# Elegante Algorithmen

## Vortragsausarbeitungen des Proseminars

## Fakultät für Informatik, TU Dortmund

Prof. Dr. Sven Rahmann

mit Beiträgen von
Martin Cmok
Thorben Seeland
Fabian Bienek
Daniel Hüsmert
Paul Ruppersberger
Janet Fiedler
Karl Kanafa
Mark Brockmann
Igor Blyufshteyn
Tobias Steinrücken
Dino Menges
Daniel Wulfert
Bernd Hesse

Sommersemester 2008
Entwurf vom 26. Januar 2009

# Inhaltsverzeichnis

**1 Der ggT Algorithmus** **1**
1.1 Einleitung . . . . . 1
1.2 Methoden zur Berechnung des ggT . . . . . 2
1.3 Implementierung des ggT in verschiedenen Programmiersprachen . . . . . 6
1.4 Fazit . . . . . 11
1.5 Quellenangaben . . . . . 11

**3 Das Eleganteste Quicksort** **13**
3.1 Motivation . . . . . 13
3.2 Quicksort entwickeln . . . . . 14
3.3 Quicksort analysieren . . . . . 19
3.4 Fazit . . . . . 28

**4 Elegante Tests – Binäre Suche** **31**
4.1 Einleitung - Was ist eigentlich elegant? . . . . . 31
4.2 Elegante Tests . . . . . 32
4.3 Smoke Tests . . . . . 33
4.4 Boundary Tests . . . . . 33
4.5 Ausführliche Tests . . . . . 36
4.6 Performance-Tests . . . . . 39
4.7 Zusammenfassung der Tests . . . . . 40

**5 Kolninearität von 3 Punkten** **41**
5.1 Problemstellung . . . . . 41
5.2 Algorithmus 1 . . . . . 41
5.3 Algorithmus 2 . . . . . 43
5.4 Algorithmus 3 . . . . . 44
5.5 Algorithmus 4 . . . . . 46
5.6 Quellenverzeichnis . . . . . 48

**6 Anzahl der 1-Bits in einem Integer 49**
6.1 Einleitung . . . 49
6.2 Erster Ansatz . . . 50
6.3 Divide-And-Conquer Ansatz . . . 51
6.4 Rechnen mit PopCounts . . . 53
6.5 PopCount eines Arrays . . . 55

**7 Aho-Corasick-Automat 57**
7.1 Einleitung . . . 57
7.2 Aho-Corasick-Algorithmus . . . 58
7.3 Laufzeit . . . 60
7.4 Einsatzgebiete des Aho-Corasick-Algorithmus . . . 61
7.5 Quellen . . . 61

**8 Bit-paralleles Pattern Matching 63**
8.1 Einleitung . . . 63
8.2 Bit-parallele Operationen . . . 63
8.3 Grundlegende Konzepte . . . 64
8.4 Brute-Force Algorithmus . . . 64
8.5 Shift-And/Or Algorithmus . . . 65
8.6 Backward Nondeterministic Dawg Matching . . . 68
8.7 Quellen . . . 71

**9 A Regular Expression Matcher 73**
9.1 Einleitung . . . 73
9.2 Einfürung - Definition . . . 73
9.3 Syntax für reguläre Ausdrücke . . . 74
9.4 Der Algorithmus von Rob Pike . . . 76
9.5 Erweiterungen und Verbesserungen . . . 81
9.6 Fazit . . . 83

**10 Approximative Teilstringsuche 85**
10.1 Fragestellung . . . 85
10.2 Definition des Problems . . . 86
10.3 Abstandsfunktionen . . . 86
10.4 Idee für die Algorithmen . . . 87
10.5 Algorithmus von Sellers . . . 88
10.6 Laufzeit . . . 89
10.7 Idee für die Verbesserung . . . 89
10.8 Beispiel Algorithmus von Ukkonen . . . 89
10.9 Laufzeit . . . 91
10.10Quellen . . . 91

**11 Dictionarys and Hashmaps in Python 93**
11.1 Anforderungen . . . 93
11.2 Implementierung mit einfachen Datentypen . . . 93
11.3 Hashmaps . . . 95
11.4 Analyse . . . 97

11.5 Quellen . . . 98

**13 Mehrdimensionale Iteratoren** **99**
13.1 Definition . . . 99
13.2 Aufbau eines Iterators . . . 100
13.3 Vorteile des Iterators . . . 105
13.4 Broadcasting . . . 105

**14 Schnelle Matrixmultiplikation** **107**
14.1 Einleitung . . . 107
14.2 Iterative Berechnung . . . 108
14.3 Rekursive Berechnung . . . 109
14.4 Der Strassen-Algorithmus . . . 111
14.5 Nicht quadratische Matrizen . . . 114

**15 Suffixarray-Konstruktion** **117**
15.1 Einführung . . . 117
15.2 Konstruktion von Suffixarrays . . . 118
15.3 Quellen . . . 125

**99 Hinweise zur richtigen Benutzung von LaTeX** **127**
99.1 Einleitung . . . 127
99.2 Häufig gemachte Fehler . . . 127

**Literaturverzeichnis** **131**

# Vorbemerkungen

Dieses Dokument enthält die Vortragsausarbeitungen des Proseminars Elegante Algorithmen, das im Sommersemester 2008 an der Fakultät für Informatik an der TU Dortmund von Prof. Sven Rahmann veranstaltet wurde.

Ziel war es, überraschend einfache und effiziente Algorithmen zur Lösung elementarer häufig auftretender Probleme kennenzulernen und vorzustellen. Die Themenauswahl ist zu einem großen Teil subjektiv, war aber vor allem inspiriert durch das folgende Buch.

> Andy Oram & Greg Wilson
> Beautiful Code
> O'Reilly, 2007

Als Veranstalter hoffe ich, dass alle Teilnehmer viel gelernt haben und Spaß bei der Arbeit hatten.

KAPITEL 1

# Der ggT Algorithmus

**Ausarbeitung von Martin Cmok**

## 1.1 Einleitung

**Der euklidische Algorithmus** ist ein Algorithmus, der den größten gemeinsamen Teiler aus zwei natürlichen Zahlen $a$ und $b$ berechnet, welche sowohl $a$ als auch $b$ teilt. Das Verfahren ist nach dem griechischen Mathematiker Euklid benannt, der schon vor ber 2000 Jahren die Berechnung des größten gemeinsamen Teilers in seinem Werk *Die Elemente* beschrieben hat und gehört somit wohl zu den ältesten bekannten Algorithmen. Es wurde von ihm als geometrisches Problem formuliert, auf der Suche nach einem gemeinsames Maß beobachtete er, dass wenn man wiederholt die kleinere von der größeren zweier Strecken abzieht, letztendlich das gemeinsame Maß übrig bleibt. Daher wurde dieser klassische Euklidische Algorithmus auch als Wechselwegnahme bezeichnet.

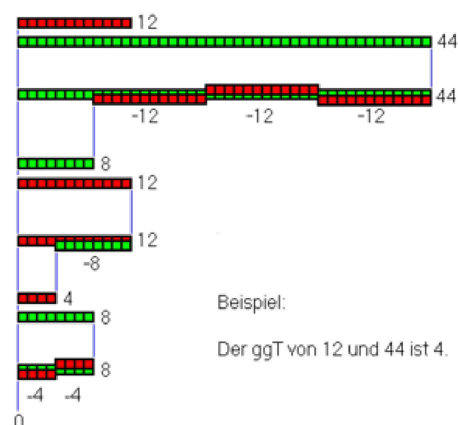



Abbildung 1.1: Beispiel ggT

**Der klassische Euklidische Algorithmus in Java**

```
public static int Euklid(int x, int y) {
 while (x!=y)
  if (x>y)
   x=x-y;
  else
   y=y-x; }
 return x;
}
```

Es gibt heutzutage verschiedene Varianten den größten gemeinsamen Teiler zu berechnen, neben dem klassischen Euklidischen Algorithmus gibt es noch die **Primfaktorzerlegung** , den **erweiterten Euklidischen Algorithmus** und den **Steinschen Algorithmus**. Alle Methoden errechnen zwar den richtigen Wert unterscheiden sich aber deutlich in der Effizienz der Berechnung, so ist z.B. der Steinsche Algorithmus effizient, weil er für eine Berechnung in einem Computer konzipiert wurde.

## 1.2 Methoden zur Berechnung des ggT

### 1.2.1 Primfaktorzerlegung

Der größte gemeinsame Teiler zweier Zahlen kann aus ihren Primfaktorzerlegungen ermittelt werden. Eine Primfaktorzerlegung ist, wenn man eine natürliche Zahl nur als Produkt von Primzahlen schreibt.

**Beispiel** Berechnung des größten gemeinsamen Teilers von 14 von 24

$$Ggt(24, 14)$$
$$24 \div 2 = 12$$
$$24 = \{2 \cdot 2 \cdot 2 \cdot 3\}$$
$$12 \div 2 = 6$$
$$6 \div 2 = 3$$
$$24 \div 3 = 8$$

$$14 \div 2 = 7$$
$$14 = \{2 \cdot 7\}$$

Somit ist die Zahl 2 die größte Zahl, die in beiden Teilmengen vorkommt. Die Zerlegung einer natürlichen Zahl ist sehr ineffizient, weil es sehr zeitaufwendig ist, diese zu berechnen. Es gibt bis heute keine effizienten Algorithmen für die Primfaktorzerlegung, der NFS Algorithmus

ist bislang der schnellste allerdings hat auch dieser eine exponentielle Laufzeit. Somit ist die Primfaktorzerlegung nur bei kleineren Zahlen zur Bestimmung des größten gemeinsamen Teilers geeignet.

### 1.2.2 Der moderne Euklidische Algorithmus

Der Nachteil bei **klassischen Euklidischen Algorithmus** ist eine häufige wiederholte Benutzung von Subtraktionsschritten, wenn die Differenz der zu berechnenden Zahlen sehr groß ist. Um dies zu umgehen ersetzte man die Subtraktionsschritte durch eine Division mit Rest. Weil sich die Zahlen in jedem Schritt mindestens halbieren, ist der **moderne Euklidische Algorithmus** auch bei großen Zahlen extrem schnell, außerdem ist man auch im Stande Euklidische Ringe damit zu berechnen.

#### Rechenbeispiel

$$Ggt(555, 432)$$

$$555 = 1 \cdot 432 \; Rest \; 123$$

$$432 = 3 \cdot 123 \; Rest \; 63$$

$$123 = 1 \cdot 63 \; Rest \; 60$$

$$63 = 1 \cdot 60 \; Rest \; 3$$

$$60 = 20 \cdot 3 \; Rest \; 0$$

Man dividiert solange die größere durch die kleinere Zahl und anschließend jeweils die kleinere Zahl durch den entstandenen Rest bis kein Rest mehr übrig bleibt. Falls bei diesem Verfahren der Rest 1 übrig bleibt, sind die beiden Zahlen teilerfremd.

$$a = q_1 * r_0 + r_1$$

$$r_0 = q_2 * r_1 + r_2$$

$$...$$

$$...$$

$$r_n - 1 = q_n + 1 + r_n$$

$$ggt(a, b) = r_n$$

Wenn bei der Eingabe zwei aufeinander folgende Fibonacci Zahlen gewählt werden ist die Laufzeit im worst-case-Fall $\theta(logab)$ Bei aufeinander folgenden Fibonacci Zahlen ergibt sich als Rest immer die nächst kleinere Fibonacci-Zahl. Da die für die Division zweier Zahlen benötigte Zeit wiederum von der Anzahl der Ziffern der Zahlen abhängt, ergibt sich eine tatsächliche Laufzeit von $O(logab)^2$).

Der Divisor (r) der letzten Division ist dann der größte gemeinsame Teiler.

#### Der Moderne Euklidische Algorithmus in Java

```
public class Euklid(int x,int y) {
 int r;
 while (y!=0) {
  r= x mod y;
  x = y;
  y = r;
 return x; }
 }
```

### 1.2.3 Der erweiterte Euklidische Algorithmus

Der **erweiterte Euklidische Algorithmus** berechnet neben dem größten gemeinsamen Teiler zweier natürlicher Zahlen $a$ und $b$ noch zwei ganze Zahlen $s$ und $t$, die folgende Gleichung erfüllen:

$$ggt(a, b) = s * a + t * b$$

Mit dem **erweiterten Euklidischen Algorithmus** lassen sich so die Inversen Elemente bestimmen, die vor allem für die Berechnung von Restklassenringen notwendig sind. In Restklassen werden ganze Zahlen mit gleichem Rest bei der Division durch $n$ zusammengefasst. Die Restklassen bilden mit der Multiplikation und Addition dann den Restklassenring.

#### Rechenbeispiel

$$99 = 1 * 78 + 21$$

$$78 = 3 * 21 + 15$$

$$21 = 1 * 15 + 6$$

$$6 = 2 * 3 + 0$$

Die Zahl 3 ist der gesuchte gemeinsame Teiler von 99 und 78. Löst man den euklidischen Algorithmus rückwärts auf, so erhält man immer eine Darstellung des $ggT(a, b)$ als Linearkombination von $a$ und $b$.

$$3 = 15 - 2 * 6$$

$$= 15 - 2 * (21 - 1 * 15) = 3 * 15 - 2 * 21$$

$$= 3 * (78 - 3 * 21) - 2 * 21 = 3 * 78 - 11 * 21$$

$$= 3 * 78 - 11 * (99 - 1 * 78)$$

#### Der erweiterte Euklidische Algorithmus in Java

```
public class ExtendedEuklid
{
 public int g, u, v;
 public ExtendedGcd(int a, int b)
  {
   ExtendetEuklid(a, b);
  }
   public void ExtendetEuklid(int a, int b)
  {
     int q, r, s, t;
     u=t=1;
     v=s=0;
     while (b>0)
     {
      q=a/b;
      r=a-q*b; a=b; b=r;
      r=u-q*s; u=s; s=r;
      r=v-q*t; v=t; t=r;
      }
      g=a;
    }
```

### 1.2.4 Der Steinsche Algorithmus

Der **Steinsche Algorithmus** dient ebenfalls der Berechnung des größten gemeinsamen Teilers. Der Algorithmus wurde 1967 von Josef Stein entwickelt und nutzt die schnelle Berechnung einer Division durch 2 in einem Rechner aus. Daher wird dieser Algorithmus auch **Binärer Euklidischer Algorithmus** genannt. Dabei nutzt er folgende Rechenregeln aus :

Ggt($a$,$b$) = 2 · ggt($a/2$ , $b/2$) , falls $a$ und $b$ gerade
Ggt($a$,$b$) = ggt($a/2$ , $b$) , falls $a$ gerade und $b$ ungerade
Ggt($a$,$b$) = ggt(($a$-$b$)/2 , $b$) , falls $a$ und $b$ ungerade

Der Algorithmus ersetzt also zeitaufwändige Divisionen durch Divisionen durch 2 und diese sind auf Rechnern einfach durchzuführen. Somit ist der Steinsche Algorithmus effizienter gegenüber dem Euklidischen Algorithmus und kommt auf eine Laufzeit von $O(log\overline{2}(a+b))$:

#### Beispiel in Java

```
Public class EuklidBin(int x; int y)
{ int d = 1;
  while(x mod 2 == 0)&&(y mod 2 == 0)
  {
   x=x/2; y=y/2; d=2*d;
  }
```

```
    while(x!= 0)
    {
     while(x mod 2 == 0) x = x/2;
     while(y mod 2 == 0) y = y/2;
      if(x < y) swap(x,y)
       x= x - y;
     }
     return(y*d);
  }
```

#### Rechenbeispiel

- (a) setze $m = a; n = b$
- (b) dividiere $m$ und $n$ durch 2 solange, bis eine der beiden Zahlen ungerade ist. Die Zahl der Divisionsschritte sei $k$.
- (c) Falls $n$ gerade ist, vertausche $m$ und $n$.
- (d) dividiere $m$ durch 2, bis $m$ ungerade ist
- (e) ist $m - n$, so vertausche diese Zahlen setze $m = m - n$
- Nach Ablauf erhält man $ggT(a, b) = n * 2^k$.

| | ggT(14540,2480) |
|---|---|
| 1. | (a)14540,2480 → 3660,1120 → 1830,560 (b)915,280 |
| 2. | (c)280,915 → 140 → 70 → 35 |
| 3. | (d)915,35 |
| 4. | (e)880,35 |
| 5. | (c)440 → 220 → 110 |
| 6. | (e)55,35 → 20,35 |

Der größte gemeinsame Teiler ist somit $5 \cdot 24 = 80$

## 1.3 Implementierung des ggT in verschiedenen Programmiersprachen

### 1.3.1 Programmiersprache: ADA

**ADA** ist eine strukturierte Programmiersprache mit einer strengen Typisierung. Sie wurde Anfang 1975 entwickelt und ist von ihrer Struktur ähnlich zur Programmiersprache Pascal aber auch strenger in der Programmierung. Aufgrund der hohen Anforderung, die Compiler erfüllen müssen, hat sie sich vor allem in sicherheitskritischen Bereichen durchgesetzt.

```
PROCEDURE do_ggT IS

  FUNCTION ggT (x: Integer; y: Integer) Return Integer IS
   BEGIN
```

```
    IF x \leq 0 THEN
      Return y;
    ELSE
      Return ggT (y MOD x, x);
    END IF;
   END ggT;

 Zahl1, Zahl2: Integer;
BEGIN
  Put_Line ("Berechnung des ggt.");
  Put ("Erste Zahl eingeben: ");
  Get (Zahl1);
  Put ("Zweite Zahl eingeben: ");
  Get (Zahl2);
  Put ("Der ggT ist: ");
  Put_Line (ggT (Zahl1, Zahl2));
END do_ggT;
```

### 1.3.2 Programmiersprache: COBOL

**COBOL** ist eine Programmiersprache, die in der Frühzeit der Computerentwicklung 1960 entstanden ist und bis heute verwendet wird. Der Stil dieser Programmiersprache ist wortreich und stark an die natürliche Sprache angelehnt. **COBOL** entstand aus dem Wunsch heraus, eine problemorientierte Sprache fr die Erstellung von Programmen für den betriebswirtschaftlichen Bereich zu haben. Die Programmierung kaufmännischer Anwendungen unterscheidet sich von technisch-wissenschaftlichen Anwendungen durch die Bearbeitung von groen Datenmengen anstatt umfangreicher Berechnungen.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ZAHL-1 PICTURE IS 9(5).
01 ZAHL-2 PICTURE IS 9(5).
01 HILFSVARIABLEN.
  05 X PICTURE IS 9(5).
  05 Y PICTURE IS 9(5).
  05 REST PICTURE IS 9(5).
  05 DIV-ERG PICTURE IS 9(5).
  05 MULT-DIV-ERG PICTURE IS 9(5).
01 ERGEBNIS PICTURE IS 9(5).

PROCEDURE DIVISION.
EINGABE.
  DISPLAY ggT..
  DISPLAY Erste Zahl eingeben:.
  ACCEPT ZAHL-1.
  IF ZAHL-1 IS LESS THAN 1
    THEN PERFORM ZAHL-ZU-KLEIN.
```

```
  DISPLAY Zweite Zahl eingeben:.
  ACCEPT ZAHL-2.
  IF ZAHL-2 IS LESS THAN 1
    THEN PERFORM ZAHL-ZU-KLEIN.
  MOVE ZAHL-1 TO X.
  MOVE ZAHL-2 TO Y.

BERECHNUNG.
 IF X IS EQUAL TO 0 THEN
   MOVE Y TO ERGEBNIS
   GO TO FERTIG.
REST = Y - (Y / X) Gerundet * X.
 DIVIDE X INTO Y GIVING DIV-ERG ROUNDED.
 MULTIPLY DIV-ERG BY X GIVING
   MULT-DIV-ERG.
SUBTRACT MULT-DIV-ERG FROM Y
   GIVING REST.
MOVE X TO Y.
MOVE REST TO X.
GO TO BERECHNUNG.

FERTIG.
  DISPLAY Der ggT von ,
    ZAHL-1,  und , ZAHL-2,  ist: , ERGEBNIS.
  STOP RUN.

ZAHL-ZU-KLEIN.
  DISPLAY Die Zahl soll grer als 0 sein.
  STOP RUN.
```

### 1.3.3 Programmiersprache: Fortran

**Fortran** gilt als die erste realisierte höhere Programmiersprache und wurde bereits zwischen 1954 und 1957 entwickelt. **FORTRAN** wurde ursprünglich vor allem für die Lösung mathematischer Berechnungen im naturwissenschaftlich-technischen Bereich entwickelt, seit Fortran 77 kann die Sprache jedoch fast universell eingesetzt werden. **Fortran** ist insbesondere für die Programmierung numerischer Verfahren geeignet, aufgrund einer schnellen Programmausführung und einer effektiven Speichernutzung. Die Sprache verfügt über mächtige Elemente zur Feld- und Matrizen- Verarbeitung. Fortran-Compiler sind in der Lage, Programme zu optimieren und zu parallelisieren, weswegen Fortran im Bereich der Supercomputer eine bedeutende Rolle spielt. Aufgrund des Alters von **Fortran** stehen viele bewährte Programmpakete zur Verfgung.

```
Integer Zahl1, Zahl2
Integer ggT

Print *, Berechnung des ggT zweier positiver Zahlen.
```

```
Print *, Bitte erste Zahl eingeben:
Read (*,*) Zahl1
If (Zahl1 .le. 0) Then
  Print *, Na, groesser als 0 sollte die Zahl schon sein.
  Stop
EndIf
Print *, Bitte zweite Zahl eingeben:
Read (*,*) Zahl2
if (Zahl2 .lt. 0) Then
  Print *, Na, groesser als 0 sollte die Zahl schon sein.
      Stop
Endif
write (*,33) Zahl1, Zahl2, ggT (Zahl1, Zahl2)
stop 33
 format (Der ggT von ,
1 i8, und ,i8, ist ,i8)
End

Integer Function ggT (a, b)
Integer a, b, rest
If (a .lt. 0 .or. b .lt. 0) Then
  ggT = 0
  Return
EndIf
Loop
  rest = b - (b / a) * a
  b = a
  a = rest
Until (rest .eq. 0)
ggT = b
Return
End
```

### 1.3.4 Programmiersprache: Fourth

**Forth** wurde Anfang 1970 entwickelt und hauptsächlich zur Steuerung von Radioteleskopen eingesetzt. Die Sprache entstand als Alternative zur fehleranfälligen Assemblerprogrammierung. Sie hat sich aber im Laufe der Zeit nicht durchsetzen können und kommt heute nur noch zum Erstellen von Programmen für Workstations und einfache Steuerrechner zum Einsatz.

```
Programm ggT.4th
: ggt
  dup 0 = if drop else
  over 0 = if swap drop else
  over mod swap ggt then then ;
: start ."ggT zweier Zahlen"cr
        ."Bitte erste Zahl eingeben: ccept
```

```
        dup 1 < abort"Groesser als 0 sollte sie schon sein."
        ."Bitte zweite Zahl eingeben: ccept
        dup 1 < abort"Groesser als 0 sollte sie schon sein."
        ggt ."Der grte gemeinsame Teiler ist ". ;
```

### 1.3.5 Programmiersprache: Logo

Die Programmiersprache **Logo** wurde 1960 entwickelt und ist ein Verwandter der Programmiersprache Lisp. **Logo** galt als leicht zu erlernen, hatte zu dieser Zeit eine sehr hohe Leistungsfähigkeit durch dynamische Listen, frei definierbaren und rekursiv aufrufbaren Funktionen. Trotzdem konnte sich Logo nicht gegenüber anderen ersten Programmiersprachen wie z. B. BASIC durchsetzen, was auch daran lag, dass sie kindgerecht entwickelt und daher von vielen unterschätzt wurde. Eingesetzt wurde Logo vor allem im pädagogischen Bereich.

```
to ggt :x :y
  if (equalp :x 0) [output :y]
  output ggt (remainder :y :x) :x
end

to start
  print [ggT zweier Zahlen]
  print [Bitte erste Zahl eingeben:]
  make "zahl1 readword
  if lessp :zahl1 1 [pr [Positiv sollte die Zahl
     schon sein] stop]
  print [Bitte zweite Zahl eingeben:]
  make "zahl2 readword
  print [ggT ist:]
  if lessp :zahl2 1 [pr [Positiv sollte die Zahl schon
     sein] stop]
  print ggt :zahl1 :zahl2
end
```

### 1.3.6 Programmiersprache: Prolog

**Prolog** ist eine deklarativ orientierte Programmiersprache, die Anfang 1972 entwickelt wurde. Das Ziel war die Sprachanalyse und Implementierung eines Frage-Beantwortungsverfahrens. Sie beruht auf den mathematischen Grundlagen der Prädikatenlogik und ist im Grunde eine Sammlung von Horn-Klauseln. In den 1980er Jahren spielte die Sprache eine wichtige Rolle beim Bau von Expertensystemen. Die Sprache wird heute noch in den Bereichen Computerlinguistik und der künstlichen Intelligenz verwendet.

```
Programm ggT.Pro
ggT(N,0,N).
ggT(N,M,T):-
  M > 0,
```

```
  R is N mod M,
  ggT(M,R,T).

start():-
  write("ggT."),
  nl,
  write("Bitte erste Zahl eingeben: "),
  read(X),
  write("Bitte zweite Zahl eingeben: "),
  read(Y),
  ggT(X,Y,Z),
  write("ggT ist: "),
  write(Z),
  nl.
```

## 1.4 Fazit

Der Euklidische Algorithmus hat auch in der heutigen Zeit eine groe Bedeutung in der Mathematik, denn er findet sich neben seinem Haupteinsatzgebiet, der Ermittlung eines größten gemeinsamen Teilers, auch in anderen mathematischen Verfahren, wie z.B im Chinesischen Restsatz, Restklassenringen oder den Fibonacci Zahlen, wieder. Daher findet man den Euklidischen Algorithmus nicht nur in den gängigen imperativen und funktionalen Programmiersprachen, sondern auch in beispielsweise Logikbasierten Programmiersprachen.

## 1.5 Quellenangaben


- http: www8.informatik.unierlangen. de/IMMD8/Lectures/THINF3/Folien04/arithmetik4.pdf
- http: www-dm.informatik.unituebingen. de/lehre/kryptoVL/ws0607/EGGT.pdf
- http: de.encarta.msn.com/encyclopedia721546422=EuklidischerAlgorithmus:htmlhttp : staff:fim:uni .. passau:de=algebra=teaching=crypto=02:pdf
- http://wikipedia.de/
- Knuth, Donald E.: The art of computer programming, volume 3, chapter 4, 1998.
- Brill, Manfred : Mathematik fr Informatiker , 2001
- http://juergen.deswahnsinns.de/babel.html 14

KAPITEL 3

# Das Eleganteste Quicksort

**Ausarbeitung von Thorben Seeland**

## 3.1 Motivation

### Warum Quicksort?

**Quicksort** ist der Sortieralgorithmus, der unter UNIX am häufigsten verwendet wird. Er hat zwar, wie wir später sehen werden, im schlechtesten Fall quadratische Laufzeit (langsam), ist aber im Durchschnitt und im Normalfall sehr effizient.

### Wer hat es erfunden?

Der Quicksort-Algorithmus wurde 1960 von C. A. R. Hoare entwickelt und 1962 veröffentlicht[1]. Seitdem wurden noch Optimierungen vorgenommen, wobei mir die Namen Wegner und Bentley begegnet sind[2].

### Was ist elegant?

Wenn ich Code schreiben soll, finde ich genau den Code elegant, den ich schnell schreiben kann, also kurzen Code.
Wenn ich Code lesen soll, finde ich den Code elegant, den ich schnell verstehen kann.
Eine Idee, z.B. für einen Algorithmus, ist elegant, wenn sie mir einfällt und ich sie schnell so aufschreiben kann, dass ich sie auch später noch verstehen und nachvollziehen kann.
Ein Algorithmus, wie auch Quicksort, soll jedoch nicht nur elegant sein, sondern auch

[1] vgl. http://www.linux-releated.de/index.html?/coding/sort/sort_quick.htm

[2] Software - Practice and experience, Vol. 23, November 1993, „Engeneering a Sort Function“, Bentley, McIlroy

effizient. Effizient heißt: So schnell wie möglich und so platzsparend wie möglich.

Quicksort ist ein schönes Beispiel dafür, dass sich diese Anforderungen für Eleganz wunderbar widersprechen können.

Suchen wir also **das eleganteste Quicksort**.

## 3.2 Quicksort entwickeln

Fangen wir damit an, einen Sortieralgorithmus zu entwickeln. Wir betrachten hierbei der Einfachheit halber nur aufsteigende Sortierung, da sie sich nicht wesentlich von der Absteigenden unterscheidet.

### 3.2.1 Insertionsort

#### Wieso denn das?

Insertionsort verhält sich so wie Quicksort im Worst-Case, nur verkehrt herum. Insertionsort betrachtet in jedem Durchlauf ein Element mehr als im vorhergegangenen Durchlauf, Quicksort betrachtet im schlechesten Fall genau ein Element weniger, als im vorhergegangenen Durchlauf.

#### Wie funktioniert Insertionsort?

Insertionsort sortiert erst die ersten zwei Elemente und schiebt danach in jedem Durchgang das nächste Element solang nach vorn, bis es an der richtigen Position steht.

```
public void iisort(int[] array){
  for(int i=1;i<array.length; i++) {
    for(int j=i;(j>0)&&(array[j-1]>array[j]);j--){
      swap(j, j-1, array);
    }
  }
}
```

Man sortiert die ersten $i$ Elemente des Arrays und schiebt das $i+1$-te Element solang zum Anfang des Arrays, bis es an der richtigen Stelle ist.

Wir können nun also sortieren, und das Stückchen Code, das uns hier vorliegt, ist kurz, überschaubar und verständlich.
Leider ist das Stückchen Code nicht unbedingt effizient, und es ist leicht zu sehen, dass im schlimmsten Fall quadratische Zeit benötigt wird. Also kann die Idee nicht die Beste sein.

Es fällt auf, dass sehr häufig Werte vertauscht werden, besonders, wenn kleine Werte am Ende des Arrays stehen.

Ein Verbesserungsvorschlag:

### 3.2.2 Einfaches Quicksort

Quicksort „wählt“ genau wie Insertionsort in jedem Durchlauf ein Vergleichselement (vorerst das erste Element der Teilfolge), jedoch setzt Quicksort dieses Element nicht einfach an die Stelle, wo es in diesem Moment am besten passt, sondern an die Stelle, wo es hingehört. Quicksort tut dies, indem es alle Elemente, die kleiner sind als das Vergleichselement, an eine Seite und alle größeren Elemente an die andere Seite legt. Die gleichen Elemente werden bei dieser Implementierung zu den größeren Elementen gelegt.

```
public void quicksort(int[] array,int erstes, int letztes){
  if(letztes-erstes>0){
    int j=erstes;
    for(int i=erstes+1;i<=letztes;i++){
      if(array[i]<array[erstes])
        swap(i, ++j, array);
    }

    swap(erstes, j, array);

    quicksort(array, erstes, j-1);
    quicksort(array, j+1, letztes);
  }
}
```

Wir sind nun also ein wenig schneller als Insertionsort und wenn auch der Code ein wenig gewachsen ist, immernoch in einem überschaubaren Umfang des Programmcodes.

Jetzt fällt jedoch auf, dass besonders für einen relativ großen Vergleichswert oft vertauscht werden muss, da man zwangsweise viele kleinere Werte findet.

### 3.2.3 Optimiertes Quicksort

Wenn man die Zahlenfolge vom kleinsten Index aufwärts nach einem Element größer dem Vergleichselement durchsucht, bis man eins gefunden hat und dasselbe absteigend vom größten Index für ein kleineres tut, kann man einige Vertauschungen sparen.

```
public void quicksort(int[] array, int p0, int pN){
  if(letztes-p0>0){
    int i=p0+1;
    int j=pN;
    while(i<=j){
      for(;array[i]<array[p0];i++);
      for(; array[j]>array[p0];j--);
```

```
        if(j<i)
          break;

      swap(i, j, array);
    }
    swap(p0, j, array);

    quicksort(array, p0, j-1);
    quicksort(array, j+1, pN);
  }
}
```

In dieser Implementierung stoppen die for-Schleifen auch bei gleichwertigen Elementen, da nur ein Element an die richtige Position gerückt werden soll. Dies macht den Schritt zur nächsten Verbesserung etwas leichter.

Wir haben nun ein Quicksort, das mit möglichst wenigen Vertauschungen in einem Durchlauf genau ein Element an die richtige Position setzt. Es ist wieder ein bisschen gewachsen und ein bisschen schneller. Wenn der Code also verständlich ist, ist dies eine durchaus elegante Implementierung des Quicksort-Algorithmus.

Aber diese Implementierung hat noch eine Schwäche: Arrays mit vielen gleichwertigen Elementen.
Diese Schwäche ist ein Spezialfall, doch wenn man Quicksort auf jeden wie auch immer beschaffenen Array anwenden möchte, sollte auch dieser Spezialfall abgedeckt werden.

### 3.2.4 Quicksort für kleine Wertebereiche

Hierbei stoppen die for-Schleifen auch wieder bei gleichwertigen Elementen, jedoch um sie an den Rand zu schieben und so rekursive Aufrufe für gleichwertige Elemente zu sparen.
Es werden also gleichwertige Elemente, die bei der Suche nach einem größeren Element gefunden werden, mit dem kleineren Element mit dem niedrigsten Index getauscht. Bei der Suche nach einem kleineren Element wird das gefundene gleichwertige Element mit dem größeren Element mit dem höchsten Index getauscht.

```
public void quicksort(int[] a, int p0, int pN){
  if(pN-p0>0){
    int iterL;
    int putL=iterL=p0+1;
    int iterR=pN;
    int putR=pN;
    while(iterL<=iterR){
      while(iterL<=iterR &&(a[iterL]<=a[p0])){
        if(a[iterL]==a[p0])
          swap(putL++, iterL, a);
        iterL++;
      }
```

```
      while(iterL<=iterR && (a[iterR]>=a[p0])){
        if(a[iterR]==a[p0])
          swap(putR--, iterR, a);
        iterR--;
      }
      if(iterL>iterR)
        break;

      swap(iterL, iterR, a);
    }
  }
}
```

Nun sind die Werte des Arrays fast sortiert, es müssen nur noch die gleichwertigen Elemente an die richtige Position verschoben werden. Hierzu bestimmen wir die Anzahl der zu verschiebenden Elemente am niedrig indizierten Rand. Da es vorkommen kann, dass es mehr gleichwertige als kleinere Elemente gibt, bestimmen wir das Minimum aus der Anzahl der gleichen Elemente am Rand und den kleineren Elemente und tauschen von der Position aus, an der sich die Zeiger gekreuzt haben an.

```
  int anz=Math.min(putL-p0, iterL-putL);

  for(int i=p0, h=iterL-anz; anz>0 ;anz--)
    swap(i++, h++, a);

  anz=Math.min(pN-putR, putR-iterR);

  for(int i=iterR, h=pN+1-anz;anz>0;anz--)
    swap(i++, h++, a);

  quicksort(a, p0, p0+iterL-putL);

  quicksort(a, pN-putR+iterR, pN);
}
```

Wir haben nun eine Version von Quicksort entwickelt, die prinzipiell genau so schnell ist, wie die Vorhergegangene (s. Optimiertes Quicksort) und in dem Spezialfall, dass manche Werte häufig vorkommen sogar noch schneller ist. Also ist diese Version auf jeden Fall effizienter, doch der Schreibaufwand ist doch recht hoch und die Lesbarkeit ist doch um ein großes Stück zurückgegangen.

### 3.2.5 Geschicktes Wählen des Vergleichswerts

Bis jetzt konnte man immer feststellen, dass die schlechteste Eingabe für alle Implementationen ein aufsteigend sortiertes Array ist, da wir als Vergleichswert immer das erste Element des Arrays gewählt haben. Somit dürften wir Quicksort nicht anwenden, wenn wir wüßten,

dass die Arrays zu einem großen Teil vorsortiert sind.
Ich möchte nun zwei Möglichkeiten vorstellen, dies zu umgehen.

1. **Randomisierte Wahl des Vergleichselements**
   Wenn der Vergleichswert zufällig gewählt wird, ist es schwer eine Worst-Case-Eingabe zu generieren.

   ```
   int i=(int)(java.lang.Math.random()*1000000);
   i=i%(pN-p0+1)+p0;
   swap(i, p0, a);
   ```

2. **Median oder Pseudomedian aus einer Teilmenge wählen**
   Der Median aus einer Teilmenge (hier exemplarisch 3 Werte) ist der Wert, der wenn man die Werte sortiert, in der Mitte steht.
   Der Pseudomedian (hier exemplarisch aus 9 Werten) ist der Median, der 3 Mediane aus jeweils 3 Werten. Er ist nicht zwangsläufig der Median der 9 Werte, aber doch mit einiger Wahrscheinlichkeit näher am Median des Arrays als ein einfacher Median aus 3 Werten. Und je näher der Vergleichswert am Median des Arrays ist, umso weiter ist er vom Rand des Arrays entfernt, sodass es, je mehr Werte man betrachtet, unwahrscheinlicher wird den schlechtesten Wert zu treffen.
   Es ist jedoch wiederum „teuer“ einen Median zu ermitteln (bis zu 3 signifikante Vergleiche), daher sollte man die Verwendung von der Anzahl der zu betrachtenden Elemente abhängig machen.

   ```
   int number=pN-p0+1;
   int mIndex=(pN+erstes+1)/2;

   if(number>7){
     int indexE=p0;
     int indexL=pN;
     if(number>40){
       int s=number/8;
       indexE=median(a, indexE, indexE+s, indexE+2*s);
       mIndex=median(a, mIndex-s, mIndex, mIndex+s);
       indexL=median(a, indexL-2*s, indexL-s, indexL);
     }
     mIndex=median(a, indexE, mIndex, indexL);
   }
   swap(mIndex,p0, a);
   ```

   [3]

Dadurch, dass wir in unseren Implementierungen immer das erste Element als Vergleichselement gewählt haben, können wir das zufällig oder durch den Pseudomedian ausgewählte Element mit dem ersten Element tauschen und unsere Implementierungen von Quicksort weiterverwenden.

[3] Werte übernommen aus: Software & Practice and experience, Vol. 23, November 1993, „Engeneering a Sort Function“, Bentley, McIlroy

## 3.3 Quicksort analysieren

Intuitiv sind die Varianten von Quicksort immer effizienter geworden (Tests in „Engeneering a Sort Function“ belegen dies). Ich möchte jedoch auf die letze Optimierung eingehen, die wir vorgenommen haben und testen, in wie weit sich der Mehraufwand an Codezeilen gelohnt hat.

### 3.3.1 Allgemeiner Fall

Der allgemeine Fall ist der, dass das Eingabealphabet viel größer ist, als die Arraylänge, sodass die endgültige Position des Vergleichselements mit gleicher Wahrscheinlichkeit jede Position des Arrays sein kann.
Im 3. Kapitel in „Beautiful Code“[4] wird die Laufzeit für die Variante aus 3.2.3 auf die Folge $C_n = (n+1)(2H_{n+1} - 2) - 2n \approx 1.386 n \lg(n)$ bei einer Eingabelänge von $n$ zurückgeführt[5].
Die so ermittelte Laufzeit entspricht auch der Laufzeit für die Variante aus 3.2.4.

### 3.3.2 Für ein kleines Eingabealphabet und einen großen Array

Ein kleines Eingabealphabet und ein großes Array meint, dass jede Anzahl von Elementen, die gleich dem Vergleichswert sind, gleich wahrscheinlich ist und auch die Position des ersten gleichwertigen Elements abhängig von der Anzahl der gleichwertigen Elemente gleich wahrscheinlich ist. Dies ist der Fall, wenn das Eingabealphabet kleiner gleich der Arraylänge ist.

Für das Programm aus 3.2.3 entsteht für diesen Spezialfall kein deutlicher Unterschied zum allgemeinen Fall, da nur ein Element pro Durchlauf an seine endgültige Position geschoben wird und diese Implementierung im Best-Case auf eine Laufzeit von $\mathrm{O}(n \log(n))$ kommen kann.
Interessanter ist jedoch die Betrachtung der Implementierung aus 3.2.4, die das Vorkommen vieler gleicher Elemente ausnutzen kann.

Ich möchte hierfür einen nicht mathematischen Ansatz wählen, den ich in „Beautiful Code“ gefunden habe. Dieser Ansatz besteht daraus, ein Programm zu schreiben, das die durchschnittliche Laufzeit berechnet und dieses zu optimieren, um im Idealfall am Ende ein Programm präsentieren zu können, das mit konstanter Laufzeit die durchschnittliche Laufzeit für Quicksort-Aufrufe auf Arrays der Länge $n$ ermittelt.
Der Vorteil dieser Vorgehensweise ist, dass man, wenn man am Anfang ein Programm schreibt, das offensichtlich das richtige Ergebnis liefert, nach jedem weiteren Schritt testen kann ob man einen Fehler gemacht hat (ich spreche von „testen“ und nicht von „beweisen“).

[4]Beautiful Code, 1. Edition, Oram, Wilson (Ed.), O’Reilly, Juni 2007, Kap. 3, „The Most Beautiful Code I Never Wrote“, Bentley

[5]wobei $H_n$ die harmonische die $n$-te Harmonische Zahl $\left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}\right)$ ist

#### Feststellen der Laufzeit

Um die Laufzeit eines Programms zu ermitteln ist die einfachste Methode einfach die Zeit für einen zufälligen Durchlauf zu messen. Hierbei wird es jedoch schwierig, nachher den Bezug zur Eingabelänge herzustellen, zumal die Zeit, die für das Ausführen des Programms benötigt wird von der Hardware abhängig ist.
Eine andere Variante ist, die Operationen zu zählen, die das Programm durchführt. Wobei man sich hierbei darauf beschränken kann, nur die Anzahl sog. „signifikanter“ Operationen zu zählen.
Im Fall unseres Programms ist eine signifikante Operation ein Vergleich zweier Elemente im Array, den ich weiterhin „signifikanter Vergleich“ nennen werde.
Die einfachste Methode, signifikante Operationen zu zählen, ist, in das Programm nach einer signifikanten Operation einen Zähler einzufügen und eine Statistik für zufällige Eingaben zu führen, die der zu betrachtenden Eingabe entsprechen.
Also fügen wir einen globalen Zähler ein und erhöhen ihn nach jedem signifikanten Vergleich um 1.

```
int counter=0;

public void quicksort(int[] a, int p0, int pN){
  if(letztes-p0>0){
    int iterL;
    int putL=iterL=p0+1;
    int iterR=pN;
    int putR=pN;
    while(iterL<=iterR){
      while(iterL<=iterR &&(a[iterL]<=a[p0])){
        if(a[iterL]==a[p0])
          swap(putL++, iterL, a);
        iterL++;
        counter+=2;
      }
      while(iterL<=iterR && (a[iterR]>=a[p0])){
        if(a[iterR]==a[p0])
          swap(putR--, iterR, a);
        iterR--;
        counter+=2;
      }
      if(iterL>iterR)
        break;

      swap(iterL, iterR, a);
    }
  }
}
```

Da als Bedingung für die while-Schleife wie auch innerhalb der while-Schleife ein signifikanter Vergleich durchgeführt wird, habe ich den Zähler um 2 erhöht.

Hierbei werden jedoch nicht alle Eingaben, sondern nur beispielhafte Eingaben betrachtet. Um alle Eingaben zu betrachten, kann man eine kleine Abstraktion vornehmen.

#### Alle möglichen Eingaben betrachten

Das von uns betrachtete Modell verlangt, dass alle Anzahlen von gleichen Elementen, sowie alle Anzahlen von kleineren Elementen abhängig von der Anzahl der gleichen Elemente für einen gegebenen Vergleichswert gleich wahrscheinlich sind. Es reicht also die Eingabe unabhängig von den Werten anhand dieser Bedingungen zu simulieren, also zuerst die Anzahl der gleichen Elemente festzulegen, danach die Anzahl der Kleineren und danach zu zählen, wieviele signifikante Vergleiche benötigt werden. Jedoch muss auch jede der möglichen Kombinationen aus den Anzahlen der gleichen und kleineren Elemente gezählt werden, damit wir die durschnittliche Anzahl von signifikanten Vergleichen bestimmen können.
In dem folgenden Programm haben wir

1. einen Zähler *counter* eingeführt, der die Vergleiche zählt, die in einem Rumpfdurchlauf auf jeden Fall gemacht werden (entspricht der ersten while-Schleife im vorigen Programm),
2. einen Zähler *counter*2 eingeführt, der die Vergleiche zählt, die durch die Aufrufe anfallen, die durch die Aufteilung in kleinere und gleiche Elemente entstehen (entspricht den rekursiven Aufrufen),
3. einen Zähler *moeglichkeiten* eingefügt, der die Anzahl der möglichen Zerlegungen in kleinere und gleiche Elemente zählt,
4. die Variable *groesse* eingefügt, die gleich der Anzahl der Elemente ist, die in dem Teilarray von *p0* bis *pN* liegen.

```
public int cQuicksort(int p0, int pN){
  int counter=0;
  int counter2=0;
  int moeglichkeiten=0;
  int groesse=pN-p0+1;

  if(groesse>1){
    int iterL;
    int putL=iterL=p0+1;
    while(iterL<=pN){
      iterL++;
      counter+=2;
    }
    for(int gleiche=1;gleiche<=groesse;gleiche++){
      for(int kleinere=0;kleinere<=groesse-gleiche;kleinere++){
        counter2+=cQuicksort(p0, p0+kleinere);
        counter2+=cQuicksort(p0+kleinere+gleiche, pN);
        moeglichkeiten++;
      }
```

```
    }
  }
  return counter+counter2/moeglichkeiten;
}
```

Die Abstraktion von *pN-p0+1* zu *groesse* können wir auch schon in den Parametern realisieren, indem wir nicht die Indizes, des sowieso nicht vorhandenen Teilarrays übergeben, sondern direkt die Anzahl der Elemente, die betrachtet werden sollen. Diese wird im weiteren $n$ genannt.

```
public int cQuicksort2(int n){
  int counter=0;
  int counter2=0;
  int moeglichkeiten=0;

  if(n>1){
    for(int i=2;i<=n;i++){
      counter+=2;
    }
    for(int gleiche=1;gleiche<=n;gleiche++){
      for(int kleinere=0;kleinere<=n-gleiche;kleinere++){
        counter2+=cQuicksort(kleinere);
        counter2+=cQuicksort(n-kleinere-gleiche);
        moeglichkeiten++;
      }
    }
  }
  return counter+counter2/moeglichkeiten;
}
```

Dieses Programm ist jedoch nur bedingt verwendbar, da es durch die geschachtelten rekursiven Aufrufe in der for-Schleife exponentielle Zeit benötigt.

### Ein brauchbarer Ansatz

Man kann sich nun mit dem Ansatz der dynamischen Programmierung behelfen, indem man sich ein Array erstellt, das an der Position $n$ den Rückgabewert von $cQuicksort2(n)$ speichert, sodass man ihn nicht für jedes *kleinere* neu aufrufen muss.

Im folgenden Programm haben wir

1. das globale Array *statistic* eingefügt,
2. die for-Schleife zum berechnen von *counter* ersetzt durch die direkte Addition von $2(n-1)$ auf $statistic[n]$,
3. *kleinere* umbenannt auf *kl* und *gleiche* umbenannt auf *gl* und werden diese Benennung im Folgenden beibehalten.

```
double[] statistic;

private void analyse1(int n){
  if(n>1){
    analyse1(n-1);

    int counter2=0;
    int moeglichkeiten=0;

    for(int gl=1;gl<=n;gl++){
      for(int kl=0;kl<=n-gl;kl++){
        moeglichkeiten++;
        counter2+=statistic[kl]+statistic[n-kl-gl];
      }
    }
    statistic[n]+=counter2;
    statistic[n]=2*(n-1)+statistic[n]/moeglichkeiten;
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

Nun fällt auf, dass *moeglichkeiten* in der inneren for-Schleife in jedem Schleifendurchlauf um 1 erhöht wird und man es eigentlich aus dieser for-Schleife herausziehen kann, und dass jedes $statistic\,[kl]$ in einem vollständigen Schleifendurchlauf 2 mal zu $counter2$ addiert wird

#### Entfernen von moeglichkeiten aus den for-Schleifen

Wir ziehen nun *moeglichkeiten* aus der inneren for-Schleife, in der es $n-gl+1$ mal um 1 erhöht wurde heraus und erhöhen $counter2$ in jedem inneren Schleifendurchlauf um $2 \cdot statistic\,[kl]$.

```
private void analyse2(int n){
  if(n>1){
    analyse2(n-1);

    int counter2=0;
    int moeglichkeiten=0;

    for(int gl=1;gl<=n;gl++){
      for(int kl=0;kl<=n-gl;kl++)
        counter2+=2*statistic[kl];

      moeglichkeiten+=n-gl+1;
    }
```

```
    statistic[n]+=counter2;
    statistic[n]=2*(n-1)+statistic[n]/moeglichkeiten;
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

Für den nächsten Schritt können wir feststellen, dass $n-gl+1$ alle Werte von 1 bis $n$ annimmt, sodass nach einem vollständigen Schleifendurchlauf $moeglichkeiten = \sum_{i=1}^{n} i = \frac{(n+1)\cdot n}{2}$ gilt. Der Code ist mit dieser Veränderung der folgende:

```
private void analyse3(int n){
  if(n>1){
    analyse3(n-1);

    int counter2=0;
    int moeglichkeiten=0;

    for(int gl=1;gl<=n;gl++){
      for(int kl=0;kl<=n-gl;kl++)
        counter2+=2*statistic[kl];
    }
    moeglichkeiten=(n+1)*n/2;

    statistic[n]+=counter2;
    statistic[n]=2*(n-1)+statistic[n]/moeglichkeiten;
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

*moeglichkeiten* wird nun also in konstanter Zeit berechnet. Wenden wir uns also der nächsten Vereinfachung zu.

### Auflösen der for-Schleifen

Wenn man sich an dieser Stelle die Frage stellt: „Wie oft wird $2 \cdot statistic[kl]$ für ein gegebenes $kl$ zu $counter2$ addiert?“, stellt man fest: Für jedes $gl \geq 1$, für das gilt $n - gl \geq kl$, genau einmal. Wenn man die Bedingung umformt erhält man $n - gl \geq kl \Leftrightarrow n - gl - kl \geq 0$ also genau $n - kl$-mal, wobei $kl == 0$ vernachlässigt werden kann, da $2 \cdot statistic[0] == 0$.
So ergibt sich:

```
private void analyse5(int n){
  if(n>1){
    analyse5(n-1);

    int moeglichkeiten=0;
    int counter2;

    for(int kl=1;gl<n;kl++){
      counter2+=2*statistic[kl]*(n-kl);
    }

    moeglichkeiten=(n+1)*n/2;
    statistic[n]=counter2;

    statistic[n]=2*(n-1)+statistic[n]/moeglichkeiten;
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

Es ist nun leicht zu sehen, dass jedes $2 \cdot statistic\,[kl]$ für $n+1$ genau einmal öfter addiert werden muss als für $n$.
Um die Multiplikationen zu sparen, versuchen wir also die Summe der $2 \cdot statistic\,[kl] \cdot ((n-1)-kl)$ aus dem schon bekannten Durchschnittswert $statistic[n-1]$ zum Berechnen von $analyse6(n)$ zu nutzen.
Wir wissen, dass

$$statistic\,[n-1] = 2 \cdot (n-1-1) + \frac{counter2}{moeglichkeiten}$$

wobei wir genau $counter2$ suchen. Wir erhalten also

$$counter2 = (statistic\,[n-1] - 2 \cdot (n-1-1)) \cdot moeglichkeiten$$

wobei $counter2 = \sum_{kl=1}^{n-1} 2 \cdot statistic\,[kl] \cdot ((n-1)-kl)$ und wir nun noch $\sum_{i=1}^{n-1} statistic\,[i]$ zu dem eben ermittelten $counter2$ addieren müssen um an $counter2$ für $n$ zu gelangen.
Hierfür führen wir eine globale Variable $zwischensumme$ ein, in der wir genau $\sum_{i=1}^{n-1} 2 \cdot statistic\,[i]$ gespeichert haben, sodass

$$counter2 = (statistic\,[n-1] - 2 \cdot (n-1-1)) \cdot moeglichkeiten + zwischensumme$$

und

$$statistic\,[n] = 2 \cdot (n-1-1) + \frac{counter2}{moeglichkeiten}.$$

Hieraus ergibt sich dann folgender Code

```
double zwischensumme;

private void analyse6(int n){
  if(n>1){
    analyse6(n-1);

    int moeglichkeiten=(n+1)*n/2;
    int moeglichkeiten1=(n-1)*n/2;
    int counter2=(statistic[n-1]-2*(n-2))*moeglichkeiten1+zwischensumme;

    statistic[n]= 2*(n-1)+counter2/moeglichkeiten;

    zwischensumme+=2*statistic4[n];
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

wobei textitmoeglichkeiten für die Berechnung für $n$ und *moeglichkeiten1* für die Berechnung für $n-1$ berechnet ist.

Nun haben wir zwar keine for-Schleife mehr, jedoch einen neue versteckte Rekursion über die globale Variable *zwischensumme*, da unser Ziel ist, den Code so weit zu vereinfachen, dass wir in konstanter Zeit die durchschnittliche Laufzeit für eine Arraylänge $n$ berechnen wollen, sollten wir diese, wenn es geht, so auflösen, dass wir *zwischensumme* in konstanter Zeit berechnen können.

Hierfür müssen wir $counter2$ entfernen, ihn also direkt in der Zuweisung für $statistic[n]$ berechnen, sodass

```
statistic[n]=2*(n-1)+((statistic[n-1]-2*(n-2))*moeglichkeiten1+zwischensumme)
                    /moeglichkeiten;
```

und wir nutzen, dass

$$\text{zwischensumme}(n) = \text{zwischensumme}(n-1) + 2 * statistic[n-1]$$

wobei *zwischensumme* zwischensumme($n$) entspricht.
Da *statistic* $[n]$ noch unbekannt ist, stellen wir zuerst die Zuweisung um, sodass wir *statistic* $[n-1]$

erhalten

$$\begin{aligned}
statistic[n-1] &= 2\cdot(n-2) \\
&\quad+\frac{(statistic[n-2]-2\cdot(n-3))\cdot \text{moeglichkeiten}(n-2)}{\text{moeglichkeiten}(n-1)} \\
&\quad+\frac{\text{zwischensumme}(n-1)}{\text{moeglichkeiten}(n-1)} \\
&\Updownarrow \\
\text{zwischensumme}(n-1) &= (statistic[n-1]-2\cdot(n-2))\cdot \text{moeglichkeiten}(n-1) \\
&\quad-(statistic[n-2]-2\cdot(n-3))\cdot \text{moeglichkeiten}(n-2)
\end{aligned}$$

Um zwischensumme($n$) zu erhalten, müssen wir nun noch $2 * statistic[n-1]$ addieren, sodass sich folgender Code ergibt

```
private void analyse7(int n){
  if(n>1){
    analyse7(n-1);

    int mgl=(n+1)*n/2;
    int mgl1=(n-1)*n/2;
    int mgl2=(n-1)*(n-2)/2;
    double zwischensumme=
              (statistic[n-1]-2*(n-2))*mgl1
              -(statistic[n-2]-2*(n-3))*mgl2
              +2*statistic[n-1];

    statistic[n]=zwischensumme
                     +(statistic[n-1]-2*(n-2))*mgl1;

    statistic[n]=2*(n-1)+statistic[n]/mgl;
  }
  if(n==0)
    statistic[0]=0;

  if(n==1)
    statistic[1]=0;
}
```

Hierbei ist $mgl$ = moeglichkeiten($n$), $mgl1$ = moeglichkeiten($n-1$) und $mgl2$ = moeglichkeiten($n-2$).

### In eine mathematische Form bringen

Im folgenden Schritt haben wir

1. *zwischensumme*, *mgl*, *mgl*1 und *mgl*2 direkt in die Zuweisung für $statistic[n]$ eingefügt

2. die Rekursion in eine for-Schleife aufgelöst, wobei nun beim Eintritt in den for-Schleifen-Rumpf $st1 = statistic\,[n-1]$ und $st2 = statistic\,[n-2]$ und nach dem for-Schleifen-Rumpf $st1 = statistic\,[n]$ und $st2 = statistic\,[n-1]$.

```
private double analyse9(int n){
  double st2=0;
  double st1=0;
  double sth;
  for(int i=2; i<=n;i++){
    sth=st1;
    st1=2*(i-1)
        +(
          (st1-2*(i-2))*(i-1)*i/2
          -(st2-2*(i-3))*(i-1)*(i-2)/2
          +2*st1+(st1-2*(i-2))*(i-1)*i/2
         )
         /((i+1)*i/2);
    st2=sth;
  }
  return st1;
}
```

Wenn wir dies in eine Folge $\mathrm{a}(n)$ umwandeln erhalten wir:

$$\mathrm{a}(n) = 2\cdot(n-1) + \frac{(\mathrm{a}(n-1)-2\cdot(n-2))\cdot n\cdot(n-1)-(\mathrm{a}(n-2)-2\cdot(n-3))\cdot\frac{(n-1)\cdot(n-2)}{2}+2\cdot\mathrm{a}(n)}{\frac{(n+1)\cdot n}{2}}$$

Leider übersteigt es meine mathematischen Fähigkeiten, diese Folge in eine einfache von $n$ abhängige Funktion umzuwandeln, daher habe ich versucht eine Funktion zu finden, die sich für große Werte von $n$ ähnlich verhält wie $\mathrm{a}(n)$, jedoch im Zweifel eine obere Schranke bildet. Für diese Funktion, die $a(n)$ approximiert, habe ich

$$\mathrm{f}(n) = 0.915 * (n) * \log(\log(n)) + 3.78 * n$$

gewählt, was zeigt, dass $\mathrm{a}(n)$ fast linear wächst.

## 3.4 Fazit

Wir haben nun also festgestellt, dass die Varianten aus 3.2.3 und 3.2.4 im Allgemeinen die gleiche Laufzeit haben.
Die Variante aus 3.2.4 in dem eher unwahrscheinlichen Spezialfall für kleine Wertebereiche mit einer Laufzeit von $\mathrm{O}(n\log(\log(n)))$ eine deutlich bessere Laufzeit vorweisen kann, als die aus 3.2.3.

Die Letztgenannte hat jedoch mit einer Laufzeit von $O(n \log(n))$ auch eine sehr gute Laufzeit, wenn das Vertauschen günstig realisiert werden kann.

Im Endeffekt ist die Variante aus 3.2.3 also ein sehr geeigneter Algorithmus um Arrays unbekannter Beschaffenheit zu sortieren und gleichzeitig auch in wenigen Zeilen Code zu implementieren.
Wenn ich also noch einmal gefragt werde:
„Was ist **das eleganteste Quicksort**?“
werde ich auf diese Variante aus 3.2.3 verweisen.

KAPITEL 4

# Elegante Tests – Binäre Suche

**Ausarbeitung von Fabian Bienek**

## 4.1 Einleitung - Was ist eigentlich elegant?

**Was ist eigentlich elegant?** Wenn man an Elegante Algorithmen denkt, fällt es eigentlich nicht schwer, verschiedene Arten von „elegant“ zu finden. Ein Algorithmus kann mit niedriger oberer Schranke funktionieren, einen kurzen und leicht lesbaren Code besitzen oder einfach nur schnell programmiert worden sein. Man findet mit Sicherheit noch andere Kriterien. Jedoch kann man relativ leicht feststellen, dass sich nicht immer alle Kriterien miteinander vereinbaren lassen.
In jedem Fall muss der Code aber richtig funktionieren. Was sich zunächst trivial anhört, ist auf den zweiten Blick gar nicht so einfach nachzuweisen. Ein gutes Beispiel ist die Binäre Suche (siehe 4.1).
Auf den ersten Blick sieht der Code korrekt aus. Erst bei näherem Betrachten fällt auf, dass die Berechnung des *middle*-Integers einen Fehler enthält. So kann die Berechnung die maximale Integergröße übersteigen, wenn *(left + right)* zu groß wird. Das resultiert in einem Aufruf auf einen negativen Index und damit in einer *ArrayIndexOutOfBoundException.*
Der Fehler lässt sich zwar relativ einfach beheben (int *middle* = *left* + ((*right* - *left*)/2)), allerdings zeigt er auch, das selbst kleine Programme nicht immer hundertprozentig fehlerfrei sind.

```
public static int binSearch (int[] a; int x) {
   int left = 0;
   int right = a.length-1;

   while (left <= right) {
     int middle = (left+right)/2;

     if (a[middle] < x)
        left = middle+1;
     else if (a[middle] > x)
        right = middle-1;
     else
        return middle;
   }
   return -1;
}
```

Abbildung 4.1: Binäre Suche mit Fehler

## 4.2 Elegante Tests

**Auch Tests können elegant sein.** Um einen Algorithmus als fehlerfrei zu erkennen, reicht es nicht, den Code oberflächlich zu begutachten. Oft ist es nötig, tiefer ins Detail zu gehen und viele Testumgebungen zu überprüfen.
Im Folgenden wird die Binäre Suche anhand von vier verschiedenen Testarten geprüft, um Sicherheit zu gewinnen, dass der Code richtig funktioniert. Diese Testarten sind:

- **Smoke Tests**, um die grundlegenden Funktionen des Algorithmus zu testen
- **Boundary Tests** für die Grenzwert-Instanzen (insbesondere kleine und große Arrays und die Lage des gesuchten Elementes)
- **Ausführliche Tests**, die auf randomisierte Arraygrößen und -inhalte eingehen
- **Laufzeit-Tests** zur Überprüfung der O-Notation.

**JUnit.** JUnit wurde von Kent Black und Erich Gamma mit der Idee entwickelt, Tests in Java möglichst ohne Mehraufwand durchzufüren. Um dies zu tun, reicht es, JUnit zu importieren und die zu testenden Methoden dann mit „@org.junit.Test“ zu markieren (siehe 4.2).

```
@Test public void testmethod() ...
```

Abbildung 4.2: Beispielmethode für JUnit

Die wichtigsten drei Methoden, die in diesem Kapitel verwendet werden, sind:

**assertEquals (Int, Int)** Vergleich der beiden Argumente. Bei Ungleichheit: *AssertionError*

**assertTrue(boolean)** Überprüfung des Arguments auf *true*. Bei Fehlschlag: *AssertionError*

**assertFalse(boolean)** Überprüfung des Arguments auf *false*. Bei Fehlschlag: *AssertionError*

Alle drei Methoden haben gemeinsam, dass sie keinen Rückgabewert besitzen (in Java also *void*). Dies hat zur Folge, dass sich mehrere Testfunktionen hintereinander schreiben lassen, ohne eine Ausgabe abfangen zu müssen. Sollte einer der Tests nicht bestehen, so bricht das Programm mit einem *AssertionError* ab.

## 4.3 Smoke Tests

**Elementare Tests** Smoke Tests rahmen die elementaren Funktionen eines Programms ein. Im Falle der Binären Suche wird also die Ausgabe des Algorithmus mit einem erwarteten Wert verglichen. Dazu initialisiert der Tester selbst ein fest belegtes Array und testet mit *assertEquals* diese Gleichheit (4.3).

```
public void smokeTests() {
    int[] testfeld={1,5,16,32};
    assertEquals(2, Util.Arrays.binarySearch(testfeld, 16));
    assertEquals(-1, Util.Arrays.binarySearch(testfeld, 42));
}
```

Abbildung 4.3: Beispiel Smoke Test

**Warum nicht mehr?** Am Beispiel oben fällt dem aufmerksamen Leser auf, dass es sich um sehr schlichte Tests handelt. Der Grund dafür ist, dass Smoke Tests nur die „kleinsten Schritte“ eines Codes testen sollen. Insbesondere erschließen sich für größere Programme sehr viele Smoke Tests. Um die Funktion dieser Testgruppe einzuhalten, werden daher die Tests möglichst simpel gehalten.

## 4.4 Boundary Tests

**Grenzwertüberprüfung** Die Testgruppe der Boundary Tests hat zum Ziel, sicherzustellen, dass der programmierte Algorithmus auch auf extremen Instanzen richtig funktioniert. Im Falle der Binären Suche zwingt sich unweigerlich auf, sowohl Tests über die Größe des Arrays, als auch über die Lage des zu suchenden Elementes durchzuführen.

**Arraygrößen.** Bei den Tests zur Größe eines Arrays gilt es, zwei Aspekte abzudecken: „zu kleine“ und „zu große“ Arrays. Kleine Arrays sind relativ schnell geprüft. Da wir immer noch auf festen, selbst gewählten Instanzen testen, reicht es, das Array selbst mit einer kritischen Größe zu initialisieren, und dann darauf Suchanfragen auszuführen (siehe 4.4).

```
public void searchEmptyArray() {
    int[] testfeld = new int[]();
    assertEquals(-1, Util.Arrays.binarySearch(testfeld, 16));
}

public void searchSizeOneArray() {
    int[] testfeld = {16};
    assertEquals(0, Util.Arrays.binarySearch(testfeld, 16));
    assertEquals(-1, Util.Arrays.binarySearch(testfeld, 33));
}
```

Abbildung 4.4: Tests für kleine Arrays.

Beide Tests bestehen ihren Durchlauf und ich kann somit sicher gehen, dass der Algorithmus auf kleinen Instanzen richtig arbeitet. Etwas problematischer wird es, wenn man große Arrays begutachtet. Man kann nicht sicher sein, dass die Suchfelder klein genug bleiben, damit es kein Speicherplatzproblem gibt. Um das zu umgehen, lässt sich ein Testansatz nutzen, der stark an eine Induktion erinnert:

- Auf kleinen Arrays lässt sich die Binäre Suche gut testen.
- Auf großen Arrays kann man den Mittelpunkt errechnen, ohne das ganze Array zu kennen (die Arraygröße reicht).

Sind beide Bedingungen erfüllt, kann man auch in großen Instanzen Zuversicht gewinnen. Tests auf kleinen Arrays haben wir bereits behandelt. Es bleibt also ein Test für die Mittelpunktberechnung.
Um den Mittelpunkt zu überprüfen, rahmen wir die Berechnung desselben in eine Methode ein und testen dann diese (4.5).

Wenn die Binäre Suche richtig programmiert wurde, sollte dieser Testlauf reibungslos klappen. Damit hat der Programmierer jetzt ein gewisses Maß an Zuversicht, dass es keine Probleme bezüglich Arraygröße geben wird. Was noch nicht überprüft wurde, ist die Lage des Suchelementes.

```
static int calculateMidpoint(int left, int right) {
    return middle = left + ((right - left)/2);
}

public void calculateMidpointWithBoundaryValues() {
    assertEquals(0, calculateMidpoint(0,1));
    assertEquals(1, calculateMidpoint(0,2));
    assertEquals(120000, calculateMidpoint(110000,130000));
   assertEquals(Integer.MAX_VALUE-1, calculateMidpoint(Integer.MAX_VALUE-1,
                                                        Integer.MAX_VALUE));
}
```

Abbildung 4.5: Tests für Mittelpunktberechnung

**Item Location.** Jedem Programmierer, der schon mal einen Suchalgorithmus auf einem Array durchgeführt hat, sollten drei wichtige Positionen für die Lage des gesuchten Elementes einfallen: Es kann direkt auf der Position des mittleren Elementes liegen, oder es kann sich ganz links oder rechts im Array befinden. Alle anderen Positionen sind Kombinationen dieser Fälle.
Es reicht also, auf einem festen Array die Ausgabe der Suche mit einem Erwartungswert zu vergleichen. Abbildung 4.6 zeigt, wie simpel ein solcher Test aussehen kann.

```
public void boundaryTestforItemLocation() {
    int[] testfeld = new int[] (-32,-14,0,23,Integer.MAX_VALUE);
    assertEquals(0, Util.binarySearch(testfeld, -32));
    assertEquals(2, Util.binarySearch(testfeld, 0));
    assertEquals(4, Util.binarySearch(testfeld, Integer.MAX_VALUE));
}
```

Abbildung 4.6: Item Location

Insbesondere wurden bei diesem Test zum ersten Mal negative Zahlen und extreme Werte (hier Integer.MAX_VALUE) in das Testarray übernommen. Dieser Schritt ist ein wichtiger Bestandteil im Testverlauf für jedes Programm. Wenn man selbst einen Algorithmus testet, erkennt man oft während der Testprogrammierung weitere Fälle, die geprüft werden müssen.

## 4.5 Ausführliche Tests

Bis jetzt wurden alle Tests auf festen Arrayinstanzen durchgeführt. Dieser Abschnitt soll jetzt dem Rechner diese Aufgabe zuteilen, sodass wir auch auf nicht bedachten Arrays eine gewisse Sicherheit bekommen.

**Random Arrays.** Um dem Rechner diesen Test beizubringen, benötigen wir zunächst randomisierte Arrays. Diese sollen sowohl randomisiert in ihrer Größe, als auch in ihrem Inhalt sein.
Java stellt hier den *Random*-Operator zur Verfügung, mit welchem sich ein solches Array recht einfach generieren lässt (siehe 4.7).

```
public int[] generateRandomArray (int maxArrSize, int maxV) {
    Random rand = new Random();
    int Arraysize = 1 + rand.nextInt(maxArrSize);
    int[] randomArray = new Int[Arraysize];
    for (int i=0; i<Arraysize; i++) {
        randomArray[i] = rand.nextInt(maxV);
    }
    Arrays.sort(randomArray);
    return randomArray;
}
```

Abbildung 4.7: Randomisierte Arrays

Mit dieser Methode kann man nun randomisierte, sortierte (siehe *Arrays.sort()*) Arrays generieren, die der Rechner für seine Tests nutzen kann.
Was nun fehlt, sind Aspekte, die auf solchen Instanzen überprüft werden können.

**Theorien** Was nun benötigt wird, sind Annahmen und Regeln, die auf jeder Instanz der Binären Suche gelten. Diese Regeln nennt man Theorien. Schaut man sich die gewollte Ausgabe des Algorithmus an, fallen zwei Theorien sofort auf. Was gilt, wenn die Suche -1 ausgibt und was bedeutet eine Ausgabe, die 0 oder positiv ist?

**Theorie 1:** Wenn die Binäre Suche -1 zurückgibt, dann ist das gesuchte Element nicht im Array

**Theorie 2:** Wenn die Binäre Suche ein nicht negatives $n$ zurückgibt, dann ist das gesuchte Element an Position $n$ vorhanden

Der aufmerksame Leser erkennt, dass mit diesen zwei Theorien die Fälle getestet werden, in denen der Algorithmus eine gültige Ausgabe generiert. Es wäre theoretisch möglich, einen Wert, der kleiner als -1 ist, ausgegeben zu bekommen (zB. wenn der Code eine falsche Ausgabe für Elemente, die nicht im Array sind, generiert).
Ein solcher Test für falsche Eingaben wäre aber recht unsinnig. Eleganter ist es, Theorie 1 und 2 „umzudrehen“. Beide sind von der Form $p \Rightarrow q$. Theorie 3 und 4 werden nun $p \Leftarrow q$ sichern.

**Theorie 3:** Wenn das gesuchte Element nicht im Array ist, dann wird -1 zurückgegeben.

**Theorie 4:** Wenn das gesuchte Element an Position $n$ existiert, dann gibt der Algorithmus $n$ zurück

Mit diesen vier Theorien kann man nun sicherstellen, dass eine korrekt Ausgabe auch korrekt angezeigt wird und umgekehrt (siehe 4.8).

```
public void assertTheory1(int[] testfeld, int target, int returnV) {
    if (returnV == -1)
     assertFalse(arrayContainsTarget(testfeld, target));
}


public void assertTheory2(int[] testfeld, int target, int returnV) {
    if (returnV >= 0)
     assertEquals(target, testfeld[returnV]);
}


public void assertTheory3(int[] testfeld, int target, int returnV) {
    if (!arrayContainsTarget(testfeld, target))
     assertEquals (-1,returnV);
}


public void assertTheory4(int[] testfeld, int target, int returnV) {
    assertEquals(getTargetPosition(testfeld, target), returnV);
}
```

Abbildung 4.8: Methoden für Theorien

**Die verflixte Theorie 4.** Wenn alle Tests nun korrekt programmiert wurden, kann man zuversichtlich sein, dass die Binäre Suche korrekt ist. Allerdings besteht Theorie 4 nicht immer. Die Fehlermeldung signalisiert, dass der Index des gesuchten Elementes manchmal um eine Stelle verschoben gefunden wird.
Um den Fehler zu identifizieren, benötigt man einen Blick auf das Array. Glücklicherweise kann eine Methode von JUnit hier Abhilfe schaffen:

**assertEquals (String, Int, Int)** Vergleich der beiden Integer. Sind diese verschieden, wird ein *Assertion Error* ausgegeben, der das Stringargument mit ausgibt.

Wir initialisieren also einen String mit

String Data = „Array =”+
printArray(testfeld) +
„ target =”+ target;

und rufen dann in Theorie 4

assertEquals($Data$, getTargetPosition($testfeld$, $target$), $returnV$)

auf. Fällt nun ein Test von Theorie 4 durch, so erhält man einen Error der Form

- java.lang.AssertionError: Array =[2,4,11,32,32,55,56,112,123] target = 32
  *expected* :< 3 > but was:< 4 >

Die Fehlermeldung zeigt das Problem: Randomisierte Arrays können gleiche Elemente enthalten. Das wäre bis zu diesem Punkt nur Wenigen aufgefallen. Allerdings ist zu erwähnen, dass nicht die Binäre Suche falsch ist (sie tut ja genau das, was sie soll). Vielmehr haben wir Theorie 4 zu implizit aufgestellt (wir haben ja einfach Theorie 2 umgedreht). Richtiger wäre eine Theorie der Form:

**Theorie 4':** Wenn das gesuchte Element an den Positionen $x_1$ bis $x_n$ existiert, dann gibt der Algorithmus **ein** $x_i$ zurück

Diese Theorie lässt sich aber mit der normalen Binären Suche schlecht testen und bleibt daher dem Leser überlassen. Man müsste anstatt einer Zahl ein Array mit allen gefundenen Positionen zurückgeben.
Die falsche Theorie 4 zeigt jedoch, dass nicht nur Code-Programmierung falsch verlaufen kann. Auch Tests müssen einer gewissen Kontrolle unterliegen, um korrekt zu funktionieren.

## 4.6 Performance-Tests

**O-Notation.** Die bisherigen Tests überprüften Arrayinstanzen und darauf ausgeführte Suchen. Der letzte Teil des Kapitels geht nun ein wenig auf Laufzeiten ein.
Klar ist, dass die Binäre Suche in $O(\log_2 n)$ eine Ausgabe erzeugt. Das kann man sich zu Nutzen machen, indem man die Laufzeit unseres Algorithmus errechnet. Benötigt dieser länger als $O(\log_2 n)$ Schritte, dann stimmt etwas am Code nicht. Damit ist ein solcher Test weniger zur Fehlersuche geeignet, als viel mehr für mehr Zuversicht, dass der Algorithmus richtig läuft.
Ein einfacher Ansatz, die Laufzeit zu berechnen, ist es, die Binäre Suche so zu modifizieren, dass sie am Ende nicht die Position des Suchelementes, sondern die Anzahl der Durchläufe ausgibt (Abbilung 4.9)

```
int comparisonCount = 0;
while(left <= right) {
    comparisonCount++;
    [...];
}
[...];
return comparisonCount;
```

Abbildung 4.9: Codefragment der modifizieren Binären Suche

Überschreitet ein Rückgabewert nun die obere Grenze, dann geht etwas in der Errechnung schief und der Code ist somit nicht fehlerfrei.
Für dieses Testszenario lässt sich eine fünfte Theorie aufstellen (siehe 4.10).

**Theorie 5:** Wenn das Array die Größe $n$ besitzt, dann gibt die modifizierte Binäre Suche einen Wert $<= \log_2 n$ zurück

```
public void assertTheory5(int[] testfeld, int target) {
    int comparisons = Util.Arrays.binarySearchComparisonCount(testfeld, target);
    assertTrue(comparisons <= 1 + log2(testfeld.length));
}
```

Abbildung 4.10: Theorie 5 mit modifizierter Binärer Suche

Dem aufmerksame Leser wird eine Addition von 1 zur oberen Grenze auffallen. Im Nachhinein wird es aber klar, warum der Vergleichswert um 1 erhöht werden muss (vgl Binärer Baum, ein Baum von zB. 8 Elementen benötigt unter Umständen 4 Vergleiche statt 3).
Theorie 5 ist also das Resultat aus dem Wissen, mit welcher Laufzeit die Binäre Suche rechnet. Desweiteren kann man also mit diesem Test wieder ein Stück mehr Zuversicht gewinnen, dass wir etwas richtig machen. Kein unerheblicher Schritt, wenn man bedenkt, dass allein von Erkennung des Fehlers in der Binären Suche bis zu seiner Behebung 12 Jahre vergangen sind.

## 4.7 Zusammenfassung der Tests

- Die Binäre Suche zeigt, dass auch ein sehr einfacher Code eine Menge Probleme bereiten kann.
- Um sicher zu sein, dass Code richtig funktioniert, muss man testen.
- Manche Tests erschließen sich erst, wenn man andere Tests geschrieben hat.
- Testen kann den Code eleganter machen, Fehler beheben oder sicher stellen, dass der Code richtig funktioniert.
- Auch Tests können fehlschlagen. Es lohnt sich also auch, trivial vermutende Annahmen zu testen.

# KAPITEL 5

# Kolnlinearität von 3 Punkten

**Ausarbeitung von Daniel Hüsmert**

## 5.1 Problemstellung

Gegeben sind drei Punkte in einer Ebene. Die Punkte sollen also auf Kolineariät getestet werden.Es soll also überprüft werden, ob diese Punkte Elemente einer lineare Funktion sind. Dies führt uns zu der Frage: Wie sind lineare Funktionen deffiniert ?

**Lineare Funktion** Eine lineare Funktion ist eine Gerade, die die folgende allgemeine Funktionsvorschrift besitzt: $y = m * x + n$ mit $m$ als Steigung der Geraden und $n$ als Y-Achsenabschnitt.

## 5.2 Algorithmus 1

Die Definition der Problemstellung liefert auch die Idee für den ersten, naiven Algorithmus, der das Problem in zwei Schritten löst. Die Eingabe besteht aus drei Punkten P,Q und R. Der Algorithmus bestimmt aus den ersten beiden Punkten eine eindeutige lineare Funktion und überprüft dann, ob der dritte Punkt die Funktionsvorschrift erfüllt.

### 5.2.1 Vorgehensweise des Algorithmus

**Schritt A** In diesem Schritt bestimmt man eine Gerade zwischen den Punkten P und Q. Es werden also die Steigung $m$ und der Y-Achsenabschnitt $n$ berechtnet und somit die Funktionsvorschrift ermittelt. Die Steigung berechnet sich mit Hilfe der Formel
$m = \frac{\triangle Y}{\triangle X}$ mit $\triangle Y$ als Differenz der Y-Werte der beiden Punkte P und Q.

Den Y-Achsenabschnitt bestimmen wir, indem wir den Punkt P und die die errechnete Steigung m in die Funktionsvorschrift einsetzten und nach $n$ auflösen. $n = y - m * x$
Somit erhalten wir die eine vollständige Beschreibung unserer linearen Funktion der Form $y = m * x + n$ mit $n$ und $m$ bekannt.

**Schritt B** In diesem Schritt wird getestet, ob der dritte Punkt R die vorher berechnete Funktionsvorschrift erfüllt indem wir den Punkt in die Gleichung einsetzten. $y_r = m * x_r + n$
Wenn diese Gleichung erfüllt ist, sind die Punkte P,Q und R kolinear.

### 5.2.2 Java-Quellcode

Listing 5.1: Implementierung des ersten Algorithmus

```
public static boolean algo01 (int px,int py,int qx,int qy,int rx,int ry) {
//Steigung
        double m;
        m = (py-qy)/(px-qx);
// Y-Achsenabschitt
        double n;
        n = py - (m*px) ;
//pruefe ob der dritte Punkt auf der Gerade liegt
        boolean b;
        b = (ry == m*rx +n );
        return b;
        }
```

### 5.2.3 Problemfälle und Beispiel

An einem simplem Beispiel erkennen wir aber die eingeschränkten Möglichkeiten des Algorithmus. Wählen wir die Punkte P (3| 5), Q(3|6) und R(3|7), offensichtlich liegen diese Punkte auf einer Geraden parallel zur Y-Achse. Wenn wir den Algorithmus auf diese Punkte anwenden, entsteht ein Fehler bei der Berechnung der Steigung.
$m = \frac{\triangle Y}{\triangle X}$

In den konkreten Fall:

$\frac{6-5}{3-3} = \frac{1}{0}$. Die Division durch Null ist nicht definiert und somit müssen wir erkennen, dass wir eine neue Idee für einen Algorithmus benötigen.

## 5.3 Algorithmus 2

Wir werden schnell feststellen, dass wir nicht die gesamte Funktionsvorschrift aufstellen müssen.Die Eingabe besteht aus drei Punkten P, Q und R. Es wird ausreichen, zwei Steigungen auszurechnen und diese miteinander zu vergleichen. Auch dieser zweite Algorithmus wird in zwei Phasen unterteilt.

### 5.3.1 Vorgehensweise des Algorithmus

**Schritt A** In diesem Schritt werden zwei Steigungen berechnet.Ausgehend von Punkt P, bestimmt man die Steigung zum Punkt Q und die Steigung zum Punkt R. Somit erhalten wir $m_1$ := Steigung zwischen P und Q, sowie $m_2$ := Steigung zwischen P und R.

**Schritt B** Vergleiche die Steigungen $m_1$ und $m_2$. Wenn diese identisch sind, müssen die Punkte auf einer Geraden liegen und sind somit kolinear.

### 5.3.2 Java-Quellcode

Listing 5.2: Implementierung des zweiten Algorithmus

```
public static boolean steigung (int px, int py,int qx,int qy,int rx,int ry){
        double m1,m2;
        boolean b;
        //Steigungen berechnen
        m1 = berechne_stg(px,qx,py,qy);
        m2 = berechne_stg (px,rx,py,ry);
        //Vergleiche die Steigungen
        b=(m1 == m2);
        return b;
        }

        //Methode zum Berechnen der Steigungen
public double berechne_stg (int x1, int x2,int y1,int y2){
                        double a;
                        if (x1 == x2){a=0.5/0.0;}else {
                                a= (y2-y1)/(x2-x1);
                        }
                        return a;
                }
```

Auch dieser Algorithmus ist leicht zu implementieren. Um eine Division durch Null zu verhindern, fangen wir diesen Fall durch einige *IF-Anweisungen* ab.

### 5.3.3 Problemfälle und Beispiel

Für triviale Beispiele arbeitet der Algorithmus fehlerfrei.Wenn wir Spezialfälle betrachtet, stoßen wir schnell auf Probleme. Wählen wir die Punkte so, dass zwei davon identisch sind,

wir also eigentlich nur 2 Punkte gegeben haben. Diese können nur kolinear sein. P(3|5), Q(3|5) und R(4|6). Beim der Berechnung von $m_1$ würde der Algorithmus einen Wert von *positiv infinity* liefern. Die Steigung $m_2$ hätte den Wert 1. Im *Schritt B* würde der Vergleich negativ ausfallen, obwohl die Punkte kolinear sind. Wir erkennen das Problem liegt bei der Berechung einer Steigung, also suchen wir einen Algorithmus der einen anderen Ansatz benutzt.

## 5.4 Algorithmus 3

Gehen wir von der Kontrapostition aus. Die drei gegebene Punkte A, B und C sind nicht kolinear und bilden somit ein Dreieck (vgl.5.1).

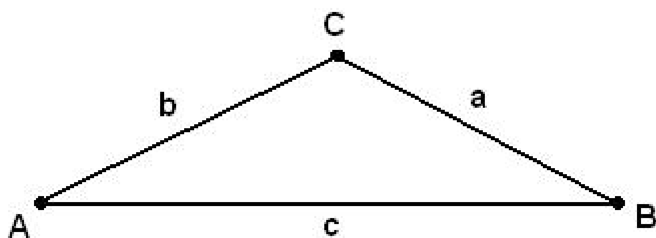



Abbildung 5.1: Punkte bilden ein Dreieck

**Eigenschaften eines Dreiecks** Die Eigenschaft die wir ausnutzen wollen ist, dass die Summe $S$ der Längen der beiden kürzeren Seiten ($a$ und $b$ ) größer ist, als der Betrag $E$ der Längsten Seite ($c$ vgl.5.1).

Wenn wir $S$ mit $E$ vergleichen und diese den selben Wert aufweisen, gibt es nur eine Möglichkeit wie die Punkte angeordent seien können. Sie liegen auf einer Geraden und wären somit kolinear.

### 5.4.1 Vorgehensweise des Algorithmus

Zuerst berechnen wir die Länge der Strecken $|\overline{AB}|$, $|\overline{BC}|$ und $|\overline{CA}|$. Danach sortieren wir sie der Größe nach und bilden die Summe $E$, indem wir die kürzeren Seiten addieren. Im letzten Schritt vergleichen wir $E$ mit der betragsmäßig größten Seite. Ist der Vergleich positiv, sind also beide Werte gleich, sind die Punkte kolinear.

**Wie bestimme ich den Abstand zwischen zwei Punkten A und B?** Hierzu werden wir den *Euklid'schen Abstand* berechnen. Gegeben sind zwei Punkte A($x_a|y_a$) und B ($x_b|y_b$)
$d = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$

### 5.4.2 Java-Quellcode

Listing 5.3: Implementierung des dritten Algorithmus

```
public static boolean flaeche (int px, int py,int qx,int qy,int rx,int ry){
        double []a = new double[3];
        // Abstand zwischen P und Q
        a[0] = abstand(px,py,qx,qy);
        //Abstand zwischen Q und R
        a[1] = abstand(qx,qy,rx,ry);
        //Abstand zwischen P und R
        a[2] = abstand (px,py,rx,ry);
        java.util.sort(a);
        boolean b = (a[0]==a[1]+a[2]);
        return b;
        }

public static double abstand(int px,int py,int qx,int qy)      {
        double d =0;
        d = java.lang.Math.sqrt (((px-qx)*(px-qx))+((py-qy)*(py-qy))) ;
        return d;
}}
```

Hier werden zwei Methoden implementiert. Die Methode *abstand* berechnet den Abstand zwischen zwei Punkten, der Vergleich findet in der Methode *flaeche* statt.Diese Methode liefert den Wert *true* zurück, falls die Punkte kolinear sind.

### 5.4.3 Problemfälle und Beispiel

Wählen wir die Punkte A(12), B(23) und C (34), die die Eigenschaft der Kolinearität erfüllen. Berechnen wir die Abstände erhalten wird:
d(AB)= $\sqrt{2}$,
d(BC)= $\sqrt{2}$
d(CD)= $\sqrt{8} = 2 \cdot \sqrt{2}$
Allerdings wird in dem Datentyp *float* anstatt der reelen Zahl $\sqrt{2}$ und $\sqrt{8}$ die Dezimalzahl 1.414213562373095*1* und 2.828427124746190*3* gespeichert. Bei dem Vergleich der beiden Zahlen würde die Methode *false* zurückliefern, obwohl die Punkte kolinear sind.

## 5.5 Algorithmus 4

Bei diesem Algorithmus gehen wir davon aus, das die drei gegebenen Punkte nicht kolinear sind. Also können wir aus den drei Punkten zwei Geraden bilden, die ein Parallelogramm aufspannen.Der Flächeninhalt eines dieses Parallelogramms kann nur Null betragen, falls der Punkt R auf der Gerade PQ liegt. Die Punkte P,Q und R wären kolinear (5.2).

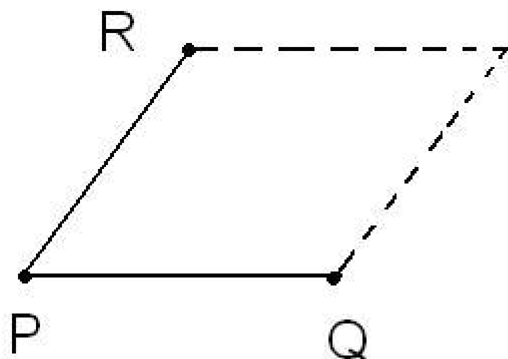



Abbildung 5.2:

**Flächeninhalt einen Parallelogramms I** Um den Flächeninhalt eines Parallelogramms zu berechnen wenden wir folgende Formel an: $A = g \cdot h$ mit $A$ als Flächeninhalt, $g$ als Betrag der Grundseite des Parallelogramms und $h$ als Betrag der Höhe. Allerdings müssen wir bei der Vorgehensweise wieder Beträge berechnen. Das führte bereits beim dritten Algorithmus zu Problemen.

**Flächeninhalt einen Parallelogramms II** Wir betrachen die Geraden als Vektoren und setzten diese in die Formel für den Flächeninhalt ein: $A = |\vec{a}| \cdot |\vec{h}|$. Drücken wir nun die Höhe $h$ mit Hilfe des Winkel $\Phi$ aus (5.3) $|\vec{h}| = |\vec{b}| \cdot \sin(\Phi)$. Setzten wir das nun in die Formel ein, erhalten wir als Flächeninhalt: $A = ||\vec{a}| \cdot |\vec{b}| \cdot \sin(\Phi)|$

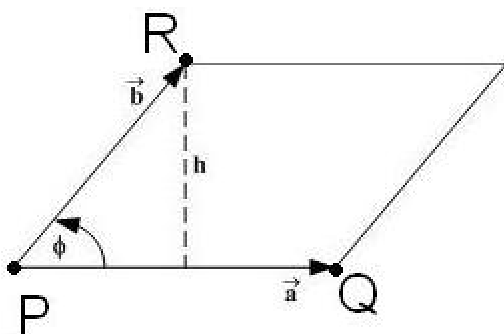



Abbildung 5.3: $h$ mit Hilfe des Winkel $\Phi$

**Kreuzprodukt** Das Kreuzprodukt zweier Vektoren $\vec{a}$ und $\vec{b}$ im *dreidimensionalen*, reellen Vektorraum ist ein Vektor, der senkrecht auf der von den beiden Vektoren aufgespannten Ebene steht. Die Länge dieses Vektors ist proportional zur Fläche des Parallelogramms mit den Seiten $|\vec{a}|$und $|\vec{b}|$ . Allerdings sind die gegebenen Punkte nur zweidimensional.Dies lösen wir, indem wir eine dritte Dimension hinzufügen und diese mit Null belegen. $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$

Allgemein ist das Kreuzprodukt definiert als:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 \cdot b_3 - a_3 \cdot b_2 \\ a_3 \cdot b_1 - a_1 \cdot b_3 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix}$$

In unserem Sonderfall vereinfacht sich einiges.

$$\begin{pmatrix} a_1 \\ a_2 \\ 0 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ 0 \end{pmatrix} = \begin{pmatrix} a_2 \cdot 0 - 0 \cdot b_2 \\ 0 \cdot b_1 - a_1 \cdot 0 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix}$$

Der Betrag dieses Vektors lässt sich leicht ermitteln:

$$\left| \begin{pmatrix} 0 \\ 0 \\ a_1 \cdot b_2 - a_2 \cdot b_1 \end{pmatrix} \right| = \sqrt{0^2 + 0^2 + (a_1 \cdot b_2 - a_2 \cdot b_1)^2} = (a_1 \cdot b_2 - a_2 \cdot b_1)$$

Wir sehen, das wir den Betrag des Vektors berechnen können, ohne eine Wurzel zu ziehen.

### 5.5.1 Vorgehensweise des Algorithmus

Aus den drei gegebenen Punkten berechnen wir

1. den Vektor $\vec{a}$ von Punkt P($P_x|P_y$) nach Punkt Q($Q_x|Q_y$): $\vec{a} = \begin{pmatrix} Q_x - P_x \\ Q_y - P_y \end{pmatrix}$

2. den Vektor $\vec{b}$ von Punkt P($P_x|P_y$) nach Punkt R($R_x|R_y$): $\vec{b} = \begin{pmatrix} R_x - P_x \\ R_y - P_y \end{pmatrix}$

Dann berechnen wir aus den Vektoren $\vec{a}$ und $\vec{b}$ den Betrag des Kreuzproduktes und damit den Flächeninhalt des Parallelogramms, das von den Vektoren $\vec{a}$ und $\vec{b}$ aufgespannt wird. Wenn dieser Betrag Null ergibt, sind die drei Punkte kolinear.Wir überprüfen folgende Aussage: $|\vec{a} \times \vec{b}| = (a_1 \cdot b_2 - a_2 \cdot b_1) = 0$, durch Umstellen der Gleichung erhalten wir folgenden Ausdruck: $a_1 \cdot b_2 = a_2 \cdot b_1$

### 5.5.2 Java

Listing 5.4: Implementierung des vierten Algorithmus

```
public  boolean flaecheninhalt(int px, int py,int qx,int qy,int rx,int ry){
        boolean b =((qx-px)*(ry-py)==(rx-px)*(qy-py));
        return b;

}
```

Wie wir sehen, besteht der gesamte Code aus wenigen, verständlichen Zeilen. Im Vergleich zu den anderen, von mir vorgestellten Algorithmen, ist dieser der eleganteste!

### 5.5.3 Beispiele

**Beispiel 1** Die drei Punkte sind P(2|5), Q(3|6)und R(4|7). Unser Algorithmus auf die Punkte angewandt ergibt folgendes:
$(3-2)\cdot(7-5) = (4-2)\cdot(6-5)$
$\Leftrightarrow 1\cdot 2 = 2\cdot 1$
$\Leftrightarrow 2 = 2$ dies ist offensichtlich wahr und damit sind die Punkte kolinear.

**Beispiel 2** Nehmen wir an unsere Punkte seien P(5|8) Q(5|8) R(6|9). Zwei Punkte sind identisch. Unser Algorithmus auf diese Punkte angewandt ergibt folgendes:
$(5-5)\cdot(9-8) = (6-5)\cdot(8-8)$
$\Leftrightarrow 0\cdot 1 = 1\cdot 0$
$\Leftrightarrow 0 = 0.$
Der Algorithmus klassifiziert diese, sowie alle anderen Eingabe richtig.

## 5.6 Quellenverzeichnis


- beautiful code, Andrew Oram & Greg Wilson , O'Reilly Media
- frei Enzyklopädie Wikipedia
  http://de.wikipedia.org/wiki/Kreuzprodukt

KAPITEL 6

# Anzahl der 1-Bits in einem Integer

**Ausarbeitung von Paul Ruppersberger**

## 6.1 Einleitung

Dieses Kapitel widmet sich der Bestimmung der auf 1 gesetzten Bits in einem Integer, dem so genannten „Population Count“ (Im folgenden nur als PopCount bezeichnet). Es wird insbesonders auf Algorithmen zur Berechnung des PopCount und zur Auswertung des PopCount eingegangen. Dazu wird zuerst ein naiver Algorithmus vorgestellt, der im weiteren Verlauf weiter optimiert wird. Grundlegend wird zunächst geklärt warum ein effizienter Algorithmus zur Berechnung des PopCounts überhaupt gebraucht wird.

**Anwendungen** Der PopCount wird vor allem in der Kodierungs- und Informationstheorie benötigt. Hier hauptsächlich zur Feststellung und Korrektur von Bitfehlern. Dies geschieht anhand des Hamming-Abstands zweier Wörter (Anzahl der Stellen an denen die Wörter unterschiedlich sind). Der Hamming-Abstand dann entspricht der Anzahl der 1-Bits des Terms $x \oplus y$. Aufgrund des engen Zusammenhangs zwischen PopCount und Hamming-Abstand wird der PopCount oft auch als Hamming-Gewicht bezeichnet (Es wird in diesem Kapitel nicht weiter auf den Hamming-Abstand eingegangen). Weitere Anwendungen des PopCounts liegen in der Kryptographie. Hier wird er insbesonders zur Erzeugung von Pseudozufallszahlen verwendet, da es sich um eine Einwegfunktion handelt. D.h. eine Funktion die keine eindeutige Umkehrfunktion hat. Dies ist beim PopCount offensichtlich, da $\text{popcount}(1) = \text{popcount}(2) = 1$ ist.

**Definitionen** In diesem Kapitel ist mit einem Integer ein vorzeichenloser 32-Bit-Integer gemeint. Wenn bei der Analye der Algorithmen die Anzahl der Instruktionen angegeben ist, werden Zuweisungen nicht betrachtet.

## 6.2 Erster Ansatz

Der erste naive Algorithmus den PopCount zu Bestimmen betrachtet alle Bits des Integer und erhöht einen Zähler falls das betrachtete Bit auf 1 gesetzt ist. Die C++ Implementierung ist in Listing 6.1 dargestellt.

Listing 6.1: Implementierung des naiven Algorithmus

```
1 int popcount_1(unsigned int x) {
2     int count = 0;
3     for (int i = 0; i < 32; i++) {
4         if (x & 1) count++;
5         x >>= 1;
6     }
7     return count;
8 }
```

Diese Implementierung enthält sieben Instruktionen innerhalb der For-Schleife, davon zwei Sprünge. Zur Analyse werden folgende Fälle betrachtet:

$$
\begin{aligned}
\text{Case}_{best} &= 2 + 32 \cdot 6 = 194 \text{ Instruktionen} \\
\text{Case}_{worst} &= 2 + 32 \cdot 7 = 226 \text{ Instruktionen} \\
\text{Case}_{avg} &= 2 + 32 \cdot 6.5 = 210 \text{ Instruktionen}
\end{aligned}
$$

Im besten Fall besteht der Integer aus 32 0-Bits, dann wird die Anweisung für das Inkrementieren des Zählers ausgelassen. Im schlechtesten Fall sind es 32 1-Bits und im durchschnittlichen Fall enthält der Integer 16 1-Bits. Dieser Algorithmus ist sicherlich nicht sehr elegant, so kann nach der Betrachtung des höchstwertigsten 1-Bits die Schleife abgebrochen werden. Der Ausdruck $x \wedge 1$ liefert 1, wenn das Bit an der niederwertigsten Stelle 1 ist, ansonsten 0. Dies kann verwendet werden um die if-Anweisung innerhalb der Schleife zu eliminieren. Aus diesen Optimierungsansätzen ergibt sich die in Listing 6.2 gezeigte Implementierung.

Listing 6.2: Implementierung des verbesserten naiven Algorithmus

```
1 int popcount_2(unsigned int x) {
2     int count = 0;
3     while (x) {
4         count += (x & 1);
5         x >>= 1;
6     }
7     return count;
8 }
```

Die Schleife enthält jetzt nur noch 5 Instruktionen, davon einen Sprung. Der beste Fall ist trivial bei $x = 0$. Es ist nur die Überprüfung und der Sprung ans Ende der Schleife auszuführen. Der schlechteste Fall benötigt nur noch $\text{Case}_{worst} = 2 + 32 \cdot 5 = 162$ Instruktionen. Einen durchschnittlichen Fall anzugeben fällt schwer, da nicht nur die Anzahl der 1-Bits, sondern auch ihre Position innerhalb des Integers von Bedeutung ist. Daher wird dieser Fall hier nicht betrachtet, da jetzt eine effizientere und elegantere Implementierung betrachtet wird.

Der Ausdruck $x \wedge (x-1)$ liefert $x$ mit dem niederwertigsten 1-Bit auf 0 gesetzt. Dies wird am folgenden Beispiel verdeutlicht:

$$\begin{aligned} x &= 00100110 \\ x-1 &= 00100101 \\ x \wedge (x-1) &= 00100100 \end{aligned}$$

Dieser Ausdruck führt dazu, dass die Schleife nur noch so oft durchlaufen wird bis alle 1-Bits in $x$ eliminiert sind. Die Implementierung ist Listing 6.3 zu entnehmen.

Listing 6.3: Implementierung des optimierten Algorithmus

```
1 int popcount_3(unsigned int x) {
2     int count = 0;
3     while (x) {
4         count++;
5         x &= (x - 1);
6     }
7     return count;
8 }
```

Der beste und schlechteste Fall bleiben dabei gleich. Hier kann auch wieder ein durchschnittlicher Fall angegeben werden, der wie bei der ersten Version des Algorithmus bei 16 1-Bits eintritt; $\text{Case}_{avg} = 2 + 16 \cdot 5 = 82$ Instruktionen. Daraus ergibt sich die Laufzeitfunktion

$$f(n) = 2 + 5n \tag{6.1}$$

wobei $n$ der Anzahl der 1-Bits entspricht.

## 6.3 Divide-And-Conquer Ansatz

Wie in vielen Bereichen der Algorithmik kann auch in diesem Fall zur Lösung des Problems über eine Lösung nach dem Divide-And-Conquer Prinzip nachgedacht werden. Dazu sind einige Vorüberlegungen notwendig.

Angenommen der PopCount soll für ein 32-Bit Integer berechnet werden und es existiert ein Algorithmus der den PopCount für einen 16-Bit Integer berechnen kann. Dann kann der PopCount für den 32-Bit Integer durch Anwenung dieses Algorithmus auf die linken und die rechten 16-Bit des Integers (Divide-Schritt) und anschließendes Addieren der Ergebnisse (Conquer-Schritt) berechnet werden. Dieses Vorgehen liefert aber leider bei sequentieller Ausführung auf beiden Hälften keinen Geschwindigkeitsvorteil. Hier ist die Zeit, die zur Durchführung der Berechnung benötigt wird, proportional zur Breite des Integers (Bei einem 32-Bit Integer also $32k$ mit $k$ als Proportionalitätskonstante). Bei Aufteilung in zwei 16-Bit breite Teile und anschlieender Addition ergibt sich daraus:

$$16k + 16k + 1 = 32k + 1 \tag{6.2}$$

Einen Zeitvorteil wird erst dann erzielt, wenn es möglich ist die Berechnung der beiden Hälften parallel durchzuführen. Dadurch wird der Aufwand von $32k$ auf $16k + 1$ reduziert. Durch weitere Aufteilung in 8-, 4-, 2- und 1-Bit große Teile wird der triviale Fall bei einer Breite von ein Bit erreicht. Hier ist der Wert des Bits auch der PopCount. Das Prinzip wird duch Abbildung 6.1 veranschaulicht.

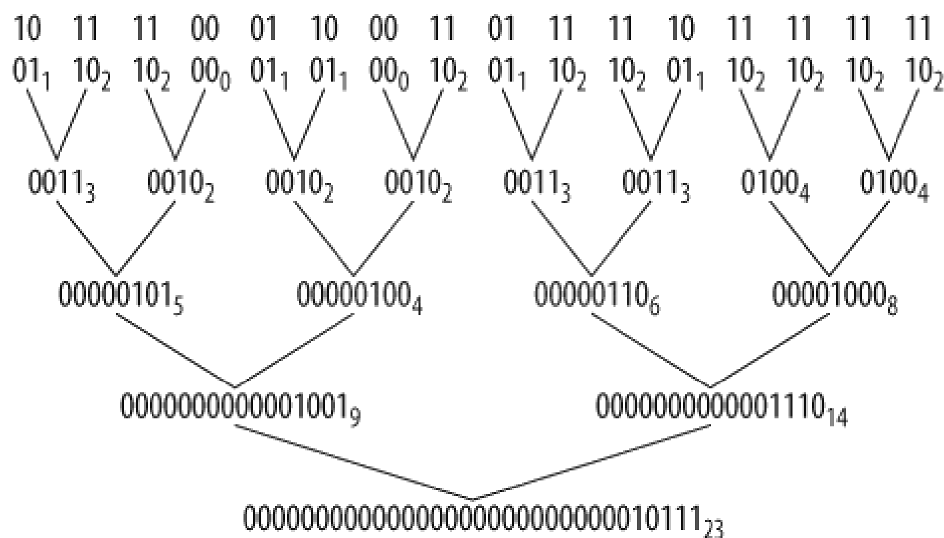



Abbildung 6.1: Divide-And-Conquer Ansatz (Wilson and Oram, 2007b, Kapitel 10.2)

Die erste Zeile der Abbildung 6.1 enthält den Integer selbst. In der zweiten Zeile ist dann die erste Stufe des Algorithmus ausgeführt worden. Es werden jeweils die Anzahl der 1-Bits von jeweils zwei Bit in diese zwei Bit geschrieben. Bei der nächsten Stufe werden dann zwei dieser Zweiergruppen zusammengefasst, so dass jeweils Gruppen von vier Bit die Anzahl der 1-Bits in diesen vier Bit enthalten. Dieses wird dann fortgesetzt bis nur noch ein Teil der Breite 32 existiert der das Endergebnis enthält. Eine erste einfache Implementierung sieht wie in Listing 6.4 dargestellt aus.

Listing 6.4: Divide-And-Conquer Algorithmus (frei nach Wikipedia (2008b))

```
1  const unsigned int m1  = 0x55555555;
2  const unsigned int m2  = 0x33333333;
3  const unsigned int m4  = 0x0F0F0F0F;
4  const unsigned int m8  = 0x00FF00FF;
5  const unsigned int m16 = 0x0000FFFF;
6
7  int popcount_4(unsigned int x) {
8      x = (x & m1 ) + ((x >>  1) & m1 );
9      x = (x & m2 ) + ((x >>  2) & m2 );
10     x = (x & m4 ) + ((x >>  4) & m4 );
11     x = (x & m8 ) + ((x >>  8) & m8 );
12     x = (x & m16) + ((x >> 16) & m16);
13     return x;
14 }
```

Da dieser Algorithmus keine Schleife benötigt wird eine konstante Laufzeit erreicht. Es werden insgesamt nur 20 Instruktionen für die Berechnung des PopCount benötigt. Allerdings

gibt es auch hier noch Optimierungspotenzial. Bei näherer Betrachtung fällt auf, dass der Ausdruck $x >> 16$ mit 16 0-Bits beginnt und somit die AND-Operation mit der Maske 0x0000FFFF keine Auswirkung mehr hat. Ebenso können AND-Operationen eingespart werden bei denen kein Überlauf bei der Zusammenfassung von zwei Bit-Gruppen auftreten. Die Implementierung dieses Algorithmus ist Listing 6.5 zu entnehmen.

Listing 6.5: Optimierung des Divide-And-Conquer Algorithmus (Wilson and Oram (2007b))

```
const unsigned int m1 = 0x55555555;
const unsigned int m2 = 0x33333333;
const unsigned int m4 = 0x0F0F0F0F;

int popcount_4(unsigned int x) {
    x = x - ((x >> 1) & m1;
    x = (x & m2) + ((x >>  2) & m2);
    x = (x + (x >> 4)) & m4;
    x = x + (x >> 8 );
    x = x + (x >> 16);
    return x & 0x3F; // 00111111
}
```

Die Zeile 6 der Funktion aus Listing 6.5 basiert auf den ersten beiden Termen der Gleichung

$$\text{popcount}\,(x) = x - \lfloor\frac{x}{2}\rfloor - \lfloor\frac{x}{4}\rfloor - \cdots - \lfloor\frac{x}{2^{31}}\rfloor \tag{6.3}$$

und schreibt die Anzahl der 1-Bits die in diesen zwei Bit enthalten sind in diese zwei Bit (Vgl. Abbildung 6.1). Der Beweis dieser Gleichung wird an dieser Stelle nicht geführt. Diese Implementierung erreicht die konstante Laufzeitfunktion von $f(n) = 15$.

## 6.4 Rechnen mit PopCounts

**Summenbildung** Um die Summe von zwei PopCounts zu bilden ist der erste ersichtliche Ansatz sicherlich $\text{popcount}\,(x) + popcount(y)$. Es werden dazu $2 * 15 + 1$ Instruktionen benötigt. Es gibt jedoch eine noch effizientere Methode. Diese verwendet den bereits erläuterten Divide-And-Conquer Algorithmus. Bei Betrachtung der Stufe, in der jeweils die Anzahl der 1-Bits in Gruppen aus vier Bit gebildet werden, fällt auf, dass in diesen vier Bit maximal die Zahl 4 hineingeschrieben wird (Maximaler PopCount auf dieser Stufe). Vier Bit bieten jedoch Platz für einen maximalen PopCount von 15. Es wird also mit $x$ und $y$ bis zur zweiten Stufe getrennt verfahren und anschließend werden beide Zahlen addiert. Das Ergebnis der Addition enthält dann die Anzahl der 1-Bits, die $x$ und $y$ (zusammen) in je vier Bit haben. Jetzt wird genau wie bei der normalen PopCount-Berechnung verfahren. Die Implementierung ist Listing 6.6 zu entnehmen. Es werden statt 31 nur noch 24 Instruktionen benötigt. Dieser Algorithmus funktioniert auch problemlos zur Addition von drei Integern.

**Differenzbildung** Der Ansatz bei der Subtraktion zweier PopCount ist ähnlich der bei der Addition. Es wird zunächst getrennt von einander der PopCount von $x$ und $y$ berechnet

Listing 6.6: Implementierung des Additionsalgorithmus

```
int popadd(unsigned int x, unsigned int y) {
    x = x - ((x >> 1) & m1;
    x = (x & m2) + ((x >>  2) & m2);
    y = y - ((y >> 1) & m1;
    y = (y & m2) + ((y >>  2) & m2);
    x = x + y;
    x = (x & m4) + ((x >>  4) & m4);
    x = x + (x >> 8 );
    x = x + (x >> 16);
    return x & 0x3F; // 00111111
}
```

und Anschließend werden die Ergebnisse subtrahiert. Aber auch hier kann eleganter verfahren werden. Dazu wird ausgenutzt, dass der PopCount eines Integers gleich 32 minus dem Einerkomplement des Integers ist.

$$\text{popcount}\,(x) = 32 - \text{popcount}\,(\bar{x}) \tag{6.4}$$

Daraus leitet sich dann für die Substraktion folgende Formel ab:

$$\begin{aligned}\text{popcount}\,(x) - \text{popcount}\,(y) &= \text{popcount}\,(x) - (32 - \text{popcount}\,(\bar{y})) \\ \text{popcount}\,(x) - \text{popcount}\,(y) &= \text{popcount}\,(x) + \text{popcount}\,(\bar{y}) - 32\end{aligned}$$

Es kann für die Substraktion jetzt der gleiche Grundalgorithmus wie für die Addition verwendet werden. Es sind nur zwei weitere Instruktionen notwending, eine für die Invertierung des Integers und eine Subtraktion.

**Vergleich zweier PopCounts** Um die PopCounts zweier Integer zu vergleichen, also herauszufinden welcher Integer mehr 1-Bits enthält, gibt es zwei Methoden. Bei der Ersten wird einfach die Differenz gebildet und dann das Ergebnis untersucht. Bei der zweiten Methode werden zuerst 1-Bits, die in beiden Integers gesetzt sind, auf 0 gesetzt. Anschließend werden beide Integer solange um je ein 1-Bit reduziert bis einer den Wert 0 hat. Der Integer der zuerst keine 1-Bits mehr enthlt ist dann der Integer mit dem kleineren PopCount. Die Implementierung zeigt Listing 6.7.

Listing 6.7: Implementierung des Vergleichalgorithmus (Wilson and Oram (2007b))

```
int popcmp(unsigned int ax, unsigned int ay) {
    unsigned int x = ax & ~ay;
    unsigned int y = ay & ~ax;
    while(true) {
        if (x == 0) return y | -y;
        if (y == 0) return 1;
        x = x & (x-1);
        y = y & (y-1);
    }
}
```

## 6.5 PopCount eines Arrays

Um den PopCount eines Arrays von Integern zu berechnen wird zunächst die Möglichkeit in Betracht gezogen den PopCount jedes Array-Elements zu bestimmen und die Einzelergebnisse dann zu addieren. Doch wie bereits bei der Addition gesehen existiert ein Algorithmus, der die PopCount-Summe von drei Integern effizient berechnet. Dieser kann auch zur Berechnung des Array-PopCounts herangezogen werden. Dazu wird dieser mit drei Array-Elementen aufgerufen und das Ergebnis zum Gesamtergebnis hinzuaddiert. Bei Array-Größen die sich nicht ohne Rest durch drei teilen lassen, müssen die PopCounts der verbliebenden Elemente mit der herkömmlichen Weise hinzuaddiert werden. Listing 6.8 zeigt wie eine Implementierung ausehen könnte.

Listing 6.8: Implementierung des Array-Algorithmus

```
int poparray(unsigned int a[], int size) {
    int sum = 0;
    int i;

    for (i = 0; i < size-2; i += 3) {
        sum += popadd(a[i], a[i+1], a[i+2]);
    }

    for (; i < size; i++) {
        sum += popcount(a[i]);
    }

    return sum;
}
```

KAPITEL 7

# Aho-Corasick-Automat

**Ausarbeitung von Janet Fiedler**

## 7.1 Einleitung

**Pattern Matching** ist die Mustererkennung bzw. Suche nach Schlüsselwörtern und wird heutzutage in vielen Bereichen der Informatik immer wichtiger.

Stellen wir uns vor, wie wir vorgehen würden, wenn wir in einem uns vorliegenden Text ein Schlüsselwort finden möchten. Wir durchlaufen den Text sozusagen der Reihe nach Buchstabe für Buchstabe und vergleichen mit dem Schlüssewort. Diese Art der Suche nach einem Muster $x$ der Länge $m$ im Text $t$ der Länge $n$ ist der naivste Ansatz des Pattern Matching und ergibt eine Laufzeit von $O(n \cdot m)$ bei $(n - m + 1)$ zu durchsuchenden Stellen im Text.

Einleitend dazu wollen wir uns vorher noch einen verbesserten Ansatz, den Knuth-Morris-Pratt-Algorithmus ansehen um dann ausführlicher auf das eigentliche Thema einzugehen.

### 7.1.1 Knuth-Morris-Pratt-Algorithmus

Im Gegensatz zum naiven Algorithmus speichert der KMP-Algorithmus Informationen über bereits gewonnene Erkenntnisse bei der Zeichensuche ab (Präfixanalyse), so ist es nicht notwendig die Suche nach eine fehlenden Übereinstimmung von vorne zu beginnen, somit werden wiederholte Vergleiche vermieden und die Laufzeit kann auf $O(n + m)$ verbessert werden. Diese Informationen werden in einer Sprungtabelle gespeichert, so dass im Text Zeichen übersprungen werden können, die schon eine Übereinstimmung geliefert haben.

Was ist aber wenn nach mehreren Schlüsselwörtern gesucht werden soll? Bei einer Suchlaufzeit von $O(m + n)$ bei einem Schlüsselwort würde sich dann bei $k$ Schlüsselwörter eine Laufzeit von $O(m + k \cdot n)$ ergeben. Das heißt, dass man den Text für jedes Schlüsselwort

erneut durchlaufen muss. Ziel ist es deshalb den Faktor $k$ zu eliminieren und das Suchen mehrerer Schlüsselwörter gleichzeitig bei einem Suchdurchgang zu ermöglichen.

## 7.2 Aho-Corasick-Algorithmus

Die zwei kanadischen Informatiker Alfred V. Aho und Margeret J. Corasick entwickelten 1975 den Aho-Corasick-Algorithmus, der die gleichzeitige Suche nach mehreren Schlüsselwörtern ermöglicht. Der Algorithmus konstruiert einen deterministischen endlichen Automaten und wird durch folgendes 6-Tupel der Form $(Q, \Sigma, g, f, out, q_0)$ beschrieben:

- $Q$: endliche Menge von Zuständen
- $\Sigma$: endliches Eingabealphabet mit k = $|S|$
- $g$: Übergangsfunktion (Goto)
- $f$: Fehlerfunktion (Failure)
- $o$: Ausgabefunktion (Output)
- $q_0$: Startzustand

Wichtig für die Konstruktion des Automaten sind vor allem die Funktionen g, f und o die wir uns nun genauer ansehen wollen.

### 7.2.1 Die Übergangsfunktion (goto-Funktion)

Die Übergangsfunktion $g$ stellt eine Repräsentation der Schlüsselwörter in einer Baumstruktur dar. Die Konstruktion des Baumes basiert auf folgenden Eigenschaften. Jeder Baum besitzt eine Wurzel, die den Startzustand $q_0$ symbolisiert. Die von $q_0$ ausgehenden Kanten zu weiteren Zuständen sind mit unterschiedlichen Symbolen beschriftet. Dasselbe gilt auch für Knoten tieferer Ebene. Dabei wird jede Kante mit einem Symbol des Eingabealphabets markiert. Die Schlüsselwörter werden gegebenenfalls alphabetisch der Reihe nach eingefügt.

Angenommen es existiert bereits eine Kante, die schon mit dem ersten Symbol des einzufügenden Schlüsselwortes markiert ist, wird diese Kante verwendet und somit in den nächsten schon bestehenden Knoten gewechselt. Eine Verzweigung erfolgt erst wenn der Endknoten des Präfixes erreicht ist.

In der Abbildung 1.1 ist ein Suchbaum mit der Schlüsselwortmenge = { her, their, eye, iris, he, is} abgebildet. Die Suche nach diesen Schlüsselwörtern erfolgt durch Ablaufen der Buchstaben des Suchbaumes. Sehen wir uns das an einem konkreten Textbeispiel in Bezug auf die gegebene Abbildung an.

Textbeispiel 1: *disthero*

Wir befinden uns im Startzustand. Das erste Symbol $d$ wird gelesen. Es wird festgestellt, dass keine Kante mit der entsprechenden Markierung existiert und somit wird im Zustand $q_0$ verblieben. Als nächstes wird $i$ gelesen und eine entsprechende Kante gefunden und in den Zustand 7 gewechselt. Auch $s$ wird als Folgesymbol gefunden und es findet ein Wechsel in Zustand 11 statt. Sobald man in einem Endknoten gelandet ist, hat man ein Schlüsselwort gefunden. Ebenso finden Zustandswechsel bei den Eingaben t, h und e statt.

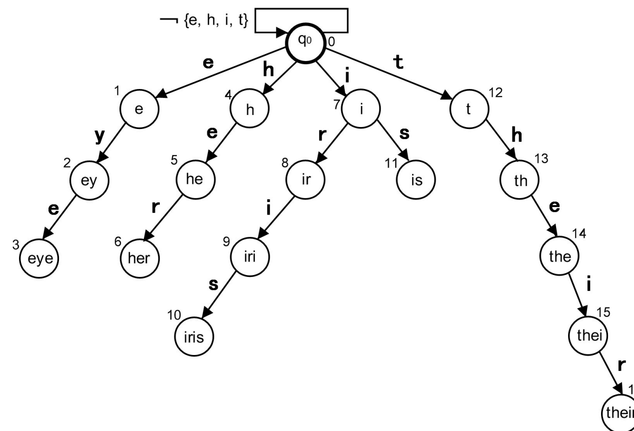



Abbildung 7.1: Graph der Übergangsfunktion

Wir befinden uns jetzt im Zustand 14 und sind sozusagen in einer Sackgasse gelandet, da im nächsten Schritt keine Kante gefunden wird, die die Bezeichnung $o$ hat. Um dieses Problem zu lösen, sehen wir uns die nächste wichtige Funktion an auf die der Aho-Corasick-Automat basiert.

### 7.2.2 Fehlerfunktion (failure-Funktion)

Die Fehlerfunktion definiert die so genannten Fehler-Links, die im Falle des oben gezeigten Problems einen Ausweg aus der Sackgasse angeben. Der Link verweist von Zustandsknoten $v$ auf einen Knoten $w$ im Baum. Dabei bilden die Markierungen der Kanten von der Wurzel bis zu dem Knoten $w$ das längstmögliche Suffix der bereits gefundenen zusammenhängenden Symbole.

Die Fehler-Links der Knoten der Tiefe 1 führen grundsätzlich zur Wurzel zurück. Die Berechung der Links der anderen Knoten wird unter der Beachtung der Reihenfolge der Tiefen durchgeführt. Für die Knoten wird die Fehlerfunktion des Vorgängers in die Übergangsfunktion der Form $g(q, s)$ des aktuellen Zustandes eingesetzt. Deshalb ist es wichtig bei der Berechnung die Reihenfolge zu beachten.

In der Abbildung 1.2 wurden die Links der Fehlerfunktion ergänzt, jedoch nur um die Links, die nicht zur Wurzel führen um eine gewisse Übersichtlichkeit beizubehalten. Die Berechnung wollen wir uns nun noch mal anhand ausgewählter Knoten genauer ansehen.

Die Fehlerfunktion für den Knoten 7 verweist auf die Wurzel. Auch der Link von Knoten 8 landet bei der Wurzel. Formal geschrieben sieht das so aus:

$$f(8)=g(f(7),s)=g(0,s)=0.$$

Das heißt, die Fehlerfunktion des Knoten 8 lässt sich auch durch die Übergangsfunktion beschreiben. Von Zustand 7 ist schon bekannt, dass die Fehlerfunktion zur Wurzel zurückführt. Übrig bleibt noch das gefundene $s$. Von der Wurzel aus gibt es aber keine Kante, die mit

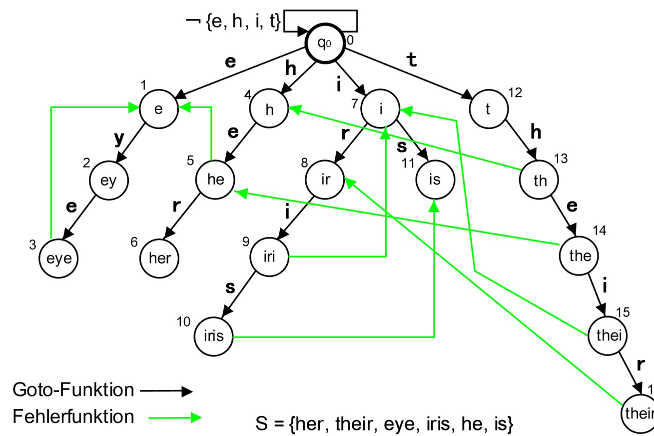



Abbildung 7.2: Failure-Links

einem $s$ markiert ist. Somit verweist der Fehlerlink weiterhin auf den Startzustand. Bei der Fehlerfunktion für Knoten 9 sieht das schon etwas anders aus.

$$f(9)=g(f(8),i)=g(0,i)=7$$

Die Berechnung der Fehlerfunktion von Zustand 8 hat auf den Startzustand verwiesen. Diesmal gibt es aber eine ausgehende Kante mit der Markierung $i$ und somit landet der Fehler-Link bei Zustand 7. Wird also nach dem Finden des Teilwortes $iri$ kein Buchstabe $s$ gefunden, wird so das längste Suffix des Teilwortes ermittelt und weiterverwendet. Die Ergebnisse der Berechnung der Fehler-Links sind in der Tabelle aufgelistet.

| $q$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(q)$ | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 7 | 11 | 0 | 0 | 4 | 5 | 7 | 8 |

### 7.2.3 Ausgabefunktion (output-Funktion)

Die Ausgabefunktion gibt zu jedem Zustand eine Menge von Schlüsselwörtern an, die in diesem Zustand gefunden wurden. Die Konstruktion der Ausgabefunktion erfolgt in zwei Schritten. Bei jeder Intitalisierung der Übergangsfunktion, die nach jedem Lesen eines Symbols des Textes stattfindet, wird überprüft ob die Funktion eine nicht-leere Menge liefert. Ist das der Fall, wurde ein Schlüsselwort vollständig gefunden und somit ausgegeben. Beim Erzeugen der Fehlerfunktion werden die Suffix-Begriffe ebenfalls hinzugefügt, aber nicht ausgegeben. In Abbildung 1.3 sind die Ausgaben durch die Kästchen markiert, somit ist diese Abbildung auch die vollständige Darstellung des Aho-Corasick-Automaten.

## 7.3 Laufzeit

Bei dem Aho-Corasick-Algorithmus ergibt sich eine Gesamtlaufzeit von $O(m+n+k)$, wobei $m$ die Gesamtanzahl der Zeichen aller Schlüsselwörter, $n$ die Länge des Textes und $k$

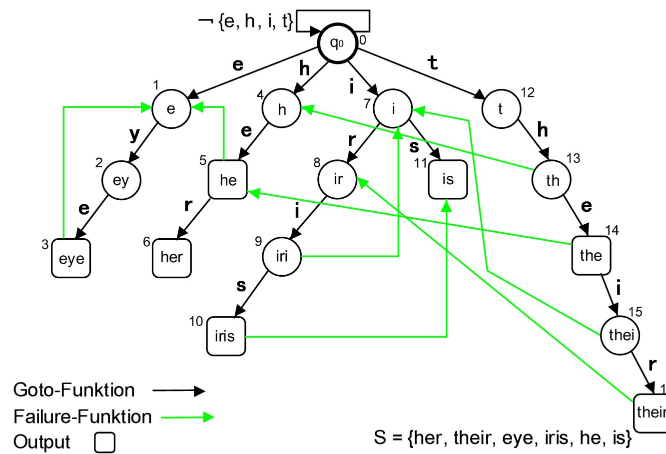



Abbildung 7.3: Aho-Corasick-Automat

die Anzahl der vorkommenden Schlüsselwörter im Text ist. Dabei ist zu beachten das die Übergangsfunktion für jede Schlüsselwortmenge neu erzeugt werden muss, vom zu durchsuchenden Text aber unberührt bleibt und somit unabhängig ist.

## 7.4 Einsatzgebiete des Aho-Corasick-Algorithmus

Ursprünglich entwickelten Aho und Corasick den Algorithmus um vor allem Bibliothekssystemen effizienter zu gestalten. Die Suchlaufzeit konnte so um das 5- bis 10-fache verbessert werden. Nebenbei wurde dadurch auch die Eingabe nach mehreren Suchwörtern ermöglicht. Der Aho-Corasick-Algorithmus hat sich aber auch in vielen anderen Bereichen der Suchanwendungen durchgesetzt. Man findet diesen Algorithmus ebenfalls bei der Verwendung im Unixbefehl *fgrep*. Zum Einsatz kommt der Algorithmus auch in der Bildverarbeitung, wo er besonders beim Bildervergleich sehr nützlich ist. Eine weitere Suchanwendung die auf dem Aho-Corasick-Algorithmus basiert, ist die Erkennung von Virenmustern die sich in unseren Netzwerken befinden. Einzige Einschränkung dabei ist, dass das jeweilige Muster schon in der Datenbank enthalten sein muss. Explizit in der Bioinformatik wird vor allem bei der Durchsuchung der menschlichen DNA dieser Algorithmus verwendet. Aber auch bei vielen anderen Suchen in der Medizin ist der Aho-Corasick-Algorithmus sehr hilfreich, da dort vor allem mit sehr groen Datenmengen gearbeitet wird.

## 7.5 Quellen


- Dori, Shiri und Landau, Gad M. : Construction of the Aho Corasick Automaton in Linear Time for Integer Alphabets
- Manacher, Glenn: Efficient String Matching: An aid to Bibliographic Search
- Müller-Hannemann, M. : Vorlesungsfolien, Lecture 2, 2006, http://zuseex.algo.informatik.tu-darmstadt.de/lehre/2006ss/bioinf/mat/lecture2-p4.pdf
- Wikipedia

KAPITEL 8

# Bit-paralleles Pattern Matching

**Ausarbeitung von Karl Kanafa**

## 8.1 Einleitung

Dieses Kapitel widmet sich dem Problem der Suche nach Mustern mit einer bestimmten Eigenschaft in einer Sequenz von Symbolen, speziell der Suche nach Zeichenketten in einem gegebenen Text. Mit steigender Größe eines Textes, wie zum Beispiel in Datenbanken oder der Bioinformatik, wächst auch das Interesse besonders schnelle und effiziente Algorithmen einzusetzen. Erwartungsgemäß existiert eine Reihe von Lösungen mit verschiedenen Ansätzen. Ich möchte eine Auswahl der simpelsten, aber auch elegantesten Algorithmen vorstellen, die sich die Eigenschaft von Rechnern zu Nutze machen, Bits parallel verarbeiten zu können.

## 8.2 Bit-parallele Operationen

Die Bit-Parallelität beschreibt eine Technik zur gleichzeitigen Rechnerinternen Verarbeitung aller Bits eines Wortes $w$. Als Wort bezeichnet man die Registerlänge der CPU, also die Anzahl von Bits, auf der ein Rechner Operationen ausführt. Mit dieser Technik ist es möglich die Laufzeiteffizienz um den Faktor $|w|$ zu steigern, was sich selbst bei heutigen handelsüblichen PCs mit der Wortlänge $|w| = 32$ Bit bzw. 64 Bit signifikant bemerkbar macht.

Umgesetzt wird diese Technik, indem man logische und arithmetische Operationen, die auf ganzen Registern arbeiten, in die Suchalgorithmen einbettet (siehe Tabelle 8.1). Auf einen Satz von Worten angewendet, ist darauf zu achten, dass sich der Übertrag auf das jeweils nächste Wort fortpflanzt.

| Operation | Java Operator | Beschreibung |
|---|---|---|
| bitweise UND | & | verknüpft alle Bits beider Operanden paarweise mit dem logisch UND |
| bitweise ODER | \| | verknüpft alle Bits beider Operanden paarweise mit dem logisch ODER |
| bitweise exklusiv ODER | ^ | verknüpft alle Bits beider Operanden paarweise mit dem logisch exklusiv ODER |
| Bitkomplement | ~ | invertiert alle Bits des Operanden |
| Linksschieben | << | Linksschieben ( shift ) aller Bits, füllt mit Nullen auf |
| Rechtsschieben | >> (>>>) | Rechtsschieben aller Bits, füllt mit Vorzeichen (Nullen) auf |
| Addition | + | |
| Subtraktion | − | |

Tabelle 8.1: Bit-Parallelle Operationen in Java

## 8.3 Grundlegende Konzepte

Die Beschreibungen der Algorithmen stützen sich auf einige Begriffe und Bezeichnungen, welche wir in diesem Abschnitt vereinbaren wollen.

Gesucht wird ein String $P = p_1p_2p_3...p_m$ mit der Länge $|P| = m$ in einem gegebenem Text $T = t_1t_2t_3...t_n$ der Länge $|T| = n$ auf dem Alphabet $\Sigma$. Dabei verwenden wir ein Suchfenster $D$ der Breite $|D| = |P|$ des Strings, das sich den Text von Links entlang arbeitet.

Text und String werden innerhalb dieses Fensters verglichen indem man drei Ansätze verfolgt. Sind $x, y, z$ Zeichenketten so bezeichnet man $x$ Präfix von $xy$, Suffix von $yx$ und Faktor von $yxz$. In diesem Kapitel werden diese begriffe ausschließlich auf Zeichenketten der länge $\leq 1$ angewendet, da der leere String immer Teil jeder Zeichenkette ist. Die *Präfixsuche* sucht vorwärts innerhalb des Suchfensters nach dem längstes Präfix des Fensters, das auch Präfix Suchstrings ist. Beispiele: Brute-Force, Shift-And, Shift-Or. Die *Faktorsuche* durchläuft das Suchfenster rückwärts und sucht nach dem längsten Suffix, welches auch ein Faktor des Strings ist. Beispiel: BNDM. Und der dritte Ansatz ist die *Suffixsuche*. Das Suchfenster wird rückwärts, auf der Suche nach dem längsten Suffix des Suchfensters, das auch Suffix des Suchstrings ist, durchlaufen. Dieser Ansatz wird hier nicht weiter verfolgt.

## 8.4 Brute-Force Algorithmus

Durch seine schlichte Arbeitsweise und schnelle Implementierung eignet sich der Brute-Force Algorithmus besonders gut als Referenzalgorithmus. Es handelt sich dabei um einen naiven Ansatz einer präfixbasierenden Suche ohne Bit-Parallelität.

Die Suche erfolgt in zwei geschachtelten while-Schleifen. Die äußere Schleife sorgt dafür, dass nach jedem Durchgang der Inneren das Suchfenster um eine Stelle weiter geschoben wird. In der inneren Schleife werden die Zeichen vorwärts im Suchfensters mit dem Zeichen des

Suchstrings verglichen, bis der String gefunden und sein Vorkommen gemeldet wird oder die Zeichen nicht übereinstimmen und die innere Schleife abgebrochen wird.

Die Nachteile des Algorithmus sind schnell ersichtlich: Es wird jedes Zeichen des Textes $T$ mindesten ein und maximal $|P|$ Mal verglichen. Daraus ergibt sich eine worst-case Laufzeit von O( $|P|\cdot|T|$ ). Das lässt sich gut nachvollziehen indem man den Algorithmus auf eine worst-case Eingabe anwendet: Suche in einer DNA-Sequenz $T =$ AAAAAT nach dem Muster $P =$ AAT. Bei einer unabhängigen identischen Verteilung aller Zeichen des Alphabets $\Sigma$ im Text und einer Antreffwahrscheinlichkeit von $|\Sigma|^{-1}$ gelangt man zu einer average-case Laufzeit von O( $|T| \cdot (1 + (|\Sigma| - 1)^{-1})$ ), was allerdings schon nahe an der angestrebten linearen Laufzeit liegt.

## 8.5 Shift-And/Or Algorithmus

Den größten Nachteil des Brute-Force Algorithmus, nämlich die wiederholten Durchläufe der Textzeichen, umgeht der Shift-And Algorithmus, indem er sich zur Laufzeit alle gelesenen Präfixe des Strings im Suchfenster merkt. Da es genügt die Anfangsposition jedes Präfixes und damit auch jedes möglichen Vorkommen des Strings zu kennen, führen wir eine Bitmaske $D = d_m, \ldots, d_1$ für das Suchfenster ein. Die Maske stellt für jedes Zeichen aus dem Suchfenster ein Bit zur Verfügung, das im aktiven Zustand ein Präfix markiert. Beim Shift-And Algorithmus nennen wir ein 1-gesetztes Bit aktiv. Da die Länge der Bitmaske des Fensters der Stringlänge entspricht $|D| = |P|$, bedeutet die Aktivierung des $m$-ten Bits der Maske $D$ ein Vorkommen des Strings an der Position des Fensters. Denn das Präfix vom String, dessen Länge der Stringlänge $|P|$ entspricht, ist der String selbst.

Die Umsetzung des präfixbasierenden Shift-And Algorithmus geschieht in zwei Phasen:

- Preprocessing: Zunächst wird eine Bitmaskentafel $B = \Sigma \times w$ erstellt, in welcher für jedes Zeichen aus dem Alphabet $\Sigma$ ein Speicherbereich der Größe eines Wortes $|w|$ zugewiesen und mit 0 initialisiert wird. Anschließend wird der String vorwärts durchlaufen und für jedes Zeichen $c$ an der Position $j$ ( $c = p_j$ ), das $j$-te Bit der Bitmaske des Zeichens in der Tafel aktiviert $B_j[\ c\ ] = 1$.
- Searching: Für die Suche wird eine mit 0 initialisierte Fenstermaske $D$ verwendet. Für jedes neu gelesene Zeichen $c$ aus dem Text $T$ an der Stelle $j$ ( $T[\ j\ ] = c$ ) wird auf $D$ ein Links-Shift ausgeführt und das erste Bit aktiviert. Anschließend wird die Fenstermaske mit der Bitmaske für das Zeichen $c$ aus der Tafel ver-UND-et. Ist das $m$-te Bit der Fenstermaske aktiv, wird das Vorkommen des Strings an der Position $j - |P| + 1$ gemeldet.

Bildhaft kann man sich den Algorithmus folgendermaßen vorstellen: Wir erstellen uns einen Satz von Filtern - das ist unsere Bitmaskentafel - und ordnen die Filter den Textzeichen entsprechend an. Anschließend feuern wir Einsen diagonal durch diese Filterreihe durch (Abbildung 8.1 (1)). Dabei kann eine Eins nur dann alle Filter passieren, wenn diese in der richtigen Reihenfolge angeordnet sind - entspricht der Reihenfolge der Stringzeichen. Das ist nur dann der Fall, wenn wir auf den gesuchten String stoßen.

Realen Rechner steht bisher jedoch nur eine begrenzte Parallelität zur verfügung, sodass nicht alle diese Einsen simultan abgefeuert werden können. Deshalb begnügt man sich mit

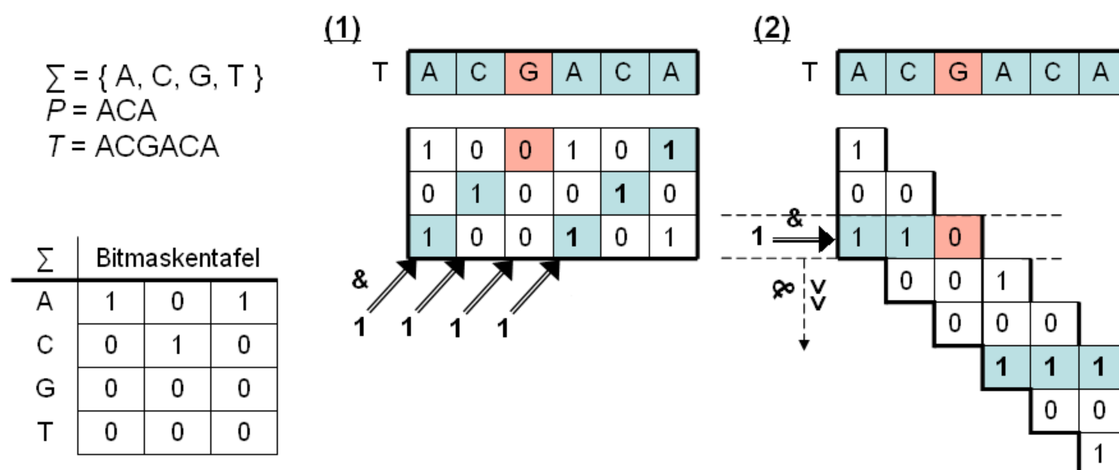



Abbildung 8.1: Anschauliche Darstellung des Shift-And Algorithmus

einer Äbschussvorrichtungünd einer Filterreihe der Länge des Fensters (Abbildung 8.1 (2)). Simuliert wird das abfeuern mittels des Shift-Operators, wobei mit 1 aufgefüllt wird ( UND 1 ), und indem man das bitweise UND auf die Bitmasken anwendet, wird das Filtern realisiert.

Listing 8.1: Java-Implementierung des Shift-And Algorithmus für Stringlänge $|P| \leq 32$ (int)

```java
public void Shift_And_Search( char[] Text, char[] Pattern,
                              int TextLength, int PatternLength )
{
    //------------------ INIT --------------------
    int Pos, Window = 0;                     // Initialisierungen:j, D
    // B fuer alle ASCII Zeichen reservieren
    int BitTable[] = new int[256];
    int Expected = 1 << PatternLength -1;    // Vergleichsbitmaske
    // BitTable mit 0x00000000 fuellen etfaellt in Java

    //------------- PREPROCESSING ---------------
    // Bitmasken-Tafel anhand des Suchstrings errechnen
    for ( Pos = 0; Pos < PatternLength; Pos++ )
        BitTable[ Pattern[Pos] ] |= 1 << Pos;

    //------------------ SEARCHING ---------------
    Pos = 0;                                 // Textposition setzen
    for ( Pos = 0; Pos < TextLength; Pos ++ )
    {
        // Fenstermaske fuer Textposition Pos berechnen
        Window = ( (Window << 1) | 1 ) & BitTable[ Text[Pos] ];
        // erstes Bit der Fenstermaske gesetzt  ?
        if (( Window & Expected ) == Expected )
            // Textposition des gefundenen Strings ausgeben
            System.out.println( Pos - PatternLength +2 );
    }
}
```

Geht man davon aus, dass man für einen String der Länge $|P|$ $k = \left\lceil \frac{|P|}{|w|} \right\rceil$ nacheinander geschaltete Worte benötigt, dann beträgt die Laufzeit des Preprocessing O( $k \cdot |\Sigma| + |P|$ ). Für jedes Zeichen aus dem Alphabet müssen $k$ Worte reserviert und initialisiert werden, um dann für jedes Zeichen des Strings je ein Bit mittels Shift und AND zu aktivieren.
Für das Searching wird jedes Zeichen genau ein Mal durchlaufen, für welchen dann ein Shift auf $k$ Wörtern ausgeführt werden muss.
Das führt dann zu einer Laufzeit von O( $k \cdot |T|$ ) für das Searching und einer Gesamtlaufzeit von O( $|P| + k \cdot (|\Sigma| + |T|)$ ) = O( $|P| + \left\lceil \frac{|P|}{|w|} \right\rceil \cdot (|\Sigma| + |T|)$ ), gültig für den worst-case, average-case und best-case. Im Vergleich zum Brute-Force Algorithmus erlangen wir eine um den Faktor $\frac{1}{w}$ verkürzte Laufzeit auf kosten eines Preprocessing und um $k \cdot |w| \cdot ( |\Sigma| + 1 )$ höherem Speicheraufwand.

Der Shift-Or Algorithmus arbeitet analog zum Shift-And Algorithmus mit dem Unterschied, dass invertierte Bitmasken verwendet werden. Die Konsequenz ist das die 0-gesetzten Bits als aktiv angesehen werden. Anschaulich schießen wir jetzt Nullen durch die Filterreihe und auch nur von den aktiven Bits durchgelassen werden.
Realisiert wird es durch die Shift- und ODER-Operationen, wobei das auffüllen mit aktivem Bit aufgrund der Arbeitsweise des << Operators entfällt. Dadurch spart man sich eine Operation während der Suche ( für ein aufwendigeres Preprocessing in Java ).

Eine schöne Eigenschaft des Shift-And/-Or Algorithmus ist, dass er durch die Bit-parallele Arbeitsweise einen nichtdeterministischen endlichen Automaten (NEA) simulieren kann (Abbildung 8.2). Als Zustände nimmt man die Bits der Suchfensterbitmaske und als Zustandsübergänge die Operationen Shift und AND/OR. Was logischerweise auffällt, das ist dass die Zustandsmenge $Z = z_m, ..., z_0$ des NEAs größer ist, als die Länge des Suchfensters $D = d_m, ..., d_1$. Da der Startzustand aufgrund der Sigma-Übergangsbedingung immer aktiv ist, spart man sich ein Bit und schiebt die Konstante 1 (bzw. eine 0 beim Shift-Or Algorithmus) ein.

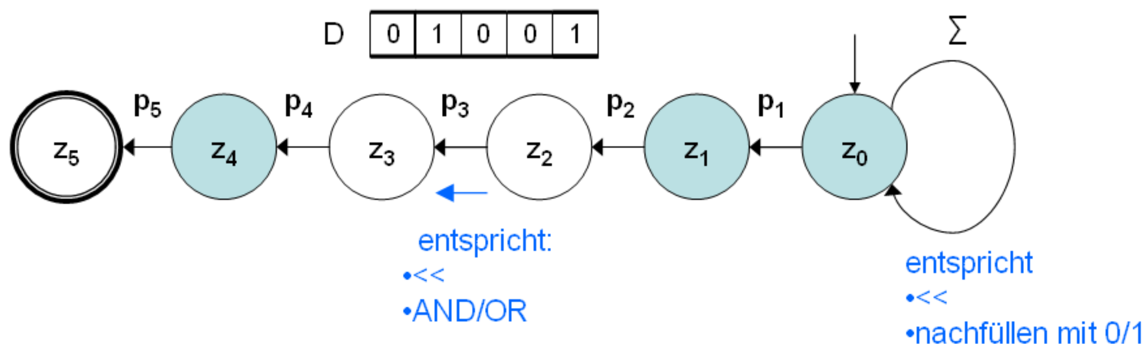



Abbildung 8.2: Nichdeterninistischer Automat, der den String $P = p_1p_2p_3p_4p_5$ akzeptiert

Ist nach einem Schleifendurchlauf das $j$-te Bit der Maske aktiv, so ist auf den NEA übertragen auch der $j$-te Zustand aktiv. Gegeben durch die Transitionsbedingung ist der Startzustand immer aktiv. Weil mehrere Bits gleichzeitig gesetzt sein können und Bit-parallele Operationen auf ganzen Wörtern arbeiten, ist eine korrekte Simulation des nichtdeterministischen Automaten gewährleistet.

## 8.6 Backward Nondeterministic Dawg Matching

Was beim Shift-And Algorithmus auffällt ist, dass seine Laufzeit nur von der Eingabelänge abhängt und nicht von der Eingabe selbst. Auf Sortieralgorithmen übertragen, würde man sagen, dass der Shift-And Algorithmus nicht adaptiv sei. Wenn zum Beispiel das Stringalphabet wesentlich kleiner ist als das Textalphabet und dadurch das Stringpräfixvorkommen sehr gering ist, durchläuft die Shift-And Suche trotzdem ßturSZeichen nach Zeichen den ganzen Text.
Der Backward Nondeterministic Dawg Matching Algorithmus ist da wesentlich flexibler. Er geht davon aus, dass, wenn der Text mit Sprüngen der Länge des gesuchten Strings/Suchfensters $|P| = |D|$ durchläuft, man kein Vorkommen des Strings $P$ im Text $T$ verpassen kann. Bei jedem Sprung kann einer der folgenden drei Fälle auftreten (Abbildung 8.3):

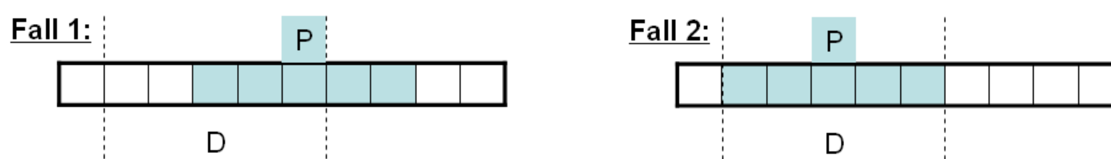



Abbildung 8.3: Mögliche Positionen eines Strings im Suchfenster

1. Das Fenster trifft auf den gesuchten String im Text, jedoch liegt nur sein(e) Präfix(e) im Suchfenster als Suffix(e) des Fensters vor. Dieser Fall wird durch die Faktorsuche des BNDM abgedeckt. Die relative Position des längsten Präfixes im Fenster wird in der zusätzlichen Variablen *Last* gespeichert und als Ziel für den nächsten Sprungs benutzt.
2. Der String liegt genau im Suchfenster. Das ist allerdings nur ein Spezialfall von (1.), denn der String selbst ist auch ein Präfix des Strings. Also deckt die Faktorsuche auch diesen Fall ab und es wird wie in (1.) verfahren. Zusätzlich wird ein Vorkommen an der Position des Fensters gemeldet und in *Last* nur das zweitlängste Präfix gespeichert.
3. Es liegt kein Stringpräfix der Länge grösser Null im Suchfenster. Es wird ein weiterer Sprung um eine Fensterlänge ausgeführt.

Die Umsetzung findet wieder in zwei Phasen statt:

- Preprocessing: Es wird wie beim Shift-And Algorithmus eine Bitmaskentafel $B = \Sigma \times w$ erstellt und mit 0 initialisiert. Dann lesen wir den String $P$ vorwärts und aktivieren für jedes Zeichen $c$ an der Position $j$ $( c = p_j )$ das $(m-j)$-te Bit der Maske $B_{m-j}[ c ] = 1$ für das Zeichen $c$ in der Tafel. Im Vergleich zum Shift-And Preprocessing erhalten wir jetzt eine gespiegelte Bitmaskentafel.
- Searching: Die Suche besteht aus zwei geschachtelten Schleifen.
  Bei jedem Durchlauf der äußeren Schleife wird die Bitmaske des Suchfensters $D$ mit $1^{|w|}$ initialisiert. Die relative Leseposition im Fenster und die Variable *Last* werden ans Ende des Fensters gesetzt. Nach dem Aufruf der inneren Schleife wird das Fenster um *Last* Zeichen vorwärts geschoben, falls der übrige Text ab *Last* länger als $|P|$ ist, ansonsten endet der Algorithmus.
  Die innere Schleife wird ausgeführt, bis die Fensterbitmaske $D = d_1, ..., d_m = 0^{|w|}$ ist. Darin wird das Suchfenster rückwärts durchlaufen und zunächst die Bitmasken

des Suchfensters und des gelesenen Zeichens ver-UND-et. Ist danach das erste Bit des Fensters aktiv, so wurde ein Präfix gelesen und es ist anhand der Leseposition im Fenster zu prüfen ob dieses die Länge des String hat. Trifft es zu (Fall 2.), dann wird ein Vorkommen des Strings gemeldet, sonst wird *Last* auf die aktuelle Leseposition gesetzt - das erste Zeichen des Präfixes (Fall 1.). Anschließend wird die Bitmaske des Fensters und die Leseposition um eine Stelle nach Links ge-Shift-et bzw. verschoben.

Leider stellen sich die reversen Masken der Bittafel und Fenster nachteilig für die Implementierung heraus, da bei Stringlänge kleiner Wortlänge das erste Bit des Fensters mitten im Wort liegt und beim nächsten Shift nicht herausgeschoben wird. Im C++ - Beispiel (siehe Listing 8.2) habe ich deshalb auf die Spiegelung verzichtet und in die umgekehrte Richtung geschoben.

Listing 8.2: C++ - Implementierung des BNDM Algorithmus für Stringlänge $|P| \leq 32$

```
void BNDM_Suche( char Pattern[], char Text[],
                 unsigned int PatternLength, unsigned int TextLength )
{   // --------------------- INIT -----------------------
    // Indextransformation 1..m -> 0..m-1 , 1..n ->  0..n-1
    PatternLength--; TextLength--;
    unsigned int Pos, BitTable[256],Window, Last, WindowPos;
    // ------------------ PREPROCESSING -------------------
    // BitTable-Eintraege 0 setzen
    memset( BitTable, 0, sizeof( BitTable ));
    // Masken fuer Textzeichen in der Tafel speichern
    for( Pos = 0; Pos <= PatternLength; Pos++ )
        BitTable[ Pattern[ Pos ]] |= 1 << Pos;
    //--------------------- SEARCHING  -------------------
    Pos = 0;                        // Textposition
    while ( Pos < ( TextLength - PatternLength ))
    {
        WindowPos = Last = PatternLength;
        // Fenstermaske mit 0xFFFFFFFF initialisieren
        Window = -1;
        while ( Window != 0 )
        {
            Window &= BitTable[ Text[ Pos + WindowPos ]];
            if ( Window & 1 )       // Faktor ist Praefix?
                // Factorlaenge < Stringlaenge
                if ( WindowPos > 0 ) Last = WindowPos;
                // Factorlaenge ==  Stringlaenge
                else printf( "%i\n", Pos + 1 );
                Window >>= 1; WindowPos--;
        }
        Pos += Last;                // Suchfenster verschieben
    }
}
```

Der Backward Nondeterministic Dawg Matching Algorithmus bring im worst-case keine Leistungssteigerung. Ein Beispiel einer worst-case Eingabe für den BNDM Algorithmus: DNA-Suche in der Sequenz $T$=TTTTTC nach dem String $P$=TTC. Führt man die Suche darauf aus, erhält man eine Laufzeit der Suche von O( $|T| \cdot |P|$ ), da Textzeichen bis zu $|P|$ mal durchlaufen werden können. Jedoch können auch Sequenzen der mit Suchstringlänge $|P-1|$

übersprungen werden. So ergibt sich für die average-case Laufzeit der Suche $O( \frac{|T| \cdot \log_{|\Sigma|} |P|}{|P|} )$. Die Laufzeit des Preprocessing bleibt bei $O( k \cdot |\Sigma| + |P| )$ und der Speicheraufwand bei $k \cdot |w| \cdot ( |\Sigma| + 1 )$ mit $k = \lceil \frac{|P|}{|w|} \rceil$ .

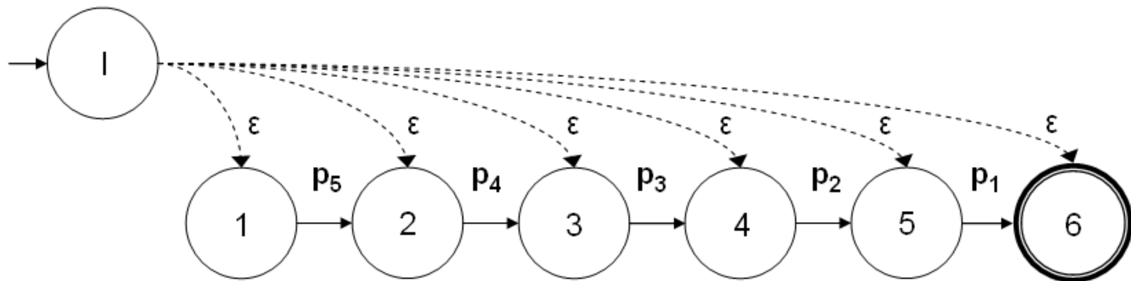



Abbildung 8.4: Ein entsprechender nichtdeterministischer endliche Automat, welcher alle reversen Präfixe eines Strings $P = p_1 p_2 p_3 p_4 p_5$ akzeptiert.

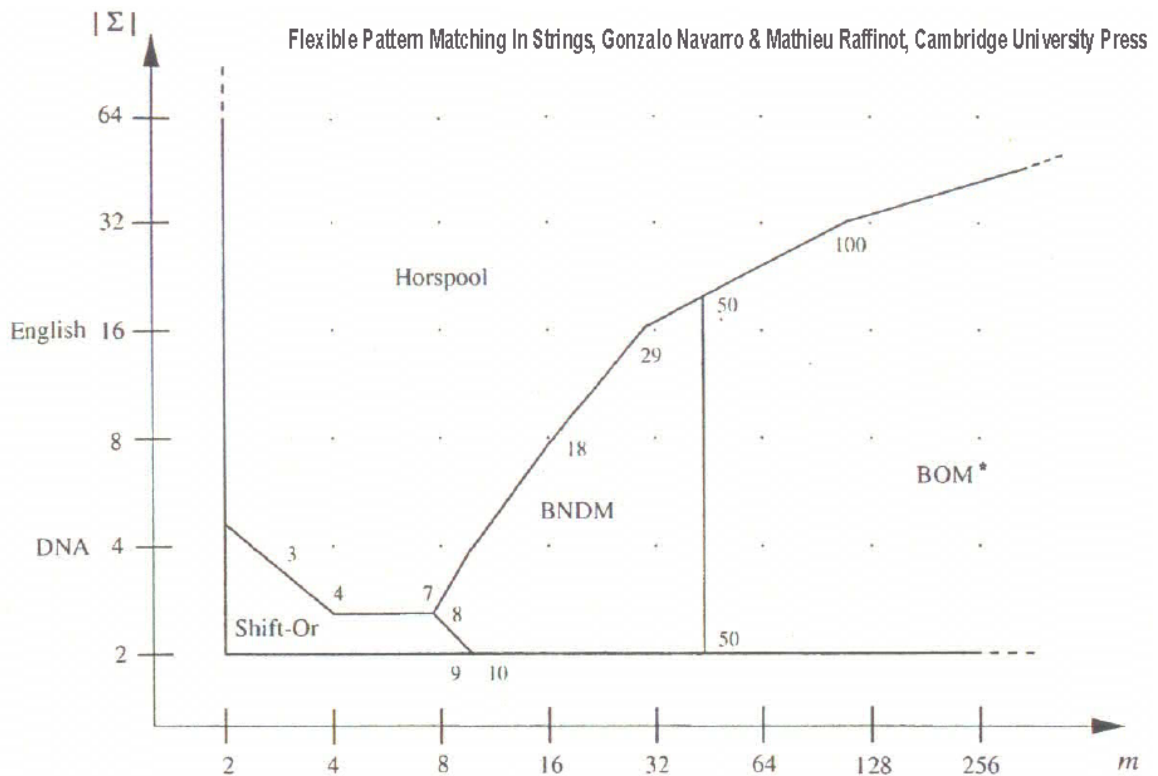



Abbildung 8.5: Experimentelle Effizienzkarte - * Backward Oracle Matching

Die Abbildung 8.5 zeigt eine experimentell ermittelte Effizienzkarte der Anwendungsbereiche einiger Pattern matching Algorithmen. Testsystem war ein Ultra Sparc 1 mit Wortlänge $|w| = 32$ Bit und einem SunOS System, angewendet auf 10 MB zufällig erstelltem Text und zufälligem String als Eingabe. Auf der horizontalen Achse ist die Stringlänge und auf der Vertikalen die Größe des Alphabets aufgetragen.

## 8.7 Quellen


- Flexible Pattern Matching In Strings, Gonzalo Navarro & Mathieu Raffinot, Cambridge University Press
- Vorlesungsfolien ( SS08 ) Einfache Textsuche, Prof. Dr. Sven Rahmann, TU-Dortmund `http://ls11-www.cs.tu-dortmund.de/people/rahmann/teaching/ss2008/AlgorithmenAufSequenzen/01-einfache-textsuche.pdf`

KAPITEL 9

# A Regular Expression Matcher

**Ausarbeitung von Mark Brockmann**

## 9.1 Einleitung

In diesem Kapitel wird ein Regular Expression Matcher Algorithmus vorgestellt, der schlank ist, aber trotzdem noch nützlich und dabei die grundlegenden Ideen veranschaulicht. Reguläre Audrücke wurden in den 1950er Jahren von Stephen Kleene als Notation für endliche Automaten erfunden. Erstmals in einem Programm umgesetzt, hatte es Ken Thompson in den 1960ern für den *QED* Text Editor. 1967 stellte er dafür einen Patentantrag, der 1971 genehmigt wurde und damit eines der ersten Softwarepatente überhaupt wurde. Für den Unix Editor *ed* und das Tool *grep* wurde der Code etwas schlanker, allerdings auch langsamer, da ein Backtracking hinzugefügt wurde. Rob Pike programmierte auf diesen Grundlagen 1998 einen C-Code mit gerade mal 30 Zeilen, der im folgenden vorgestellt wird.

## 9.2 Einfürung - Definition

Ragular Expression Matcher suchen wie Pattern Matcher eine Zeichenfolge (Pattern) in einer anderen Zeichenfolge (String). Dabei werden neben den direkt zu vergleichenden Zeichen zusätzliche Metacharacter verwendet, die z. B. als Quantoren verwendet werden. In dem Abschnitt Syntax werden die wichtigsten Metazeichen aufgelistet. Der hier vorgestellte Algorithmus wird sich nur dem Entscheidungsproblem eines regulären Ausdruckes annehmen und somit weder Position noch Länge des Fundes mit ausgeben.

## 9.3 Syntax für reguläre Ausdrücke

Das Finden eines regulären Ausdruckes hängt maßgeblich von der richtigen Benutzung der Metazeichen ab. Da es verschiedene Implementierungen gibt und keine einheitlichen Regeln nach denen sich alle richten müssen, können Metazeichen verschieden gedeutet werden. Ebenfalls vorsichtig muss man sein, wenn ein gesuchtes Zeichen etwa einem Metacharacter entspricht. Gibt es keine weiteren Zeichen für einen solchen Fall, so kann es auch vorkommen, dass man nach einigen Zeichen erst gar nicht suchen kann, weil sie immer als Metazeichen interpretiert werden.
Metazeichen lassen sich in verschiedene Gruppen einteilen. Die wichtigste hierbei ist die Zeichenauswahl bzw. die Zeichenklassen, hiermit kann ein Zeichen des Strings direkt mit einer ganzen Gruppe von Zeichen verglichen werden. Will man eine Häufigkeit eines Zeichens überprüfen oder ist einem diese Häufigkeit egal, so kann man Quantoren einsetzen. Weiter gibt es noch spezielle Zeichen, die sich auf den Anfang oder das Ende eines Strings beziehen. Die hier vorgestellten Metazeichen sind die allgemein gebräuchlichsten und beziehen sich nicht alle auf den hier vorgestellten Quellcode.

**Zeichenauswahl/-klassen** Bei der Zeichenauswahl gibt es ein rudimentäres Metazeichen, dass in der Regel für alle Zeichen stehen kann. Meistens wird hierfür der '.' verwendet, der neben alphanumerischen Zeichen auch jegliche Art von Sonderzeichen akzeptiert. Desweiteren gibt es häufig vorgegebene Zeichenklassen, z. B. für Ziffern/Zahlen und/oder Buchstaben. Darüber hinaus bieten auch viele Programme die Möglichkeit eigene Klassen zu definieren. Im einfachsten Fall reichen Eingaben wie "[a-z]" für alle Kleinbuchstaben, aber auch Auflistungen sind möglich. Eine Übersicht und detailiert Auflistung ist in den Tabellen 9.1 bis 9.3 zu finden, diese zeigt die meist verwendeten Varianten. Zeichenklassen müssen an genau der Stelle stehen wo ein Zeichen dieser Gruppe vorkommen soll.

<table>
<tr><td>[:alnum:]</td><td>Alphanumerische Zeichen: [:alpha:] oder [:digit:].</td></tr>
<tr><td>[:alpha:]</td><td>Buchstaben: [:lower:] oder [:upper:].</td></tr>
<tr><td>[:blank:]</td><td>Leerzeichen oder Tabulator.</td></tr>
<tr><td>[:cntrl:]</td><td>Steuerzeichen. Im ASCII sind das die Zeichen 00 bis 1F, und 7F (DEL).</td></tr>
<tr><td>[:digit:]</td><td>die Ziffern 0 bis 9.</td></tr>
<tr><td>[:graph:]</td><td>Graphische Zeichen: [:alnum:] oder [:punct:].</td></tr>
<tr><td>[:lower:]</td><td>Kleinbuchstaben: nicht notwendigerweise nur von a bis z.</td></tr>
<tr><td>[:print:]</td><td>Druckbare Zeichen: [:alnum:], [:punct:] und Leerzeichen.</td></tr>
<tr><td>[:punct:]</td><td>Zeichen wie: ! ”# $ % & ’ ( ) * + , - . / : ; < = > ? @ [ \ ] ˆ _ ‘ { | } ˜ .</td></tr>
<tr><td>[:space:]</td><td>Whitespace: Horizontaler und vertikaler Tabulator, Zeilen- und Seitenvorschub, Wagenrücklauf und Leerzeichen.</td></tr>
<tr><td>[:upper:]</td><td>Großbuchstaben: nicht notwendigerweise nur von A bis Z.</td></tr>
<tr><td>[:xdigit:]</td><td>Hexadezimale Ziffern: 0 bis 9, A bis F, a bis f.</td></tr>
</table>

Tabelle 9.1: Zeichenklassen (Quelle: http://de.wikipedia.org/wiki/Regulärer_ Ausdruck)

| \d | eine Ziffer [0-9] |
|---|---|
| \D | ein Zeichen, das keine Ziffer ist, also [^\d] |
| \w | ein Buchstabe, eine Ziffer oder der Unterstrich, also [a-zA-Z_ 0-9]<br>(und evtl. weitere Buchstaben, z. B. Umlaute) |
| \W | ein Zeichen, das weder Buchstabe noch Zahl noch Unterstrich ist, also [^\w] |
| \s | Whitespace; meistens die Klasse der Steuerzeichen \f, \n, \r, \t und \v |
| \S | ein Zeichen, das kein Whitespace ist [^\s] |

Tabelle 9.2: Vordefinierte Zeichenklassen (Quelle: http://de.wikipedia.org/wiki/Regulärer_ Ausdruck)

| [egh] | eines der Zeichen 'e', 'g' oder 'h' |
|---|---|
| [0-6] | eine Ziffer von '0' bis '6' (Bindestriche sind Indikator für einen Bereich) |
| [A-Za-z0-9] | ein beliebiger lateinischer Buchstabe oder eine beliebige Ziffer |
| [^a] | ein beliebiges Zeichen außer 'a'<br>('^' am Anfang einer Zeichenklasse negiert selbige) |
| [-A-Z], [A-Z-]<br>bzw. [A-Z\-a-z] | Auswahl enthält auch das Zeichen '-', wenn es das erste oder das letzte Zeichen einer Zeichenklasse ist bzw. wenn seine Metafunktion innerhalb einer Auswahl durch ein vorangestelltes '\' -Zeichen aufgehoben wird |

Tabelle 9.3: Zeichenauswahl (Quelle: http://de.wikipedia.org/wiki/Regulärer_ Ausdruck)

**Quantoren** Um eine Anhäufung von gleichen Zeichen im regulären Ausdruck zu vermeiden bzw. ein mögliches, aber nicht zwanghaftes Vorkommen eines Zeichen anzugeben, werden Quantoren genutzt. Diese reichen von Optionalität über beliebige Häufigkeit bis hin zur fest angegebenen Anzahl. Dabei steht der Quantor immer hinter dem referenzierten Zeichen. Quantoren können durch entsprechende Eingaben mit Hilfe anderer Quantoren in gewisser Weise 'simuliert' werden. Die Auflistung in Tabelle 9.4 zeigt die bekanntesten Varianten für Quantoren. Um ein "a+" zu 'simulieren', kann der '*'-Operator verwendet werden "aa*". Dies entspricht, wie in der Tabelle 9.4 zu sehen ist, auch dem min-Operator.

Quantoren beziehen sich immer auf den voranstehenden Ausdruck

| ? | optional, er kann einmal vorkommen, muss es aber nicht,<br>d. h. der Ausdruck kommt null- oder einmal vor. (Dies entspricht {0,1}) |
|---|---|
| + | muss mindestens einmal vorkommen, darf aber auch mehrfach vorkommen.<br>(Dies entspricht {1,}) |
| * | darf beliebig oft (auch keinmal) vorkommen. (Dies entspricht {0,}) |
| {n} | muss exakt n-mal vorkommen. |
| {min,} | muss mindestens min-mal vorkommen. |
| {,max} | darf maximal max-mal vorkommen. |
| {min,max} | muss mindestens min-mal und darf maximal max-mal vorkommen. |

Tabelle 9.4: Quantoren (Quelle: http://de.wikipedia.org/wiki/Regulärer_ Ausdruck)

**Beginn und Ende** Oft soll auch nur am Anfang oder Ende eines Strings gesucht werden, hierzu gibt es entsprechende Start- und Endezeichen, die auch im Ausdruck dann vorne respektive am Ende eingegeben werden müssen. Ansonsten werden diese häufig als einfaches Zeichen anstatt als Metazeichen interpretiert. In der Tabelle 9.5 finden sich auch noch weitere wichtige Zeichen unter anderem zur Aufhebung der Bedeutung eines Metazeichen.

| | |
|---|---|
| ˆ | steht für den Zeilenanfang<br>(nicht zu verwechseln mit ˆ bei der Zeichenauswahl mittels [ und ]). |
| $ | kann je nach Kontext für das Zeilen- oder Stringende stehen, wobei noch ein \n folgen darf. Das tatsächliche Ende wird von \z gematcht. |
| \ | hebt gegebenenfalls die Metabedeutung des nächsten Zeichens auf. Beispielsweise lässt der Ausdruck (A*)+ die Zeichenketten A*, A*A*, usw. zu. Auf diese Weise lässt sich auch ein Punkt . mit \. suchen, während nach \ mit \\ gesucht wird. |
| \b | leere Zeichenkette am Wortanfang oder am Wortende |
| \B | leere Zeichenkette, die nicht den Anfang oder das Ende eines Wortes bildet |
| \< | leere Zeichenkette am Wortanfang |
| \> | leere Zeichenkette am Wortende |
| \n | ein Zeilenumbruch (im Unix-Format) |

Tabelle 9.5: Beginn und Ende eines String (Quelle: http://de.wikipedia.org/wiki/Regulärer_Ausdruck)

**Weitere Metazeichen und Kombinationen** Weitere gern benutzte Metazeichen sind logische oder mathematische Verhältnisse. Ob eine Zusammenfassung unter Zuhilfenahme von Klammern oder logische Verknüpfungen mit '&', '|' für und- und oder-Beziehungen, sind in vielen Implementierungen keine Grenzen gesetzt. Auch Kombinationen mehrerer Metacharacter ist in den meisten Fällen möglich. Zur Veranschaulichung hier zwei Besipiele:

".*" findet eine beliebige Anzahl an irgendwelchen Zeichen
"(abc)+" findet Strings in denen mindestens einmal die Zeichenkette "abc" vorkommt

## 9.4 Der Algorithmus von Rob Pike

Der Algorithmus von Rob Pike ist wie bereits erwähnt sehr schlank gehalten und nur auf die grundlegenden Metazeichen ausgelegt. Dementsprechend kommt dieser Algorithmus auch mit gerade mal drei Funktionen/Methoden aus. Allerdings sind auch nur vier Metazeichen vorgesehen, diese sind '.'-beliebiges Zeichen, 'ˆ'-Stringbeginn, '$'-Stringende und '*'-beliebig häufiges Vorkommen (auch null erlaubt).
Die nun vorgestellten Methoden beziehen sich auf Grund der Parameter auf den C-Code von Rob Pike. Die Java- und Haskell-Implementierungen benutzen andere Parameter, die jeweiligen Methoden/Funktionen füren aber die gleichen Operationen durch.

**matchhere(char*, char*)** Die wichtigste Funktion matchhere() überprüft die Suffixe von regulärem Ausdruck und String auf Gleichheit. Dabei werden genau die Character an den übergebenen Pointer-Positionen verglichen. In dieser Funktion werden die vier Fälle nacheinander abgearbeitet, bis einer dieser Fälle einen Erfolg meldet.
Hierbei spielt die Reihenfolge eine wichtige Rolle, zuerst muss der simpelste Fall überprüft werden, der leere Restausdruck. Tritt dieses ein, so sind alle vorherigen Zeichen/Ausdrücke im String bereits gefunden worden und die Suche kann mit einer Erfolgsmeldung abgebrochen werden.
Darauf folgen die Spezialfälle, vor zu ziehen ist hier ein '*' an zweiter Position des Ausdrucksuffix. In diesem Fall wird die matchstar()-Funktion bemüht und dessen Ergebnis zurückgegeben.
Ist auch dieser Fall nicht eingetreten, so wird das Ausdrucksuffix auf Länge 1 und das Endezeichen '$' getestet. Als Rückgabe wird das Überprüfungsergebnis auf leere des Stringsuffix gegeben.
Als letztes wird der nichtleere Reststring überprüft. Zeigen der Patternzeiger und der Stringzeiger auf ein gleiches Literal oder ist das führende Zeichen des Ausdrucksuffix ein '.', so ruft sich matchhere() rekursiv auf, um die um 1 verkürzten Suffixe weiter zu matchen.
Trifft keiner dieser Fälle ein, so wird mit einer 0 die erfolglose Suche signalisiert.

**match(char*, char*)** Die erste der drei Funktionen die aufgerufen wird, ist match(). Diese überprüft zuerst ob der reguläre Ausdruck mit dem Startmetazeichen ('^') beginnt und gibt in diesem Fall den Rückgabewert des matchhere()-Aufrufes zurück. Wobei der zu testene Ausdruck um das Zirkumflex ('^') verringert wird. Beginnt das Pattern nicht mit dem Zirkumflex so werden alle Suffixe des String mit matchhere() getestet. Sobald einer der Suffixe dem regulären Ausdruck entspricht, wird die Suche mit Erfolg beendet. Sind alle Suffixe des Textes überprüft und es wurde kein Treffer gefunden, so wird die 0 als Misserfolg zurückgegeben.

**matchstar(int, char*,char*)** Die Funktion matchstar() überprüft alle Stringsuffixe auf den Suffix vom regulären Ausdruck ab der Position nach dem gefundenen '*'. Hierfür wird wie bei der Funktion match() eine Schleife verwendet, die so lange läuft, bis der String keine weiteren Zeichen mehr enthält. Zusätzlich muss der neu erzeugte Suffix vom String um das quantifizierte Zeichen, dessen Position als erster Parameter übergeben wurde, verkürzt werden können. Ist dieses das Metazeichen '.', so wird immer ein um 1 verkürzter Suffix erzeugt. Auch hier gilt, sobald ein Reststring dem Suffix des regulären Ausdruckes entspricht wird die Funktion erfolgreich beendet. Sollte keiner der Stringsuffixe dem Rest des Ausdruckes entsprechen, so wird die erfolglose Suche mit der 0 zurück gegeben.

### 9.4.1 Java-Implementierung mit substring()

Eine Möglichkeit in Java den Regular Expression Matcher zu implementieren ist es, jedes mal wenn eine Zeichen überprüft wurde einen Substring zu erzeugen ohne die bereits getesteten Zeichen. Da ein substring()-Aufruf aber auch immer einen neuen String erzeugt und dies relativ lang dauert, ist diese Variante insbesondere bei längeren regulären Ausdrücken und

Strings nicht gut geeignet. Die Laufzeit ist im schlechtesten Fall um das sechs- bis siebenfache höher als bei dem C-Code von Rob Pike.

```
public static boolean match(String regexp, String text) {
     regexp = regexp + '\0';     //verhindert OutOfBoundsExceptions
     text = text + '\0';         //verhindert OutOfBoundsExceptions
     if (regexp.charAt(0) == '^')
          return matchhere(regexp.substring(1), text);
     do {
          if (matchhere(regexp, text))
               return true;
          text = text.substring(1);
     } while (text.charAt(1) != '\0');
     return false;
}

public static boolean matchhere(String regexp, String text) {
     if (regexp.charAt(0) == '\0')
          return true;
     if (regexp.charAt(1) == '*')
          return matchstar(regexp.charAt(0), regexp.substring(2), text);
     if (regexp.charAt(0) == '$' && regexp.charAt(1) == '\0')
          return text.charAt(0) == '\0';
     if (text.charAt(0) != '\0' &&
               (regexp.charAt(0) == '.' || regexp.charAt(0) == text.charAt(0)))
          return matchhere(regexp.substring(1), text.substring(1));
     return false;
}

public static boolean matchstar(char c, String regexp, String text) {
     text = '\0' + text;     //wird benoetigt um ersten substring()-Aufruf aufzuheben
     do {
          text = text.substring(1);     //vor matchhere, damit Schleifenbed. richtig arbe
          if (matchhere(regexp, text))
               return true;
     } while (text.charAt(0) != '\0' && (text.charAt(0) == c || c == '.'));
     return false;
}
```

### 9.4.2 Java-Implementierung mit charArray

Wie bereits im vorherigen Abschnitt zu sehen, werden die Character mit charAt() nochmals erst ausgelesen. Warum also nicht direkt Integer verwenden und diese nur zu erhöhen und beim charAt()-Aufruf als Parameter setzen. Selbst dann müssen aus dem String immer wieder einzelne Zeichen umgewandelt und ausgelesen werden, also wandelt man besser direkt den ganzen String in ein charArray um und hat so keine weiteren Umwandlungen mehr. Character

können direkt verglichen werden, anstatt immer wieder neuen Strings werden jetzt nur noch neue Integer erzeugt, was auch den Speicher entlastet. Im Quellcode nicht angezeigt aber notwendig ist auch eine vorherige Erweiterung der Strings um jeweils ein Endzeichen (siehe Zeilen 2 und 3 im vorherigen Code).

```
public static boolean match(int reg, int txt) {
     if (regexp[reg] == '^')
          return matchhere(reg+1, txt);
     do {
          if (matchhere(reg, txt))
               return true;
     } while (text[txt++] != '\0');
     return false;
}
```

```
public static boolean matchhere(int reg, int txt) {
     if (regexp[reg] == '\0')
          return true;
     if (regexp[reg+1] == '*')
          return matchstar(reg, reg+2, txt);
     if (regexp[reg] == '$' && regexp[reg+1] == '\0')
          return text[txt] == '\0';
     if (text[txt] != '\0' && (regexp[reg] == '.' || regexp[reg] == text[txt]))
          return matchhere(reg+1, txt+1);
     return false;
}
```

```
public static boolean matchstar(int c, int reg, int txt) {
     do {
          if (matchhere(reg, txt))
               return true;
     } while (text[txt] != '\0' && (text[txt++] == regexp[c] || regexp[c] == '.'));
     return false;
}
```

### 9.4.3 C-Implementierung mit Pointern (Rob Pike)

CharArrays für die nur der Integerwert für die auszulesende Position geändert wird, fungieren schon ähnlich wie Pointer. Aber echte Pointer gibt es in Java nicht, daher wird jetzt in die Programmiersprache C und zu dem Code von Rob Pike aus "Beautiful Code" gewechselt.

Dies macht einige Operationen schneller und einfacher umzusetzen.

```
int match(char *regexp, char *text) {
     if (regexp[0] == '^')
          return matchhere(regexp+1, text);
     do {
          if (matchhere(regexp, text))
               return 1;
     } while (*text++ != '\0');
     return 0;
}
```

```
int matchhere(char *regexp, char *text) {
     if (regexp[0] == '\0')
          return 1;
     if (regexp[1] == '*')
          return matchstar(regexp[0], regexp+2, text);
     if (regexp[0] == '$' && regexp[1] == '\0')
          return *text == '\0';
     if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
          return matchhere(regexp+1, text+1);
     return 0;
}
```

```
int matchstar(int c, char *regexp, char *text) {
     do {
          if (matchhere(regexp, text))
               return 1;
     } while ((*text++ == c || c == '.'));
     return 0;
}
```

### 9.4.4 Haskell-Implementierung

Die Autoren von "blog.thoughtfolder.org" haben Rob Pikes Code genommen und versucht ein möglichst schönes Haskell-Programm daraus zu schreiben. Dank der Vereinfachung von if-Abfragen in Haskell verkürzt sich der Quellcode. Außerdem hilft das "any" in Zeile 6 diesen

Abschnitt auch deutlich kürzer zu implementieren.

```
module BasicRegex (match, matchShell) where
import Data.List( tails )

match :: [Char] -> [Char] -> Bool
match ('^':regex) text = matchhere regex text
match regex text = any (matchhere regex) (tails text)

matchhere :: [Char] -> [Char] -> Bool
matchhere [] _ = True
matchhere (c:'*':regex) text = matchstar c regex text
matchhere regex [] = regex == ['$']
matchhere (r:regex) (t:text) = charMatch r t && matchhere regex text

matchstar :: Char -> [Char] -> [Char] -> Bool
matchstar c regex [] = matchhere regex []
matchstar c regex (t:text) =
          matchhere regex (t:text) || charMatch c t && matchstar c regex text

charMatch :: Char -> Char -> Bool
charMatch c t = c == '.' || c == t
```

(Quelle: http://blog.thoughtfolder.com/2008-02-04-even-more-beautiful-code-c-haskell-.html)

## 9.5 Erweiterungen und Verbesserungen

Nun stellt sich die Frage, ob man den Algorithmus noch verbessern kann oder mehr Metazeichen implementieren kann ohne die bisherige Eleganz zu verlieren. Dafür wird im folgenden die C-Implementierung etwas erweitert und verändert.

**Effizienz erhöhen** Mit einigen Pattern Matching Verfahren, die in den vorherigen Kapiteln vorgestellt wurden, könnte die Laufzeit des Algorithmus deutlich verbessert werden. Aber hier wird nun ein spezielles Problem des Regular Expression Matchers von Rob Pike angegangen. Die matchstar()-Funktion hat bisher eine schlechte Effizienz, wenn sich ein Literal im String häufig wiederholt und der reguläre Ausdruck diesen Bereich mit einem '*'-Quantor durchsucht. Der übrig bleibende Ausdruck wird an jeder Stelle ab der ersten möglichen Stelle, also einer Wiederholung von nullmal gesucht. Ersetzt man den non-greedy Ansatz nun durch eine gierige Implementierung, so wird zumindest für einen solchen Fall eine deutliche Effizienzsteigerung hervorgerufen. Durch diese Änderung wird statt dem kürzesten und

ersten Auftreten des regulären Ausdruckes der erste und längste gefunden.

```
int matchstar(int c, char *regexp, char *text) {
     do {
          if (matchhere(regexp, text))
               return 1;
     } while ((*text++ == c || c == '.'));                          /*entfernen*/
     return 0;
}
```

```
int matchstar(int c, char *regexp, char *text) {
     char *t                                                        /*hinzufuegen*/
     for (t = text; t != '\0' && (*t == c || c == '.'); t++);  /*hinzufuegen*/
     do {
          if (matchhere(regexp, text))
               return 1;
     } while (t-- > text);                                          /*hinzufuegen*/
     return 0;
}
```

```
matchstarG c regex [] = matchhere regex []
matchstarG c regex (t:text) =
          charMatch c t && matchstarG c regex text || matchhere regex (t:text)
```

(Quelle: http://blog.thoughtfolder.com/2008-02-04-even-more-beautiful-code-c-haskell-.html)

**Metazeichen hinzufügen** Hinzufügen von Metazeichen kann je nach Funktion relativ einfach sein, aber auch relativ schwer. Deswegen werden exemplarisch nur wenige Zeichen eingefügt, die aufzeigen wie der elegante und knappe Stil fortgeführt werden kann, aber auch wie die Eleganz durch zu viele Sonderfälle verloren geht. Wie im folgenden Code noch zu sehen ist, lässt sich der '+'-Operator noch einfach implementieren, kann aber wie bereits erwähnt noch einfacher vom Benutzer durch ein veränderte Eingabe ausdrückt werden ("a+" –> "aa*"). Die Optionalität hingegen ist schon nicht mehr so intuitiv und kurz zu implementieren, dabei

ist dieses noch eine der einfachten Erweiterungen.

```
int matchhere (char *regexp, char *text)
{
     /* Ende des regul\"aren Ausdruckes? */
     ...
     /* '*'-Quantor an 2. Position des RegEx? */
     ...
     /* '+'-Operator, erstes Zeichen testen --> einmaliges Vorkommen, danach '*' */
     if (regexp[1] == '+' && (regexp[0] == *text || regexp[0] == '.'))
          return matchstar(regexp[0], regexp+2, text+1);
     if (regexp[1] == '?') {
          /*Optionalitaet, falls bei einem Vorkommen nich, dann bei 0*/
          if ((regexp[0] == *text || regexp[0] == '.') && matchhere(regexp+2, text+1))
               return 1;
          return matchhere(regexp+2, text);
     }
     /* '\$' am Ende des regul\"aren Ausdruck? */
     ...
     /* Literale vergleichen */
     ...
}
```

```
matchhere_ (r:'+':regex) (t:text) = charMatch r t && matchstar r regex text
matchhere_ (r:'?':regex) (t:text) =
          (charMatch r t && matchhere regex text) || matchhere regex (t:text)
```

(Quelle: http://blog.thoughtfolder.com/2008-02-04-even-more-beautiful-code-c-haskell-.html)

## 9.6 Fazit

Ein Regular Expression Matcher kann nur elegant sein, wenn dieser auf wenige Metacharacter beschränkt wird. Eine große Auswahl an Sonderzeichen ist nur auf Kosten der Übersichtlichkeit und Eleganz realisierbar. Ein effizienter und brauchbarer RegEx Matcher, der zusätzlich noch kurz und elegant ist, ist daher kaum umsetzbar, dies sieht man auch daran, dass die heutzutage verwendeten Implementierungen alle mindestens mehrere hundert Zeilen lang sind. Dieses ist auch bedingt dadurch, dass die Laufzeit des Algorithmus durch ungünstige Eingaben enorm erhöht werden kann. Die Erkennung und Behebung solcher "schlechten" Eingaben würde zwar bei gut zu verwertenden Inputs weitere Laufzeit kosten, aber bei ungünstigen regulären Ausdrücken die Laufzeit um ein vielfaches verringern. Außerdem haben wir gesehen, wie unterschiedlich verschiedene Programmiersprachen mit Strings arbeiten und die Wahl der richtigen Sprache stark von der zu lösenden Aufgabe abhän! gt. Ein optimales Ergebnis ist dementsprechend nur mit geeigenten Einschränkungen und der Wahl der richtigen Programmiersprache möglich.

# KAPITEL 10

# Approximative Teilstringsuche

**Ausarbeitung von Igor Blyufshteyn**

## 10.1 Fragestellung

Wir haben folgendes Problem: eine Zeichenkette $p$ (Pattern) soll in einer Zeichenkette $t$ (Text) gesucht werden, wobei gewisse Fehler erlaubt sind. Mit anderen Worten ist die approximative Teilstringsuche eine Suche in einem Datensatz nach einem bestimmen Datum, wobei als Antworten auch hinreichend ähnliche Daten akzeptiert werden. Diese Problemstellung tritt in den vielfältigsten Anwendungsgebieten auf. An dieser Stelle sollen nur einige exemplarisch genannt werden.

Text Retrieval:
Die Korrektur von falsch geschriebenen Wörtern ist eine der wichtigsten Anwendungsgebiete für approximative Teilstringsuche. Eine große Bedeutung hat diese Art der Fehler bei der Information Retrieval (IR), also dem Finden relevanten Informationen in großen Text-Sammlungen. Auch andere Anwendungen, die sich mit der Verarbeitung von Texten beschäftigen, sind ohne approximative Teilstringsuche undenkbar, zum Beispiel sind Rechtschreibprüfung, Schnittstellen zur Spracheingabe und vieles weiteres zu nennen.

Computational Biology:
Viele in der Molekularbiologie und Genetik wichtige Probleme befassen sich mit DNS- und Proteinsequenzen. Probleme dieser Art sind zum Beispiel die Suche nach bestimmten Eigenschaften in DNS-Strängen, die Bestimmung der Unterschiedlichkeit zweier genetischer Sequenzen, das Zusammensetzen von DNS-Fragmenten zu einem DNS-Strang oder auch die Suche nach ähnlichen Proteinen.

Aufgrund von vielfältigen Anwendungsgebieten wurde der grundsätzliche Lösungsansatz mehrfach unabhängig voneinander entwickelt und auch später entstanden verschiedenste verbesserte Lösungen.

## 10.2 Definition des Problems

Gegeben seien ein Pattern $x \in \Sigma^m$, ein Text $y \in \Sigma^n$ aus dem Alphabet $\Sigma$, eine Abstandfunktion $d$ und eine Grenze $k \geq 0$. Das approximative Teilstringsucheproblem besteht aus Finden von Textsubstring $y'$ aus $y$, so dass $d(x, y') \leq k$ ist, wo $d(\cdot, \cdot)$ den Abstand angibt. Das heißt, wir bekommen eine Fehlerschranke und eine Abstandsfunktion, die Operationen wie Löschen, Einfgen und Substitution unterstützt, vorgegeben. Dabei sollen wir unter Beachtung dieser Schranke in dem Text unseren Pattern wiederfinden.

## 10.3 Abstandsfunktionen

Im diesen Abschnitt werden wir drei Abstandsfunktionen beispielhaft erläutern. Folgende Operationen werden benutzt. Wir können einen Buchstaben löschen, ersetzen und substituieren, wobei noch weitere Operationen vorliegen können. Die Operationen sind mit Kosten versehen. Es existieren verschiedene Kostenfunktionen. Wir setzen die Kosten für die Operationen konstant 1. An dieser stelle sehen wir uns eine Definition von Levenshtein-Distanz. Die Levenshtein-Distanz bezeichnet in der Informationstheorie ein Maß für den Unterschied zwischen zwei Zeichenketten bezüglich der minimalen Anzahl der Operationen Einfügen, Löschen und Ersetzen, um die eine Zeichenkette in die andere zu überführen.

### 10.3.1 Levenshtein- Abstand

Levenshtein-Abstand: Der Levenshtein-Abstand erlaubt Einfüge-, Lösch- und Substituionsoperationen.
Beispiel:
$x = INDUSTRY$
$y = INTEREST$
Levenshtein- Abstand = 6, etwa: $D \to T$, $U \to E$, $\epsilon \to R$, $\epsilon \to E$, $R \to \epsilon$, $Y \to \epsilon$, wobei zum Beispiel $\epsilon \to R$ das Einfügen von $R$ bedeutet und $R \to \epsilon$ das Löschen von $R$ bedeutet.
An dieser Stelle zeigen wir dieses Beispiel mit Hilfe eines Alignments:

$$\begin{array}{cccccccccc} I & N & D & U & - & - & S & T & R & Y \\ I & N & T & E & R & E & S & T & - & - \end{array}$$

Bemerkung: wir müssen noch beachten, dass $d(s, t)$ das Minimum der Anzahl der benötigten Operationen ist, die $s$ in $t$ Transformieren.

### 10.3.2 Episod- Abstand

Episod- Abstand: Zählt man nur die Ersetz- und Löschoperationen, erhält man den Episod-Abstand.
Beispiel:
$x = INDUSTRY$
$y = INTEREST$
Episod- Abstand = 8 ($D \to T$ ersetzen durch $D \to \epsilon$, $\epsilon \to T$, usw.)

### 10.3.3 Hammingabstand

Hamming- Abstand: Der Hamming- Abstand erlaubt nur Substitutionen.
Beispiel:
$x = INDUSTRY$
$y = INTEREST$
Hamming- Abstand = 6, wobei die Berechnung des Abstandes nur bei gleich langen Ketten funktioniert.

## 10.4 Idee für die Algorithmen

An dieser Stelle definieren wir den Abstand formal. $D(i, j)$ ist der minimaler Abstand, unter einer gegebenen Abstandfunktion $d$ zwischen $P_i = p_1...p_i$ und einem approximativen Matching, das in $ti$ endet. Also betrachten wir $T_j = t_1...t_j$.
Die Idee hinter der Lösung ist die Zerlegung des Problems in kleinere Teilinstanzen, die zu größeren Instanzen zusammengesetzt werden (Dynamische Programmierung). Für das so resultierende Problem wird für jede Stelle in $T$ der Wert des besten approximativen Matching bestimmt, das dort endet. Für die Berechnung dieses Abstands sind drei Fälle von Bedeutung:
$D(i-1, j-1)$ die Ersetzung von $t_j$ durch $p_i$.
$D(i-1, j)$ die Ersetzung von $t_j$ mit dem leeren Wort (d.h. Löschen $t_j$).
$D(i, j-1)$ die Ersetzung des leeren Worts mit $p_i$ (d.h. Einfügen von $p_j$).
Dargestellt als Alignment sieht dies folgendermaßen aus:

$$\begin{array}{cccccc} t_{1...i-1} & t_i & t_{1...i-1} & t_i & t_{1\cdots i} & - \\ p_{1...j-1} & p_j & p_{1\cdots j} & - & p_{1...j-1} & p_j \end{array}$$

Mit zusätzlicher Betrachtung der initialen Teilprobleme bei leerem $T$ und $P$ ergibt sich eine Rekursionsgleichung, die jede Probleminstanz löst ($i = 0...m, j = 0....n$):

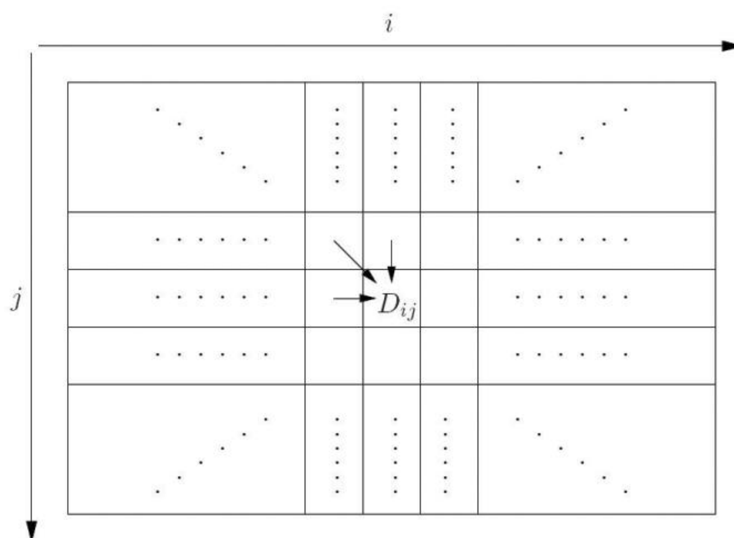
$D(i, 0) = \Sigma_{k=1}^{i} d(\epsilon, p_k)$, für $i \geq 0$

$D(0, j) = 0$, für $j \geq 0$

$$D(i,j) = \min \left\{ \begin{array}{c} D(i-1,j) + d(t_i,\epsilon) \\ D(i-1,j-1) + d(t_i,p_j) \\ D(i,j-1) + d(\epsilon,p_j) \end{array} \right\}, \text{ für } i,j > 0$$

Bemerkung: diese Rekursionsgleichung gilt nur für Levenshtein-Abstand, nicht für die Abstände in 10.3.2 und 10.3.3
Die über diese Rekursiongleichung gewonnenen Ergebnisse werden üblicherweise in der DP-Matrix dargestellt. Die unten, in Abbildung 10.1 dargestellte Matrix verdeutlicht allgemein die derartige DP- Matrix für diese Rekursionsgleichung.

Abbildung 10.1: Die allgemeine DP- Matrix




## 10.5 Algorithmus von Sellers

Algorithmus.1 zeigt den Pseudocode:

**Input**: pattern $x \in \Sigma^m$, text $y \in \Sigma^n$, threshold $k \geq 0$, cost function *cost* with indel cost $\gamma > 0$, defining an edit distance $d(\cdot, \cdot)$
**Output**: ending position $j$ such that $d(x, y') \leq k$, where $y' = y[j'...j]$ for some $j' \leq j$

1: Initialize 0−th column of the edit matix D:
2: **for** $i \leftarrow 1...m$ **do**
3: $D[i][0] \leftarrow i \cdot \gamma$
4: Procced column-by-column:
5: **for** $j \leftarrow 1...n$ **do**
6: $D[0][j] \leftarrow 0$
7: **for** $i \leftarrow 1...m$ **do**
8: $D[i][j] \leftarrow \min \{D[i-1][j-1] + cost\binom{x_i}{y_j}, D[i][j-1] + \gamma, D[i-1[j] + \gamma\}$
9: **if** $D[m][j] \leq k$ **then**
10: **report** ($j$, D[$m$][$j$]) report match ending at column j and its cost

Algorithmus 1: Der Algorithmus von Sellers in Pseudocode

Beispiel:
Es sei $x = abcde$ und $y = aceabpcqdeabcr$ gegeben. Gesucht sind alle Positionen in $y$, an denen Teilzeichenketten auftreten, deren Levenshtein-Abstand zur $x \leq 2$ ist.

Die DP- Matrix sieht dann folgendermaßen aus:

| | y | a | c | e | a | b | p | c | q | d | e | a | b | c | r |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| b | 2 | 1 | 1 | 2 | 1 | 0 | 1 | 2 | 2 | 2 | 2 | 1 | 0 | 1 | 2 |
| c | 3 | 2 | 1 | 2 | 2 | 1 | 1 | 1 | 2 | 3 | 3 | 2 | 1 | 0 | 1 |
| d | 4 | 3 | 2 | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 2 | 1 | 1 |
| e | 5 | 4 | 3 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 3 | 3 | 2 | 2 |

Die vier gefundenen Teilzeichenketten sind:
1. auf Endposition 3 die Zeichenkette *ace*
2. auf Endposition 10 die Zeichenkette *abpcqde*
3. auf Endposition 13 die Zeichenkette *abc*
4.auf Endposition 14 die Zeichenkette *abcr*.

## 10.6 Laufzeit

Im Algorithmus von Sellers geben wir nur die Kosten und die Endposition. Falls wir das ganze Alignment zurückgeben wollen, müssen die Pointer für den Backtrackingalgorithmus abgespeichert werden. Da hier dies nicht der Fall ist, brauchen wir nur die aktuelle Spalte und die vorherige, aus der wir die aktuelle berechnen, zu speichern. Daher ist der Speicherplatz $O(m)$, wobei $m$ die Länge des Patterns ist. Der Algorithmus von Seller löst das Problem in $O(mn)$ Zeit. Dies wird aus der Tatsache klar, dass wir eine DP- Matrix Größe mindestens, $O(1)$ Zeit pro Eintrag aufbauen müssen.

## 10.7 Idee für die Verbesserung

An der DP- Matrix, die mit Algorithmus von Sellers berechnet wurde, wird ersichtlich, dass manche Einträge nicht berechnet werden müssen, da sie die vogegebene Grenze überschreiten. Um die Berechnung dieser Einträge zu vermeiden, führen wir den *letzten wesentlichen Index* $i^*$ ein. Wir definieren den Index wie folgt: der *letzte wesentliche Index* $i^*$ für eine vorgegebene Grenze $k$ von einem Vektor $d \in \mathbb{N}_0^m$ ist $i^*(d) = \max\{i : d_i \leq k\}$. Da wir die DP- Matrix Spalte für Spalte berechnen, ist das letzte wesentliche Index, der letzte Eintrag in der jeweiligen Spalte, der kleiner als die Grenze $k$ ist.

## 10.8 Beispiel Algorithmus von Ukkonen

Der verbesserte Algorithmus ist der Algorithmus von Ukkonen(Algorithmus.2). An dieser Stelle führen wir den Algorithmus an einem Beispiel vor.
Es sei $S = EEDD$, $T = DEDEEDEDD$ und $k = 1$ gegeben. Der Algorithmus von Ukknonen berechnet folgende Matrix:

| S \ T | | D | E | D | E | E | D | E | D | D |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | ① | ① | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| E | | 2 | ① | 1 | 1 | 0 | 1 | 1 | 1 | 2 |
| D | | | | ① | 2 | ① | 0 | 1 | 1 | 1 |
| D | | | | | 2 | | ① | ① | ① | ① |

Dabei ist der letzte wesentliche Index umkreist und die Matches sind in der letzten Zeile unterstrichen. Die vier gefunden Teilzeichenketten werden an dieser Stelle als Alignment dargestellt:

| I: | II: | III: | IV: |
|---|---|---|---|
| S: EEDD | S: EEDD | S: EED-D | S: EEDD |
| T: EED- | T: EEDE | T: EEDED | T: DEDD |

**Input**: pattern $x \in \Sigma^m$, text $y \in \Sigma^n$, threshold $k \geq 0$, cost function *cost* with indel cost $\gamma > 0$, defining an edit distance $d(\cdot, \cdot)$
**Output**: ending position $j$ such that $d(x, y') \leq k$, where $y' = y[j'...j]$ for some $j' \leq j$

1: Initialize 0−th column of the edit matix D:
2: **for** $i \leftarrow 1...m$ **do**
3: $\quad D[i][0] \leftarrow i \cdot \gamma$
4: $i_0^* \leftarrow \lfloor k/\gamma \rfloor$
5: Procced column-by-column:
6: **for** $j \leftarrow 1...n$ **do**
7: $\quad D[0][j] \leftarrow 0$
8: $\quad i^+ \leftarrow \min\{m, i_j^* - 1 + 1\}$
9: $\quad$ **for** $i \leftarrow 1...i^+$ **do**
10: $\quad\quad D[i][j] \leftarrow \min\left\{D[i-1][j-1] + cost\binom{x_i}{y_j}, D[i][j-1] + \gamma, D[i-1[j] + \gamma\right\}$
11: $\quad$ **if** $D[i^+][j] \leq k$ **then**
12: $\quad\quad i_j^* \leftarrow \min\{(m, i^+ + \lfloor (k - D\,[i^+]\,[j])/\gamma \rfloor)\}$
13: $\quad\quad$ **for** $i \leftarrow i^+ + 1...i_j^*$ **do**
14: $\quad\quad\quad D[i][j] \leftarrow D[i-1][j] + \gamma$
15: $\quad$ **else**
16: $\quad\quad i_j^* \leftarrow \max\{i \in [0, i^+ - 1] : D[i][j] \leq k\}$
17: $\quad$ **if** $i_j^* = m$ **then**
18: $\quad\quad$ **report** ($j$, D$[m][j]$) report match ending at column j and its cost

Algorithmus 2: Der Algorithmus von Ukkonen in Pseudocode

Bemerkung zu dem Algorithmus von Ukkonen:
Die fehlende Einträge in der Matrix sind überflüssig. Woher wissen wir aber welche Einträge in der nächsten Spalte nicht berechnet müssen? Dies funktioniert folgendermaßen: wir merken uns die Zeile, wo ein Wert $> k$ auftritt. Es sei $p - 1$ diese Zeile, dann müssen in der nächsten Spalte höchstens $p$ Werte berechnet werden.

## 10.9 Laufzeit

Der Speicherplatz ist der gleiche wie in Algorithmus von Sellers, weil hier auch nur die aktuelle und die vorherige Spalte gemerkt werden muss. Also ist der Speicherplatz $O(m)$. Die Verbesserung spiegelt sich in der Laufzeit wieder. Das Verfahren von Ukkonen löst das $k$-Differenzen Problem für zufällige Zeichenketten $x$ und $y$ in $O(kn)$ Schritten. An dieser Stelle ist die Laufzeit nicht ganz präzise formuliert, da die Laufzeit von $O(mn)$ im schlimmsten Fall ebenfalls auftreten kann, jedoch im Durchschnitt gehen wir in der Spalte nur $O(k)$ Schritte nach unten.

## 10.10 Quellen


- Sequence Analysis 1+2 von Prof. Dr. Sven Rahmann
- Wikipedia
- Approximative Zeichenkettensuche von Prof. Dr. R. Parchmann

KAPITEL 11

# Dictionarys and Hashmaps in Python

**Ausarbeitung von Tobias Steinrücken**

## 11.1 Anforderungen

Dictionarys sind ein zentraler Bestandteil der Programmiersprache Python. Selbst wenn der Benutzer selbst auf den Einsatz von Dictionarys verzichtet, kommen sie wärend der Programmausführung zum Einsatz, da die Übergabe von Funktionsparametern als Wörterbuch erfolgt, jede Klasse eine Sonderform eines solchen Wörterbuches ist, und selbst eingebaute Funktionen wie len() in einen Dictionary nachgeschlagen werden. Daher ist bei Python eine *effiziente* Realisierung dieser Wörterbücher ein wesentlicher Faktor bei der gesamtperformance der ausgefürten Anwendungen, und zwar möglichst bei allen unterstützten Operationen

- *insert*: Einen Schlüssel zum Dictionary hinzufügen
- *delete*: Einen Schlüssel aus dem Dictionary löschen
- *get*: Einen Schlüssel auslesen
- ggf. *contains*: Überprüfen, ob der Schlüssel im Dictionary existiert

wobei sich *contains* auch durch einen *get()* -Aufruf implementieren lässt.

## 11.2 Implementierung mit einfachen Datentypen

Im Folgenden nehmen wir an, dass eine Hashfunktion zur Verfügung steht, die den Inhalt eines jeden unterstützten Datentypes in konstanter Zeit auf $\mathbb{Z}$ abbilden kann und dabei eine Gleichverteilung über $\mathbb{Z}$ garantiert.

### 11.2.1 Arrays

Die einfachste Methode, ein solches Dictionary zu implementieren ist sicherlich als Array. Hierbei werden alle Elemente einfach hintereinander in einem Array abgelegt. Leider ist eine

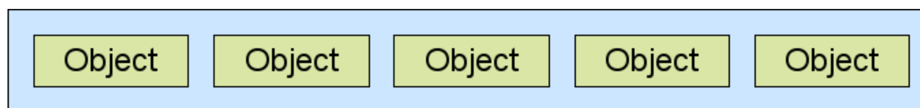



Abbildung 11.1: Implementierung als Array

solche Realisierung so langsam wie einfach:

- *insert*: O(n), da, falls ein Element am Anfang des Arrays eingefügt werden muss, alle Folgenden umkopiert werden müssen
- *delete*: O(n), da auch hier, falls das zu löschende Element am Anfang liegt, alle Folgenden aufrücken müssen
- *get*: O(1), da ein direkter Zugriff auf Indezes möglich ist. (Unter der Annahme, das keine Schlüsselkollisionen auftreten können)

Zusätzlich lassen sich in einem solchen Array nur Elemente gleichen Types ablegen, womit eine Verwendung als Grundlage für Klassen ausgeschlossen wäre, da eine Klasse zumindest aus Funktionen und elementaren Datentypen besteht.

### 11.2.2 Liste

Um die Schwächen des Arrays beim Einfügen und Löschen zu beseitigen, wäre eine alternative Idee, das Dictionary mittels einer einfach verketteten Liste zu implementieren.

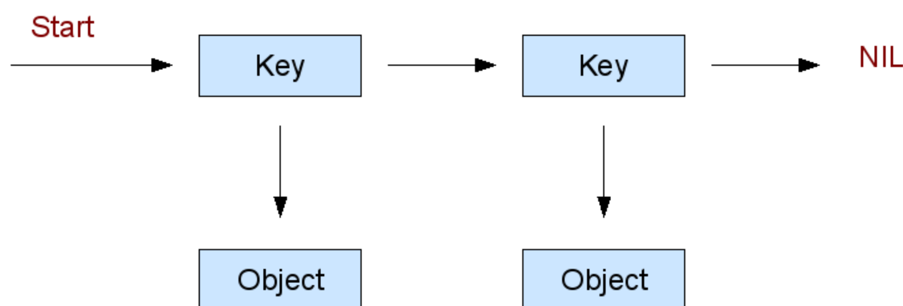



Abbildung 11.2: Implementierung als Liste

Hierbei wird unter anderem das Problem behoben, dass sich nur gleichartige Elemente in einem Dictionary ablegen lassen; die Laufzeit verbessert sich jedoch nur unwesentlich:

- *insert*: O(1), neue Einträge lassen sich einfach am Anfang der Liste einfügen
- *delete*: O(1)/O(n), abhängig davon, ob das zu löschende Element erst gesucht werden muss, oder seine Position bekannt ist
- *get*: O(n), da die Liste linear durchsucht werden muss

### 11.2.3 Bäume

Ein anderer erfolgversprechender Datentyp sind (wie-auch-immer balancierte) Bäume. Hierbei

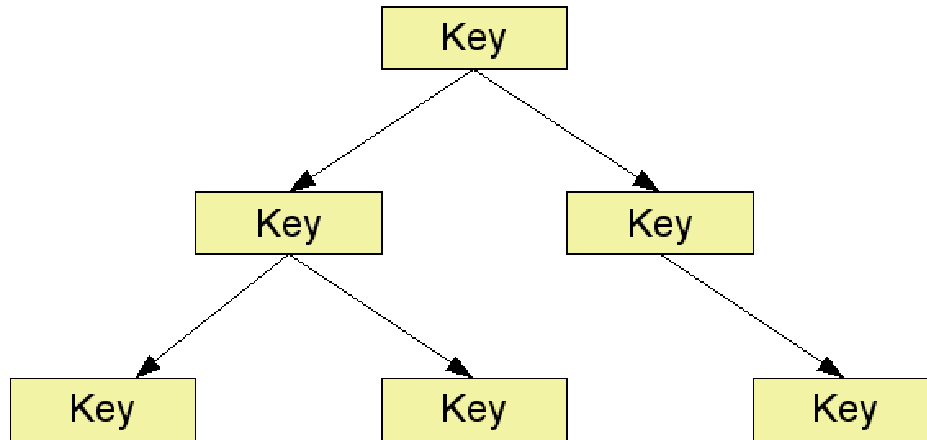



Abbildung 11.3: Implementierung als (AVL)-Baum

werden alle erforderlichen Operationen in logartitmischer Zeit unterstützt:

- *insert*: O( log(n) ). Die Suche der entsprechenden Position erfolgt in log(n), die eigentliche Einfügeoperation in O(1), und anschliessend muss der Baum ggf. durch Rotationen (jeweils O(1) auf log(n) Ebenen) rebalanciert werden
- *delete*: O( log( n )), siehe einfügen
- *get*: O( log(n) ), siehe Binäre Suche

An sich liefern Bäume schon eine sehr effiziente Methode, Dictionarys zu implementieren. Jedoch werden diese in Python so häufig verwendet, das sich selbst kleinste Optimierungen bezahlt machen und es sich lohnt, noch einmal genau hinzusehen.

## 11.3 Hashmaps

Wie bereits gesehen, sind Laufzeiten in O(1) für Teilbereiche (get, und insert/delete) bereits möglich. Wünschenswert wäre also eine Datenstruktur, die die Vorzüge der oben genannten kombiniert und alle Operationen in O(1) bereitstellt. Das Array bietet hier eine gute Grundlage, da es nur 2 „Probleme“ gibt, die sich eigentlich recht einfach lösen lassen. Zunächst werden nicht mehr die Objekte selbst im Array gespeichert, sondern nur noch Zeiger auf diese. Dies ermöglicht das Vorhalten von Elementen unterschiedlichen Typs und damit die Verwendung als Grundlage für Klassen. Die zweite wesentliche Änderung besteht darin, die Einfüge und Lösch -Operationen zu beschleunigen. Diese sind deshalb so langsam, da jeweils grosse Teile des Arrays umkopiert werden mussten, um Platz für neue Einträge zu schaffen bzw. entstandene Lücken zu schliessen. Daher ist es sinnvoll, das Array so auszulegen, das bereits genug Platz für die einzufügenden Elemente bereitsteht und dafür Lücken zuzulassen. Damit würden alle Operationen in konstanter Zeit unterstützt:

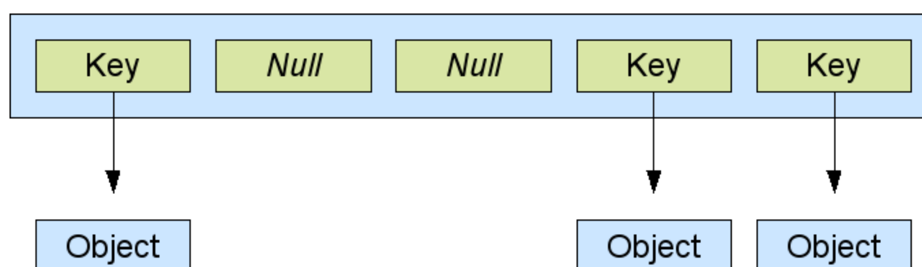



Abbildung 11.4: Implementierung als Hashmap

- *insert*: O(1), Zeiger auf neue Einträge lassen sich nun einfach einfügen ohne nachfolgende Einträge verschieben zu müssen
- *delete*: O(1), hier werden bisher belegte Felder einfach wieder als Frei markiert
- *get*: O(1), ein Zugriff auf ein Feld mittels Index ist direkt möglich

Leider ist es in der Praxis nicht ganz so einfach. Egal wie gross die Hashmap am Anfang gewählt wird - irgendwann ist sie voll und muss vergrössert werden. Ausserdem können Schlüsselkollisionen auftreten, bei denen die Hash-Funktion für 2 oder mehr Elemente den selben Slot liefert. Für Schlüsselkollisionen gibt es verschiedene Lösungsansätze. Eine Möglichkeit besteht darin, in einem solchen Fall eine Liste für diesen Slot zu führen, in der alle Elemente, die auf diesen Slot hashen, aufgeführt sind. Eine bessere Lösung ist jedoch das sog. *Open Addressing* [1]. Hierbei werden im Falle von Schlüsselkollisionen nach einem festgelegten Schema weitere Slots ausprobiert, bis ein freier Slot gefunden wird. Das einfachste dieser Schemata ist das *Linear Probing*, bei dem Linear der jeweils nächste Slot ausprobiert wird (und beim erreichen des letzten Elements zum Ersten gesprungen wird). Jedoch werden in Python öfters ganze Blöcke in ein Dictionary geschrieben (z. B. fortlaufende Nummern von 1-1000), weshalb hier eine Kollision auf Slot 2 eine extrem schlechte Laufzeit produzieren würde. Deshalb wird in Python folgender, etwas komplizierter Algorithmus verwendet. Hierbei wird zunächst

Listing 11.1: Slotprobing in Python

```
1 /* Erster Slot */
2 slot = hash;
3
4 /* Anfaenglicher  stoerfaktor  */
5 perturb = hash;
6 while( <slot ist nicht leer> and <key nicht der gesuchte Schluessel>){
7     slot = (5*slot) + 1 * perturb;
8     perturb >>= 5; // Rechtsschift um 5 Bits und auffuellen mit 0
9 }
```

der eigentliche Slot ausprobiert, und im folgenden wird in $5 * slot_nr$-Schritten durch die Hashmap gesprungen, wobei zunächst noch ein zusätzlicher Störfaktor einfliesst, der von dem Hashwert abhängt. Das Problem beim vergrössern/verkleinern der Hashmap lässt sich leider nicht so elegant lösen. Jedoch vergrössert Python die Tabelle (die Grösse ist immer ein $2^n$ mit $n \in \mathbb{N}$) um den Faktor 4 (bei mehr als 50.000 Einträgen nur noch um den Faktor 2),

[1] http://en.wikipedia.org/wiki/Open_addressing

sodass dieses nur sehr selten auftritt (bei 50.000 Einträgen nur 8-mal). Des weiteren haben Analysen von bestehenden Python Programmen gezeigt, das nur selten Einträge aus Dictionarys gelöscht werden, und wenn, dann in der Regel alle und das Dictionary wird danach verworfen. Daher wird in Python eine Hashmap *niemals* wieder verkleinert, da es effizienter ist, den Speicher auf einen Schlag frei zugeben, als sie inkrementell zu verkleinern. An diesem

Listing 11.2: Beispiel einer Hashmap

```
1  int ma_fill = 3  // Anzahl Slots, die durch Keys belegt sind
2  int ma_used = 3  // Anzahl Slots, die belegt sind oder es waren
3  int ma_mask = 7  // Groesse d. Tabelle -1 (zur schnellen Modulo Berechnung)
4
5  ma_table[] =
6      [0]:    aa    => 1  // hash(aa) = -1549758592, -1549758592 & 7 = 0
7      [1]:    cc    => 3  // hash(cc) = -1537434360,-1537434360 & 7 = 0
8      [2]: null => null
9      [3]: null => null
10     [4]: null => null
11     [5]: null => null
12     [6]:    bb    => 2  // hash(bb) = 603887302, 603887302 & 7 = 6
13     [7]: null => null
```

Beispiel einer Hashmap der Grösse 8 (dies ist die minimale Grösse mit der die Hashmaps initialisiert werden, damit bei der Übergabe von Funktionsparametern (idr. hat eine Funktion weniger als 6 Parameter) kein Overhead fürs Vergrössern der Hashmap entsteht) , in dem nacheinander die Schlüssel „aa“ , „bb“ und „cc“ eingefügt wurden, die auf die Werte 1, 2 und 3 zeigen. Beim Einfügen von „cc“ist eine Schlüsselkollision entstanden, und der Algorithmus zum wählen eines neuen Slots hat nach 2 Durchläufen Slot NR. 1 berechnet. Hierbei wird auch deutlich, dass es einen Unterschied zwischen „freien“ und „nie belegten“ Slots gibt: Bei der Suche nach einem Schlüssel muss die Suche fortgesetzt werden, wenn der betrachtete Slot einen anderen Schlüssel enthält oder als wieder „frei“ markiert ist, wärend bei einem noch nie belegten Slot abgebrochen werden kann.

## 11.4 Analyse

Bei der Analyse der durchschnittlichen Laufzeit zeigt sich, dass diese vom Belegungsgrad der Hashmap abhängt. Da bei Python diese maximal zu $\frac{2}{3}$ gefüllt ist, ergibt sich hier eine fast konstante Laufzeitfunktion, da hier im Schnitt beim Einfügen beim 3.ten Versuch ein freier Slot gefunden wird, welcher das Element aufnehmen kann, weshalb sich beim Suchen das gesuchte Element meistens auch unter den ersten 3 befindet.

- *insert*: O(≈1), da jeder 3.te Slot frei ist
- *delete*: O(≈1), siehe get
- *get*:
  - O(≈1), falls der gesuchte Schlüssel enthalten ist
  - O(≤n), falls der gesuchte Schlüssel nicht enthalten ist

Einzig das Suchen von nicht existierenden Schlüsseln bereitet hier noch Probleme, da hier, falls jeder Slot schon einmal belegt wurde, die gesamte Hashmap durchsucht werden muss. Falls jedoch auch nur ein einziger Slot noch nie belegt wurde, reduziert sich dies schon auf $\frac{n}{2}$.

## 11.5 Quellen


- *Was ist Python?*: http://sommercampus2007.informatik.uni-freiburg.de/PythonKurs
- *Beautiful Code*
- *Python QuellCode (v 2.5.2)*: http://www.python.org
- *Python_(Programmiersprache)*: http://www.Wikipedia.de

KAPITEL 13

# Mehrdimensionale Iteratoren

**Ausarbeitung von Dino Menges**

## 13.1 Definition

Iteratoren sind Zeiger, die zum Iterieren über die Elemente einer Menge genutzt werden. Der Unterschied zu einem Index besteht darin, dass Iteratoren unabhängig von der Datenstruktur sind. (**?**, Kapitel 11.1)

Mehrdimensionale Iteratoren werden genutzt, um mehrdimensionale Arrays zu durchlaufen. Mit ihnen kann man sehr leicht nur Teile dieses Arrays auslesen. Das besondere daran ist, dass der Iterator auf dem Speicher arbeitet und somit keine Kopie des Arrays erzeugt werden muss. Das spart logischerweise Speicherplatz.

Um zu sehen wie praktisch diese *elegante* Abstraktion ist, wird im Folgenden erklärt, was in $N$-dimensionalen Arrays gespeichert werden kann. In einem dreidimensionalen Array kann ein Farbbild gespeichert werden und vierdimensionale Arrays können den Druck in einem Raum während eines Konzerts speichern. Angenommen wir hätten ein Bild der Größe 656x498 Pixel, möchten daraus eine Region aus der Mitte ausschneiden und diese auf ein 160x120 Pixel großes Bild schrumpfen lassen, müssten wir dank des Iterators keinen neuen Speicherplatz für dieses neue Bild mehr bereitstellen, da es sich diesen mit dem ihm zugrunde liegenden Bild teilt.

## 13.2 Aufbau eines Iterators

### 13.2.1 Vorbemerkungen

Betrachten wir die mehrdimensionalen Iteratoren aus *NumPy* (Numerical Python). In *NumPy* ist es sehr einfach einen Ausschnitt aus einem mehrdimensionalen Array zu erzeugen. Aus der Matrix $A$ (13.1) soll die Matrix $B$ (13.2) ausgeschnitten werden. Dies geschieht in Numpy mit einem einzigen Befehl: `B=A[1:3, 1:4]`.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix} \tag{13.1}$$

$$B = \begin{pmatrix} 7 & 8 & 9 \\ 12 & 13 & 14 \end{pmatrix} \tag{13.2}$$

(Wilson and Oram, 2007a, Figure 19-1)

Was hinter diesem Befehl steckt soll jetzt genauer analysiert werden. Zunächst werfen wir einen Blick auf seine Struktur. $A$ und $B$ sind die Namen der Matrizen. In Klammern stehen die Ausschnitte der Dimensionen, welche durch Kommata voneinander getrennt werden. Um die Dimensionen zu charakterisieren, wird die Notation `start:stop:stride` verwendet. Von `start` bis `stop` wird über die Elemente der Dimension iteriert. Dabei ist zu beachten, dass `stop` die Iteration anhält und somit nicht mit ausgelesen wird. Außerdem beginnen wir bei null zu zählen und nicht bei eins. In der ersten Zeile einer Matrix hat die erste Dimension also den Index null. Analog dazu hat die zweite Dimension in der ersten Spalte der Matrix ebenfells den Index null. Der Wert `stride` gibt an, wie viele Elemente der Dimension übersprungen werden bis das nächste Element betrachtet wird. Da im obigen Beispiel die Elemente aufeinander folgen darf `stride` weggelassen werden, alternativ darf der Wert auch eins betragen. Der Befehl `B=A[1:3:1, 1:4:1]` liefert die gleiche Matrix $B$ wie oben.
Als nächstes ist es wichtig zu wissen, wie $N$-dimensionale Arrays im Speicher abgelegt werden. Das schauen wir uns wieder am Beispiel der Matrix $A$ (13.1) an. Hierbei sind die Speicherzellen, auf die auch $B$ zugreift grau hinterlegt. Wie man sieht, wird ein $N$-dimensionales Array linear im Speicher abgebildet (13.3). Es fällt weiter auf, dass der Speicher der Matrix $B$ nicht mehr zusammenhängend ist. Der Iterator muss, während er über das Array im Speicher läuft Speicherzellen überspringen, um die Matrix $B$ richtig auszulesen.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

(13.3)

(Wilson and Oram, 2007a, Figure 19-1)

### 13.2.2 Durchlaufen des Arrays

Ein zweidimensionales Array zu durchlaufen stellt kein besonders großes Problem dar. Zwei geschachtelte `for`-Schleifen sind wahrscheinlich das Erste woran jeder denkt. Um ineinander geschachtelte `for`-Schleifen zu nutzen, muss allerdings die Anzahl der Dimensionen vorher bekannt sein. Der Iterator soll aber auf $N$-dimensionalen Arrays mit unterschiedlichen Größen arbeiten können. Mit Hilfe einer Rekursion lässt sich dieses Problem umgehen (Listing 13.1). Im Beispielalgorithmus sind $a$ und $b$ Zeiger, die zwei $N$-dimensionale Arrays durchlaufen. Die Werte, die der Zeiger $a$ liest, werden an die Stelle auf die $b$ zeigt kopiert. Die Zeiger starten beide beim ersten im Speicher abgelegten Wert des jeweiligen Arrays und durchlaufen diese im Speicher. Der Wert `stride` gibt jetzt an, wie viele Bytes übersprungen werden müssen, um zum nächsten Wert zu gelangen (z.B. `stride`=4 bei einem Int-Array). Rekursive Algorithmen sind leider zu Beginn ihrer Entwicklung häufig noch sehr langsam und je weiter optimiert wird, desto unübersichtlicher und dadurch auch unverständlicher werden sie. Außerdem wird ein Funktionsaufruf benötigt. Bei einer *min*- oder *max*-Suche wird das momentane Minimum/Maximum, als Teil des Funktionsaufrufs, in spätere Rekursionen übergeben, was zu einem erhöhten Speicherbedarf führt. Jede rekursive Methode kann man in eine iterative Methode (Listing 13.2) übersetzen. In unserem Fall wird zuerst der Iterator initialisiert. Anschließend beginnt die `while`-Schleife über die Elemente des Arrays zu iterieren. Bei `process` findet die eigentliche Verarbeitung des Wertes statt, bevor der Zeiger auf den nächsten Wert gesetzt wird.

Listing 13.1: Rekursives Kopieren eines $N$-dimensionalen Arrays in Pseudocode

```
function copy_ND (a, b, N){
  if(N==0){
    copy from a to b;
    return;
  }
  set up ptr_to_a and ptr_to_b;
  for(n=0 to size of the first dimension){
    copy_ND (ptr_to_a, ptr_to_b, N-1);
    add stride_a[0] to ptr_to_a;
    add stride_b[0] to ptr_to_b;
  }
}
```

(Wilson and Oram, 2007a, Kapitel 19.1)

Listing 13.2: Iteratives Durchlaufen eines $N$-dimensionalen Arrays in Pseudocode

```
set up iterator (including curr_value=first_value);
while iterator not done:
  process curr_value;
  curr_value = next_value;
```

(Wilson and Oram, 2007a, Kapitel 19.4)

**Positionen** Wenn wir, wie bei unserem rekursiven Algorithmus, ein Array kopieren wollen muss sicher gestellt sein, dass die Arrays nicht nur dieselbe Anzahl an Elementen fassen, sondern auch die einzelnen Dimensionen gleich groß sind, ansonsten wären die Arrays nicht identisch. Die Positionen an denen die Werte im Array stehen sind also sehr wichtig. Eine Position im $N$-dimensionalen Array kann durch ein Tupel der Stellen in den jeweiligen Dimensionen eindeutig bestimmt werden. Da wir bei null anfangen zu zählen, ist der erste Wert des Arrays an Postition $(0, ..., 0)$. Jede null steht hierbei für die erste Stelle der ¡ten Dimension. Das letzte Element des $n_1$ x $n_2$ x ... x $n_N$-Array steht an Position $(n_1 - 1, ..., n_N - 1)$, wobei $n$ die Anzahl der in der Dimension enthaltenen Elemente ist. Beim Durchlaufen des Arrays wird zuerst die letzte Dimension inkrementiert (hinterste Zahl im Tupel). Wird die Position $(0, ..., n_N - 1)$ erreicht, muss die letzte Dimension zurückgesetzt und die Dimension davor um eins erhöht werden. Ist diese Dimension auch durchlaufen, steigt der Wert der Dimension vor dieser und so weiter bis die letzte Position (s.o.) erreicht ist. Wenn ein 6x5x4-Array durchlaufen wird, sähe das so aus: $(0, 0, 0)(0, 0, 1)(0, 0, 2)(0, 0, 3)(0, 1, 0)...(0, 4, 3)(1, 0, 0)...(5, 4, 3)$.

### 13.2.3 Ein erster Ansatz

Tragen wir zusammen, was wir für unseren Iterator auf jeden Fall brauchen:

**coords(i)** Merkt sich die Stelle in der **i**ten Dimension; $(0, 0, 0) \hat{=}$ `coords[]`$=\{0, 0, 0\}$.

**dims_m1(i)** Speichert die $AnzahlderElemente - 1$ der **i**ten Dimension.

**strides(i)** Speichert wie viele Bytes übersprungen werden müssen, um die **i**te Dimension zu erhöhen.

**backstrides(i)** Speichert wie viele Bytes der Zeiger zurück gehen muss, um die **i**te Dimension wieder auf null zu setzen; `coords[i]`=0.

**nd_m1** Merkt sich die Anzahl der Dimensionen; $N - 1$, da wir bei null starten.

Betrachten wir nun die Arbeitsweise des Counters. Der Code in Listing 13.3 (Wilson and Oram, 2007a, Kapitel 19.4.4) wird jedes Mal aufgerufen, wenn der Counter inkrementiert wird. Er soll die Positionen in der Reihenfolge wie oben beschrieben durchlaufen. Solange die letzte Dimension noch erhöht werden kann, ist alles ganz einfach. Schwieriger ist es, die erste Dimension zu erhöhen, weil dann alle hinter ihr liegenden Dimensionen zuückgesetzt werden.

Listing 13.3: Counter tracking in C

```
1  for (i=it->nd_m1;i>=0;i--){
       if (it->coords[i] < it->dims_m1[i]){
           it->coords[i]++;
           it->currptr += it->strides[i];
           break;
6      }
       else{
           it->coords[i]=0;
           it->currptr -= it->backstrides[i];
       }
11 }
```

**Ein komplexeres Beispiel** Bevor wir uns das komplette Iterator-Objekt anschauen, möchte ich noch ein komplexeres Beispiel einfügen. Es soll aus der Matrix $C$ (13.4) die Matrix $D$ (13.5) betrachtet werden. Durch `D=C[3::2, 2:9:3]` wird die Matrix $D$ erstellt. Hierbei fällt auf, dass auch `start` oder `stop` weggelassen werden dürfen. In diesem Fall beginnt die Matrix am Anfang der Dimension, bzw. läuft bis zur letzten Stelle der Dimension und liest diese mit aus, sofern der `stride` das zulässt. Der Iterator rechnet auch die Koordinaten der Matrizen um. Die Position $(0,0)$ in Matrix $D$ entspricht der Position $(3,2)$ in Matrix $C$. Der Speicher, auf den die Matrix $D$ zugreift hat jetzt keine zusammenhängenden Teilstücke mehr.

$$C = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 & 50 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 \\ 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 \\ 71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 \\ 81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 & 90 \\ 91 & 92 & 93 & 94 & 95 & 96 & 97 & 98 & 99 & 100 \end{pmatrix} \tag{13.4}$$

$$D = \begin{pmatrix} 33 & 36 & 39 \\ 53 & 56 & 59 \\ 73 & 76 & 79 \\ 93 & 96 & 99 \end{pmatrix} \tag{13.5}$$

### 13.2.4 Der Iterator

Kommen wir nun zum Iterator, wie er in *NumPy* implementiert wurde. Da *NumPy* in $C$ geschrieben ist, ist der Iterator natürlich auch in $C$ implementiert (Listing 13.4).

Listing 13.4: Das Iterator-Objekt in C

```c
typedef struct {
    PyObject_HEAD
    int nd_m1;
    npy_intp index, size;
    npy_intp coords[NPY_MAXDIMS];
    npy_intp dims_m1[NPY_MAXDIMS];
    npy_intp strides[NPY_MAXDIMS];
    npy_intp backstrides[NPY_MAXDIMS];
    npy_intp factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char *dataptr;
    npy_bool contiguous;
} PyArrayIterObject;
```

(Wilson and Oram, 2007a, Kapitel 19.4.5)

Hier sollen die Teile des Iterators erklärt werden, die noch nicht behandelt wurden.

**PyObject_HEAD** beinhaltet den Teil, den jedes PyObject besitzt; für uns nicht wichtig.

**index** ist eine Art counter und läuft von null bis $size - 1$.

**size** speichert die Anzahl aller Elemente, die sich im Array befinden.

**factors()** Mit Hilfe dieses Arrays ist es *schnell* möglich den eindimensionalen Index aus dem `coords[]`-Array zu berechnen; es wird allerdings nur beim Aufruf `PyArray_ITER_GOTOID` gebraucht.

**ao** ist ein Zeiger auf den Beginn des Arrays, das dem Iterator zugrunde liegt.

**dataptr** ist ein Zeiger auf das erste Byte der *curr_value*.

**contiguous** gibt an, ob der folgende Teilausschnitt des Arrays zusammenhängend ist.

Auffällig ist, dass alle Arrays in ihrer Länge durch `NPY_MAXDIMS` begrenzt sind und somit auch der Iterator nur über eine begrenzte Anzahl an Dimensionen iterieren kann. `NPY_MAXDIMS` kann man aber selbstständig verändern, so dass die Anzahl der Dimensionen (solange genügend Speicherplatz vorhanden ist) nicht begrenzt ist.

In *NumPy* stehen einige Makros zur Verfügung, wenn man einen Iterator benutzt. Hierzu betrachten wir die $Max$-Suche (Listing 13.5). Durch `it=PyArray_IterNew(ao)` wird der Iterator initialisiert, `PyArray_ITER_DATA(it)` liefert einen Zeiger auf den aktuellen Wert und durch `PyArray_ITER_NEXT(it)` wird der nächste im Array enthaltene Wert angesprochen. Dieses Codebeispiel zeigt sehr schön, dass es mit Hilfe des Iterators einfach ist einen zusammenhängenden Code zu schreiben, ohne zu wissen, ob das Array auf dem der Iterator arbeitet selbst zusammenhängend ist (vgl. Listing 13.2).

Listing 13.5: Max-Suche mit Hilfe des Iterators

```
1 #include <float.h>
  double *currval, maxval= -DBL_MAX;
3 PyObject *it;
  it=PyArray_IterNew (ao);
5 while (PyArray_ITER_NOTDONE(it)){
      currval=(double*) PyArray_ITER_DATA(it);
7     if(*currval > maxval)
          maxval = *currval;
9     PyArray_ITER_NEXT(it);
  }
```

(Wilson and Oram, 2007a, Kapitel 19.5)

### 13.2.5 Terminierung

Wir wissen jetzt wie ein Iterator aufgebaut ist und wie er über eine Menge iteriert. Doch wie arbeitet das Makro `PyArray_ITER_NOTDONE(it)`? Die einfachste Methode wäre, den letzten Wert des Counters zu speichern und diesen mit dem aktuellen Wert zu vergleichen. Leider verfolgen zusammenhängende Abschnitte den Counter nicht. Eine bessere Lösung ist, die Anzahl aller Elemente zu speichern (`size`) und diese mit jedem Schritt zu dekrementieren. *NumPy* greift diese Idee auf, zählt allerdings von null an aufwärts.

## 13.3 Vorteile des Iterators

Iteratoren sind eine *elegante* Abstraktion, weil Sie es uns ermöglichen, zusammenhängenden Code für unzusammenhängende Arrays zu schreiben, der dadurch gut verständlich und trotzdem noch schnell ist. Gegenüber zusammenhängenden Arrays gibt es nur kleine Geschwindigkeitseinbußen. Diese stammen aus dem Makro `PyArray_ITER_NEXT`, das bei jedem Aufruf überprüft, ob das nächste zu lesende Teilarray zusammenhängend ist. Es ist, dank des Iterators, nicht mehr nötig im Speicher nicht zusammenhängende Arrays so zu kopieren, dass diese anschließend zusammenhängend vorliegen. Das spart nicht nur Speicherplatz, sondern auch Zeit. Außerdem haben Iteratoren geholfen, Broadcasting in *NumPy* einzukapseln, was aber bei ihrer Entwicklung nicht geplant war.

## 13.4 Broadcasting

Mein letzter Abschnitt erläutert das Broadcasting. In *NumPy* bedeutet Broadcasting, dass über mehrere Arrays, die nicht dieselbe Form haben müssen, gleichzeitig iteriert wird. Dies ist bei einer Matrix-Addition vorteilhaft. Eigentlich müsste man für jedes Array einen eigenen Iterator nutzen, stattdessen gibt es aber einen Multi-Iterator. Dafür müssen ein paar Vorbereitungen getroffen werden. So werden an Arrays mit weniger Dimensionen weitere Dimensionen angehängt, die mit Einsen aufgefüllt werden. Alle Dimensionen, die gleichzeitig durchlaufen werden, müssen entweder die gleiche Größe haben oder eine dieser Dimensionen darf nur ein Element besitzen. Sollten diese Dimensionen nicht die gleiche Anzahl an Elementen haben, so wird im resultierenden Array die Dimension aus dem Array mit den meisten Elementen in dieser Dimension gewählt. Sollte ein Array in einer Dimension nur ein Element haben, wird so getan, als ob dieses Element an jeder Position während der Iteration steht.

Ein Beispiel liefert die Matrix-Addition (Listing 13.6). Die Matrizen sollen dieselbe Form haben. Die gegebenen Matrizen `in1` und `in2` sind vom Typ double. Als Pointer dienen `i1p` und `i2p`, mit denen diese Matrizen im Speicher durchlaufen werden. Bei `PyArray_MultiNew` bestimmt das erste Argument, aus wie vielen Matrizen der Multi-Iterator zusammengebaut wird. Die folgenden Argumente sind dann die Input-Matrizen. Im Ausgabe-Array `out` steht in diesem Fall die Addition der Matrizen. Nachdem das output-Array erzeugt ist, wird `op` auf den Beginn des output-Arrays gesetzt. Danach beginnt die Iteration über beide input-Arrays. In Zeile 13 und 14 werden die Zeiger auf die aktuellen Werte gesetzt, um anschließend die Addition auszuführen. Jetzt wird der Zeiger des output-Arrays eine Stelle weiter gesetzt und zuletzt werden die beiden input-Iteratoren parallel durch den Aufruf `PyArray_MultiIter_NEXT(multi)` erhöht.

Listing 13.6: Max-Suche mit Hilfe des Iterators

```
PyObject *multi;
PyObject *in1, *in2;
double *i1p, *i2p, *op;

multi=PyArray_MultiNew(2, in1, in2);

out=PyArray_SimpleNew(PyArray_MultiIter_NDIM(multi),
        PyArray_MultiIter_DIMS(multi), NPY_DOUBLE);

op=PyArray_DATA(out);

while(PyArray_MultiIter_NOTDONE(multi)) {
    i1p=PyArray_MultiIter_DATA(multi, 0);
    i2p=PyArray_MultiIter_DATA(multi, 1);
    *op = *i1p + *i2p;
    op += 1;
    PyArray_MultiIter_NEXT(multi);
}
```

(Wilson and Oram, 2007a, Kapitel 19.6.2)

KAPITEL 14

# Schnelle Matrixmultiplikation

**Ausarbeitung von Daniel Wulfert**

## 14.1 Einleitung

### 14.1.1 Wofür braucht man Matrizen?

Matrizen sind für fast alle Bereiche, von Problemen der Linearen Algebra, der Physik, in Graphenalgorithmen und unzähligen anderen Anwendungen, von Bedeutung. In vielen Fällen ist ein Algorithmus von der Schnelligkeit des Matrixmultiplikationsalgorithmus abhängig.

### 14.1.2 Die Suche nach dem Exponenten $\omega$

Der Exponent der Matrixmultiplikation ist die kleinste reelle Zahl $\omega$, so dass für alle $\epsilon > 0$, $\mathcal{O}(n^{\omega+\epsilon})$ arithmetische Operationen für die Multiplikation zweier $n \times n$ Matrizen ausreichen.

### 14.1.3 Eine untere Schranke

Vor Strassen glaubten viele, dass ein Algorithmus zur Multiplikation zweier Matrizen mindestens $\mathcal{O}(n^3)$ Operationen braucht, und der Standard Algorithmus optimal sei. Auch wenn der Coppersmith-Winograd Algorithmus seit mehr als zwanzig Jahren der bisher schnellste Algorithmus mit einer Laufzeit von $\mathcal{O}(n^{2{,}38})$ ist, glaubt man das ein Algorithmus der Größenordnung $\omega = 2$ zur Matrixmultiplikation ausreicht. Das $\omega$ nicht kleiner als zwei sein kann ist nachvollziehbar, da eine $n \times n$ Matrix $n^2$ Werte hat, und jeder dieser mindestens einmal gelesen werden muss.

### 14.1.4 Eine obere Schranke

Die beste, bekannteste obere Schranke für den Exponenten ist $\omega < 2,83$. Dieses Ergebnis wurde mit einem Beweis von Winograd und Coppersmith im Jahre 1987 erzielt. Allerdings hat der Winograd-Coppersmith-Algorithmus für die Praxis wenig Relevanz, da der asymptotische Gewinn, wegen der großen Konstanten hinter der $\mathcal{O}$-Notation, nur für astronomisch große Matrizen bemerkbar wäre. Zur Zeit wird ein weiterer Ansatz für das Erreichen der Schranke $\omega = 2$ verfolgt, der auf Gruppentheorie basiert.

## 14.2 Iterative Berechnung

### 14.2.1 Algorithmus

Gegeben seien zwei Matrizen $A = (a_{i,j})_{i=1,\dots,n;j=1,\dots,l}$ und $B = (b_{i,j})_{i=1,\dots,l;j=1,\dots,m}$. Dann ergibt sich die Ergebnismatrix $A \cdot B = (c_{i,j})_{i=1,\dots,n;j=1,\dots,m}$ mit Hilfe der Summenformel $c_{i,j} = \sum_{k=1}^{l} a_{i,k} \cdot b_{k,j}$. Dieser recht einfache Algorithmus berechnet also iterativ für Matrizen beliebiger Größe, wobei die Zeilengröße der ersten Matrix mit der Spaltengröße der zweiten Matrix übereinstimmen muss, das Produkt beider.

Beispiel für eine $2 \times 2$ Matrix:

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} = \begin{pmatrix} a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} & a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2} \\ a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{2,1} & a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} \end{pmatrix}$$

### 14.2.2 Laufzeit

Die Laufzeit des Algorithmus hängt von der Größe der beiden Matrizen ab, die multipliziert werden. Wie in der obigen Definition sei hier eine Matrix $A$ der Größe $n \times l$ und eine Matrix $B$ der Größe $l \times m$ gegeben. Die Größe der Produktmatrix $C$ ist dann $n \times m$ und die Summenformel die jedes der $n \cdot m$ vielen $c_{i,j}$ berechnet, hat dann jeweils $l$ Multiplikationen und $l - 1$ Additionen. Daraus ergibt sich eine Laufzeit von $n \cdot m \cdot l$ Multiplikationen und $n \cdot m \cdot (l - 1)$ Additionen also insgesamt $2 \cdot n \cdot m \cdot l - n \cdot m$ Operationen.

Da sich die folgenden Algorithmen auf quadratische Matrizen der Größe $n \times n$ beziehen, wobei $n = 2^m$ ist und $m \in \mathbb{N}$ ist, wollen wir kurz die Laufzeit des iterativen Algorithmus für quadratische Matrizen ergänzen. Da wir nur für die Größen $l$ und $m$ $n$ einsetzen müssen folgt daraus, dass der Algorithmus für die Berechnung zweier $n \times n$ Matrizen $n^2 \cdot n = n^3$ Multiplikationen sowie $n^2 \cdot (n - 1) = n^3 - n^2$ Additionen braucht. Damit ergibt sich eine Laufzeit von $2n^3 - n^2$ bzw. $\mathcal{O}(n^3)$.

### 14.2.3 Implementierung

Die Implementierung dieses Algorithmus ist denkbar einfach und kurz, besteht aus 2 `for`-Schleifen die alle $c_{i,j}$ der Ergebnismatrix durchgehen und einer weiteren `for`-Schleife, die die Summenformel zur Berechnung von $c_{i,j}$ bereitstellt. Das Ganze sieht in Java-Code wie in Listing 14.1 aus.

Listing 14.1: Implementierung des iterativen Algorithmus

```
public static int[][] mul(int[][] a, int[][] b){
   int[][] product=new int[a.length][b[0].length];
   for (int row=0; row<a.length; row++){
      for (int bcol=0; bcol<b[0].length; bcol++){
         for (int acol=0; acol<a[row].length; acol++)
            product[row][bcol]+=a[row][acol]*b[acol][bcol];
      }
   }
   return product;
}
```

## 14.3 Rekursive Berechnung

### 14.3.1 Algorithmus

Für den rekursiven Algorithmus gehen wir von quadratischen Matrizen der Größe $n \times n$ aus wobei $n = 2^m$ und $m \in \mathbb{N}$ sein soll. Es lassen sich auch nicht quadratische Matrizen mit diesem Algorithmus berechnen, dazu müssen sie allerdings in eine quadratische Form gebracht werden, aber dazu später mehr. Im ersten Schritt werden nun die Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$ große Submatrizen aufgeteilt.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Mit Hilfe dieser Submatrizen wird mit ähnlicher Formel wie beim iterativen Algorithmus zur Berechnung einer $2 \times 2$ Matrix das Ergebnis berechnet.

$$C_{11} := A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} := A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} := A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} := A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Bei der Berechnung der Multiplikationen folgt nun der rekursive Aufruf, bei der nur noch eine $\frac{n}{2}$-mal so große Matrix betrachtet werden muss. Diese Aufrufe gehen dann so weit bis die Matrix eine Größe von 1 hat und nur noch eine gewöhnliche Zahlenmultiplikation durchgeführt werden muss.

### 14.3.2 Laufzeit

Für einen Schritt bzw. rekursivem Aufruf werden 8 Multiplikationen sowie 4 Additionen verwendet.

Sei $n = 2^m$ und

$$T(1) = 1$$

$$T(2) = 8 + 4 = 12$$

$$T(4) = 8 \cdot 12 + 4 \cdot 4 = 112$$

$$T(8) = 8 \cdot 112 + 4 \cdot 4 \cdot 4 = 960$$

dann erkennt man, dass

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + 4^{\log_2 n}$$

ist und da $4^{\log n} = 4^{\log 2^m} = 4^m = 2^{m2} = n^2$ ist folgt nun

$$= 8 \cdot T\left(\frac{n}{2}\right) + n^2$$

Kommen wir nun zum lösen der Rekursionsgleichung. Mehrmaliges anwenden von $T(n)$ liefert

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + n^2$$

$$= 64 \cdot T\left(\frac{n}{4}\right) + 8 \cdot \left(\frac{n}{2}\right)^2 + n^2$$

$$= 64 \cdot T\left(\frac{n}{4}\right) + 2n^2 + n^2$$

$$= 512 \cdot T\left(\frac{n}{8}\right) + 4n^2 + 2n^2 + n^2$$

was erkennen lässt, was passiert wenn man diese Schritte bis $T(1)$ weiter auflösen würde. Daraus folgt nun

$$T(n) = 8^m \cdot T(1) + \sum_{i=1}^{m} \left(2^{i-1} \cdot n^2\right)$$

$$= 8^m + n^2 \cdot \sum_{i=0}^{m-1} 2^i$$

Wegen $n = 2^m$ ist $8^m = n^{\log_2 8} = n^3$ und nach Anwendung der geometrischen Reihe auf die Summe folgt

$$T(n) = n^3 + \frac{1 - 2^m}{1 - 2} = n^3 + n^3 - n^2$$

$$= 2n^3 - n^2$$

was, wie man bereits erwartet hat, genau der Laufzeit des iterativen Algorithmus entspricht. Die gilt allerdings nur für quadratische Matrizen deren Größe eine Zweierpotenz ist.

Auch wenn die theoretische Laufzeit mit der der iterativen Variante bei entsprechenden Matrizen übereinstimmt, ist die rekursive Variante, in der Praxis, der iterativen in Bezug auf die Effizienz unterlegen. Grund hierfür ist die Beanspruchung des Stacks und der Overhead beim wiederholten Funktionsaufruf. Außerdem müssen für die Submatrizen neue Objekte angelegt werden, was die Laufzeit zusätzlich verlängert.

### 14.3.3 Implementierung

Für die eigentliche Implementierung werden zusätzlich noch Funktionen zur Matrixaddition und -subtraktion benötigt. Für nicht quadratischen Matrizen oder Matrizen deren Größe keine Zweierpotenz ist, kann man entsprechende Vorschaltfunktionen verwenden, auf die im späteren Kapitel eingegangen wird. Die Abbruchbedingung für die Rekursion ist, wie oben in der Rekursionsgleichung, bei einer Matrixgröße von eins.

Auf einen Beispielcode wird an dieser Stelle verzichtet, da er ähnlich dem des Strassen-Algorithmus ist, und die Implementierung in der Praxis, wegen der bei der Laufzeit erwähnten Effizienz, eher eine untergeordnete Rolle spielt.

## 14.4 Der Strassen-Algorithmus

### 14.4.1 Algorithmus

Der Strassen-Algorithmus, benannt nach dem Mathematiker Volker Strassen, arbeitet ähnlich dem rekursiven, naivem Algorithmus. Auch er teilt die Matrizen in jeweils vier $\frac{n}{2}$ große Matrizen. Nur die Berechnung an sich läuft anders. Zuerst werden sieben Hilfsmatrizen berechnet

$$M_1 := (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$M_2 := (A_{12} + A_{22}) \cdot B_{11}$$

$$M_3 := A_{11} \cdot (B_{12} - B_{22})$$

$$M_4 := A_{22} \cdot (B_{21} - B_{11})$$

$$M_5 := (A_{11} + A_{12}) \cdot B_{22}$$

$$M_6 := (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$M_7 := (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

Danach werden die Ergebnismatrizen folgendermaßen berechnet

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Durch zusätzliche Additionen und Subtraktionen wird also eine Multiplikation eingespart. Das sich dieser Aufwand trotzdem lohnt werden wir bei der Laufzeitberechnung sehen.

### 14.4.2 Laufzeit

Für die Berechnung der Hilfsmatrizen sind jeweils 10 Additionen bzw. Subtraktionen und 7 Multiplikationen notwendig. Für die Berechnung der Ergebnismatrix kommen noch einmal 8 Additionen und Subtraktionen dazu, also kommen wir auf insgesamt 18 Additionen und Subtraktionen und 7 Multiplikationen pro rekursivem Aufruf.

Nun ist

$$\begin{aligned} T(1) &= 1 \\ T(2) &= 7 + 18 = 25 \\ T(4) &= 7 \cdot 25 + 4 \cdot 18 = 247 \\ T(8) &= 7 \cdot 247 + 4 \cdot 4 \cdot 18 = 2017 \end{aligned}$$

und man erkennt, dass

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + 18 \cdot 4^{\log_2 n - 1} \\ &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2} \cdot 4^{\log_2 n} \\ &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2} \cdot n^2 \end{aligned}$$

ist. Nun wenden wir diese Formel wieder mehrfach hintereinander an, um diese rekursive Form in eine explizite Formel zu bringen.

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2} \cdot n^2 \\ &= 7 \cdot 7 \cdot T\left(\frac{n}{4}\right) + 7 \cdot \frac{9}{2} \cdot \left(\frac{n}{2}\right)^2 + \frac{9}{2} \cdot n^2 \\ &= 49 \cdot T\left(\frac{n}{4}\right) + \frac{63}{8} \cdot n^2 + \frac{9}{2} \cdot n^2 \\ &= 349 \cdot T\left(\frac{n}{8}\right) + \frac{441}{32} \cdot n^2 + \frac{63}{8} \cdot n^2 + \frac{9}{2} \cdot n^2 \end{aligned}$$

Daraus folgt nun schließlich, dass

$$\begin{aligned} T(n) &= 7^m \cdot T(1) + \sum_{i=1}^{m} \left[ \left(\frac{7}{4}\right)^{i-1} \cdot \frac{9}{2} n^2 \right] \\ &= 7^m + \frac{9}{2} n^2 \cdot \sum_{i=0}^{m-1} \left(\frac{7}{4}\right)^i \\ &= 7^m + \frac{9}{2} n^2 \cdot \frac{4}{3} \cdot \left(\frac{7}{4}\right)^m - \frac{9}{2} n^2 \cdot \frac{4}{3} \\ &= 7^m + 6n^2 \cdot \left(\frac{7}{4}\right)^m - 6n^2 \end{aligned}$$

ist, und da $n = 2^m$ folgt weiter

$$= n^{\log_2 7} + 6n^2 \cdot n^{\log_2 \frac{7}{4}} - 6n^2$$

Da $n^2 \cdot n^{\log_2 \frac{7}{4}} = n^2 \cdot n^{\log_2 7 - \log_2 4} = n^{\log 7}$ ist, erhalten wir

$$T(n) = 7n^{\log_2 7} - 6n^2$$

und somit eine Laufzeit von $\mathcal{O}\left(n^{\log_2 7}\right)$. Im Vergleich zu den naiven Algorithmen, die eine Laufzeit von $\mathcal{O}(n^3) = \mathcal{O}\left(n^{\log_2 8}\right)$ haben, ist das zwar eine Verbesserung, wenn auch auf den ersten Blick eine geringe.

### 14.4.3 Implementierung

Bei der praktischen Implementierung wird nur so lange iteriert bis die Matrixdimension klein genug und der Standard-Algorithmus effizienter ist. Der so genannte Cut-Off hängt von der Implementierung und der Hardware des Systems ab, liegt aber meistens zwischen $n = 32$ bis $n = 128$.

Die Implementierung des Strassen Algorithmus ist in Listing 14.2 zu sehen.

Listing 14.2: Implementierung des Strassen Algorithmus

```
1  public static int[][] strassenCore(int [][] a, int[][] b){
2     //wenn die Matrix Dimension klein genug ist wird der
3     //Standard-Algorithmus verwendet
4     if (a.length<cutOff) return mul(a, b);
5     int subSize=a.length/2;
6
7     //Generierung der Submatrizen
8     int[][] a11=subMatrix(subSize, a, 0, 0);
9     int[][] a12=subMatrix(subSize, a, 0, 1);
10    int[][] a21=subMatrix(subSize, a, 1, 0);
11    int[][] a22=subMatrix(subSize, a, 1, 1);
12    int[][] b11=subMatrix(subSize, b, 0, 0);
13    int[][] b12=subMatrix(subSize, b, 0, 1);
14    int[][] b21=subMatrix(subSize, b, 1, 0);
15    int[][] b22=subMatrix(subSize, b, 1, 1);
16
17    //Berechnung der Hilfsmatrizen
18    int[][] m1=strassenCore(plus(a11, a22), plus(b11, b22));
19    int[][] m2=strassenCore(plus(a21, a22), b11);
20    int[][] m3=strassenCore(a11, sub(b12, b22));
21    int[][] m4=strassenCore(a22, sub(b21, b11));
22    int[][] m5=strassenCore(plus(a11, a12), b22);
23    int[][] m6=strassenCore(sub(a21, a11), plus(b11, b12));
24    int[][] m7=strassenCore(sub(a12, a22), plus(b21, b22));
25
26    //Berechnung des Produktes
27    int[][] c11=plus(sub(plus(m1, m4), m5), m7);
28    int[][] c12=plus(m3, m5);
29    int[][] c21=plus(m2, m4);
30    int[][] c22=plus(plus(sub(m1, m2), m3), m6);
31
32    //Submatrizen zusammenfuehren
33    int[][] product=new int[a.length][a.length];
34    for (int i=0;i<subSize;i++){
```

```
35          System.arraycopy(c11[i], 0, product[i], 0, subSize);
36          System.arraycopy(c12[i], 0, product[i], subSize, subSize);
37       }
38       for (int i=0;i<subSize;i++){
39          System.arraycopy(c21[i], 0, product[i+subSize], 0, subSize);
40          System.arraycopy(c22[i], 0, product[i+subSize], subSize, subSize);
41       }
42       return product;
43    }
```

Durch Verwendung des Cut-Off und da Multiplikationen in der Praxis teurer sind als Additionen, lohnt sich der Strassen Algorithmus meisst schon ab einer Matrixgröße von $n = 64$. Bei noch größeren Matrizen war er in praktischen Tests schon doppelt so schnell wie die iterative Variante.

## 14.5 Nicht quadratische Matrizen

Der oben aufgeführte Kern-Strassen-Algorithmus kann nur Matrizen multiplizieren die quadratisch sind und deren Größe eine Zweierpotenz ist. Um mit Hilfe des Strassen-Algorithmus oder des Rekursiven Algorithmus auch nicht quadratische Matrizen zu berechnen, kann man gewisse Vorschaltfunktionen verwenden.

### 14.5.1 „Auffüll“-Methode

Die einfachste Art eine nicht quadratische Matrix in eine quadratische Matrix zu überführen ist, den verbleibenden Teil der zur nächsten quadratischen Matrix mit Zweierpotenzgröße fehlt, für die Berechnung mit Nullen zu füllen und nachher wieder „abzuschneiden“. So lassen sich sämtliche Matrizen auch rekursiv berechnen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \longrightarrow \begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Der Nachteil, dieser recht einfach zu implementierenden Methode ist, dass die Laufzeit nicht mehr stetig mit der Matrizengröße wächst, sondern bei kleineren Veränderungen sprunghaft ansteigen kann. Die Laufzeit für die Berechnung einer Matrix $A$ der Größe $n \times l$ und einer Matrix $B$ der Größe $l \times m$ wäre zum Beispiel beim rekursiven Algorithmus $2 \cdot \left(2^{\lceil \log_2 \max(n,m,l) \rceil}\right)^3 - \left(2^{\lceil \log_2 \max(n,m,l) \rceil}\right)^2$.

### 14.5.2 „Aufteil“-Methode

Eine weitere Möglichkeit das Produkt zweier nicht quadratische Matrizen bzw. einer Matrix deren Größe keine Zweierpotenz ist rekursiv zu berechnen ist sie so aufzuteilen, dass eine möglichst große quadratische Matrix und drei kleinere Matrizen entstehen. Die quadratische Matrix kann nun rekursiv berechnet werden, während die anderen Drei mit dem

Standard-Algorithmus berechnet werden. Die Matrizen müssen dann nur noch zusammengeführt werden.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \longrightarrow \left(\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ \hline a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}\right)$$

Nachteil dieser Methode ist die etwas schwierigere Implementierung und die, bei quadratischen Matrizen minimal langsamere, dafür viel stabilere, Laufzeit gegenüber der „Aufteil“-Methode.

Eine Beispielimplementierung der „Aufteil“-Methode als Vorschaltfunktion für den Strassen Algorithmus in Java findet man in Listing 14.3.

Listing 14.3: Implementierung des „Aufteil“-Algorithmus

```
1  public static int[][] strassen(int[][] a, int[][] b){
2     //Berechnung der kleinsten Seite
3     int minSize=Math.min(Math.min(a.length, a[0].length),b[0].length);
4     boolean quadratic=(a.length==a[0].length)&&(b.length==b[0].length)
5        &&(a.length==b.length);
6
7     //wenn Matrizen quadratisch sind und Zweierpotenzgroesse haben werden
8     //sie mit dem Kern-Strassen-Algorithmus berechnet
9     if (quadratic&&minSize==Math.pow(2.0, Math.floor(
10       Math.log(minSize)/Math.log(2.0)))) return strassenCore(a, b);
11
12    //Matrizen die zu klein oder zu entartet sind werden mit dem
13    //Standard Algorithmus der Matrixmultiplikation berechnet
14    if ((quadratic&&minSize<cutOff)||minSize-1<cutOff)
15       return mulx(a, b);
16
17    //Berechnung der groessten quadratischen Submatrix innerhalb der zu
18    //multiplizierenden Matrizen A und B
19    int size=(int)Math.pow(2.0, Math.floor(Math.log(minSize-1)/Math.log(2.0)));
20
21    int[][] a11=new int[size][size], b11=new int[size][size];
22    for (int i=0;i<size;i++){
23       System.arraycopy(a[i], 0, a11[i], 0, size);
24       System.arraycopy(b[i], 0, b11[i], 0, size);
25    }
26
27    int[][] a12=new int[size][a[0].length-size];
28    int[][] b12=new int[size][b[0].length-size];
29    for (int i=0;i<size;i++){
30       System.arraycopy(a[i], size, a12[i], 0, a[0].length-size);
31       System.arraycopy(b[i], size, b12[i], 0, b[0].length-size);
32    }
33
34    int[][] a21=new int[a.length-size][size];
35    int[][] b21=new int[b.length-size][size];
36    for (int i=size;i<a.length;i++)
```

```
37          System.arraycopy(a[i], 0, a21[i-size], 0, size);
38       for (int i=size;i<b.length;i++)
39          System.arraycopy(b[i], 0, b21[i-size], 0, size);
40
41       int[][] a22=new int[a.length-size][a[0].length-size];
42       int[][] b22=new int[b.length-size][b[0].length-size];
43       for (int i=size;i<a.length;i++)
44          System.arraycopy(a[i], size, a22[i-size], 0, a[0].length-size);
45       for (int i=size;i<b.length;i++)
46          System.arraycopy(b[i], size, b22[i-size], 0, b[0].length-size);
47
48       //Eigentliche Berechnung
49       int[][] c11=plus(strassenCore(a11, b11), strassen(a12, b21));
50       int[][] c12=plus(strassen(a11, b12), strassen(a12, b22));
51       int[][] c21=plus(strassen(a21, b11), strassen(a22, b21));
52       int[][] c22=plus(strassen(a21, b12), strassen(a22, b22));
53
54       //Zusammenfuehren der Submatrizen zur Ergebnismatrix
55       int[][] product=new int[a.length][b[0].length];
56       for (int i=0;i<size;i++){
57          System.arraycopy(c11[i], 0, product[i], 0, size);
58          System.arraycopy(c12[i], 0, product[i], size, b[0].length-size);
59       }
60       for (int i=size;i<a.length-size;i++){
61          System.arraycopy(c21[i-size], 0, product[i], 0, size);
62          System.arraycopy(c22[i-size], 0, product[i], size, b[0].length-size);
63       }
64       return product;
65    }
```

KAPITEL 15

# Suffixarray-Konstruktion

**Ausarbeitung von Bernd Hesse**

## 15.1 Einführung

### Einleitung

Dieses Kapitel beschäftigt sich mit verschiedenen Möglichkeiten ein Suffixarray zu konstruieren.

### 15.1.1 Was ist ein Suffixarray?

Ein Suffixarray ist ein Array das die Suffixe eines Strings in lexikographischer Reihenfolge enthält.

Der String *Raumschiff* enthält 11 Suffixe: Raumschiff, aumschiff, umschiff, mschiff, schiff, chiff, hiff, iff, ff, f und das leere Suffix. Diese Suffixe in lexikographische Reihenfolge gebracht sieht so aus:
{ '', 'aumschiff', 'chiff', 'f', 'ff', 'iff', 'mschiff', 'Raumschiff', 'schiff', 'umschiff' }
Das leere Suffix wird immer an den Anfang sortiert, hat aber in den meisten Fällen keinerlei Relevanz. Aus diesem Grund wird das leere Suffix im Folgenden ignoriert und weggelassen. In den meisten Fällen ist der Original-String, zu dem das Suffixarray gebildet wurde referenzierbar, um nicht unnötig Speicherplatz zu verschwenden gibt man daher nicht den Suffix selbst im Suffixarray an, sondern die Startposition des jeweiligen Suffix im Original-String. Der String *Raumschiff* ist zehn Zeichen lang:

| R | a | u | m | s | c | h | i | f | f |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Das Suffixarray dazu sieht dann also so aus: {1, 5, 9, 8, 6, 7, 3, 0, 4, 2}

### 15.1.2 Anwendungsmöglichkeiten

Viele Probleme lassen sich mit Hilfe von Suffixarrays einfacher lösen. Algorithmen die Suffixarrays konstruieren können durch Modifikation der Eingabe auch zur Lösung anderer Probleme genutzt werden. Ein paar Anwendungsmöglichkeiten sollen hier nun kurz genannt werden:

**Speicherplatz sparen** Es hat sich herausgestellt das Suffixbäume nicht gerade sparsam mit Speicherplatz umgehen - Um den Speicherverbrauch zu senken wurden überhaupt erst Suffixarrays entwickelt.

**Als Hilfsmittel zur Substring-Suche** Ist das Suffixarray zu einem String bereits konstruiert, lassen sich mit Hilfe des Suffixarrays schnell alle Substrings finden, die mit einem bestimmten String beginnen. Man kann zum Beispiel auf ein Suffixarray einfach eine binäre Suche anwenden und findet alle Vorkommen der gesuchten Substrings sehr schnell, da diese im Array alle direkt hintereinander liegen.

**Burrows-Wheeler-Transformation** Durch Modifikation der Eingabe können Algorithmen die Suffixarrays konstruieren auch verwendet werden um die Burrows-Wheeler Transformation durchzuführen. Die Burrows-Wheeler Transformation wird zum Beispiel bei der vor allem in der Unix-Welt bekannten bzip2-Datenkomprimierung genutzt.

## 15.2 Konstruktion von Suffixarrays

### 15.2.1 Umweg über einen Suffixbaum

In diesem Abschnitt wird erst einmal beschrieben wie ein Suffixbaum definiert ist. Anschliessend wird über eine recht simple 'zu Fuß'-Methode ein Suffixbaum aufgebaut und aus Diesem dann im letzten Schritt ein Suffixarray gebildet.

#### Was ist ein Suffixbaum?

Das Buch 'Algorithms on Strings, Trees and Sequences' (siehe 15.2.2) definiert Suffixbäume so:

> A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out S[i...m].

Um diese formellere Beschreibung direkt an einem Beispiel nachvollziehen zu können, ist in Abbildung 15.1 ein Beispiel-Suffixbaum zum String *abbab* abgebildet.

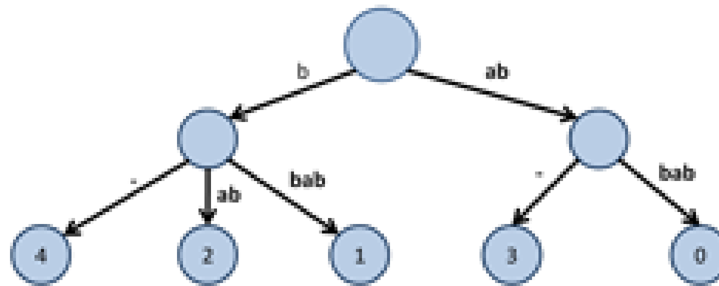



Abbildung 15.1: Suffixbaum zu 'abbab'

#### Konstruktion eines Suffixbaums

Die naive Methode einen Suffixbaum zu erzeugen soll nun einfach anhand eines Beispiels gezeigt werden. Als Beispiel-String soll hier *aabbabbab* dienen.

Zunächst haben wir lediglich die Wurzel die erstmal noch keine Kanten besitzt. Wir werden nun mit dem kürzesten Suffix beginnen und uns dann zum längsten Suffix voran arbeiten.

Der kürzeste Suffix von *aabbabbab* ist (den leeren Suffix ausgenommen) *b*. Wir schauen nun ob von der Wurzel bereits eine ausgehende Kante mit dem Buchstaben *b* existiert. Bisher existiert noch keine Kante, also können wir direkt eine Kante mit Beschriftung *b* erzeugen, einen Knoten an die neue Kante setzen und an diesen direkt eine weitere Kante mit Beschriftung -. Die mit - beschriftete Kante soll nun noch auf ein Blatt mit Inhalt *8* zeigen. *8* ist die Position des Suffix *b* in *aabbabbab*. Das Blatt haben wir für den Fall das weitere Suffixe mit *b* beginnen nicht direkt an die *b*-Kante gehangen. Dieser Schritt ist in Abbildung 15.2 zu sehen.

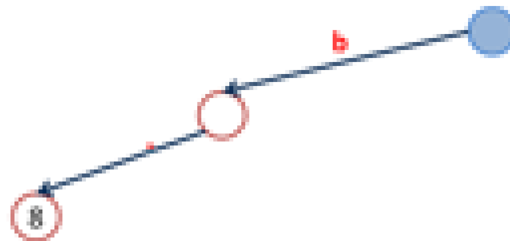



Abbildung 15.2: Anfangsphase der Suffixbaum-Konstruktion für 'aabbabbab'

Als nächstes folgt das Suffix *ab*. Es existiert bisher an der Wurzel keine Kante die mit *'a'* beginnt, also wird wieder wie beim ersten Suffix eine neue Kante *ab* erzeugt (und ebenso ein '-'-Knoten und ein Blatt mit Beschriftung '7'). Um Kanten und Knoten zu sparen wird *ab* hier nicht in einzelne Kanten mit Zwischenknoten aufgeteilt.

Das dritte Suffix das betrachtet wird ist *bab*. Es werden wieder erstmal die Kanten die die Wurzel verlassen betrachtet: Eine Kante 'b' existiert bereits und dieser Kante wird nun gefolgt. Am neuen Knoten werden wieder die ausgehenden Kanten untersucht: Hier gibt es allerdings keine Kante die mit 'a' beginnt, daher erzeugen wir einfach eine Kante 'ab' und

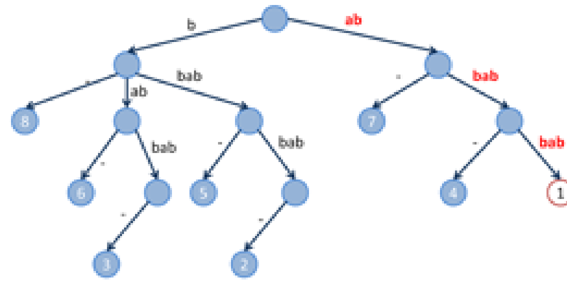



Abbildung 15.3: Fast vollständiger Suffixbaum für 'aabbabbab'

hängen wieder einen Knoten und eine Kante die auf ein Blatt mit Beschriftung '6' an die neue Kante.

Diese Vorgehensweise kann für alle Suffixe bis zum längsten Suffix angewandt werden. Beim Suffix *'aabbabbab'* gibt es nun allerdings eine Besonderheit: Es gibt bereits eine Kante die mit 'a' beginnt - Diese Kante kann aber nicht verfolgt werden, da der zweite Buchstabe 'b' nicht mit dem zweiten Buchstaben unseres Suffixes übereinstimmt. In diesem Falle muss die Kante 'ab' daher aufgeteilt werden. Im letzten Schritt werden noch die Knoten die nur einen Nachfolger haben direkt durch das jeweilige daran hängende Blatt ersetzt. Der Suffixbaum kurz vor dem längsten Suffix ist abgebildet in Abbildung 15.3 und der vollständige Suffixbaum in Abbildung 15.4.

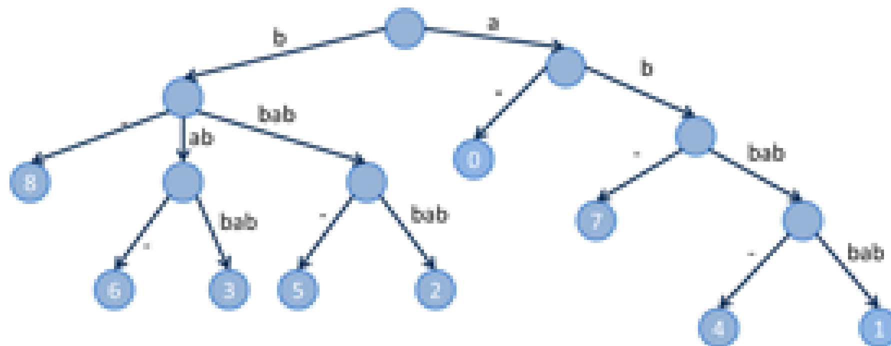



Abbildung 15.4: Vollständiger Suffixbaum für 'aabbabbab'

#### Suffixarray aus Suffixbaum erzeugen

Ein Suffixarray aus einem Suffixbaum zu erzeugen ist eine sehr einfache Angelegenheit: Man muss den Suffixbaum lediglich in lexikographischer Reihenfolge traversieren.

Auf dem Weg jeweils zu einem Blatt ergibt sich durch Konkatenation der Kantenbeschriftungen jeweils das Suffix. Das erreichte Blatt gibt die Position des Suffixes im String an.

Liest man das Suffixarray des Strings *'aabbabbab'* aus dem Suffixbaum (siehe Abbildung 15.4) ab, erhält man daher: { *0, 7, 4, 1, 8, 6, 3, 5, 2* }

### 15.2.2 Direkter Weg mit dem Skew-Algorithmus

Um Suffixbäume schnell zu konstruieren hatte Farach den Ansatz zunächst den Suffixbaum mit Suffixen die an geraden Stellen beginnen zu bilden, dann den Suffixbaum mit Suffixen die an ungeraden Stellen beginnen. Diese beiden Suffixbäume lassen sich dann effizient verschmelzen.

Juha Kärkkäinnen und Peter Sanders kamen mit dem Skew-Algorithmus auf die Idee diesen Ansatz auf Suffixarrays zu übertragen. Es stellte sich jedoch heraus dass gerade der letzte Merge-Schritt sich auf Eigenschaften von Suffixbäumen stützt, die in Suffixarrays so nicht vorhanden sind. Bildet man aber nicht die Suffixarrays der Suffixe an geraden und ungeraden Stellen, sondern die Suffixarrays der Suffixe mit Stellen i mod 3 != 0 und Stellen i mod 3 = 0, lässt sich der Merge-Schritt wieder einfach durchführen.

#### Funktionsweise

Der Skew-Algorithmus besteht aus drei Teilen:

1. Suffixarray der Suffixe an Positionen i mod 3 != 0 konstruieren
2. Suffixarray der Suffixe an Positionen i mod 3 = 0 konstruieren
3. Beide Suffixarrays verschmelzen

Wir werden den Skew-Algorithmus nun anhand des Strings *abrakadabra* nachvollziehen. Die Arbeitsweise ist grob in Abbildung 15.5 anhand des Strings *abrakadabra* in Anlehnung an die Abbildung aus dem Original-Dokument zum Skew-Algorithmus (siehe 15.2.2) nachgebildet. Wir werden nun die einzelnen Schritte für den String *abrakadabra* einmal durchgehen.

Hinweis: Die folgende Beschreibung ignoriert die Tatsache das eigentlich nicht direkt mit den Suffixen gearbeitet wird, sondern diese eigentlich nur über die Position der Suffixe im Original-String referenziert werden und tut an mehreren Stellen, aus Gründen der übersicht, so als ständen in den Arrays nicht die Positionen der Suffixe sondern die Suffixe selbst.

#### 1. Schritt: Suffixarray der Suffixe an Positionen i mod 3 != 0 konstruieren

Zuerst werden Tripel gebildet die aus Substrings bestehen, die jeweils an den Positionen $i$ mod $3! = 0$ im Original-String beginnen. Die $i \mod 3 = 1$ Tripel wären also: bra, kad, abr und a. Die i mod 3 = 2 Tripel sind: rak, ada und bra.

Diese Tripel werden nun mittels Radix-Sort sortiert: Die Tripel in sortierter Reihenfolge lauten dann: a, abr, ada, bra, bra, kad, rak. Also sind die Tripel mit $i \mod 3 = 1$ a, abr, bra und kad und die Tripel mit $i \mod 3 = 2$ sind ada, bra und rak.

Im nächsten Schritt vergibt man den sortierten Tripeln lexikographische Namen, dabei wird den Suffixen in der Reihenfolge wie sie sortiert vorkommen eine Zahl zugewiesen. Unterscheidet sich das Suffix nicht vom davor sortierten Suffix erhält es dieselbe Zahl, ansonsten die nächsthöhere Zahl. Das erste Suffix bekommt den Namen '1'. Der lexikographische Name dient also als Identifizierer für ein Tripel. Wenn alle Tripel einen eigenen lexikographischen Namen bekommen sind die Tripel also alle verschieden. Wenn die Tripel aber alle verschieden sind kann der lexikographische Name des Tripels auch als lexikographischen Namen für den

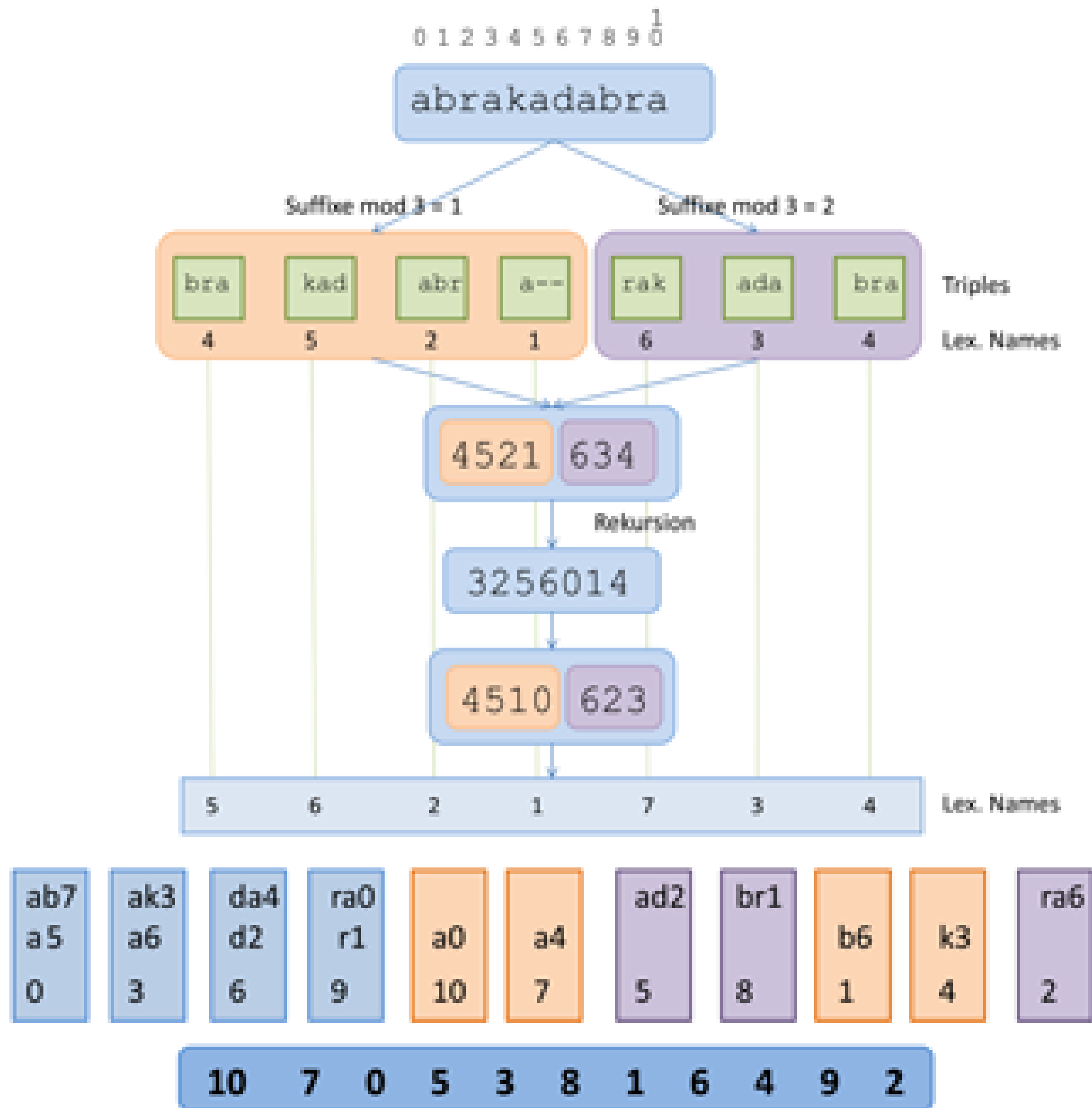



Abbildung 15.5: Skew-Algorithmus angewandt auf 'abrakadabra'

gesamten Substring dienen, der an derselben Stelle wie das jeweilige Tripel im String beginnt: Da die Tripel alle unterschiedlich sind können auch die entsprechenden Suffixe anhand der ersten 3 Zeichen (was gerade die Tripel sind) unterschieden werden.

Sind die lexikographischen Namen nicht eindeutig bedarf es etwas mehr Arbeit: In unserem Beispiel sind die Namen leider nicht sofort eindeutig, wir haben hier die Namen { 4, 5, 2, 1, 6, 3, 4 } (getrennt nach Suffixen mit Rest 1 gefolgt von den Suffixen mit Rest 2!). Mit diesen Namen, also *4521634*, rufen wir nun einfach wieder den Skew-Algorithmus auf, welcher uns natürlich die Suffixe dieses Strings in sortierter Reihenfolge zurückliefert. Die Suffixe von *4521634* lauten: { 4521634, 521634, 21634, 1634, 634, 34, 4 }, in sortierter Reihenfolge: { 1634, 21634, 34, 4, 4521634, 521634, 634 } oder konkret als Suffixarray (also Positionen in *4521634* statt der konkreten Suffixe): { 3, 2, 5, 6, 0, 1, 4 }. Das Ergebnis kann man nun so lesen: Das Suffix an der Position 3 wird vor alle anderen Suffixe sortiert, Das Suffix an der Position 2 folgt darauf, etc. Mit diesem Ergebnis kann man nun wieder lexikographische Namen bilden: Dem Suffx an Stelle 3 in *4521634* weisen wir den Namen *1* zu, dem Suffix an Stelle *2* den Namen2, dem Suffix an Stelle 5 den Namen 3, etc. Diese Namen können nun über die lexikographischen Namen, die wir dem Skew-Algorithmus hier ja als Eingabe gegeben haben, auch auf die *abrakadabra*-Suffixe bezogen werden und wir haben den zweiten Schritt abgeschlossen.

#### 2. Schritt: Suffixarray der Suffixe an Positionen i mod 3 = 0 konstruieren

Nun sind die Suffixe alle über einen lexikographischen Namen eindeutig identifizierbar und gleichzeitig gibt der lexikographische Name über dessen Wert auch die Sortierung der Suffixe an. Diese Informationen kann man praktischerweise direkt im zweiten Schritt ausnutzen.

Die Suffixe beginnend an Positionen $i \mod 3 = 0$ sind in unserem Falle: abrakadabra, akadabra, dabra und ra. Lässt man den ersten Buchstaben dieser Suffixe weg erhält man die Suffixe an Positionen $i \mod 3 = 1$. Dessen Sortierung kennen wir bereits aus dem ersten Schritt und ist über die lexikographischen Namen bereits gegeben. Um die Suffixe $imod3 = 0$ zu sortieren reicht es also die Paare $(S[i], S'[i])$ zu sortieren, wobei S[i] das i. Zeichen im String sein soll und S'[i] der lexikographische Name des im String darauf folgenden $i \mod 3 = 1$ Suffixes. Da die Sortierung der lexikographischen Namen bekannt ist reicht also ein Radix-Sort auf die erste Stelle der Paare $(S[i], S'[i])$.

Wir sortieren also bei diesem Beispiel die Paare: ('a',4), ('a',5), ('r',1). Hier sind diese Paare zufälligerweise bereits vor dem Radix-Sort-Aufruf in sortierter Reihenfolge. Beim Paar ('a',4) steht hier also das *'a'* für das erste *'a'* in *abrakadabra* und die *4* ist der lexikographische Name des Suffixes das auf das erste *'a'* im String folgt (also *brakadabra*). Damit wäre auch der zweite Schritt abgeschlossen.

#### 3. Schritt: Suffixarrays verschmelzen

Wir haben nun also das Suffixarray der Suffixe $i \mod 3! = 0$ (im folgenden mit SA12 bezeichnet) und das Suffixarray der Suffixe $i \mod 3 = 0$ (im folgenden als SA0 bezeichnet). SA12 und SA0 geben beide die Suffixe in sortierter Reihenfolge an und wir müssen diese nun zu einem Suffixarray verschmelzen.

Intuitiv würde man nun also das erste SA0-Suffix mit dem ersten SA12-Suffix vergleichen und den kleineren Suffix in das Ergebnis-Suffixarray übernehmen. Der erste SA0-Suffix ist bei *abrakadabra* gerade der String selbst und der erste SA12-Suffix ist *a* (gerade das Suffix mit dem kleinsten lexikographischen Namen). *a* ist dabei kleiner als *abrakadabra* und würde somit in das Ergebnis-Suffixarray übernommen. Der erste Suffix aus SA12 ist jetzt also weggefallen. Man wird als nächstes also *abra* aus SA12 betrachten und wieder *abrakadabra* aus dem SA0-Array. Hier ist wieder das SA12-Suffix kleiner und wird in das Ergebnis-Array übernommen. Diese Vergleiche werden solange fortgeführt bis alle Suffixe eines der beiden Arrays bereits in das Ergebnis-Array übernommen wurden. Die noch fehlenden Suffixe des Arrays das noch nicht komplett durchlaufen wurde können dann direkt in das Ergebnis-Array übernommen werden.

Bei den Suffixvergleichen kann man nun aber wieder auf Wissen aus dem ersten Schritt zurückgreifen und die Vergleiche dadurch verkürzen: Der Skew-Algorithmus benutzt zwei verschiedene Vergleiche, je nachdem ob der aktuell betrachtete SA12-Suffix ein $i \mod 3 = 1$-Suffix oder ein $i \mod 3 = 2$-Suffix ist.

Im ersten Fall wird das Paar (S[i],L'[i]) mit (S[j],L"[j]) verglichen. i soll dabei die Startposition des SA12-Suffixes im String sein, j die Startposition des SA0-Suffixes im String. S[i] bzw. S[j] soll das Zeichen an der jeweiligen Stelle im String sein. L'[i] soll der lexikographische Name des $i \mod 3 = 2$-Suffixes sein, welcher im String direkt nach Position i folgt und L"[j] soll der lexikographische Name des $i \mod 3 = 1$-Suffixes sein, welcher im String direkt auf Position j folgt. Gibt es jeweils gar kein Folge-Suffix wird als lexikographischer Name einfach die 0 verwendet.

Im zweiten Falle sieht der Vergleich ähnlich aus, hier wird allerdings ein Tripel verglichen: (S[i],S[i+1],L'[i+1]) und (S[j],S[j+1],L"[j+1]). Es werden also jeweils die ersten beiden Zeichen verglichen und die lexikographischen Name der jeweils darauf folgenden SA12-Suffixe.

Da die Ordnung der SA12-Suffixe aus dem ersten Schritt bekannt sind, wird hier also - wie auch im zweiten Schritt bei der Sortierung der SA0-Suffixe - durch bilden von Paaren (und hier auch Tripel) der Vergleich abgekürzt. Es wird ausgenutzt das auf das erste und zweite Zeichen eines SA0-Suffixes gerade ein SA12-Suffix folgt. Ebenso folgt auf das erste Zeichen eines SA12-Suffixes mit Rest 1 immer ein SA12-Suffix mit Rest 2. Auf ein SA12-Suffix mit Rest 2 folgt allerdings erst nach dem zweiten Zeichen ein SA12-Suffix, daher muss hier ein Tripel gebildet werden.

Das SA12-Suffix das wir zuerst verglichen hatten war *a*, ein $i \mod 3! = 0$-Suffix. Das erste SA0-Suffix war *abrakadabra*. Auf *a* folgt kein SA12-Suffix mehr und wir bilden das Paar ('a',0). Beim SA0-Suffix folgt auf das 'a' noch das SA12-Suffix 'brakadabra' mit dem lexikographischen Namen '5' - Hier wird also das Paar ('a',5) gebildet.('a',0) (entspricht also dem Suffix 'a') ist kleiner als ('a',5) (entspricht also dem Suffix abrakadabra') und die Position des SA12-Suffix wird daher in das Ergebnis übernommen. Wir rücken im SA12-Suffix nun also ein Suffix weiter: *abra* ist das neue SA12-Suffix, wieder ein Suffix mit Rest 1. Auf 'a' folgt hier das SA12-Suffix 'bra' mit lexikographischem Namen '4', das Paar lautet also ('a',4). Wir vergleichen nun also ('a',4) aus SA12 und ('a',5) aus SA0: Wieder ist das SA12-Suffix kleiner und die Position vom SA12-Suffix wird in das Ergebnis-Array übernommen. Wir rücken also wieder einen weiter in SA12: *adabra* ist das nächste SA12-Suffix, diesmal aber ein Suffix mit Rest 2 und unser SA0-Suffix ist immer noch *abrakadabra*. Wir bilden nun also ein Tripel: Das Tripel aus SA12 gebildet lautet: ('a','d',2) und das Tripel aus SA0: ('a','b',7). 'a', 'd' sind

wieder die Anfangsbuchstaben des SA12-Suffixes und die 2 ist der lexikographische Name des darauf folgenden SA12-Suffixes ('abra'). Ebenso sind 'a' und 'b' die Anfangsbuchstaben des SA0-Suffixes und die sieben der lexikographische Name des darauf folgenden SA12-Suffixes ('rakadabra'). Hier ist nun das SA0-Suffix kleiner als das SA12-Suffix und dessen Position wird daher in das Ergebnis-Array aufgenommen. Als nächstes wird man also in SA0 das nächste Suffix ziehen und in SA12 auf dem zuletzt betrachteten Suffix stehen bleiben.

Diese Vergleiche werden entsprechend fortgeführt bis eines der Array vollständig durchlaufen wurde, die restlichen Suffixe des anderen Arrays können dann direkt in das Ergebnis-Array übernommen werden.

### Laufzeit

Im ersten Schritt müssen die Suffixe an Positionen $i \mod 3! = 0$ erst sortiert werden, das schafft Radix-Sort in linearer Zeit. Anschliessend werden lexikographische Namen vergeben was auch in linearer Zeit passiert. Sind die Namen nicht eindeutig ruft der Skew-Algorithmus sich selbst wieder auf mit den bisherigen lexikographischen Namen als Eingabe. Der Schritt braucht also $T(n) = \mathcal{O}(n) + T(2n/3)$ (n ist die Eingabelänge). Der zweite Schritt besteht hauptsächlich aus einem Radix-Sort und benötigt ebenfalls $\mathcal{O}(n)$. Auch der Merge-Schritt benötigt nur lineare Zeit. Wir sind also insgesamt bei $T(n) = \mathcal{O}(n) + T(2n/3)$. Löst man die Rekursionsgleichung auf kommt man zu dem Ergebnis: $T(n) = \mathcal{O}(n)$.

## 15.3 Quellen

Für die Erarbeitung des Themas wurden folgende Quellen genutzt:


- Simple Linear Work Suffix Array Construction
  Juha Kärkkäinen, Peter Sanders, 2003
  `http://www.cs.helsinki.fi/juha.karkkainen/publications/icalp03.pdf`
- Algorithms on Strings, Trees, and Sequences
  Dan Gusfield, 1997
  Cambridge University Press
- wikipedia.org: 'Suffixarray' & 'Suffixbaum'
  `http://de.wikipedia.org/wiki/Suffixarray`
  `http://de.wikipedia.org/wiki/Suffixbaum`

KAPITEL 99

# Hinweise zur richtigen Benutzung von LaTeX

## 99.1 Einleitung

In diesem Kapitel stellen wir einige Hinweise zur richtigen Benutzung von LaTeX bereit. Insbesondere werden häufig gemachte Fehler vorgestellt.

## 99.2 Häufig gemachte Fehler

**Zu große Abstände nach Satzzeichen.** LaTeX setzt hinter einem Punkt einen größeren Abstand als sonst zwischen Wörtern. Normalerweise ist das gewollt, damit zwei Sätze voneinander besser getrennt sind. Es führt aber zu Problemen bei Abkürzungen wie z. B. Prof. Rahmann (beachte die zu großen Abstandslänge; LaTeX versucht schlau zu sein und interpretiert einen einzelnen Buchstaben wie `B.` nicht als Satz). Erstens kann LaTeX einen häufig nicht gewollten Zeilenumbruch einfügen; zweitens sollte der Abstand die normale Länge haben.

Will man einen Zeilenumbruch ausschließen, so kann man einen nichtumbrechenden Zwischenraum (non-breaking space) verweden, in LaTeX geht das mit dem Zeichen `~`. Will man den Umbruch zulassen, muss man dem Leerzeichen einen backslash voranstellen:

```
z.~B.\ Prof.~Rahmann
```

verhindert Umbrüche zwischen `z.` und `B.`, sowie swischen `Prof` und `Rahmann`.

**Fehlende Abstände nach Befehlen.** Der Befehl `\LaTeX` erzeugt LaTeX. wenn man nun schreibt (wie oben): `\LaTeX setzt`, erhält man: LaTeXsetzt; es fehlt der Abstand! Dasselbe Problem ergibt sich nach allen Befehlen, da das Leerzeichen das Befehlsende markiert.

Zur Lösung kann man entweder den Befehl einklammern, `{\LaTeX} setzt`, oder (einfacher), wieder backslash-space verwenden: `\LaTeX\ setzt`.

**Elementare mathematische Funktionen.** Variablennamen werden grundsätzlich kursiv geschrieben: $i, j, m, n$. Dazu wechselt man mit $ in den Mathe-Modus (und auch wieder zurück). Falsch wäre es, hier etwa *italics* text, *i, j, m, n*, zu benutzen (beachte das unterschiedliche Schriftbild).

Konstanten hingegen schreibt man nicht kursiv, insbesondere die imaginäre Einheit $\mathrm{i} = -1$ oder $\mathrm{e} \approx 2.71...$. Im Mathemodus bekommt man das z.B. mit

```
\mbox{\upshape i}
```

hin; eleganter aber ist es, wenn man sich einen eigenen Befehl dafür definiert, zum Beispiel `\imunit` für imaginary unit:

```
\newcommand{\imunit}{\mbox{\upshape i}}
```

Wichtig ist auch, dass Namen elementarer Funktionen wie sin, cos, log nicht kursiv geschrieben werden, ebenso wie Operatorennamen. Dafür stellt LaTeX bereits häufig vordefinierte Befehle zur Verfügung, etwa

`\sin`, `\cos`, `\log`.

Hingegen würde ein vorgebildeter Leser *log* als das Produkt $l \cdot o \cdot g$ interpretieren. Auch das Differential d wird nicht kursiv geschrieben: $\int x \, \mathrm{d}x = x^2/2$. Das wird leider häufig auch in ansonsten guten Büchern falsch gemacht.

**WYSIWYG.** Häufig wird der (oft nicht sichtbare) Fehler gemacht, bestimmte Textstellen nur optisch statt logisch auszuzeichnen. Nehmen wir an, dass wir sowohl zu definierende Begriffe als auch Spezies-Namen kursiv hervorheben wollen:

Eine *kontextfreie Grammatik* ist ein 4-Tupel ...
Das Bodenbakterium *C. glutamicum* lebt ...

Wenn wir dies jeweils mit `\textit{}` machen und uns dann entscheiden, dass wir alle zu definierenden Begriffe doch lieber unterstrichen fett haben wollen, bekommen wir ein Problem! Wir müssen jedes `\textit` durchgehen und prüfen, ob es für eine Definition oder anderweitig verwendet wird. Besser ist es, wenn wir eigene Befehle definieren, etwa:

```
\newcommand{\df}[1]{\emph{#1}}
\newcommand{\species}[1]{\textit{#1}}
```

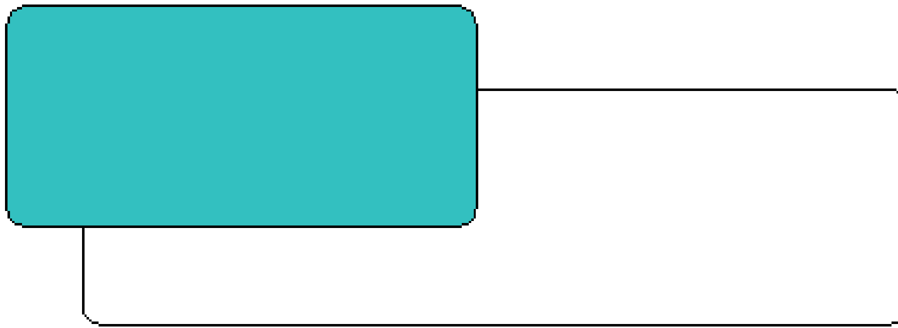


Abbildung 99.1: Ein Kasten

**Einbindung von Abbildungen und Tafeln.** Keinesfalls sollten Abbildungen und Tafeln direkt in den Text geschrieben werden, da dies im Zweifelsfall zu einem sehr schlechten Seitenumbruch führen kann. Sinnvoller ist es, wenn solche Objekte als frei verschiebbar definiert werden und man Hinweise zur gewünschten Positionierung gibt. Dies geschieht mit Hilfe der `figure` und `table`-Umgebungen.

Wir beschreiben den typischen Fall, das eine Grafik aus einer Datei eingebunden werden soll. Je nachdem, ob mit `latex` nach dvi und dann ps oder mit `pdflatex` direkt in ein pdf übersetzt wird, müssen die Abbildungen als encapsulated postscript (eps) oder als pdf/jpg/png vorliegen. Wenn man beim Dateinamen keine Endung angibt und beide Versionen zur Verfügung stellt, wird die richtige automatisch gewählt.

Der Code zur Einbindung von Abbildung 99.1 sieht so aus:

```
\begin{figure}[t!]
\includegraphics{kasten}
\caption{\label{fig:kasten}Ein Kasten}
\end{figure}
```

Dabei nehmen wir an, dass Dateien `kasten.eps` und `kasten.png` im aktuellen Verzeichnis existieren.

Kleine Tabellen kann man direkt in den Text einbinden:

| $x$ | 1.20 |
|---|---|
| $y$ | 12.30 |

. Das ist aber nicht so schön. Man könnte sie zwischen zwei Absätze in eine `center`-Umgebung einfügen, allerdings ergibt sich bei größeren Tabellen wieder das Problem des Seitenumbruchs.

| $x$ | 1.20 |
|---|---|
| $y$ | 12.30 |

Besser setzt man auch Tabellen beweglich in eine `\table`-Umgebung, etwa mit folgendem Code für Tabelle 99.1:

```
\begin{table}[b!]\centering
\begin{tabular}{l|r}
$x$ &  1.20\\\hline
```

```
$y$ & 12.30\\
\end{tabular}
\caption{\label{tab:daten}Wichtige Daten}
\end{table}
```

Wichtig ist auch, dass jedes `figure` oder `table`-Objekt im Text referenziert werden muss, und wenn es durch ein einfaches (siehe Abbildung 99.1) ist. Der Leser will schließlich wissen, *wann* er auf die Abbildung schauen soll.

Jedes Objekt bekommt mit Hilfe von `\label{}` einen Namen, auf den man sich mit `\ref{}` beziehen kann. Auch die Seite, auf der ein Objekt steht, kann man mit `\pageref{}` ausgeben lassen: `Tabelle~\ref{tab:daten} auf Seite~\pageref{tab:daten}` erzeugt: Tabelle 99.1 auf Seite 130.

| $x$ | 1.20 |
|---|---|
| $y$ | 12.30 |

Tabelle 99.1: Wichtige Daten

# Literaturverzeichnis

Wikipedia. Iterator — wikipedia, die freie enzyklopädie, 2008a. URL `http://de.wikipedia.org/w/index.php?title=Iterator&oldid=44084583`. [Online; Stand 17. Juli 2008].

Wikipedia. Hamming weight — wikipedia, the free encyclopedia, 2008b. URL `http://en.wikipedia.org/w/index.php?title=Hamming_weight&oldid=215030740`. [Online; accessed 26-May-2008].

G. Wilson and A. Oram, editors. *Beautiful Code*. O'Reilly, 2007a.

G. Wilson and A. Oram, editors. *Beautiful Code*. O'Reilly, 2007b.