in “Jan has some bananas.”
- “frisch” in “fischer fritze fischt frische fische.”

4.3.1 Naives Suchen

So wie wir Menschen es machen würden. Suche das erste “e”, dann schaue ob ein “x” danach kommt, ...

Aufwand: $O(nm)$

4.3.2 Knuth-Morris-Pratt

Die Idee ist, dass man ja nicht immer nur um eins weiterschieben muss, denn wenn man schon 4 Buchstaben verglichen hat, kann man ja vielleicht gleich 4 Buchstaben weiter fortfahren. Dazu legt man eine Tabelle an, in der man nachsehen kann, wie weit man jeweils schieben kann.

Aufwand: $O(n)$

4.3.3 Boyer-Moore

Der meistens effizienteste Algorithmus, der auch in den meisten Editoren für Suchen und Ersetzen verwendet wird.

Die geniale Idee ist, dass man rechts statt links mit vergleichen anfängt. Wenn der Buchstabe im Suchwort nicht vorkommt, kann man sofort um die Länge des Suchwortes weiterschieben.

Aufwand: $O(\frac{n}{m})$ (Best-Case)

4.4 Greedy-Algorithmen

Versuchen immer erst das größte Element.

Beispielproblem: Bankautomat gibt Geld aus in möglichst großen Scheinen.

```
def pay(amount):
    """amount is how much money we should pay"""
    out = []
    for note in [500,200,100,50,20,10,5]:
        while amount - sum(out) >= note:
            out.append(note)
            if sum(out) == amount:
                return out
    return "not possible"
```

Greedy ist der Algorithmus durch die Reihenfolge von [500,200,...]

4.5 Graphenalgorithmmen

4.5.1 Warshall

Findet die vollständige, transitiven Hülle eines Graphen.

Nach Cormen (Keine Verbindung bekommt den Wert ∞ !)

```
for k in [0..n-1]:
    for i in [0..n-1]:
        for j in [0..n-1]:
            A[i,j] = min( A[i,j], A[i,k]+A[k,j])
```

Anschaulich (auf dem Papier, mit 0 und 1)

```
for zeile in [0..n-1]:
    for spalte in [0..n-1]:
        for vergleich in [0..n-1]:
            if A[zeile,vergleich] == 1 and A[vergleich, spalte] == 1:
                A[zeile, spalte] := 1
```

4.5.2 Kruskal

Wenn wir einen ungerichteten Graphen mit gewichteten Kanten haben, berechnet Kruskal den minimal-spannenden Baum (Alle Knoten verbunden, aber keine Zyklen).

1. Kanten nach Gewicht sortieren
2. Nehme immer die kleinste Kante und füge sie hinzu, wenn es keinen Zyklus ergibt

Kruskal ist in $O(m \log m)$ (m Anzahl der Kanten), nicht-deterministisch (Wenn Kanten dieselbe Gewichtung haben) und ein Greedy-Algorithmus.

4.5.3 Prim

Löst dasselbe Problem wie Kruskal, ist aber für dicht besetzte Graphen effizienter.

1. Wähle einen beliebigen Knoten als Startpunkt
2. Füge immer eine Kante minimalen Gewichts hinzu, so dass der Baum um einen Punkt erweitert wird

Prim ist in $O(n \log n + m)$ (m Anzahl der Kanten, n Anzahl der Knoten)

4.6 Sortieralgorithmen

4.6.1 Mergesort

```
merge :: Ord a => [a] -> [a] -> [a]
merge a [] = a
merge [] a = a
merge (x:xs) (y:ys) = if x > y then y:(merge (x:xs) ys) else x:(merge xs (y:ys))
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [a] = [a]
mergesort xs = merge (mergesort left) (mergesort right) where
    (left, right) = splitAt (div (length xs) 2) xs
```

4.6.2 Insertsort

```

insert :: Integer -> [Integer] -> [Integer]
insert a [] = [a]
insert a (x:xs) = if a > x then x:(insert a xs) else a:x:xs
insertsort :: [Integer] -> [Integer]
insertsort [] = []
insertsort (x:xs) = insert x (insertsort xs)

```

4.6.3 Quicksort

```

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (pivot:rest) = quicksort [y | y <- rest, y < pivot]
                        ++ [pivot] ++
                        quicksort [y | y <- rest, y >= pivot]

```

Quicksort ist im Worst-Case $O(n^2)$, aber meistens $O(n \log n)$ und deswegen der effizienteste Sortieralgorithmus für große n .

5 Algorithmenübersicht

Name	terminiert	Aufwand
Quicksort	Ja	$O(n \log n)$ (Worst-Case $O(n^2)$)
Mergesort	Ja	$O(n \log n)$
Heapsort	Ja	$O(n \log n)$
Insertsort	Ja	$O(n^2)$
Bubblesort	Ja	$O(n^2)$
Huffman	Ja	$O(n \log n)$
Binarysearch	Ja	$O(\log n)$
Hashsearch	Ja	$O(n)$
Breitensuche	Ja	$O(n)$
Tiefensuche	Ja	$O(n)$
Knuth-Morris-Pratt	ja	$O(n)$
Boyer-Moore	Ja	$O(n) \rightarrow O(\frac{n}{m})$
Miller-Rabin	Ja	$O(\log^3 n \lg \frac{1}{\epsilon})$
Warshall	Ja	$O(n^3)$
Kruskal	Ja	$O(m \log m)$
Prim	Ja	$O(n \log n + m)$
Pollard ρ	Meistens	$O(\sqrt[4]{n})$

6 Algorithmenanalyse

6.1 Überblick

Sei $f \in O(g(n))$ bzw. $o, \Omega, \omega, \Theta$ c eine feste Konstante x eine beliebige Zahl

Notation	Definition	Formel	Umgangssprachlich
O	asymptotische obere Schranke	$c * f < g$	Aufwand ist kleiner als das
o	asymptotisch vernachlässigbar	$x * f < g$	wie O, nur für alle Faktoren
Ω	asymptotische untere Schranke	$c * f > g$	Aufwand ist größer als das
ω	asymptotisch dominant	$x * f > g$	wie Ω , nur für alle Faktoren
Θ	asymptotisch scharfe Schranke	$c * f = g$	f ist ziemlich genau g

Nicht vergessen, dass immer noch ein Faktor davorstehen kann und alles erst ab einem bestimmten n_0 gilt.

6.2 O Notation

$\exists c : c * f \geq g \Leftrightarrow f \in O(g(n))$

f ist eine Obere Schranke für g. Wenn ich also einen Algorithmus g habe und $f = n^2 \in O(g(n))$ bedeutet das, dass mein Algorithmus quadratisch skaliert.

Wenn mein Webserver z.B. 100 Anfragen pro Sekunde verarbeiten kann und ich weiß, dass er quadratisch skaliert, bedeutet das, dass er für 10mal soviel Anfragen 100mal Zeit braucht. Also 1000 Anfragen 100 Sekunden.

Vereinfachungen:

$$O(3n^2 + 20n + 10000017) = O(n^2)$$

6.3 o Notation

Eine Art verschärftes O. Bei dem Beispiel oben muss die Konstante $c > 3$ sein, damit $c * n^2 > 3n^2 + 20n + 10000017$. Wenn aber $f \in o(g(n))$ ist, dann kann jede Zahl Konstante sein, vor allem jede beliebig Kleine. Beispiel:

$$n^2 \in o(3n^3 + 20n + 10000017)$$

Egal wie klein c ist, ab einem bestimmten n_0 ist $f > g$.

6.4 Ω Omega Notation

Während O die obere Grenze darstellt ist Ω die untere Grenze. Meine Algorithmus g ist also aufwendiger als f.

6.5 ω omega Notation

ω ist für Ω was o für O ist. Beispielsweise ist $n \in \omega(n^2)$. Egal welche Konstante ich davor setze, g ist irgendwann größer.

6.6 Θ Theta Notation

Kurz: $\Theta = O \cap \Omega$

Das bedeutet, dass mein Algorithmus g so stabil ist, dass obere und untere Schranke in derselben Größenordnung liegen.

Für Quicksort ist $\Theta = \{\}$, denn Quicksort variiert zwischen n^2 und $n \log n$.

6.7 Vom Programm zur Rekurrenz

Problem: Man hat eine Funktion die rekursiv definiert ist und hätte gerne eine geschlossene Form davon.

Meist kommt man mit Raten weiter. Dabei hilft die Überlegung, wie der Zuwachs ist, also quasi die Ableitungen.

Sei $f_n := f_{n-1} * 2 + 1, f_0 := 1$

Index	Funktionswert	Zuwachs	Zuwachs des Zuwachses
0	1	+2	*2
1	3	+4	*2
2	7	+8	*2
3	15	+16	*2
4	31	+32	...
5	63

Die vierte Spalte ist also immer der Faktor 2, vielleicht wäre dann der Ansatz $a2^{bn+c} + d$ nicht dumm?

Nun wäre ein Gleichungssystem angebracht, um a,b,c,d auszurechnen. Dieses Beispiel funktioniert sogar mit einem LGS mit $f_n = a2^n + b$, aber das ist nicht allgemeingültig.

$$f_0 = 1 = a2^{0*n+c} + d \quad (1)$$

$$f_1 = 3 = a2^{1*b+c} + d \quad (2)$$

$$f_2 = 7 = a2^{2*b+c} + d \quad (3)$$

$$f_3 = 15 = a2^{3*b+c} + d \quad (4)$$

Oder einfach nochmal scharf hinsehen und man erkennt, dass es wohl $2^{n+1} - 1$ sein dürfte. Das muss man jetzt nur noch per Induktion beweisen.

6.8 Generierende Funktion

Eine andere Methode um die geschlossene Form zu bekommen ist, die Funktion f in eine Reihe zu packen. Das ganze funktioniert in vier Schritten.

Sei $f_n := f_{n-1} * 2 + 1, f_0 := 1$

1. Wir wollen nur eine Gleichung und das unter der Nebenbedingung, dass alle $f_n = 0$ sind, für negative n .

Der Trick ist zusätzliche Summanden einzuschleusen. $f_n = 2f_{n-1} + 1 + [f_0 = 1]$. Die eckige Klammer sei definiert als eine Funktion die am Punkt $0 = 1$ und sonst Null ist. Voila.

2. Multipliziere mit z^n und summiere über alle n . Das ist nur Schreibweise, also nicht zu mathematisch genau nehmen.

$$G(z) := \sum_n f_n z^n = \underbrace{\sum_n f_{n-1} z^n}_{=z * \sum_n f_{n-1} z^{n-1}} + \underbrace{\sum_n z^n}_{=\frac{z}{1-z}} + \sum_n [f_0 = 1] * z^n$$

Oder anders

$$G(z) = z * G(z) + \frac{z}{1-z} + 1$$

3. Löse nach $G(z)$ auf.

$$G(z) = \frac{\frac{z}{1-z} + 1}{1-z} = \frac{1}{(1-z)^2}$$

4. Schreibe $G(z)$ als Potenzreihe, dann sind die Koeffizienten die geschlossene Form von f_n . Tja das ist der komplizierte Teil, denn meistens ist das nicht so leicht wie es sich anhört.

$$G(z) = \left(\frac{1}{1-z}\right)^2 = \left(\sum_n z^n\right)^2 = \text{deep magic stuff here} = \sum_n (2^{n+1} - 1) z^n$$

So jetzt erstmal ne Pause ...

6.9 Mastertheorem

Ein Spezialfall. Wenn die Rekurrenz folgendermaßen aussieht:

$$T(n) = aT(n/b) + f(n)$$

wobei $a \geq 1$, $b > 1$ und $f(n)$ eine asymptotisch positive Funktion. Sei $d := \log_b a$. Jetzt gibt es drei Fälle bezüglich des f

1. $f \in O(n^{d-\epsilon})$ also f polynomial kleiner als n^d
dann ist $T(n) \in \Theta(n^d)$
2. $f \in \Theta(n^d)$
dann ist $T(n) \in \Theta(n^d \lg n)$
3. $f \in \Omega(n^{d+\epsilon})$ also f polynomial größer als n^d
dann ist $T(n) \in \Theta(f(n))$

Das Mastertheorem funktioniert nicht immer, denn es besteht eine Lücke zwischen den Fällen. Falls f dort hineinfällt, kann die Methode nicht angewandt werden.

7 Haskell

Naja ... äh ... <http://haskell.org/learning.html>

8 Nachschlag

8.1 Ehre wem Ehre gebührt

Hier will ich versuchen All die zu nennen, die mir geholfen haben (Meist ohne es zu wissen)

- Google und das Internet, speziell Wikipedia
- unika04.de/forum
- \LaTeX Mitschrieb von Joachim Breitner initiiert (<http://lkwiki.nomeata.de/>)
- Introduction to Algorithms alias “Der Cormen” (ISBN: 81-203-2141-3)
- meine Schwester mit einem motivierenden Telefonat

8.2 Changelog

Danke an Daniel Morlock! Das mit der Resolution war wirklich einfach falsch. Man darf keine Prädikate in derselben Klausel zusammenfassen. Jetzt stimmt's aber. (2.5.1)

Dank an Subby für die $\langle \leftrightarrow \rangle$ Dreher (6.1), taurus84 für den `head↔tail` Dreher (3.1.3), Hinni für den Hinweis mit dem Faktor-statt-Differenz (6.7)

Diesmal gibts Dank dan_f diese Klammern: (8.2) und Dank beerman ein korrektes Beispiel (6.3)

Ein weiteres Dankeschön an induktiv_gehärtet für einen Rechtschreibfehler (2.3), Stefan Widmann, dass ein Stack keine Schlange ist (3.1.1), David Förster für zwei Klarstellungen (3.2, 6.3)

Admin (Unika04 Forum) hat das mit nem netten LGS gemacht, die Idee fand ich gut. (6.7)