

Algoritmer og Sandsynlighed

Kompendium

Kevin Vinther

January 6, 2024

Contents

1	Basic Counting Problems	3
1.1	Pigeonhole	3
2	Inclusion Exclusion	3
3	Discrete Probability	3
4	Randomized Algorithms	3
5	Probabilistic Analysis	3
6	Indicator Random Variables	3
7	Universal Hashing	3
8	String Matching	3
8.1	Notation	3
8.1.1	Køretids Overview	5
8.2	Naive Algoritme	5
8.2.1	Køretid	6
8.3	Rabin-Karp	6
8.3.1	Forventede antal hits	8
8.4	Finite Automaton Based	9
8.5	Introduktion	9
8.6	Streng-matchende automat	10
8.7	Find Transition Function	14
9	Flows	14

10 Min-Cut

14

1 Basic Counting Problems

- Pigeonhole (inkl Generalized)
- Permutationer og Kombinationer
- Subsets med repetition
- Pascal's Trekant
- Binomialkoefficienter
- Bevis for binomialsætning vha kombinatorisk argument
- Bevis $n^2 + 1$ delsekvenser med mindst $n + 1$ der er strikt nedad- eller opadgående.

1.1 Pigeonhole

Dueslagsprincippet (pigeonhole principle) er et simpelt princip, men kan bruges til meget i beviser.

Theorem 1 (Dueslagsprincippet).

2 Inclusion Exclusion

3 Discrete Probability

4 Randomized Algorithms

5 Probabilistic Analysis

6 Indicator Random Variables

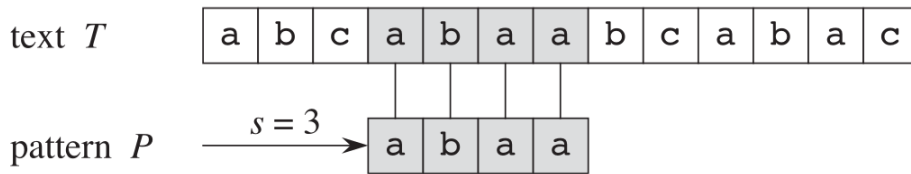
7 Universal Hashing

8 String Matching

8.1 Notation

Jeg tænker ikke at der skal snakkes om det her til eksamen, men følgende er en liste af notation der er nødvendige for for forståelse:

- **Streng**: Arrays med karakterer (ligesom i programmeringssprog)
- **Shift**: Hvor langt inde i en streng
- $P[1..m]$: Mønster med længde m
- $T[1..n]$: Tekst med længde n
- $T[1..n-m]$: Den tekst vi leder efter. Vi er ikke interesseret i de sidste m , da de er længere end mønsterstrengen.
- **P forekommer med shift s**: Du finder mønstret s karakterer inde i teksten.
- **Validt shift** et shift hvor mønsteret P forekommer
- **Invalidt shift** et shift hvor mønsteret P **ikke** forekommer.



- Σ^* (Sigma-Stjerne) er sættet af alle endelige strenge der kan bliver lavet fra karaktererne i Σ .
- ε , den **tomme streng**, er strengen uden noget indhold. Den er også en del af Σ^* .
- $|x|$ er længden af streng x .
- **Concatenation** af to strenge x og y , skrevet xy har længde $|x| + |y|$ og er karaktererne i x efterfulgt af karaktererne i y .
- **Præfiks** af streng x , denoted $w \sqsubset x$, gælder hvis $x = wy$ hvor $y \in \Sigma^*$, altså, w er en del af streng x i starten af strængen. y er den resterende del af streng x , som ikke er w .
- **Suffiks**: denoted $w \sqsupset x$ omvendt.

Lemma 2 (31.1 (Overlapping-suffix lemma) (Cormen)). *Suppose that x, y , and z are strings such that $x \sqsubset z$ and $y \sqsubset z$. If $|x| \leq |y|$, then $x \sqsubset y$. If $|x| \geq |y|$, then $y \sqsubset x$. If $|x| = |y|$ then $x = y$.*

Proof. Se Figur 1

□

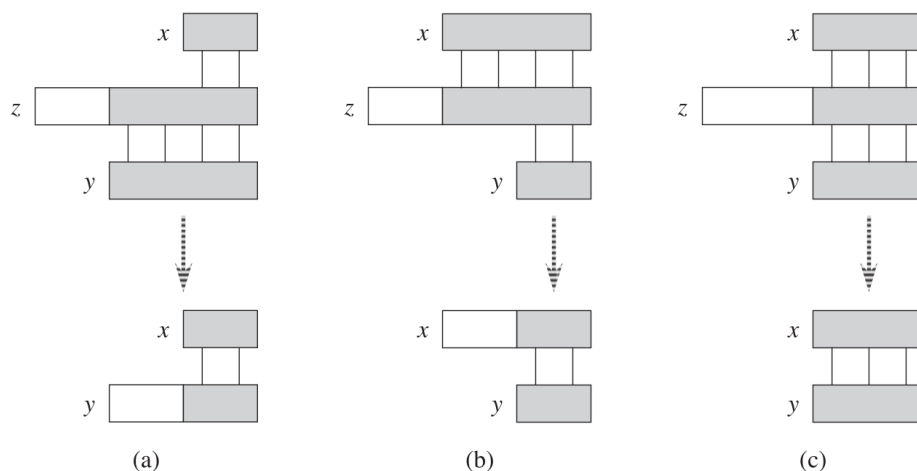


Figure 1: Vi antager at $x \sqsupseteq z$ og $y \sqsupseteq z$. De tre dele af figuren illustrerer de tre cases af lemmaet. **(a)** Hvis $|x| \leq |y|$, så $x \sqsupseteq y$. **(b)** Hvis $|x| \geq |y|$, så $y \sqsupseteq x$. **(c)** Hvis $|x| = |y|$, så er $x = y$.

Vi antager at tiden det tager for at finde ligheden mellem to strenge er $\Theta(t + 1)$ hvor t er størrelsen af den længste streng. +1, til hvis $t = 0$.

8.1.1 Køretids Overview

Algorithm	Preprocessing Time	Matching Time
<i>Naive</i>	0	$O((n - m + 1)m)$
<i>Rabin-Karp</i>	$\Theta(m)$	$O((n - m + 1)m)$
<i>Finite Automaton</i>	$O(m \Sigma)$	$\Theta(n)$
<i>Knuth-Morris-Pratt</i>	$\Theta(m)$	$\Theta(n)$

8.2 Naive Algoritme

- Hvad er den?
- Hvorfor er den dårlig?
- Hvad er worst-case?

Den naive algoritme er virkelig det, naiv. **Source Code:**

```
Naive-String-Matcher(T,P)
n = T.length
m = P.length
for s = 0 to n - m
    if P[1..m] == T[s+1..s+m]
        print "Pattern occurs with shift " s
```

8.2.1 Køretid

Den er virkelig skrald. Køretiden er $O((n - m + 1)m)$. Dens worst case sker hvis teksten er a^n og mønsteret der ledes efter er a^n (begge er mængder af a 'er, på længde hhv. m og n). I dette tilfælde finder den matches hver gnag, og der tager dermed $O(n^2)$ tid.

Der er **ingen** preprocessing tid, da der ikke skal gøres noget før algoritmen kører.

8.3 Rabin-Karp

- Hvad er hovedidéen?
- Hvorfor er den bedre end naive?

Trods at Rabin-Karp har en worst-case køretid på $\Theta((n - m + 1)m)$ er dens gennemsnitlige køretid bedre.

Algoritmen konverterer bogstaverne til tal, i radix- d notation, hvor d er størrelsen på alfabetet, $|\Sigma|$.

I følgende eksempler vil vi gå ud fra at $d = 10$, og $\Sigma = \{0, 1, \dots, 9\}$. Husk at $P[1..m]$ er mønsteret vi leder efter. Ved rabin-karp skelner vi mellem $P[1..m]$ og p , hvor p er dets decimalværdi. Dvs., hvis $P[1..m] = 1372$, så er $p = 1372$ i decimalværdi. Eksemplet virker forsimplet idet vores alfabet også er tal, men tænk hvis alfabetet var $\Sigma = \{a, b, \dots, j\}$, i dette tilfælde ville p ikke være ændret, men $P[1..m] = acgb$. Ydermere er teksten $T[1..n]$'s decimal counterpat t_s . Den bliver udregnet på samme måde. Hvis $t_s = p$ så $T[s + 1..s + m] = P[1..m]$.

Vi vil gerne have en måde hvorpå vi kan lave alfabetet om til tal, som vi kan regne på. Hvis vi kan konverterer mønsteret $P[1..m]$ til p på $\Theta(m)$ tid, så kan vi konvertere t_s på $\Theta(n - m + 1)$ tid. Til at gøre dette bruger vi **Horner's Rule**, som er meget vigtig at kende, se Definition 1.

Definition 1 (Horner's Rule). Horner's Rule er en regel hvorpå du hurtigt (specielt for computere) kan udregne decimaltal. Dette gør du ved at tage det sidste tal der skal udregnes først, derefter tager du tallet på 10'ernes plads, ganger det med 10^1 , etc. indtil du er ved d 'ende plads, og ganger det med 10^{d1} . Se følgende billede.

Example of calculation using Horner's rule

$$\begin{aligned}
 47632 &= 4 \cdot 10^4 + 7 \cdot 10^3 + 6 \cdot 10^2 + 3 \cdot 10 + 2 \cdot 10^0 \\
 &= 2 \cdot 10^0 + 3 \cdot 10^1 + 6 \cdot 10^2 + 7 \cdot 10^3 + 4 \cdot 10^4 \\
 &= 2 + 10(3 + 10(6 + 10(7 + 10 \cdot 4)))
 \end{aligned}$$

Noget af det smarteste med Horner's Rule, er at, når du går til næste værdi, så kan du udregne det hurtigt uden at tage det hele om igen. Dette giver køretid $\Theta(n - m)$:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s + 1]) + T[s + m + 1]$$

Forklaring 3. *Skip dette hvis du ikke har meget tid.* $10^{m-1} \cdot T[s + 1]$ fjerner det højeste ciffer. Ved at gange det med 10 skifter du tallet til venstre med en cifferposition. Ved at tilføje $T[s + m + 1]$ får du det nye, laveste ciffer.

Problem! p og t_s er muligvis for større til at de kan være i et computer word. Hvis dette er tilfældet, og P indeholder m karakterer, så tager vi tallet **modulo** q . $p \bmod q$ bliver udregnet på $\Theta(m)$ tid (størrelsen af p .) Alle t_s værdier i $\Theta(n - m + 1)$ tid.

Hvilken q skal vi dog vælge? Simpelt! Vælg et primtal således der er plads til $10q$ i én computer word. Derefter kan vi udføre alle udregninger simpelt. Ved at bruge modulo-udregning, ændrer Horner's udregning sig til at blive: $t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$, hvor $h \equiv d^{m-1}(\bmod q)$ er værdien af ciffret 1 i højeste position.

Problem igen! Hvad hvis $p \equiv t_s$, men $P[1..m] \neq T[s + 1..n - m]$? Altså, tallene er ens, men de er strengene ikke grundet modulo? Dette kalder vi et **spurious hit**, og er pisse irriterende, men desværre end nødvendig onde.

¹Dette gælder kun i base-10. Rabin-karp kører i base- b . Konverter dette til b^d

Derfor, når vi finder et **hit** om det er spurious eller ej, så tjekker vi også strengene.

RABIN-KARP-MATCHER(T, P, d, q)

```

1   $n = T.length$ 
2   $m = P.length$ 
3   $h = d^{m-1} \bmod q$ 
4   $p = 0$ 
5   $t_0 = 0$ 
6  for  $i = 1$  to  $m$                                 // preprocessing
7       $p = (dp + P[i]) \bmod q$ 
8       $t_0 = (dt_0 + T[i]) \bmod q$ 
9  for  $s = 0$  to  $n - m$                                 // matching
10     if  $p == t_s$ 
11         if  $P[1..m] == T[s + 1..s + m]$ 
12             print "Pattern occurs with shift"  $s$ 
13     if  $s < n - m$ 
14          $t_{s+1} = (d(t_s - T[s + 1])h + T[s + m + 1]) \bmod q$ 

```

Figure 2: Rabin Karp Algoritmen

Worst-case er i samme situation som ved den naive algoritme. Hvis teksten er en del a'er, og det samme med mønstret, så vil vi få en masse hits.

8.3.1 Forventede antal hits

Vi vil gerne finde det forventede antal hits. Vi antager at $\bmod q$ agerer som en tilfældig mapping (funktion) fra alfabetet til heltal base q , altså $\Sigma^* \rightarrow \mathbb{Z}_q$. Ydermere antager vi at alle værdierne modulo q er lige sandsynlige, i.e., $p(t_s \equiv p \bmod q) = \frac{1}{q}$. Det vil sige at antallet af falske hits er $\frac{O(n)}{q} = O(\frac{n}{q})$. Dette får vi fra antal af hvor mange der modulerer til samme værdi. Hvis der er 10 forskellige bogstaver, og vi er i base-3, så $\frac{10}{3} = 3.\bar{3}$ ca. tal der mapper til det samme.

Den forventede køretid bliver derfor $O(n) + O(m(v + \frac{n}{q}))$ hvor v er antallet af korrekte hits, det's køretid er $O(1)$ og $q \geq m$. Dermed bliver den totale

køretid $O(n + m) = O(n)$ da $n \geq m$.

8.4 Finite Automaton Based

- Hvordan laver man en DFA?

Jeg tænker **ikke** du behøver at forklare hvad en DFA er osv. Du får her en kort introduktion, som du bare kan springe over, givet at du forstår finite automata fint.

8.5 Introduktion

Mange algoritmer bygger en Finite Automata (herfra forkortet som DFA), da den er utroligt hurtig til at finde matches. Hver karakter bliver kigget på præcis én gang, og bruger tid $O(1)$ per gang den bliver kigget på. Efter maskinen bliver bygget er matching tiden $\Theta(n)$. Dog kan tiden der bruges til at bygge maskinen være meget stor hvis Σ er stort.

Definition 2 (Finite Automata). A **finite automaton** M , is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of **states**
- $q_0 \in Q$ is the **start state**
- $A \subseteq Q$ is a distinguished set of **accepting states**,
- Σ is a finite **input alphabet**,
- δ is a function from $Q \times \Sigma$ into Q called the **transition function** of M .

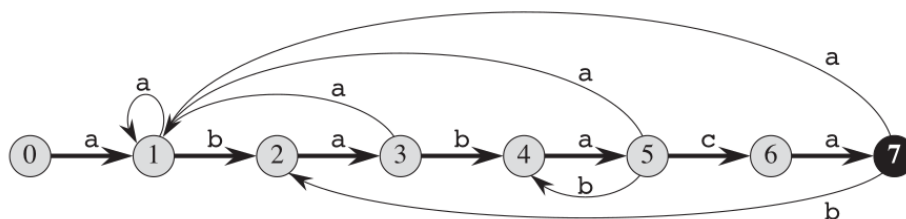
Ved hvert state, læser maskinen et input, og går fra den state, q , til den næste defineret state, $\delta(q, a)$. Hvis q er en del af A , og, efter strengen er blevet “spist”, ender maskinen i $q \in A$, så er strengen “accepteret”, ellers er den ikke.

Ydermere bliver funktionen ϕ defineret som **Final-state funktion** fra Σ^* til Q således at $\phi(w)$ er den state som M ender op i, efter den scanner strengen w . Så, M accepterer streng w hvis **og kun hvis**, $\phi(w) \in A$.

$$\begin{aligned}\phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \text{ for } w \in \Sigma^*, a \in \Sigma\end{aligned}\tag{1}$$

8.6 Streng-matchende automat

For at givet mønster P , vil vi lave en streng-matchende automat som preprocessing skridt før den bruges til at søge efter strengen. Se figur 3 for hvordan vi konstruerer automaten for mønstret $P = ababaca$.



(a)

state	input			P
	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figure 3: For mere information om figuren, se pp. 997 i Cormen

Definition 3 (Suffiks Funktion). Givet et mønster $P[1..m]$, definerer vi funktionen σ , kaldet **suffiks funktion** korresponderende til P . Funktionen σ mapper Σ^* til $\{0, 1, \dots, m\}$, således at $\sigma(x)$ er længden af det længste præfix af P som også er et suffiks af x :

$$\sigma(x) = \max\{k : P_k \sqsupseteq x\}$$

Suffiks Funktionen er **well-defined** (hvert element mapper til noget), da $P_0 = \varepsilon$ er et suffiks er hver streng.

Example 1 (Eksempler på Suffiks Funktionen). Givet mønstret $P = ab$, har vi $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$ og $\sigma(ccab) = 2$. Givet mønstret med længde m har vi $\sigma(x) = m$ hvis og kun hvis $P \sqsubset x$. Fra definitionen af suffiksfunktionen betyder $x \sqsubset y$ også at $\sigma(x) \leq \sigma(y)$.

Vi definerer en streng-matchende automat som korresponderende til et mønster $P[1..m]$ som følger:

- Sættet af states Q er $\{0, 1, \dots, m\}$. Start staten q_0 , og staten m er den eneste accepterede state.
- Transition function δ er defineret ved følgende ligning, for hver state q og karakter a :

$$\delta(q, a) = \sigma(P_q a) \quad (2)$$

Det vil sige, at givet en karakter a , vil vi gå fra state q til længden af det længste præfiks af P , som også er et suffiks af x . Så, altså, hvis a får dig én længere, vil du også gå én state tilbage. Men, hvis du går tilbage til kun at være 2 inde, så er du tilbage på state 2.

Forklaring 4 (Yderligere forklaring). Vi definerer $\delta(q, a) = \sigma(P_q a)$ fordi vi vil holde fast i det længste præfiks af mønstret P der har matchet strengen T indtil videre.

Antag at $p = \phi(T_i)$, så, efter at have læst T_i , så er automatonet i state q . Vi designer δ således at state q fortæller os længden af det længste præfiks af P der er et suffiks af T_i . Det vil sige, i state q , $P_q \sqsubset T_i$ og $q = \sigma(T_i)$. Dette vil også sige at **hvis $q = m$, så har vi fundet et match!** Dermed, siden $\phi(T_i)$ og $\sigma(T_i)$ begge er lig q , ser vi at automaten holder følgende invariant:

$$\phi(T_i) = \sigma(T_i) \quad (3)$$

Dermed, hvis vi er i state q , og automaten læser karakter $T[i + 1] = a$, så skal vores transition lede til det korresponderende længste præfiks af P som er et suffiks af $T_i a$. Den state er $\sigma(T_i a)$.

Fordi P_q er det længste præfiks af P som er et suffiks af T_i , så er det længste præfiks af P som er et suffiks af $T_i a$ ikke kun $\sigma(T_i a)$, men også $\delta(P_q a)$. (Dette bliver bevist senere)

Der er to states vi skal kigge på, den første, $a = P[q + 1]$, så er $\delta(q, a) = q + 1$. Den næste, $a \neq P[q + 1]$, så skal vi finde et mindre præfiks af P som også er et suffiks af T_i .

Lad os kigge på et eksempel. Streng-matching automaten fra Figur 3 har $\delta(5, c) = 6$, som så er first case, hvor vi bare går videre. Et sekmepl på second case er $\delta(5, b) = 4$. Vi laver denne transition fordi, hvis automaten læser et b når $q = 5$, så $P_q b = ababab$, og det længste præfiks af P som også er et suffiks af $ababab$ er $P_4 = abab$.

Følgende er algoritmen for at lave en finite automata til streng-matching. Sættet af states $Q = \{0, 1, \dots, m\}$, start staten $q_0 = 0$, den eneste accepting state er m , $m \in A$.

Finite-Automaton-Matcher(T, d, m):

```

n = T.length
q = 0
for i = 1 to n
    q = d(q, T[i])
    if q == m
        print "Pattern occurs with shift" i - m

```

Figure 4:

Transition function er som beskrevet tidligere. Hvis dette ikke er klart, se Forklaring 4.

Det er her nemt at se at køretiden på en tekst-streng af længde n er $\Theta(n)$. Før vi viser pre-processing tid, kigger vi på et bevis for at algoritmen kører som forventet.

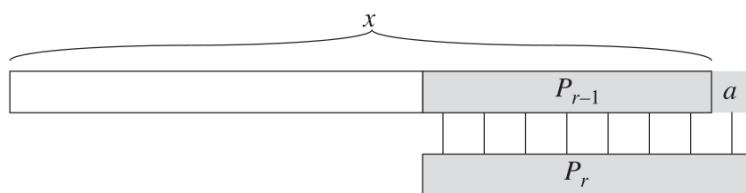


Figure 5: En illustration til beviset af Lemma 5. Figuren viser $r \leq \delta(x) + 1$, hvor $r = \delta(xa)$

Lemma 5 (Suffix-Function Inequality). *For hver streng x og karakter a , har vi at $\sigma(xa) \leq \sigma(x) + 1$.*

Proof. Se Figur 5. Hvis $r = 0$, så $\sigma(xa) = r \leq \delta(x) + 1$ er trivielt løst, da $\sigma(x)$ er nonnegativt. Antag at $r > 0$, så $P_r \sqsubset xa$ per definition af σ . Dermed, $P_{r-1} \sqsubset x$, ved at fjerne a fra enden af P_r og fra enden af xa . Dermed $r - 1 \leq \sigma(x)$, siden $\sigma(x)$ er det største k således $P_k \sqsubset x$, og således $\sigma(xa) = r \leq \sigma(x) + 1$ \square

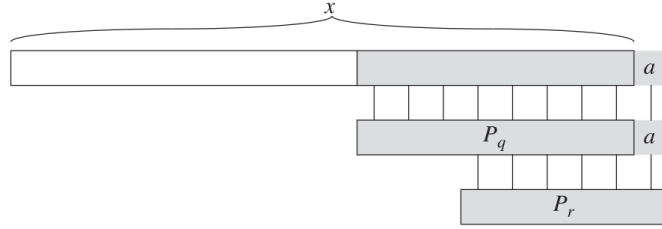


Figure 6: Illustration af beviset for Lemma 6. Figuren viser at $r = \delta(P_q a)$, hvor $q = \sigma(x)$ og $r = \sigma(xa)$

Lemma 6 (Suffix-Function Recursion Lemma). *For enhver streng x og karakter a , hvis $q = \sigma(x)$, så $\sigma(xa) = \sigma(P_q a)$.*

Proof. Vi ved fra definitionen af σ at $P_q \sqsubset x$. Som vist i figur 6, har vi også $P_q a \sqsubset xa$. Hvis $r = \sigma(xa)$, så $P_r \sqsubset xa$ og, gennem Lemma 5, $r \leq q + 1$. Dermed har vi at $|P_r| = r \leq q + 1 = |P_q a|$. Derfor, $r \leq \sigma(P_q a)$, dermed $\sigma(xa) \leq \sigma(P_q a)$. Vi har dog også $\sigma(P_q a) \leq \sigma(xa)$, siden $P_q a \sqsubset xa$. Dermed $\sigma(xa) = \sigma(P_q a)$ \square

Tid til det vigtigste skridt. Vi skal vise at automaten vedligeholder invarianten i ligning 2.

Theorem 7. *If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then $\phi(T_i) = \sigma(T_i)$ for $i = 0, 1, \dots, n$.*

Proof. Vi beviser gennem induktion på i .

Ved $i = 0$ er teoremet sandt, da $T_0 = \varepsilon$ dermed $\phi(T_0) = 0 = \sigma(T_0)$. \square

Vi antager nu at $\phi(T_i) = \sigma(T_i)$ og beviser at $\phi(T_{i+1}) = \sigma(T_{i+1})$. Lad q være $\phi(T_i)$, og lad a være $T[i + 1]$. Så:

$$\begin{aligned}
\phi(T_{i+1}) &= \phi T(i) \text{ (fra definition på } T_{i+1} \text{ og } a) \\
&= \delta(\phi(T_i), a) \text{ (fra definitionen på } \phi) \\
&= \delta(q, a) \text{ (af definitionen på } q) \\
&= \sigma(P_q a) \text{ (fra definitionen tidligere)} \\
&= \sigma(T_i a) \\
&= \sigma(T_{i+1})
\end{aligned}$$

8.7 Find Transition Function

Vi vil gerne finde transition funktion. Vi har allerede defineret den tidligere, men vi vil have en algoritmisk metode hvorpå vi kan gøre det.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```

1   $m = P.length$ 
2  for  $q = 0$  to  $m$ 
3      for each character  $a \in \Sigma$ 
4           $k = \min(m + 1, q + 2)$ 
5          repeat
6               $k = k - 1$ 
7          until  $P_k \sqsubset P_q a$ 
8               $\delta(q, a) = k$ 
9  return  $\delta$ 

```

Figure 7: Algoritmen for at finde transition function på.

Køretiden på algoritmen er $O(m^3|\Sigma|)$.

9 Flows

10 Min-Cut