# Kapitel 5

# Textalgorithmen

## 5.4 Der Algorithmus von Boyer und Moore

Der Algorithmus von Boyer und Moore findet ein oder alle Vorkommen eines Musters $P = P[1..m]$ in einem Text $S = S[1..n]$. Er löst also dieselbe Aufgabe wie der KMP-Algorithmus. Auch hier gibt es eine Vorverarbeitung des Musters und dann die Suche im Text.

Eigentlich handelt es sich dabei nicht um einen einzelnen Algorithmus, sondern um mehrere Varianten. Man kann von Algorithmen „vom Boyer-Moore-Typ“ sprechen. Allen Varianten ist gemeinsam, dass das Muster vorverarbeitet wird, um sogenannte *Shiftwerte* zu berechnen. Die Methoden, um zu diesen Shiftwerten zu kommen, sind aber unterschiedlich.

Algorithmen vom Boyer-Moore-Typ werden in der Praxis dem KMP-Algorithmus vorgezogen, da sie in der Suchphase oft schneller arbeiten als der KMP-Algorithmus. Das liegt daran, dass oft nicht alle Buchstaben von $S$ angesehen werden müssen, dass also viel weniger als $n$ Buchstabenvergleiche ausgeführt werden. Ein kleiner Nachteil, der aber nur in Sonderfällen relevant ist, ist folgender: Wenn das Muster im Text sehr oft vorkommt ($\Omega(n)$-mal, also mit vielen Überlappungen), dann kann die Laufzeit $\Omega(nm)$ betragen, im Gegensatz zum KMP-Algorithmus ($O(n+m)$ garantiert). Wenn das Muster nicht oder nur einmal vorkommt oder man nur nach dem ersten Vorkommen sucht, ist die Laufzeit $O(n + m)$. Die Laufzeitanalyse des BM-Algorithmus ist allerdings komplex; hierfür verweisen wir auf die Literatur [Dan Gusfield: Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology. Cambridge University Press 1997] oder [Volker Heun, Grundlegende Algorithmen, 2. Aufl., Vieweg, 2003].

### 5.4.1 Die Grundideen

Algorithmen vom Boyer-Moore-Typ suchen nach Vorkommen des Musters im Text, indem sie für eine strikt wachsende Folge von Werten $i \in \{m, m+1, \ldots, n\}$ testen, ob das Muster $P$ „an der Stelle $i$ in $S$ vorkommt“, das heißt, ob $P = S[i-m+1..i]$ gilt. Hierzu wird das Muster $P[1..m]$ Buchstabe für Buchstabe mit $S[i-m+1..i]$ verglichen. (**Achtung:** Die Stelle $i$ entspricht hier dem *letzten* Zeichen $P[m]$ des Musters.) Im Gegensatz zum KMP-Algorithmus läuft man dabei im Muster „von rechts nach links“.

**function compare**($i$) // Vorbedingung: $m \le i \le n$
(1) `j` $\leftarrow m$;
(2) **while** `j` $\ge 1 \wedge$ `P[j]` $=$ `S[`$i-m+$`j]` **do** `j--`;
(3) **return** `j`.

**Abbildung 5.1** Buchstabenvergleich „von rechts nach links“ an Stelle $i$.

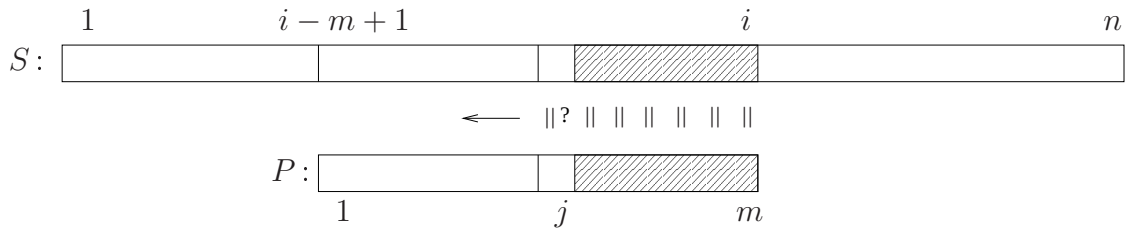


**Abbildung 5.2**
Buchstabenvergleich an Stelle $i$ von rechts nach links.

Der Rückgabewert ist eine Zahl $j$, wobei entweder $j$ die erste (von rechts gesehen) Position in $P$ ist, die nicht zur entsprechenden Stelle in $S$ passt, d. h., für die

$$j \ge 1 \text{ und } P[j+1..m] = S[i-m+j+1..i] \wedge P[j] \ne S[i-m+j]$$

gilt, oder $j = 0$ ist. Die Situation $j = 0$ bedeutet, dass das Muster gefunden worden ist.

In einem naiven Verfahren würde man diesen Vergleich für jedes $i = m, m+1, \ldots, n$ durchführen, d. h., man würde das Muster immer wieder um eine Position nach rechts verschieben. Die Kernidee der Algorithmen vom Boyer-Moore-Typ ist, aus der Information, die man bei der Suche an der Stelle $i$ gewonnen hat, abzuleiten, dass man vielleicht einige Positionen, nämlich $i+1, i+2, \ldots, i+s-1$ für ein $s \ge 1$, überspringen kann, weil das Muster an diesen Stellen auf keinen Fall passt. Die nächste Vergleichsposition ist dann $i+s$. Dieser *Shiftwert* $s$ sollte natürlich möglichst groß

sein, aber nicht so groß, dass man ein Vorkommen des Musters übersieht. Der Shiftwert hängt von $j$ und dem Muster und eventuell auch von einem Buchstaben in $S[i-m+j..i]$ ab. Methoden, um zu solchen Shiftwerten zu kommen, werden durch zwei Ideen geliefert. Die erste Idee ist, einen Buchstaben im Bereich $S[i-m+j..i]$ zu benutzen, um die nächste sinnvolle Position zu bestimmen („bad character rule“ und „Horspool-Regel“). Die zweite Idee, die den eigentlichen Boyer-Moore-Algorithmus ausmacht, ist es, nur aufgrund der Position $j$ des ersten Mismatch-Buchstabens zu entscheiden, wohin das Muster verschoben werden kann. Dies wird durch die „good suffix rule“ ausgedrückt. Diese führt zu einer (am Ende tabellierten, nur von $P$ abhängigen) Funktion $\{0,1,\ldots,m\} \ni j \mapsto \mathrm{GS}(j)$, die zu jedem $j$ den benötigten Shiftwert $s$ liefert. Wie beim KMP-Algorithmus ist zur Berechnung dieser Funktion aus $P$ eine etwas knifflige Vorverarbeitung des Musters nötig.

Wir bemerken noch, dass, im Gegensatz zum KMP-Algorithmus, Algorithmen vom Boyer-Moore-Typ nach dem Übergang zu einer neuen Position $i+s$ alle Information aus früheren Runden und Vergleichen vergessen und wieder mit dem naiven Vergleich „von rechts nach links“ beginnen.

### 5.4.2 Die BC-Regel

Wir beschreiben zunächst die „bad character rule“ („Schlechte-Buchstaben-Regel“, BC-Regel), die zur Ermittlung von Shiftwerten dient. Wenn ein Vergleich wie in Abb. 5.1 mit einem $j \in \{1,\ldots,m\}$ endet, dann hat man soeben den Buchstaben $x := S[i-m+j]$ gelesen, mit $x \neq P[j]$ („bad character“). An Stelle $i$ kann das Muster also nicht vorkommen. Die BC-Regel benutzt nun die Einsicht, dass das Muster in $S$ nur an Positionen $i+s$ sitzen kann, wo der Buchstabenposition $S[i-m+j]$ eine Stelle in $P$ gegenübersteht, die gleich $x$ ist. Man betrachtet also zunächst einmal alle Positionen in $P[1..m-1]$, die gleich $x$ sind[1], und wählt davon die maximale, entsprechend einem möglichst kurzen Shift, damit keine mögliche Position von $P$ ausgelassen wird. Rechnerisch ausgedrückt: Sei

$$\mathrm{BC}(x) := \max(\{0\} \cup \{k < m \mid P[k] = x\}),$$

das ist der größte Index $k < m$ mit $P[k] = x$, falls $x$ in $P[1..m-1]$ vorkommt, und 0, falls $x$ nicht vorkommt. Wenn $j \leq \mathrm{BC}(x)$, wählen wir Shiftwert 1. Wenn $j > \mathrm{BC}(x)$, dann kann für $i < i' < i + (j - \mathrm{BC}(x))$ keinesfalls $P = S[i'-m+1..i']$ gelten: Weil $\mathrm{BC}(x) < j+i-i' < m$, haben wir $P[j+i-i'] \neq x$; andererseits gilt $S[i'-m+(j+i-i')] = S[i-m+j] = x$. Wenn wir also als nächste zu testende Position $i + \max\{1, j - \mathrm{BC}(x)\}$ wählen, wird garantiert keine mögliche Position des Musters ausgelassen. Die Situation ist in Abb. 5.3 veranschaulicht.

[1] Ein Vorkommen von $x$ an der Stelle $P[m]$ kann man ignorieren. Da $P$ mindestens um 1 nach rechts geschoben wird, kann der letzte Buchstabe des Musters nach dem Schieben nie unter einem Buchstaben aus $S[i-m+j..i]$ stehen. Dies gilt für alle ähnlichen Verfahren, die einen Buchstaben aus $S[i-j+1..i]$ betrachten, wie z. B. die Horspool-Regel.

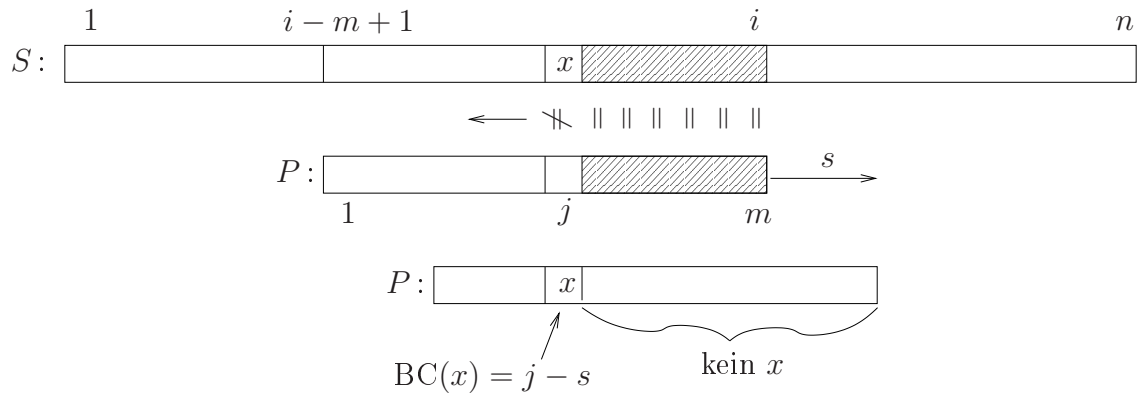


**Abbildung 5.3** BC-Regel: Wir wählen als Shiftweite $s = j - \mathrm{BC}(x)$, falls dies positiv ist. Wenn $j \leq \mathrm{BC}(x)$, hilft die Regel nichts, und wir wählen $s = 1$.

Die Werte $\mathrm{BC}(x)$, $x \in \Sigma$, lassen sich ganz einfach in Zeit $O(m)$ berechnen. Wir benutzen dazu ein (assoziatives) Array BC, das für jeden Buchstaben $x \in \Sigma$ eine Position hat. Es genügt ein Durchlauf durch das Muster von links nach rechts. Man beobachte, wie durch die Benutzung von $\mathrm{BC}[P[\mathtt{k}]]$ als Speicherziel mit $k = 1, \ldots, m-1$ die maximale Position $k < m$ mit $T[k] = x$ in $\mathrm{BC}[x]$ gespeichert wird, falls $x$ in $P[1..m-1]$ vorkommt.[2]

**function buildBCTable**($P[1..m]$)
(1) **for** $x \in \Sigma$ **do** $\mathrm{BC}[x] \leftarrow 0$;
(2) **for** $\mathtt{k}$ **from** 1 **to** $m$ **do** $\mathrm{BC}[P[\mathtt{k}]] \leftarrow \mathtt{k}$;
(3) **return** $\mathrm{BC}[x\colon \Sigma]$.

Wenn die Buchstaben in $P$ nicht das ganze Alphabet $\Sigma$ ausschöpfen, wird man die „Fehlbuchstaben“, die in $P$ nicht vorkommen, in *einem* Eintrag der Tabelle BC zusammenfassen: $\mathrm{BC}[\text{„Fehlbuchstabe“}] = 0$.

*Beispiel*: $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ und $P[1..11] = \mathtt{abracadabra}$. Dann ist BC für $P$ durch die folgende Tabelle gegeben. Man beachte, dass der Eintrag $P[11] = \mathtt{a}$ ignoriert wird und daher $\mathrm{BC}(\mathtt{a}) = 8$ ist.

| $x$ | a | b | c | d | r | sonstige |
|---|---|---|---|---|---|---|
| $\mathrm{BC}(x)$ | 8 | 9 | 5 | 7 | 10 | 0 |

Die ***einfache* BC-Regel** sagt nun: Wenn an der Stelle $j$ ein Mismatch auftritt, dann verschiebe das Muster um $s := \max\{1, j - \mathrm{BC}(x)\}$ nach rechts. Dann verfahre wieder

[2] Dieser Trick zur Bildung eines Maximums wird uns im Weiteren noch mehrfach begegnen.

wie in Abb. 5.1. Wenn das Muster an Stelle $i$ gefunden worden ist, verschiebe um $s = 1$. – Es ergibt sich der folgende Algorithmus.

**Algorithmus 5.4.1** (Boyer-Moore mit einfacher BC-Regel).
**Eingabe:** `P[1..m]`, `S[1..n]` // `P`: Muster, `S`: Text
**Vorberechnet:** BC-Shiftwerte als Tabelle `BC`$[x\colon \Sigma]$ ;
**Ausgabe:** Menge `A` $\subseteq \{1, \ldots, n-m+1\}$; initialisiert: `A` $\leftarrow \emptyset$;

(1) **int** `i` $\leftarrow m$;
(2) **while** `i` $\leq n$ **do**
(3) `j` $\leftarrow m$;
(4) **while** `j` $\geq 1 \wedge$ `P[j]` $=$ `S[i-`$m$`+j]` **do** `j--`;
(5) **if** `j` $= 0$
(6) **then** `A` $\leftarrow$ `A` $\cup \{$`i-`$m$`+1`$\}$; `i++`;
(7) **else** `i` $\leftarrow$ `i +` $\max(1,$ `j -` BC`[S[i-`$m$`+j]]`$)$;
(8) **return** `A`.

*Beispiel*: (Blanks zählen als Buchstaben.)[2]

$S[1..42] =$ `IM␣HEU-␣ODER␣NUDELHAUFEN␣FINDE␣ALLE␣NADELN`, $P[1..5] =$ `NADEL`.

Vorverarbeitung:

| $x$ | `A` | `D` | `E` | `N` | sonstige |
|---|---|---|---|---|---|
| BC($x$) | 2 | 3 | 4 | 1 | 0 |

| | | |
|---|---|---|
| $i = 5$ | Mismatch bei $j = 5$ mit $x =$ `E` | $s = 5 - \mathrm{BC}($`E`$) = 1$ |
| $i = 6$ | Mismatch bei $j = 5$ mit $x =$ `U` | $s = 5 - \mathrm{BC}($`U`$) = 5$ |
| $i = 11$ | Mismatch bei $j = 5$ mit $x =$ `E` | $s = 5 - \mathrm{BC}($`E`$) = 1$ |
| $i = 12$ | Mismatch bei $j = 5$ mit $x =$ `R` | $s = 5 - \mathrm{BC}($`R`$) = 5$ |
| $i = 17$ | Mismatch bei $j = 5$ mit $x =$ `E` | $s = 5 - \mathrm{BC}($`E`$) = 1$ |
| $i = 18$ | Mismatch bei $j = 2$ mit $x =$ `U` | $s = 2 - \mathrm{BC}($`U`$) = 2$ |
| $i = 20$ | Mismatch bei $j = 5$ mit $x =$ `A` | $s = 5 - \mathrm{BC}($`A`$) = 3$ |
| $i = 23$ | Mismatch bei $j = 5$ mit $x =$ `E` | $s = 5 - \mathrm{BC}($`E`$) = 1$ |
| $i = 24$ | Mismatch bei $j = 5$ mit $x =$ `N` | $s = 5 - \mathrm{BC}($`N`$) = 4$ |
| $i = 28$ | Mismatch bei $j = 5$ mit $x =$ `N` | $s = 5 - \mathrm{BC}($`N`$) = 4$ |
| $i = 32$ | Mismatch bei $j = 5$ mit $x =$ `A` | $s = 5 - \mathrm{BC}($`A`$) = 3$ |
| $i = 35$ | Mismatch bei $j = 5$ mit $x =$ `E` | $s = 5 - \mathrm{BC}($`E`$) = 1$ |
| $i = 36$ | Mismatch bei $j = 5$ mit $x =$ `␣` | $s = 5 - \mathrm{BC}($`␣`$) = 5$ |
| $i = 41$ | Muster gefunden, $j = 0$ | $s = 1$ |
| $i = 42$ | Mismatch bei $j = 5$ mit $x =$ `N` | $s = 5 - \mathrm{BC}($`N`$) = 4$ |
| $i = 46$ | Stop. | |

[2] Zum Ausprobieren: `https://people.ok.ubc.ca/ylucet/DS/BoyerMoore.html` .
Aber Achtung: Die App behandelt mit BC(`L`) = 5 den letzten Buchstaben „`L`“ des Musters anders (ungeschickter) als unser Algorithmus.

Es werden 22 Buchstabenvergleiche ausgeführt. Das ist wenig im Vergleich zur Länge des betrachteten Textes (42), besonders wenn man bedenkt, dass allein für das Finden des Musters schon fünf (erfolgreiche) Vergleiche nötig sind.

Nicht in allen Situationen sind die möglichen weiten Sprungweiten so hilfreich: Betrachte $S[1..13] = \texttt{abababcababac}$, $P[1..4] = \texttt{caba}$.

Vorverarbeitung:

| $x$ | a | b | c | sonstige |
|---|---|---|---|---|
| BC($x$) | 2 | 3 | 1 | 0 |

| | | |
|---|---|---|
| $i = 4$ | Mismatch bei $j = 4$ mit $x = \texttt{b}$ | $s = 4 - \mathrm{BC}(\texttt{b}) = 1.$ |
| $i = 5$ | Mismatch bei $j = 1$ mit $x = \texttt{b}$ | $s = \max\{1, \underbrace{1 - \mathrm{BC}(\texttt{b})}_{=-2}\} = 1.$ |
| $i = 6$ | Mismatch bei $j = 4$ mit $x = \texttt{b}$ | $s = 4 - \mathrm{BC}(\texttt{b}) = 1.$ |
| $i = 7$ | Mismatch bei $j = 4$ mit $x = \texttt{c}$ | $s = 4 - \mathrm{BC}(\texttt{c}) = 3.$ |
| $i = 10$ | Muster gefunden, $j = 0$ | $s = 1.$ |
| $i = 11$ | Mismatch bei $j = 4$ mit $x = \texttt{b}$ | $s = 4 - \mathrm{BC}(\texttt{b}) = 1.$ |
| $i = 12$ | Mismatch bei $j = 1$ mit $x = \texttt{b}$ | $s = \max\{1, 1 - \mathrm{BC}(\texttt{b})\} = 1.$ |
| $i = 13$ | Mismatch bei $j = 4$ mit $x = \texttt{c}$ | $s = 4 - \mathrm{BC}(\texttt{c}) = 3.$ |
| $i = 16$ | Stop. | |

Ablauf von Algorithmus 5.4.1 als Schema: „ “ bezeichnet „erfolgreicher Vergleich“, „ “ bedeutet „Mismatch“. Die unterstrichenen Buchstaben sind diejenigen, die von der BC-Regel unter die passenden Textbuchstaben gesetzt werden. Positionen ohne unterstrichene Buchstaben kommen durch einen Shiftwert von 1 zustande, wenn die BC-Regel nicht anwendbar ist.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | a | b | a | b | c | a | b | a | b | a | c | – | – | – |
| 4 | c | a | b | a | | | | | | | | | | | | |
| 5 | | c | a | b | a | | | | | | | | | | | |
| 6 | | | c | a | b | a | | | | | | | | | | |
| 7 | | | | c | a | b | a | | | | | | | | | |
| 10 | | | | | | | c | a | b | a | | | | | | |
| 11 | | | | | | | | c | a | b | a | | | | | |
| 12 | | | | | | | | | c | a | b | a | | | | |
| 13 | | | | | | | | | | c | a | b | a | | | |
| 16 | Stop. | | | | | | | | | | | | c | a | b | a |

Bei Textlänge 13 gibt es 17 Vergleiche, davon 7 erfolglose („Mismatches“) und 10 erfolgreiche. Ein Buchstabe des Textes ($S[1]$) wird nicht gelesen, andererseits wird $S[10]$ zweimal erfolgreich verglichen. Bei größeren Mustern und Texten kann dies natürlich auch noch häufiger auftreten.

Die einfache BC-Regel wirkt nicht, wenn $j < \mathrm{BC}(x)$ ist, wenn also $x$ rechts von der Mismatch-Stelle $j$ in $P$ vorkommt, oder nach dem Finden des Musters. Man

schiebt dann nur um einen Buchstaben. (Im letzten Beispiel ist bei Positionen $i = 5$ und $i = 12$ der Wert $j$ zu klein; bei $i = 10$ wurde das Muster gefunden.) Dies kann bei kleinen Alphabeten eher vorkommen als bei großen. Besonders ungünstig sind hier zum Beispiel das binäre Alphabet $\{0, 1\}$ und vierelementige Alphabete wie $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Wir betrachten nun Methoden, die diesem Problem ausweichen.

*Variante* 1: Anstelle des „Mismatch-Buchstabens“ $x = S[i - m + j]$ betrachtet man $y = S[i]$ und schiebt das Muster um $s = m - \mathrm{BC}(y)$ Positionen nach rechts. (Der Buchstabe $S[i]$ ist natürlich nicht unbedingt ein „bad character“.) Dadurch erhält man immer eine positive Verschiebung aus der BC-Regel, und das am weitesten rechts stehende $y$ in $P[1..m-1]$ kommt an die Position von $y = S[i]$. Es ergibt sich folgender Algorithmus, der 1980 von Horspool[4] vorgeschlagen wurde:

**Algorithmus 5.4.2** (Boyer-Moore: Horspool-Variante)**.**
**Eingabe:** `P[1..`$m$`]`, `S[1..`$n$`]` // `P`: Muster, `S`: Text
**Vorberechnet:** BC-Shiftwerte als Tabelle `BC`$[x\colon \Sigma]$ ;
**Ausgabe:** Menge `A` $\subseteq \{1, \ldots, n - m + 1\}$; initialisiert: `A` $\leftarrow \emptyset$;

(1) **int** `i` $\leftarrow m$;
(2) **while** `i` $\leq n$ **do**
(3) `j` $\leftarrow m$;
(4) **while** `j` $\geq 1 \wedge$ `P[j]` $=$ `S[i-`$m$`+j]` **do** `j--`;
(5) **if** `j` $= 0$ **then** `A` $\leftarrow$ `A` $\cup \{$`i-`$m$`+1`$\}$;
(6) `i` $\leftarrow$ `i + `$m$` - BC[S[i]]`; // *nur hier* Unterschied zu Algorithmus 5.4.1
(7) **return** `A`.

Ablauf von Algorithmus 5.4.2 als Schema. „ “ bezeichnet „erfolgreicher Vergleich“, „ “ bedeutet „Mismatch“. Die unterstrichenen Buchstaben sind diejenigen, die von der Horspool-Regel unter die passenden Textbuchstaben gesetzt werden.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | a | b | a | b | c | a | b | a | b | a | c | – |
| 4 | c | a | b | a | | | | | | | | | | |
| 5 | | c | a | b | a | | | | | | | | | |
| 7 | | | | c | a | b | a | | | | | | | |
| 10 | | | | | | | c | a | b | a | | | | |
| 12 | | | | | | | | | c | a | b | a | | |
| 14 | Stop. | | | | | | | | | | c | a | b | a |

Hier gibt es nur 5 Mismatches und 10 erfolgreiche Vergleiche. Textpositionen $S[1]$, $S[6]$, $S[13]$ werden nicht gelesen; Textposition $S[10]$ wird zweimal erfolgreich verglichen.

Man vergleiche die Abläufe der beiden Algorithmen und beachte, dass sich BM mit BC-Regel und BM-Horspool an der Stelle $i$ gleich verhalten, wenn ein Mismatch bei $j = m$ auftritt. Wenn der Mismatch bei $j < m$ passiert oder wenn das Muster gefunden worden ist, schiebt BM-Horspool immer um denselben Wert $s = m - \mathrm{BC}(P[m])$

[4] R. Nigel Horspool, britisch-kanadischer Informatiker.

weiter. (Im Beispiel ist dies 2.)

Die einfache BC-Regel, insbesondere die Horspool-Variante, hat den Vorteil, sehr einfach anwendbar und sehr effizient zu sein. Es wird ihr nachgesagt, dass sie alleine (ohne die noch folgenden komplexeren Teile des BM-Algorithmus!) zu sehr schnellen Suchzeiten in natürlichsprachigen Texten führt. Allgemein wirkt die Regel umso besser, je größer das Alphabet ist. Wenn dann in **compare** (Abb. 5.1) schon früh, also für große $j$, ein Mismatch mit einem solchen Buchstaben auftritt, kann das Muster ohne große Kosten auf einen Schlag sehr weit geschoben werden.

**Bemerkung.** Wenn die Buchstabenverteilung zufällig ist, dann tritt typischerweise früh ein Mismatch ein: An einer Position $i$ erwarten wir nur konstant viele Buchstabenvergleiche. Wenn zudem das Alphabet groß ist, dann ist die erwartete Shiftweite groß, nämlich $\Theta(\min\{m, |\Sigma|\})$. Daraus ergibt sich eine erwartete Rechenzeit für die Textsuche von $O(n/\min\{m, |\Sigma|\})$. Wir skizzieren eine heuristische Rechnung, die dies belegt. Nehmen wir an, das Alphabet $\Sigma$ habe Größe $\sigma$ und Text $S[1..n]$ und Muster $P[1..m]$ sind rein zufällig gewählt, d. h., an jeder Stelle steht jeder Buchstabe mit derselben Wahrscheinlichkeit $1/\sigma$, und die Positionen sind unabhängig.[3]
Nehmen wir zur Vereinfachung weiter an, dass die Buchstaben im Text nach jeder Verschiebung des Musters aufs Neue zufällig bestimmt werden, so dass wir uns um Abhängigkeiten bei überlappenden Musterpositionen nicht kümmen müssen. **Übung:** (i) Dann ist die erwartete Anzahl von Buchstabenvergleichen an Stelle $i$ genau $\frac{\sigma}{\sigma-1}(1-(1/\sigma)^m)$, ein Wert in $(1, 2)$, also $\Theta(1)$. (ii) Die erwartete Shiftdistanz ist $\sigma(1-(1-1/\sigma)^m)$. Für $m > \frac{\sigma}{2}$ führt dies zur Abschätzung $\sigma(1-(1-1/\sigma)^{\sigma/2}) = O(\sigma)$. Die Shiftdistanz ist nie größer als $m$. (iii) Insgesamt ist damit die erwartete Anzahl von Buchstabenvergleichen in $O(n/\min\{m, \sigma\})$.

*Variante* 2: Eine gewisse Verbesserung kann man auch durch Verwendung der ***starken Version*** der BC-Regel erreichen. Wenn $j < \mathrm{BC}(x)$ ist, wäre es wünschenswert, das erste Vorkommen von $x$ in $P$ *strikt links von* Position $j$ zu finden. Dazu sei $\mathrm{BC}'(x, j)$ der größte Index $k < j$ mit $P[k] = x$ (dabei sei $\mathrm{BC}'(x, j) = 0$, falls $x$ in $P[1..j-1]$ nicht vorkommt). Dann können wir Shiftwert $s = j - \mathrm{BC}'(x, j) \geq 1$ wählen.

Man kann die $\mathrm{BC}'$-Funktion auf naive Weise bereitstellen: mit $\Theta(m|\Sigma|)$ Speicherplatz und Vorbereitungszeit. Dann kostet jeder Zugriff $O(1)$ Zeit. Allerdings möchte man besonders für größere Alphabete keinen solchen Aufwand treiben. Ein guter Kompromiss ist folgender: Man legt für jeden Buchstaben $x \in \Sigma$ eine lineare Liste $L_x$ an, in der die Elemente der Menge $\{k \mid 1 \leq k < m, P[k] = x\} \cup \{0\}$ in fallender Reihenfolge aufgeführt sind. (Diese Vorverarbeitung kann in Zeit $O(m)$ durchgeführt werden. Auch der Platzbedarf ist $O(m)$; wenn $m$ viel kleiner als $|\Sigma|$ ist, kann man eine Hashtabelle o. ä. benutzen, um viele leere Listen zu vermeiden.)

*Beispiel*: Für $P[1..14] =$ `araratararatar` ist $L_{\mathtt{a}} = (13, 11, 9, 7, 5, 3, 1, 0)$, $L_{\mathtt{r}} = (10, 8, 4, 2, 0)$, $L_{\mathtt{t}} = (12, 6, 0)$ und $L_{\text{sonst}} = (0)$ bzw. die Konvention, dass Buchstaben, die nicht in $P$ vorkommen, stets den Wert 0 liefern.

[3] Vorsicht: Diese Annahme trifft in in wirklichen Anwendungen nicht zu. Aber sie zeigt das Einsparpotenzial des BM-Horspool-Algorithmus auf.

Mit Hilfe dieser Listen lässt sich für jedes $x$ und $j$ der Wert $\mathrm{BC}'(x, j)$ in Zeit $O(m-j)$ berechnen: Man liest Liste $L_x$ bis zum ersten Eintrag, der kleiner als $j$ ist. Da Zeit $\Omega(m-j)$ auch für den Durchlauf der Schleife in der Funktion **compare** (Abb. 5.1) benötigt wird, bis Position $j$ erreicht ist, erhöht sich der Zeitbedarf höchstens um einen konstanten Faktor gegenüber dem, was bei der Benutzung einer kompletten Tabelle für $\mathrm{BC}'$ nötig ist.

Trotz aller Erfolge und Vorteile der verschiedenen BC-Regeln ist doch festzuhalten, dass sie im schlechtesten Fall alle zu einer sehr schlechten Rechenzeit führen können, wie die folgende Übungsaufgabe zeigt.

**Übung:** Zeigen Sie, dass die Boyer-Moore-Varianten mit der BC-Regel, der Horspool-Regel und auch der $\mathrm{BC}'$-Regel auf der Eingabe $(P, T)$ mit $P = P[1..m] = \mathtt{ba}^{m-1}$ und $S = S[1..n] = \mathtt{a}^{n-m}\mathtt{ba}^{m-1}$ $(n-m+1)m$ Buchstabenvergleiche durchführen. (Der Grund ist bei allen drei Varianten, dass für eine Position $i$ immer alle Buchstaben des Musters angesehen werden und dass die Shiftweite jeweils nur 1 beträgt.)

### 5.4.3 Die GS-Regel

Die eigentliche Idee des von Boyer und Moore vorgeschlagenen Algorithmus ist die „good suffix rule“ („Gute-Suffix-Regel“, GS-Regel). Dabei geht es darum, gewisse Shiftweiten für das Muster dadurch auszuschließen, dass die bereits erfolgreich verglichenen Buchstaben $P[j+1..m]$ betrachtet werden sowie für $j \geq 1$ die Tatsache eines Mismatches an dieser Stelle. Von den verbleibenden „zulässigen“ Shiftweiten wird dann die kleinste gewählt, damit keine mögliche Position ausgelassen wird. Erst die GS-Regel führt zu garantierter Suchzeit $O(n)$. Die Überlegungen zu dieser Regel liefern eine Funktion $\mathrm{GS}\colon \{0, \ldots, m\} \ni j \mapsto \mathrm{GS}(j) \in \{1, \ldots, m\}$, die für die Anwendung im Algorithmus tabelliert wird. Dann wird im Algorithmus nach dem Testen an Stelle $i$ mit erster Mismatch-Stelle $j$ zur neuen Position $i + \mathrm{GS}(j)$ gesprungen. Die Definition der Funktion GS erfordert einige Überlegung, die Berechnung der Tabelle in Linearzeit weitere Mühe.

Sei $0 \leq j \leq m$, und sei $j$ die Zahl, die vom Aufruf **compare**$(i)$ (s. Abb. 5.1) geliefert wird. Wenn $1 \leq j \leq m$, dann handelt es sich um eine Mismatch-Stelle, bei $j = 0$ wurde soeben das Muster gefunden. Sei $1 \leq s \leq m$. Wir überlegen, ob es sinnvoll sein kann, das Muster um $s$ Stellen nach rechts zu schieben. Es gibt zwei Fälle.

**1. Fall:** $s < j$ („kleiner Shift“).

Wir wissen nach dem Ergebnis von **compare**$(i)$: $P[j+1..m] = S[i-m+j+1..i]$ und $P[j] \neq S[i-m+j]$. Wenn nach Verschieben um $s$ Stellen nach rechts (an die Position $i+s$) das Muster „passt“, also $P[1..m] = S[i+s-m+1..i+s]$ gilt, dann muss $P[j-s+1..m-s] = S[i-m+j+1..i]$ und $P[j-s] = S[i-m+j]$ gelten (s.

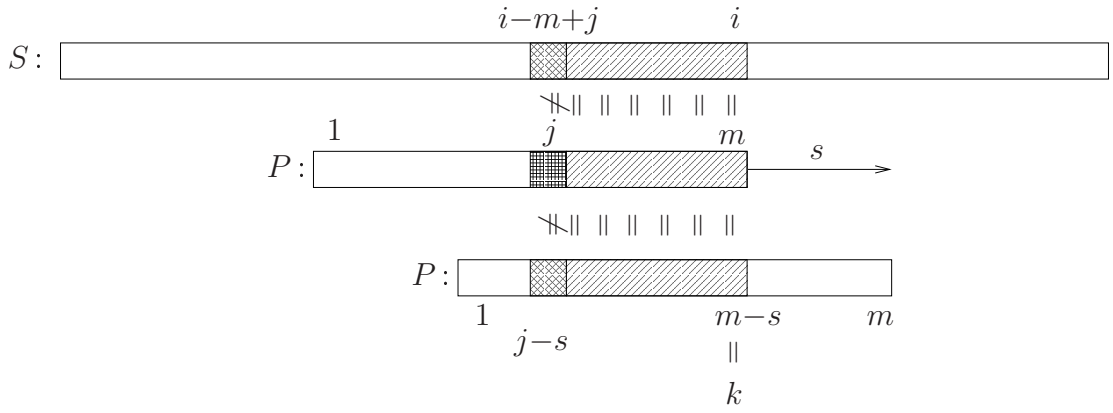


**Abbildung 5.4** GS-Regel, 1. Fall: „kleiner Shift“. Gleichheiten und Ungleichheiten zwischen Positionen des Musters und des Textes, falls ein Shift von $s < j$ zu einer Position $i + s$ im Text führt, an der das Muster vorkommt.

Abb. 5.4). Es folgt:

$$P[j+1..m] = P[j-s+1..m-s] \text{ und } P[j] \neq P[j-s]. \tag{5.1}$$

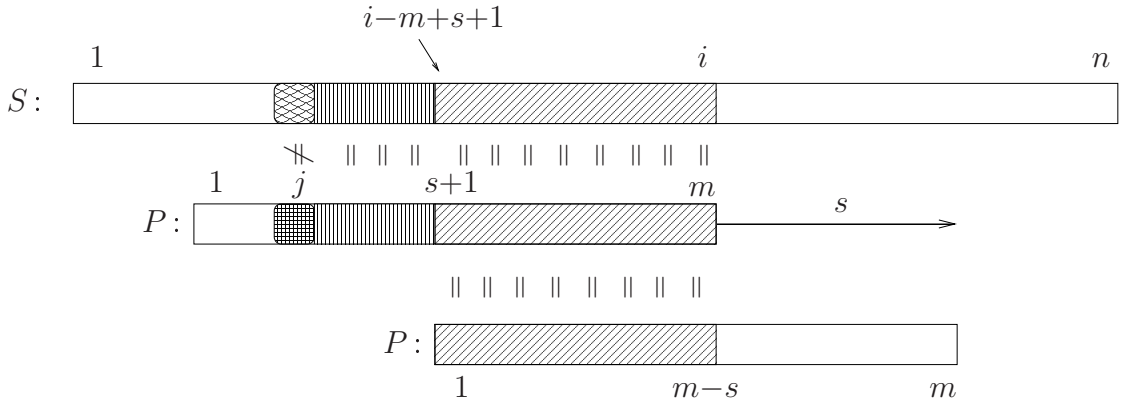


**Abbildung 5.5** GS-Regel, 2. Fall: „großer Shift“. Gleichheiten und Ungleichheiten zwischen Positionen des Musters und des Textes, falls ein Shift von $s \geq j$ zu einer Position $i + s$ im Text führt, an der das Muster vorkommt.

**2. Fall:** $j \leq s \leq m$ („großer Shift“).

(Hier ist der Fall $j = 0$ eingeschlossen, was bedeutet, dass das Muster an Stelle $i$ gefunden worden ist.) Nach dem Ergebnis von **compare**$(i)$ wissen wir: $P[j+1..m] = S[i-m+j+1..i]$, also insbesondere $P[s+1..m] = S[i-m+s+1..i]$. Bei einer Verschiebung um $s$ steht das Muster $P$ komplett rechts von der Stelle $j$, der nicht passende Buchstabe spielt also keine Rolle. Aus $S[i+s-m+1..i+s] = P[1..m]$ folgt

$S[i+s-m+1..i] = P[1..m-s]$, also (s. Abb. 5.5):

$$P[s+1..m] = P[1..m-s] \qquad (\text{d.h.: } P[1..m-s] \text{ ist Rand von } P). \tag{5.2}$$

**Bemerkungen:** (a) Bedingungen (5.1) und (5.2) betreffen das Suffix $P[j..m]$ bzw. das Suffix $P[s+1..m]$ von $P$, was zu dem Namen "good suffix rule" führt.
(b) Manchmal unterscheidet man „schwache GS-Regel" (*ohne* die Bedingung „$P[j] \neq P[j-s]$" in (5.1)) und „starke GS-Regel" (*mit* dieser Bedingung). Es stellt sich heraus, dass die Mismatch-Bedingung für die lineare Laufzeit entscheidend ist, daher betrachten wir *nur* die starke Version.

**Definition 5.4.3.** *Sei $P[1..m]$ das Muster und $0 \leq j \leq m$. Eine Zahl $s$, $1 \leq s \leq m$, heißt **zulässige Shiftweite für $j$**, wenn $1 \leq s < j$ und (5.1) oder $j \leq s \leq m$ und (5.2) gilt.*

Wegen $P[m+1..m] = \varepsilon = P[1..0]$ ist die Shiftweite $s = m$ immer zulässig.

**Definition 5.4.4.** $\mathrm{GS}(j) := \min\{s \mid s \text{ ist zulässige Shiftweite für } j\}$ *heißt der **GS-Shiftwert** für $j$.*

*Beispiel*: $P[1..14] =$ `araratararatar`.

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathrm{GS}(j)$ | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 12 | 12 | 12 | 12 | 12 | 4 | 14 | 1 |

Die Werte 6 für $j = 0, \ldots, 6$ in dieser Tabelle rühren daher, dass der längste Rand von $P[1..14]$ das Teilwort `araratar` (mit Länge $8 = 14 - 6$) ist. Es ist eine gute Übung, auch die anderen Werte anhand der Definition zu überprüfen. Insbesondere der Wert $\mathrm{GS}(13) = 14$ ist interessant. Er bedeutet, dass im Fall $S[i] =$ `r` und $S[i-1] \neq$ `a` das Muster um 14 Positionen weitergeschoben werden kann. Tatsächlich: Links von jedem Vorkommen von `r` im Muster steht ein `a`, so dass es keine Möglichkeit gibt, dass Position $S[i]$ in einem Vorkommen des Musters enthalten ist.

Aus den bisherigen Überlegungen folgt: Wenn wir nach einer Runde für $i$, die entweder mit einem Mismatch $P[j] \neq S[i-m+j]$ für $j \geq 1$ oder mit einem Auffinden des Musters ($j = 0$) geendet hat, den nächsten Versuch an Position $i := i + \mathrm{GS}(j)$ starten, so wird hierdurch kein Vorkommen des Musters $P$ in $S$ übersehen. Nun können wir den Boyer-Moore-Algorithmus formulieren. Wir tun dies zunächst in der ursprünglichen Version.

**Algorithmus 5.4.5** (Boyer-Moore)**.**
**BM-Textsuche**(S[1..$n$],P[1..$m$])
**Eingabe:** P[1..$m$], S[1..$n$] // P: Muster, S: Text
**Vorberechnet:** GS[0..$m$]: GS-Regel-Shiftwerte als Tabelle;
**Ausgabe:** Menge A $\subseteq \{1,\ldots,n-m+1\}$; initialisiert: A $\leftarrow \emptyset$;

```
(1)  int i ← m;
(2)  while i ≤ n do
(3)    j ← m;
(4)    while j ≥ 1 ∧ P[j] = S[i-m+j] do j--;
(5)    if j = 0 then A ← A ∪ {i-m+1};
(6)    i ← i + GS[j];
(7)  return A.
```

Man bewundere die Einfachheit dieses Algorithmus. (Das Geheimnis liegt natürlich in der GS-Funktion.) Wir können den Algorithmus noch erweitern, indem wir auch die BC-Regel in der einen oder anderen Form berücksichtigen.

**Algorithmus 5.4.6** (Boyer-Moore mit BC($'$)-Regel)**.**
**BM+BC-Textsuche**(S[1..$n$],P[1..$m$])
**Eingabe:** P[1..$m$], S[1..$n$] // P: Muster, S: Text
**Vorberechnet:** GS[0..$m$]: GS-Regel-Werte als Tabelle;
Datenstruktur für (BC($x$) oder) BC$'(x,j)$, $x \in \Sigma$, $1 \le j \le m$;
**Ausgabe:** Menge A $\subseteq \{1,\ldots,n-m+1\}$; initialisiert: A $\leftarrow \emptyset$;

```
(1)  int i ← m;
(2)  while i ≤ n do
(3)    j ← m;
(4)    while j ≥ 1 ∧ P[j] = S[i-m+j] do j--;
(5)    if j = 0 then A ← A ∪ {i-m+1};
(6)    s ← GS[j];
(7)    if j ≥ 2 then s ← max(s,j-BC'[S[i-m+j],j]);
(8)    i ← i+s;
(9)  return A.
```

Der folgende Satz zum Berechnungsaufwand wird hier nicht bewiesen. Für Interessierte findet sich der Beweis im Buch von V. Heun oder dem von D. Gusfield.

**Satz 5.4.7.** *Algorithmus 5.4.5 führt maximal $4n$ Buchstabenvergleiche durch und hat Laufzeit $O(n)$, wenn das Muster nicht im Text vorkommt.*

Es ist eine typische Subtilität im Verhalten dieser Algorithmen vom Boyer-Moore-Typus und eine große Schwierigkeit in der Zeitanalyse, dass nicht ohne Weiteres gesagt ist, dass die Anzahl der Vergleiche in Algorithmus 5.4.6 geringer ist als die in Algorithmus 5.4.5. (Wieso? Der Grund ist, dass durch die BC-Shifts ganz andere $i$-Werte erreicht werden können als im gewöhnlichen Ablauf ohne BC-Shifts. Was dies dann auf die Laufzeit für Auswirkungen hat, ist schwierig zu sehen.) Man kann

aber auch im Fall des BM-Algorithmus mit BC-Shifts lineare Laufzeit beweisen, *falls das Muster nicht vorkommt*. Daraus folgt auch lineare Laufzeit, wenn man nur das erste Vorkommen des Musters finden will. Wenn man *alle* Vorkommen finden will und dennoch lineare Laufzeit garantiert sein soll, muss man den Algorithmus etwas modifizieren („Galil-Modifikation“, siehe das schon erwähnte Buch von Gusfield).

Was ist noch zu tun? Wir müssen einen Linearzeit-Algorithmus für die Berechnung der GS-Regel-Shiftwerte angeben. Dies erfordert zusätzlichen (mentalen und Rechen-) Aufwand, und einen eigenen Abschnitt.

**WS 2019/20: Ende des prüfungsrelevanten Teils.**

### 5.4.4 Vorbearbeitung des Musters für den BM-Algorithmus

Die Ermittlung der Shiftwerte GS($j$) für die GS-Regel erfordert mehrere Schritte. Der erste und zentrale Schritt ist die Berechnung der sogenannten *Präfixwerte* oder *Z-Werte* eines Wortes $T = T[1..n]$. Diese Werte liefern auch für andere Anwendungen nützliche Strukturinformation über $T$. Dazu gespiegelt definieren wir *Suffixwerte*. Wenn die Suffixwerte des Musters vorliegen, lassen sich die Shiftwerte recht einfach ermitteln.

#### 5.4.4.1 Die Präfixwerte eines Wortes

**Definition 5.4.8.** *Sei $T[1..n]$ ein Wort. Für $2 \leq i \leq n$ definieren wir den* Z-Block an Stelle $i$ *als das längste Präfix $T[i..i+z-1]$ von $T[i..n]$, das auch Präfix von $T$ (also gleich $T[1..z]$) ist. Die Länge $z$ dieses Z-Blocks nennen wir $Z_i$.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[i]$ | a | r | a | b | a | r | a | b | a | r | a | r | t | a | r | a |
| 2 | | a | | | | | | | | | | | | | | |
| 3 | | | a | r | | | | | | | | | | | | |
| 4 | | | | a | | | | | | | | | | | | |
| 5 | | | | | a | r | a | b | a | r | a | b | | | | |
| 6 | | | | | | a | | | | | | | | | | |
| 7 | | | | | | | a | r | | | | | | | | |
| 8 | | | | | | | | a | | | | | | | | |
| 9 | | | | | | | | | a | r | a | b | | | | |
| 10 | | | | | | | | | | a | | | | | | |
| 11 | | | | | | | | | | | a | r | a | | | |
| 12 | | | | | | | | | | | | a | | | | |
| 13 | | | | | | | | | | | | | a | | | |
| 14 | | | | | | | | | | | | | | a | r | a |
| 15 | | | | | | | | | | | | | | | a | |
| 16 | | | | | | | | | | | | | | | | a |
| $Z_i$ | – | 0 | 1 | 0 | 7 | 0 | 1 | 0 | 3 | 0 | 2 | 0 | 0 | 3 | 0 | 1 |

**Abbildung 5.6** Z-Blöcke und Präfixwerte $Z_i$ für $T[1..16]$ = `arabarabarartara` (a ... a bedeutet Übereinstimmung mit einem Präfix von $T$, a bedeutet Mismatch).

*Beachte*: Wenn $i+Z_i-1 < n$, dann folgt auf den Z-Block $T[i..i+Z_i-1]$ ein Buchstabe $T[i+Z_i] \neq T[1+Z_i]$. Wenn $i+Z_i-1 = n$, dann reicht der Block an Stelle $i$ bis ans Ende des Wortes, das heißt, dass dieser Block ein Rand von $T$ ist. Der Z-Block an Stelle $i$ hat Länge 0 genau dann, wenn $T[i] \neq T[1]$ gilt.

*Beispiel*: Abbildung 5.6 stellt die Z-Blöcke für $T[1..16] =$ `arabarabarartara` dar und gibt alle Präfixwerte an. Man erkennt, dass auf Z-Blöcke im Inneren des Wortes ein Mismatch-Buchstabe folgt, auf Z-Blöcke am Ende des Wortes nicht – sie sind Ränder von $T$. Man sieht auch, dass es für Z-Blöcke keine Schachtelungsregeln gibt. Jedoch kann man beobachten, dass die Blöcke für $i = 2, \ldots, 7$ im Abschnitt $T[1..7]$ dasselbe Muster bilden wie die Blöcke für $i = 6, \ldots, 11$ innerhalb des Z-Blocks $T[5..11]$.

Wir wollen die Präfixwerte $Z_2, \ldots, Z_n$ berechnen. Natürlich geht das ganz einfach in Zeit $O(n + Z_2 + \cdots + Z_n)$, indem man $T[i..n]$ mit $T[1..n-i+1]$ vergleicht, buchstabenweise von links nach rechts, für jedes $i = 2, \ldots, n$. Wenn wir mit Rechenzeit $O(n)$ auskommen wollen, müssen wir cleverer vorgehen. Aber auch dann ist das „lineare Durchmustern“ (*linear scan*) nötig und hilfreich. Wir formulieren eine entsprechende Prozedur **scan**. Gegeben sind $u$ und $v$ mit $1 \leq u < v \leq n+1$. Die Werte $T[u+z]$ und $T[v+z]$ werden verglichen, für $z = 0, 1, 2, \ldots$, bis entweder ein „Mismatch“ auftritt ($T[u+z] \neq T[v+z]$) oder das Wortende erreicht ist (mit $v + z = n + 1$). In beiden Fällen registrieren wir $z$. Offenbar ist dann $z$ die größte Zahl in $\{0, 1 \ldots, n - v + 1\}$ mit $T[u..u+z-1] = T[v..v+z-1]$. Man beachte den Spezialfall $z = 0$, der auftritt, wenn $v \leq n$ und $T[u] \neq T[v]$ oder wenn $v = n + 1$. Der Mechanismus ist in der in Abb. 5.7 angegebenen Prozedur **scan** zusammengefasst.[6]

**function scan**$(u, v)$ // Vorbedingung: $1 \leq u < v \leq n + 1$
(1) `z` $\leftarrow 0$;
(2) **while** $v$`+z` $\leq n \wedge$ `T[`$u$`+z]` $=$ `T[`$v$`+z]` **do** `z++`;
(3) **return** `z`.

**Abbildung 5.7** Direkter Buchstabenvergleich „von links nach rechts“ ab Stellen $u$ und $v$ mit $1 \leq u < v \leq n + 1$.

Unser Algorithmus berechnet die Werte $Z_i$ nacheinander, in Runden $i = 2, \ldots, n$. Zur Verbesserung der Effizienz wollen wir erreichen, dass jeder Buchstabe von $T[2..n]$ höchstens einmal „erfolgreich verglichen“ wird, das heißt, als zugehörig zu einem Z-Block erkannt wird. In Abb. 5.8 wird der Ablauf veranschaulicht.

Wir betrachten Runde $i$, und nehmen an, dass $Z_2, \ldots, Z_{i-1}$ schon vorliegen. Weiter benötigen wir

$$r := r_{i-1} := \max(\{l + Z_l - 1 \mid 2 \leq l < i, Z_l > 0\} \cup \{0\}), \text{ für } i = 1, \ldots, n,$$

den maximalen Endpunkt eines nichtleeren Z-Blocks, der strikt links von Position $i$ beginnt. Wenn es keinen solchen Z-Block gibt, setzen wir (künstlich) $r = r_{i-1} = 0$. (Insbesondere ist $r_1 = 0$.) Wenn $r \geq 2$ gilt, dann soll $l = l_{i-1} < i$ Startpunkt eines

[6] Wenn man dies tatsächlich programmiert, wird man zur Beschleunigung folgenden Standardtrick verwenden: Der Text $T$ steht in einem Array der Länge mindestens $n + 1$, an Position $n + 1$ steht als „Stopper“ („*sentinel*“) ein Buchstabe, der in $T[1..n]$ nicht vorkommt. Damit entfällt der Test auf das Erreichen des Arrayendes, und Zeile (2) reduziert sich zu **while** `T[`$u$`+z]` $=$ `T[`$v$`+z]` **do** `z++`;

Blocks mit Endpunkt $r$ sein. (Der Algorithmus arbeitet immer mit dem größten solchen $l$ – das ist aber nicht wesentlich.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | Fall |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[i]$ | a | r | a | b | a | r | a | b | a | r | a | r | t | a | r | a | |
| 2 | | a | | | | | | | | | | | | | | | 1 |
| 3 | | | a | r | | | | | | | | | | | | | 1 |
| 4 | | | | a | | | | | | | | | | | | | 1 |
| 5 | | | | | a | r | a | b | a | r | a | b | | | | | 1 |
| 6 | | | | | | a | | | | | | | | | | | 2a |
| 7 | | | | | | | a | r | | | | | | | | | 2a |
| 8 | | | | | | | | a | | | | | | | | | 2a |
| 9 | | | | | | | | | a | r | a | b | | | | | 2b |
| 10 | | | | | | | | | | a | | | | | | | 2a |
| 11 | | | | | | | | | | | a | r | a | | | | 2b |
| 12 | | | | | | | | | | | | a | | | | | 2a |
| 13 | | | | | | | | | | | | | a | | | | 1 |
| 14 | | | | | | | | | | | | | | a | r | a | 1 |
| 15 | | | | | | | | | | | | | | | a | | 2a |
| 16 | | | | | | | | | | | | | | | | a | 2b |
| $Z_i$ | – | 0 | 1 | 0 | 7 | 0 | 1 | 0 | 3 | 0 | 2 | 0 | 0 | 3 | 0 | 1 | |
| $r_i$ | 0 | 0 | 3 | 3 | 11 | 11 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 16 | 16 | 16 | |
| $l_i$ | – | – | 3 | 3 | 5 | 5 | 5 | 5 | 9 | 9 | 11 | 11 | 11 | 14 | 14 | 16 | |

**Abbildung 5.8** Berechnung der Z-Blöcke und Präfixwerte $Z_i$ für $T[1..16]$ = `arabarabarartara` durch Algorithmus 5.4.9. Dargestellt sind: Präfixwerte $Z_i$, Hilfszahlen $r_i$ und $l_i$, Nummern der Fälle im Algorithmus. ( ... bedeutet erfolgreiche Buchstabenvergleiche in **scan**, bedeutet Mismatch in **scan**, ... bedeutet im Fall 2 abgeleitete Übereinstimmung, bedeutet im Fall 2 abgeleitete Mismatchposition.)

**Fall 1:** $r < i$.

Das bedeutet: Keine Position in $T[i..n]$ ist in einem bisher bekannten Z-Block enthalten, d. h., diese Z-Blöcke enthalten keine Information über den Bereich $T[i..n]$. Eventuell ist Stelle $T[i]$ schon einmal oder mehrmals getestet worden, aber es handelte sich dann nur um Mismatch-Ergebnisse. Durch die Festlegung $r_1 = 0$ wird erreicht, dass für die erste Runde, $i = 2$, dieser Fall eintritt.

In diesem Fall finden wir den Z-Block an Stelle $i$ durch eine direkte Durchmusterung, nämlich durch einen Aufruf $z := \mathbf{scan}(1, i)$. Dann ist $T[i..i + z - 1]$ der Z-Block an Stelle $i$, und es gilt $Z_i = z$.

Die Aktualisierung von $r$ und $l$ in Vorbereitung für die nächste Runde $i + 1$ ist auch einfach: Wenn $Z_i = 0$, ist der neu gefundene Z-Block leer, und wir behalten die alten

Werte $r$ und $l$ bei. (In Abb. 5.8 ist dies für $i = 2, 4, 13$ der Fall.) Wenn $Z_i > 0$, ist $T[i..i + Z_i - 1]$ der Z-Block mit Startpunkt $< i + 1$, der am weitesten nach rechts reicht, und wir setzen $r := r_i := i + Z_i - 1$ und $l := l_i := i$. (In Abb. 5.6 geschieht dies für $i = 3, 5, 14$.)

**Fall 2:** $r \geq i$.

Das bedeutet: Position $i$ ist im schon bekannten nichtleeren Z-Block $T[l..r]$ enthalten. Wir benutzen dies, um bei der Ermittlung des Z-Blocks an Stelle $i$ Buchstabenvergleiche einzusparen.

Wir haben $l < i \leq r$. Da $T[l..r]$ ein nichtleerer Z-Block ist, gilt $T[l..r] = T[1..r-l+1]$. Uns interessieren in $T[l..r]$ nur Positionen von $i$ ab nach rechts, daher definieren wir $j := i - l + 1$ (dann ist $2 \leq j < i$) und halten fest:

$$T[i..r] = T[j..r - l + 1]. \tag{5.3}$$

Wir verfolgen nun (in Gedanken, nicht algorithmisch) $T[i + z]$, $T[j + z]$, $T[1 + z]$, für $z = 0, 1, 2, \ldots$, bis entweder ein $z \geq Z_j$ erreicht wird (dann kann man $Z_i$ direkt ablesen, ohne einen einzigen Buchstabenvergleich) oder $i + z$ den Wert $r$ erreicht und (5.3) nicht mehr weiterhilft (dann muss man weitere Buchstaben vergleichen). Es ergeben sich zwei Fälle.

*Fall 2a*: $Z_j \leq r - i$.

Dann gilt $j + Z_j \leq (i - l + 1) + (r - i) = r - l + 1$. Damit liegt der Z-Block an Stelle $j$ inklusive seiner Mismatch-Stelle $j + Z_j$ vollständig im Teilwort $T[j..r - l + 1]$. Weil dieses nach (5.3) mit $T[i..r]$ identisch ist, gilt $T[j..j + Z_j] = T[i..i + Z_j]$. Also haben wir $T[i..i+Z_j-1] = T[j..j+Z_j-1] = T[1..Z_j]$ und $T[i+Z_j] = T[j+Z_j] \neq T[1+Z_j]$. Daraus folgt $Z_i = Z_j$, und wir haben $Z_i$ ohne weiteren Buchstabenvergleich ermittelt. Die Werte $r$ und $l$ werden beibehalten, weil (offensichtlich) das rechte Ende des Blocks an Stelle $i$ bei $i + Z_i - 1 = i + Z_j - 1 \leq i + (r - i) - 1 < r$ liegt.

Im Beispiel in Abb. 5.8 tritt Fall 2a mit $Z_i = 0$ für $i = 6$, 7, 8, 10, 12, 15 ein. Für $i = 7$ haben wir $r = r_6 = 11$ und $l = l_6 = 5$, also $T[l..r] = T[5..11] =$ `arabara`, der Wert $j$ ist $7 - l + 1 = 3$ mit $Z_3 = 1 < 5 = r - 7 + 1$ und wir haben $T[7..8] = T[3..4]$ (= `ab`). Ohne $T[1..2]$ anzusehen, können wir schließen, dass $Z_7 = Z_3$ (= 1) gilt.

*Fall 2b*: $Z_j > r - i$.

Dann ist das (nichtleere) Teilwort $T[j..r - l + 1]$ vollständig im Z-Block an Stelle $j$ enthalten, und wir erhalten:

$$T[1..r - i + 1] = T[j..r - l + 1] \overset{(5.3)}{=} T[i..r]. \tag{5.4}$$

Mit dem Aufruf $z := \mathbf{scan}(r - i + 2, r + 1)$ führen wir weitere direkte Buchstabenvergleiche durch, ab Positionen $r - i + 2$ und $r + 1$ in $T$ nach rechts gehend. Das Ergebnis $z$ gibt die Anzahl der übereinstimmenden Buchstaben an. Also gilt nun $T[r - i + 2..r - i + z + 1] = T[r + 1..r + z]$ und ($r + z = n$ oder $T[r - i + z + 2] \neq$

$T[r+z+1]$). Zusammen mit (5.4) ergibt sich $T[1..r-i+z+1] = T[i..r+z]$. Daher gilt $Z_i = r-i+z+1$. Weil $i+Z_i-1 = r+z \geq r$ und $Z_i \geq 1$ gilt, haben wir $r_i = r+z$, und wir können $l_i = i$ setzen.

Im Beispiel in Abb. 5.8 tritt Fall 2b für $i = 9, 11, 16$ ein. Für $i = 9$ stellt sich in **scan** sofort ein Mismatch ein, für $i = 11$ verlängert sich die Folge der bekannten Buchstaben, im Fall $i = 16$ wird die Schleife in **scan** gar nicht ausgeführt, weil $r + 1 = r_{16} + 1 > 16$ ist.

Wir setzen nun diese Überlegungen in Pseudocode um. Die jeweils relevanten Werte $r$ und $l$ werden in Variablen `r` bzw. `l` aufbewahrt, die nur falls nötig mit neuen Werten überschrieben werden.

**Algorithmus 5.4.9** (Präfixwerte)**.**
**Z-Algorithmus**(`T[1..n]`)
**Eingabe:** `T[1..n]`: Text;
**Ausgabe:** `Z[2..n]`: Vektor der Präfixwerte.

```
(1)   r ← 0;
(2)   for i from 2 to n do
(3)     if i > r
(4)       then      // 1. Fall
(5)         z ← scan(1, i);
(6)         Z[i] ← z;
(7)         if z > 0 then r ← i+z-1; l ← i;
(8)       else      // 2. Fall
(9)         if Z[i-l+1] < r-i+1
(10)          then      // Fall 2a
(11)            Z[i] ← Z[i-l+1];
(12)          else      // Fall 2b
(13)            z ← scan(r-i+2, r+1);
(14)            Z[i] ← r-i+z+1;
(15)            r ← r+z; l ← i;
(16)  return Z[2..n].
```

Zum besseren Verständnis führe man Algorithmus 5.4.9 mit Hilfe von Abb. 5.8 auf der Beispieleingabe `arabarabarartara` aus.

Für die Aufwandsanalyse von Algorithmus 5.4.9 überlegen wir Folgendes: Für jedes $i \in \{2, \ldots, n\}$ werden außerhalb der **scan**-Aufrufe in Zeilen (5) und (13) nur einige Zahlen berechnet und Fallunterscheidungen vorgenommen. Dies zusammen kostet $O(1)$ Zeit für jedes $i$. Wenn ein **scan**-Aufruf erfolgt, dann werden sequentiell Positionen $T[u+z]$ und $T[v+z]$ des Textes mit anderen verglichen, und zwar beginnt $v+z$ bei $\max\{r_{i-1}+1, i\} > r_{i-1}$. Jeder solche Aufruf liefert eventuell Übereinstimmungen („erfolgreiche Vergleiche“) und am Ende ein Mismatch. Wenn ein Aufruf sofort zu einem Mismatch führt, ist er in konstanter Zeit beendet, mit einem Vergleich. Wenn erfolgreiche Vergleiche vorkommen, liegen die übereinstimmenden Positionen $v+z$

im Bereich $\max\{r_{i-1}+1, i\}, \ldots, r_i$. Da die Schleife mit $i = 2$ beginnt und $r_n \leq n$ gilt, gibt es insgesamt maximal $n-1$ solche erfolgreichen Vergleiche, und die **scan**-Aufrufe zusammen haben Rechenzeit $O(n)$. – Wir haben damit die folgende Behauptung bewiesen.

**Proposition 5.4.10.** *Algorithmus 5.4.9 berechnet die Präfixwerte für $T[1..n]$ in Zeit $O(n)$ und führt dabei maximal $2n-2$ Vergleiche zwischen Buchstaben durch.* □

#### 5.4.4.2 Anwendung der Präfixwerte

Wir zeigen hier, dass die Berechnung der Präfixwerte direkt zu einem Linearzeitalgorithmus für Textsuche führt und dass man die Randfunktion und die KMP-Fehlerfunktion eines Musters direkt aus den Präfixwerten berechnen kann.

**Bemerkung 5.4.11.** Der Z-Algorithmus kann unmittelbar zu einem Textsuchalgorithmus mit linearer Laufzeit ausgebaut werden. Wenn Muster $P[1..m]$ und Text $S[1..n]$ gegeben sind, setzt man $T := T[1..n+m] := P \circ S$ (Konkatenation von Muster und Text) und wendet auf $T$ Algorithmus 5.4.9 an. Es ist leicht zu sehen, dass $S[i..i+m-1] = P[1..m]$ genau dann gilt, wenn der Z-Block in $T[1..n+m]$ an Stelle $m+i$ Länge mindestens $m$ hat, d. h., wenn $Z_{m+i} \geq m$ ist. Man berechnet also die Präfixwerte und sucht dann die heraus, die $\geq m$ sind. Dieser Algorithmus würde (wie Algorithmus 5.4.9) alle $n+m$ Präfixwerte speichern. Eine leichte Modifikation dieser Idee erlaubt es sogar, den Platzbedarf auf $O(m+|A|)$ zu senken, wo $A = \{i \mid S[i..i+m-1] = P[1..m]\}$ die Ausgabemenge ist. Man arbeitet mit $T := T[1..n+m+1] := P \circ \# \circ S$, wobei $\#$ ein Buchstabe ist, der in $P \circ S$ nicht vorkommt. Dann gilt $P[1..m] = S[i..i+m-1]$ genau dann wenn $Z_{m+1+i} \geq m$ ist. Wegen des Fremdbuchstabens an Stelle $m+1$ kann kein Z-Block von $T$ mehr als $m$ Buchstaben haben. Das bedeutet, dass für den Wert $j$ im 2. Fall in Abschnitt 5.4.4.1 die Ungleichung $j = i - l + 1 \leq r - l + 1 \leq m$ gilt. Daher werden in Algorithmus 5.4.9 in Zeilen (9) und (11) nur Werte $Z_2, \ldots, Z_m$ abgefragt. Die weiteren Z-Werte $Z_{m+1}, Z_{m+2}, \ldots, Z_{m+n+1}$ werden nur berechnet und mit $m$ verglichen, aber nicht gespeichert.

**Bemerkung 5.4.12.** Wir betrachten ein Muster $P[1..m]$ und überlegen, wie man aus der Folge der Präfixwerte $(Z_2, \ldots, Z_m)$ für $P$ die Randfunktion (als Tabelle `f[0..m]`) berechnen kann, ohne $P$ nochmals anzusehen. Man erinnere sich: Für $0 \leq q' < q \leq m$ heißt $P[1..q']$ ein *Rand* von $P[1..q]$, wenn $P[1..q'] = P[q-q'+1..q]$ gilt. Die *Randfunktion* für $P$ ist durch

$$f_{\text{bord}}(q) := f^P_{\text{bord}}(q) = \begin{cases} -1 & \text{für } q = 0 \\ \max\{q' \mid P[1..q'] \text{ ist Rand von } P[1..q]\} & \text{für } 1 \leq q \leq m \end{cases}$$

definiert. Wir wollen diese Funktion als Tabelle `f[0..m]` berechnen. Offensichtlich muss man `f[0]` $\leftarrow -1$ setzen. Ab hier betrachten wir nur noch $1 \leq q \leq m$. Da $\varepsilon$ Rand jedes nichtleeren Wortes $P[1..q]$ ist, gilt $f_{\text{bord}}(q) \geq 0$ für $1 \leq q \leq m$.

Wir unterscheiden zwei Typen von Rändern $P[1..q']$ für ein $P[1..q]$:

**Typ 1:** (Nicht verlängerbar) $q = m$ oder ($q < m$ und $P[q'+1] \neq P[q+1]$).

**Typ 2:** (Verlängerbar) $q < m$ und $P[q'+1] = P[q+1]$.

Wir definieren:

$$f_1(q) := \max(\{0\} \cup \{q' \mid P[1..q'] \text{ ist Typ-1-Rand von } P[1..q]\}) \text{ für } 1 \leq q \leq m. \quad (5.5)$$

Wir berechnen zunächst $f_1(q)$. Betrachte dazu einen beliebigen Typ-1-Rand $P[1..q']$ von $P[1..q]$, also $P[1..q'] = P[q-q'+1..q]$. Wir möchten, dass $q - q' + 1 \leq m$ gilt. Dies ist im Fall $q < m$ sicher erfüllt. Bei $q = m$ gibt es nur eine Ausnahme, nämlich $q' = 0$. Wenn $P[1..m]$ außer $\varepsilon$ keine weiteren Ränder hat, gilt $f_1(m) = f_{\text{bord}}(m) = 0$. Im Folgenden betrachten wir nur noch nichtleere Ränder $P[1..q']$ von $P[1..m]$. Für $i := q - q' + 1$ haben wir dann $2 \leq i \leq m$. Dass der Rand $P[1..q']$ nicht verlängert werden kann, bedeutet, dass $P[i..q]$ der Z-Block an Stelle $i$ ist und dass $q' = Z_i$ gilt, also $i + Z_i - 1 = q$. Wir erhalten damit:

$$f_1(q) = \max(\{0\} \cup \{Z_i \mid 2 \leq i \leq m \wedge i + Z_i - 1 = q\}).$$

Um das größte $Z_i$ zu erhalten, muss $i$ so klein wie möglich gewählt werden.

Um die Werte $f_1(q)$ zu berechnen, könnten wir für jedes $q$ einzeln die Folge $i = m, \ldots, 2$ durchlaufen, und im Fall $i + Z_i - 1 = q$ die Zuweisung `f[q]` $\leftarrow Z_i$ ausführen. Da das kleinste passende $i$, entsprechend dem größten $Z_i$, zuletzt getestet wird, steht am Ende das größte passende $Z_i$ in `f[q]` (bzw. immer noch 0, wenn es kein passendes $i$ gibt). Man kann diese Berechnung mit einem netten Trick beschleunigen: Initialisiere zunächst `f[1..m]` mit Nullen, für $1 \leq q \leq m$. Da $q = i + Z_i - 1$ aus $i$ und $Z_i$ berechnet werden kann, genügt die einfache Schleife

**for** `i` **from** $m$ **downto** 2 **do** `f[`$i + Z_i - 1$`]` $\leftarrow Z_i$

mit Rechenzeit $O(m)$, um *alle* `f[q]` auf den Wert $f_1(q)$ zu bringen. Wenn $q < m$ gilt und $P[1..q]$ gar keinen Typ-1-Rand hat, oder wenn $q = m$ gilt und $P[1..m]$ keinen nichtleeren Rand hat, steht in `f[q]` immer noch der Initialisierungswert 0.

Nun wollen wir in einem weiteren linearen Durchlauf mit $q = m-1, \ldots, 1$ in `f[1..m]` die korrekten Werte $f_{\text{bord}}(q)$ erzeugen. Dazu überlegen wir Folgendes: Der Wert `f[m]` $= f_1(m)$ ist schon korrekt, da $P[1..m]$ keinen Typ-2-Rand hat. Für $q < m$ gibt es zwei Fälle: (1) Der längste Rand von $P[1..q]$ ist vom Typ 1. Dann gilt $f_{\text{bord}}(q) = f_1(q)$. Weiter gilt $f_{\text{bord}}(q+1) - 1 \leq f_{\text{bord}}(q) = f_1(q)$. (Jeder Rand von $P[1..q+1]$ liefert einen für $P[1..q]$ mit um 1 geringerer Länge.) (2) Der längste Rand von $P[1..q]$ hat Typ 2. Dann gilt $f_1(q) \leq f_{\text{bord}}(q) = f_{\text{bord}}(q+1) - 1$. – In beiden Fällen erhalten wir $f_{\text{bord}}(q) = \max\{f_1(q), f_{\text{bord}}(q+1) - 1\}$. Daher vervollständigt die folgende Schleife die Berechnung der Randfunktion:

**for** q **from** $m-1$ **downto** 1 **do** f[$q$] $\leftarrow \max\{$f[$q$], f[$q+1$] $-1\}$.

Das gesamte Verfahren zur Berechnung der Randfunktion ist in Algorithmus 5.4.13 dargestellt.

**Algorithmus 5.4.13** (Randfunktion aus Präfixwerten)**.**
**bord-from-prefix-numbers**($Z_2, \ldots, Z_m$)
**Eingabe:** ($Z_2, \ldots, Z_m$): Präfixwerte von $P[1..m]$;
**Ausgabe:** f[$0..m$]: Randfunktion von $P[1..m]$ als Tabelle.
(1) f[0] $\leftarrow -1$; **for** q **from** 1 **to** $m$ **do** f[$q$] $\leftarrow 0$;
(2) **for** i **from** $m$ **downto** 2 **do** f[$i + Z_i - 1$] $\leftarrow Z_i$;
(3) **for** q **from** $m-1$ **downto** 1 **do** f[$q$] $\leftarrow \max\{$f[$q$], f[$q+1$] $-1\}$;
(4) **return** f[$0..m$].

Beispiel: Im Beispiel in Abb. 5.6 wird f[11] mit 0 initialisiert und in der Schleife in Zeile (2) erst mit $i = 9$ auf 3, dann mit $i = 5$ auf $f_1(11) = 7$ verändert. Die Schleife in Zeile (3) vergleicht f[11] mit f[12] $-1 = 1$, das Maximum ist 7. Bei f[7] ist der Ablauf der folgende: Initialisierung mit 0, Änderung in Zeile (2) auf 1, mit $i = 7$, Änderung in Zeile (3) auf 3, weil f[8] $= 4$ ist.

**Bemerkung 5.4.14.** Noch viel einfacher als die Berechnung der Randfunktion gestaltet sich die Berechnung der KMP-Fehlerfunktion aus den Präfixwerten $Z_2, \ldots, Z_m$ eines Musters. Erinnerung: $f_{\text{KMP}}(q)$ ist die Länge $q'$ eines längsten Randes $P[1..q']$ von $P[1..q']$ vom Typ 1 (siehe Bem. 5.4.12). Wenn $P[1..q']$ keinen Typ-1-Rand hat, ist $f_{\text{KMP}}(q) = -1$. Um diese Werte zu berechnen, gehen wir wie folgt vor: Wir initialisieren f_KMP[$q$] mit $-1$, für $0 \leq q < m$, und f_KMP[$m$] mit 0 (weil $P[1..m]$ immer den Typ-1-Rand $\varepsilon$ hat). Dann korrigieren wir die Einträge, so dass f_KMP[$q$] die Länge $q'$ des längsten Typ-1-Randes von $P[1..q]$ ist, wenn ein solcher überhaupt existiert. Dafür können wir, wie oben diskutiert, Zeile (2) in Algorithmus 5.4.13 benutzen. Es ergibt sich Algorithmus 5.4.15.

**Algorithmus 5.4.15** (KMP-Fehlerfunktion aus Präfixwerten)**.**
**KMP-from-prefix-numbers**($Z_2, \ldots, Z_m$)
**Eingabe:** ($Z_2, \ldots, Z_m$): Präfixwerte von $P[1..m]$;
**Ausgabe:** f_KMP[$0..m$]: KMP-Fehlerfunktion von $P[1..m]$ als Tabelle.
(1) **for** q **from** 0 **to** $m-1$ **do** f_KMP[$q$] $\leftarrow -1$; f_KMP[$m$] $\leftarrow 0$;
(2) **for** i **from** $m$ **downto** 2 **do** f_KMP[$i + Z_i - 1$] $\leftarrow Z_i$;
(3) **return** f_KMP[$0..m$].

#### 5.4.4.3 Suffixwerte eines Textes

Gegeben sei ein Text $T[1..n]$. Spiegelverkehrt zu Präfixwerten sind „Suffixwerte“ $N_k$, für $1 \leq k < n$.

**Definition 5.4.16.** *Sei $T[1..n]$ ein Text. Für $1 \leq k < n$ definieren wir $N_k$ als die Länge $z$ des längsten Suffixes $T[k-z+1..k]$ von $T[1..k]$, das auch Suffix von $T$ (also gleich $T[n-z+1..n]$) ist.*

Die Suffixwerte $N_k$ zu berechnen ist offensichtlich dasselbe Problem wie das in Abschnitt 5.4.4.1 ausführlich betrachtete *Präfix*werteproblem, nur spiegelverkehrt. Man kann daher das Spiegelwort $T^{\mathsf{R}}[1..n]$ von $T$ bilden, mit $T^{\mathsf{R}}[i] = T[n-i+1]$, für $1 \leq i \leq n$, und dann mit Algorithmus 5.4.9 die Z-Werte $Z_2, \ldots, Z_n$ von $T^{\mathsf{R}}[1..n]$ berechnen. Dann gilt:

$$N_k = Z_{n-k+1}, \text{ für } 1 \leq k < n.$$

Alternativ kann man eine gespiegelte Variante von Algorithmus 5.4.9 benutzen, der Indizes $i = 2, \ldots, n$ durch $k = n - i + 1$ ersetzt. Diese ist in Anhang A.1 dargestellt.

Wir geben in Abb. 5.9 zur Veranschaulichung späterer Verfahren die Suffixwerte und die zugehörigen Blöcke von $T^{\mathsf{R}}[1..16] =$ `aratrarabarabara` an, dem Spiegelbild des Wortes in Abb. 5.6.

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $T[k]$ | a | r | a | t | r | a | r | a | b | a | r | a | b | a | r | a |
| $N_k$ | 1 | 0 | 3 | 0 | 0 | 2 | 0 | 3 | 0 | 1 | 0 | 7 | 0 | 1 | 0 | – |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | a | | | | | | | | | | | | | | | |
| 2 | | a | | | | | | | | | | | | | | |
| 3 | a | r | a | | | | | | | | | | | | | |
| 4 | | | | a | | | | | | | | | | | | |
| 5 | | | | | a | | | | | | | | | | | |
| 6 | | | | a | r | a | | | | | | | | | | |
| 7 | | | | | | | a | | | | | | | | | |
| 8 | | | | | b | a | r | a | | | | | | | | |
| 9 | | | | | | | | | a | | | | | | | |
| 10 | | | | | | | | | r | a | | | | | | |
| 11 | | | | | | | | | | | a | | | | | |
| 12 | | | | | b | a | r | a | b | a | r | a | | | | |
| 13 | | | | | | | | | | | | | a | | | |
| 14 | | | | | | | | | | | | | r | a | | |
| 15 | | | | | | | | | | | | | | | a | |

**Abbildung 5.9** Suffixwerte $N_k$ und zugehörige Blöcke für $T[1..16] =$ `aratrarabarabara` ( ... bedeutet Übereinstimmung mit einem Suffix von $T$, bedeutet Mismatch).

#### 5.4.4.4 Die Berechnung der GS-Werte für ein Muster $P$

Gegeben sei das Muster $P[1..m]$. Wir nehmen an, dass die Suffixwerte $N_k$, $1 \leq k < m$, für $P$ schon vorliegen. Diese werden nun geschickt benutzt, um $\mathrm{GS}(j)$ zu berechnen, für $0 \leq j \leq m$.

Man erinnere sich an Definitionen 5.4.3 und 5.4.4. Danach ist $\mathrm{GS}(j)$ die *kleinste* Zahl $s \geq 1$, die (5.1) oder (5.2) erfüllt. Zur Notationsvereinfachung schreiben wir

$$k = m - s, \text{ also } s = m - k,$$

und formen (5.1) und (5.2) um. Für jedes $j \in \{0, \ldots, m\}$ suchen wir die *größte* Zahl $k < m$, die

$$m - j < k < m \text{ und } P[j+1..m] = P[j-m+k+1..k] \text{ und } P[j] \neq P[j-m+k] \tag{5.6}$$

oder

$$0 \leq k \leq m - j \text{ und } P[m-k+1..m] = P[1..k] \tag{5.7}$$

erfüllt. Zur Illustration siehe die Abbildungen 5.11 und 5.12.

Bei einem „großen“ Shift mit $s \geq j$ gilt $0 \leq k \leq m - j$, und $P[1..k]$ ist ein Rand des Musters. Unter diesen $k$ suchen wir das größte, das Länge $\leq m - j$ hat. Wir definieren:

$$\ell(j) := \max\{k \leq m - j \mid P[1..k] \text{ ist ein Rand von } P\}, \text{ für } 0 \leq j \leq m.$$

Der Wert $j = 0$ ist ein Sonderfall. Da ein längster Rand höchstens $m-1$ Buchstaben haben kann, kommen auch für $j = 0$ nur Werte $k \leq m - 1$ in Frage, und es gilt $\ell(0) = \ell(1)$. Es bleibt die Aufgabe, $\ell(j)$ für $1 \leq j \leq m$ zu berechnen. Wenn $P[1..k]$ ein nichtleerer Rand von $P$ ist, gilt $N_k = k$. (Illustration: $N_1 = 1$ und $N_3 = 3$ in Abb. 5.9.) Wir haben also:

$$\ell(j) = \max\{k \leq m - j \mid k = 0 \text{ oder } N_k = k\}, \text{ für } 1 \leq j \leq m.$$

Wir berechnen $\ell(j)$ für $j = m, m-1, \ldots, 1$ nacheinander. Bei $j = m$ ist die Sache klar: Es kommt nur der leere Rand, also $k = 0$, in Frage, also ist $\ell(m) = 0$. Für $1 \leq j < m$ gibt es zwei Möglichkeiten: Setze $k(j) = m - j$. Wenn $N_{k(j)} = k(j)$ gilt, dann ist $P[1..k(j)]$ ein Rand der Länge $k(j) = m - j$, also gilt $\ell(j) = k(j) = m - j$. Wenn aber $N_{k(j)} < k(j)$ gilt, dann ist das maximale $k \leq m - j$, für das $k = 0$ oder $N_k = k$ gilt, in Wirklichkeit sogar $\leq m - (j+1)$, d. h., wir haben $\ell(j) = \ell(j+1)$.

In Abb. 5.10 ist Pseudocode für die Berechnung der $\ell$-Werte angegeben.

Nun wenden wir uns „kleinen Shifts“ mit $1 \leq s < j$ zu. Damit eine solche Shiftweite in Frage kommt, muss für $k = m - s > m - j$ die Bedingung (5.6) gelten. Wir setzen:

$$L(j) := \max(\{k \mid m - j < k < m \mid k \text{ erfüllt } (5.6)\} \cup \{0\}), \text{ für } 0 \leq j \leq m.$$

(1) `l[`$m$`]` $\leftarrow 0$;
(2) **for** `j` **from** $m-1$ **downto** $1$ **do**
(3) **if** `N[`$m$`-j] = `$m$`-j` **then** `l[j]` $\leftarrow$ $m$`-j` **else** `l[j]` $\leftarrow$ `l[j+1]`.
(4) `l[0]` $\leftarrow$ `l[1]`.

**Abbildung 5.10** Berechnung der $\ell$-Werte

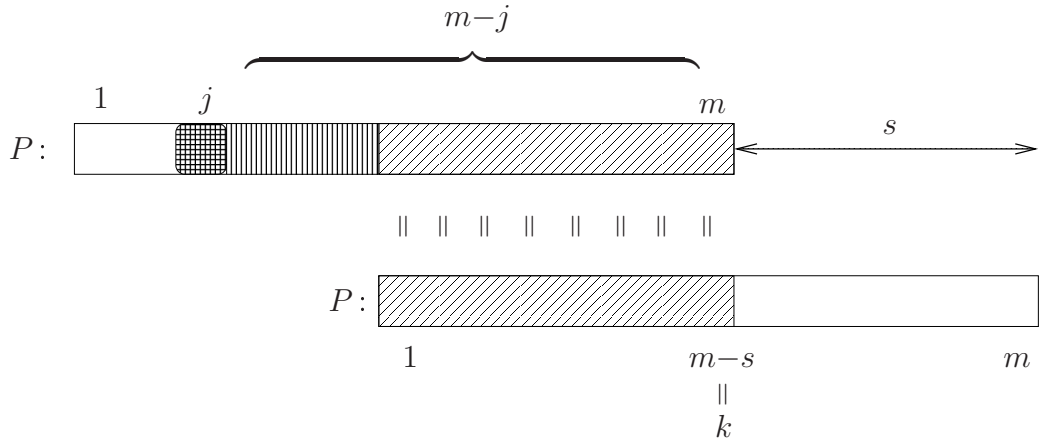


**Abbildung 5.11** Ein „großer Shift“ $s = m - k \geq j$ ist zulässig nach Mismatch an Stelle $j$ genau dann wenn $P[1..k]$ ein Rand von $P[1..m]$ der Länge $\leq m - j$ ist.

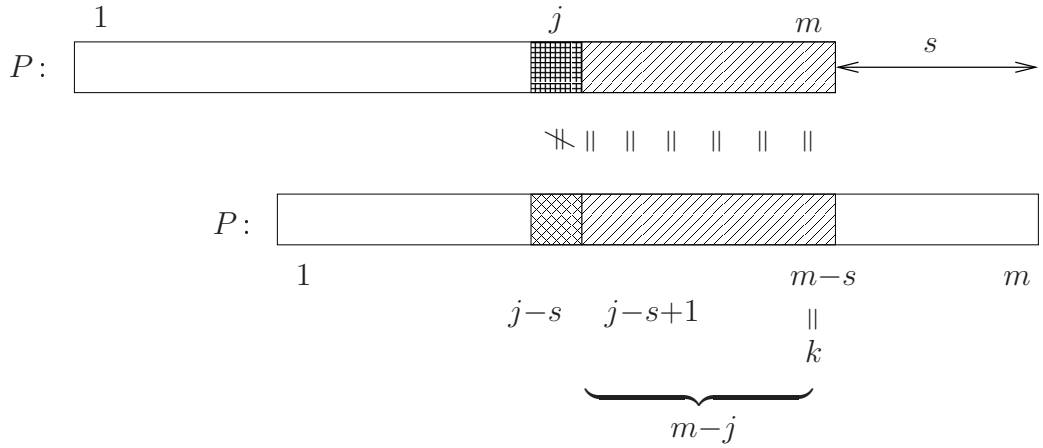


**Abbildung 5.12** Ein „kleiner Shift“ $s = m - k < j$ ist zulässig nach Mismatch bei $j$ genau dann wenn $N_k = m - j$ gilt. Eine kleinste Shiftweite $s$ mit dieser Eigenschaft entspricht einem möglichst großen solchen $k > m - j$. Es muss kein solches $k$ geben.

**Berechnung der $L(j)$:** Es kommt hier darauf an, alle $L(j)$-Werte, von denen jeder eine Maximumsbildung enthält, in Linearzeit zu berechnen. Der zentrale Trick ist folgende Beobachtung, die einen Zusammenhang zwischen den Suffixzahlen $N_k$

und Bedingung (5.6) herstellt, siehe auch Abb. 5.12:

**Lemma 5.4.17.** *Für $m-j<k<m$ gilt* (5.6) *genau dann wenn $N_k = m-j$ ist.*

*Beweis*: Bedingung (5.6) lautet, etwas umgeschrieben:

$$P[j+1..m] = P[k-(m-j)+1..k] \wedge P[j] \neq P[k-(m-j)].$$

Das heißt: Wenn man an der Stelle $P[k]$ startend nach links geht, findet man $m-j$ Buchstaben, die mit $P[m-j+1..m]$ übereinstimmen, der nächste Buchstabe passt nicht mehr. Nach Definition der Suffixzahlen heißt das, dass $N_k = m-j$ ist. □

Die in Abb. 5.13 angegebene Schleife berechnet die Zahlen $L(j)$, gespeichert in einem Array `L[0..m]`, aus den Suffixwerten. Der Werte $L(0) = L(1) = 0$ werden einfach gesetzt.

```
(1)   for j from 0 to m do L[j] ← 0;
(2)   for k from 1 to m − 1 do
(3)     j ← m − N[k]; L[j] ← k;
```

**Abbildung 5.13** Berechnung der L-Werte

Um zu verstehen, was im Programmstück in Abb. 5.13 passiert, betrachtet man ein festes $j$ und beobachtet den Inhalt von `L[j]`. Nach der Initialisierung ist dies 0. Für jedes $k$ mit $N_k = m-j$, in aufsteigender Reihenfolge, wird $k$ nach `L[j]` geschrieben. Am Ende enthält `L[j]` natürlich das maximale solche $k$ (oder immer noch 0, wenn es kein solches $k$ gibt), und dies ist nach der Definition von $L(j)$ und Lemma 5.4.17 genau $L(j)$.

Wir haben nun $\ell(j)$-Werte, $0 \le j \le m$, und $L(j)$-Werte, $0 \le j \le m$. Wie oben schon gesagt, ist dann

$$\mathrm{GS}(j) = m - \max\{\ell(j), L(j)\}, \quad \texttt{für } 0 \le j \le m.$$

Wir fassen das Ganze in einem Programm zur Berechnung der GS-Shiftwerte zusammen. Angenommen ist, dass die Suffixzahlen $N_k$, $1 \le k < m$, des Musters vorliegen.

**Algorithmus 5.4.18** (Berechnung der GS-Shiftwerte)**.**
**BM-GS-Preprocessing**(`N[1..m − 1]`)
**Eingabe:** `N[1..m − 1]`, die Suffixzahlen des Musters `P[1..m]`;
**Ausgabe:** `GS[0..m]`, die GS-Shiftwerte.

```
  // Berechnung der ℓ(j), vgl. Abb. 5.10 :
(1)   l[m] ← 0;
(2)   for k from 1 to m − 1 do
(3)     if N[k]=k then l[m-k] ← k else l[m-k] ← l[m-k+1].
```

(4) `l[0]` $\leftarrow$ `l[1]`;
// Berechnung der $L(j)$, vgl. Abb. 5.13
(5) **for** `j` **from** 0 **to** $m$ **do** `L[j]` $\leftarrow 0$;
(6) **for** `k` **from** 1 **to** $m-1$ **do**
(7) `j` $\leftarrow m -$ `N[k]`; `L[j]` $\leftarrow$ `k`;
// Berechnung der GS($j$):
(8) **for** `j` **from** 0 **to** $m$ **do** `GS[j]` $\leftarrow m$ - `max{l(j),L(j)}`;
(9) **return** `GS`$[0\,..\,m]$.

Wir betrachten den Ablauf des Algorithmus 5.4.18 auf dem Beispielwort `araratarararatar`. Wir stellen uns vor, die $N_k$-Werte liegen schon vor:

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `P[`$k$`]` | a | r | a | r | a | t | a | r | a | r | a | t | a | r |
| $N_k$ | 0 | 2 | 0 | 2 | 0 | 0 | 0 | 8 | 0 | 2 | 0 | 0 | 0 | $-$ |

Die $\ell(j)$-Werte, eingetragen in der Reihenfolge $j = m$, $j = m-k = m-1, m-2, \ldots, m-(m-1) = 1$, dann $\ell(0) = \ell(1)$, sind:

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\ell(j)$ | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 |

Wir überprüfen, dass der Eintrag $\ell(9) = 2$ korrekt ist. Der längste Rand von $P$ mit Länge höchstens $14-9 = 5$ ist `ar`, mit Länge 2. Dagegen gilt $\ell(3) = 8$, denn der längste Rand von $P$ mit Länge höchstens $14-3 = 11$ ist `araratar`, mit Länge 8.
Nun betrachten wir die Berechnung der $L(j)$-Werte. Um zu verdeutlichen, wie diese sich verändern, werden sie untereinander geschrieben, wo eigentlich Überschreiben stattfindet. Man beachte, dass die Zahlen $k = 1, \ldots, m-1$ in dieser Reihenfolge eingetragen werden. Die Initialisierung erfolgt mit 0.

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $L(j)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | 8 | | | | | | 2 | | 1 |
| | | | | | | | | | | | | | 4 | | 3 |
| | | | | | | | | | | | | | 10 | | 5 |
| | | | | | | | | | | | | | | | 6 |
| | | | | | | | | | | | | | | | 7 |
| | | | | | | | | | | | | | | | 9 |
| | | | | | | | | | | | | | | | 11 |
| | | | | | | | | | | | | | | | 12 |
| | | | | | | | | | | | | | | | 13 |

Eintragungen:
$k = 1$ an Position $14 - N(1) = 14 - 0 = 14$,
$k = 2$ an Position $14 - N(2) = 14 - 2 = 12$,
$k = 3$ an Position $14 - N(3) = 14 - 0 = 14$,
$k = 4$ an Position $14 - N(4) = 14 - 2 = 12$,
$k = 5$ an Position $14 - N(5) = 14 - 0 = 14$,
$\vdots$
$k = 8$ an Position $14 - N(8) = 14 - 8 = 6$,
$\vdots$

Die unterste Zahl in Spalte $j$ ist $L(j)$. Man beachte, wie durch die Eintragungsreihenfolge automatisch das Maximum gefunden wird. Eine Ausnahme bilden die $j = m - N_k$, wo $k = m - j$ ist. Diese erfüllen $N_k = k$. Für solche $j$ setzt das Programm `L[`$j$`]` auf $k$. Man beachte, dass dies auch der Wert $\ell(k)$ ist, so dass bei der Maximumsbildung nichts passiert. Im Beispiel sind $j = 6$, $k = 8$ mit $N(8) = 8$ eine solche Konfiguration.
Ergebnis durch Maximumsbildung und Komplementierung:

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\max\{\ell(j), L(j)\}$ | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 2 | 2 | 2 | 2 | 2 | 10 | 0 | 13 |
| $\mathrm{GS}(j)$ | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 12 | 12 | 12 | 12 | 12 | 4 | 14 | 1 |

Wir können einige dieser Shiftwerte interpretieren: $\mathrm{GS}(14) = 1$ heißt: Wenn beim Vergleich im BM-Algorithmus das rechte Ende von $P[1..14]$ unter einem Buchstaben $\neq$ `r` steht, kann man nur um 1 verschieben.
$\mathrm{GS}(13) = 14$ heißt: Wenn $P[14]$ unter einem `r` steht, aber $P[13]$ nicht unter einem `a`, dann passt keine Stelle des Musters hierher (im Muster steht vor jedem `r` ein `a`), also ergibt sich ein großer Sprung von 14.
$\mathrm{GS}(9) = 12$ heißt: Wenn $P[10..14] =$ `ratar` passen, aber $P[9] =$ `a` nicht, dürfen wir um 12 Positionen schieben – die Überlappung zwischen alter und neuer Position des Musters sind nur die beiden letzten Buchstaben `ar`.
$\mathrm{GS}(5) = 6$ heißt: bei einer Übereinstimmung mit dem Suffix $P[6..14] =$ `atararatar`, und einem Fehler bei $P[5] = a$ können wir um 6 Positionen schieben, so dass das Präfix $P[1..8] =$ `araratar` an die Stelle zu stehen kommt, wo vorher das Suffix $P[7..14] =$ `araratar` stand.

Die in Algorithmus 5.4.18 gewählte Notation für die Vorverarbeitung hat den Vorteil, dass man die dahinterliegenden Ideen noch einigermaßen erkennen kann. Es ist aber unnötig, drei Arrays bereitzustellen: eines genügt vollkommen. In ihm werden zuerst die Zahlen $m - \ell(j)$, $0 \leq j \leq m$, berechnet. Hierbei spart man sich durch die Verwendung einer Variablen `ell`, die den laufenden $\ell(m-k)$-Wert enthält, Zugriffe auf das Array und Kopiervorgänge. Danach erfolgt die Korrektur für die $L(j)$-Zahlen wie in Abb. 5.13, wobei auch hier der Eintrag gleich in der komplementären Form $m - k$ erfolgt. Die Minimumsbildung erfolgt dabei automatisch, da die Werte $k$ in aufsteigender Reihenfolge bearbeitet werden. Einträge $L(j) = 0$ können ignoriert werden. Algorithmus 5.4.19 gibt das resultierende Programm an (das natürlich nicht mehr zu verstehen ist).

**Algorithmus 5.4.19** (Berechnung der GS-Shiftwerte direkt in *einem* Array)**.**
**BM-GS-Preprocessing**(`N[`$1\,..\,m-1$`]`)
**Eingabe:** `N[`$1\,..\,m-1$`]`: die Suffixzahlen des Musters `P[`$1\,..\,m$`]`;
**Ausgabe:** `GS[`$0\,..\,m$`]`: die GS-Shiftwerte.

```
  // Berechnung der Werte ℓ(m−k) in ell und der Werte m−ℓ(m−k) in GS[..]:
(1)   ell ← 0;
(2)   GS[m] ← m;
(3)   for k from 1 to m−1 do
(4)     if N[k]=k then ell ← k;
(5)     GS[m-k] ← m - ell;
(6)   GS[0] ← m - ell;
  // Korrektur durch die Werte m−L(j):
(7)   for k from 1 to m−1 do GS[m-N[k]] ← m - k;
(8)   return GS[0..m].
```

# Anhang A

# A

## A.1 Direkte Berechnung von Suffixwerten

Wir geben an, wie man in Analogie zu Algorithmus 5.4.9 die Suffixwerte eines Wortes direkt berechnet, indem man es einmal von rechts nach links liest. Zunächst formulieren wir die gespiegelte **scan**-Funktion:

**function invscan**$(u, v)$ // Vorbedingung: $0 \leq u < v \leq n$
(1) `z` $\leftarrow 0$;
(2) **while** $u$-`z` $\geq 1 \wedge$ `T[`$u$-`z]` $=$ `T[`$v$-`z]` **do** `z++`;
(3) **return** `z`.

**Abbildung A.1** Direkter Buchstabenvergleich $T[u-z] \stackrel{?}{=} T[v-z]$ „von rechts nach links“ ab Stellen $u$ und $v$ mit $0 \leq u < v \leq n$. Die Ausgabe ist die größte Zahl $z \in \{0, 1 \ldots, u\}$ mit $T[u-z+1..u] = T[v-z+1..v]$. (Die Ausgabe $z = 0$ erscheint, wenn $u \geq 1$ und $T[u] \neq T[v]$ gilt oder wenn $u = 0$ gilt.)

**Algorithmus A.1.1** (Suffixwerte eines Textes)**.**
**Suffixwert-Algorithmus**(T[1..$n$])
**Eingabe:** T[1..$n$]: Text;
**Ausgabe:** N[1..$n-1$]: Vektor der Suffixwerte.

```
(1)   l ← n+1;
(2)   for k from n−1 downto 1 do
(3)     if k < l
(4)       then      // 1. Fall, gespiegelt
(5)         z ← invscan(k, n);
(6)         N[k] ← z;
(7)         if z > 0 then l ← k-z+1; r ← k;     // (Fall 1b)
(8)       else      // 2. Fall, gespiegelt
(9)         if N[n-r+k] < k-l+1
(10)          then      // (Fall 2a)
(11)            N[k] ← N[n-r+k];
(12)          else      // (Fall 2b)
(13)            z ← invscan(l-1, n-r+l-1);
(14)            N[k] ← k-l+z+1;
(15)            l ← l-z; r ← k;
(16)  return N[1..n−1].
```

Die Tabelle in Abb. A.2 gibt die Suffixwerte für den Text `araratararatar` der Länge 14 an.

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T[$k$] | a | r | a | r | a | t | a | r | a | r | a | t | a | r |
| $N_k$ | 0 | 2 | 0 | 2 | 0 | 0 | 0 | 8 | 0 | 2 | 0 | 0 | 0 | – |

**Abbildung A.2** Die Suffixwerte für $T[1..14]$ = `araratararatar`.