

7 Stringmatching

Gegeben sei ein Muster M mit der Länge m und ein Text T der Länge n . Ziel des Stringmatching ist das Finden aller Stellen i in T , an denen sich das Muster M befindet. Formaler gesprochen sollen alle Stellen i gefunden werden, für die $T[i:i+m-1] == M$ gilt. Die folgende Abbildung veranschaulicht das Ergebnis eines Stringmatches des Musters $M = \text{kakaokaki}$ mit einem Text T . Das Ergebnis des Matches ist $i = 3$ und $i = 37$.

$M = \text{kakaokaki}$
 $T = \text{diekakaokaki istkakaomitkakiweshalbsiekkakaokakiheisst}$

\uparrow $i = 3$ \uparrow $i = 37$

In diesem Abschnitt lernen wir teilweise sehr unterschiedliche Techniken für schnelles (d. h. deutlich schneller als $O(n \cdot m)$) Stringmatching kennen:

- Stringmatching mit endlichen Automaten (Abschnitt 7.2).
- Eine Verfeinerung davon, der Knuth-Morris-Pratt-Algorithmus (Abschnitt 7.3).
- Ähnlich funktioniert auch der Boyer-Moore-Algorithmus, nur wird das Muster von der anderen Richtung über den Text geschoben (Abschnitt 7.4).
- Der Rabin-Karp-Algorithmus verwendet eine ganz andere Technik, nämlich Hashing (Abschnitt 7.5).
- Auch der Shift-Or-Algorithmus verwendet eine von den anderen Algorithmen grundauf verschiedene bitbasierte Technik (Abschnitt 7.6).

7.1 Primitiver Algorithmus

Ein primitiver Algorithmus ist schnell gefunden und implementiert:

```
1 def match(M, T):  
2     matches = []  
3     for i in range(len(T) - len(M)):  
4         if all(T[i+j] == M[j] for j in range(len(M))):  
5             matches.append(i)  
6     return matches
```

Listing 7.1: Die Funktion *match* findet alle Stellen in T , die das Muster M enthalten


$M = \text{kakaokaki}$
 $T = \text{kakakaokakigibtsnicht}$

Mismatch

Abb. 7.1: Eine Beispielsituation während eines Stringmatchings: Hier könnte man gleich an Position 3 weitersuchen.

Alle Treffer, d. h. Stellen in T an denen sich eine Kopie von M befindet, werden in der Liste *matches* aufgesammelt. Die **for**-Schleife ab Zeile 3 durchläuft alle Positionen i des Textes T und fügt die Stelle i genau dann zu *matches* hinzu, falls die nachfolgenden $\text{len}(M)$ Zeichen mit den jeweiligen Zeichen aus M übereinstimmen.

Aufgabe 7.1

Die in Listing 7.1 gezeigte Funktion *match* kann auch durch eine einzige Listenkomprehension implementiert werden. Schreiben Sie die Funktion entsprechend um, und füllen sie hierzu die in folgendem Listing freigelassene Lücke:

```
def match(M,T):
    return [ ... ]
```

Die Laufzeit dieses primitiven Stringmatching-Algorithmus ist sowohl im Worst-Case-Fall als auch im Average-Case-Fall in $O(n \cdot m)$, wobei $m = \text{len}(M)$ und $n = \text{len}(T)$. Für jede der n Textpositionen in T müssen im schlechtesten Fall $O(m)$ Vergleiche durchgeführt werden, um Klarheit darüber zu erhalten, ob sich an der jeweiligen Position eine Kopie von M befindet oder nicht.

Wir werden sehen, dass die Laufzeit der schnellsten Stringmatching-Algorithmen in $O(n + m)$ liegen.

7.2 Stringmatching mit endlichen Automaten

Entdeckt der primitive Stringmatching-Algorithmus aus Listing 7.1 einen Mismatch an Position i , so fährt er mit der Suche an Position $i + 1$ fort. Passt jedoch der Teil des Musters, der sich vor dem Mismatch befand, zu einem Anfangsteil des Musters, so könnte man – verglichen mit der Funktionsweise des primitiven Stringmatching-Algorithmus – Vergleiche sparen. Betrachten wir als Beispiel die folgende in Abbildung 7.1 dargestellte Situation. Hier wäre es ineffizient nach diesem Mismatch an Position 1 von T weiterzusuchen, denn offensichtlich stellen die zuletzt gelesenen Zeichen **kak** ein Präfix, d. h. ein Anfangsstück, eines Matches dar.

Man kann einfach einen deterministischen endlichen Automaten konstruieren, der die zuletzt gelesenen Zeichen als Präfix des nächsten Matches deuten kann. Während es bei einem nichtdeterministischen endlichen Automaten für ein gelesenes Eingabezeichen

eventuell mehrere (oder auch gar keine) Möglichkeiten geben kann, einen Folgezustand auszuwählen, muss bei einem deterministischen endlichen Automaten immer eindeutig klar sein, welcher Zustand als Nächstes zu wählen ist, d. h. jeder Zustand muss für jedes Zeichen des „Alphabets“ (das je nach Situation $\{0, 1\}$, die Buchstaben des deutschen Alphabetes, oder jede andere endlichen Menge von Symbolen sein kann) genau eine Ausgangskante besitzen. Dies trifft auch auf den in Abbildung 7.2 dargestellten deterministischen endlichen Automaten zu, der effizient alle Vorkommen von **kakaokaki** in einem Text T erkennt. Der Automat startet in Zustand „1“; dies ist durch die aus dem „Nichts“ kommende Eingangskante angedeutet. Basierend auf den aus T gelesenen Zeichen verändert der Automat gemäß den durch die Pfeile beschriebenen Zustandsübergangsregeln seinen Zustand. Immer dann, wenn er sich im Endzustand (darstellt durch den Kreis mit doppelter Linie) befindet, ist ein Vorkommen von **kakaokaki** in T erkannt. Eine Kantenmarkierung von beispielsweise „[\sim ok]“ bedeutet – in Anlehnung an reguläre Ausdrücke – dass der entsprechende Übergang bei allen Eingabezeichen außer „o“ und „k“ gewählt wird.

Wie wird ein solcher Automat konstruiert? Um dies besser nachvollziehen zu können be-

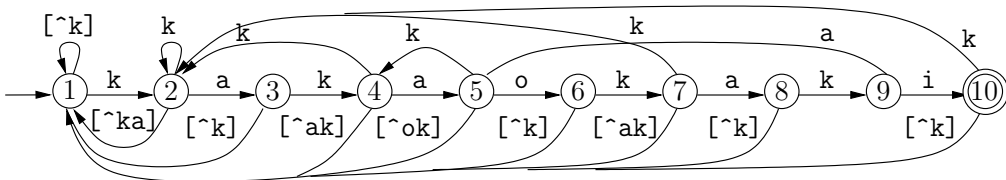


Abb. 7.2: Endlicher Automat, der ein effizientes Erkennen aller Vorkommen des Wortes **kakaokaki** erlaubt.

trachteten wir beispielsweise die Ausgangskanten des Zustands „5“: Die Ausgangskante mit Markierung „o“ gehört zum sog. *Skelettautomaten*, dessen Kantenbeschriftungen von links nach rechts gelesen genau dem zum matchenden Wort **kakaokaki** entsprechen. Wird im Zustand „5“ das Zeichen „k“ gelesen, so *muss* in den Zustand „4“ gesprungen werden – und nicht etwa in Zustand „2“ oder gar Zustand „1“, denn: Befindet sich obiger Automat in Zustand „5“ heißt dies immer, dass das zuletzt gelesene Zeichen ein „a“ und das vorletzte Zeichen ein „k“ war; diese beiden Zeichen könnten ein Anfangsteil des zu suchenden Wortes **kakaokaki** darstellen und dies wird dadurch berücksichtigt, indem der Automat nach Lesen von „k“, als Nächstes in Zustand „4“ springt.

Man kann die Funktionsweise eines endlichen Automaten direkt in einem Programm umsetzen:

```

1 def dfa(T):
2     zustand = 1
3     for t in T:
4         if zustand == 1:
5             if t != "k": zustand = 1
6             if t == "k": zustand = 2
7         if zustand == 2:
```

```

8      if t == "k": zustand = 2
9      elif t == "a": zustand = 3
10     else: zustand = 1
11     ...

```

Listing 7.2: Ein Teil der Implementierung des endlichen Automaten aus Abbildung 7.2.

Aufgabe 7.2

Vervollständigen Sie die in Listing 7.2 gezeigte Implementierung des endlichen Automaten aus Abbildung 7.2.

Aufgabe 7.3

Sie wollen alle Vorkommen des Strings **ananas** in einem Text suchen:

- Erstellen Sie den passenden endlichen Automaten, der immer dann in einem Endzustand ist, wenn er ein Vorkommen des Strings gefunden hat.
- Erstellen Sie eine entsprechendes Python-Skript, das die Funktionsweise dieses endlichen Automaten implementiert.

Die Laufzeit setzt sich zusammen aus der Konstruktion des deterministischen endlichen Automaten und dem anschließenden Durchlauf des Automaten bei der Eingabe des Textes T . Dieser Durchlauf benötigt offensichtlich $O(n)$ Schritte, denn genau daraufhin wurde der Automat ja konstruiert: Bei jedem Eingabezeichen führt der Automat einen wohl-definierten Zustandsübergang durch. Um den Automaten effizient zu konstruieren, ist jedoch ein raffinierter Algorithmus notwendig. Wir gehen jedoch nicht näher darauf ein, da der im folgenden Abschnitt beschriebene Algorithmus zwar dasselbe Prinzip verwendet, jedoch auf die Konstruktion des Automaten verzichten kann.

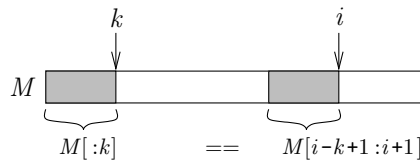
7.3 Der Knuth-Morris-Pratt-Algorithmus

Der Knuth-Morris-Pratt-Algorithmus verfolgt prinzipiell die gleiche Idee, wie sie bei der Konstruktion eines deterministischen endlichen Automaten zum Tragen kommt; nur vermeidet er, die aufwändige Konstruktion eines kompletten deterministischen endlichen Automaten und beschränkt sich auf das Wesentliche: die Suche nach Präfixen des Musters innerhalb des Musters selbst. Ein solches Präfix liegt innerhalb des Musters beispielsweise dann vor, wenn sich der deterministische Automat aus Abbildung 7.2 in Zustand „5“ befindet – dann wurden als letzte Zeichen nämlich „ka“ gelesen, was ein Präfix von „kakaokaki“ ist. Immer dann, wenn sich innerhalb des Musters ein Präfix des Musters befindet, kann um mehr als eine Position weitergeschoben werden; dies ist etwa in der in Abbildung 7.1 dargestellten Situation der Fall. Die Information, um wie

viele Positionen das Muster bei einem Mismatch weitergeschoben werden kann, wird in der sog. Verschiebetabelle P festgehalten, die wie folgt definiert ist:

$$P[i] := \max([k \text{ for } k \text{ in range(len(M)) if } M[:k] == M[i-k+1:i+1]] + [0]) \quad (7.1)$$

An der Stelle i der Verschiebetabelle ist also die Länge des (maximalen) Präfixes gespeichert, der sich *vor* Position i befindet. Die folgende Abbildung verdeutlicht dies:



Aufgabe 7.4

Schreiben Sie auf Basis der (bereits Python-artig formulierten) Formel (7.1) eine Python-Funktion, die die Verschiebetabelle eines als Parameter übergebenen Musters berechnet.

Als Beispiel betrachten wir die Verschiebetabelle für das Muster $M = \text{kakaokaki}$:

i	: 0	1	2	3	4	5	6	7	8
$P[i]$: 0	0	1	2	0	1	2	3	0
$M[i]$: k	a	k	a	o	k	a	k	i

Der Eintrag $P[7]$ ist beispielsweise deshalb „3“, weil die drei Zeichen vor der Position 7 (nämlich 'kak' ein Präfix des Musters sind; zwar ist auch das eine Zeichen (nämlich 'k') an Position 7 ein Präfix des Musters, Formel (7.1) stellt durch die Maximumsbildung jedoch sicher, dass immer das längste Teilwort vor Position i gewählt wird, das ein Präfix des Musters ist.

Aufgabe 7.5

Erstellen Sie die Verschiebetabelle für die folgenden Wörter:

- (a) ananas
- (b) 010011001001111
- (c) ababcabab

7.3.1 Suche mit Hilfe der Verschiebetabelle

Abbildung 7.3 zeigt Situationen in einem Lauf des Knuth-Morris-Pratt-Algorithmus, in denen das Muster auf Basis der in der Verschiebetabelle enthaltenen Werte weitergescho-

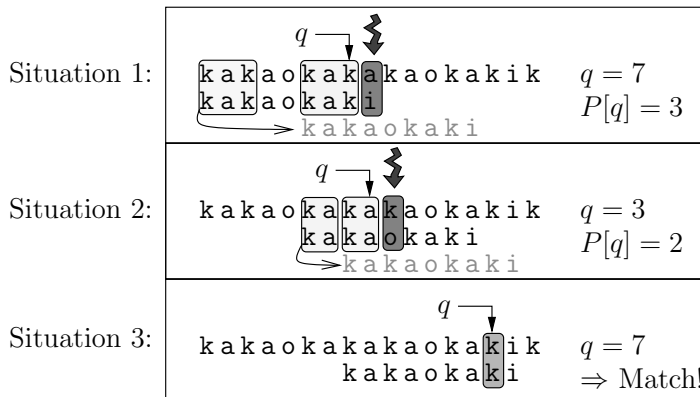


Abb. 7.3: Drei ausgewählte Schritte bei der Suche nach einem Vorkommen von *kakaokaki* mit dem Knuth-Morris-Pratt-Algorithmus. Die Zeichen des Musters werden mit den Zeichen des Textes verglichen. Tritt schließlich ein Mismatch auf (d. h. stimmt ein Zeichen des Musters nicht mit dem entsprechenden Zeichen des Textes überein), so wird das Muster weitergeschoben. Um wie viele Stellen das Muster weitergeschoben werden kann, ist in der Verschiebetabelle P hinterlegt.

ben wird. Das sind immer Situationen, in denen die jeweilige Stelle von Text und Muster nicht übereinstimmen (d. h. Situationen, in denen die Bedingung der **while**-Schleife in Listing 7.3 erfüllt ist). Sei q immer die Position im Muster, die zuletzt erfolgreich auf Gleichheit mit dem Text überprüft wurde. Betrachten wir die drei in Abbildung 7.3 dargestellten Situationen im Detail:

Situation 1: Muster M und Text T stimmen bisher bis zur Stelle $q = 7$ überein. Beim Vergleich des nächsten Zeichens von M mit der nächsten Textposition tritt ein Mismatch auf. Aus der Verschiebetabelle P geht nun hervor, dass die $P[q] = 3$ letzten Zeichen *vor* dem Mismatch ein Präfix (genauer: das maximal lange Präfix) des Musters darstellen – diese drei Zeichen und auch das darauf passende Präfix des Musters sind in Abbildung 7.3 in einem hell gefüllten Rechteck dargestellt. Um mit der Suche fortzufahren, wird nun die Variable q auf „3“ gesetzt, was einer Verschiebung des Musters entspricht, wie sie unten in Situation 1 in hell gedruckter Schrift dargestellt ist.

Situation 2: Muster M und Text T stimmen bisher bis zur Stelle $q = 3$ überein. Beim Vergleich des nächsten Zeichens von M mit der nächsten Textposition tritt ein Mismatch auf. Aus der Verschiebetabelle P geht nun hervor, dass die $P[q] = 2$ letzten Zeichen *vor* dem Mismatch ein Präfix des Musters darstellen – diese zwei Zeichen und auch das darauf passende Präfix des Musters sind in Abbildung 7.3 in einem gelben Rechteck dargestellt. Um mit der Suche fortzufahren, wird die Variable q auf „2“ gesetzt, was einer Verschiebung des Musters entspricht, wie sie unten in Situation 2 in hell gedruckter Schrift dargestellt ist.

Situation 3: Muster M und Text T stimmen bisher bis zur Stelle $q = 3$ überein. Da sich auch beim Vergleich von $M[-1]$ mit der entsprechenden Stelle des Textes T Gleichheit ergab, wird ein Match zurückgeliefert.

Listing 7.3 zeigt eine Implementierung des Knuth-Morris-Pratt-Algorithmus.

```

1 def KMP( $M, T$ ):
2      $P = \dots$  # Berechnung der Verschiebetabelle
3      $erg = []$ 
4      $q = -1$ 
5     for  $i$  in  $range(len(T))$ :
6         while  $q \geq 0$  and  $M[q+1] \neq T[i]$ :  $q = P[q]$ 
7          $q += 1$ 
8         if  $q == len(M) - 1$ :
9              $erg.append(i+1 - len(M))$ 
10             $q = P[q]$ 
11 return  $erg$ 

```

Listing 7.3: Implementierung des Knuth-Morris-Pratt-Algorithmus

In Zeile 2 wird die Verschiebetabelle P berechnet; einen schnellen Algorithmus hierfür beschreiben wir im nächsten Abschnitt. Wie auch im Beispiel aus Abbildung 7.3 gehen wir davon aus, dass q immer die Position im Muster M enthält, die zuletzt erfolgreich auf Gleichheit mit der entsprechenden Textposition geprüft wurde; zu Beginn setzen wir in Zeile 3 also q auf den Wert -1 – es wurde ja noch keine Position des Musters erfolgreich auf Gleichheit getestet. Die **for**-Schleife ab Zeile 4 durchläuft alle Positionen des Textes T . Immer dann, wenn die aktuelle Position im Text, also $T[i]$ mit der aktuell zu vergleichenden Position im Muster, also $M[q+1]$, übereinstimmt, wird q um eins erhöht und die **for**-Schleife geht in den nächsten Durchlauf und es wird mit der nächsten Textposition verglichen. Wenn jedoch $M[q+1]$ *nicht* mit $T[i]$ übereinstimmt, so wird q auf den entsprechenden in der Verschiebetabelle eingetragenen Wert erniedrigt; dies kann durchaus wiederholt geschehen, solange bis Muster und Text in der nachfolgenden Position übereinstimmen.

Aufgabe 7.6

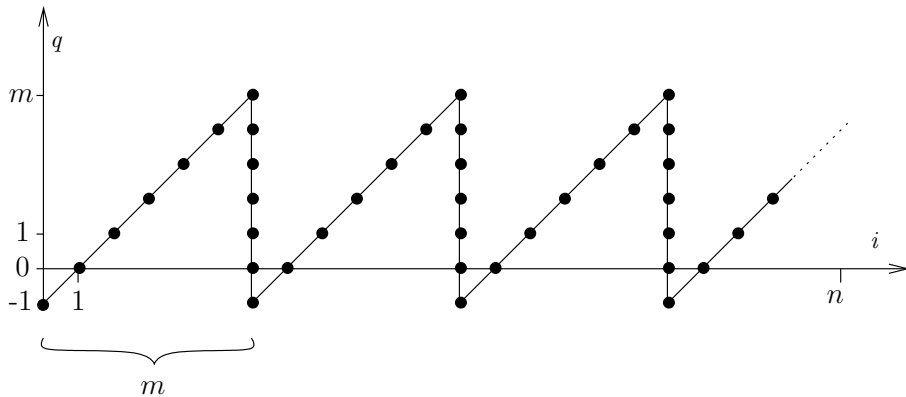
Verwenden Sie Pythons *timeit*-Modul, um die Laufzeit der in Listing 7.1 gezeigten primitiven Implementierung mit der Knuth-Morris-Pratt-Algorithmus an einigen praktischen Beispielen zu vergleichen. Was fällt auf?

7.3.2 Laufzeit

Wir stellen zunächst fest, dass in einem Durchlauf (der insgesamt $n = len(T)$ Durchläufe) der **for**-Schleife, die **while**-Schleife schlimmstenfalls $m = len(M)$ -mal durchlaufen wird, q also schlimmstenfalls in Einerschritten bis -1 erniedrigt wird. Die Gesamtkomplexität

des Algorithmus ist jedoch nicht in $\Theta(n \cdot m)^1$; dies zeigt folgende einfache Amortisationsanalyse.

Die Variable q kann nicht bei jedem Durchlauf der **for**-Schleife um m Werte erniedrigt werden. Die Bedingung der **while**-Schleife stellt sicher, dass q immer nur bis zum Wert -1 erniedrigt werden kann. Um es daraufhin erneut zu erniedrigen, muss es zunächst erhöht worden sein. Jede Erhöhung von q kann aber nur mit einer nachfolgenden Erhöhung von i einhergehen. Ein schlimmster denkbarer Fall wäre also der, dass q immer in Einerschritten erniedrigt und danach (zusammen mit i) wieder erhöht wird. Der Verlauf von i (auf der x-Achse) und q (auf der y-Achse) zeigt die folgende Abbildung:



Man erkennt, dass insgesamt n Schritte nach „oben“ (verursacht durch eine gemeinsame Erhöhung von q und i außerhalb der **while**-Schleife) und n Schritte nach unten (verursacht durch eine Erniedrigung von q innerhalb der **while**-Schleife) gegangen werden. Der Algorithmus hat also eine worst-case-Komplexität von $O(2 \cdot n) = O(n)$.

7.3.3 Berechnung der Verschiebetabelle

Die Berechnung der Verschiebetabelle erfolgt analog zur Knuth-Morris-Pratt-Suche, nur dass hier das Muster nicht in einem Text, sondern im Muster selbst gesucht wird. Listing 7.4 zeigt eine Implementierung.

```

1 def VerschTab(M):
2   q = -1 ; P = [q]
3   for i in range(1, len(M)):
4     while q ≥ 0 and M[q] ≠ M[i]: q = P[q]
5     q += 1
6     P.append(q)
7   return P

```

Listing 7.4: Implementierung der Berechnung der Verschiebetabelle.

¹Während man mit dem Landau-Symbol O eine obere Schranke beschreibt, kann man mit dem Landau-Symbol Θ die – bis auf multiplikative und additive Konstanten – exakte Laufzeit eines Algorithmus beschreiben; zur Definition der Landau-Symbole siehe Abschnitt 1.1.1

Die Variable i durchläuft alle Positionen des Musters M ; Die Variable q zeigt immer auf das Ende des längsten Präfixes, das mit den Zeichen vor der Position i im Muster M übereinstimmt. Unmittelbar nach Zeile 5 gilt immer, dass alle Positionen vor q mit den q Positionen vor i übereinstimmen, d. h. es gilt $M[:q] == M[i-q+1:i+1]$, d. h. die Zeichen vor Position i bilden ein Präfix der Länge q des Musters. Ein entsprechender Eintrag in die Verschiebetabelle erfolgt in Zeile 6.

Die Laufzeitbetrachtung ist analog zur Suche und beträgt Worst-Case $O(2 \cdot m) = O(m)$.

7.4 Der Boyer-Moore-Algorithmus

Der Boyer-Moore-Algorithmus wurde einige Jahre nach dem Knuth-Morris-Pratt-Algorithmus entdeckt [3]. Er lässt das Muster von links nach rechts über den Text laufen und versucht das Muster bei einem Mismatch um möglichst viele Positionen weiterzuschieben. Er nutzt jedoch die Tatsache aus, dass man mehr Informationen über Verschiebemöglichkeiten erhalten kann, wenn man die Musterpositionen von rechts nach links mit den aktuellen Textpositionen vergleicht, d. h. Nach einer Verschiebung des Musters M wird zuerst das Zeichen $M[-1]$ mit der entsprechenden Textposition verglichen, dann das Zeichen $M[-2]$, usw. Durch dieses Rückwärtsvergleichen ist der Boyer-Moore-Algorithmus – zumindest was die Average-Case-Komplexität betrifft – effizienter als der im letzten Abschnitt vorgestellte Knuth-Morris-Pratt-Algorithmus.

Um nach einem Mismatch zu entscheiden, um wie viele Positionen das Muster weitergeschoben werden kann, verwendet der Algorithmus zwei Tabellen: die erste Tabelle liefert einen Vorschlag gemäß der sog. Bad-Character-Heuristik, die zweite Tabelle liefert einen Vorschlag gemäß der sog. Good-Suffix-Heuristik. Beide Tabellen können unter Umständen verschiedene Vorschläge darüber abgeben, wie weit das Muster geschoben werden kann; der Boyer-Moore-Algorithmus schiebt das Muster um den größeren der beiden vorgeschlagenen Werte weiter.

7.4.1 Die Bad-Character-Heuristik

Am einfachsten zu konstruieren ist die Sprungtabelle *delta1* gemäß der sog. Bad-Character-Heuristik; diese basiert alleine auf dem Zeichen c des zu durchsuchenden Textes T , das den Mismatch verursacht hat, d. h. auf dem ersten Zeichen von rechts gesehen, das nicht mit der entsprechenden Stelle im Muster übereinstimmt. Kommt c überhaupt nicht im Muster vor, so kann das Muster an die Stelle nach dem Mismatch weitergeschoben werden. Dies tritt etwa in der in Abbildung 7.4 gezeigten „Situation 2“ ein, die eine Stringsuche ausschließlich basierend auf der Bad-Character-Heuristik zeigt. Kommt das Zeichen c , das den Mismatch verursacht, im Muster vor, so wird das Muster so weit nach rechts verschoben, dass das von rechts gesehen erste Vorkommen von c im Muster mit dem Mismatch-verursachenden Zeichen c im Text gleichauf liegt. Es kann vorkommen, dass die Bad-Charakter-Heuristik eine Linksverschiebung des Musters vorschlägt – dies wäre etwa in „Schritt 5“ der Fall: das von rechts gesehen erste Vorkommen von „a“ im Muster befindet sich hier rechts des Zeichens „a“ im Text, das den Mismatch ausgelöst hat; in diesem Fall wird das Muster einfach um eine Position weitergerückt.

Schritt 1:	kakaok o kikxkaokako-ist-kakaakakis kaka o kaki
Schritt 2:	kakaokokik x kaokako-ist-kakaakakis kakaokaki
Schritt 3:	kakaokokikxkaokako - ist-kakaakakis kakaokaki
Schritt 4:	kakaokokikxkaokako-ist-kaka o aakis kaka o kaki
Schritt 5:	kakaokokikxkaokako-ist-kakao a akis kakaok a ki
Schritt 6:	kakaokokikxkaokako-ist-kakaoa k is kakaokaki

Abb. 7.4: Es sind die sechs Suchschritte dargestellt, die notwendig sind, um das Muster *kakaokaki* in einem bestimmten Text alleine mit Hilfe der Bad-Charakter-Heuristik zu suchen. Für jeden Schritt ist jeweils immer der Text oben und das Muster unter dem Teil des Textes dargestellt, der auf Gleichheit mit dem Muster überprüft wird. Das Zeichen, das den Mismatch verursacht und das dazu passende Zeichen im Muster ist jeweils farbig hinterlegt.

Aufgabe 7.7

Angenommen, wir suchen nach einem Muster M der Länge m in einem Text T der Länge n und angenommen alle mit $M[-1]$ verglichenen Zeichen kommen nicht im Muster vor – mit zunehmender Größe des verwendeten Alphabets wird dieser Fall natürlich wahrscheinlicher. Wie viele Suchschritte benötigt der Boyer-Moore-Algorithmus, bis er festgestellt hat, dass das Muster nicht im Text vorkommt?

Aufgabe 7.8

Es stehe a^n für die n -malige Wiederholung des Zeichens „a“. Wie viele Suchschritte benötigt der Boyer-Moore-Algorithmus um ...

- ... das Muster ba^9 (also das Muster `baaaaaaaaa`) im Text a^{1000} (also einem Text bestehend aus 1000 as) zu finden?
- ... das Muster a^9b (also das Muster `aaaaaaaaab`) im Text a^{1000} zu finden?
- ... das Muster a^9b (also das Muster `aaaaaaaaab`) im Text c^{1000} zu finden?

Folgendes Listing zeigt die Implementierung der Bad-Character-Heuristik.

```

1 def makedelta1(M):
2     delta1 = {}
3     for i in range(len(M)-1):
4         delta1[M[i]] = i
5     return delta1
6
7 def badChar(delta1,c,j):
8     if c in delta1:
9         return j - delta1[c]
10    else:
11        return j+1

```

Listing 7.5: Berechnung der Sprungtabelle gemäß der Bad-Character-Heuristik

Die Funktion *makedelta1* erstellt für ein bestimmtes Muster *M* einmalig eine Sprungtabelle *delta1*, die sie als Dictionary-Objekt repräsentiert zurückliefert. Die **for**-Schleife ab Zeile 3 durchläuft alle Positionen *i* des Musters und erstellt in der Sprungtabelle für das *i*-te Zeichen *M*[*i*] des Musters einen Eintrag mit Wert *i*. Weiter rechts auftretende Vorkommen dieses Zeichens überschreiben diesen Eintrag und so enthält nach Ende der **for**-Schleife der Eintrag *delta1*[*c*] automatisch die von rechts gesehen erste Position eines Vorkommens von *c* im Muster. Der Wert dieser Position ist entscheidend zur Bestimmung der Verschiebepositionen des Musters.

Die Funktion *badChar* kann nun basierend auf der Verschiebetabelle *delta1*, dem „Bad Character“ *c* und der Position *j* des Mismatches im Muster die Anzahl der Positionen bestimmen, die das Muster weitergeschoben werden darf. Gibt es einen Eintrag *c* in *delta1*, d. h. kommt *c* im Muster vor, so kann das Muster um *j*-*delta1*[*c*] Positionen nach rechts verschoben werden. Dadurch deckt sich das am weitesten rechts befindliche Vorkommen von *c* im Muster mit dem Mismatch des Textes. Für den Fall, dass dieser Verschiebewert negativ ist (wie dies etwa in „Situation 5“ aus Abbildung 7.4 der Fall ist), wird einfach „1“ zurückgegeben. Sollte *delta1* keinen Eintrag für das Zeichen *c* enthalten, gilt also *c* **not in** *delta1*, so wird der **else**-Zweig ab Zeile 10 gegangen und der Wert *j*+1 zurückgeliefert. Das Muster kann in diesem Fall also an die auf den Mismatch folgende Stelle weitergeschoben werden.

Tabelle 7.1 zeigt die Rückgabewerte von *delta1* und der Funktion *badChar* für die in Abbildung 7.4 dargestellten Beispielsituationen. Wie man sieht, entspricht der Rückgabewert der Funktion *badChar* genau den Verschiebepositionen des Musters in der jeweiligen Situation.

Situation 1	Situation 2	Situation 3
$\text{delta1}['\text{o}']=4$ $\text{badChar}(d1, '\text{o}', 6)=2$	$\text{delta1}['\text{x}']=\text{KeyError}$ $\text{badChar}(d1, '\text{x}', 8)=9$	$\text{delta1}['-']=\text{KeyError}$ $\text{badChar}(d1, '-', 7)=8$
Situation 4	Situation 5	Situation 6
$\text{delta1}['\text{o}']=4$ $\text{badChar}(d1, '\text{o}', 8)=4$	$\text{delta1}['\text{a}']=6$ $\text{badChar}(d1, '\text{a}', 5)=\max(-1, 1)$	$\text{delta1}['\text{s}']=\text{KeyError}$ $\text{badChar}(d1, '\text{s}', 8)=9$

Tabelle 7.1: Rückgabewerte der in Listing 7.5 gezeigten Funktionen für die Beispielsituationen aus Abbildung 7.4.

Aufgabe 7.9

- Geben Sie eine alternative Implementierung der in Listing 7.5 gezeigten Funktion *makedelta1* an, die für jedes Zeichen des verwendeten Alphabets einen passenden Eintrag enthält und so eine entsprechende Abfrage in der Funktion *badChar* vermeidet.
- Testen sie die Performance der beiden Implementierungen aus den ersten beiden Teilaufgaben zusammen mit der in Listing 7.5 gezeigten Implementierung. Welche Variante ist die schnellste? Warum?

7.4.2 Die Good-Suffix-Heuristik

Die etwas komplexer zu konstruierende zweite Tabelle gibt Verschiebevorschläge gemäß der sog. Good-Suffix-Heuristik. Während die Bad-Character-Heuristik das Zeichen *c*, das den Mismatch verursacht, in Betracht zieht, zieht die Good-Suffix-Heuristik den übereinstimmenden Teil von Muster und Text rechts des Zeichens *c* in Betracht – den „hinteren“ Teil des Musters also, sprich: das Suffix. Die Good-Suffix-Heuristik schlägt eine Verschiebung des Musters so vor, so dass ein weiter links stehender mit diesem „Good-Suffix“ übereinstimmender Teil des Musters auf dieser Textstelle liegt. Abbildung 7.5 zeigt als Beispiel das Muster „entbenennen“ und einige Mismatch-Situationen. Wie man sieht, wird nach jedem Mismatch das Muster so verschoben, dass ein weiter links stehender Teil des Musters, auf dem „Good-Suffix“ (d. h. den Suffix des Musters, der mit dem Text übereinstimmt) liegt.

Schauen wir uns nun etwas systematischer an, wie die Verschiebetabelle für das Beispielmuster $M = \text{'entbenennen'}$ erstellt wird. Wir bezeichnen hierfür mit j die Länge des mit dem Text übereinstimmenden Suffixes des Wortes **entbenennen**; $j = 0$ bedeutet also, dass schon das von rechts gesehen erste Zeichen des Musters nicht mit dem Text übereinstimmt; $j = \text{len}(M) - 1$ bedeutet, dass alle Zeichen des Musters mit dem Text übereinstimmen, d. h. ein Match gefunden wurde. Das von rechts gesehen erste nicht mehr matchende Zeichen des Suffixes stellen wir durchgestrichen dar. Den im Muster weiter links befindlichen Teil, der mit dem Suffix – inklusive der Mismatch-Stelle – übereinstimmt, stellen wir unterstrichen dar. Wir stellen uns ferner virtuelle Musterpositionen vor dem ersten Eintrag $M[0]$ des Musters vor, die wir mit „.“ notieren; wir

Situation 1	nen-klei <u>nen</u> -fehler-sehen-im-nentbenennen entben <u>ennen</u>
Situation 2	nen-kleinen-fehler-sehen-im-nentbenennen entben <u>ennen</u>
Situation 3	nen-kleinen-fehler-sehen-im-nentbenennen entben <u>ennen</u>
⋮	⋮
Situation 12	nen-kleinen-fehler-seh <u>en</u> -im-nentbenennen entben <u>ennen</u>
Situation 13	nen-kleinen-fehler-sehen-im- <u>n</u> entbenennen entben <u>ennen</u>
⋮	⋮

Abb. 7.5: Beispiele für Mismatch-Situationen und entsprechende Verschiebungen gemäß der Good-Suffix-Heuristik.

nehmen an, dass das Zeichen „.“ mit jedem beliebigen Zeichen (auch mit einem durchgestrichenen) matcht; diese virtuellen Musterpositionen werden etwa in Fällen $i \geq 4$ mit einbezogen.

- $j = 0$: Das matchende Suffix ist also n. Der am weitesten rechts befindliche Teilstring von entbenennen, der auf n passt, ist das Zeichen „e“ an Stringposition 9. Durch Verschiebung des Musters um eine Position kann dieses Zeichen mit n in Deckung gebracht werden. Daher schlägt die Good-Suffix-Strategie hier eine Verschiebung um eine Position vor.
- $j = 1$: Das matchende Suffix ist also en. Der am weitesten rechts befindliche passende Teilstring ist entbenennen. Durch eine Verschiebung um 2 Positionen kann dieser mit dem matchenden Suffix in Deckung gebracht werden.
- $j = 2$: Das matchende Suffix ist also nen. Der am weitesten rechts befindliche passende Teilstring ist entbenennen. Durch eine Verschiebung um 5 Positionen kann dieser mit dem Suffix in Deckung gebracht werden.
- $j = 3$: Das matchende Suffix ist also nnen. Der passende Teilstring ist entbenennen. Durch eine Verschiebung um 3 Positionen kann dieser mit dem matchenden Suffix in Deckung gebracht werden.
- $j = 4$: Das matchende Suffix ist also ennen. Eigentlich gibt es keinen passenden Teilstring; durch oben beschriebene Expansion des Musters kann man sich den „passenden“ Teilstring jedoch denken als \dots entbennenen. Um den „passenden“ Teil \dots en mit dem matchenden Suffix in Deckung zu bringen, muss das Muster um 9 Positionen nach rechts verschoben werden.

$j = 5$: Das matchende Suffix ist also **nennen**. Genau wie im Fall $j = 4$ ist auch hier der passende Teilstring ...entbennenen; entsprechend wird auch hier eine Verschiebung um 9 vorgeschlagen.

$j = 6, j = 7, j = 8, j = 9$: Mit analoger Argumentation wird auch hier jeweils eine Verschiebung um 9 vorgeschlagen.

Die in Listing 7.6 gezeigte Funktion *makedelta2* implementiert die Berechnung der Verschiebetabelle (die als Dictionary-Objekt *delta2* zurückgeliefert wird) gemäß der Good-Suffix-Heuristik. Im j -ten Durchlauf der **for**-Schleife ab Zeile 9 wird der Eintrag *delta2*[j] berechnet; dieser gibt die Verschiebung an, falls ein „Good-Suffix“ der Länge j erkannt wurde. Die Variable *suffix* enthält immer die Zeichen des „Good-Suffix“ und die Variable *mismatch* enthält das von rechts gesehen erste Zeichen, das nicht mehr gematcht werden konnte (oben immer durch ein durchgestrichenes Zeichen notiert). In der **for**-Schleife ab Zeile 12 werden dann alle Musterpositionen k von rechts nach links durchlaufen und mittels der *unify*-Funktion überprüft, ob der an Stelle k befindliche Teilstring des Musters zu dem „Good-Suffix“ passt. Falls ja, wird der passende Verschiebebetrag in *delta2*[j] gespeichert und die „**for** k “-Schleife mittels **break** verlassen – so ist sichergestellt, dass der am weitesten rechts befindliche Teilstring von M gefunden wird, der auf das Suffix passt. Immer dann, wenn zwischen der Position k und der Position 0 sich weniger als j Zeichen befinden, werden links von Position 0 entsprechend viele „DOT“-s angehängt; dies geschieht in Zeile 13.

```

1 DOT=None
2 def unify(pat,mismatch,suffix):
3     def eq(c1,c2): return c1==DOT or c1==c2
4     def not_eq(c1,c2): return c1==DOT or c1!=c2
5     return not_eq(pat[0],mismatch) and all(map(eq,pat[1:],suffix))
6
7 def makedelta2(M):
8     m = len(M) ; delta2 = {}
9     for j in range(0,m): # Suffix der Länge j
10        suffix = [] if j==0 else M[-j:]
11        mismatch = M[-j-1]
12        for k in range(m-1,0,-1):
13            pat = [DOT for i in range(-k+j)] + list(M[max(0,k-j):k+1])
14            if unify(pat,mismatch,suffix): # Good-Suffix im Muster gefunden!
15                delta2[j]=m-1-k ; break
16            if j not in delta2: delta2[j]=m
17 return delta2

```

Listing 7.6: Implementierung der Good-Suffix-Heuristik

Aufgabe 7.10

Beantworten Sie folgende Fragen zu Listing 7.6:

- Erklären Sie die Zuweisung in Zeile 10; was würde passieren, wenn diese einfach „*suffix* = *M*[-*j*:]“ heißen würde?
- Welchen Typ hat der Parameter *pat* im Aufruf der Funktion *unify* in Zeile 13? Welchen Typ hat der Parameter *suffix*?
- Es sei *M* = 'ANPANMAN'. Was sind die Werte von *suffix* und *mismatch* und in welchem Durchlauf bzw. welchen Durchläufen der „**for** *k*“-Schleife liefert dann der Aufruf von *unify* den Wert *True* zurück, wenn wir uns ...
 - ... im **for**-Schleifendurchlauf für *j*=1 befinden.
 - ... im **for**-Schleifendurchlauf für *j*=2 befinden.

Die Funktion *unify* prüft, ob der Teilstring *pat* des Musters (der ggf. links mit *DOT*s aufgefüllt ist) mit dem „Good-Suffix“ *suffix* und dem den Mismatch verursachenden Zeichen *mismatch* „vereinbar“ ist. Wichtig ist, dass die eigens definierten Gleichheits- und Ungleichheitstests *eq* bzw. *not_eq* bei einem Vergleich mit *DOT* immer *True* zurückliefern.

7.4.3 Implementierung

Listing 7.7 zeigt die Implementierung der Stringsuche mit Hilfe der Bad-Character-Heuristik *delta1* und der Good-Suffix-Heuristik *delta2*. Für jeden Durchlauf der **while**-Schleife ist *i* die Position im Text *T* und *j* die Position im Muster *M* die miteinander verglichen werden. Die Variable *i_old* enthält immer die Position im Text, die als erstes mit dem Muster verglichen wurde (d. h. die Position im Text, die über dem rechten Zeichen des Musters *M* liegt). Nach Durchlauf der **while**-Schleife in Zeile 7 zeigen *i* und *j* auf die von rechts gesehen erste Mismatch-Stelle von Text und Muster. Gibt es keine Mismatch-Stelle (gilt also *j*== -1 nach dem **while**-Schleifendurchlauf) wurde das Muster im Text gefunden. Andernfalls wird *i* in Zeile 13 um den durch die Bad-Character-Heuristik bzw. die Good-Suffix-Heuristik vorgeschlagenen Verschiebebetrag erhöht.

```

1 def boyerMoore(T,M):
2     delta1 = makedelta1(M)
3     delta2 = makedelta2(M)
4     m = len(M) ; n = len(T) ; i=m-1
5     while i < n:
6         i_old=i ; j=m-1
7         while j≥0 and T[i] == M[j]:
8             i -= 1 ; j -= 1
9         if j == -1:
10            print "Treffer: ", i+1

```

```

11     i = i_old + 1
12     else:
13         i = i_old + max(badChar(delta1, T[i], j), delta2[m-1-j])

```

Listing 7.7: Implementierung des Boyer-Moore-Algorithmus

Aufgabe 7.11

Modifizieren Sie die in Listing 7.7 vorgestellte Funktion *boyerMoore* so, dass sie die Liste aller Matches des Musters *M* im Text *T* zurückliefert.

Aufgabe 7.12

Gerade für den Fall, dass man mit einem bestimmten Muster komfortabel mehrere Suchen durchführen möchte, bietet sich eine objekt-orientierte Implementierung mittels einer Klasse *BoyerMoore* an, die man beispielsweise folgendermaßen anwenden kann:

```

>>> p = BoyerMoore('kakaokaki')
>>> p.search(T1)
...
>>> p.search(T2)

```

Implementieren Sie die Klasse *BoyerMoore*.

7.4.4 Laufzeit

Wie viele Suchschritte benötigt der Boyer-Moore-Algorithmus zum Finden aller Vorkommen des Musters *M* (mit $m = \text{len}(M)$) im Text *T* (mit $n = \text{len}(T)$)?

Im günstigsten Fall sind dies lediglich $O(n/m)$ Schritte – dann nämlich, wenn entweder „viele“ Zeichen des Textes gar nicht im Muster vorkommen oder wenn „viele“ Suffixe kein weiteres Vorkommen im Muster haben; in diesen Fällen wird eine Verschiebung um *m* Positionen vorgeschlagen.

Im Worst-Case benötigt der Boyer-Moore-Algorithmus etwa $3n$ Schritte; die mathematische Argumentation hierfür ist nicht ganz einfach und es brauchte auch immerhin bis ins Jahr 1991, bis diese gefunden wurde; wir führen diese hier nicht aus und verweisen den interessierten Leser auf die entsprechende Literatur [6]. Die Worst-Case-Laufzeit ist also in $O(n)$.

7.5 Der Rabin-Karp-Algorithmus

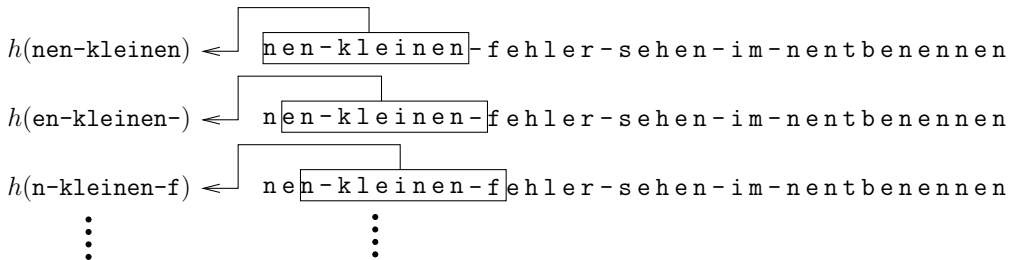
Der Rabin-Karp-Algorithmus geht einen ganz anderen Weg, um ein Muster in einem Text zu suchen: Er berechnet unter Verwendung einer Hashfunktion *h* den Hashwert

$h(M)$ des Musters M , und sucht nach Stellen im Text T , die denselben Hashwert aufweisen. Wird die Hashfunktion h geschickt gewählt, so ist mit diesem Algorithmus eine gute Laufzeit gesichert.

Der Rabin-Karp-Algorithmus ist zwar in vielen Fällen – was die Performance betrifft – dem Boyer-Moore-Algorithmus unterlegen, es gibt jedoch einige Fälle, in denen sich der Einsatz des Rabin-Karp-Algorithmus lohnt. Dies betrifft insbesondere die Suche sehr langer (evtl. auch mehrerer) Muster in einem Text. Denkbar wäre etwa der Einsatz in einer Software, die Dokumente automatisch nach Plagiaten überprüft, indem sie mehrere längere (Original-)Textausschnitte in dem zu überprüfenden Text sucht.

7.5.1 Rollender Hash

Ein rollender Hash ist eine Hashfunktion, die ihre Eingabe aus einem „Fenster“ konstanter Größe bezieht, das von links nach rechts über die Eingabe geschoben wird.



Zur Implementierung des Rabin-Karp-Algorithmus genügt die Verwendung einer sehr einfachen rollenden Hashfunktion h , die einen String s folgendermaßen abbildet:

$$h(s) = B^{k-1}s[0] + B^{k-2}s[1] + \dots + B^1s[k-2] + B^0s[k-1] \mod p \quad (7.2)$$

Um das aufwändige Rechnen mit sehr großen Zahlen zu vermeiden, rechnet die Hashfunktion mit modularer Arithmetik; entscheidend ist hier die Wahl der Basis B und die Wahl von p . Aus Performance-Gründen ist es sinnvoll eine Zweierpotenz als p , d. h. $p = 2^k$, zu wählen. Die modulare Arithmetik mit einer solchen Zweierpotenz 2^k entspricht nämlich einfach dem Abschneiden der binären Stellen ab Position k . Dies kommt der natürlichen Funktionsweise eines Rechners auf Ebene der Maschinensprache nahe: Entsteht bei einer arithmetischen Berechnung ein Überlauf, so werden die höherwertigen Stellen einfach abgeschnitten. In Python können wir dieses Abschneiden der höherwertigen Stellen durch eine binäre Und-Verknüpfung mit der Zahl $2^k - 1$ erreichen. Es gilt also:

$$x \mod 2^k = x \& \underbrace{11\dots 1}_k b = x \& (2^k - 1)$$

Wir wählen also im Folgenden $p = M = 2^{k-1}$ und ein $k \in \mathbb{N}$. Konkret könnten B und M etwa wie folgt gewählt werden:

$$B = 103$$

$$M = 2^{**16} - 1$$

Aufgabe 7.13

Wäre auch die Konstante *sys.maxint* (aus dem Modul *sys*) ein sinnvoller Wert für *M*? Begründen Sie.

Listing 7.8 zeigt eine primitive Implementierung dieser Hashfunktion. Die **for**-Schleife durchläuft den zu hashenden String *s* rückwärts; die Variable *i* enthält hierbei immer den von rechts gezählten Index, der als Potenz der Basis *B* verwendet wird. In jedem **while**-Schleifendurchlauf wird durch die Zuweisung *h* = *h* & *M* sichergestellt, dass nur die *k* niederwertigsten Bits weiter verwendet werden (um das aufwändige Rechnen mit sehr großen Zahlen zu vermeiden).

Durch Verwendung des sog. *Horner-Schemas* (siehe auch Abschnitt 3.4.1 auf Seite 74) kann die Berechnung dieses Hashwertes deutlich schneller erfolgen. Anstatt Formel 7.2 direkt zu implementieren ist es günstiger, die folgende Form zu verwenden, in der die *B*-Werte soweit als möglich ausgeklammert sind:

$$h(s) = (((s[0] \cdot B + s[1]) \cdot B + \dots) \cdot B + s[k-2]) \cdot B + s[k-1] \mod p \quad (7.3)$$

Listing 7.9 zeigt die Implementierung des Horner-Schemas mittels der *reduce*-Funktion.

```

1 def rollhash(s):
2     h = 0
3     for i, c in enumerate(s[::-1]):
4         h += (B**i) * ord(c)
5         h = h & M
6     return h

```

Listing 7.8: Primitive Berechnung der Hashfunktion

```

1 def rollhash2(s):
2     return reduce(
3         lambda h, c: (c + B*h) & M,
4         map(ord, s))

```

Listing 7.9: Berechnung der Hashfunktion mittels des Horner-Schemas

Aufgabe 7.14

Verwenden sie Pythons *timeit*-Modul, um die Laufzeiten der in Listing 7.8 und 7.9 gezeigten Funktionen *rollhash* und *rollhash2* zu vergleichen. Vergleichen Sie die Werte der *timeit*-Funktion für einen String *S* mit Länge 10, Länge 20 und Länge 50.

Angenommen in einem langen Suchtext *T* ist momentan der Hash *h* eines „Fensters“ an Position *i* der Länge *l* berechnet, d. h. es gilt *h* = *h*(*s*[*i* : *i* + *l*]). Will man nun dieses „Fenster“ dessen Hash *h* berechnet werden soll nach rechts bewegen, so erhält man den entsprechenden neuen Hashwert durch Subtraktion des Wertes *b*^{*l*−1}*s*[*i*], einer nachfolgenden Multiplikation dieses Wertes mit der Basis *b* und einer Addition mit *s*[*i* + *l*]; alle Rechnungen erfolgen mit modularer Arithmetik; der Wert *h* muss also folgendermaßen angepasst werden:

$$h = (h - B^{l-1}s[i]) \cdot B + s[i + l] \quad (7.4)$$

Will man dagegen dieses „Fenster“ dessen Hash h berechnet werden soll nach links bewegen, so erhält man den entsprechenden neuen Hashwert durch Subtraktion des Wertes $b^0 s[i + l - 1]$, einer nachfolgenden Division durch die Basis b (d. h. einer Multiplikation mit b^{-1}) und einer abschließenden Addition mit $b^{l-1} s[i - 1]$; der Wert h muss also folgendermaßen angepasst werden:

$$h = (h - B^0 s[i + l - 1]) \cdot B^{-1} + B^{l-1} s[i - 1] \quad (7.5)$$

7.5.2 Implementierung

Listing 7.10 zeigt eine Implementierung des Rabin-Karp-Algorithmus in Form der Funktion *rabinKarp*. Diese erhält zwei Parameter: eine Liste von Mustern Ms , und ein Text T , der nach Vorkommen der Muster durchsucht werden soll. Wir gehen hier davon aus, dass alle in Ms befindlichen Mustern die gleiche Länge haben.

```

1 def rabinKarp(Ms, T):
2     hashes = set(map(rollhash, Ms))
3     l = len(Ms[0])
4     h = rollhash(T[:l])
5     i = 0
6     if h in hashes:
7         if T[i:i+l] in Ms: print "Treffer bei", i
8     while i + l < len(T) - 1:
9         h = (h - ord(T[i]) * B**(l-1)) * B + ord(T[i+l]) & M
10        i += 1
11        if h in hashes:
12            if T[i:i+l] in Ms: print "Treffer bei", i

```

Listing 7.10: Implementierung des Rabin-Karp-Algorithmus

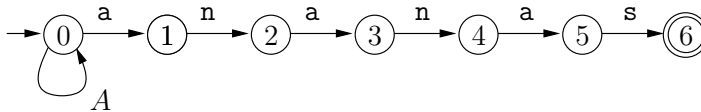
In Zeile 2 wird mittels der *map*-Funktion der Hashwert jedes in Ms gespeicherten Musters berechnet und in einer Menge *hashes* gespeichert; die Verwendung eines *set*-Objektes macht hier insbesondere aus Performance-Gründen Sinn, da so unter anderem der Test auf Enthaltensein (der ja innerhalb der **while**-Schleife in Zeile 11 wiederholt durchgeführt werden muss) laufzeitoptimiert ist. Anfangs wird in Zeile 4 der Hashwert der ersten l Zeichen des Textes T berechnet. Jeder **while**-Schleifendurchlauf schiebt dann das „Fenster“ der zu hashenden Zeichen in T um eine Position nach rechts. In Zeile 9 wird hierfür der Hashwert gemäß Formel (7.4) angepasst. Immer dann, wenn der Hashwert h des „Fensters“ in der Menge *hashes* zu finden ist, ist es wahrscheinlich – jedoch keineswegs sicher –, dass eines der Muster gefunden wurde; um sicher zu gehen, dass sich an dieser Stelle auch tatsächlich eines der Muster befindet, muss der ungehashte Text mit den Mustern abgeglichen werden; dies geschieht in den Zeilen 7 und 12.

Wurden die Basis B und das Modul M geschickt gewählt, so sollte es sehr selten vorkommen, dass „**h in hashes**“ jedoch nicht „ $T[i:i+l]$ in Ms “ gilt. Somit kann man davon ausgehen, dass die Laufzeit des Rabin-Karp-Algorithmus in $O(n)$ liegt.

7.6 Der Shift-Or-Algorithmus

Der erst 1992 beschriebene Shift-Or-Algorithmus [17] nutzt Bitoperationen und arbeitet entsprechend äußerst effizient. Eine Variante dieses Stringmatching-Algorithmus verwendet das Unixtool `grep`.

Der Shift-Or-Algorithmus simuliert einen nichtdeterministischen endlichen Automaten (NEA). Im Gegensatz zum deterministischen endlichen Automaten (DEA), der für jedes Eingabezeichen immer eindeutig einen Zustandsübergang wählt, also jeder Zustand genau $|A|$ Ausgangskanten – eine für jedes Zeichen des Alphabets – besitzen muss, gibt es solche Beschränkungen bei NEAs nicht. Beispielsweise erkennt folgender NEA Vorkommen des Wortes **ananas**:



Der Nichtdeterminismus dieses Automaten zeigt sich beispielsweise dann, wenn er sich in Zustand „0“ befindet und das Eingabezeichen „a“ liest; dann gibt es nämlich zwei mögliche Zustandsübergänge: Er kann entweder über die mit „a“ beschriftete Kante in Zustand „1“ wechseln oder er kann über die mit „A“ beschriftete Kante² im Zustand „0“ verbleiben. Man sagt, ein NEA akzeptiert ein bestimmtes Wort w , wenn der Endzustand durch Lesen der Buchstaben in w erreichbar ist.

Aufgabe 7.15

Erstellen Sie einen *deterministischen* endlichen Automaten, der Vorkommen des Wortes **ananas** erkennt.

Enthält der nichtdeterministische Automat m Zustände, so wird die Menge der nach Lesen der ersten j Zeichen des Textes (also nach Lesen von $T[:j]$) erreichbaren Zustände in einem m -Bit-breiten Datenwort Z kodiert. Hierbei enthält das von rechts gesehen i -te Bit von Z genau dann eine Eins, wenn Zustand $i \in \{0, \dots, m-1\}$ des NEA durch Lesen von $T[:j]$ erreichbar ist. Wird ein Zustand Z erreicht, dessen Bit an Position „Null“ gesetzt ist (d. h. $Z = 1 z_1 z_2 \dots z_{m-1}$), dann ist der Endzustand $m-1$ erreichbar, und es wurde an der momentanen Textposition ein Vorkommen des Wortes erkannt.

Wie genau wird nun die Funktionsweise des NEA simuliert? Hierfür assoziieren wir mit jedem Buchstaben $x \in A$ des Alphabets A einen sog. charakteristischen Vektor $b[x]$, der folgendermaßen definiert ist:

$$b[x]_i = \begin{cases} 1, & \text{falls } M[-i] = x \\ 0, & \text{sonst} \end{cases}$$

²Kanten können auch mit Zeichen-Mengen beschriftet sein; eine solche Kante kann immer dann gegangen werden, wenn eines der in der Menge befindlichen Zeichen gelesen wurde.

Für das Muster **ananas** über dem Alphabet $A = \{\mathbf{a}, \dots, \mathbf{z}\}$ hätten die charakteristischen Vektoren die folgende Form:

$$\begin{aligned} b[\mathbf{a}] &= 010101 \\ b[\mathbf{n}] &= 001010 \\ b[\mathbf{s}] &= 100000 \\ b[x] &= 000000 \text{ für } x \in A \setminus \{\mathbf{a}, \mathbf{n}, \mathbf{s}\} \end{aligned}$$

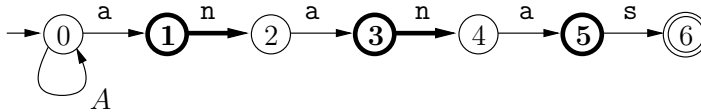
Der Shift-Or-Algorithmus speichert die charakteristischen Vektoren – ebenso wie die Zustände – in einem Datenwort der Breite m .

Der Algorithmus beginnt in Zustand „000000“, initialisiert die Variable Z also mit dem Wert „0“. Befindet sich der Algorithmus nach Lesen der ersten j Zeichen des Textes T in Zustand Z und liest er anschließend das Zeichen $T[j]$, so erhält man den neuen Zustand dadurch, indem man die folgenden bit-basierten Operationen ausführt:

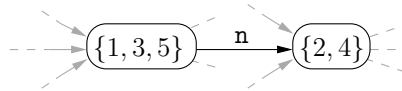
1. Die Bits des alten Zustands werden um eine Position nach links verschoben – dies entspricht beim NFA dem Weiterrücken um (jeweils) einen Zustand im Skelettau-
tomaten. Zusätzlich wird das rechteste Bit auf Eins gesetzt – dies entspricht dem
Weiterrücken des Zustands „0“ in den Zustand „1“.
2. Dieses Weiterrücken ist jedoch nur dann „erlaubt“, wenn das Zeichen, mit dem
der Zustandsübergang markiert ist, gelesen wurde. Daher erfolgt eine bitwei-
se UND-Verknüpfung der verschobenen Bits mit dem charakteristischen Vektor
 $b[T[j]]$ des aktuellen Zeichens $T[j]$. Nur dann nämlich, wenn die passenden Zu-
standsübergänge mit dem Zeichen $T[j]$ markiert sind, können die Zustände eine
Position weitergerückt werden.

Betrachten wir zur Illustration die folgende Beispielsituation: Wir nehmen an, dass durch den bisher gelesenen Text im NEA die Zustände „1“, „3“ und „5“ erreichbar wären und als Nächstes das Zeichen „n“ gelesen wird – diese Situation ist in Abbildung 7.6(a) dargestellt. Der entsprechende Kreuzproduktautomat des obigen Beispiel-NEA hat insgesamt 2^m Zustände, die mit Teilmengen der Zustände des NEA markiert sind. Dieser Kreuzproduktautomat würde sich in eben beschriebener Beispielsituation in Zustand „{1, 3, 5}“ befinden; dies würde im Falle des Shift-Or-Algorithmus dem Zustand $Z = 010101$ (bzw. in Dezimalschreibweise $Z = 21$) entsprechen – also das von rechts gesehen erste, dritte und fünfte Bit des Zustands wären gesetzt; durch Lesen des Zeichens „n“ gelangt der Kreuzproduktautomat in Zustand „{2, 4}“ – der entsprechende Ausschnitt des Kreuzproduktautomaten ist in Abbildung 7.6(b) dargestellt. Abbildung 7.6(c) zeigt das Weiterrücken der Zustände auf Bitebene (durch Anwendung der Operation „ $\ll 1 \mid 1$ “) und das anschließende Ausfiltern derjenigen Übergänge, die durch Lesen des Zeichens „n“ erlaubt sind; dies geschieht durch die bitweise UND-Verknüpfung mit dem charakteristischen Vektor $b[\mathbf{n}]$.

Abbildung 7.7 zeigt einen Beispiellauf des Shift-Or-Algorithmus, der zeigt, wie das Muster „ananas“ im Text „ananas“ gesucht wird. Es ist für jeden Leseschritt immer das Ergebnis der „ $\ll 1 \mid 1$ “-Operation, der charakteristische Vektor des gelesenen Zeichens und deren bitweise UND-Verknüpfung dargestellt, woraus sich der nächste Zustand ergibt.



(a) Beispiel-Situation während eines Durchlaufs des NEA.



(b) Beispiel-Situation während des Durchlaufs des entsprechenden Kreuzproduktautomaten.

$$\begin{array}{rcl}
 010101 & \xrightarrow{\ll 1} & 101011 \longrightarrow \underline{b[n] : 001010 \& 001010} \\
 & & Z : 101011
 \end{array}$$

(c) Entsprechende Bit-basierte Operationen um vom alten Zustand „{1, 3, 5}“ nach Lesen des Eingabezeichens „n“ zum neuen Zustand „{2, 4}“ zu kommen. Im ersten Schritt werden die Bits um eine Position nach links verschoben und durch die Oder-Operation das rechteste Bit gesetzt. Im zweiten Schritt erfolgt eine bitweise UND-Verknüpfung mit dem charakteristischen Vektor des gelesenen Zeichens „n“.

Abb. 7.6: Darstellung der folgenden Beispielsituation: Nach dem Lesen des bisherigen Eingabetextes sind die Zustände „1“, „3“ und „5“ des NEA erreichbar und das Zeichen „n“ wurde gelesen. Abbildung 7.6(a) stellt dies am NEA direkt dar, Abbildung 7.6(b) stellt dies am entsprechenden Kreuzproduktautomaten dar und Abbildung 7.6(c) zeigt die entsprechenden Bit-Operationen des Shift-Or-Algorithmus.

Aufgabe 7.16

Konstruieren sie sich den (Teil-)Kreuzproduktautomat, der für den in Abbildung 7.7 gezeigten Lauf des NEA relevant ist.

7.6.1 Implementierung

Listing 7.11 zeigt eine Implementierung des Shift-Or-Algorithmus. Zwischen Zeile 3 und Zeile 6 werden die charakteristischen Vektoren in Form eines *dict*-Objektes *b* berechnet. In Zeile 4 werden zunächst alle Einträge von *b* mit 0 initialisiert. Dann wird das Muster *M* einmal durchlaufen und für jedes Zeichen *c* des Musters wird im charakteristischen Vektor *b[c]* das Bit an der entsprechenden Position gesetzt – dies geschieht in Zeile 6.

Ab Zeile 8 erfolgt die Simulation des NEA: Zunächst wird der Anfangszustand *Z* auf „0“ gesetzt; der Endzustand des simulierten NEA wird in der Variablen *endZst* gespeichert. Die **for**-Schleife ab Zeile 9 durchläuft nun den zu durchsuchenden Text *T* zeichenweise und führt bei jedem Durchlauf die im letzten Abschnitt beschriebenen Operationen durch. In Zeile 12 wird durch bitweise UND-Verknüpfung mit *endZst* geprüft, ob der Endzustand erreichbar ist. Ist dies der Fall, so wird der entsprechende Index – hier *i* – *len(M)* + 1 – der Ergebnisliste *matches* angefügt.

Der Algorithmus hat offensichtlich eine Laufzeit von $O(n)$.

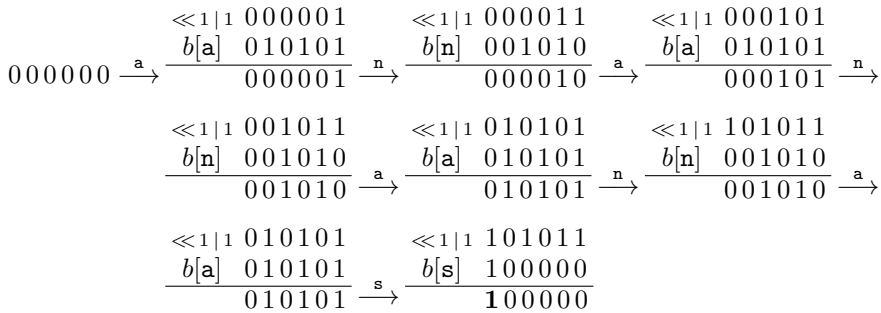


Abb. 7.7: Erkennen des Musters „*ananas*“ im Text „*ananas*“ durch Ausführung der Bitoperationen des Shift-Or-Algorithmus. Das Muster ist immer dann erkannt, wenn – in diesem Fall – das sechste Bit von rechts gesetzt wurde, wenn also der Zustand „6“ des entsprechenden NEA erreichbar ist.

```

1 def shiftOr(M,T):
2   # Berechnung der charakteristischen Vektoren
3   b={}
4   for i in range(256): b[chr(i)]=0
5   for i,c in enumerate(M):
6     b[c] = b[c] | 1<<i
7   # Simulation des NEA
8   Z=0 ; endZst = 1<<(len(M)-1) ; matches = []
9   for i,c in enumerate(T):
10    Z = Z<<1 | 1
11    Z = Z & b[c]
12    if Z & endZst: matches.append(i-len(M)+1)
13  return matches

```

Listing 7.11: Implementierung des Shift-Or-Algorithmus

Aufgabe 7.17

Führen Sie einen direkten Performance-Vergleich der bisher vorgestellten String-Algorithmen durch.

- Der Vergleich sollte mit einem relativ kurzen Muster (10 Zeichen) und einem relativ langen Muster (50 Zeichen) auf einer relativ kleinen Datenmenge (1000 Zeichen) und einer relativ großen Datenmenge (ca. 1 Million Zeichen) durchgeführt werden.
- Der Vergleich sollte mit dem naiven String-Matching-Algorithmus, dem Knuth-Morris-Pratt-Algorithmus, dem Boyer-Moore-Algorithmus, dem Rabin-Karp-Algorithmus und dem Shift-Or-Algorithmus durchgeführt werden.