

Suchen in Texten

- Sie wissen wie in einem Text gesucht werden kann
- Sie wissen wie Suchmaschinen arbeiten
- Sie können mit Regex in Java umgehen

Basiert auf Material von:

Kurt Bleisch

Stephan Neuhaus

Karl Rege

Marcela Ruiz

Jürgen Spielberger





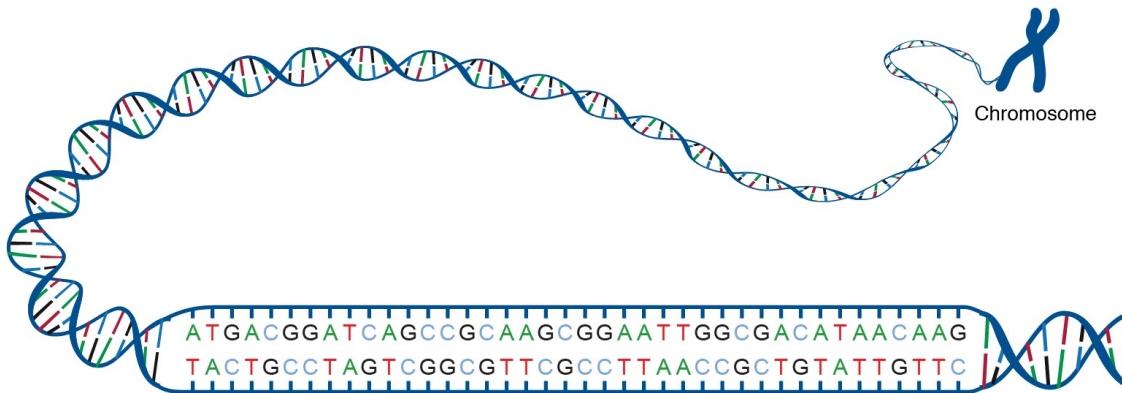
Suchen in Zeichenketten

Suchen in Texten: Motivation

Wir suchen im Genom des Menschen (ca. 3.27 Milliarden Basenpaare, Informationsgehalt von ca. 6,54 Milliarden Bit, ~1 GB) ein bestimmtes Muster (Pattern), z.B.:

TACTGCCTAGTCGGCGTTTCGCCTTAACCGCTGTATTGTTC

Wie können wir dieses effizient finden?



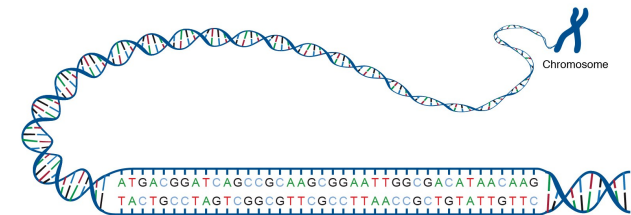
Suchen in Texten: Java

Java bietet zum Suchen von Pattern in Text (String, StringBuilder oder StringBuffer) die Methode `indexOf()`.

```
int indexOf(int ch)
int indexOf(String str)
int indexOf(String str, int from)
```

ch Unicode-Code des gesuchten Zeichens
str gesuchter Substring
from Index ab welchem das Zeichen/Substring gesucht wird

Position, an der das Zeichen
resp. Pattern beginnt. -1 falls
nicht gefunden.



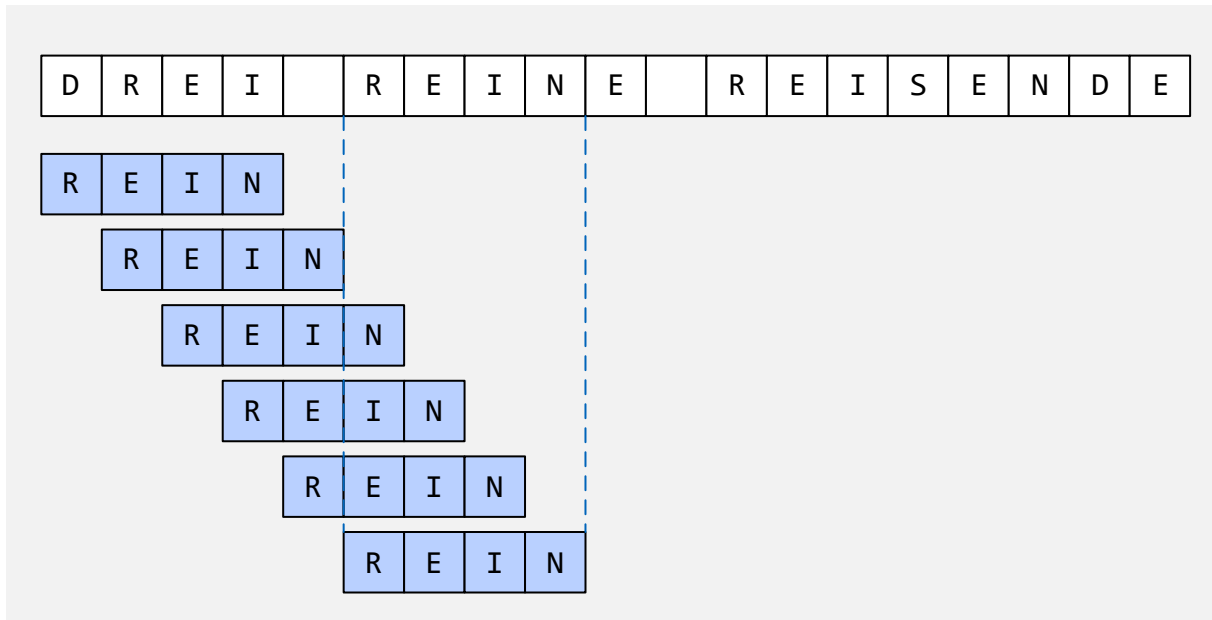
Die Suche dauert ein paar Sekunden, für eine einzige Suche noch in Ordnung – aber wir haben ein grundsätzliches Problem. Die maximale Grösse von String und Array ($2^{31}-1$, 2'147'483'647) ist zu klein um alle Buchstaben des Genoms zu speichern.

Suchen in Texten: Erste Idee

Pattern wird startend von Position 0 nach und nach verschoben:

Pattern wird jeweils mit dem String verglichen bis:

- Ende des Pattern erreicht → Erfolg
- Nichtübereinstimmung → Nächste Position des Pattern ausprobieren



Suchen in Texten: Brute-Force

Worst-Case Aufwand ist $O(n \cdot m)$ (n = Anz. Zeichen String, m = Anz. Zeichen Pattern).

```
static int indexOf(String str, String pattern) {  
    for (int i = 0; i < str.len() - pattern.len() + 1; i++) {  
        for (k = 0;  
            k < pattern.len() && str.charAt(i + k) == pattern.charAt(k); k++) ;  
        if (k == pattern.len()) return i;  
    }  
    return -1;  
}
```

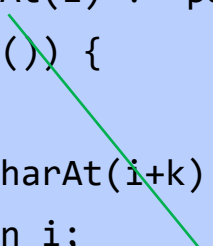
-1 falls nicht
gefunden.

Leere Anweisung: es geht
nur darum zu
überprüfen, ob k bis zur
Länge des Patterns
erhöht werden kann.

Suchen in Texten: Brute-Force die Zweite

Eine einfache Verbesserung: Es wird, vor dem Ausführen der 2. Schleife, der erste Buchstabe gesucht.

```
static int indexOf(String str, String pattern) {  
    int k;  
    for (int i = 0; i < str.len() - pattern.len() + 1; i++) {  
        while (i < str.len() && str.charAt(i) != pattern.charAt(0)) i++;  
        if (i + pattern.len() <= str.len()) {  
            for (k = 0;  
                k < pattern.len() && str.charAt(i+k) == pattern.charAt(k); k++) ;  
            if (k == pattern.len()) return i;  
        }  
    }  
    return -1;  
}
```



Sucht den ersten
übereinstimmenden
Buchstaben.

- indexOf()-Methode in Java verwendet diesen Algorithmus.
- Aufwand $O(m \cdot n)$ (m = Länge str, n = Länge pattern).

Suchen in Texten: Brute-Force die Dritte

Weitere Idee: i könnte, nachdem der Patternvergleich erfolglos war, nicht nur um eins, sondern um die Länge des abgesuchten Strings erhöht werden.

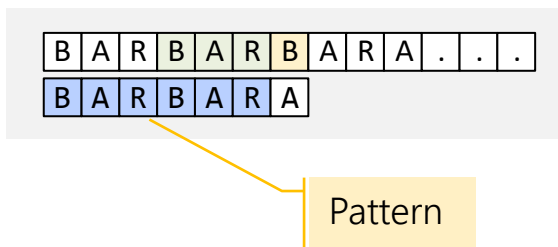
```
static int indexOf(String str, String pattern) {  
    int k;  
    for (int i = 0; i < str.len() - pattern.len() + 1; i++) {  
        while (i < str.len() && str.charAt(i) != pattern.charAt(0)) i++;  
        if (i + pattern.len() <= str.len()) {  
            for (k = 0;  
                k < pattern.len() && str.charAt(i+k) == pattern.charAt(k); k++) ;  
            if (k == pattern.len()) return i;  
        }  
        i = i + k;  
    }  
    return -1;  
}
```

Wir durchsuchen den in der zweiten Schleife untersuchten String str nicht noch mal.

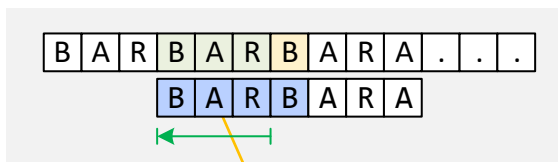
- ABER: Sich wiederholende Teile im Pattern führen zu Problemen → wir müssen intelligent verschieben: Knuth-Morris-Pratt-Algorithmus.

Knuth-Morris-Pratt Algorithmus

Idee des KMP-Algorithmus: bei einem Nicht-Match das Pattern um mehr als eine Stelle verschieben, so dass allfällig auftretende Subpattern am Anfang des Patterns erkannt werden. Beispiel:



Beim orange hinterlegten B kommt es beim Brute-Force-Verfahren zum Abbruch der inneren Schleife, dabei haben wir eigentlich an dieser Stelle schon wieder das Subpattern (Präfix) BAR am Anfang des Patterns «erkannt» (grün hinterlegt).



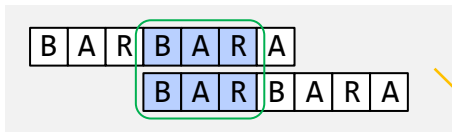
Wir müssen für jeden Buchstaben im Pattern festlegen, welcher Präfix, trotz eines Konfliktes, schon erfolgreich geprüft wurde. Diese Information legen wir in der Next-Tabelle ab.

Tritt der Konflikt nach dem 6 Buchstaben (beim 7.) auf, dann haben wir schon 3 Buchstaben erfolgreich verglichen und der Vergleich kann an Position 4 des Patterns fortgesetzt werden: $\text{Next}[6] = 3$.

Knuth-Morris-Pratt Algorithmus

Ablauf Algorithmus in 2 Phasen:

1. Vorbereitung: Am Anfang wird im Pattern nach sich wiederholenden Subpattern gesucht → Next-Tabelle.



Next[6] = 3

Tritt der Konflikt nach der 6 Stelle auf, darf das Pattern um 3 nach links verschoben angesetzt werden.

2. Suche: Text gemäss der Next-Tabelle durchsuchen.

Auf den folgenden Folien wird das Vorgehen im Detail gezeigt.

Knuth-Morris-Pratt Algorithmus

1. Phase: Sich wiederholende Subpattern suchen:

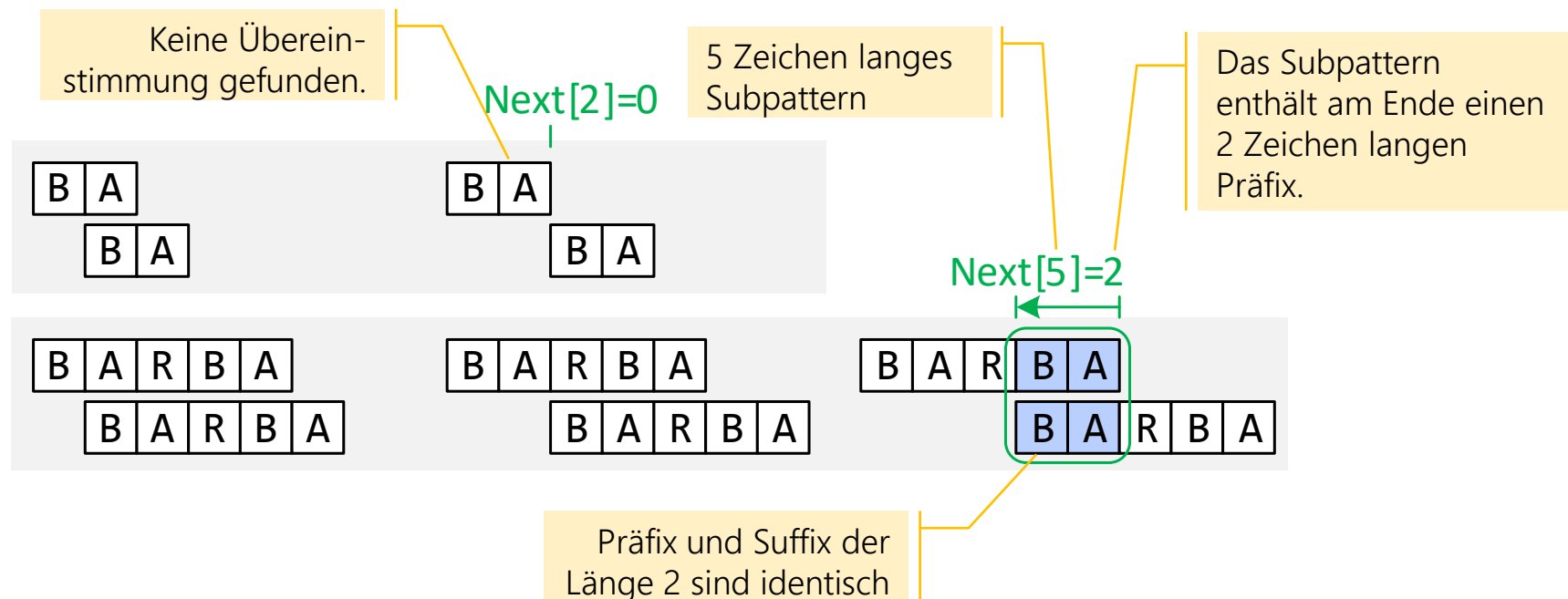
1.1 Es werden Subpattern (mögliche Präfixe) mit Länge 1 bis $n-1$ gebildet (hier 1 bis 6) gebildet.



Die Länge 7 ist irrelevant, denn dann haben wir das Pattern gefunden.

Knuth-Morris-Pratt Algorithmus

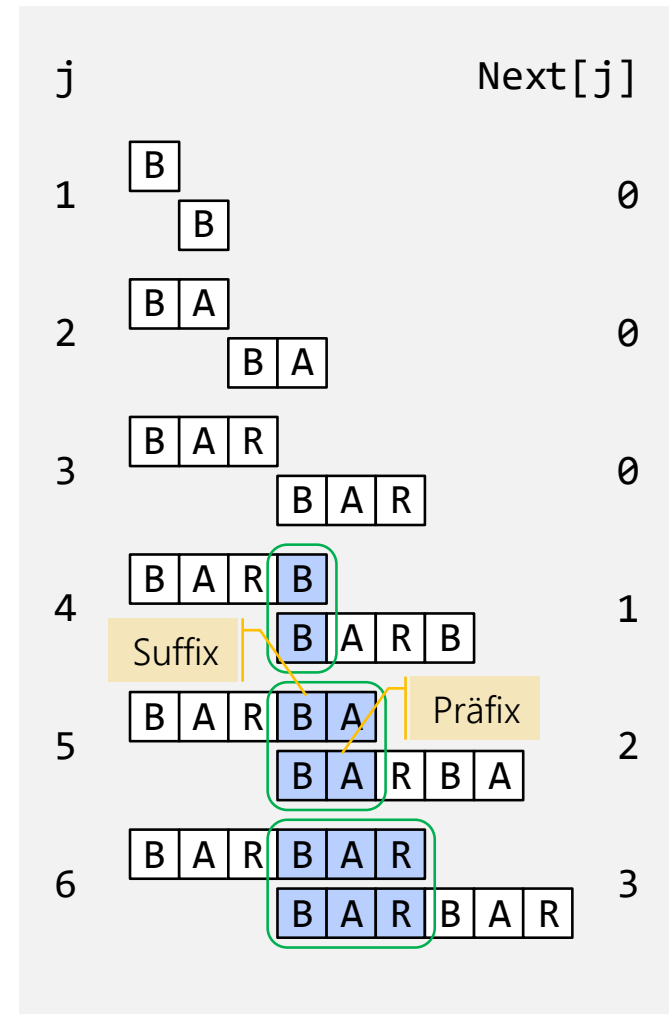
- 1.2 Danach wird jedes Subpattern von ganz links nach rechts verschoben, bis alle überlappenden Zeichen übereinstimmen, oder keine Überlappung gefunden wurde. Bei Übereinstimmung haben wir einen identischen Präfix und Suffix. Im Beispiel für das Subpattern der Länge 5 einen Präfix und Suffix der Länge 2:



Knuth-Morris-Pratt Algorithmus

Das Resultat für alle Subpattern:

next[j]: um wieviel darf ich das Pattern nach links verschoben ansetzen und die Suche fortsetzen, falls Buchstabe $j + 1$ abweicht. Oder anders ausgedrückt, wie lange ist der gemeinsame Präfix und Suffix für das Subpattern der Länge j .



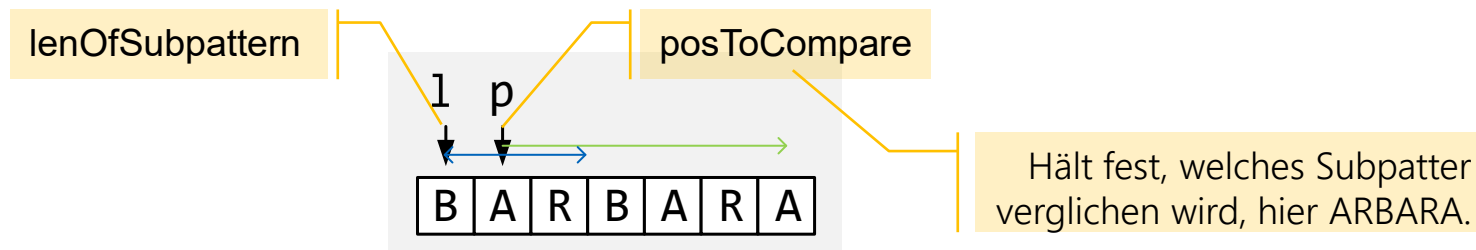
Knuth-Morris-Pratt Algorithmus

Die nächste Folie zeigt einen effizienten Algorithmus, um die Next-Tabelle zu erzeugen.

Aufwand $O(m)$, m = Länge des Pattern.

Dazu werden zwei Pointer eingeführt.

- Der Pointer $p = \text{posToCompare}$ wandert von der Position 1 Schritt um Schritt bis zum Ende des Patterns.
- Der Pointer $l = \text{lenOfSubpattern}$ zeigt, an welcher Position der Präfix aktuell verglichen wird (startet an der Position 0).



Knuth-Morris-Pratt Algorithmus

```
public static int[] buildNextTab(String pattern) {
    int lenOfPattern = pattern.length();
    int lenOfSubpattern = 0;
    int posToCompare = 1;
    int[] next = new int[lenOfPattern - 1];

    while (posToCompare < lenOfPattern - 1) {
        if (pattern.charAt(posToCompare) == pattern.charAt(lenOfSubpattern)){
            next[posToCompare] = lenOfSubpattern + 1;
            lenOfSubpattern++;
            posToCompare++;
        }
        else {
            if (lenOfSubpattern != 0) {
                lenOfSubpattern = next[lenOfSubpattern - 1];
            }
            else {
                next[posToCompare] = 0;
                posToCompare++;
            }
        }
    }
    return next;
}
```

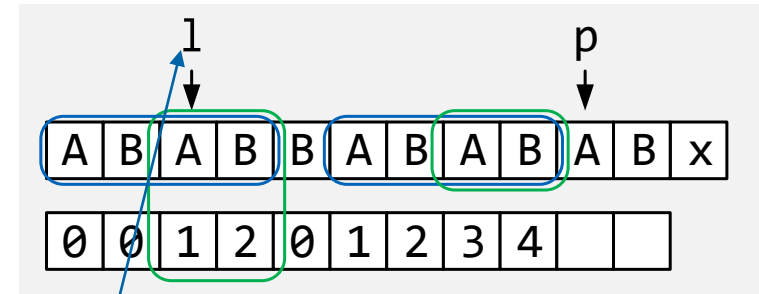
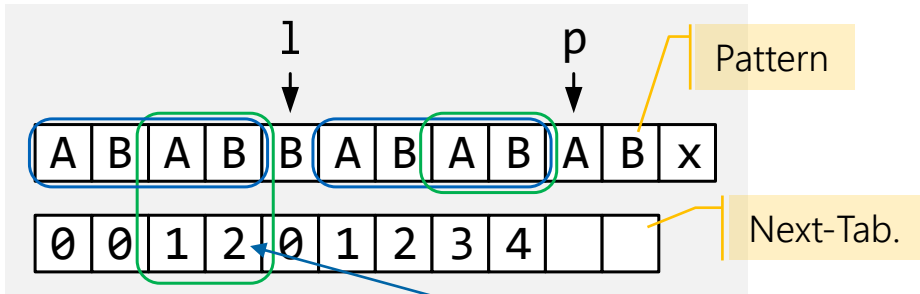
Die Zeichen stimmen überein, versuche Präfix und Suffix zu verlängern.

Schwierigster Teil des Algorithmus. Bei einer Abweichung darf die Subpatternlänge nicht einfach auf 0 gesetzt werden, da allenfalls bereits ein anderes Subpattern erkannt wurde. Es muss daher zunächst das nächst kleinere, gefundene Subpattern ausprobiert werden → siehe nächste Folie.

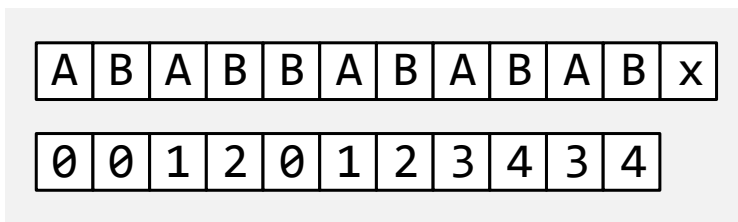
Das erste Zeichen des Subpattern und das aktuelle Zeichen weichen voneinander ab → das ist nicht der Beginn eines Subpattern.

Knuth-Morris-Pratt Algorithmus

Die Zeichen an Position l und p weichen voneinander ab. Daher ist das Subpattern (blauer Bereich) nicht 5 Zeichen lang (Bild links). :



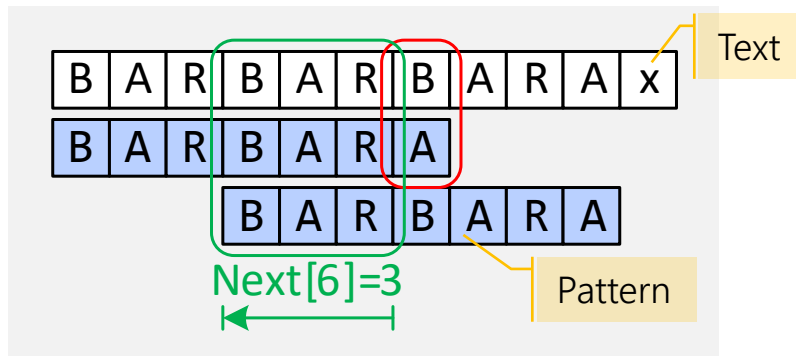
Wir wissen, dass die beiden blauen Bereiche identisch sind. Wir sehen im linken blauen Bereich auch, dass das rechte AB im grünen Bereich ein Substring ist (gem. next-Tabelle). Daher müssen wir ausprobieren, ob wir diesen Substring (auch vor p) durch das Zeichen an Position p erweitern können. In $\text{next}[l-1]$ sehen wir, dass der Substring 2 Zeichen lang ist. Wir setzen daher l auf die Position 2, das im nächsten Durchlauf zu überprüfende Zeichen. Lösung nach Ausführung:



Knuth-Morris-Pratt Algorithmus

2. Phase: Text gemäss der Next-Tabelle durchsuchen:

j	1	2	3	4	5	6
Next[j]	0	0	0	1	2	3



Beim Vergleich des 7. Buch-stabens (B <> A, rot umrandet) des Patterns kommt es zur Abweichung.

Gemäss Next-Tabelle darf für den weiteren Vergleich das Pattern um drei Positionen nach links «verschoben» angesetzt werden und die Suche fortgesetzt werden...

Dadurch wird jedes Zeichen des Texts nur maximal zwei Mal mit dem Pattern verglichen.

Aufwand $O(n)$, n = Länge des Text.

Knuth-Morris-Pratt Algorithmus

```
public static void KMP(String textToSearch, String pattern){
    int lenOfText = textToSearch.length();
    int lenOfPattern = pattern.length();
    int[] next = buildNextTab(pattern);
    next = int posOfText = 0, posOfPattern = 0;

    while ((posOfText < lenOfText) && (posOfPattern < lenOfPattern)) {
        if (textToSearch.charAt(posOfText) == pattern.charAt(posOfPattern)) {
            posOfText++;
            posOfPattern++;
        }
        else {
            if (posOfPattern != 0) {
                posOfPattern = next[posOfPattern - 1];
            }
            else {
                posOfText++;
            }
        }
    }
    if (posOfPattern == lenOfPattern) {
        println("Position: " + Integer.toString(posOfText - lenOfPattern));
    }
}
```

Die Zeichen stimmen überein, versuche Pattern und Text zu verlängern.

Starte den nächsten Vergleich mit dem Subpattern gemäss der Verschiebung in der next-Tabelle.

Das erste Zeichen des Pattern und das aktuelle Zeichen des Texts weichen voneinander ab → das ist nicht der Beginn des Pattern.

Knuth-Morris-Pratt Algorithmus

- Sehr schlauer Algorithmus.
- Laufzeit $O(n+m)$, n = Länge Text, m = Länge Pattern.
- Wenn man die beiden Methoden buildNextTab und KMP vergleicht, fällt auf, dass diese sehr ähnlich aufgebaut sind.

Hand-Aufgabe Knuth-Morris-Pratt Algorithmus:

Suchen Sie mit dem Algorithmus das Pattern nano im Text: nenananox



Invertierter Index

Invertierter Index: Motivation

Wikipedia: Millionen englischer Artikel. Wir suchen alle Artikel in denen das Wort «Twitter» vorkommt. Wie macht man das?

QA Warum implementiert die String x W Donald Trumps Umgang mit den Medien

← → ↻ 🔍 https://de.wikipedia.org/wiki/Donald_Trumps_Umgang_mit_den_Medien 1/24 Geo Webmail News Invest Weitere Favoriten

Nicht angemeldet Diskussionseite Beiträge Benutzerkonto erstellen Anmelden

Artikel Diskussion Lesen Bearbeiten Quelltext bearbeiten Versionsgeschichte Wikipedia durchsuchen

WIKIPEDIA
Die freie Enzyklopädie

Hauptseite
Themenportale
Zufälliger Artikel

Mitmachen
Artikel verbessern
Neuen Artikel anlegen
Autorenportal
Hilfe
Letzte Änderungen
Kontakt
Spenden

Werkzeuge
Links auf diese Seite
Änderungen an verlinkten Seiten
Spezialseiten
Permanenter Link
Seiteninformationen
Artikel zitieren
Wikidata-Datenobjekt

Drucken/exportieren
Buch erstellen
Als PDF herunterladen
Druckversion

In anderen Projekten
Commons

In anderen Sprachen
English
中文
Links bearbeiten

Donald Trumps Umgang mit den Medien

Diesen November ist Asiatischer Monat der Wikipedia. Nimm am Wettbewerb teil und gewinne Postkarten aus Asien. [Hilf uns bei der Übersetzung]

Dieser Artikel oder Abschnitt bedarf einer Überarbeitung. Näheres sollte auf der [Diskussionsseite](#) angegeben sein. Bitte hilf mit, ihn zu verbessern, und entferne anschließend diese Markierung.

Der **Umgang des US-Präsidenten Donald Trump mit den Medien** und deren Berichterstattung über ihn wird weltweit kontrovers diskutiert. Ihm wird beschönigt, selbst in Konkurrenz zu den über ihn berichtenden Medien zu treten.^[1] Trump selbst diskreditierte die Berichterstattung über ihn wiederholt als **Fake News** (vorgetauschte Nachrichten).^[2]

Inhaltsverzeichnis [\[Verbergen\]](#)

- Von Trump genutzte Medien
 - Twitter**
 - Breitbart News Network
- Einzelfälle
 - Attacken gegen einzelne Journalisten und Medien
 - Streit um Teilnehmerzahl bei Amtseinführung
 - „Last night in Sweden“
 - covfefe
- Einzelnachweise

Von Trump genutzte Medien [\[Bearbeiten\]](#) [\[Quelltext bearbeiten\]](#)

Twitter [\[Bearbeiten\]](#) [\[Quelltext bearbeiten\]](#)

Donald Trump äußert sich öfter als jeder andere US-amerikanische Präsident auf dem Kurznachrichtendienst **Twitter**. Nicht selten bereiten die fast alltäglichen Kurznachrichten, die der amerikanische Präsident ungefiltert verbreitet, Verdruss bei seinen Sicherheits- und Rechtsberatern.^[3]

Das politische Magazin *The New Republic* bezeichnete Trump im November 2016 wegen seiner hohen Aktivität bei **Twitter** in einer Schlagzeile als „ersten **Twitter**-Präsidenten“ und meinte: „Sei besorgt.“^[4] Diese Schlagzeile verwendete im Dezember 2016 auch *Fox News* und ergänzte, dass Trump dadurch die Möglichkeit erhalte, per *Tweet* „Rache“ zu verüben.^[5]

Trump hat mehrfach ihm unliebsame Kommentatoren seiner **Twitter**-Einträge blockiert, so dass diese nicht mehr auf seine Tweets zugreifen können. Das „Knight Institute“ an der *Columbia-Universität in New York* sieht darin einen Bruch des Grundrechts auf eine Beteiligung am öffentlichen Diskurs.^[6] Ende Mai 2018 entschied ein Gericht in *Manhattan*, dass Trump Nutzer nicht aus politischen Gründen blockieren dürfe, da es sich bei seinem **Twitter**-Konto um ein öffentliches Forum handle. Ein Verstoß dagegen stellt laut dem Gericht eine Einschränkung der im 1. Zusatzartikel zur US-Verfassung garantierten *Redefreiheit* ein, dennoch sei die Funktion des „Stummschaltens“ weiterhin eine mögliche Option.^[7]

Nach einer über einen *Tweet* getätigten Behauptung Donald Trumps am 26. Mai 2020 unterzog **Twitter** erstmals eine Aussage des US-Präsidenten einem *Faktencheck*, der die Behauptung als irreführend einschätzte.^[8] Darüber erzwung, verfügte er per Dekret, den durch Paragraph 230 (Section 230) des *Communications Decency Act* gewährten Schutz sozialer Netzwerke wie **Twitter** und Facebook vor Strafverfolgung (ähnlich dem deutschen *Providerprivileg* bezüglich *Forenhaltung*) zu beenden und die Befugnis der Betreiber jener Plattformen zu beschneiden, durch Nutzer veröffentlichte Inhalte zu moderieren.^[9]

Donald Trump (2017)

Invertierter Index

«... ein invertierter Index ist eine Indexdatenstruktur, die eine Abbildung vom Inhalt, wie z.B. Wörter oder Zahlen, auf seine Positionen ...in einem Dokument oder einer Gruppe von Dokumenten speichert.» (wikipedia)

```
T[0] = "it is what it is"
```

```
T[1] = "what is it"
```

```
T[2] = "it is a banana"
```

Texte
(Dokumente)

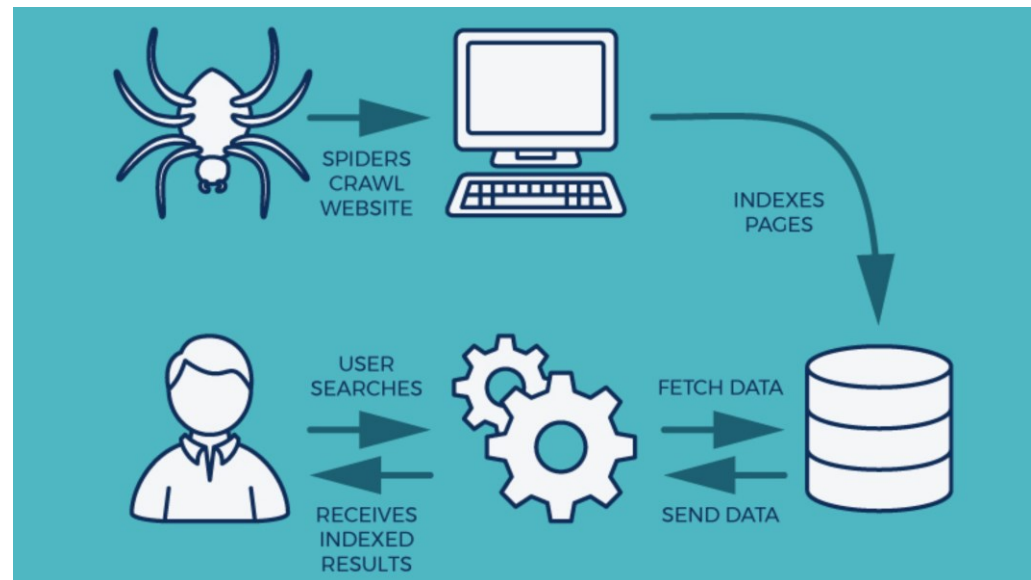
```
a          {(2, 2)}  
banana     {(2, 3)}  
is         {(0, 1), (0, 4), (1, 1), (2, 1)}  
it         {(0, 0), (0, 3), (1, 2), (2, 0)}  
what       {(0, 2), (1, 0)}
```

Invertierter
(umgekehrter) Index:
Wort und (Dokumente,
Positionen).

Invertierter Index: Suchmaschinen

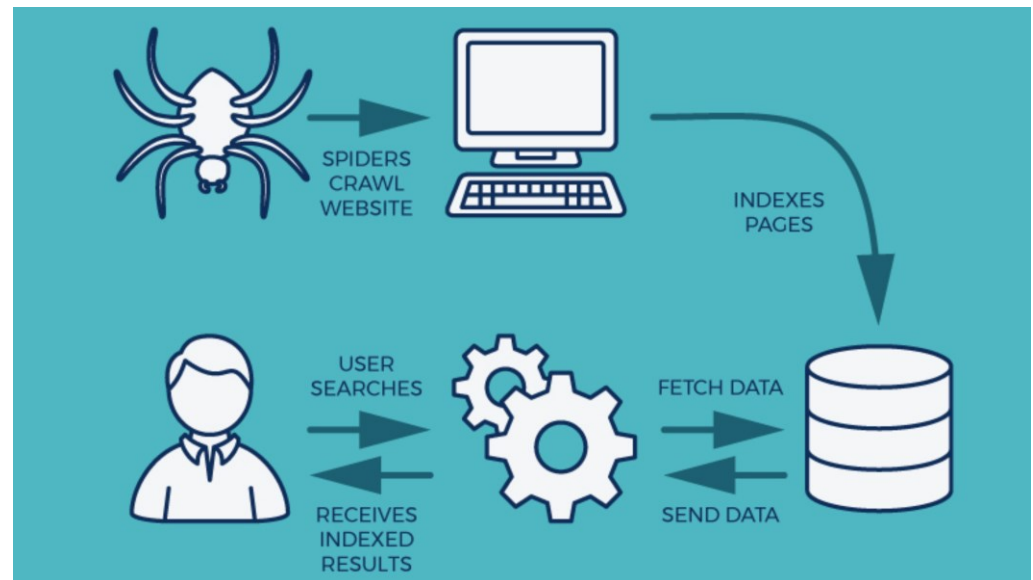
Suchmaschinen verwenden häufig invertierte Indexe.

- Web Roboter / Spider / Crawler
 - Durchsuchen regelmässig das Web nach neuen Informationen.
- **Indexierung**
 - Aufbereitung von Dokumenten.
 - Speicherung im Index / in der Datenbank der Suchmaschine.
- Retrievalsystem:
 - Suche im Index
 - Sortierung nach Relevanz
 - Wo kommen die Suchbegriffe vor?
 - Wie oft kommen die Begriffe vor?
 - In welcher Reihenfolge?
 - Wie lang ist der Text?
 - Wie viele Links verweisen auf das Dokument?
 - ...



Invertierter Index: Ordnung, Verbesserungen

- Performance Suche, Terme unsortiert: $O(n)$, n = Anzahl Wörter
- Verbesserungen?
 - Terme sortieren und binär suchen $\rightarrow O(\log(n))$
 - Terme in einen balancierten Baum $\rightarrow O(\log(n))$
 - Hashing \rightarrow dann $O(1)$, allenfalls verteilte Hashtabellen, über mehrere Systeme
 - Nur relevante Terme, Stopwords entfernen, z.B.: den, dem, die...
<https://www.ranks.nl/stopwords/german>.
 - Terme normalisieren:
Wortstamm bilden (wohn: wohnen, bewohnen, wohnlich, Wohnzimmer, ...)





Levenshtein-Distanz (Approximative Suche)

Levenshtein-Distanz

Die **Levenshtein-Distanz** (auch: Editier-Distanz) von zwei Wörtern A und B ist die minimale Anzahl Operationen, um aus dem ersten Wort das zweite Wort zu machen.

Erlaubte «Operationen»:

1. insert(c):
Buchstaben 'c' an einer Position im ersten Wort einfügen.
2. update(c→d):
Buchstaben 'c' an einer Position im ersten Wort durch 'd' ersetzen.
3. delete(c):
Buchstabe 'c' an einer Position im ersten Wort löschen.

Mit der Levenshtein-Distanz können ähnliche Wörter, oder Wörter mit kleinen Schreibfehlern gesucht werden.

Levenshtein-Distanz

Was ist die Levenshtein-Distanz von 'Haus' und 'Maus'?

```
update(H > M)  
Distanz = 1
```

Was ist die Levenshtein-Distanz von 'Saturday' und 'Sunday'?

```
delete(a)  
delete(t)  
update(r > n)  
Distanz = 3
```

Wie können wir die kürzeste Distanz mit einem Algorithmus berechnen?

Levenshtein-Distanz: Berechnung

Gegeben zwei Wörter A und B der Länge n und m.

- Konstruiere eine Matrix D mit der Grösse $(n+1) \cdot (m+1)$.
- $D[i,j]$ gibt die Levenshtein-Distanz der Präfixe von A und B der Länge i und j an.

A[i] von \ B[j] zu		S	a	t	u	r	d	a	y
	0								
S		0							
u									
n									
d									
a									
y									

TakeOver(S)

Update(u→a)

Delete(n)

Delete(d)

Aus Sund wurde Sa

Wenn wir hier ankommen, wurde Sunday mit Saturday ersetzt.

Beispiel:

A = 'Sunday', B = 'Saturday'

$D[4,2] = 3$, da 'Sund' und 'Sa' Distanz 3 haben.

Sund → Sund → Sa~~nd~~ → Sa~~d~~ → Sa

↓ Delete

Update
TakeOver
Insert

Levenshtein-Distanz: Berechnung

Die Matrix kann Zelle für Zelle gefüllt werden.

$$D[i,j] = \min$$

$$\begin{cases} D[i-1, j-1] + 0, & \text{falls } A[i] = B[j] \\ D[i-1, j-1] + 1, & \text{falls } \text{update}(A[i] \rightarrow B[j]) \\ D[i, j-1] + 1, & \text{falls } \text{insert}(B[j]) \\ D[i-1, j] + 1, & \text{falls } \text{delete}(A[i]) \end{cases}$$

B[j] zu A[i] von		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Zeile 0 = Aufwand
(alles) Einfügen.

Spalte 0 = Aufwand
(alles) Löschen.

Zelle(n, m) = gesuchter
Aufwand Umformung.

Levenshtein-Distanz: Berechnung

		Update(S) ↓	Update(a) ↓	Update(t) ↓	Update(u) ↓	Update(r) ↓	Update(d) ↓	Update(a) ↓	Update(y) ↓
		Insert(S) →	Insert(a) →	Insert(t) →	Insert(u) →	Insert(r) →	Insert(d) →	Insert(a) →	Insert(y) →
B[j] zu A[i] von		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

B[0]=A[0]	TakeOver(S)	0
B[1]=a	Insert(a)	1
B[2]=t	Insert(t)	1
B[3]=A[1]	TakeOver(u)	0
B[4]=r	Update(r)	1
B[5]=A[3]	TakeOver(d)	0
B[6]=A[4]	TakeOver(a)	0
B[7]=A[5]	TakeOver(y)	0
Levenshtein-Distanz:		3

- ↓ Delete(S) ↗ TakeOver(S)
- ↓ Delete(u) ↗ TakeOver(u)
- ↓ Delete(n) ↗ TakeOver(n)
- ↓ Delete(d) ↗ TakeOver(d)
- ↓ Delete(a) ↗ TakeOver(a)
- ↓ Delete(y) ↗ TakeOver(y)

Mögliche Umformungen
finden man über den
Rückweg.

Aufwand: $O(n \cdot m)$

Levenshtein-Distanz: Algorithmus

```
private static int minimum(int a, int b, int c) {  
    return Math.min(Math.min(a, b), c);  
}
```

Minimum dreier
Werte bestimmen

```
public static int computeLevenshteinDistance(String str1, String str2) {  
    int[][] distance = new int[str1.len() + 1][str2.len() + 1];
```

```
    for (int i = 0; i <= str1.len(); i++) distance[i][0] = i;  
    for (int j = 1; j <= str2.len(); j++) distance[0][j] = j;
```

Initialisierung der Spalte
und Zeile 0 (alle
löschen / einfügen)

```
    for (int i = 1; i <= str1.len(); i++) {  
        for (int j = 1; j <= str2.len(); j++) {  
            int minEd = (str1.charAt(i - 1) == str2.charAt(j - 1)) ? 0 : 1;  
            distance[i][j] = minimum(distance[i - 1][j] + 1,  
                                     distance[i][j - 1] + 1, distance[i - 1][j - 1] + minEd);  
        }  
    }
```

Zeile für Zeile und
Spalte für Spalte
Distanz berechnen.

Muss Update oder
TakeOver ausgeführt
werden?

```
    return distance[str1.len()][str2.len()];  
}
```

Kürzeste Distanz ist
in Zelle rechts unten

Levenshtein-Distanz: Übung

Übung:

Berechnen Sie die Levenshtein-Distance-Matrix für die beiden Wörter.

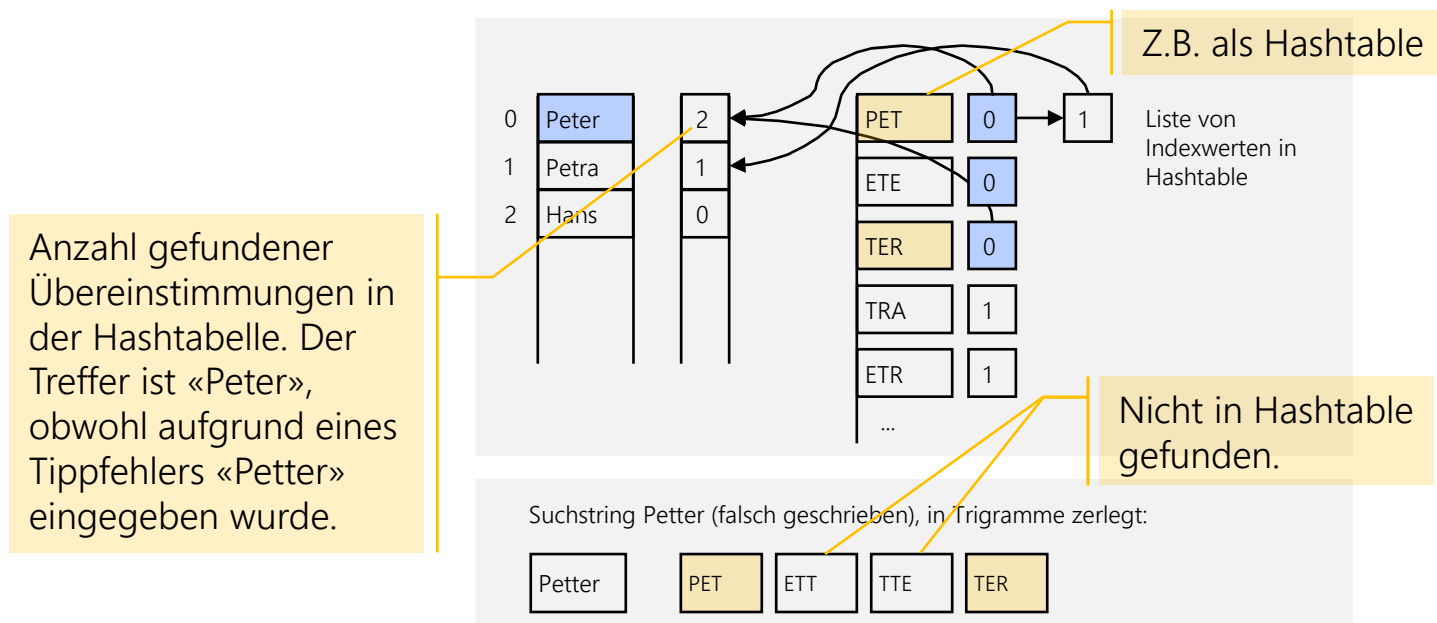
B[j] zu		W	O	R	L	D
A[i] von		W	O	R	L	D
	0	1	2	3	4	5
W	1					
O	2					
R	3					
D	4					



Trigramm-Suche

Trigramm-Suche

- **Fehlertolerante Suche** (auch für Wortverdreher, z.B Vor- und Nachname).
- Effizient für grosse Datenbestände.
- Index → Wort in 3er-Buchstaben Gruppen unterteilt:
 - Z.B. sind das bei «Peter», drei 3-er Gruppen «PET», «ETE», «TER».
 - Diese 3-er Gruppen werden für vorkommende Worte gebildet und in der Hashtabelle gespeichert.
- Das zu suchende Wort wird ebenfalls in 3-er Gruppen zerlegt.
- Das gesuchte Wort mit am meisten Übereinstimmungen wird genommen.





Phonetische Suche

Phonetische Suche

Soundex ist ein phonetischer Algorithmus zur Indizierung von Wörtern und Phrasen nach ihrem Klang.

Ein Wort besteht aus seinem ersten Buchstaben, gefolgt von drei Ziffern, z.B. "K523" (Soundex-Code):

- Die Vokale A, E, I, O und U und die Konsonanten H, W und Y sind ausser beim ersten Zeichen zu ignorieren (in D auch ä, ö, ü).
- Kurze Worte: mit 0 auffüllen, 0-werden ignoriert
- Ziffern sind Konsonanten nach folgender Tabelle

Englisch

Ziffer	Repräsentierte Buchstaben
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Deutsch

Ziffer	Repräsentierte Buchstaben
0	a, e, i, o, u, ä, ö, ü, y, j, H
1	b, p, f, v, w
2	c, g, k, q, x, s, z, ß
3	d, t
4	l
5	m, n
6	r
7	ch

Britney → BRTN → B635	bewährten → BWRTN → B1635 → B163
Spears → SPRS → S162	Superzicke → SPRZCK → S16222 → S162

Auf «Englisch» ergibt sich hier auch B635, oder?



Suchen nach Mustern

Regex: Reguläre Ausdrücke

- Zum Suchen von definierten Mustern in Texten
 - Bestimmter String: "ZHAW"
 - Unscharfe Muster: z.B. IT20a, IT19a, IT19c
 - Wiederholende (Teil-)Muster: 170.12.34.12
- Die meisten heutigen Programmiersprachen unterstützen die Suche nach Muster in Form von reguläre Ausdrücke (Regular Expressions) oder kurz Regex.
- Regex ist unabhängig von Java definiert.
- Java-Klassenbibliothek definiert im Package [java.util.regex](#).
- Klassen: [Pattern](#) und [Matcher](#).

Wir wiederholen und erweitern hier den Stoff aus Programmieren 1.

Regex: Definition

- Zuerst muss der Regex-Ausdruck vorbereitet werden.
- Mit der Klasse Pattern wird der Regex-Ausdruck in ein Pattern compiliert:

```
Pattern pat = Pattern.compile("ZHAW");
```

- Muster im einfachsten Fall ein Textzeichen-String
- Alle Zeichen sind erlaubt ausser: (`[{\^-$|]`)[?]*+.
 - Diese Zeichen müssen mit `\` vorangestellt geschrieben werden
 - Vorsicht: in Java-String-Konstante muss `"\\"` für `"\"` geschrieben werden.
Beispiel: `"wie geht's \\"`

Jetzt kann das verbotene
? verwendet werden.

Regex: Abfrage

- Ausgabe der gefundenen Stellen
- Die Klasse **Matcher** wird verwendet, um Stringoperationen am Pattern auszuführen.

```
Matcher matcher = pat.matcher("Willkommen an der ZHAW");
```

"ZHAW"

- Suche nächste Textstelle: **boolean find()**, true falls gefunden.

```
matcher.find();
```

Rückgabewert: true

- Gebe den mit **find()** gefundenen Teilstring zurück: **String group()**

```
matcher.group();
```

Rückgabewert: ZHAW

- Gefundene Start und Endposition: **int start()** und **int end()**

```
matcher.start();  
matcher.end();
```

Rückgabewert: 18

Rückgabewert: 22

Regex: Beispiel

```
import java.util.regex.*;

...

Pattern pat = Pattern.compile("ZHAW");
Matcher matcher = pat.matcher("Willkommen an der ZHAW");
while (matcher.find()) {
    String group = matcher.group();
    int start = matcher.start();
    int end = matcher.end();
    // do something
}
```

Regex: Platzhalter

- Oftmals wird nach unscharfen Mustern gesucht, z.B. alle IT Klassen.
- Es sind Platzhalter-Zeichen erlaubt, die Zeichenmengen repräsentieren.
Z.B.: . (Punkt) für beliebiges Zeichen \d für Zahl, \D für keine Zahl.

Platzhalter	Beispiel	Bedeutung	Beispiele gültige Literale
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
\d	\d\d	Digit[0-9] (Ziffern)	78, 10
\D	\D	kein Digit	a, b, c, ...
\w	\w	ein Buchstabe, eine Ziffer oder der Unterstrich	a, A, _, 0, ...
\W	\W	Weder Buchstabe, Ziffer noch Unterstrich	€, #, ...
\s	\s	Leerzeichen (Blank, etc)	<blank>, <tab>, <cr>, ...
\S	\S	kein Leerzeichen	Jedes Zeichen ausser Leerzeichen.

Aufgabe:

Geben Sie das Suchmuster für beliebige IT Klassen an (ohne Teilzeitklassen): IT20a, IT19b, IT19c

Regex: Eigene Zeichenmengen

Statt vordefinierte Zeichenmengen zu verwenden, können auch eigene definiert werden, diese werden in [und] geklammert. Varianten:

1. Aufzählung der Zeichen in der Zeichenmenge:

Ein Zeichen aus der Menge: z.B. a, b oder c: [abc]

2. Bereiche:

Z.B. alle Kleinbuchstaben [a-z], oder alle Buchstaben [a-zA-Z]

3. Negation: Alle Zeichen ausser:

Z.B. nicht a: [^a]

Aufgabe:

Geben Sie das Suchmuster für beliebige IT Klassen an, erlaubt ist nur a bis d, z.B. IT19a

Regex: Optional, Alternative und Wiederholung

- Optionale Teile: ?

Wenn einzelner Buchstaben optional, z.B. ZHA?W → ZHW oder ZHAW.

- Alternative (Oder): |

Wenn ein A oder ein B → ZH(A|B)W → ZHAW oder ZHBW

- Wiederholungen:

1. Beliebige oft: *

eine Folge von Ziffern $\backslash d^*$ → $_, 2, 23, 323, 423, \dots$

Auch 0 mal erlaubt

2. Mindestens einmal +

eine Folge von Ziffern aber mindestens eine $\backslash d^+$ → 3, 34, 234, ...

3. Bestimmte Anzahl mal {n}

eine Folge von drei Ziffern $\backslash d\{3\}$ → 341, 241, 123, ...

4. Mindestens, maximal Anzahl {n,m}

eine Folge von 1 bis 3 Ziffern $\backslash d\{1,3\}$ → 1, 23, 124, ...

- Gruppierung: ()

Zum Beispiel: IT(18|19|20) → IT18, IT19, IT20

Regex: Zusammenfassung

- (Meta-)Sprache zur Beschreibung der Bildungsregeln von Sätzen.
- Metasymbole: $([\{\backslash^-\$| \}])?^*+.$

Metasymbol	Beispiel	Bedeutung	Beispiele gültige Literale
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
\d	\d\d	Digit[0-9] (Ziffern)	78, 10, ...
\D	\D	kein Digit	a, b , c, ...
\w	\w	ein Buchstabe, eine Ziffer oder der Unterstrich	a, A, _, 0, ...
\W	\W	Weder Buchstabe, Ziffer noch Unterstrich	€, #, ...
\s	\s	Leerzeichen (Blank, etc)	<blank>, <tab>, <cr>, ...
\S	\S	kein Leerzeichen	Jedes Zeichen ausser Leerzeichen.
[]	[abc]x	1 Zeichen aus einer Menge	ax, bx, cx
[-]	[a-h]	Zeichenbereich	a, b, c, d, e, f, g, h
[^]	[^abd]	Negation (zus. mit [])	Jedes Zeichen ausser a, b oder c.
?	ax?b	x optional	ab, axb
	a b	a oder b	a, b
*	ax*b	0 oder mehrere x	ab, axb, axxb, axxxb, ...
+	ax+b	1 oder mehrere x	axb, axxb, axxxb, ...
{n}	\w{3}	n Wiederholungen	aaa, aA0, _a0, ...
{n,m}	\d{1,3}	n bis m Wiederholungen	1, 12, 123, ...
()	x(a b)x	Gruppierung	xax, xbx

Regex: Weitere Methoden

Ersetzt im gegebenen String alle, bzw. den ersten Substring, die regex entsprechen.

```
String replaceAll(String regex, String replacement);
```

```
String replaceFirst(String regex, String replacement);
```

```
String new = text.replaceAll("l+", "LL"); // HaLLo WeLLt
```

Teilt den gegebenen String in mehrere Strings, regex ist die Grenzmarke, das Resultat ist ein Array mit Teilstrings.

```
String[] split(String regex);
```

```
String data = "4, 5, 6 2,8,, 100, 18"  
String[] teile = data.split("[ ,]+"); // Menge der Zeichen " " und", "  
// 4 5 6 2 8 100 18  
// teile[0] = "4", teile[1] = "5", ...
```

Regex: Weitere Methoden

Prüfe ob ganzer String einem Regex Muster entspricht:

```
boolean matches(String regex);
```

```
String text = "Hallo Welt";  
boolean passt;  
passt = text.matches("H.*W.*");           // true  
passt = text.matches("H..o Wel?t");       // false  
passt = text.matches("H[alo]* W[elt]+");   // true  
passt = text.matches("Hal+o Welt.+");     // false
```

. Beliebiges Zeichen
? Optionales Zeichen
+ Mindestens ein Mal
. Beliebig oft
[] Menge erlaubter Zeichen

Aufgabe:

Regex zum Prüfen ob ein String eine Integer-Zahl enthält.

Aufgabe:

Regex zum Prüfen ob ein String eine IP-Adresse enthält.

Zusammenfassung

- Suche von Strings in Strings
- Suchmaschinen und Index
- Unscharfe Suche
- Suchen nach Mustern



Kontrollfragen Lektion 10
nicht vergessen – heute mit
Aschenputtel

