# Informatik II - SS 2014

## (Algorithmen & Datenstrukturen)

Vorlesung 21 (29.7.2014)

String Matching (Textsuche) II

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

# Textsuche / String Matching

UNI FREIBURG

**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 $T$ : $\boldsymbol{n}$, Länge Muster $P$ : $\boldsymbol{m}$

**Beispiel:**

- Text: dubadubadudadubidubadubidubiduda
- Pattern: dubidu

# Naiver Algorithmus

UNI FREIBURG

```
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)

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

# Rabin-Karp Algorithmus

UNI FREIBURG

**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 $0, \dots, 9$ 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

# Rabin-Karp Algorithmus

UNI FREIBURG

**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: Pseudo-Code

UNI FREIBURG

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

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

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

**for** $i := 0$ **to** $m-1$ **do**
    $p := (p \cdot b + P[i]) \bmod M$
    $t := (t \cdot b + T[i]) \bmod M$

$s := 0;$

**while** $s \leq n-m$ **do**
    **if** $p = t$ **then**
        TestPosition($s$)
    $t := \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$

# Rabin-Karp Algorithmus: Laufzeit

UNI FREIBURG

**Vorberechnung:**

**Im schlechtesten Fall:**

- 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...

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

# 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:**

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

# 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:**

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

**Schieben des Fensters:**

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

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

# 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 $m$ dazuaddieren, dann kommt man in den richtigen Bereich

# Algorithmus von Knuth, Morris, Pratt

UNI FREIBURG

- 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

UNI FREIBURG

**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 $P[0 \ldots t-1]$ übereingestimmt haben.
- Das können wir bei der weiteren Suche ausnutzen

**Beispiel:** $P = \mathrm{ABDABLABDABK}$

# Knuth-Morris-Pratt Alg.: Initialisierung

UNI FREIBURG

- Wir merken uns an jeder Stelle des Musters, wie weit wir das Suchfenster bei einem “Mismatch” weiterschieben können.

$$P = \mathrm{A\ B\ D\ A\ B\ L\ A\ B\ D\ A\ B\ K}$$

# Knuth-Morris-Pratt Algorithmus

UNI FREIBURG

**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
- $S[0] = -1, \quad S[1] = 0$
- $S[m]$: 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]`

# Knuth-Morris-Pratt Algorithmus

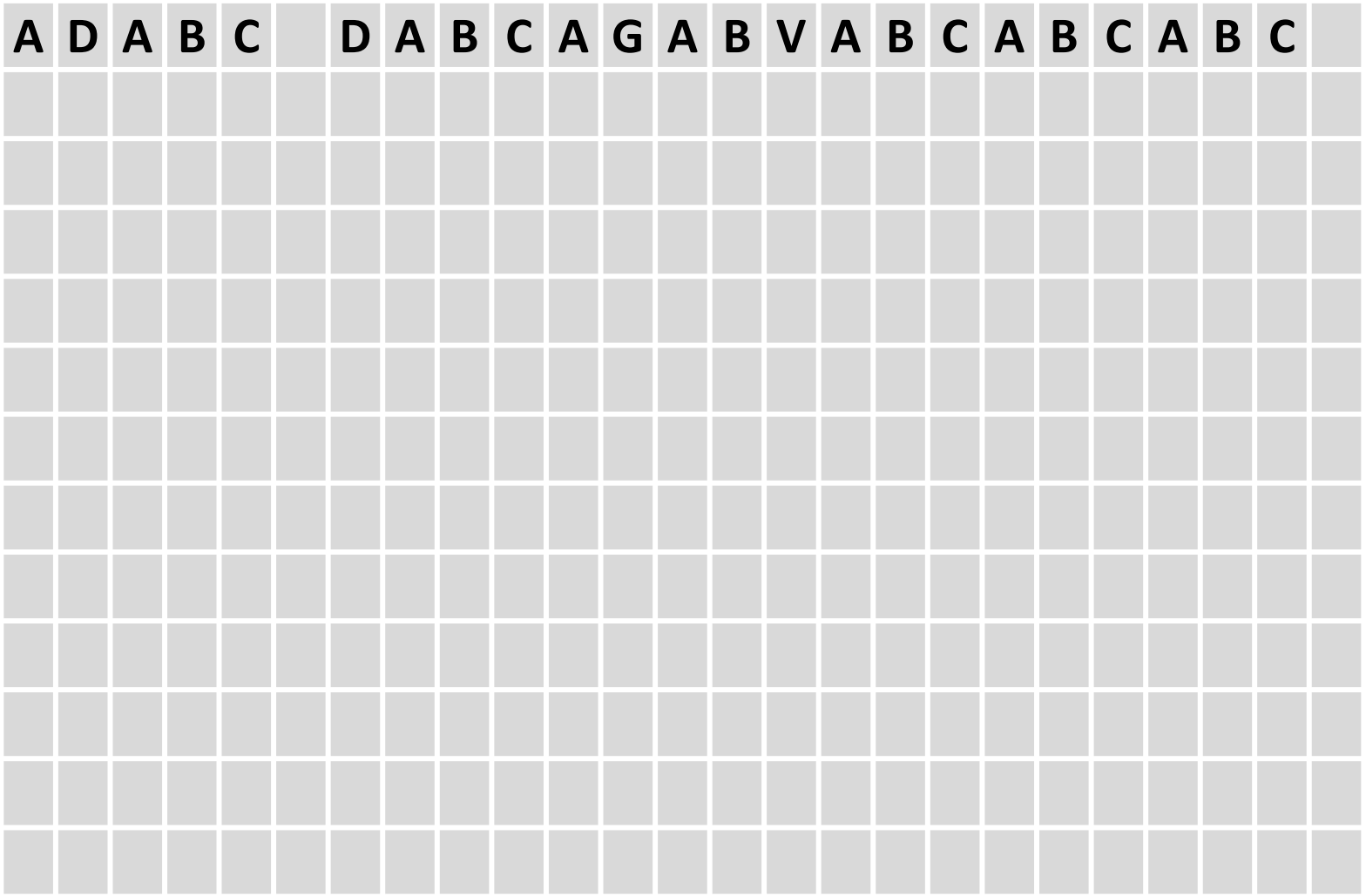
UNI FREIBURG

```
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:



# Knuth-Morris-Pratt Alg.: Laufzeit

UNI FREIBURG

**Laufzeit ohne Initialisierung des Arrays $S$:**

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

**while** $t < n$ **do**

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

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

      pattern found

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

    **else**

      $p := p + 1;\ \ t := t + 1$

  **else**

    **if** $p = 0$ **then**

      $t := t + 1$

    **else**

      $p := S[p]$

# Initialisierung

UNI FREIBURG

**Vorberechnung von Array $S$:**

- $P = [\ \mathtt{A}, \mathtt{B}, \mathtt{D}, \mathtt{A}, \mathtt{B}, \mathtt{L}, \mathtt{A}, \mathtt{B}, \mathtt{D}, \mathtt{A}, \mathtt{B}, \mathtt{D}]$

  $S = [\mathtt{-1}, \mathtt{0}, \mathtt{0}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{3}]$

- 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 $P\big[S[i-1]\big] = P[i-1]$, 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...

# Berechnung von $S[i]$: Beispiel

UNI FREIBURG

$h \coloneqq S[i-1]$

while $h \geq 0$ do

    if $P[i-1] = P[h]$ then

        $S[i] \coloneqq h+1$; $h \coloneqq -1$

    else

        $h \coloneqq S[h]$

if $h = -1$ then $S[i] = 0$

**Beispiel:** $P = [\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{L},\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{A},\mathtt{B},\mathtt{D},\mathtt{X}]$

# Berechnung von $S[i]$: Laufzeit

UNI FREIBURG

```
h := S[i − 1]
while h ≥ 0 do
    if P[i − 1] = P[h] then
        S[i] := h + 1;  h := −1
    else
        h := S[h]
if h = −1 then S[i] = 0
```

**Beobachtung:**

$$S[i] \leq S[i-1] + 1$$

**Falls $S[i] = S[i-1] + 1$:** 1 Schleifendurchlauf

**Falls $S[i] < S[i-1]$:**

- Wert von $h$ nimmt in jedem Schleifendurchlauf ab
- Am Schluss ist $S[i] = h + 1$
- Anzahl Schleifendurchläufe $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

# Berechnung von $S[i]$: Laufzeit

UNI FREIBURG

**Falls $\boldsymbol{S[i] = S[i-1] + 1}$:**

- Anzahl Schleifendurchläufe $= 1 = S[i-1] - S[i] + 2$

**Falls $\boldsymbol{S[i] < S[i-1]}$:**

- Anzahl Schleifendurchläufe $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

**Gesamtlaufzeit:**

# Knuth-Morris-Pratt Alg.: Zusammenfassung

UNI FREIBURG

**Knuth-Morris-Pratt Algorithmus:**

- Berechnet zuerst in Zeit $O(m)$ das Array $S$ der Länge $m$
  - hängt nur vom Pattern $P$ ab
  - beschreibt an jeder Position im Pattern, wo (im Pattern) man bei einem Mismatch weitersuchen muss
- Mit Hilfe von $S$ werden dann alle Vorkommen von $P$ in $T$ in Zeit $O(n)$ gefunden
  - In jedem Schritt kann man entweder die aktuelle Suchposition in $T$ oder die Position des Suchfensters in $T$ um mindestens 1 nach rechts verschieben

**Gesamtlaufzeit:** $\boldsymbol{O(m+n) = O(n)}$