Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 20 (9.7.2018)

String Matching (Textsuche)

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Textsuche / String Matching

Gegeben:

- Zwei Zeichenketten (Strings)
- Text T (typischerweise lang)
- Muster P (engl. pattern, typischerweise kurz)

Ziel:

- Finde alle Vorkommen von P in T

Annahmen:

- Länge Text T : n, Länge Muster P : m

$m \ll n$

Beispiel:

Motivation

- Ist offensichtlich wichtig...
- Wird in jedem Texteditor gebraucht
  - jeder Editor hat eine find-Funktion
- Wird von Programmiersprachen unterstützt:
  - Java: String.indexOf(String pattern, int fromThisPosition)
  - C++: std::string.find(std::string str, size_t fromThisPosition)
  - Python: strong.find(pattern, from)

Naiver Algorithmus

- Gehe den Text von links nach rechts durch
- Das Muster kann an jeder der Stellen $s = 0, \dots, n - m$ vorkommen

- Prüfe an jeder dieser Stellen ob das Muster passt
  - indem das Muster Buchstabe für Buchstabe mit dem Text an der Stelle verglichen wird
  - Werden wir gleich noch etwas genauer anschauen...

Naiver Algorithmus

TestPosition(s): // tests if $T[s, \dots, s+m-1] == P$

$t := 0$

while $t < m$ **and** $T[s+t] = P[t]$ **do**

$t := t + 1$

return $(t = m)$

Naiver Algorithmus

```
TestPosition(s):          // tests if T[s, ..., s + m − 1] == P
    t := 0
    while t < m and T[s + t] = P[t] do
        t := t + 1
    return (t = m)
```

Laufzeit:

worst case: $O(n \cdot m)$

```
String-Matching:
    for s := 0 to n − m do
        if TestPosition(s) then
            report found match at position s
```

Naiver Algorithmus Analyse

Laufzeit von `TestPosition(`s`)`:

- Annahme: erster Mismatch ist an Position x des Patterns P
  - $x = m$ falls das Pattern gefunden wird

$$\text{Laufzeit} = \begin{cases} x, & \text{falls erster Mismatch an Pos. } x \\ m, & \text{falls Muster gefunden} \end{cases}$$

Laufzeit des Algorithmus:

- **Beispiel:** $T =$ `000010001010001` $P =$ `0001`

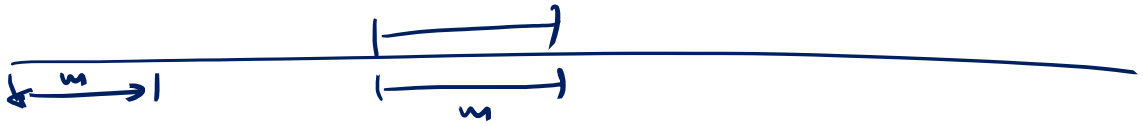
  0 0 0 1
  0 0 0 1

- **Best case:** $T =$ `KESDJETNARKRETEJ` $P =$ `ABCDE` $O(n)$

- **Worst case:** $T =$ `AAAAAAA...AAAAAAA` $P =$ `AA...A` $O(n \cdot m)$

Rabin-Karp Algorithmus

Grundidee



- Wir schieben wieder ein Fenster der Grösse m über den Text und schauen an jeder Stelle, ob das Muster passt

- Zur Einfachheit nehmen wir an, dass der Text nur aus den Ziffern $0, \ldots, 9$ besteht
  - dann können wir das Muster und das Fenster als Zahl verstehen
- Wenn wir das Fenster eins nach rechts schieben, kann die neue Zahl einfach aus der alten berechnet werden

$T = 5\,3\,2\,7\,2\,1\,8\,5\,7\,3$

32721

$27218 = (32721 - 3 \cdot 10^4) \cdot 10 + 8$

Rabin-Karp Algorithms

UNI FREIBURG

Beobachtungen:

- In jedem Schritt müssen wir einfach zwei Zahlen vergleichen
- Falls die Zahlen gleich sind, kommt das Muster an der Stelle vor
- Wenn man das Fenster um eins weiter schiebt, lässt sich die neue Zahl in $O(1)$ Zeit berechnen
- Falls wir zwei Zahlen in $O(1)$ vergleichen können, dann hat der Algorithmus Laufzeit $O(n)$

- **Problem:** Die Zahlen können sehr gross sein ($\Theta(m)$ bits)
  - Zwei $\Theta(m)$-bit Zahlen vergleichen benötigt Laufzeit $\Theta(m)$
  - Nicht besser als mit dem naiven Algorithmus

- **Idee:** Benutze Hashing und vergleiche Hashwerte
  - Wenn man das Fenster eins weiter schiebt, sollte sich der neue Hashwert wieder in $O(1)$ Zeit aus dem alten Hashwert berechnen lassen

Rabin-Karp Algorithmus

Muster Länge m

$M \ll m$

$0, \dots, M-1$

Lösung von Rabin und Karp:

- Wir rechnen alles mit den Zahlen modulo M
  - M sollte möglichst gross sein, allerdings klein genug, damit die Zahlen $0, \dots, M-1$ in einer Speicherzelle (z.B. 32 Bit) Platz haben
- Muster und Textfenster sind dann beides Zahlen aus dem Bereich

$$\{0, \dots, M-1\}$$

- Beim Schieben des Fensters um eine Stelle, lässt sich die neue Zahl wieder in $O(1)$ Zeit berechnen
  - Falls das nicht klar ist, siehe spätere Folie...
- Falls das Muster gefunden wird, sind die zwei Zahlen gleich, falls nicht, können sie trotzdem gleich sein
  - Falls die Zahlen gleich sind, dann überprüfen wir nochmals wie beim naiven Algorithmus Buchstabe für Buchstabe

Rabin-Karp Algorithmus: Beispiel

Text: 572830354826 **Muster: 283** **Modulus $M = 5$**

$283 \bmod 5 = 3$

$572 \bmod 5 = 2$

$728 \bmod 5 = 3$ ⟩ $O(1)$ Zeit!

↳ teste $728 = 283$ → kein Match

$283 \bmod 5 = 3$ $283 = 283$ → Pattern gefunden

Rabin-Karp Algorithmus: Pseudo-Code

Text $T[0 \ldots n-1]$, Muster $P[0 \ldots m-1]$, Basis b, Modulus M

oder sogar in $O(\log m)$ Zeit

$h := b^{m-1} \bmod M$ (kann man in $O(m)$ Zeit berechnen)

$p := 0;\ \ t := 0;$

for $i := 0$ **to** $m-1$ **do**

$\quad p := (p \cdot b + P[i]) \bmod M$ ← Hashwert von P

$\quad t := (t \cdot b + T[i]) \bmod M$

← $p = P \bmod M$, $t = T[0,\ldots,m-1] \bmod M$

$s := 0;$

while $s \leq n-m$ **do**

$\quad$ **if** $p = t$ **then**

$\quad\quad$ TestPosition(s)

$\quad t := \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$ ⟸ $O(1)$ Zeit

$h = b^{m-1} \bmod M$

Rabin-Karp Algorithmus: Laufzeit

Vorberechnung: $O(m)$

Im schlechtesten Fall: $O(n \cdot m)$

- Der schlechteste Fall tritt ein, falls die Zahlen in jedem Schritt übereinstimmen. Dann muss man in jedem Schritt Buchstabe für Buchstabe überprüfen, ob man das Muster wirklich gefunden hat.
  - Sollte bei guter Wahl von M nicht allzu oft geschehen...
  - ausser, wenn das Muster tatsächlich sehr oft ($\Theta(n)$ mal) vorkommt...

Im besten Fall:

- Im besten Fall sind die Zahlen nur gleich, falls das Muster auch wirklich gefunden wird. Die Kosten sind dann $O(n + k \cdot m)$, falls das Muster im Text k Mal vorkommt.

Wahl der Parameter...

b, M

Zahlendarstellung und Wahl von M

- Wir hätten gerne, dass wenn $x \neq y$, dann ist $h(x) = h(y)$ "unwahrscheinlich" (für $h(x) := x \bmod M$)
- Nehmen wir an, dass die Buchstaben in Muster und Text als Ziffern zur Basis b dargestellt werden
  - in unserem Fall, haben wir $b = 10$
- Falls b und M einen gemeinsamen Teiler haben, ist $h(x) = h(y)$ trotz $x \neq y$ nicht so unwahrscheinlich

Extremfall $b = 10$, $M = 5$ (M ist ein Teiler von b)

$$10^i \bmod 5 = \begin{cases} 1 & \text{falls } i = 0 \\ 0 & \text{falls } i \geq 1 \end{cases}$$

Muster $\alpha_{m-1}, \alpha_{m-2}, \dots, \alpha_0 = \sum_{i=0}^{m-1} \alpha_i \cdot b^i = \sum_{i=0}^{m-1} \alpha_i \cdot 10^i$

$$\left(\sum_{i=0}^{m-1} \alpha_i \cdot 10^i\right) \bmod 5 = \left(\sum_{i=0}^{m-1} (\alpha_i \cdot 10^i \bmod 5)\right) \bmod 5 = \alpha_0 \bmod 5$$

Wahl der Parameter...

UNI FREIBURG

Zahlendarstellung und Wahl von M

- Wir hätten gerne, dass wenn $x \neq y$, dann ist $h(x) = h(y)$ "unwahrscheinlich" (für $h(x) := x \bmod M$)
- Nehmen wir an, dass die Buchstaben in Muster und Text als Ziffern zur Basis b dargestellt werden
  - in unserem Fall, haben wir $b = 10$
- Falls b und M einen gemeinsamen Teiler haben, ist $h(x) = h(y)$ trotz $x \neq y$ nicht so unwahrscheinlich

Wir wählen deshalb

- Die Basis b als genug grosse Primzahl
  - bei ASCII-Zeichen muss $b > 256$ sein
- M kann dann beliebig gewählt werden, am besten als Zweierpotenz
  - Zwischenresultate sind $< M \cdot b$, das sollte also in 32 (64) Bit Platz haben

Rechnen Modulo M

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot m \ \wedge\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: addiere/subtrahiere M von x bis die Zahl im Bereich $\{0, \dots, M-1\}$ ist

Rechenregeln:

$$(a \cdot b) \bmod m = \big((a \bmod m) \cdot (b \bmod m)\big) \bmod m$$
$$(a + b) \bmod m = \big((a \bmod m) + (b \bmod m)\big) \bmod m$$

$a = k \cdot m + c$ $\quad a \bmod m = c$ $\quad (c, d \in \{0, \dots, m-1\})$

$b = \ell \cdot m + d$ $\quad b \bmod m = d$

$$(a \cdot b) \bmod m = \big(k \cdot \ell \cdot m^2 + (kd + \ell c) \cdot m + c \cdot d\big) \bmod m$$
$$= c \cdot d \bmod m = (a \bmod m) \cdot (b \bmod m) \bmod m$$

Rechnen Modulo m

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot m \ \wedge\ y \in \{0, \dots, M-1\}$$

- $x \bmod M$: addiere/subtrahiere M von x bis die Zahl im Bereich $\{0, \dots, M-1\}$ ist

$T[i] \in \{0, \dots, b-1\}$

Rechenregeln:

$$(a \cdot b) \bmod m = \big((a \bmod m) \cdot (b \bmod m)\big) \bmod m$$
$$(a + b) \bmod m = \big((a \bmod m) + (b \bmod m)\big) \bmod m$$

Schieben des Fensters:

$T[s], \dots, T[s+m-1]$ ⇒ $T[s+1], \dots, T[s+m]$

- Fenster von Stelle s nach Stelle $s+1$ schieben

$t :=$ "$T[s], \dots, T[s+m-1]$" $\quad t' :=$ "$T[s+1], \dots, T[s+m]$"

$$t' = \big((t - T[s] \cdot b^{m-1}) \cdot b + T[s+m]\big) \bmod M$$

$b^{m-1} \bmod M$

Rechnen Modulo m

$$x \bmod M = y \iff \exists q \in \mathbb{Z}\colon\ y = x + q \cdot m \ \wedge\ y \in \{0, \dots, M-1\}$$

Negative Zahlen

- Damit ist $x \bmod M$ immer im Bereich $\{0, \dots, M-1\}$

  Beispiele:

  $$24 \bmod 10 = 4, \qquad 4 \bmod 10 = 4, \qquad -4 \bmod 10 = 6$$

- **Aber:** In Java / C++ / Python ist $-x \,\%\, m = -(x \,\%\, m)$

  Beispiele:

  $$24 \,\%\, 10 = 4, \qquad 4 \,\%\, 10 = 4, \qquad -4 \,\%\, 10 = -4$$

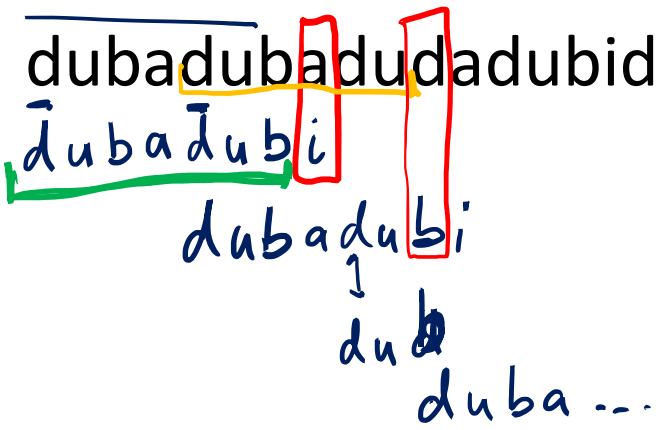
- **Workaround:** Falls das Resultat von $x \,\%\, m$ negativ ist, einfach m dazu addieren, dann kommt man in den richtigen Bereich

Algorithmus von Knuth, Morris, Pratt

- Kann wir das Problem immer in Zeit $O(n)$ lösen?
  - im schlechtesten Fall...

Schauen wir uns nochmals ein Beispiel an:

- Pattern: dubadubi
  Text: dubadubadudadubidubadubidubiduda

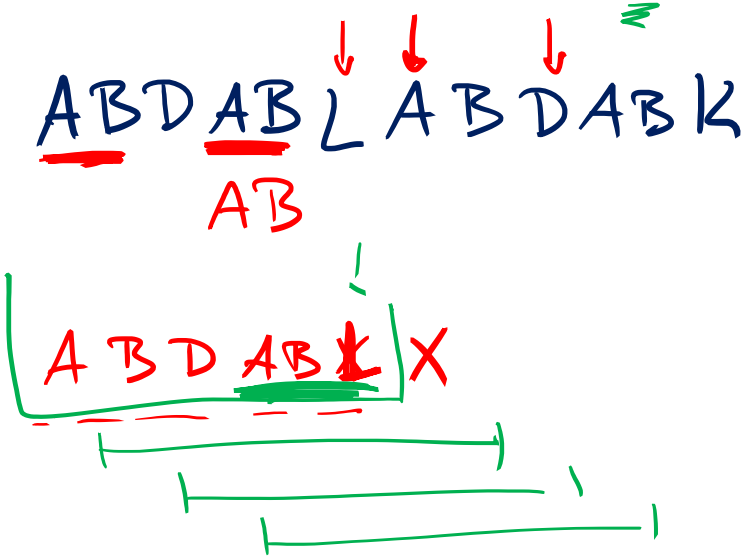


Knuth-Morris-Pratt Algorithmus

Idee:

- Falls wir beim Testen des Musters P an Stelle t feststellen, dass $P[t]$ nicht mit dem Text an der entsprechenden Stelle übereinstimmt, dann wissen wir, dass die Stellen $P[0 \dots t-1]$ übereingestimmt haben.
- Das können wir bei der weiteren Suche ausnutzen

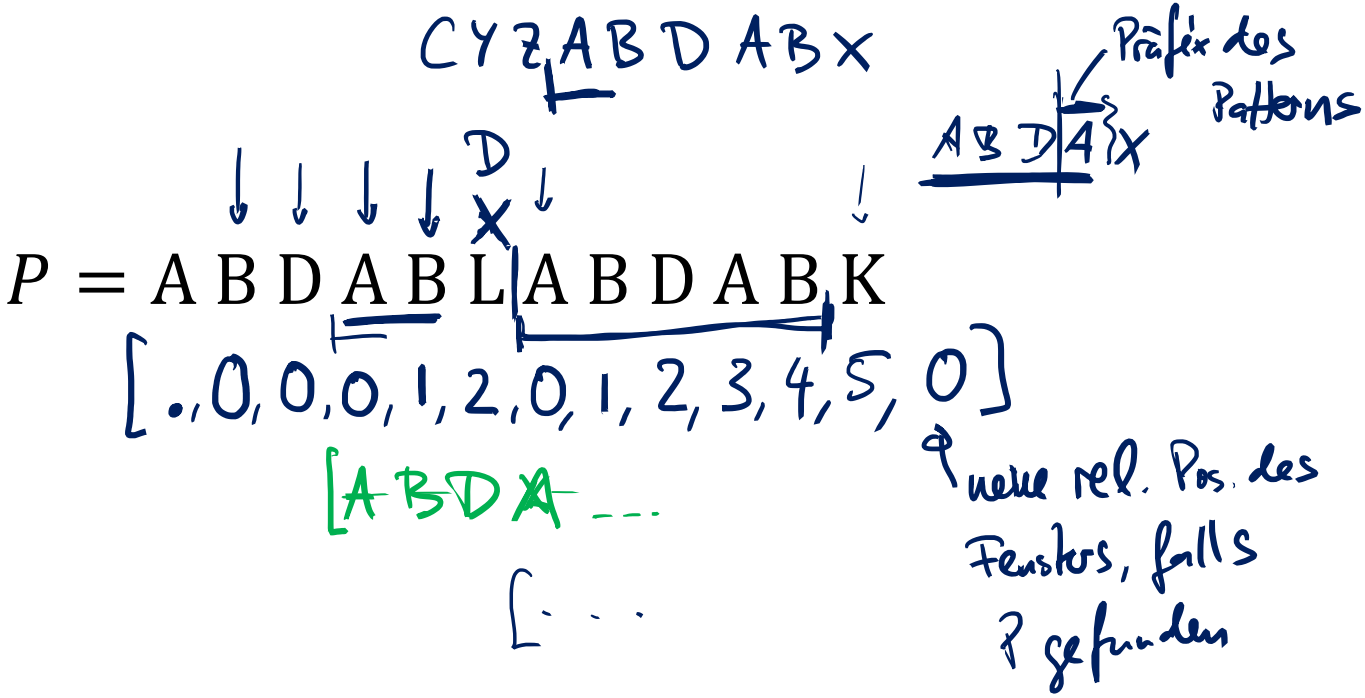
Beispiel: $P = \text{ABDABLABDABK}$



Knuth-Morris-Pratt Alg.: Initialisierung

- Wir merken uns an jeder Stelle des Musters, wie weit wir das Suchfenster bei einem "Mismatch" weiterschieben können.
  - dabei ignorieren wir den "Typ" des "Mismatches"
- Äquivalent dazu: Stelle im Muster, an welcher wir weitersuchen müssen

Weiterschieben soll unabhängig vom Text sein



Knuth-Morris-Pratt Algorithmus

Vorberechnung: Array S der Länge $m+1$

- $S[i]$: Stelle in P, an welcher man die neue Suche beginnt, falls beim Testen der Stelle i im Pattern ein Mismatch auftritt
- $S[0] = -1, \quad S[1] = 0$ } Konvention
- $S[m]$: Stelle in P, an welcher man weitersucht, nachdem P erfolgreich gefunden wurde

Beispiel:

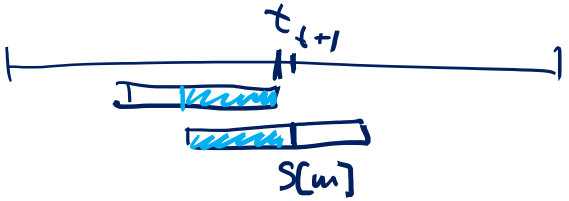
$P =$ [A, B, D, A, B, L, A, B, D, A, B, D]

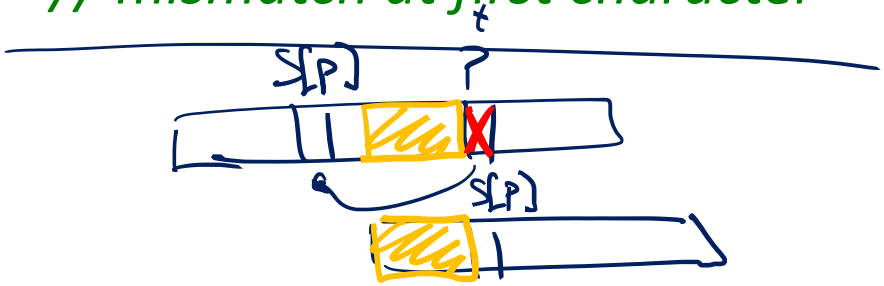
$S =$ [-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]

$S[m]$

Knuth-Morris-Pratt Algorithmus

```
t := 0; p := 0          // t: Position in Text,  p: Position im Pattern
while t < n do
    if T[t] = P[p] then          // characters match
        if p = m − 1 then        // pattern found
            pattern found at position t − m + 1
            p := S[m]; t := t + 1
        else
            p := p + 1; t := t + 1
    else                         // characters don't match
        if p = 0 then            // mismatch at first character
            t := t + 1
        else
            p := S[p]
```





Knuth-Morris-Pratt Alg.: Beispiel

Pattern: ABCABC $\quad S = [-1,0,0,0,1,2,3]$

Text:

A	D	A	B	C		D	A	B	C	A	G	A	B	V	A	B	C	A	B	C	A	B	C	

Knuth-Morris-Pratt Alg.: Laufzeit

Laufzeit ohne Initialisierung des Arrays S:

Laufzeit $\in O(n)$

$t := 0;\ p := 0$

Schleifendurchläufe $\leq 2n$

while $t < n$ **do**

  if $T[t] = P[p]$ **then**

    if $p = m - 1$ **then**

      pattern found

      $p := S[m];\ t := t + 1$ (I)

    else

      $p := p + 1;\ t := t + 1$ (II)

  else

    if $p = 0$ **then**

      $t := t + 1$ (III)

    else

      $p := S[p]$ (IV) ← Verschieben des Fensters

Fälle (I)–(III) kommen zusammen $\leq n$ mal vor

Fall (IV):

Obs: $S[p] < p$

kann auch höchstens n mal vorkommen

Initialisierung

Vorberechnung von Array S:

- $P = [$ A, B, D, A, B, L, A, B, D, A, B, D$]$

  $S = [-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3]$

- An Position in $S[i]$ (für $i \in \{2, \dots, m\}$) steht

$$\boldsymbol{S[i] := \min_{k<i}\{P[i-k \dots i-1] = P[0 \dots k-1]\}}$$

- $S[i]$: Länge des längsten echten Teilstückes von $P[0 \dots i-1]$, welches an Stelle $i-1$ endet, und welches auch Anfangsstück von P ist

Berechnung von $S[i]$:

- Falls $P[S[i-1]] = P[i-1]$, dann ist $S[i] = S[i-1] + 1$
- Sonst testen, ob es einen kürzeres, passendes Anfangsstück gibt
  - Wir werden gleich anschauen, wie man das macht...