Algorithmen und Datenstrukturen

Vorlesung 12

String Matching (Textsuche)

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 : $\boldsymbol{m}$ $(\boldsymbol{m} \ll \boldsymbol{n})$

Beispiel:

- Suche Muster $P =$"ABCA" in folgender Zeichenkette

$T =$ ABI CLABCAD LHABCABCA KAHBCA ALBCABABCABL LKAGA

Motivation

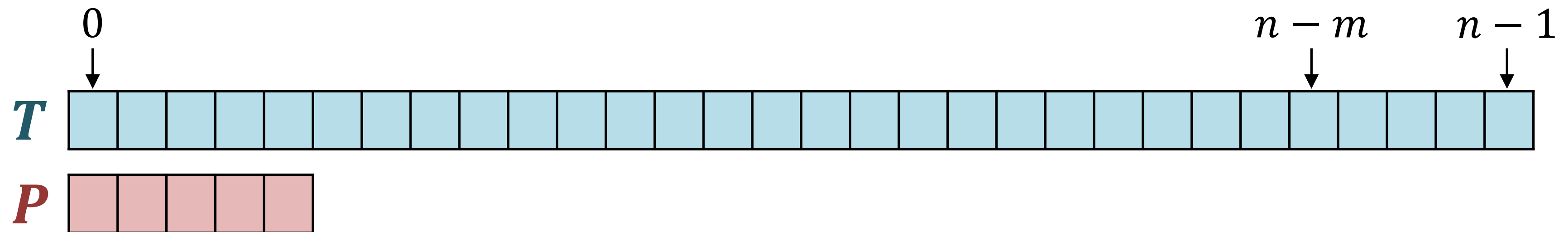
UNI FREIBURG

- 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: str.find(pattern, from), wobei str eine Zeichenkette ist

Naiver Algorithmus

UNI FREIBURG

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



- Prüfe an jeder dieser Stellen ob das Muster passt
  – indem das Muster Buchstabe für Buchstabe mit dem Text an der Stelle verglichen wird

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:

$$\#\text{Iter.} := \begin{cases} m, & \text{falls } P \text{ gefunden} \\ 1 + \min_{0<i<m} T[s+i] \neq P[i], & \text{sonst} \end{cases}$$

- Worst Case: $O(m)$
  - Im schlechtesten Fall muss man alle m Stellen von P durchgehen
  - Insbesondere ist das der Fall, wenn P gefunden wird
- Best Case: $O(1)$
  - Im besten Fall stellen wir schon beim ersten Zeichen fest, dass das Muster nicht passt (falls $T[s] \neq P[0]$)

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 from 0 to n − m do
        if TestPosition(s) then
            report found match at position s
```

Laufzeit:

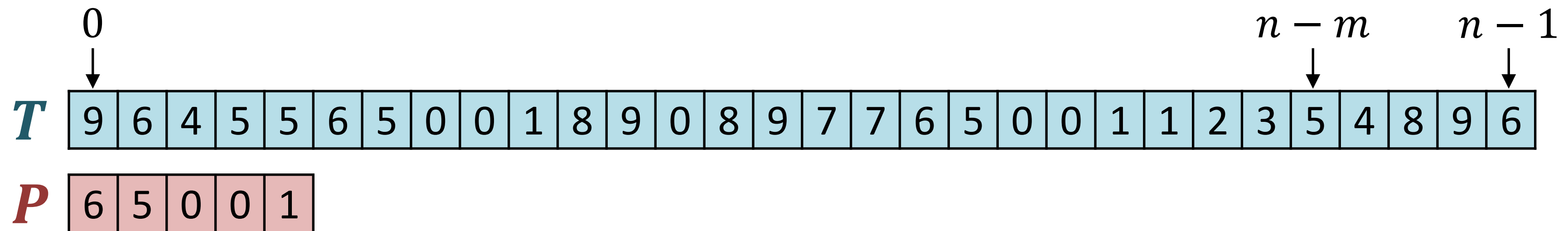
- Worst Case: $O(n \cdot m)$
- Best Case : $O(n)$

Rabin-Karp Algorithmus

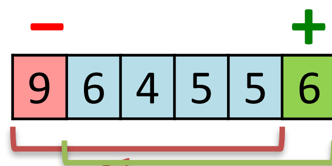
UNI
FREIBURG

Grundidee

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



- Wenn wir das Fenster eins nach rechts schieben, kann die neue Zahl einfach aus der alten berechnet werden



$$64556 = (96455 - 9 \cdot 10^{m-1}) \cdot 10 + 6$$

altes Fenster

neues Fenster

Rabin-Karp Algorithmus

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

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. 64 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
  - Das werden wir noch etwas genauer anschauen...
- 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 Zeichen für Zeichen

Rabin-Karp Algorithmus: Beispiel

UNI FREIBURG

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

Muster: $283 \bmod 5 = 3$

1. Fenster: $572 \bmod 5 = 2$

in $O(1)$ Zeit

2. Fenster: $728 \bmod 5 = 3$

teste: $728 \neq 283 \implies$ kein Match

3. Fenster: $283 \bmod 5 = 3$

teste: $283 = 283 \implies$ Pattern gefunden

Rechnen Modulo M

UNI FREIBURG

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

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

Rechenregeln:

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

$$a = k \cdot M + c \implies a \bmod M = c$$
$$b = \ell \cdot M + d \implies b \bmod M = d$$
$$(\, c, d \in \{0, \dots, M-1\}\,)$$

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

Rechnen Modulo M

UNI FREIBURG

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

- $x \bmod M$: 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

$$t \coloneqq (T[s] \dots T[s+M-1]) \bmod M,$$
$$t' \coloneqq (T[s+1] \dots T[s+M]) \bmod M$$

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

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 dazu addieren, dann kommt man in den richtigen Bereich

Rabin-Karp Algorithmus: Pseudo-Code

UNI FREIBURG

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

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

kann einfach in Zeit $O(m)$ und wenn man's richtig macht sogar in Zeit $O(\log m)$ berechnet werden

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

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

Hashwert von P: $p \coloneqq P \bmod M$

Hashwert von $T[0 \dots m-1]$: $t \coloneqq T[0 \dots m-1] \bmod M$

for $s = 0$ **to** $n-m$ **do**
    if $p == t$ **then**
        TestPosition(s)
    $t = \big((t - T[s] \cdot h) \cdot b + T[s+m]\big) \bmod M$

$O(m)$ Zeit falls Hashwerte übereinstimmten

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

aktualisiere t in $O(1)$ Zeit

Rabin-Karp Algorithmus: Laufzeit

UNI FREIBURG

Vorberechnung: $\boldsymbol{O(m)}$

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

- 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: $\boldsymbol{O(n + k \cdot m)}$ (k: Anz. Vorkommen von P in T)

- 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...

Zahlendarstellung und Wahl von M

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

Extremfall $b = 10, M = 20$ (b ist ein Teiler von M)

$$P = \alpha_{m-1}, \dots, \alpha_1, \alpha_0 = \sum_{i=0}^{m-1} \alpha_i \cdot 10^i \qquad 10^i \bmod 20 = \begin{cases} 1, & \text{if } i = 0 \\ 10, & \text{if } i = 1 \\ 0, & \text{if } i > 1 \end{cases}$$

$$\boldsymbol{P \bmod 20 = (\alpha_1 \cdot 10 + \alpha_0) \bmod 20}$$

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 $h(x) \coloneqq x \bmod M$)
- Nehmen wir an, dass die Buchstaben in Muster und Text als Ziffern zur Basis b dargestellt werden
  - in unseren Beispielen hatten wir $b = 10$
- Falls b und M einen gemeinsamen Teiler haben, ist $h(x) = h(y)$ trotz $x \neq y$ nicht so unwahrscheinlich

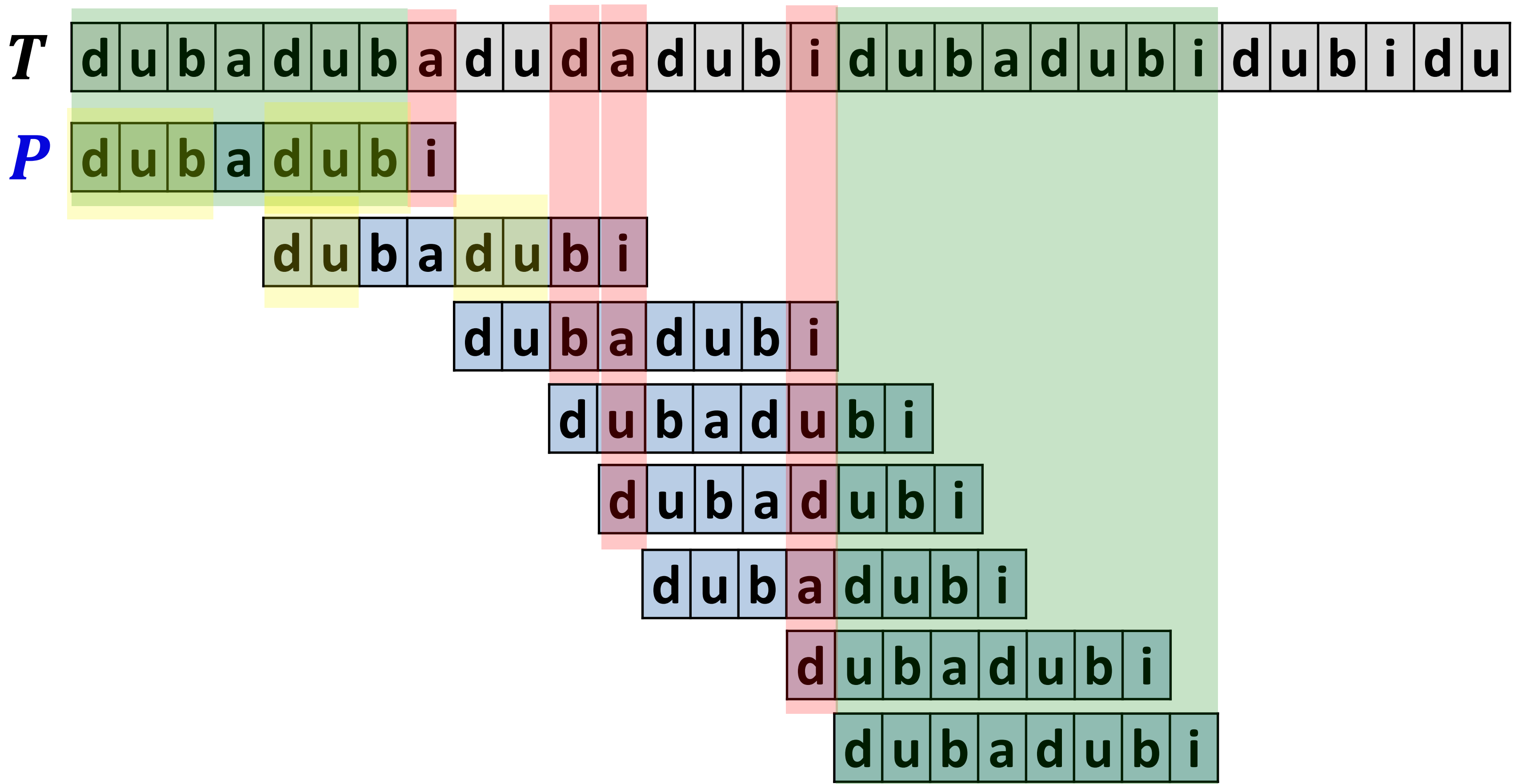
Wir wählen deshalb

- 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 z.B. in 64 Bit Platz haben

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:



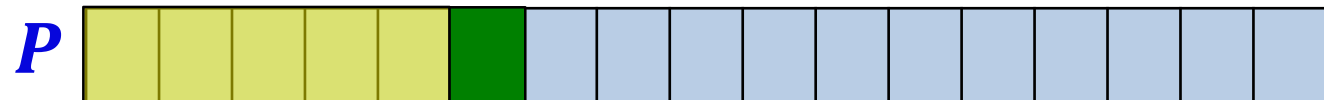
Knuth-Morris-Pratt Algorithmus

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

Längster Teil gleich vor dem Mismatch, der auch Präfix von P ist.

1. Stelle mit Mismatch



1. Stelle, die nun überprüft werden muss

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

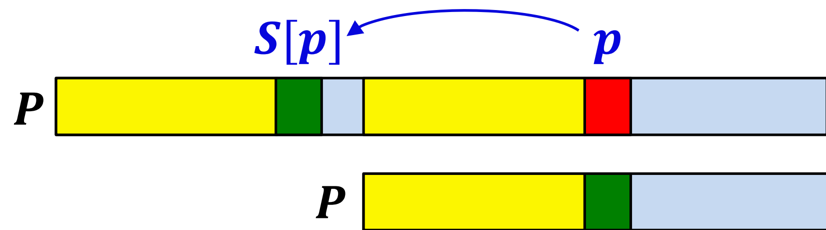
Beispiel:

$P =$ `[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


```
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 $\qquad S = [-1,0,0,0,1,2,3]$

Text:

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

Knuth-Morris-Pratt Alg.: Laufzeit

UNI FREIBURG

Laufzeit ohne Initialisierung des Arrays S: $O(n)$

$t = 0;\ \ p = 0$

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
            $p = S[p]$

In jedem Schritt wird

- die Position im Text inkrementiert

oder

- das Fenster verschoben

Initialisierung

UNI FREIBURG

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

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

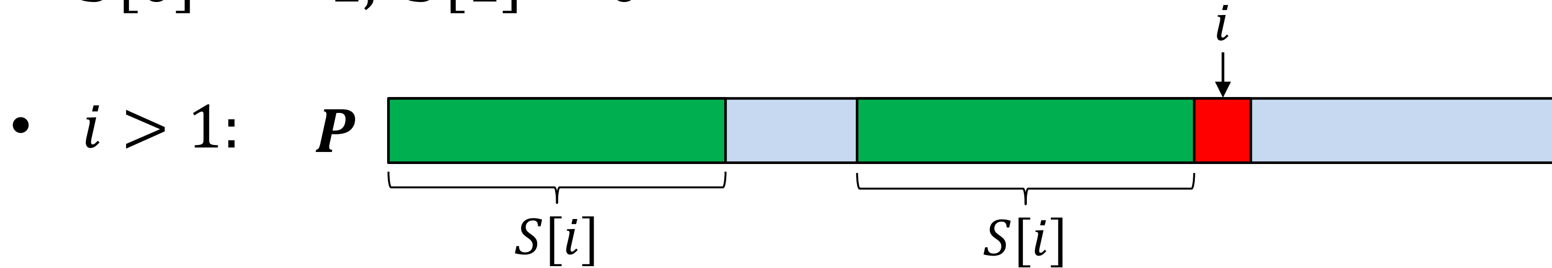
- $S[i]$: Länge des längsten echten Teilstückes von $P[0 \dots i-1]$, welches an Stelle $i-1$ endet, und welches auch Anfangsstück von P ist

Berechnung von $S[i]$:

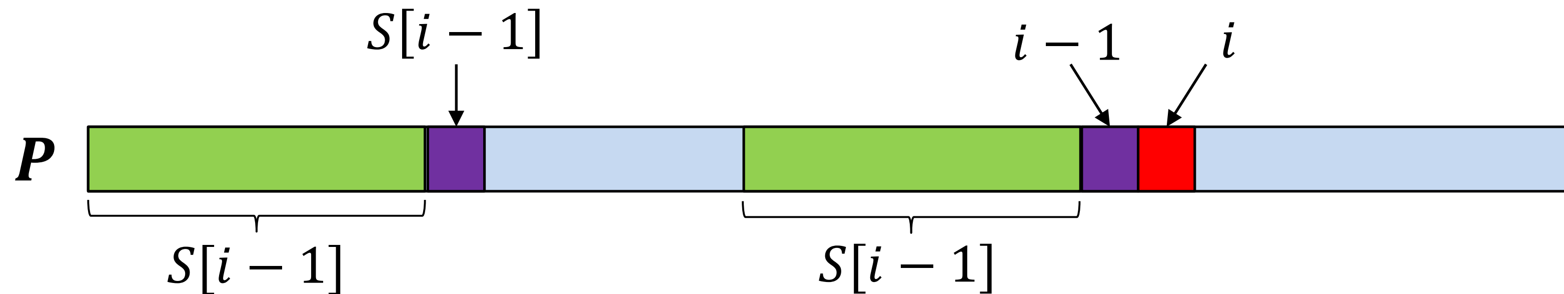
- Das werden wir gleich anschauen…

Berechnung von S[i]

- $S[0] = -1,\ S[1] = 0$
- $i > 1$:



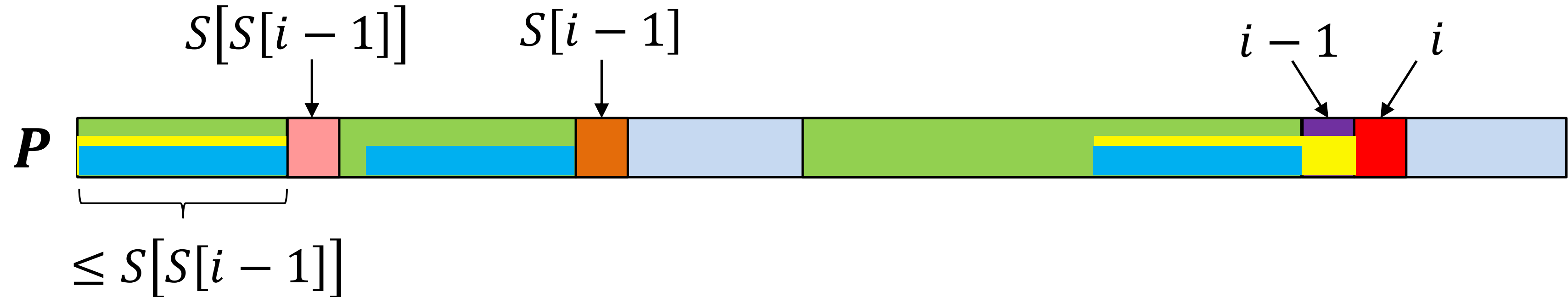
Fall 1 : $\boldsymbol{P[i-1] = P\big[S[i-1]\big]}$



- Falls $P[i-1] = P\big[S[i-1]\big]$, dann ist $S[i] = S[i-1] + 1$

Berechnung von S[i]

Fall 2 : $\boldsymbol{P[i-1] \neq P\big[S[i-1]\big]}$



- Längstes mögliches Teilstück hat Länge $S\big[S[i-1]\big]+1$
  - Teste, ob $P[i-1] = S\big[S[i-1]\big]$?
  - Falls ja, dann ist $S[i] = S\big[S[i-1]\big]+1$
  - Falls nein, dann ist die nächste Stelle, die man testen muss $S\Big[S\big[S[i-1]\big]\Big]$
  - etc.

Berechnung von $S[i]$: Pseudocode


```
h = S[i − 1]
while h ≥ 0 do
    if P[i − 1] == P[h] then
        S[i] = h + 1;  h = −2
    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 $S[i] \leq S[i-1]$:

- 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 $\boldsymbol{S[i] \leq S[i-1]}$:

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

Gesamtlaufzeit $\boldsymbol{T(m)}$:

$$T(m) \leq \sum_{i=2}^{m} (S[i-1] - S[i] + 2)$$

$$= 2(m-1) + (S[1] - S[2] + S[2] - S[3] + S[3] - \cdots + \ldots - S[m-1] + S[m-1] - S[m])$$

$$= 2(m-1) + S[1] - S[m] = O(m)$$

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