



*mn plaatjes nog toevoegen*

## Uitwerkingen

1. (Dit is ongeveer opgave R-9.2.)

We passen het brute-force pattern matching algoritme toe op de volgende tekst  $T$  en patroon  $P$ :

$$\begin{aligned} T &= aaabaadaabaaa \\ P &= aabaaa \end{aligned}$$

We nummeren de stappen, en schrijven ! achter een stap met een mismatch, en  $m$  achter een stap waarin tot een match ( $P$  zit in  $T$ ) geconcludeerd wordt. Bij een mismatch schuiven we het patroon één positie naar rechts.

$a$	$a$	$a$	$b$	$a$	$a$	$d$	$a$	$a$	$b$	$a$	$a$	$a$
1	2	3!										
$a$	$a$	$b$	$a$	$a$	$a$							
	4	5	6	7	8	9!						
	$a$	$a$	$b$	$a$	$a$	$a$						
		10	11!									
		$a$	$a$	$b$	$a$	$a$	$a$					
			12!									
			$a$	$a$	$b$	$a$	$a$	$a$				
				13	14	15!						
				$a$	$a$	$b$	$a$	$a$	$a$			
					16	17!						
					$a$	$a$	$b$	$a$	$a$	$a$		
						18!						
						$a$	$a$	$b$	$a$	$a$	$a$	
							19	20	21	22	23	24 $m$
							$a$	$a$	$b$	$a$	$a$	$a$

2. We passen het brute-force pattern matching algoritme toe op het tweede pattern matching probleem:

$$\begin{aligned} T &= 000010001010001 \\ P &= 000101 \end{aligned}$$

Nog toevoegen.

3. (Dit is ongeveer opgave C-9.1.)

Een tekst  $T$  ter lengte  $n$  en een patroon  $P$  ter lengte  $m$  (met  $n > m$ ) zo dat het brute-force pattern matching algoritme  $m \cdot (n - m + 1)$  vergelijkingen doet:  $T = a^n$  en  $P = a^{m-1}b$ .

Bijvoorbeeld voor  $n = 8$  en  $m = 3$ :

$a$	$a$	$a$	$a$	$a$	$a$	$a$	$a$
1	2	3!					
$a$	$a$	$b$					
	4	5	6!				
	$a$	$a$	$b$				
		7	8	9!			
		$a$	$a$	$b$			
			10	11	12!		
			$a$	$a$	$b$		
				13	14	15!	
				$a$	$a$	$b$	
					16	17	18!
					$a$	$a$	$b$

4. Stel we hebben een patroon is  $P = p_1 \dots p_m$  met allemaal verschillende karakters, en een tekst  $T$  ter lengte  $n$ . Idee: Loop door de tekst  $T$  heen op zoek naar  $p_1$ . Als je  $p_1$  gevonden hebt, begin je vanaf daar te kijken of het patroon matcht. Bij een mismatch op  $T[i]$ : zoek vanaf  $i$  (inclusief  $i$  zelf) naar  $p_1$  in  $T$ . Dit is in  $\mathcal{O}(n)$ .

Voorbeeld (opschuiven-en-zoeken wordt met – aangegeven):

$a$	$b$	$d$	$b$	$b$	$a$	$b$	$a$	$b$	$c$
1	2	3!							
$a$	$b$	$c$							
		4	5	6	7!				
		–	–	–	–				
					$a$	8	9!		
					$b$	$b$	$c$		
							10!		
							–		
								11	12 $m$
							$a$	$b$	$c$

(Kan nog beter door  $T[i]$  niet twee keer te bekijken.)

5. \* Stel we staan in het patroon  $P$  het speciale symbool  $*$  toe, dat matcht met een willekeurige string (ter lengte 0 of groter). De  $*$  mag willekeurig vaak in het patroon  $P$  voorkomen, maar nooit in de tekst  $T$ .

Voorbeeld: Het patroon  $ab * ba * c$  zit als volgt in de tekst  $cabccbacbacab$ :

```

c  ab  cc  ba  cba  c  ab
ab *  ba  *   c

```

Geef een pattern matching algoritme voor zulke patronen en teksten, en analyseer de tijdscomplexiteit van je algoritme.

We gaan uit van het brute-force pattern-matching algoritme. Idee: een patroon bestaat uit nul, één of meer substrings, gescheiden door sterretjes. We zoeken eerst de eerste substring in de tekst. Als dat gelukt is zoeken we de volgende substring in de rest van de tekst. Enzovoorts.

Input: (Na preprocessing verkregen)  $k$  patronen  $P_1, \dots, P_k$  die de maximale substrings zonder  $*$  zijn van  $P$  (ze komen dus niet direct achter elkaar in  $P$  voor).

**Algorithm** MatchSter( $T, P_1, \dots, P_k$ ):

```

s := 0
outputlist := null
for q := 1 to k do
  i_q := BruteForceMatch(T[s..n], P_q)
  outputlist.insertLast(i_q)
  s := i_q + m

```

De tijdscomplexiteit: Met  $m_1, \dots, m_k$  de lengtes van  $P_1, \dots, P_k$  en  $n$  de lengte van  $T$ :  $m_1 \cdot n + m_2 \cdot (n - m_1) + \dots + m_k \cdot (n - m_1 - \dots - m_{k-1}) \leq k \cdot m \cdot n$ . Dus in  $\mathcal{O}(kmn)$  met  $k$  het aantal substrings.

6. (Dit is ongeveer opgave R-9.3.)

(a) De *last* functie van het Boyer-Moore pattern matching algoritme voor het patroon  $P$  uit het eerste pattern matching probleem, en het alfabet  $\{a, b, c, d\}$ .

(NB de indices in  $P$  lopen van 0 tot  $m - 1$ .)

	a	b	c	d
last	5	2	-1	-1

(b) We passen het Boyer-Moore pattern matching algoritme toe op het eerste pattern matching probleem:

$a$	$a$	$a$	$b$	$a$	$a$	$d$	$a$	$a$	$b$	$a$	$a$	$a$
			$3!$	$2$	$1$							
$a$	$a$	$b$	$a$	$a$	$a$							
		$a$	$a$	$b$	$a$	$a$	$4!$					
							$10m$	$9$	$8$	$7$	$6$	$5$
							$a$	$a$	$b$	$a$	$a$	$a$

7. Herhaal (eventueel) de vorige opgave voor het tweede pattern matching probleem.

nog toevoegen.

8. (Dit is ongeveer opgave R-9.6.)

We doorlopen  $P$  van rechts naar links; dit is in  $\mathcal{O}(m)$ . Het initialiseren van alle waardes op  $-1$  is in  $\mathcal{O}(s)$ .

NB: worst-case tijdscomplexiteit voor het Boyer-Moore pattern matching algoritme wordt bereikt voor  $T = a^n$  en  $P = ba^{m-1}$ .

9. Kun je het Boyer-Moore pattern matching algoritme aanpassen zó dat het *alle* (nul of meer) voorkomens van een patroon  $P$  in een tekst  $T$  vindt?

Idee: return in plaats van alleen een rank  $i$  een lijstje met ranks. Als het lijstje leeg is dan is er geen match, als het lijstje niet leeg is bevat het de posities in  $T$  waar het patroon  $P$  begint. In het algoritme zoals in het boek (index  $i$  loopt door de tekst  $T$  en index  $j$  loopt door het patroon  $P$ ) vervangen we de regel met **return**  $i$  door:

```

insertLast(i)
i := i + m
j := m - 1

```

Hier kijken we puur naar meerdere voorkomens van  $P$  in  $T$ . Dus bijvoorbeeld het patroon  $aaa$  komt 4 keer voor in de tekst  $aaaaaa$ . Je zou ook kunnen kijken naar voorkomens die ‘tegelijk’ in  $T$  voorkomen, dan komt het patroon  $aaa$  maar 2 keer voor in de tekst  $aaaaaa$ .

10. (Dit is ongeveer opgave R-9.4.)

(a) De *failure* functie van het Knuth–Morris–Pratt pattern matching algoritme voor het patroon  $P$  uit het eerste pattern matching probleem:

$i$	0	1	2	3	4	5
$P[i]$	$a$	$a$	$b$	$a$	$a$	$a$
$f(i)$	0	1	0	1	2	2

$f(i)$  is de lengte van de langste prefix van  $P$  die een suffix is van  $P[1 \dots i]$ .

- (b) We passen het Knuth–Morris–Pratt pattern matching algoritme toe op het eerste pattern matching probleem:

$a$	$a$	$a$	$b$	$a$	$a$	$d$	$a$	$a$	$b$	$a$	$a$	$a$	
$a$	$a$	$b$	$a$	$a$	$a$								
1	2	3!											
		4	5	6	7	8!							
		$a$	$a$	$b$	$a$	$a$							
					$a$	$a$	9!						
						$a$	$a$	$a$					
						$a$	10!						
							$a$	$b$	$a$	$a$	$a$		
								11!					
							$a$	$a$	$b$	$a$	$a$	$a$	
								12	13	14	15	16	17 $m$
								$a$	$a$	$b$	$a$	$a$	$a$

11. Herhaal (eventueel) de vorige opgave voor het tweede pattern matching probleem.  
nog toevoegen.
12. (Dit is ongeveer opgave C-9.2.)

We laten zien dat het algoritme om de failure functie te berekenen (zie boek) in  $\mathcal{O}(m)$  is, met  $m$  de lengte van het patroon. We onderscheiden drie gevallen, analoog aan de analyse van het algoritme *KMPMatch*. We laten zien dat in alledrie de gevallen ofwel de index  $i$  toeneemt en  $j - i$  gelijk blijft, ofwel  $j - i$  toeneemt en  $i$  gelijk blijft, ofwel  $i$  en  $j - i$  alletwee toenemen. Verder geldt  $i \leq m$  en  $j - i \leq m$ .

- (a) Geval  $P[j] = P[i]$ . Dan neemt  $i$  toe (met 1), en  $i - j$  blijft gelijk.
- (b) Geval  $P[i] \neq P[j]$ , en  $j > 0$ . Dan blijft  $i$  gelijk, en  $j$  wordt  $f(j - 1)$ . Er geldt:  $f(j - 1)$  is de lengte van de langste prefix van  $P$  die een suffix is van  $P[1 \dots (j - 1)]$ . Dus  $f(j - 1) \leq (j - 1)$  dus  $f(j - 1) < j$ , dus  $i - j < i - f(j - 1)$ .
- (c) Geval  $P[i] = P[j]$ , en  $j = 0$ . Dan neemt  $i$  toe (met 1), en ook  $i - j$  neemt toe met 1.

De iteratie wordt dus ten hoogste  $2m$  keer uitgevoerd, met  $m$  de lengte van het patroon  $P$ . Dus het algoritme voor de failure-functie is in  $\mathcal{O}(m)$ .

13. \* Over de tijdscomplexiteit van het Knuth–Morris–Pratt algoritme, niet echt me aggregate analysis, maar meer zoals in het boek (p427).

We gaan ervan uit dat de failure functie al berekend is. De tijdscomplexiteit van het algoritme `KMPMatch` wordt bepaald door het aantal keren dat de while-loop wordt uitgevoerd. Het is niet direct duidelijk hoe vaak dat is, want in het eerste en derde geval neemt  $i$  toe, maar in het tweede geval blijft  $i$  gelijk. Dus het aantal keer dat de while-loop wordt uitgevoerd wordt niet begrensd door  $n$ . We laten zien dat het wel begrensd wordt door  $2n$ .

We hebben  $i \leq n$  waarbij  $i$  de index is die door  $T$  loopt, en  $i - j \geq n$ , met  $j$  de index die door  $P$  loopt. In elk van de drie mogelijke gevallen in de while-loop voor  $i$  en voor  $i - j$ : ze nemen geen van tweeën af, en ten minste één van de twee wordt groter. In het eerste geval wordt  $i$  groter en  $i - j$  blijft gelijk. In het tweede geval blijft  $i$  gelijk, en  $i - j$  wordt groter omdat  $j$  kleiner wordt. In het derde geval wordt  $i$  groter, en omdat  $j$  gelijk blijft wordt daarmee ook  $i - j$  groter. Totaal wordt de while-loop dus maximaal  $2n$  keer uitgevoerd. Daarmee is het algoritme in  $\mathcal{O}(n)$ .

14. Als een codering van strings prefix-free is, dan is hij niet ambigu. Toon met een voorbeeld aan dat de omgekeerde implicatie niet geldt, dwz, geef een voorbeeld van een codering die niet prefix-free is maar wel non-ambigu. Gebruik 01 en 0.

15. De frequentietabel is als volgt:

h	o	w	m	u	c	d	l	a	k
4	6	4	1	4	5	3	1	1	2

Een Huffman-coderingsboom :  
nog toevoegen

16. De initialiseringsstap is in  $\mathcal{O}(n)$  want we moeten elk van de totaal  $n$  karakters van de input-string  $X$  bekijken. In de for-loop wordt een min-heap van totaal  $d$  karakters gemaakt. Dit is in  $\mathcal{O}(d)$ . In de while-loop wordt twee keer verwijderd van en één keer toegevoegd aan een priority queue  $Q$ . Als de priority queue geïmplementeerd is als heap kan dit in  $\log d$ ; en de while-loop wordt  $d$  keer uitgevoerd. De tijdscomplexiteit van de while-loop is dus in  $d\mathcal{O}(d)$  Het hele algoritme is dus in  $\mathcal{O}(n + d \log(d))$ .
17. In plaats van twee binaire bomen met minimale frequenties samen te voegen tot één nieuwe binaire boom, voegen we nu drie bomen met minimale frequenties samen tot één nieuwe ternaire boom. Als we op deze manier het voorbeeld van opgave 15 uitwerken krijgen we inderdaad een goede codering, maar hij is niet optimaal omdat we de 2 niet gebruiken (maar wel lagere strings).

Wat moet er nog anders? Zorg dat we starten met een goed aantal bladeren voor het vormen van een ternaire boom, dus  $3 + 2k$  bladeren voor zekere  $k$ . Als we een blad tekort komen, voegen we een dummy-blad toe

met frequentie 0. Daarna voegen we steeds drie bomen met miniamle frequentie samen tot een nieuwe boom, dus analoog aan het algoritme voor binaire coderingen.

plaatjes toevoegen.

18. Een standard trie voor de strings  $\{abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca\}$ .  
plaatje toevoegen