Hoofdstuk 2

Strings en Patroonherkenning

Strings in Scheme: Constructoren

Een string is een eindige sequentie van karakters

<u>constructoren</u>:

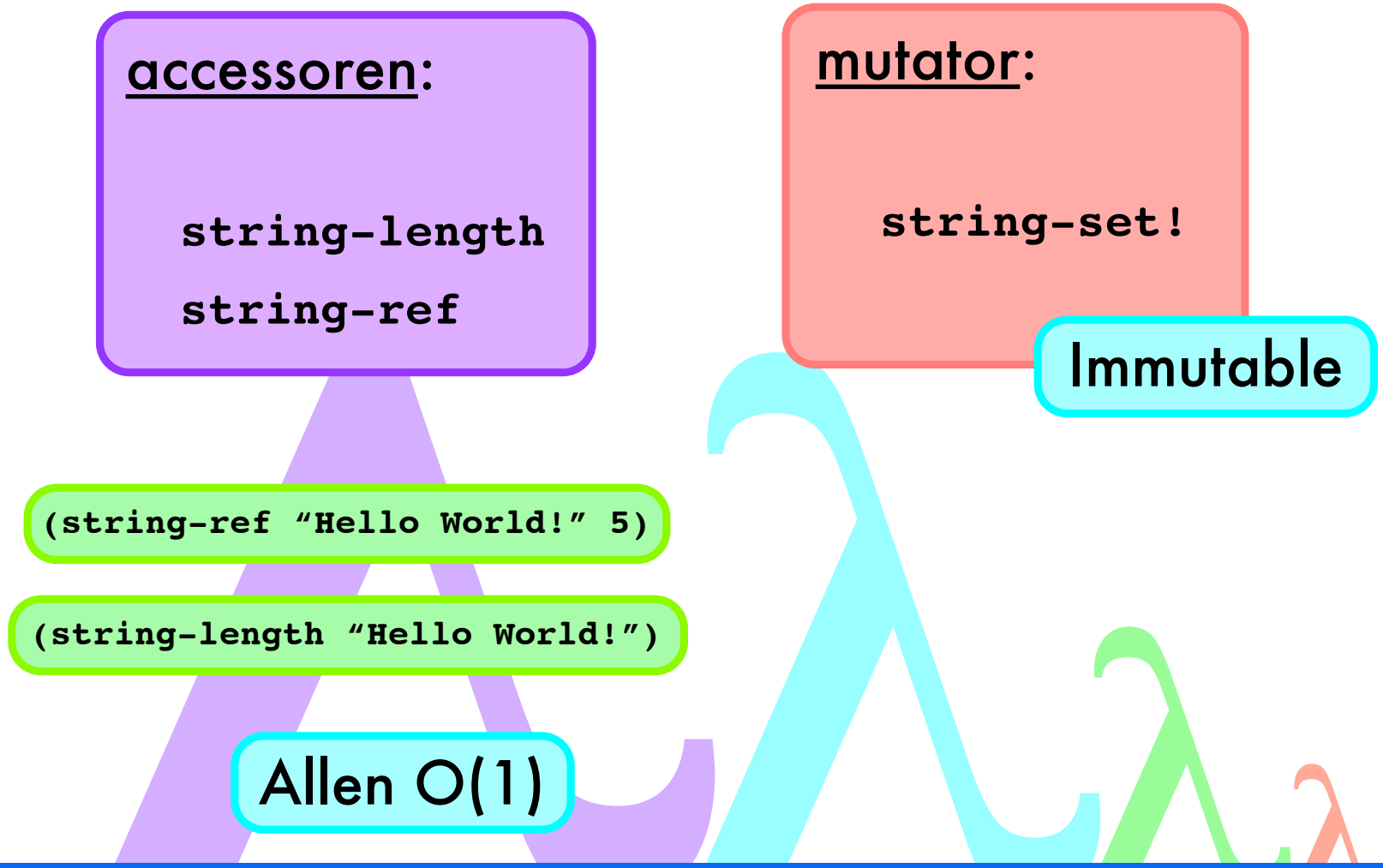
```
" ..."
(make-string n c)
(string ...)
```

```
"Madam, I'm Adam"
```

```
(make-string 5 #\a)
```

```
(string #\S #\c #\h #\e #\m #\e)
```

Strings: Accessoren & Mutatoren



ASCII waarden 0 - 127

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	☺	SOH	033	!	065	A	097	a
002	☻	STX	034	"	066	B	098	b
003	♥	ETX	035	#	067	C	099	c
004	♦	EOT	036	$	068	D	100	d
005	♣	ENQ	037	%	069	E	101	e
006	♠	ACK	038	&	070	F	102	f
007	(beep)	BEL	039	'	071	G	103	g
008	◘	BS	040	(	072	H	104	h
009	(tab)	HT	041	)	073	I	105	i
010	(line feed)	LF	042	*	074	J	106	j
011	(home)	VT	043	+	075	K	107	k
012	(form feed)	FF	044	,	076	L	108	l
013	(carriage return)	CR	045	-	077	M	109	m
014	♫	SO	046	.	078	N	110	n
015	☼	SI	047	/	079	O	111	o
016	►	DLE	048	0				
017	◄	DC1	049	1				
018	↕	DC2	050	2				
019	!!	DC3	051	3				
020	¶	DC4	052	4				
021	§	NAK	053	5				
022	▬	SYN	054	6				
023	↨	ETB	055	7		W		w
024	↑	CAN	056	8	088	X	120	x
025	↓	EM	057	9	089	Y	121	y
026	→	SUB	058	:	090	Z	122	z
027	←	ESC	059	;	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	|
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	_	127	⌂

American Standard Code for Information Interchange

ASCII waarden 128-255

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
128	Ç	160	á	192	└	224	α
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Γ
131	â	163	ú	195	├	227	π
132	ä	164	ñ	196	─	228	Σ
133	à	165	Ñ	197	+	229	σ
134	å	166	ª	198	╞	230	µ
135	ç	167	º	199	╟	231	τ
136	ê	168	¿	200	╚	232	Φ
137	ë	169	⌐	201	╔	233	Θ
138	è	170	¬	202	╩	234	Ω
139	ï	171	½	203	╦	235	δ
140	î	172	¼	204	╠	236	∞
141	ì	173	¡	205	═	237	∅
142	Ä	174	«	206	╬	238	ε
143	Å	175	»	207	╧	239	∩
144	É	176	▒	208	╨	240	≡
145	æ	177	▒				
146	Æ	178	▓				
147	ô	179	│				
148	ö	180	┤				
149	ò	181	╡				
150	û	182	╢				
151	ù	183	╖				
152	ÿ	184	╕				
153	Ö	185	╣	217	┘	249	·
154	Ü	186	║	218	┌	250	·
155	¢	187	╗	219	▒	251	√
156	£	188	╝	220	▒	252	n
157	¥	189	╜	221	▌	253	2
158	Pt	190	╛	222	▐	254	■
159	ƒ	191	┐	223	▒	255	(blank 'FF')

Eigenlijk voorbijgestreefd door Unicode standaard

Strings: Operaties

vergelijken

```
(string=? s1 s2)
(string-ci=? s1 s2)
(string<? s1 s2)
(string>? s1 s2)
(string<=? s1 s2)
(string>=? s1 s2)
(string-ci<? s1 s2)
(string-ci>? s1 s2)
(string-ci<=? s1 s2)
(string-ci>=? s1 s2)
```

O(min(n,m))

conversies

```
string->list
list->string
```

O(n)

Lexicografische orde

```
(char<? c1 c2)
        ⇔
(< (char->integer c1)
   (char->integer c2))
```

Strings: Operaties

"optellen en aftrekken"

```
string-append
substring
```

```
> (string-append "Hello" " " "World" "!")
"Hello World!"
```

```
> (define s "Hello World!")
> (substring s 2 4)
"ll"
> s
"Hello World!"
```

Het patroonherkenningsprobleem

Gegeven een tekst (of "hooiberg")

```
"Deze tekst is een voorbeeld van zo'n hooiberg"
```

Gegeven een patroon (of "naald")

```
"voorbeeld"
```

Wat is de index van de naald in de hooiberg?

Heet ook "offset" of "shift"

```
18
```

Toepassingen in tekstverwerkers en bioinformatica

Notaties

Het patroon p

De tekst t

De lengte van het patroon n_{-p} of n_p

De lengte van de tekst n_{-t} of n_t

Een string s die uit twee delen u en v bestaat noteren we als s = u.v

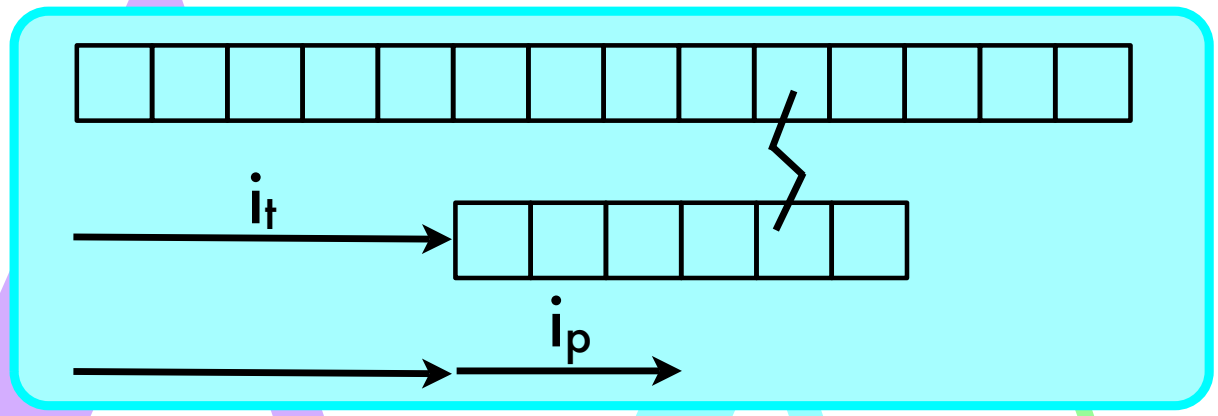
Als voor een string s = u.v dan heet u een prefix van s en v een suffix

De lengte van een string s is het aantal karakters in s. Notatie: |s|

Bij een string s is s_k het k'de karakter. $s_{i \to j}$ is de substring van i t/m j

Het Brutekracht Algoritme

Een "brutekracht algoritme" is een algoritme dat een probleem oplost door alle combinaties uit te proberen.



een "alignment"

een "match"

een "mismatch"

Het Brutekracht Algoritme

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t 1) 0)))))
```

Performantie

De `loop` wordt uitgevoerd voor `i-t` gaande van `0` tot `n-t - n-p`. Voor elke `i-t` gaat `i-p` van `0` tot `n-p`.

Dus:

$b_{match}(n_t,n_p) \in O(1)$

$r_{match}(n_t,n_p) \in O(n_t.n_p)$

En Dus:

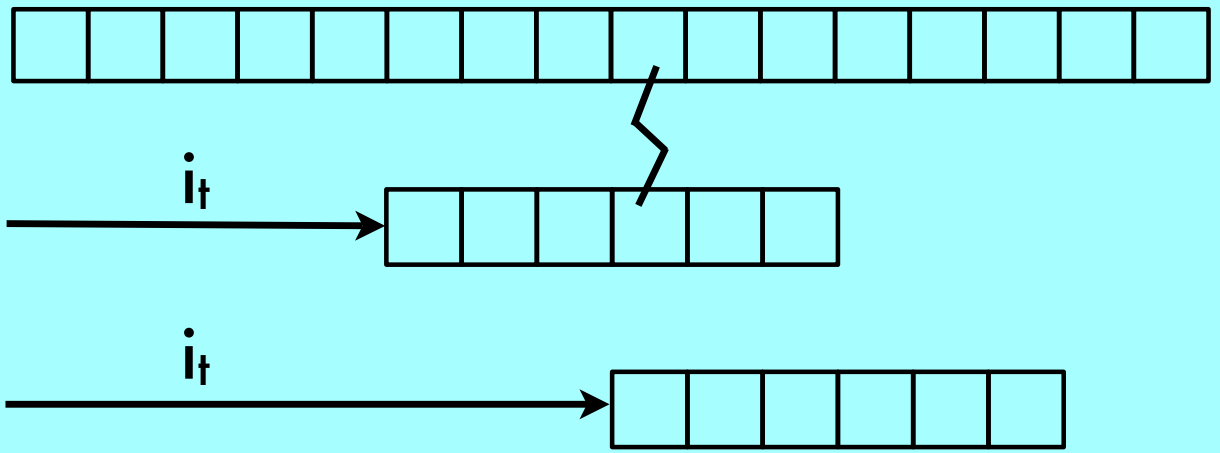
$f_{match}(n_t,n_p) \in O(n_t.n_p)$

n_t noch n_p zijn constant!

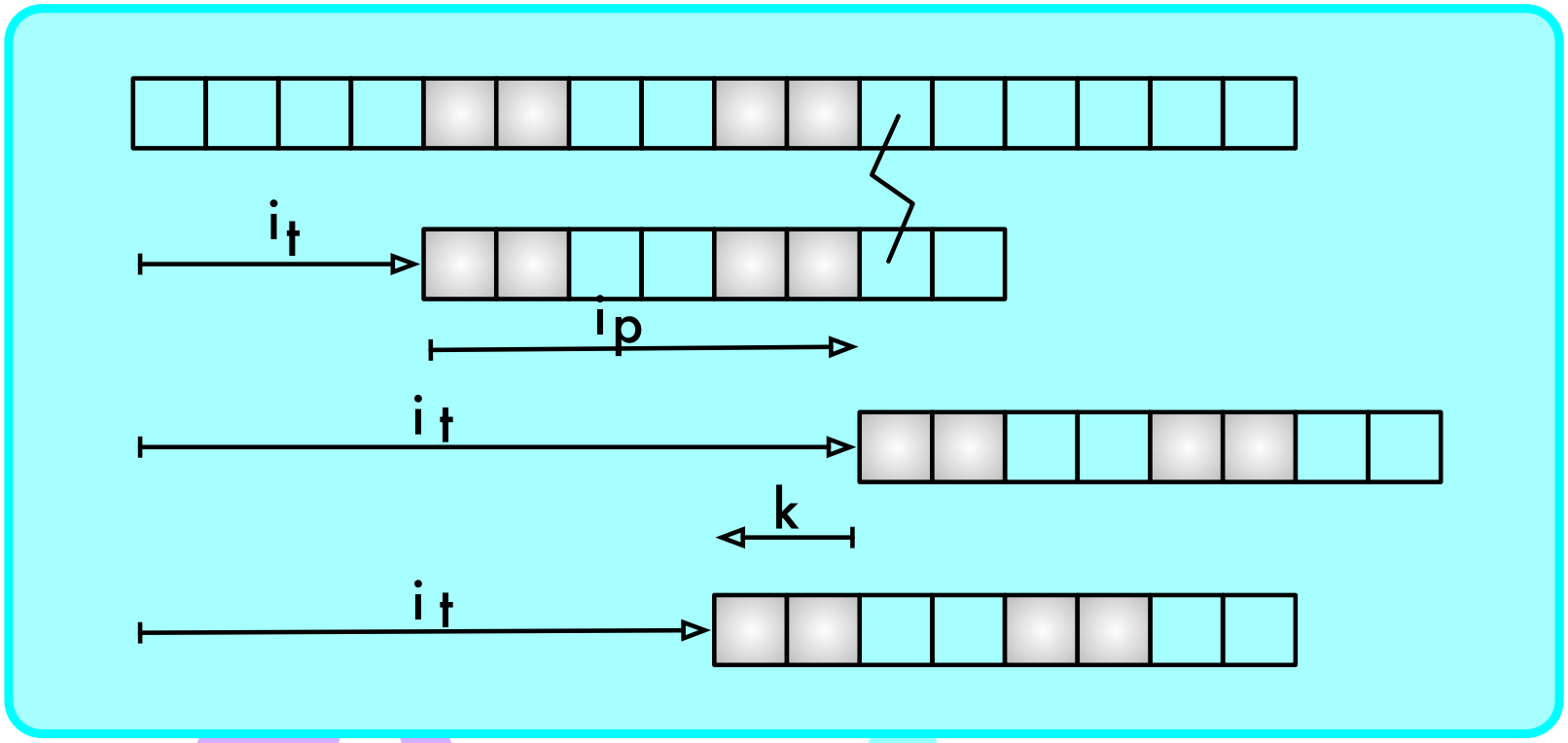
Het Knuth-Morris-Pratt Algoritme

Uitgevonden in 1977 door Knuth&Pratt, en onafhankelijk door Morris. Samen gepubliceerd.

Naief idee:



Probleem: Herhalingen



Oplossingen worden overgeslagen

Dus: Maximale terugkeer k zoeken

Terugkeren met σ

Het getal k hangt niet van de tekst af, enkel van het patroon.

Basisidee van KMP: stel vóór het matchproces op basis van het patroon een functie σ op zodat $k=\sigma(i_p)$. Dit heet preprocessing.

σ heet de failure function

Bij het matchen verschuif je het patroon ter waarde van $i_p - \sigma(i_p)$

De eerste $\sigma(i_p)$ moeten we niet meer checken.

Het KMP algoritme

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define sigma (compute-failure-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (loop (+ i-t (- i-p (sigma i-p))) (if (> i-p 0)
                                             (sigma i-p)
                                             0))))))
```

Per conventie is $\sigma(0) = -1$

Voorbeeld

```
(KMP "I'm singing lalala down in lalaland" "lalaland")
```

```
i-t = 0 i-p = 0
i-t = 1 i-p = 0
i-t = 2 i-p = 0

...

i-t = 7 i-p = 0
i-t = 8 i-p = 0
i-t = 9 i-p = 0
i-t = 10 i-p = 0
i-t = 11 i-p = 0
i-t = 12 i-p = 0, 1, 2, 3, 4, 5, 6
i-t = 14 i-p = 4
i-t = 16 i-p = 2
i-t = 18 i-p = 0

...

i-t = 24 i-p = 0
i-t = 25 i-p = 0
i-t = 26 i-p = 0
i-t = 27 i-p = 0, 1, 2, 3, 4, 5, 6, 7, 8
```

Wat is de betekenis van σ ?

```
text    = b a b b a b b a b b a b c
pattern = b a b b a b b a b c
```

naief

```
text    = b a b b a b b a b b a b c
pattern =                   b a b b a b b a b c
```

σ

```
text    = b a b b a b b a b b a b c
pattern =           b a b b a b b a b c
pattern =     b a b b a b b a b c
```

Zoek het <u>langste</u> prefix van p dat ook suffix is van $p_{0 \to i p-1}$

Het opstellen van σ

```
(define (compute-failure-function p)
  (define n-p (string-length p))
  (define sigma-table (make-vector n-p 0))
  (let loop
    ((i-p 2)
     (k 0))
    (cond
      ((>= i-p n-p)
       (vector-set! sigma-table (- n-p 1) k))
      ((eq? (string-ref p k)
            (string-ref p (- i-p 1)))
       (vector-set! sigma-table i-p (+ k 1))
       (loop (+ i-p 1) (+ k 1)))
      ((> k 0)
       (loop i-p (vector-ref sigma-table k)))
      (else ; k=0
       (vector-set! sigma-table i-p 0)
       (loop (+ i-p 1) k))))
  (vector-set! sigma-table 0 -1)
  (lambda (q)
    (vector-ref sigma-table q)))
```

Voorbeeld

```
i-p = 2 / k = 0
    #(l a l a l a n d)
    #(0 0 0 0 0 0 0 0)
plaats 0 in 2 en start k=0
```

```
i-p = 3 / k = 0
    #(l a l a l a n d)
    #(0 0 0 1 0 0 0 0)
plaats 1 in 3 en doe voort
```

```
i-p = 4 / k = 1
    #(l a l a l a n d)
    #(0 0 0 1 2 0 0 0)
plaats 2 in 4 en doe voort
```

```
i-p = 5 / k = 2
    #(l a l a l a n d)
    #(0 0 0 1 2 3 0 0)
plaats 3 in 5 en doe voort
```

```
i-p = 6 / k = 3
    #(l a l a l a n d)
    #(0 0 0 1 2 3 4 0)
plaats 4 in 6 en doe voort
```

```
i-p = 7 / k = 4
    terugval op k = 2
i-p = 7 / k = 2
    terugval op k = 0
```

```
i-p = 7 / k = 0
    #(l a l a l a n d)
    #(0 0 0 1 2 3 4 0)
plaats 0 in 7 en start k=0
```

```
i-p = 8 / k = 0
    plaats 0 in 8, eindig
    #(l a l a l a n d)
    #(0 0 0 1 2 3 4 0)
```

```
#(-1 0 0 1 2 3 4 0)
```

Performantie van KMP

De evolutie van `i-t + i-p` in de `loop`: een maatstaaf voor de vooruitgang.

- Ofwel match: `i-t + (i-p + 1)`
- Ofwel geen match: `(i-t + i-p - σ(i-p)) + σ(i-p)`

De som `i-t + i-p` stagneert of verhoogt in elke lus. Dus wordt de lus maximaal $2n_t$ keer uitgevoerd.

Het bepalen van σ

- in de tweede tak: van `ip` en `k` naar `ip+1` en `k+1`
  dus is `ip - k` constant
- in de derde tak: `ip` blijft en `k` naar `σ(k)`
  dus is `ip - k` verhoogt
- in de derde tak: `ip` naar `ip+1` en `k` blijft
  dus is `ip - k` verhoogt

`σ(X) < X`

worst-case: alternering

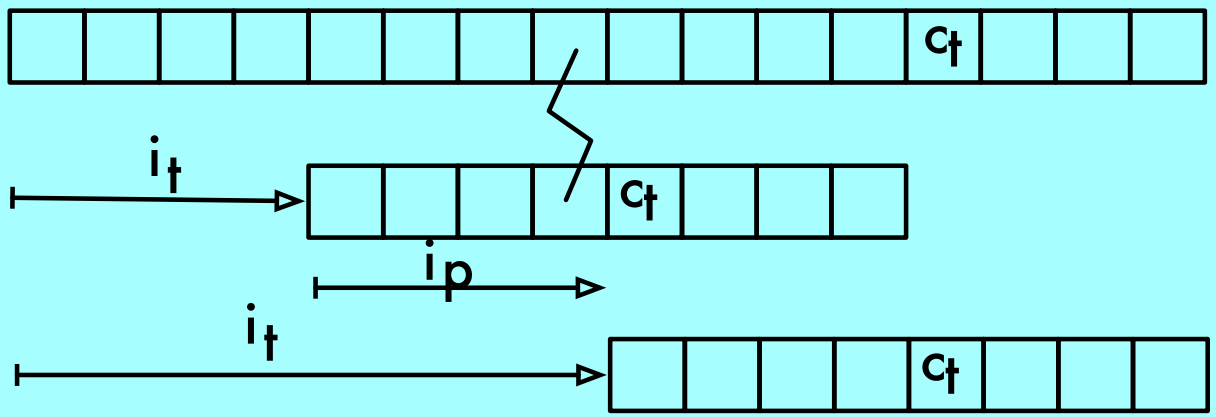
`ip - k` ≤ `ip` en `ip` ≤ n_p en dus wordt de lus maximaal $2n_p$ keer uitgevoerd

Conclusie: worst-case $f_{match}(n_t, n_p) \in O(n_t + n_p)$

QuickSearch

Uitgevonden in 1990 door D. M. Sunday. Is het snelste algoritme vandaag bekend.

Idee:



QuickSearch

```
(define (match t p)
  (define n-t (string-length t))
  (define n-p (string-length p))
  (define shift (compute-shift-function p))
  (let loop
    ((i-t 0)
     (i-p 0))
    (cond
      ((> i-p (- n-p 1))
       i-t)
      ((> i-t (- n-t n-p))
       #f)
      ((eq? (string-ref t (+ i-t i-p)) (string-ref p i-p))
       (loop i-t (+ i-p 1)))
      (else
       (let ((c-t (string-ref t (mod (+ i-t n-p) n-t))))
         (loop (+ i-t (shift c-t)) 0))))))
```

QuickSearch: de shift-tabel

```
(define (compute-shift-function p)
  (define n-p (string-length p))
  (define min-ascii (char->integer (string-ref p 0)))
  (define max-ascii min-ascii)

  (define (create-table index)
    (if (< index n-p)
      (begin
        (set! min-ascii (min min-ascii (char->integer (string-ref p index))))
        (set! max-ascii (max max-ascii (char->integer (string-ref p index))))
        (create-table (+ index 1)))
      (make-vector (- max-ascii min-ascii -1) n-p)))
  (define (fill-table index)
    (if (< index n-p)
      (let* ((ascii (char->integer (string-ref p index))))
        (vector-set! shift-table (- ascii min-ascii) (- n-p index))
        (fill-table (+ index 1)))))

  (define shift-table (create-table 0))
  (fill-table 0)
  (lambda (c)
    (let ((ascii (char->integer c)))
      (if (>= max-ascii ascii min-ascii)
        (vector-ref shift-table (- ascii min-ascii))
        n-p))))
```

Performantie

- Worst-case $f_{\texttt{match}}(n_t,n_p) \in O(n_t.n_p)$
- In de praktijk lineair en sneller dan al de rest
- Heeft sublineair potentieel:

$$O\left(N + \frac{n_t}{n_p+1}\right) \qquad N = \max(\delta_p, n_p)$$

Niet bruikbaar als je alle karakters in de tekst wil gezien hebben!

Moraal v/h Verhaal

Patroonherkenning is moeilijk en tijdrovend. Het is één van de dingen waar computers zeer slecht in zijn en mensen heel goed in zijn. Het is één van de centrale vraagstukken van de artificiële intelligentie, zowel wat betreft tekst, beeld als geluid.

Strings in een geheugen stoppen is niet moeilijk. Strings er correct weer uithalen is moeilijk en tijdrovend. Strings dienen bijgevolg enkel om met gebruikers te communiceren! Het is de armste data structuur die er bestaat.

Hoofdstuk 2

2.1 Strings in Scheme

2.2 Het Pattern Matching Probleem

2.3 Het brutekracht Algoritme

2.4 Het Knuth-Morris-Pratt Algoritme

2.5 Het QuickSearch Algoritme