Technische Hochschule Rosenheim
Technical University of Applied Sciences

Algorithmen und Datenstrukturen
Kapitel 7: String Matching

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Hinweis: Ein Teil der Folien ist angelehnt an [4]

String Matching

- ***Definition***
  - Finde Muster der Länge ***m*** in einem Text der Länge ***n***.
  - Meist ***n*** viel größer als ***m***.
  - Beispiel

- ***Anwendungen***
  - Texteditor: `Strg+F`
  - Java: `indexOf` der Klasse `String`
  - Durchsuche Speicher, Festplatte nach Signaturen: `grep`
  - Überwachung: Suche in Internetverkehr nach verdächtigen Bitmustern.
  - Spamerkennung
  - Extrahiere relevante Daten aus einer Webseite.

Publikums-Joker: Boyer-Moore

Gegeben sei der folgende String. Was liefert indexOf zurück?

```
String text = "HalloHallo"
```

A. `text.indexOf("ll") → 2`

B. `text.indexOf("ll") → 3`

C. `text.indexOf("ll") → 7`

D. `text.indexOf("ll") → 8`

Übersicht

- **Brute-Force**
- Boyer-Moore
- Rabin Karp

Brute-Force Ansatz

- ***Idee***
  - Prüfe für jede Textposition ***i***, ob an dieser Stelle das Muster steht.
  - ***i:*** Zeiger in den Text, ***j:*** Zeiger auf Muster

- ***Laufzeit***
  - Worst Case: Vergleich von ***mn*** Zeichen
  - Meist deutlich besser als Worst Case!

Suche das Master `pat` im String `txt`
Rückgabe: Textposition an der Muster steht, `n` falls Muster nicht enthalten.

```
BRUTE-FORCE(String txt, String pat)
1    n = txt.length
2    m = pat.length
3    for (i= 0; i ≤ n − m; i++)
4       for (j= 0; j < m; j++)
5          if (txt.charAt(i+j) != pat.charAt(j))
6             break;
7    if (j ==m)
8       return i
9    return n
```

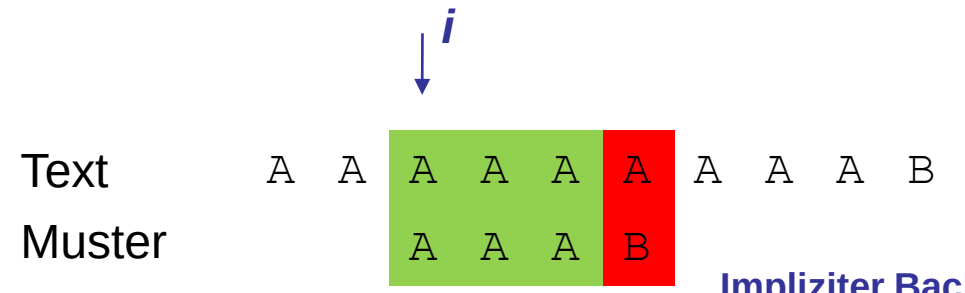
Quellcode: BruteForce.java

String Matching – Ziele

- ***Lineare Laufzeit:*** Annähernd ***n*** Vergleiche (in den meisten Fällen), nicht ***mn*****!**

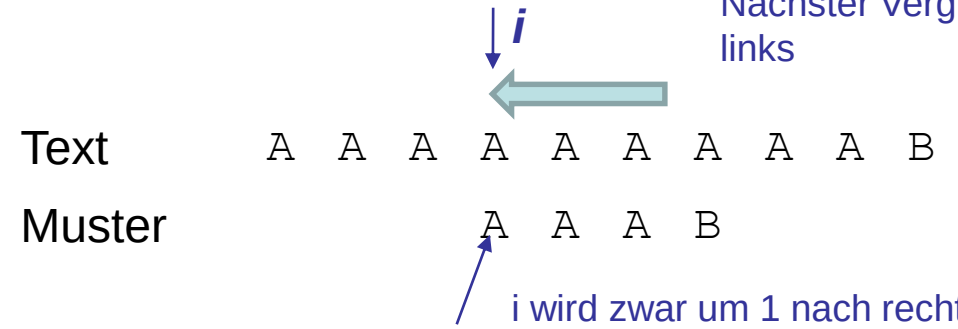
- ***Kein Backup*** (dt. "Zurückspringen") bei Mismatch
  - Backup: Betrachte Zeichen, das weiter links steht als gerade verglichenes Zeichen.
  - Backup besonders teuer, wenn nur sequentiell gelesen werden kann (Dateien, Streams). Dort oft Puffer, um die letzten *m* Zeichen zu speichern
  - Beispiel: Suche Muster `A A A B`

***i*=2:** Die ersten 3 Zeichen "matchen". Mismatch bei *j*=3



Impliziter Backup:
Nächster Vergleich weiter links

Um Vergleich für *i*=3 zu starten → **Backup**



i wird zwar um 1 nach rechts verschoben.
Dieses Zeichen wurde aber schon "gelesen"

Übersicht

- Brute-Force
- **Boyer-Moore**
- Rabin Karp

Algorithmus von Boyer-Moore (1977)

- ***Ziel***
  - Algorithmus, der „häufig" mit $O(\frac{n}{m})$ Vergleichen auskennt.

- ***Idee: „Mismatch Character Heuristik“***
  - Lese den Text von links nach rechts.
  - Vergleiche aber das Muster mit dem aktuellen Text ***rechts nach links!***
  - Bei einem Mismatch kann man ggfs. mehr als ***m*** Textzeichen weiterspringen.

Idee an einem Beispiel

1. Lege das Muster auf den Anfang, vergleiche das ***rechte*** Zeichen des Musters „*E*“ mit dem „*N*“. ***Mismatch!***

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Text	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
Muster	N	E	E	D	L	E																		

2. *N* Teil des Musters. Verschiebe Muster gleich um 5 Zeichen, so dass das "rechteste" *N* im Muster mit aktuellen Textzeichen übereinstimmt. Nun wieder ***Mismatch*** *der* ***rechten*** *Zeichen*!

F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
					N	E	E	D	L	E														

3. S überhaupt nicht Teil des Musters. Verschiebe Muster um 6 Zeichen (=Musterlänge). Vergleiche ***von rechts nach links***. Die letzten beiden Zeichen "matchen".

F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
											N	E	E	D	L	E								

4. Vergleiche von rechts nach links. Mismatch bei *N* im Text. Schiebe Muster nach rechts, so dass rechtestes *N* mit *N* des Texts ausgerichtet ist.

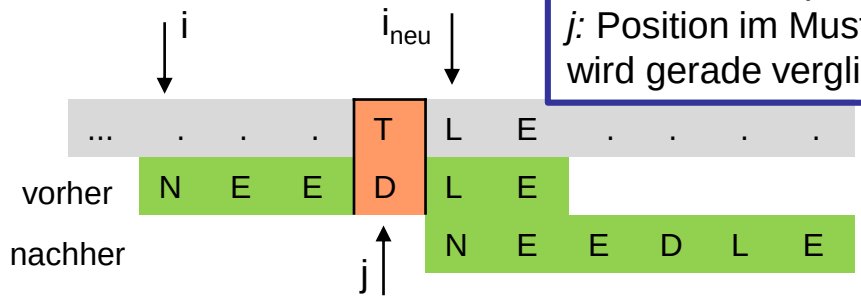
Match!

F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
															N	E	E	D	L	E				

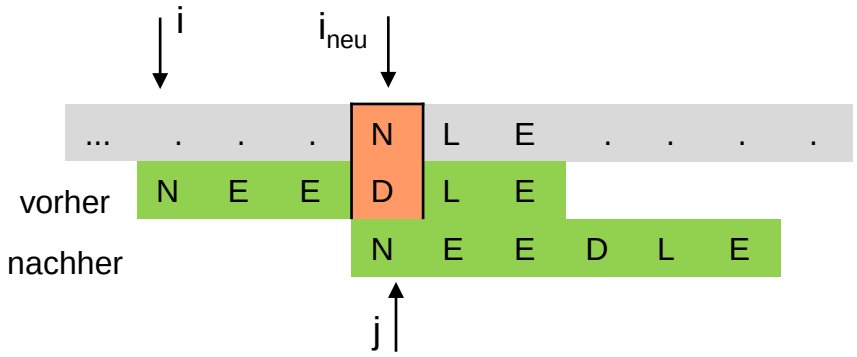
Wie das Muster weiter schieben?

i: Aktuelle Textposition
j: Position im Muster, was wird gerade verglichen?

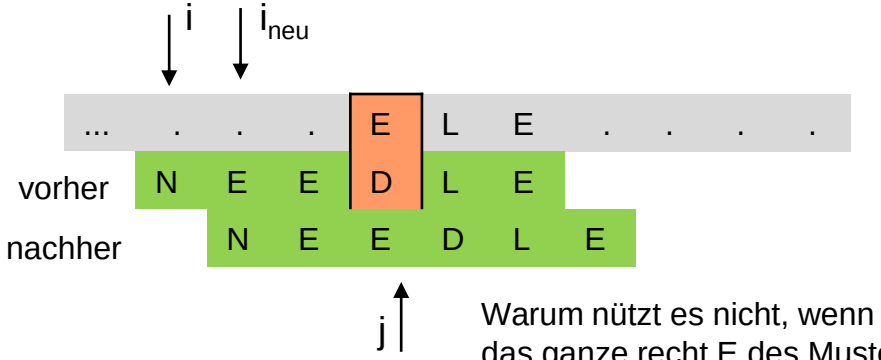
Fall 1: Textzeichen, das nicht mit Musterzeichen übereinstimmt („***Mismatch Character***“ *T*), ist nicht im Muster → Erhöhe *i* um *j*+1, d.h. setze Muster hinter Mismatch



Fall 2a: Mismatch Character *N* ist Teil des Musters, Heuristik hilfreich!
→ Richte das am weitesten rechts liegende *N* des Musters an diesem N aus.



Fall 2b: Mismatch Character *E* ist Teil des Musters. Heuristik nicht hilfreich, da das „letzte Vorkommen von E rechts des Mismatches liegt“
→ Erhöhe einfach *i* um 1



Warum nützt es nicht, wenn man das ganze recht E des Musters am Mismatch Character E ausrichtet?

Erstes Vorkommen von rechts?

- Effizientes Bestimmen des letzten Vorkommens eines Zeichens im Muster?
  - „Rechteste“ Position eines Zeichens im Muster
  - Benötigt für Fallunterscheidung, siehe vorher
- Idee: Berechne das ***vorab*** und nur ***einmal*** im Array `right` für jedes Zeichen.
  - Konvention: -1 falls Zeichen nicht vorkommt.
  - Array Index ist ASCII-Wert des Characters.
- Beispielmuster: N E E D L E

Berechne `right` für den Muster-String `pat`

```
1    for c = 0 to R
2       right[c] = -1
3    for j = 0 to m
4       right[pat.charAt(j)]=j
```

Quellcode: BoyerMoore.java

Muster **`pat`** *m*

Index	0	1	2	3	4	5
	N	E	E	D	L	E

Ergebnisarray **`right`**

Index	'A'	'B'	'C'	'D'	'E'	...	L	M	N	...
	-1	-1	-1	3	5		4	-1	0	

Boyer-Moore: Implementierung

Suche das Master `pat` im String `txt`

```
BOYER-MOORE(String txt, String pat)
1    … compute array right, see previous slide
2    n = txt.length
3    m = pat.length
4    for (i = 0;  i ≤ n − m;  i += skip)
5       skip = 0
6       for (j = m − 1;  j ≥ 0;  j--)
7          if (pat.charAt(j) != txt.charAt(i+j))
8             skip = j − right[txt.charAt(i+j)]
9             if (skip < 1)
10               skip = 1    // Fall 2b
11            break
12      if (skip == 0)
13         return i
14   return n
```

Quellcode: BoyerMoore.java

← Keine Übereinstimmung an Position i + j des Textes (implizit Fall 1 und 2a)

← Muster im Text gefunden

← Muster nicht gefunden

Animation: https://people.ok.ubc.ca/ylucet/DS/BoyerMoore.html

Boyer-Moore: Laufzeit

- Erinnerung
  - *n*: Textlänge
  - *m*: Musterlänge
- Laufzeit
  - Fast immer: $\boldsymbol{O(\frac{n}{m})}$
  - Worst Case: $\boldsymbol{O(mn)}$
- Wann ist Boyer-Moore ungünstig? → Publikumsjoker
- Hinweis
  - Original-Algorithmus führt zu garantiert linearer Laufzeit, zusätzliche Heuristiken.
  - Hier wurde nur die *Mismatch Character Heuristik* diskutiert.
- GNU grep/frgrep verwendet teilweise Boyer-Moore
  - https://github.com/c9/node-gnu-tools/tree/master/grep-src#

Publikums-Joker: Boyer-Moore

Welche der folgende Fälle ist am ungünstigsten bzgl. der Laufzeit?

A. Muster "BAAA", Text "BBBBBBB…"

B. Muster: "ABBB", Text "BBBBBBB…"

C. Muster: "ABAB", Text "BBBBBBB…"

D. Muster: "AAAA", Text "BBBBBBB…"

Übersicht

- Brute-Force
- Boyer-Moore
- **Rabin Karp**

Algorithmus von Rabin-Karp (1987)

- ***Ziel***
  - Schneller als Brute-Force
  - Kein zusätzlicher Speicher wie Boyer-Moore.
  - Erweiterbar auf 2D-Muster bzw. auf das Finden mehrerer Muster.

Rabian, Karp

- ***Idee 1:*** Interpretiere einen String der Länge l als Zahl mit l Stellen und der Basis R
  - *R* hängt ab von Größe des Zeichensatzes
  - ISO 8859-1 / erweitertes ASCII: $R = 256$ Zeichen
  - Beispiel 1: "A" → 65 (siehe ASCII Tabelle)
  - Beispiel 2: "12" → $1 \cdot 256 + 2 = 258$
  - **Der Einfachheit gehen die folgenden Spielbeispiele davon aus, dass ein String nur aus den Ziffern 0 bis 9 besteht und für die Basis somit gilt: $R = 10$**

- ***Idee 2***: Vergleiche 2 Strings durch den Vergleich ihrer Hashwerte (== Fingerprints)

Rabin-Karp: Idee

- Berechne für Muster-String `pat[0..(m-1)]` einen Hashwert (***Modulo q)***
  - $(2 \cdot 10^4 + 6 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0) \bmod 997 = 613$
  - Hier *q* = 997
  - Hier: *R* = 10

Muster **`pat`**

Index	0	1	2	3	4	
Zeichen	2	6	5	3	5	% 997 = **613**

- Für jede Textposition `i`
  - Berechne nach gleichem Verfahren Hashwert für `txt[i..(m+i)-1]`
  - Bei Übereinstimmung mit Muster-Hashwert, prüfe ob Teilstring auch tatsächlich vorliegt.

Text **`txt`**

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Zeichen	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
	3	1	4	1	5	% 997 = 508											
		1	4	1	5	9	% 997 = 201										
			4	1	5	9	2	% 997 = 715									
				1	5	9	2	6	% 997 = 971								
					5	9	2	6	5	% 997 = 442							
						9	2	6	5	3	% 997 = 929						
							2	6	5	3	5	% 997 = **613**					

Wie berechnet man den Hashwert?

- ***Problem***
  - Lange Strings → Zahlen werden sehr groß.
  - Beispiel: String hat $l = 1000$ Zeichen
    - Ergibt Zahl der Größenordnung 10^{1000}
    - Kann nicht in Datentyp `long` gespeichert werden.

- ***Trick:*** "***Horner-Schema***" ähnlich wie bei Polynomen
  - Multipliziere jede Stelle mit R (hier: $R = 10$, statt $R = 256$), füge die nächste Stelle hinzu und rechne Modulo.
  - Beispiel: 2132, $R = 10$, q $= 997$
    - Anstatt $(2 \cdot 10^3 + 1 \cdot 10^2 + 3 \cdot 10^1 + 2) \bmod 997 = 138$
    - rechne $((((2 \cdot 10 + 1) \bmod 997) \cdot 10 + 3) \bmod 997) \cdot 10 + 2) \bmod 997 = 138$
  - Laufzeit für String der Länge *l*?

```
HASH(s, l)
1    h = 0
2    for j=0 to l-1
3        h = (R * h + s.charAt(j)) % q
4    return h
```

Berechne Hashwert für String `s` der Länge `l`

R: Größe des Zeichensatzes
q: Modulowert

Quellcode: RabinKarp.java

Effiziente Berechnung der Hashwerte für Text

- Wie oft muss Hashwert berechnet werden?
  - Für Muster: 1mal, Vorabberechnung möglich
  - Für Text: $\boldsymbol{O(n)}$ mal, für jede Textposition

Auszug aus Text

Aktueller Wert	4	1	5	9	2	6	5
Neuer Wert	4	1	5	9	2	6	5

- ***Notation***
  - t_i: Zeichen an Position `i` des Textes, `txt.charAt(i)`
  - $x_i = t_i R^{m-1} + t_{i+1} R^{m-2} + \cdots + t_{i+m-1} R^0$
  - $x_{i+1} = t_{i+1} R^{m-1} + t_{i+2} R^{m-2} + \cdots + t_{i+m} R^0$

- Wie berechnet man Hashwert für Position `i+1` aus Position `i`?

- ***Trick: "Rollende" Hashfunktion***
  - $x_{i+1} = (x_i - t_i R^{m-1}) \cdot R + t_{i+m}$

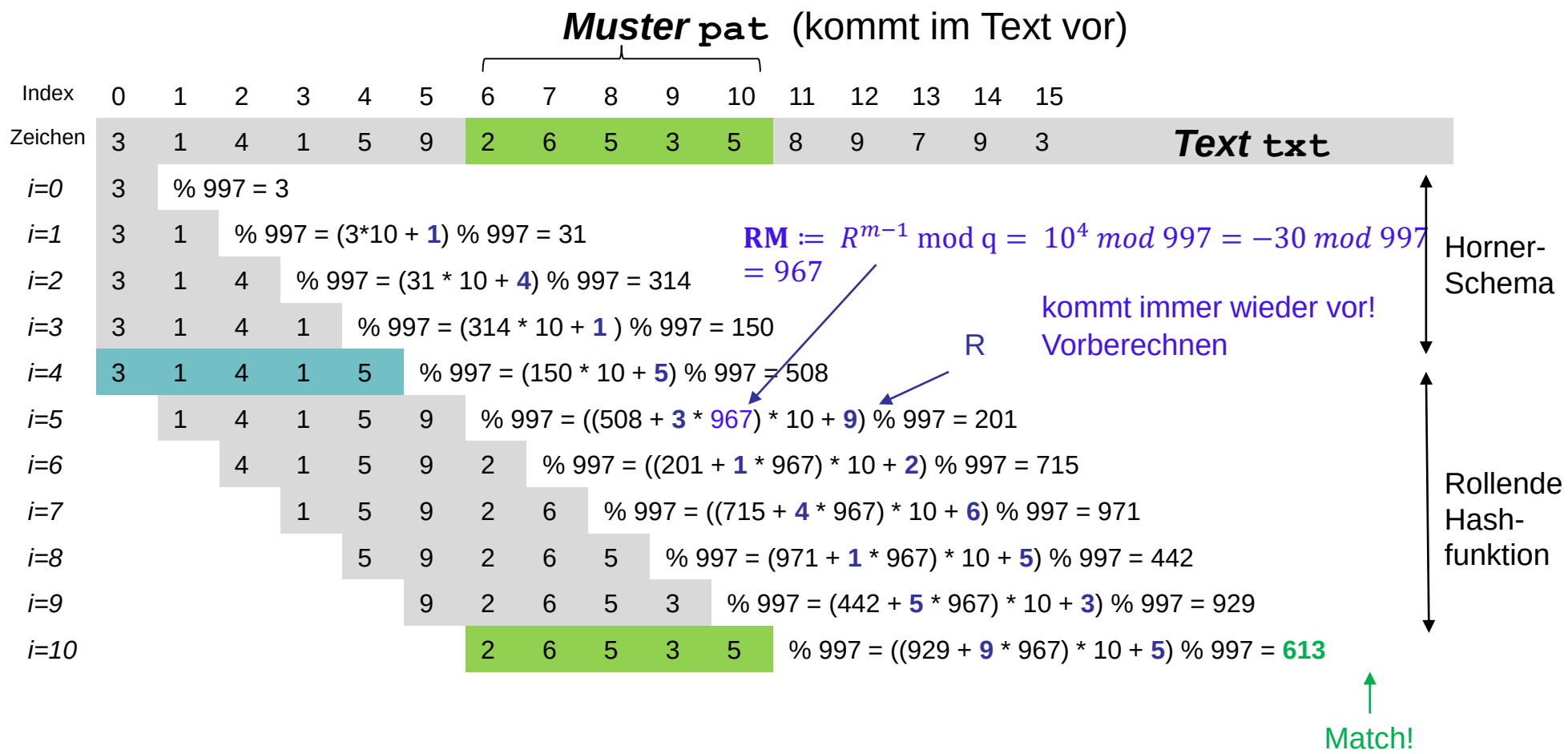
Aktueller Wert — Ziehe führende Stelle ab — Multipliziere mit Basis — Füge eine Stelle weiter rechts hinzu

Berechnung neuer Wert aus aktuellem Wert

Aktueller Wert	4	1	5	9	2
-	4	0	0	0	0
		1	5	9	2
			*	1	0
	1	5	9	2	0
				+	6
Neuer Wert	1	5	9	2	6

Laufzeit?

Rabin-Karp: Beispiel



- Horner-Schema für die ersten R Vergleiche
- Rollende Hashfunktion für die verbleibenden Vergleiche
  - $x_{i+1} = (x_i - t_i R^{m-1}) \cdot R + t_{i+m}$

Publikums-Joker: Rabin-Karp

Was ist korrekt?

A. Rabin-Karp eignet sich für den Vergleich mit mehreren Mustern.

B. Rabin-Karp funktioniert nur wenn die Zeichen Zahlen sind.

C. Rabin-Karp ist effizient, unabhängig von der Wahl der Hashfunktion (kleines q).

D. Rabin-Karp ist schneller als Boyer-Moore.

Rabin-Karp: Monte-Carlo Version

Suche das Master `pat` im String `txt` (`R`: Größe des Zeichensatzes, `q`: modulo-Wert)

```
RABIN-KARP(String txt, String pat)
1    n = txt.length
2    m = pat.length
3    RM = 1
4    for (i= 1; i < m; i++)
5       RM = (R * RM) % q
6    txtHash = hash(txt, m)
7    patHash = hash(pat, m)
8    if (patHash == txtHash)
9       return 0
10   for (i= m; i < n; i++)
11      txtHash = (txtHash + q –
12                   RM * txt.charAt(i-m) % q) % q
13      txtHash = (txtHash * R + txt.charAt(i)) % q
14      if (patHash == txtHash)
15         return i – m + 1
16   return n
```

Quellcode: RabinKarp.java

Berechne R^{m-1} % q → RM;
q wird häufig zufällig gewählt.

Hashwert der ersten m Textzeichen,

Muster-Hashwert

Muster im Text gefunden

"Rollende Hashfunktion"

Eigentlich müsste man nun
noch die Strings abgleichen!

- **Achtung:** Algorithmus liefert falsches Ergebnis, wenn `patHash` und `txtHash` verschieden sind, aber den gleichen Hashwert ergeben (Kollision)

Rabin-Karp: Monte Carlo vs. Las Vegas

- **Randomisierter Algorithmus**
  - Der Modulo-Wert q wird meist zufällig gewählt

- Wie reagiert man, falls sich in Zeile 14 zwei identische Hashwerte ergeben?

- **Variante 1 / Monte Carlo**
  - Gleiche Text nicht mit Muster ab.
  - Annahme: Gleicher Hashwert → gleiche Strings
  - ***Monte Carlo*** == Randomisierter Algorithmus, der nur mit sehr geringer Wahrscheinlichkeit ein falsches Ergebnis liefert.

- **Variante 2 / Las Vegas**
  - Verifiziere, dass Text wirklich mit Muster übereinstimmt.
  - Algorithmus immer korrekt, aber etwas längere Laufzeit.
  - ***Las Vegas*** == Randomisierter Algorithmus, der immer ein korrektes Ergebnis liefert.

String Matching – Diskussion

Algorithmus	Version	Worst Case	Average Case	Korrekt?	Speicher
Brute Force		mn	$1{,}1n$	ja	1
Boyer-Moore	Mismatch Character Heuristik	mn	n/m	ja	R
Rabin-Karp	Monte Carlo	$7n$	$7n$	jein*	1
Rabin-Karp	Las Vegas	$7n$	$7n$	Ja	1

* Bei Einsatz spezieller Hashfunktionen dennoch immer korrekt

- ***Brute-Force***
  - Einfach, funktioniert meist gut.
  - `indexOf()` Methode der Java-Klasse `String`.
- ***Boyer-Moore***
  - Schneller als linear
  - Bsp.: GNU Grep, falls nur nach einem Muster gesucht wird.
- ***Rabin-Karp***
  - Gut, wenn man nach mehreren Mustern oder 2D-Mustern sucht.
  - Nachteil: Arithmetische Vergleiche langsamer als Vergleiche von `char`.

- Jeder Algorithmus hat seine Vor- und Nachteile. Es gibt noch viele weitere!
  - http://www-igm.univ-mlv.fr/~lecroq/string/

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 1.2.3, 5. Auflage, Spektrum Akademischer Verlag, 2012. (xxx)

[3] https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/05DemoQuickSelect.pdf (abgerufen am 03.11.2016)

[4] Sedgewick, Robert. Algorithms, Chapter 5.3, https://algs4.cs.princeton.edu/53substring/ (abgerufen am 10.09.2019)