Hoofdstuk 2

Strings en Patroonherkenning

Inhoud

1. Characters, Strings, en Patroonherkenning
2. Brutekracht algoritme
3. Quicksearch algoritme
4. Knuth-Morris-Pratt algoritme

2.1 Characters, Strings, en Patroonherkenning

Characters: ASCII-waarden 0-127

American Standard Code for Information Interchange

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	080	P	112	p
017	◄	DC1	049	1	081	Q	113	q
018	↕	DC2	050	2	082	R	114	r
019	‼	DC3	051	3	083	S	115	s
020	π	DC4	052	4	084	T	116	t
021	§	NAK	053	5	085	U	117	u
022	▬	SYN	054	6	086	V	118	v
023	↨	ETB	055	7	087	W	119	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	⌂


```
> (char->integer #\a)
97
> (char->integer #\newline)
10
> (integer->char 99)
#\c
```

Characters: 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	β
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	
135	ç	167	º	199	╟	231	
136	ê	168	¿	200	╚	232	
137	ë	169	⌐	201	╔	233	
138	è	170	¬	202	╩	234	
139	ï	171	½	203	╦	235	
140	î	172	¼	204	╠	236	
141	ì	173	¡	205	═	237	
142	Ä	174	«	206	╬	238	
143	Å	175	»	207	╧	239	
144	É	176	░	208	╨	240	
145	æ	177	▒	209	╤	241	
146	Æ	178	▓	210	╥	242	
147	ô	179	│	211	╙	243	
148	ö	180	┤	212	╘	244	
149	ò	181	╡	213	╒	245	⌡
150	û	182	╢	214	╓	246	÷
151	ù	183	╖	215	╫	247	≈
152	ÿ	184	╕	216	╪	248	°
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

```
> (char<? #\a #\B)
#f
> char-ci<?
✗ ✗ char-ci<?: undefined;
 cannot reference an identifier before its definition
> (import (scheme char))
> (char-ci<? #\a #\B)
#t
> (char-upper-case? #\Z)
#t
```

Vele nuttige procedures zitten in de library `(scheme char)`

Strings

Een string is een eindige sequentie van karakters.

```
> "Madam I'm Adam"
"Madam I'm Adam"
> (make-string 5 #\a)
"aaaaa"
> (string #\a #\S #\t #\r #\i #\n #\g)
"aString"
> > (define ad "algoritmen en datastructuren")
> (string-ref ad 10)
#\space
> (string-set! ad 10 #\*)
✗ ✗ string-set!: contract violation
  expected: mutable-string?
  given: "algoritmen en datastructuren"
```

Immutable

Constructors

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

```
string-length
string-ref
```

Beide O(1)

O(n)

Conversies

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

Optellen en Aftrekken

```
string-append
substring
```

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

Vergelijken

O(min(n,m))

```
(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)
```

Lexicografische orde

```
> (string=? "map" "aap")
#f
> (import (scheme char))
> (string-ci=? "man" "MaN")
#t
```

Het patroonherkenningsprobleem

Gegeven een tekst (of “hooiberg”). Gegeven een patroon (of “naald”). Wat is de index van de naald in de hooiberg?

Toepassingen in tekstverwerkers en bioinformatica

```
match
   (string string → number ∪ {#f})
```

```
> (define t "Deze tekst is een voorbeeld van zo’n hooiberg")
> t
"Deze tekst is een voorbeeld van zo’n hooiberg"
> (define p "voorbeeld")
> p
"voorbeeld"
> (match t p)
18
> (match t "123")
#f
```

Dit noemt men de “offset” of de “shift”

Notaties

*De tekst noteren we met t. Het patroon met p. De lengte van het patroon **n-p** of n_p. De lengte van de tekst **n-t** of n_t*

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 deelstring van i tot en met j

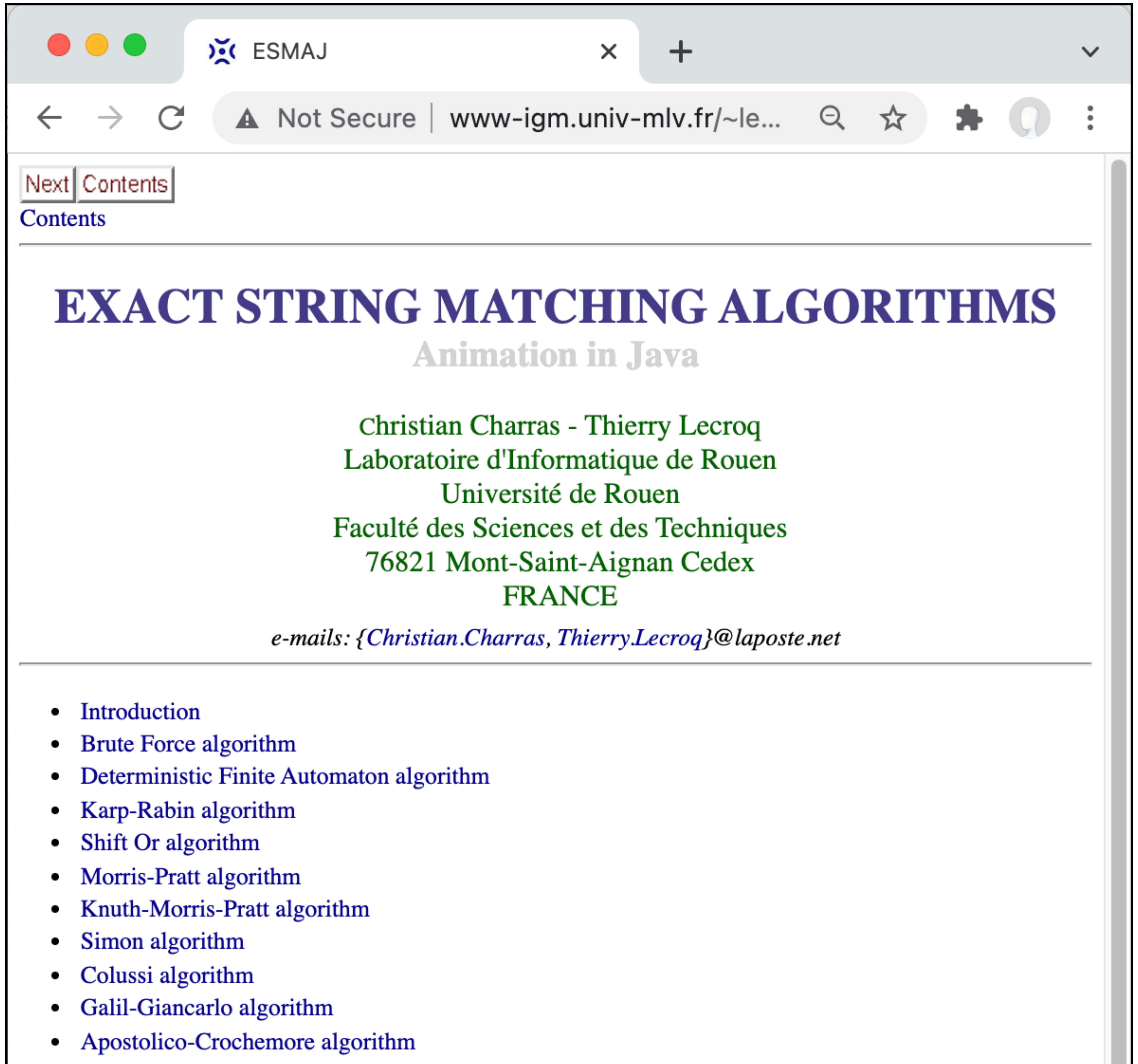
Een string s die uit twee delen u en v bestaat noteren we als s = u.v

u heet dan een prefix van s, en v een suffix

We bestuderen 3 algoritmen

Er bestaat geen uniek best algoritme

1. Het brutekracht algoritme
2. Het QuickSearch algoritme
3. Het Knuth-Morris-Pratt algoritme

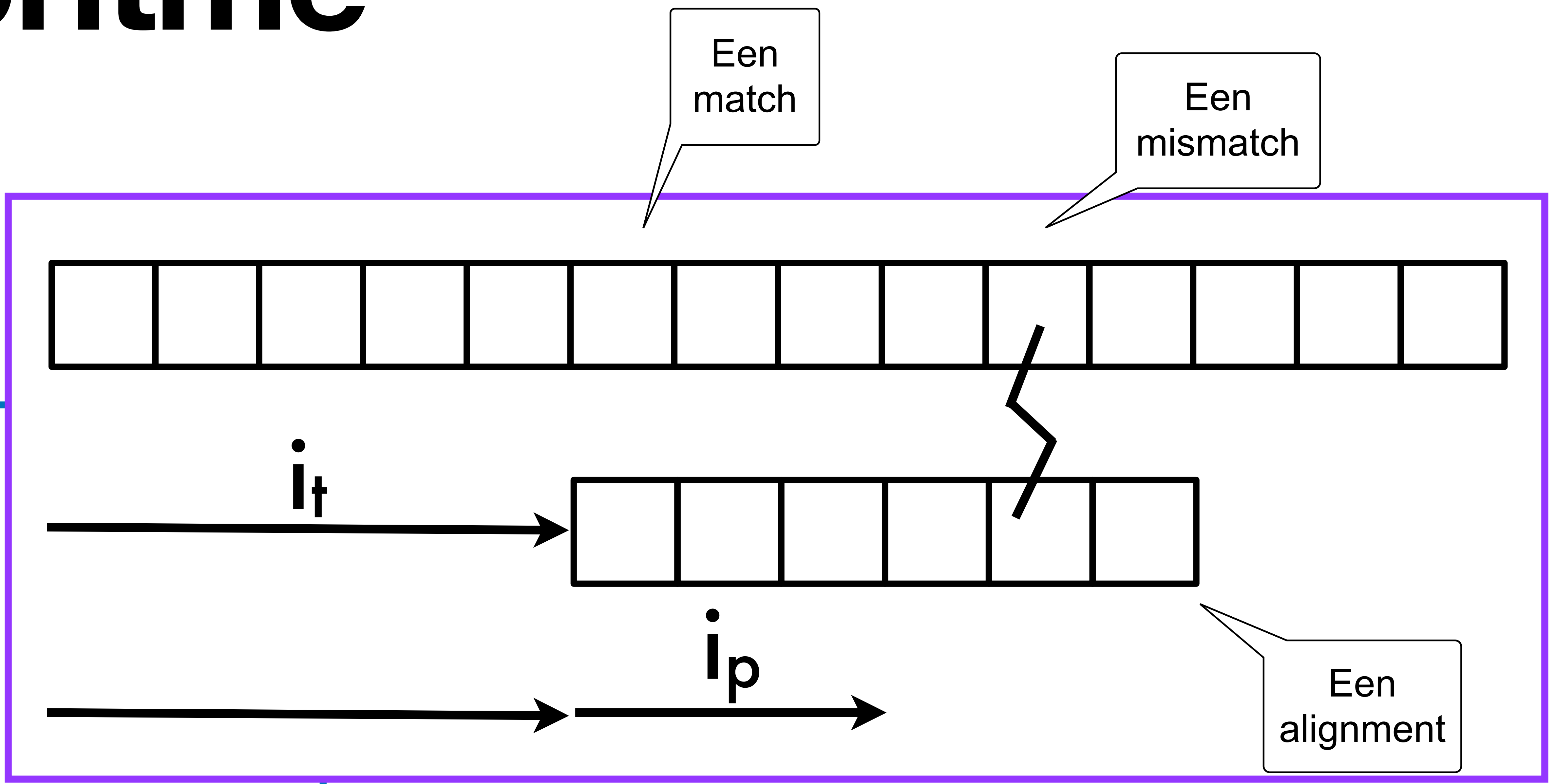


http://www-igm.univ-mlv.fr/~lecroq/string/index.html

2.2 Brutekracht Algoritme

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)))))
```



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

Performantie van het Brutekracht algoritme

n_t noch n_p zijn constant!

De loop wordt uitgevoerd voor i_t gaande van 0 tot $n_t - n_p$.
Voor elke i_t gaat i_p in het slechtste geval van 0 tot $n_p - 1$.

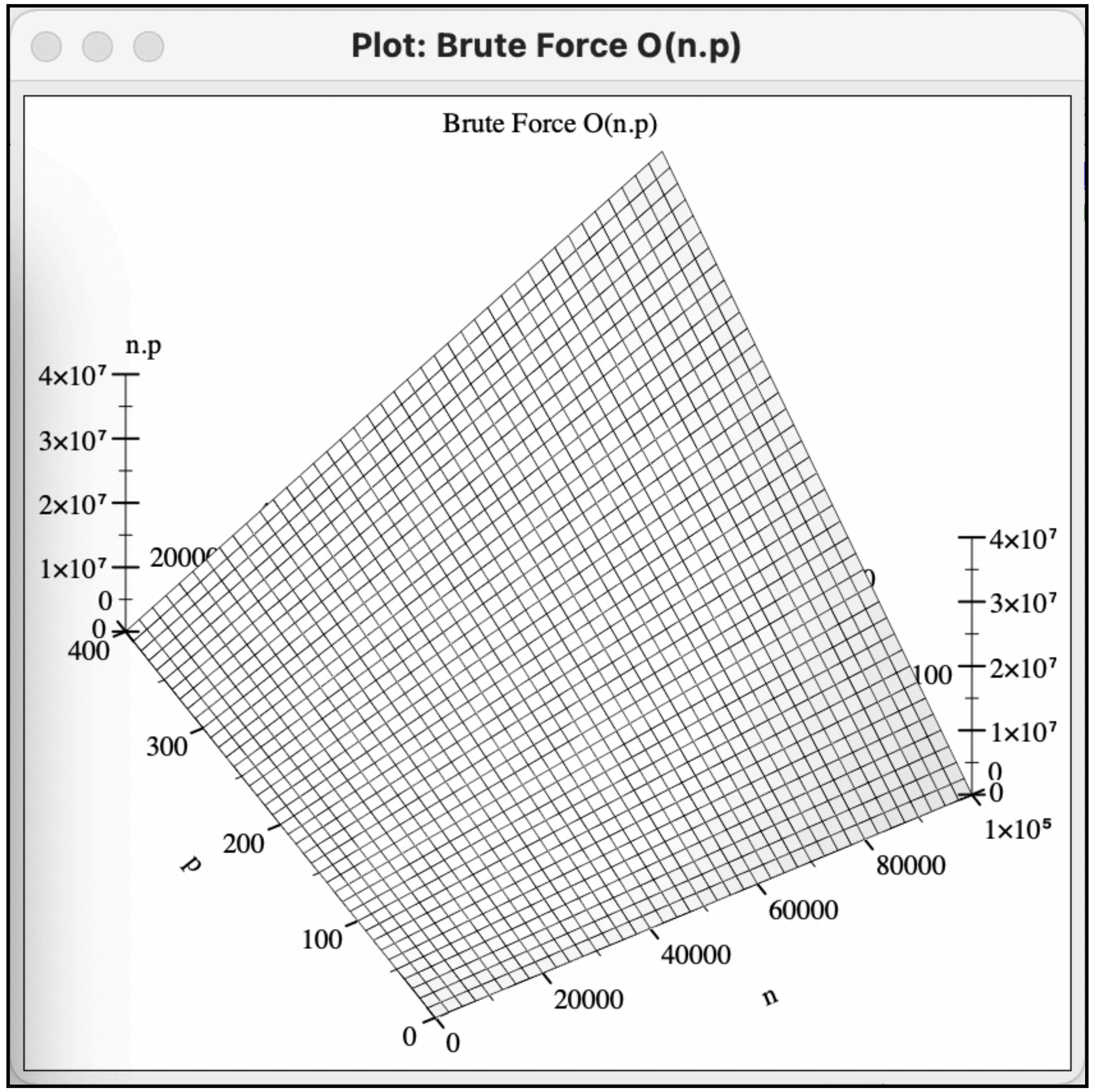
Dus:

$b_{match(nt,np)} \in O(1)$

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

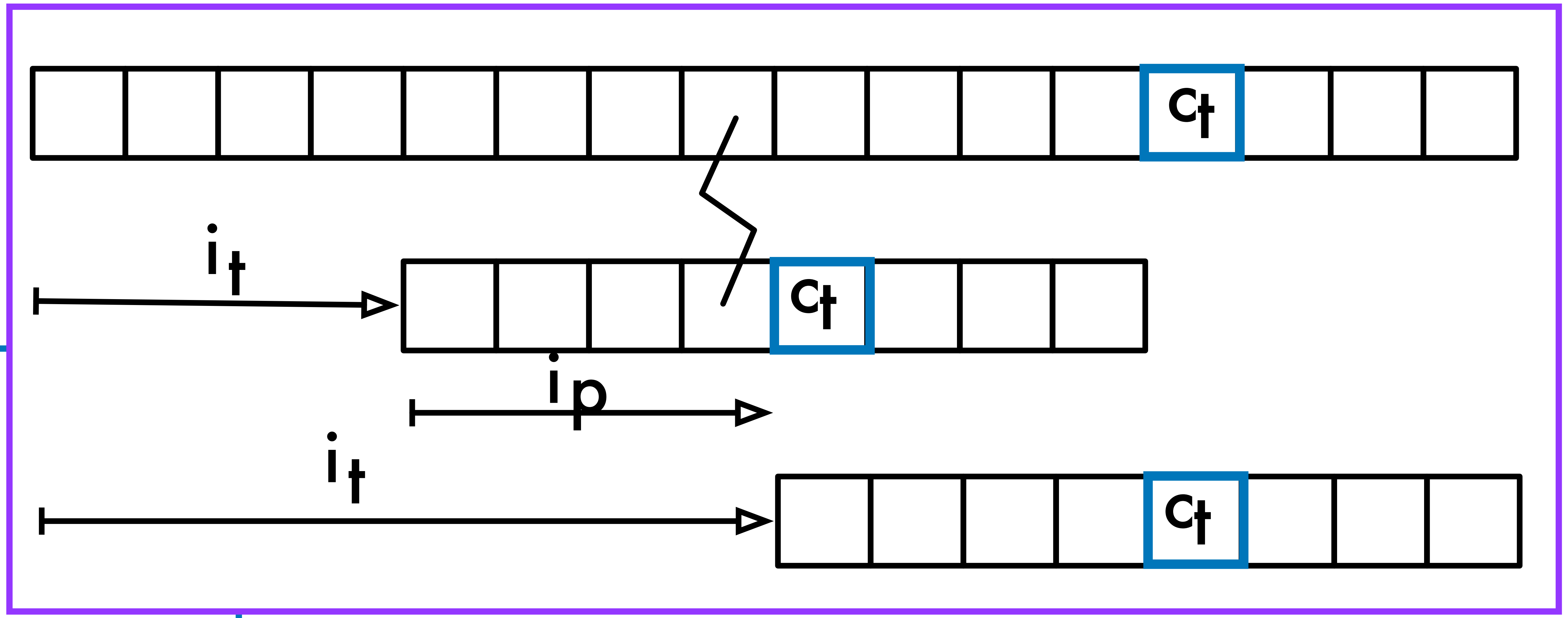
En Dus:

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



2.3 Quicksearch Algoritme

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 (modulo (+ i-t n-p) n-t))))
         (loop (+ i-t (shift c-t)) 0))))))
```

Waarom? (technisch detail)

Uitgevonden in 1990 door D. M. Sunday. Is het snelste algoritme vandaag bekend in de praktijk. Worst-case zelfde als brutekracht.

Goede didactische opstap naar KMP

Voorbeeld

```
My stepsister prefers stepping.
stepping
```

M≠s

c_t=i

c_t=e

```
My stepsister prefers stepping.
   stepping
```

s=s; t=t; e=e; p=p; s≠p

c_t=f

```
My stepsister prefers stepping.
         stepping
```

s=s; t=t; e=e; r≠p

c_t=p

```
My stepsister prefers stepping.
                  stepping
```

e≠s

```
My stepsister prefers stepping.
                      stepping
```

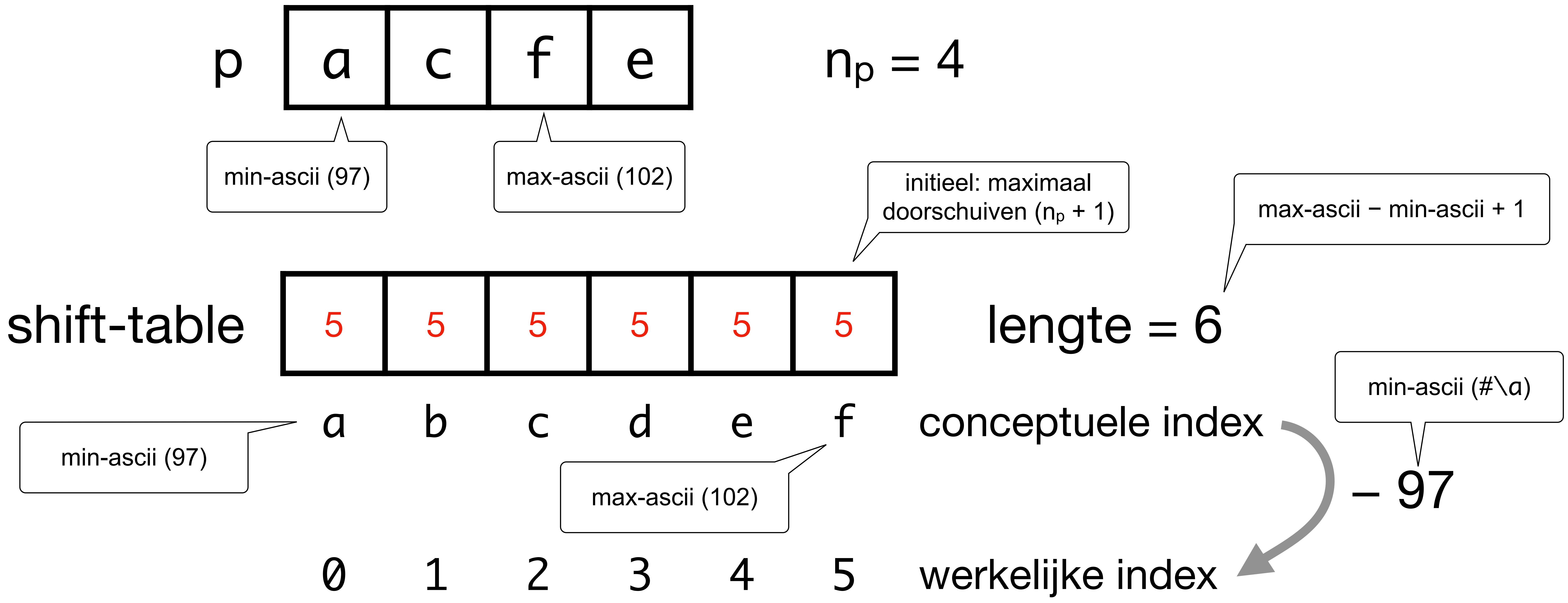
s=s; t=t; e=e; p=p; ...

p komt 2 keer voor. Er zijn dus 2 opties om c_t mee te alignen.

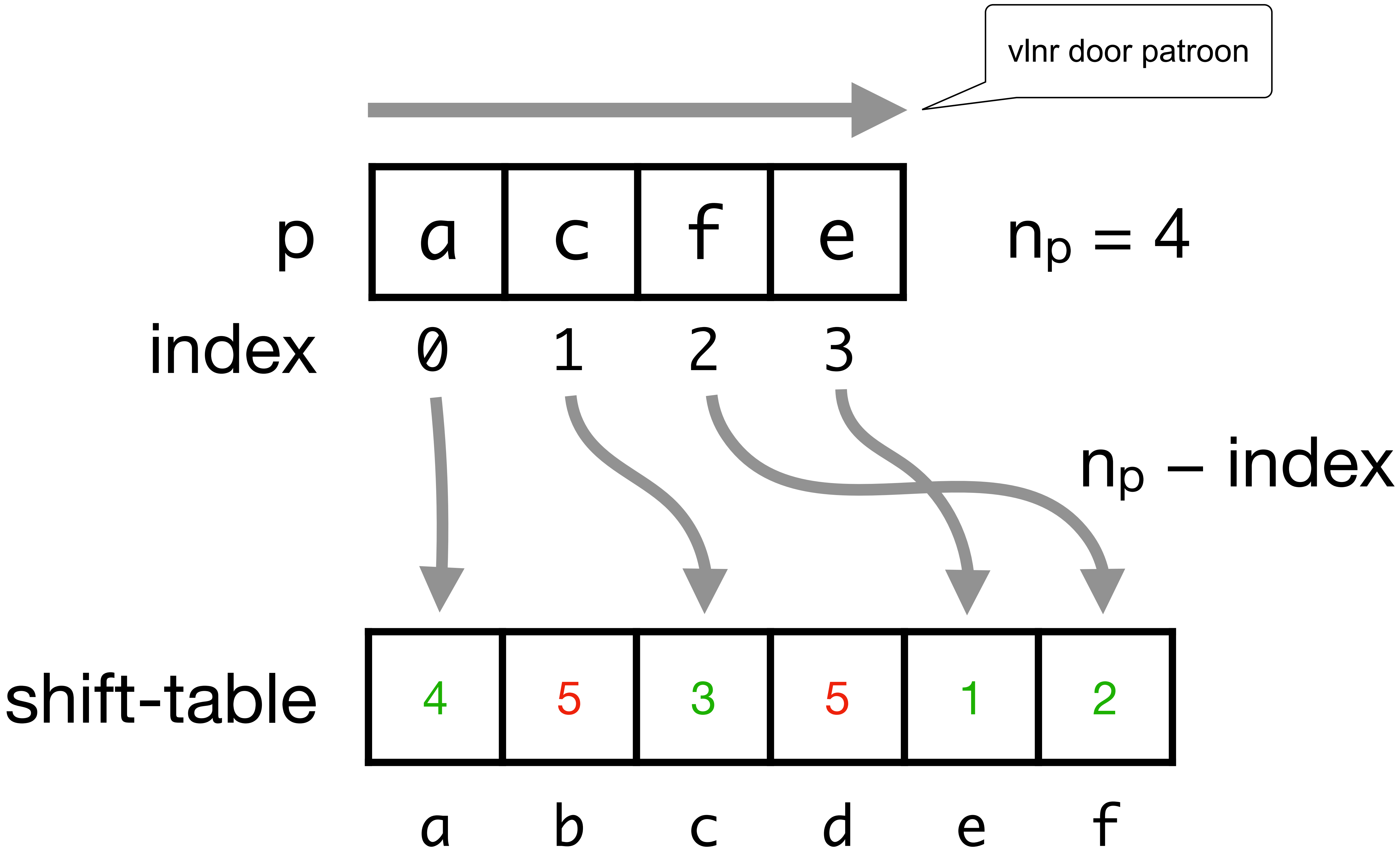
We zouden een oplossing kunnen overshooten. Neem daarom steeds het meest rechtse voorkomen van c_t in patroon.

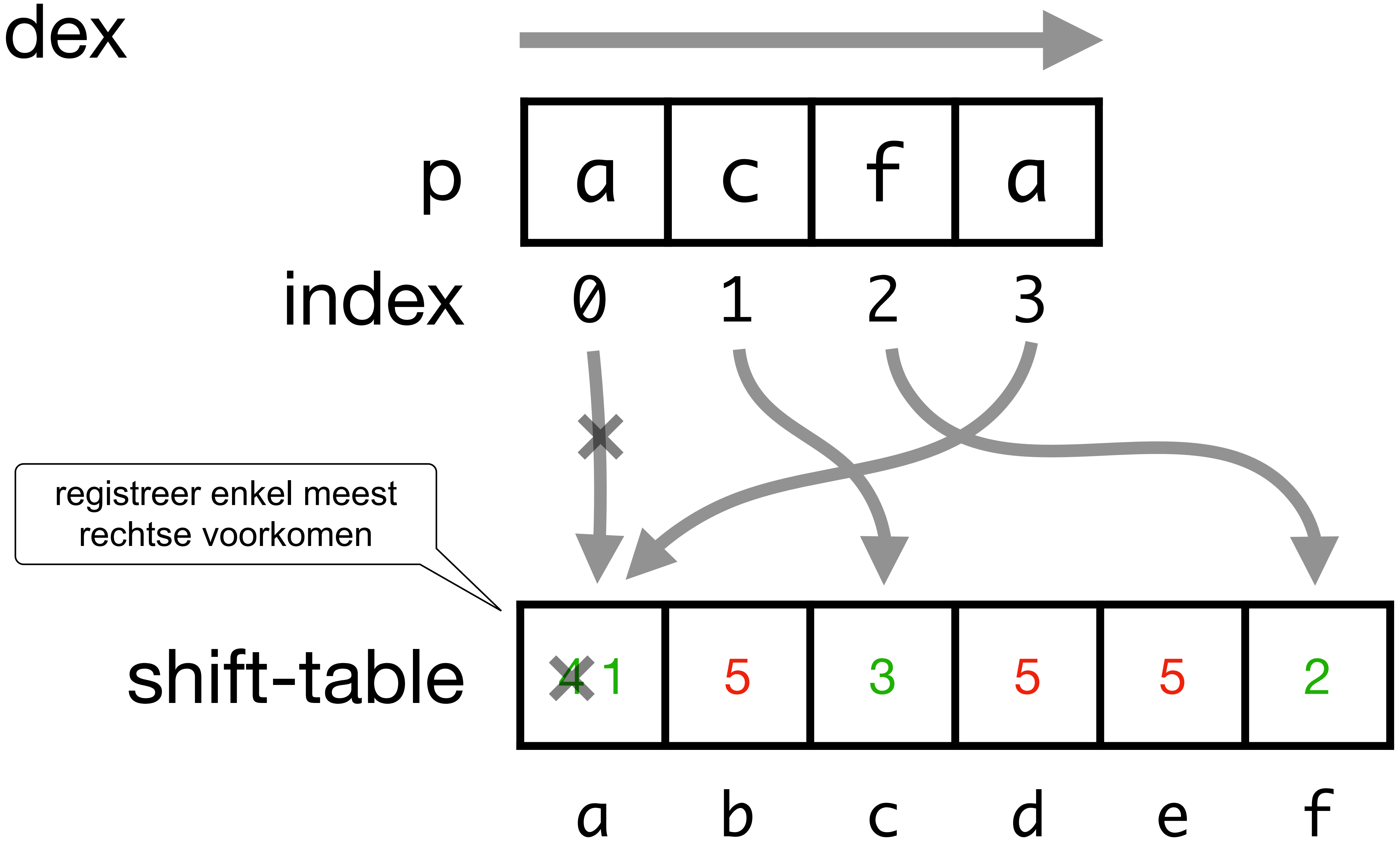
```
My stepsister prefers stepping.
                      stepping
```

Opstellen van de shift-tabel (1/2): `create-table`



Opstellen van de shift-tabel (2/2): `fill-table`





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 1))))
  (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 1)))))
```

Performantie

Zie WPO

Worst-case:

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

- *In de praktijk lineair en sneller dan al de rest (op Engelse tekst).*
- *Vertoont soms sublineair gedrag:* $O\left(N + \frac{n_t}{n_p + 1}\right)$

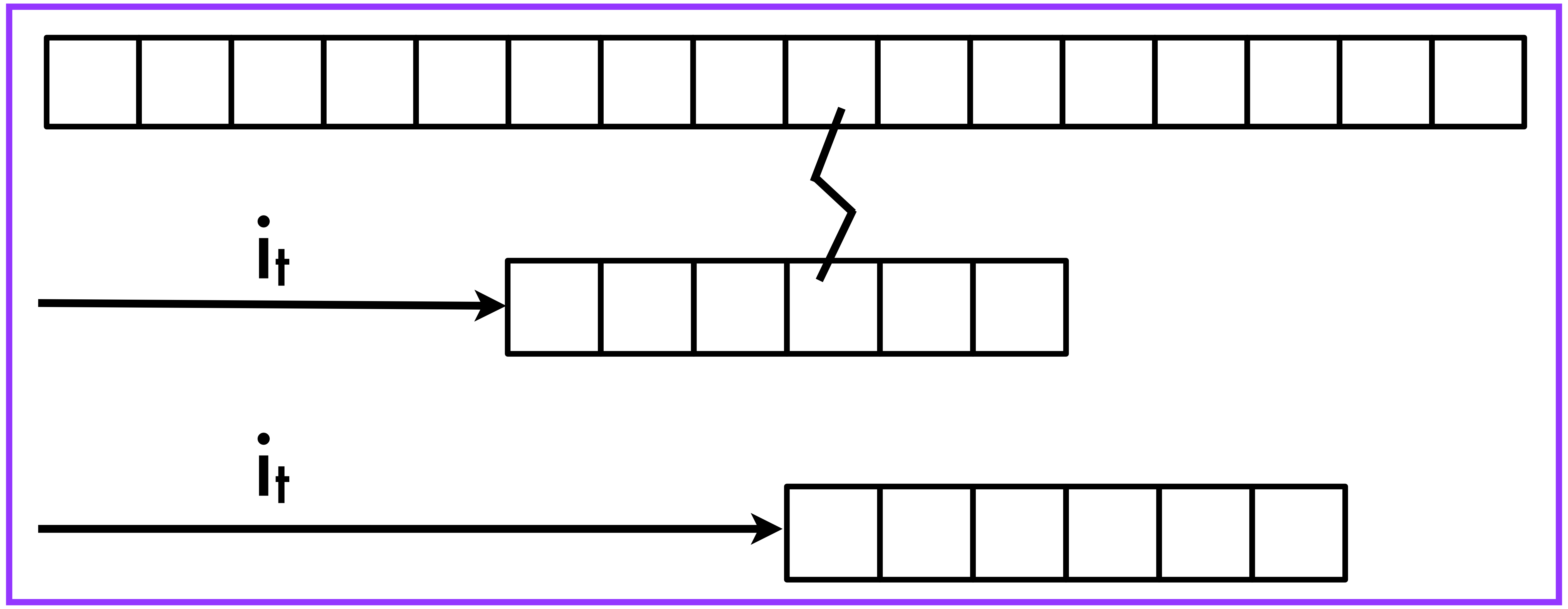
$N = \max(\delta_p, n_p)$

Het algoritme is dus niet bruikbaar als je alle characters in de tekst gezien wil hebben

De "spanwijdte" van het "alfabet" van het patroon (c.f. opvullen vector)

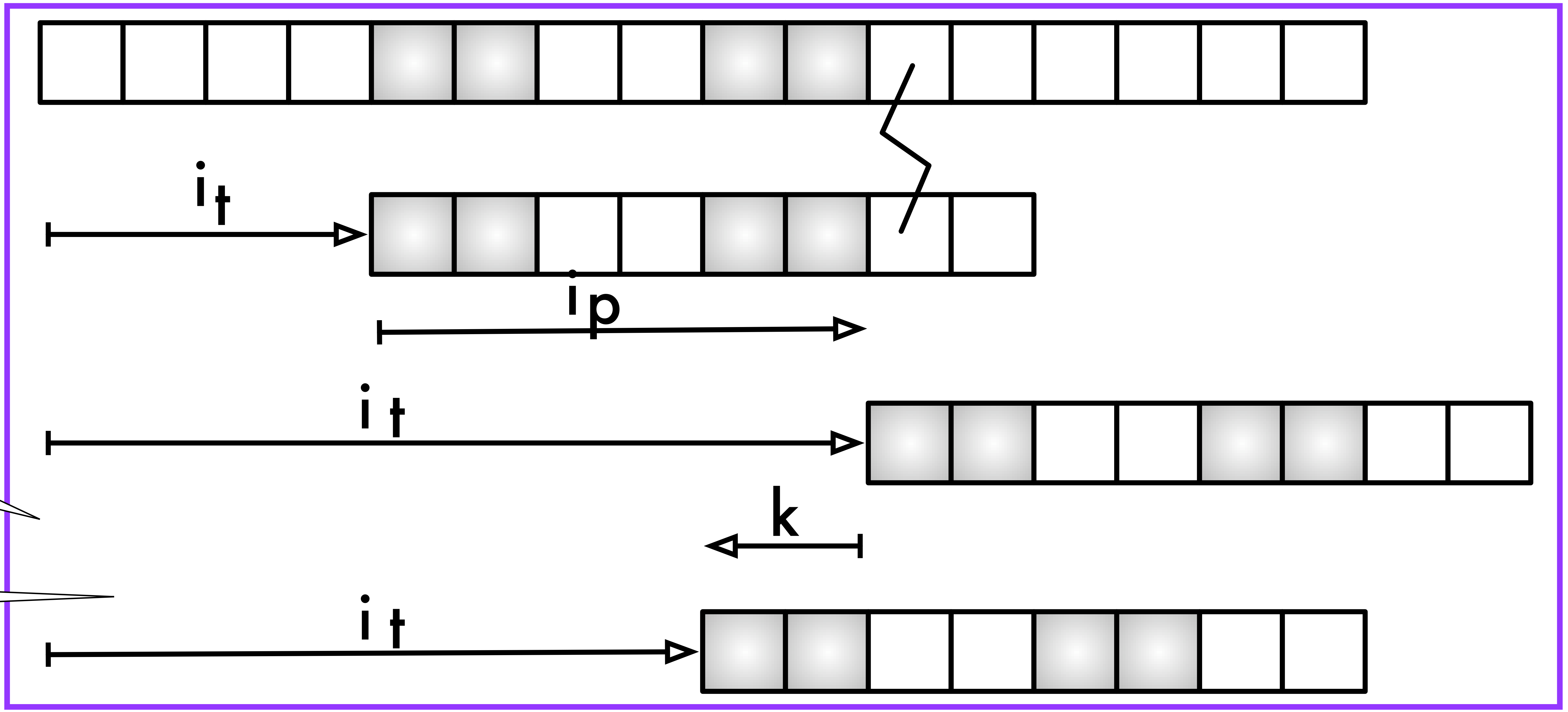
2.4 Knuth-Morris-Pratt Algoritme

Het Knuth-Morris-Pratt Algoritme



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

Naïef idee: begin met het patroon op de plaats waar het misliep

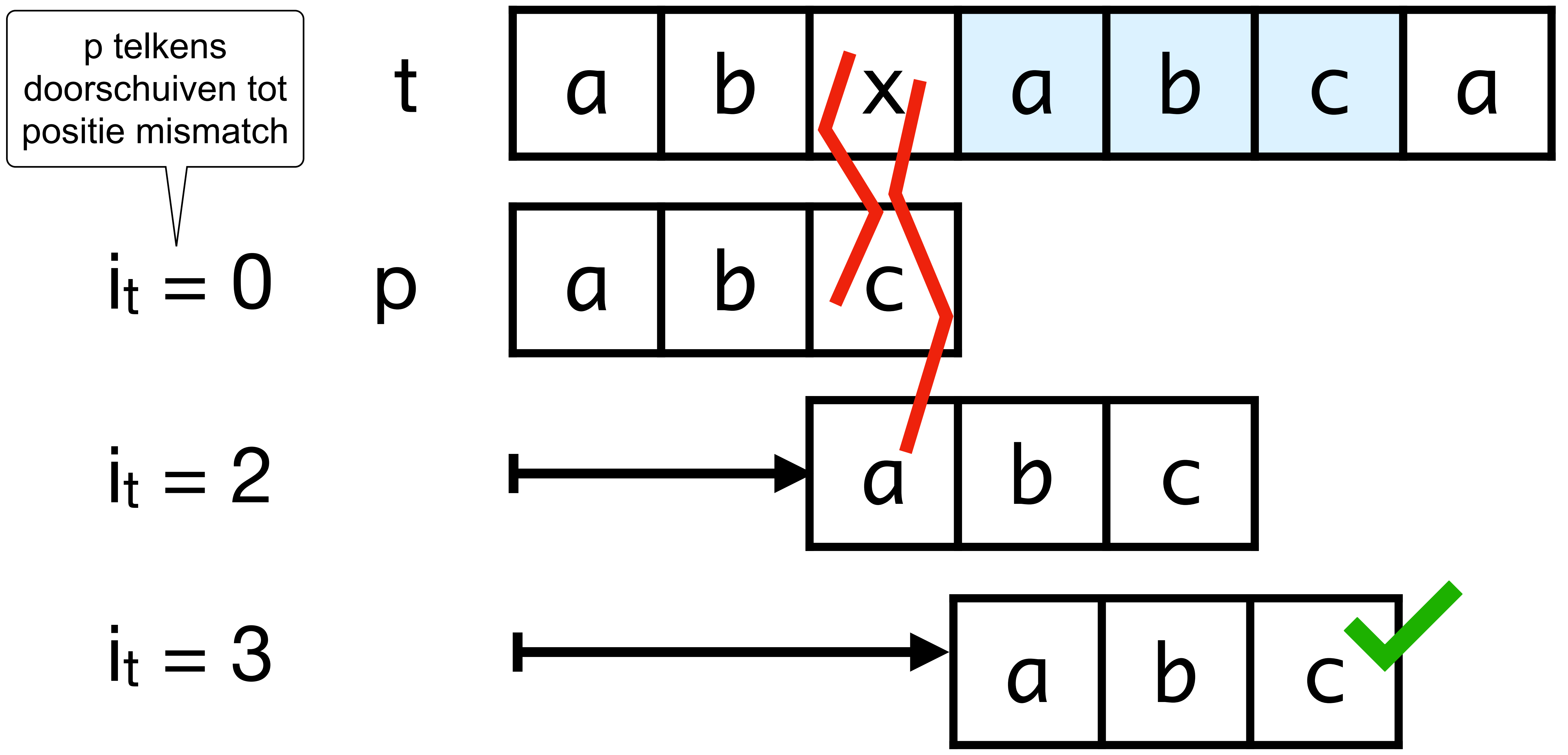


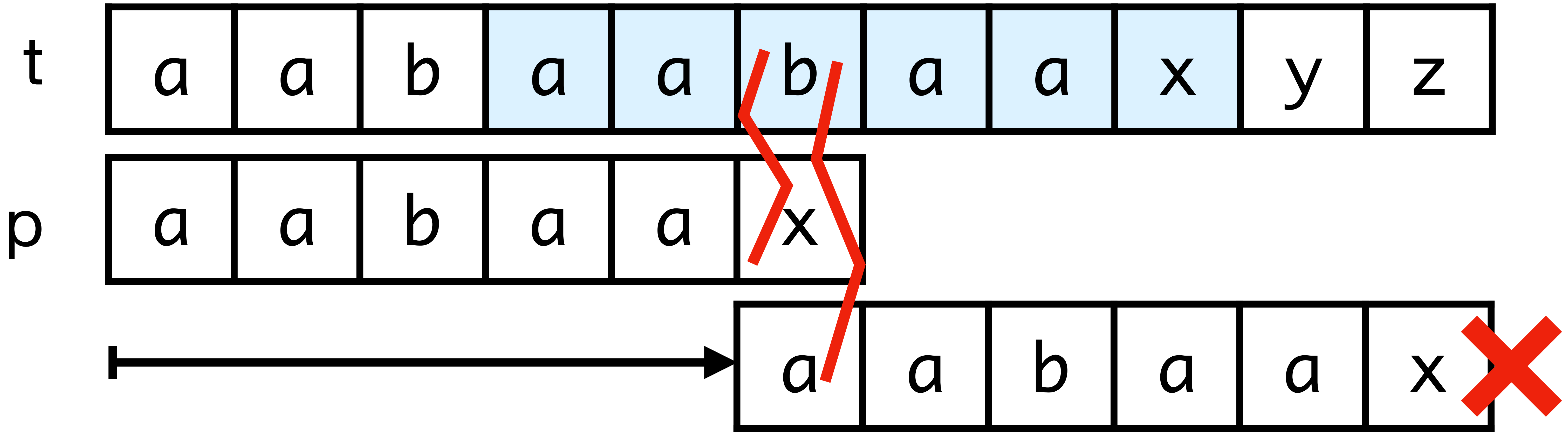
Maar dat dreigt oplossingen te overshooten indien het patroon herhalingen bevat

We keren zoveel terug "als nodig is". Hoeveel is dat? k

KMP: idee (1/2)

Kinderversie



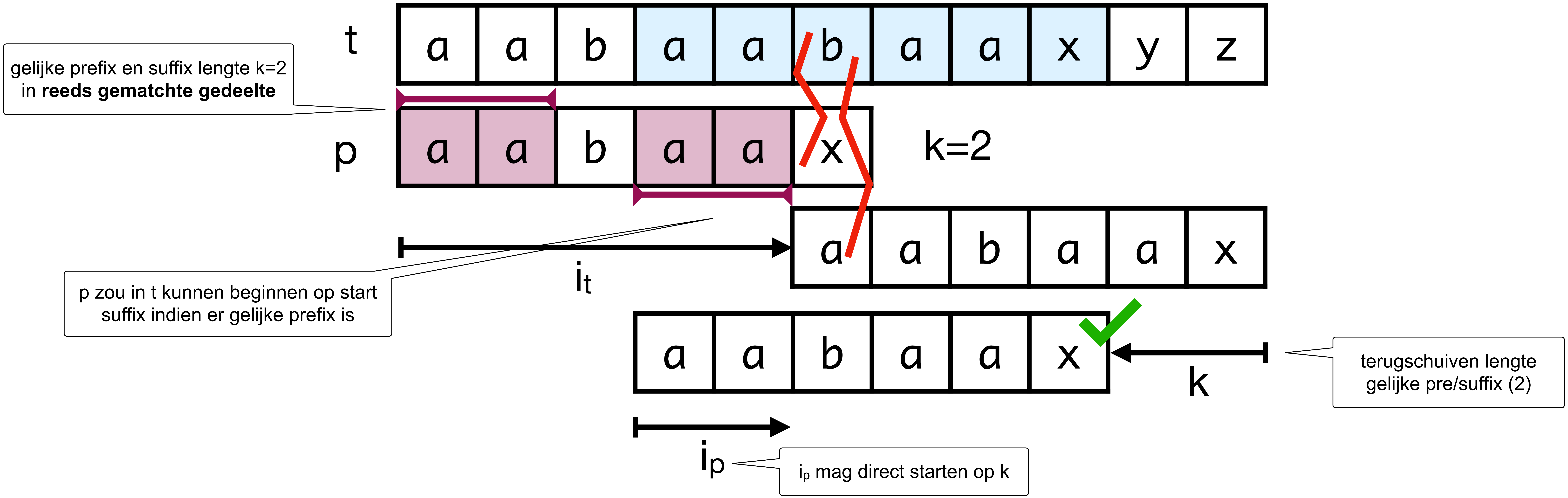


De "kinderversie" kan oplossingen overshooten

wanneer?

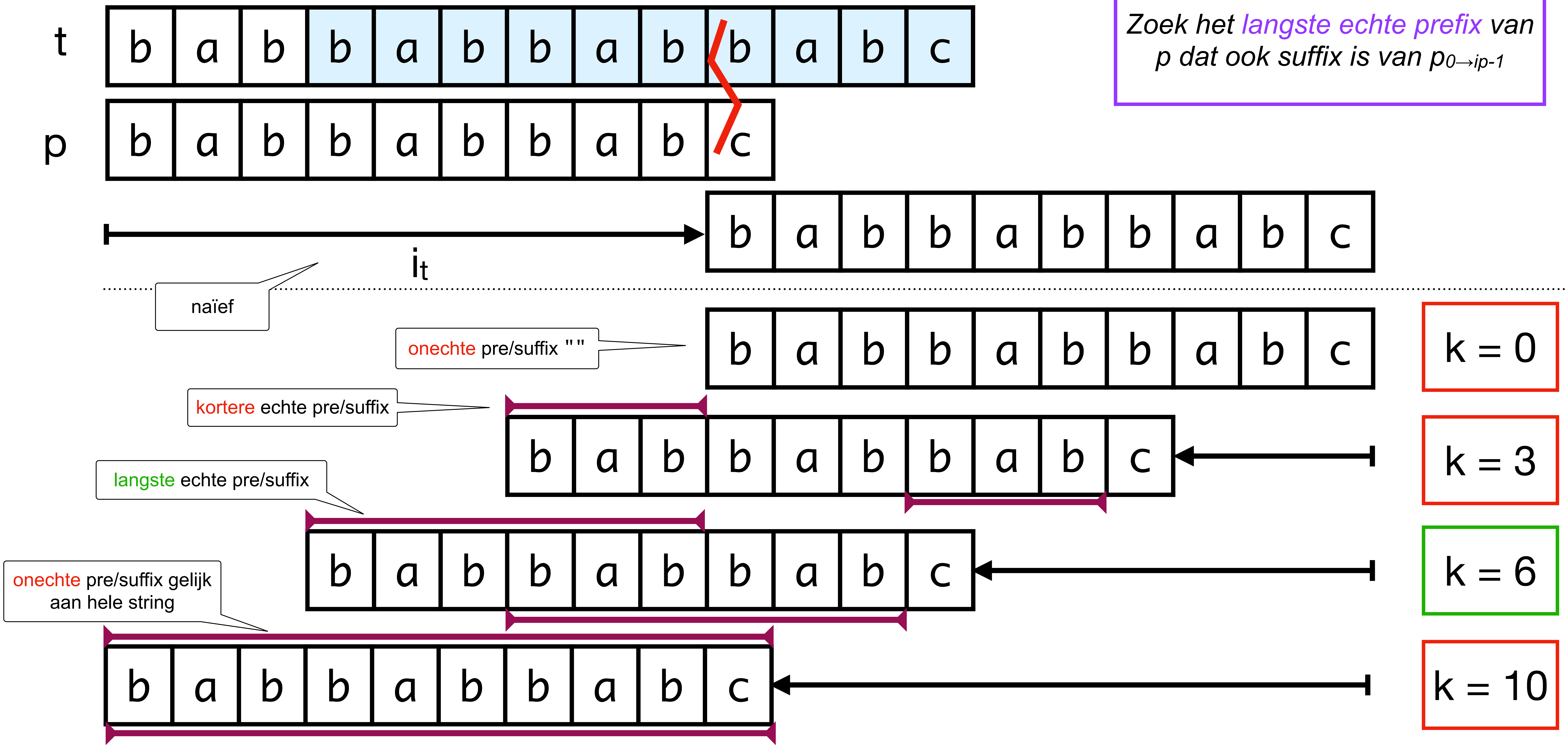
KMP: idee (2/2)

Kinderversie + terugkeren



KMP: hoeveel terugkeren?

Zoek het langste echte prefix van p dat ook suffix is van $p_{0 \to ip-1}$



Het KMP Algoritme

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

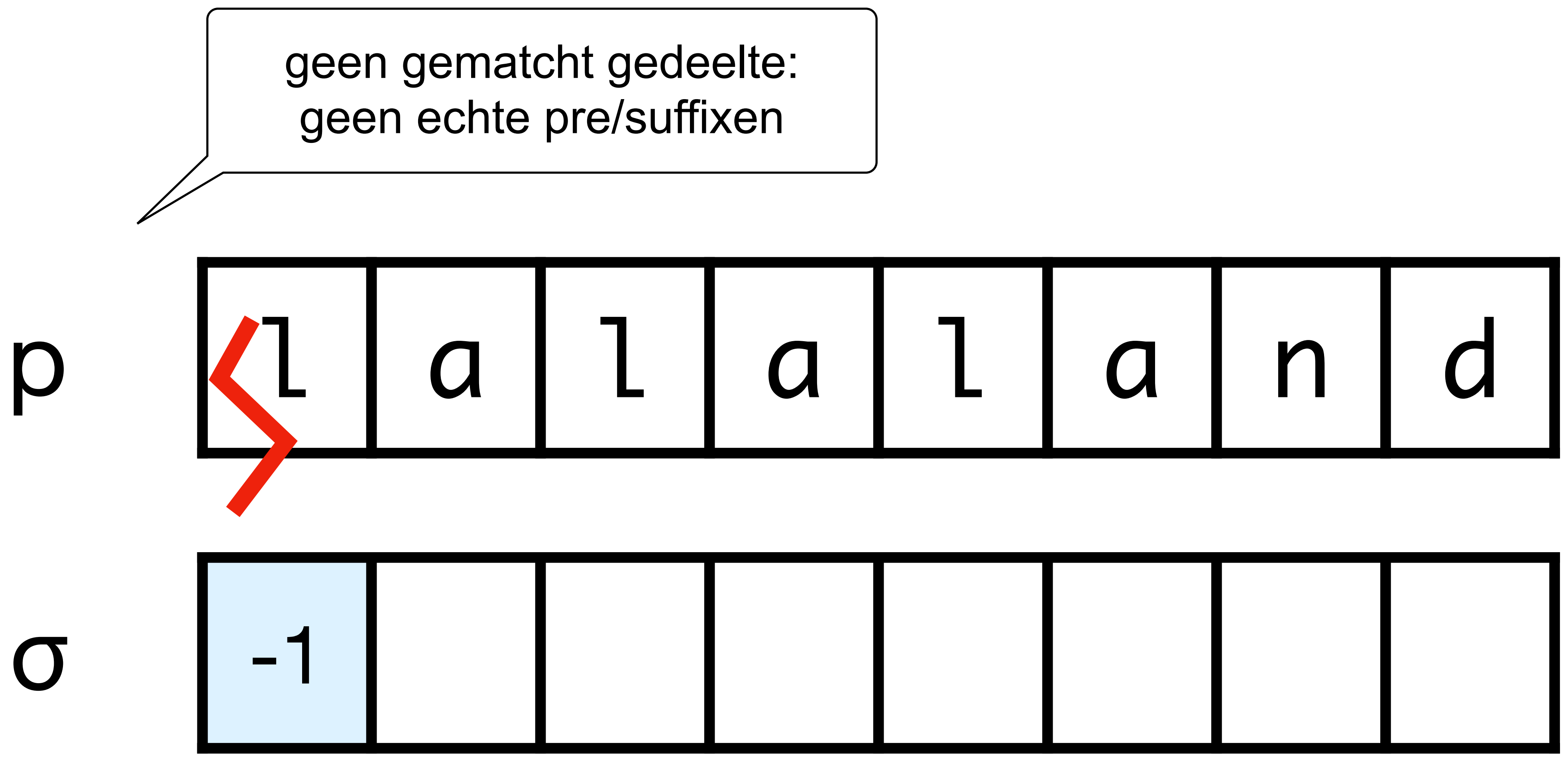
σ heet de failure function

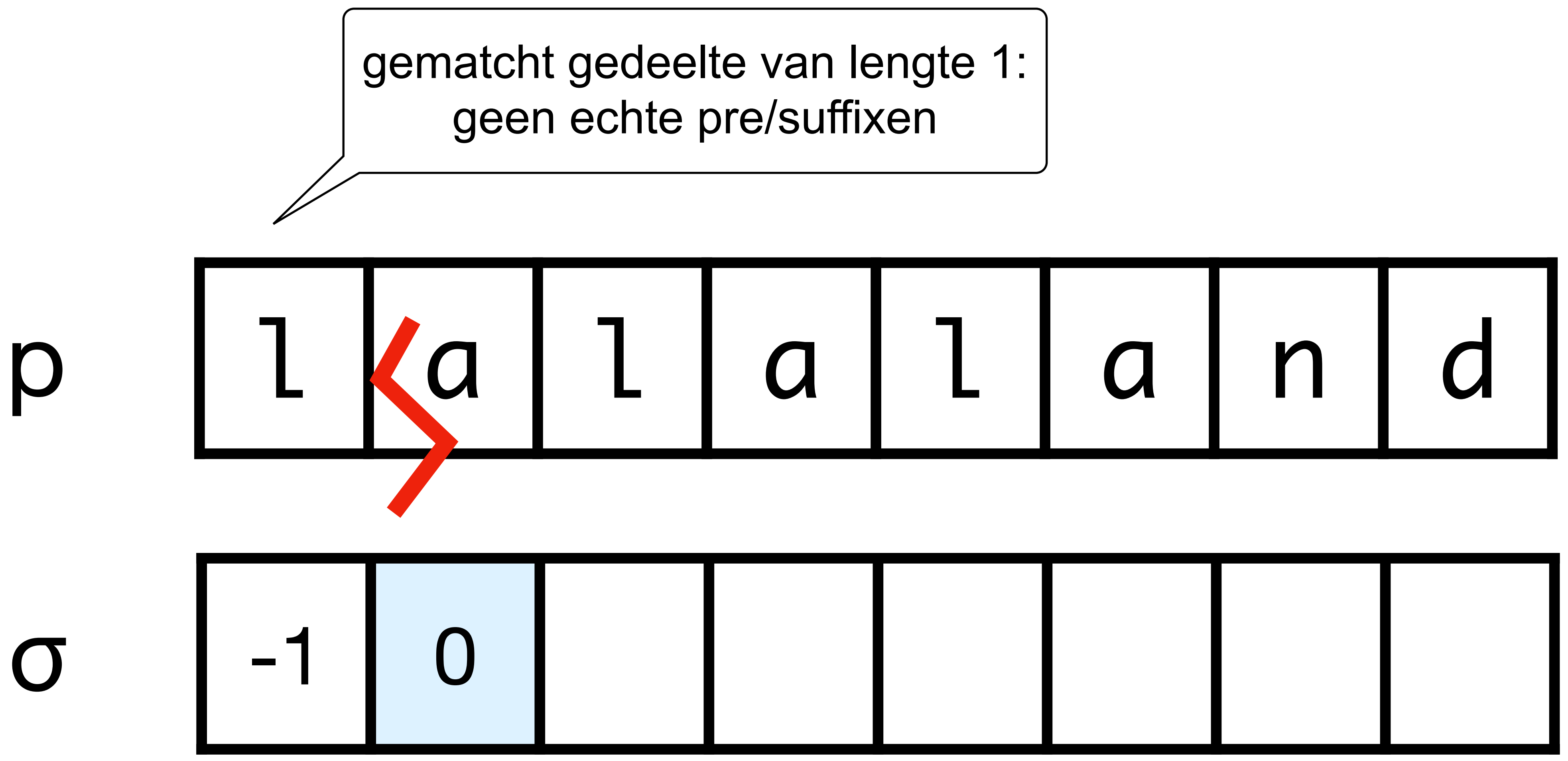
Bij een mismatch verschuif je het patroon ter waarde van $i_p - \sigma(i_p)$, en nieuwe i_p start op $\sigma(i_p)$ (of 0 indien mismatch op $i_p = 0$)

omdat altijd σ(0) = -1
(zie volgende slide)

```
(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)))))))
```

Opstellen van de σ-tabel: op het zicht (1/2)





Er zijn geen echte pre/suffixen indien mismatch op $i_p = 0$ of $i_p = 1$.
We moeten in beide gevallen 1 opschuiven, d.w.z. $i_p - \sigma(i_p) = 1$.
Daarom nemen we altijd $\sigma(0) = -1$ en $\sigma(1) = 0$.

Opstellen van de σ-tabel: op het zicht (2/2)

p	l	a	l	a	l	a	n	d
σ	-1	0	0					

p	l	a	l	a	l	a	n	d
σ	-1	0	0	1				

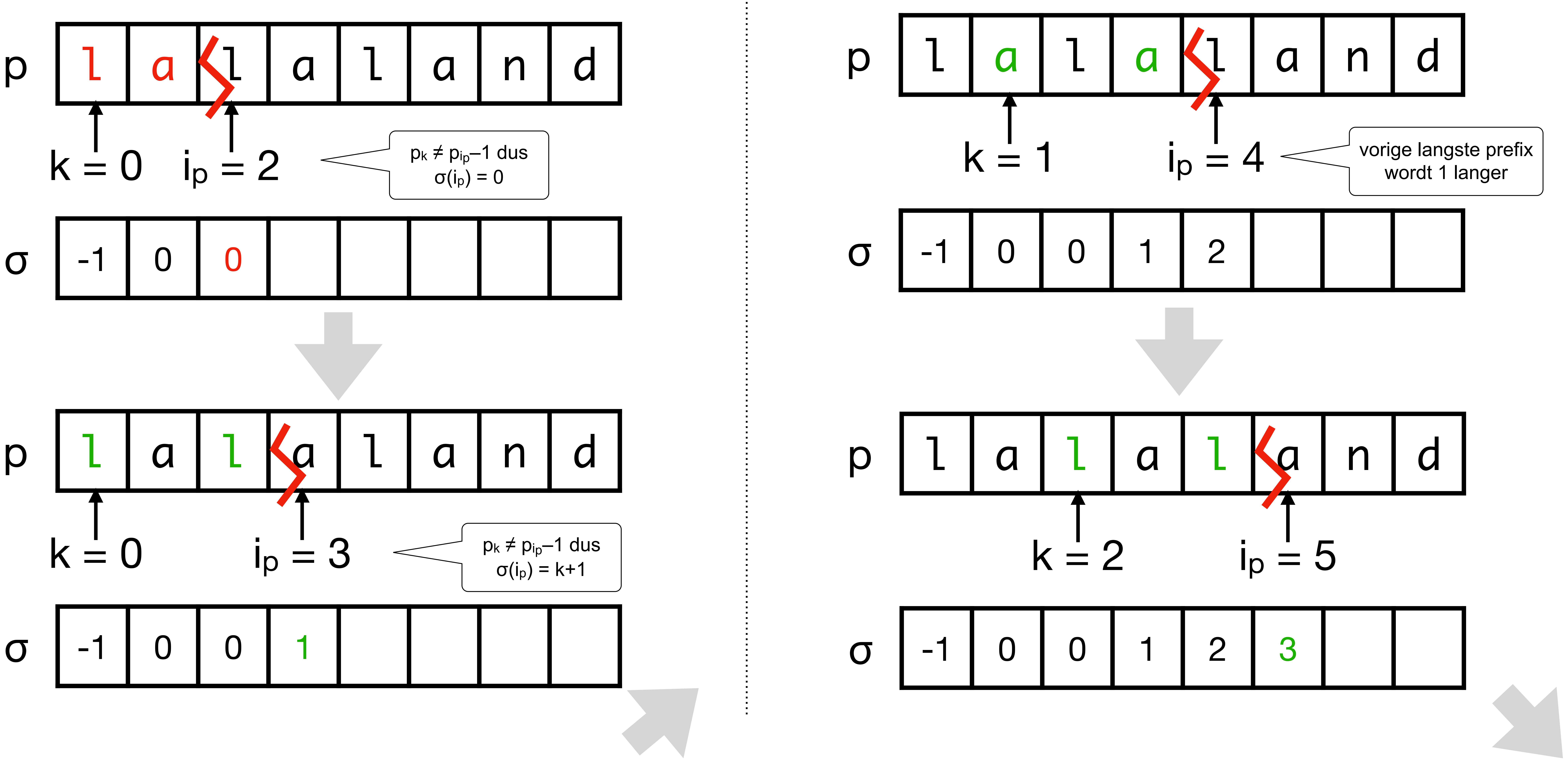
p	l	a	l	a	l	a	n	d
σ	-1	0	0	1	2			

p	l	a	l	a	l	a	n	d
σ	-1	0	0	1	2	3		

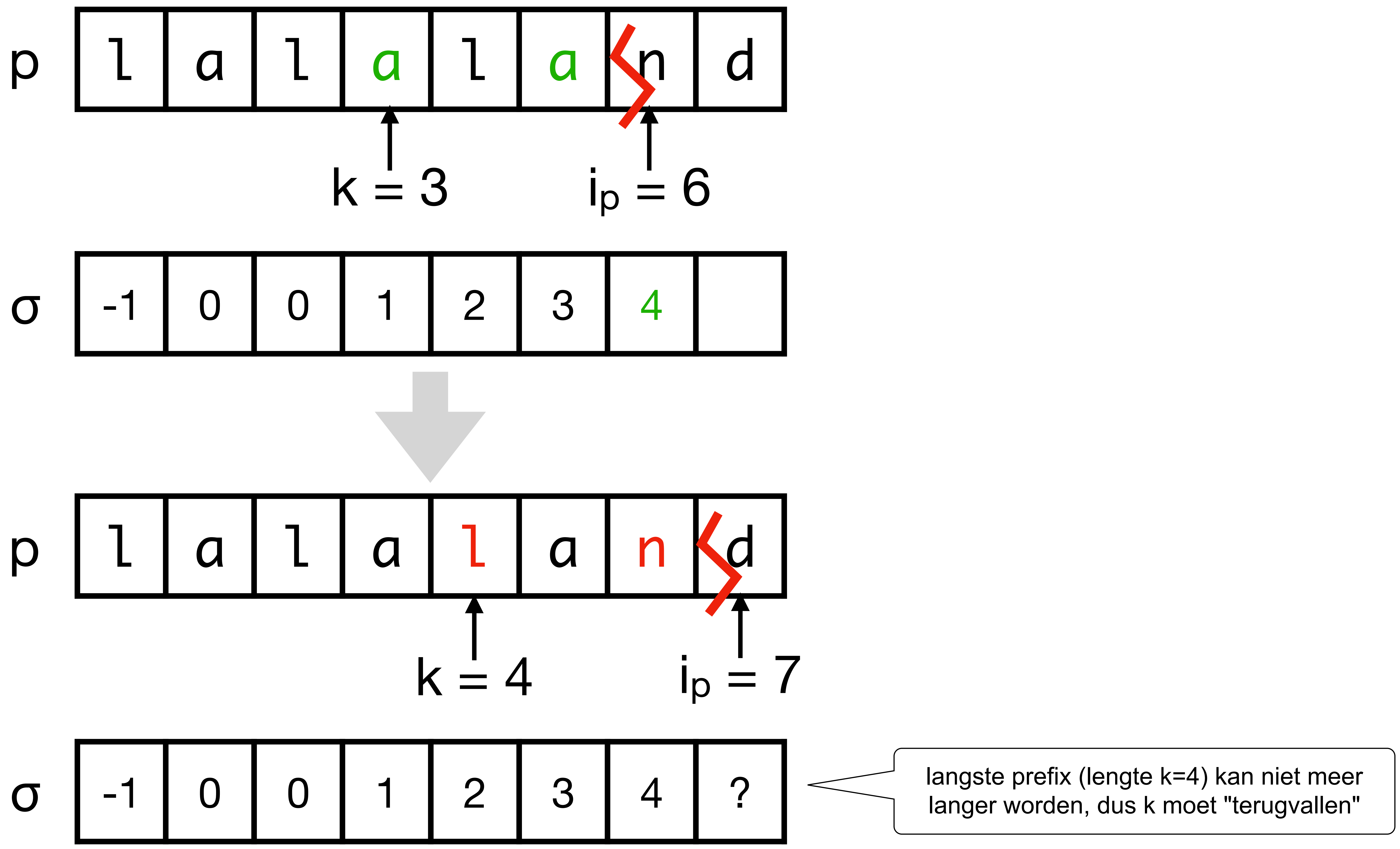
p	l	a	l	a	l	a	n	d
σ	-1	0	0	1	2	3	4	

p	l	a	l	a	l	a	n	d
σ	-1	0	0	1	2	3	4	0

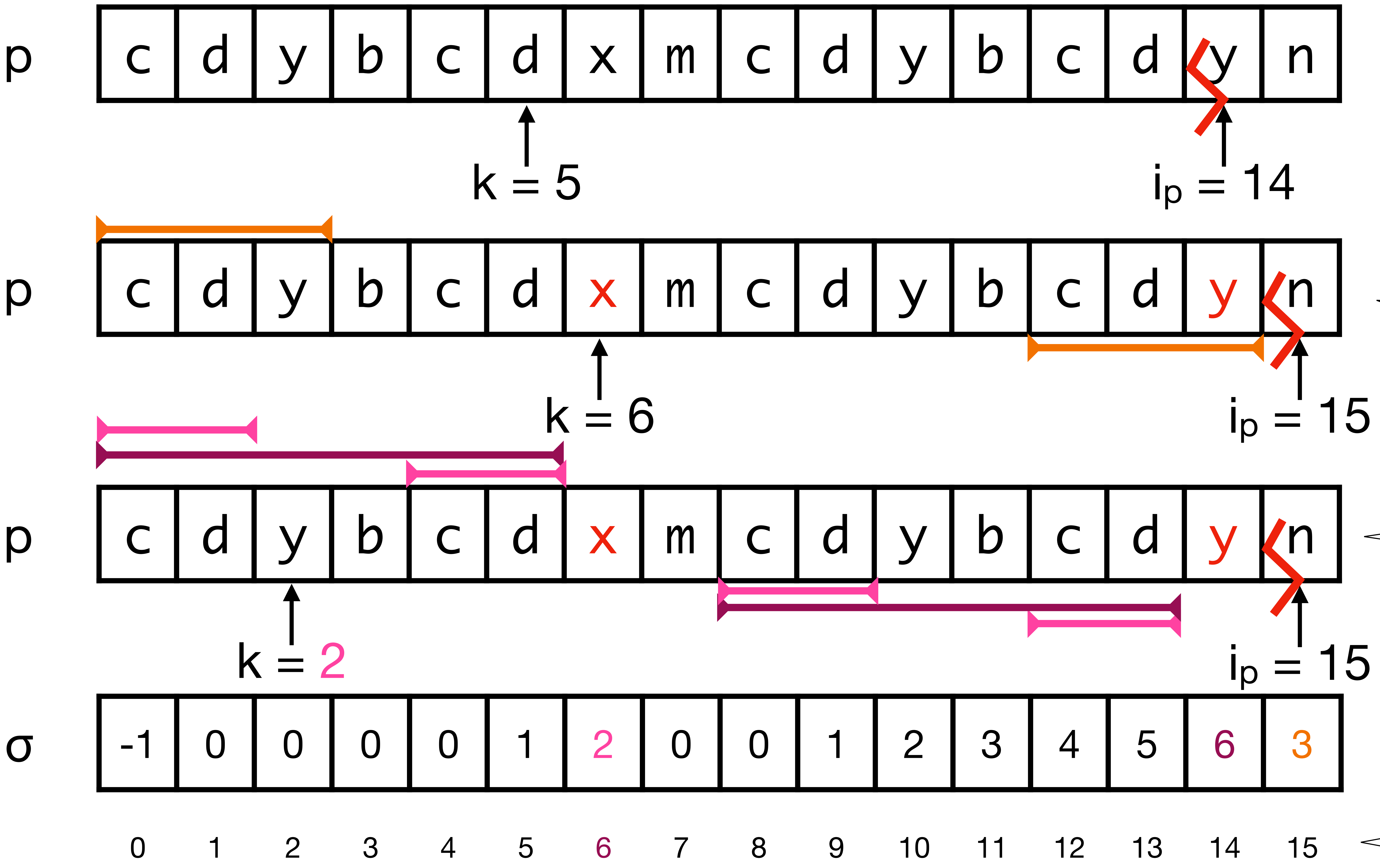
Opstellen van de σ-tabel: algoritme (1/2)



Opstellen van de σ-tabel: algoritme (2/2)



Opstellen van de σ-tabel: terugval k



op het zicht zien we k = 3

hoe kunnen we efficiënt nieuwe k berekenen zonder k=5→4→3→...?

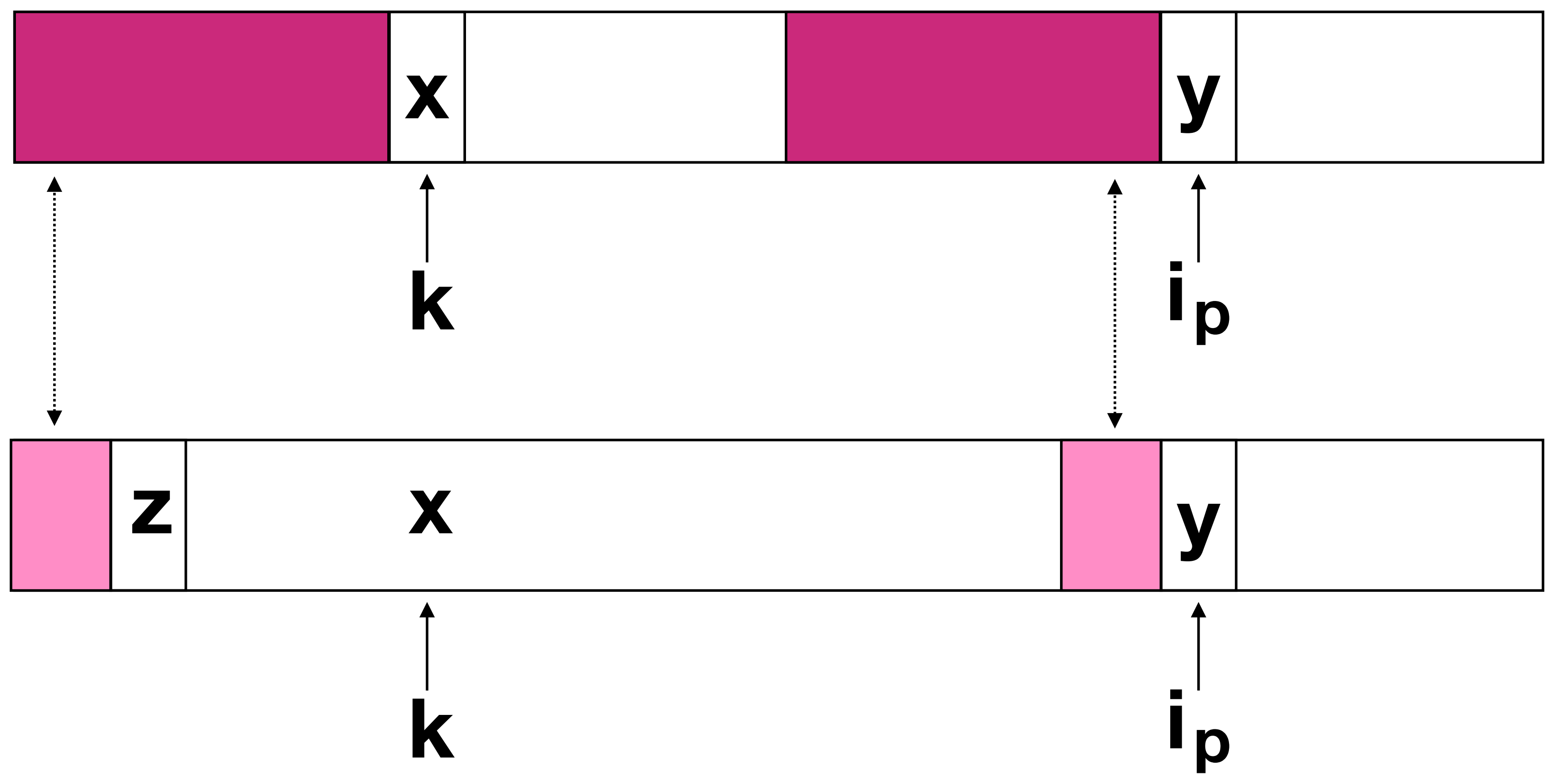
we zoeken een gelijke pre/suffix binnen huidige langste prefix met lengte k

de langste pre/suffix binnen prefix van p met lengte k = σ(k)

Opstellen van de σ-tabel: vergelijk patroon met zichzelf

Voor elke i_p zoeken we de $k=\sigma(i_p)$, d.w.z. het langste ***roze*** *stuk dat zowel prefix als suffix-tot-i_p is:*

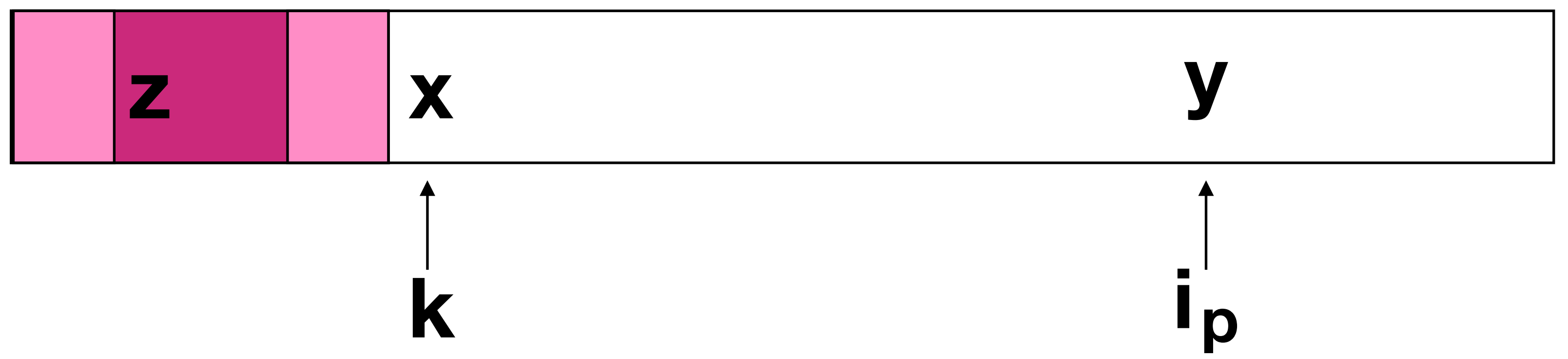
Is x=y? Zoja, schuif k en i_p op en beschouw de volgende x en y om misschien een nóg langer roze stuk te vinden



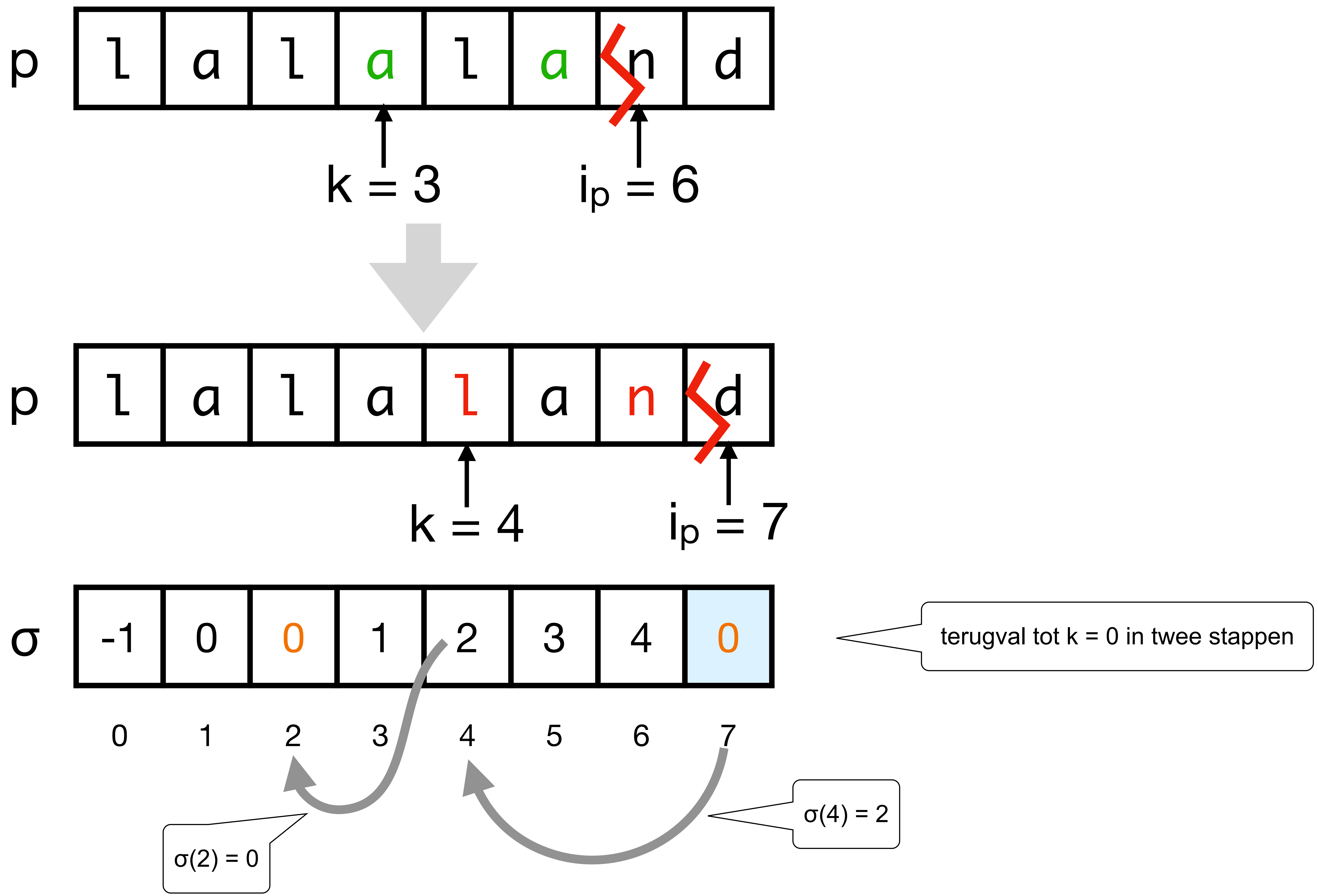
Zonee, is er dan misschien een korter prefix/suffix-tot-i_p dat y (samen met een zekere z) dan misschien langer maakt?

Maar zo'n korter prefix/suffix-tot-i_p moet noodzakelijkerwijs ook een prefix/suffix van het roze stuk zélf zijn (zie beide stippellijnen)

Dus zoeken we in dat geval het langste prefix/suffix-tot-k (= reeds berekend toen i_p nog k was).
En dat kennen we al, namelijk σ(k). Dus zoeken we verder met σ(k) en i_p



Opstellen van de σ-tabel: herneming eerste voorbeeld



Het Algoritme voor 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))
    (when (< i-p n-p)
      (cond
        ((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)))
```

$\forall$ i_p *van 2 tot* n_p*: zoek de langste k*

Terugval van k naar σ(k)

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

i-p = 3 / k = 1
    #(l a l a l a n d)
    #(0 0 0 1 0 0 0 0)

i-p = 4 / k = 2
    #(l a l a l a n d)
    #(0 0 0 1 2 0 0 0)

i-p = 5 / k = 3
    #(l a l a l a n d)
    #(0 0 0 1 2 3 0 0)

i-p = 6 / k = 4
    #(l a l a l a n d)
    #(0 0 0 1 2 3 4 0)

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)
```

Performantie Fase #1: bepalen van σ

Evolutie van i_p - k is maatstaf voor "vooruitgang"

Beschouw $i_p - k$:

i_p ging omhoog

- *in de eerste tak: van i_p en k naar $i_p +1$ en $k+1$ dus is $i_p - k$ constant*

$\sigma(k) < k$

- *in de tweede tak: i_p blijft en k naar $\sigma(k)$ dus is $i_p - k$ verhoogd*
- *in de derde tak: i_p naar $i_p +1$ en k blijft dus is $i_p - k$ verhoogd*

i_p - k gaat naar omhoog, en anders gaat i_p omhoog

$i_p - k \leq i_p$ en $i_p \leq n_p$ en dus wordt de lus maximaal $2n_p$ keer uitgevoerd

Conclusie: het bepalen van σ is in $O(n_p)$

Performantie Fase #2: het KMP Algoritme zélf

De evolutie van $i_t + i_p$ in de loop: een maatstaf voor de vooruitgang.

- *Ofwel match:* $i_t + i_p \Rightarrow i_t + (i_p + 1)$
- *Ofwel geen match:* $i_t + i_p \Rightarrow (i_t + i_p - \sigma(i_p)) + \sigma(i_p) = i_t + i_p$

$\sigma(i_p) < i_p$

Dus ofwel gaat i_t+i_p omhoog,
ofwel gaat i_t omhoog

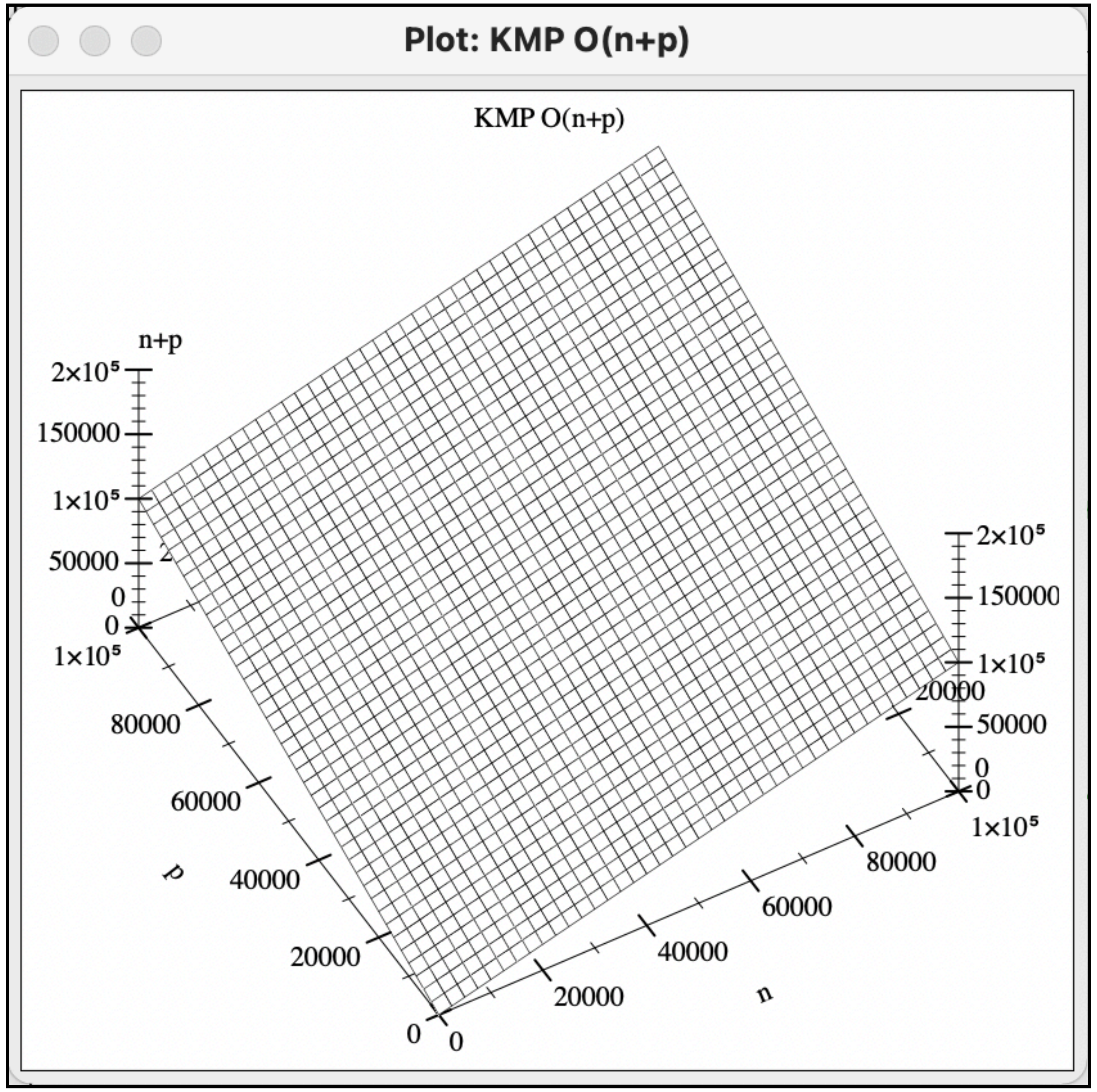
De som $i_t + i_p$ verhoogt of stagneert (maar dan gaat i_t omhoog) in elke slag van de lus. Dus wordt de lus maximaal $2n_t$ keer uitgevoerd.

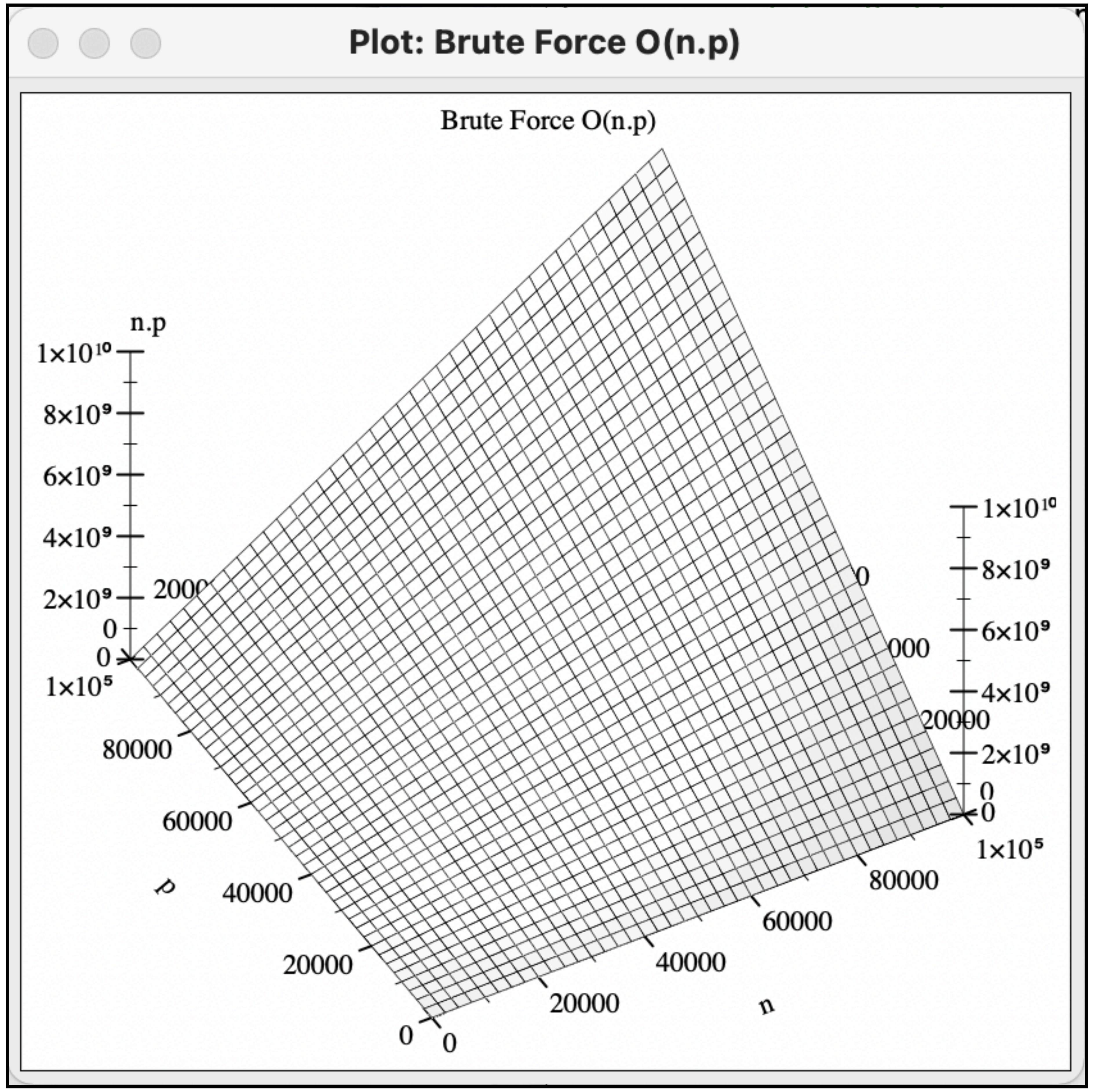
Conclusie: het matchen zélf is in $O(n_t)$

Conclusie KMP Performantie

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

Vergelijk de Z-as!





Hoofdstuk 2: Moraal van het 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 aan mekaar plakken is niet moeilijk. Strings correct weer uit mekaar halen is moeilijk en tijdrovend. Strings dienen bijgevolg enkel om met gebruikers te communiceren! Strings vormen de armste datastructuur die er bestaat.

Hoofdstuk 2