String Matching

String Matching (1/42)

Έστω ένα αλφαριθμητικό το οποίο είναι αποθηκευμένο σε ένα πίνακα T[1..n] με μήκος n και μια διάταξη (pattern) P[1..m] μήκους m<=n

Υποθέτουμε πως τα στοιχεία των T και P είναι χαρακτήρες μέσα από ένα αλφάβητο Σ

Λέμε πως το P συναντάται με μετατόπιση s στο T (ισοδύναμα λέμε πως το P ξεκινά στη θέση s+1) εφόσον 0<=s<=n-m και T[s+1..s+m]=P[1..m]

Ισχύει πως T[s+j]=P[j] με 1<=j<=m

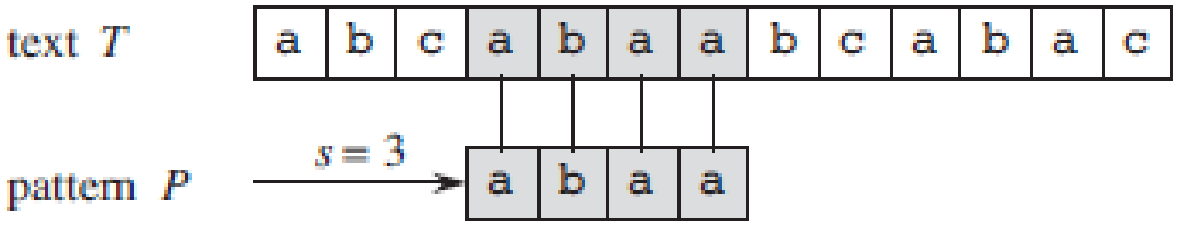
String Matching (2/42)

Αν το P συναντάται στο T τότε το s είναι μια **έγκυρη μετατόπιση (valid shift)**, διαφορετικά είναι μια **μη έγκυρη μετατόπιση (invalid shift)**

Το string matching problem έγκειται στο να βρούμε όλες τις έγκυρες μετατοπίσεις s

String Matching (3/42)

Παράδειγμα



Αλγόριθμοι

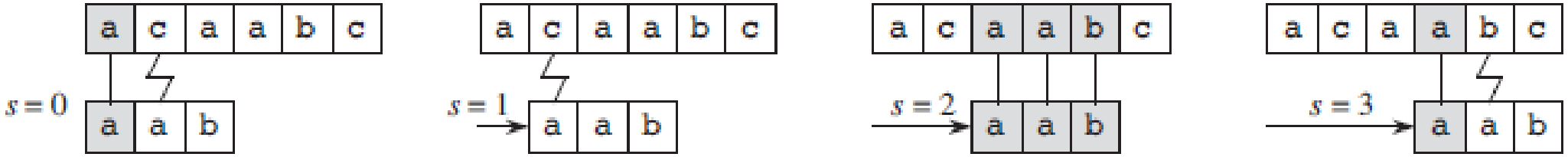
Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m \lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

String Matching (4/42)

Naïve String Matching

- Βρίσκει όλες τις εμφανίσεις του pattern ψάχνοντας σε όλες τις θέσεις του T
- Παράδειγμα εκτέλεσης

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```



String Matching (5/42)

Naïve String Matching

- Για κάθε μια από τις πιθανές τιμές του s (n-m+1) , ο αλγόριθμος στη γραμμή 4 πρέπει να ελέγξει όλες τις m θέσεις του pattern
- Η πολυπλοκότητα στη χειρότερη περίπτωση θα είναι Θ((n-m+1)m) το οποίο μπορεί να καταλήξει σε $\Theta(n^2)$ αν m = n/2
- Ο αλγόριθμος δεν απαιτεί preprocessing

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

String Matching (6/42)

Naïve String Matching

- Ο αλγόριθμος δεν εκμεταλλεύεται την πληροφορία που αποκτά για το T κατά τις πρώτες επαναλήψεις
- Αυτή η πληροφορία μπορεί να είναι χρήσιμη κατά τις υπόλοιπες επαναλήψεις

String Matching (7/42)

The Rabin Carp Algorithm

- Υιοθετεί preprocessing κόστους Θ(m) και η πολυπλοκότητα στη χειρότερη περίπτωση είναι Θ((n-m+1)m)
- Βασιζόμενοι σε κάποιες υποθέσεις, η επίδοση στη μέση περίπτωση είναι πολύ καλύτερη
- Ο αλγόριθμος υπολογίζει **μια αριθμητική τιμή (hash)** για το P και για κάθε substring μήκους m του T
- Στη συνέχεια συγκρίνει τα πραγματικά σύμβολα
- Αν βρει κάποιο ταίριασμα, τότε συγκρίνει το P με το συγκεκριμένο substring με τη βοήθεια του naïve matcher

String Matching (8/42)

The Rabin Carp Algorithm

- Έστω p η αριθμητική τιμή του P
- Έστω t_s η αριθμητική τιμή του substring μήκους m του T
- **Προφανώς $p=t_s$ εφόσον T[s+1..s+m] = P[1..m]**
- Συνεπώς το s είναι μια έγκυρη μετατόπιση εφόσον $p=t_s$
- Ο υπολογισμός του p γίνεται με τη βοήθεια του κανόνα του Horner

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$$

- Ομοίως, μπορούμε να υπολογίσουμε το t_0 από το T[1..m]
- Κάνουμε την υπόθεση πως στο T και το P έχουμε αριθμητικές τιμές

String Matching (9/42)

The Rabin Carp Algorithm

- Για τον υπολογισμό των υπόλοιπων t_1, t_2,…, t_{n-m} παρατηρούμε πως μπορούμε να υπολογίσουμε το t_{s+1} από το t_s σε σταθερό χρόνο αφού

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

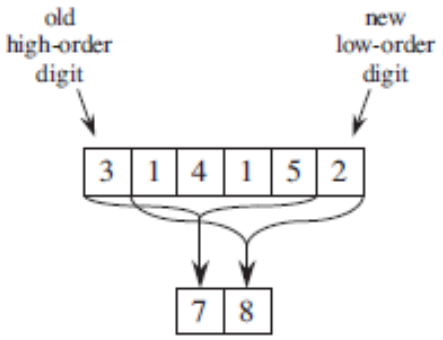
- t_s → κώδικας για τους χαρακτήρες [s+1, s+m]
- Η αφαίρεση του $10^{m-1}T[s+1]$, εξαλείφει τα πιο σημαντικά ψηφία του t_s και ο πολλαπλασιασμός με το 10 προκαλεί μετατόπιση των αριθμών προς τα αριστερά κατά ένα ψηφίο
- Η πρόσθεση του T[s+m+1] μας φέρνει στη σωστή θέση τα λιγότερο σημαντικά ψηφία

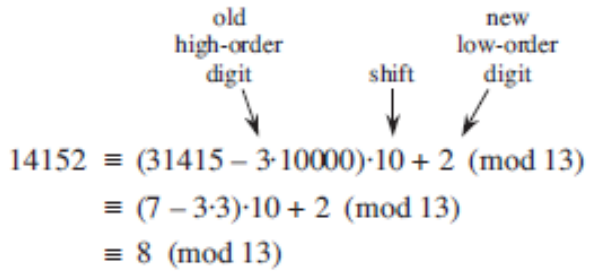
String Matching (10/42)

The Rabin Carp Algorithm

- Παράδειγμα
  - m=5, t_s=31415
  - Πρέπει να εξαλείψουμε τα T[s+1] = 3 σημαντικά ψηφία και να φέρουμε στη σωστή θέση τα λιγότερο σημαντικά ψηφία (T[s+5+1] = 2)
  - Παίρνουμε

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \end{aligned}$$





Η μεταβλητή q = 13 είναι ένας πρώτος αριθμός τέτοιος ώστε το 10q να χωράει στη λέξη του Η/Υ

String Matching (11/42)

The Rabin Carp Algorithm

- Για τον υπολογισμό των p και t_s παίρνουμε το υπόλοιπο της διαίρεσης με μια μεταβλητή q
- Αν επιλέξουμε το q ως ένα πρώτο αριθμό τέτοιο ώστε το 10q να χωράει στη λέξη του Η/Υ τότε όλοι οι υπολογισμοί θα γίνει με απλή ακρίβεια
- Γενικά, αν έχουμε ένα αλφάβητο d στοιχείων {0,1,2,3, ..., d-1}, επιλέγουμε το q τέτοιο ώστε το **dq** να χωρά στη λέξη του Η/Υ
- Η εξίσωση υπολογισμού γίνεται

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

- με $h = d^{m-1}$

String Matching (12/42)

The Rabin Carp Algorithm

- **Το d το λαμβάνουμε ίσο με |Σ|**
- **Όλοι οι χαρακτήρες αναπαριστούνται ως d ψηφία**

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

```
1  n = T.length
2  m = P.length
3  h = d^(m−1) mod q
4  p = 0
5  t_0 = 0
6  for i = 1 to m  // preprocessing
7      p = (dp + P[i]) mod q
8      t_0 = (d t_0 + T[i]) mod 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]) mod q
```

String Matching (13/42)

The Knuth-Morris-Pratt Algorithm

- Υιοθετεί μια συνάρτηση που υπολογίζεται με βάση το P
- Αποθηκεύουμε την πληροφορία σε ένα πίνακα π[1..m]
- Ο π μας επιτρέπει να υπολογίσουμε μια συνάρτηση μετάβασης δ (δ: QXΣ → Q
- Q είναι ένα σύνολο καταστάσεων σχετικά με ελέγχους που κάνουμε πάνω στο T)
- Για κάθε κατάσταση q = 0,1,2, …, m και κάθε χαρακτήρα α που ανήκει στο Σ, η τιμή π[q] περιέχει την πληροφορία που απαιτείται για να υπολογιστεί το δ(q,α) που δεν εξαρτάται από το α
- Ο π έχει m και η δ έχει Θ(m |Σ|) στοιχεία

String Matching (14/42)

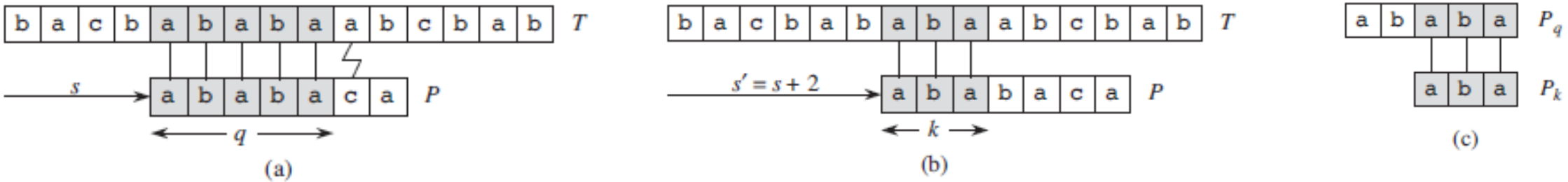
The Knuth-Morris-Pratt Algorithm

- Η π περιλαμβάνει γνώση για το πως το P ταιριάζει σχετικά με μετακινήσεις του
- Αυτή η γνώση υιοθετείται ώστε να αποφευχθούν άσκοπες μετακινήσεις του P
- Επίσης αποφεύγουμε να υπολογίσουμε όλα τα στοιχεία της δ

String Matching (15/42)

The Knuth-Morris-Pratt Algorithm

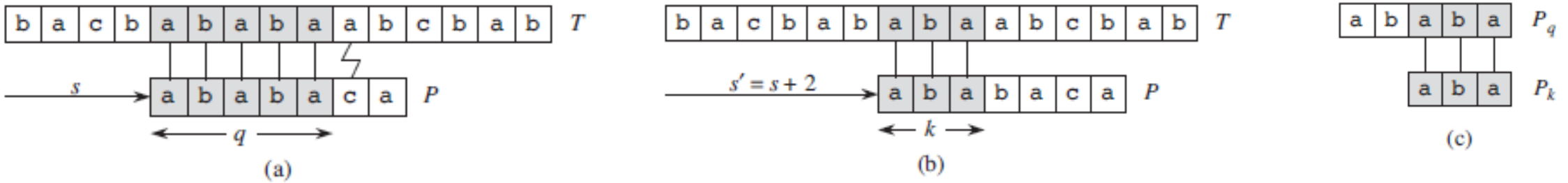
- Στο επόμενο παράδειγμα παίρνουμε q=5 και P=ababaca
- Βλέπουμε πως 5 χαρακτήρες ταιριάζουν επιτυχώς ενώ στον 6ο έχουμε αποτυχία
- Γνωρίζοντας τους q χαρακτήρες μπορούμε να βρούμε κάποιες μη έγκυρες μετακινήσεις



String Matching (16/42)

The Knuth-Morris-Pratt Algorithm

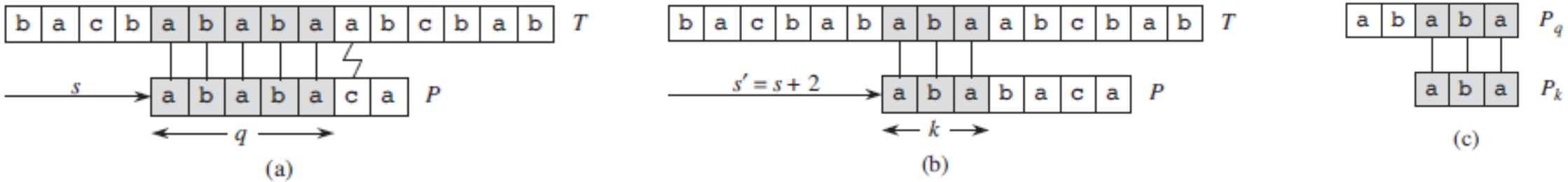
- Η μετακίνηση s+1 δεν είναι έγκυρη αφού ο χαρακτήρας a θα ευθυγραμμιστεί με ένα χαρακτήρα του T που ξέρουμε πως δεν ταιριάζει αλλά ταιριάζει με το 2ο χαρακτήρα (b)
- Η μετακίνηση s’=s+2 ευθυγραμμίζει 3 χαρακτήρες που ταιριάζουν



String Matching (17/42)

The Knuth-Morris-Pratt Algorithm

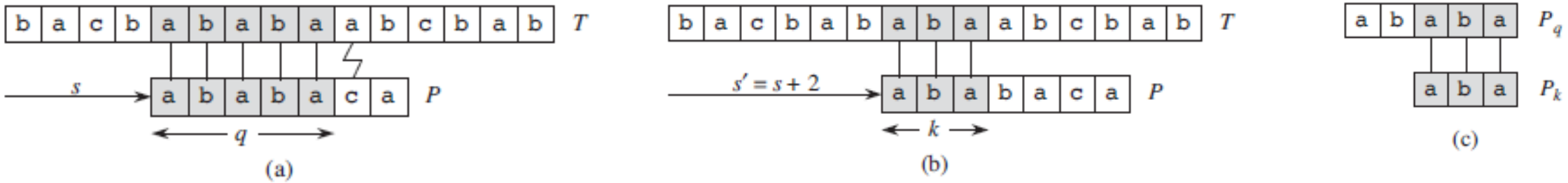
- Το ερώτημα που γεννάται είναι:
  - Δοσμένου των P[1..q] χαρακτήρων που ταιριάζουν με χαρακτήρες του T, T[s+1..s+q], ποια είναι η ελάχιστη μετατόπιση s'>s τέτοια ώστε για κάποιο k<q να έχουμε P[1..k]=T[s'+1..s'+k] όπου s'+k=s+q;



String Matching (18/42)

The Knuth-Morris-Pratt Algorithm

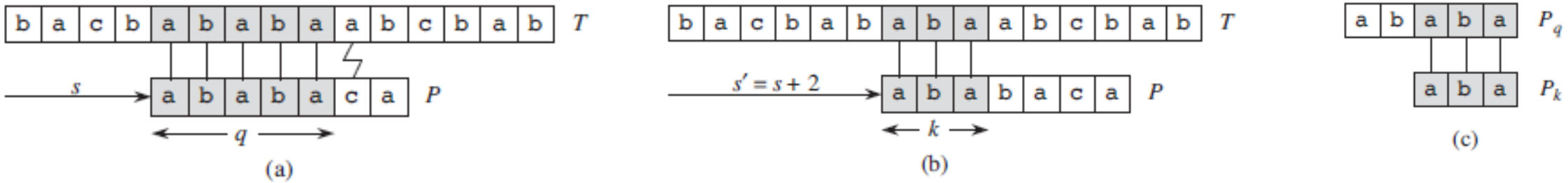
- Προσθέτουμε τη διαφορά q-k του μήκους των prefixes του P στο s έτσι ώστε να παραχθεί μια νέα μετατόπιση s’=s+q και παράγουμε τις μετατοπίσεις s+1, s+2, …, s+q-1
- Σε κάθε μετατόπιση δεν χρειάζεται να συγκρίνουμε τους πρώτους k χαρακτήρες διότι η εξίσωση P[1..k]=t[s’+1..s’+k] εξασφαλίζει το ταίριασμα



String Matching (19/42)

The Knuth-Morris-Pratt Algorithm

- Η τελευταία εικόνα μας δείχνει πως μπορούμε να προ-υπολογίσουμε την απαραίτητη πληροφορία συγκρίνοντας το P με τον εαυτό του
- Αφού το T[s'+1..s'+k] είναι τμήμα του T, ζητάμε το μεγαλύτερο k έτσι ώστε το P_q να περιλαμβάνει το P_k



String Matching (20/42)

The Knuth-Morris-Pratt Algorithm

- Η συνάρτηση π ορίζεται ως εξής:

$$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$$
$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

- Το π[q] είναι το μήκος του μεγαλύτερου prefix του P που είναι ένα κατάλληλο επίθημα (suffix) του P_q

String Matching (21/42)

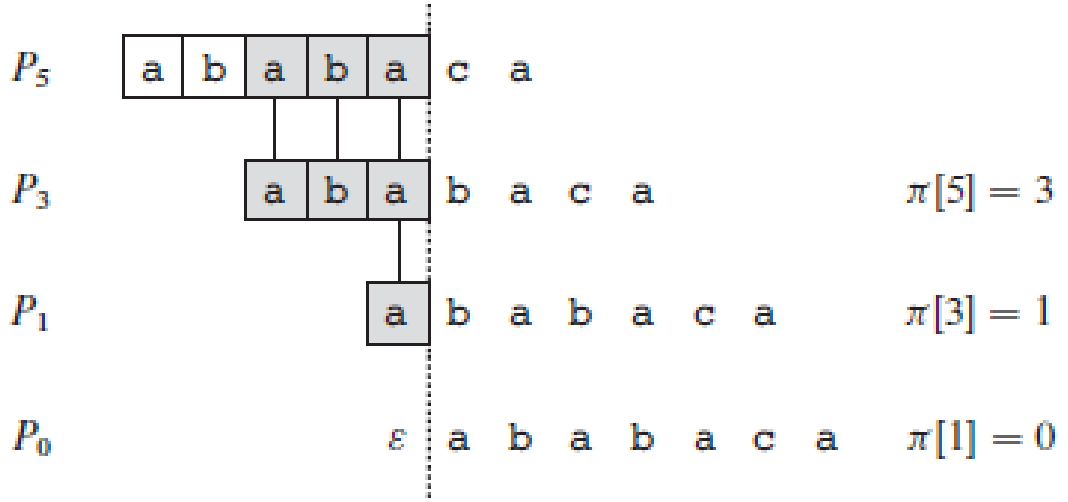
The Knuth-Morris-Pratt Algorithm

- Παράδειγμα

$$\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$$
$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

String Matching (22/42)

The Knuth-Morris-Pratt Algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1 . . m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

String Matching (23/42)

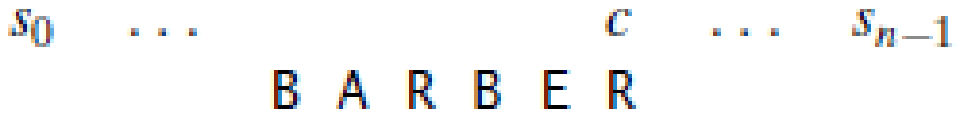
The Horspool's Algorithm

- Ας υποθέσουμε πως ψάχνουμε τη συμβολοσειρά BARBER σε ένα κείμενο T
- Μπορούμε να ξεκινήσουμε από το τελευταίο R και να μετακινούμαστε προς τα αριστερά
- Συγκρίνουμε ζεύγη χαρακτήρων
- Αν όλοι οι χαρακτήρες είναι ίδιοι τότε έχουμε βρει τη συμβολοσειρά
- Σε αυτή την περίπτωση η αναζήτηση μπορεί να τερματίσει ή να συνεχιστεί για την εύρεση επόμενων εμφανίσεων

String Matching (24/42)

The Horspool's Algorithm

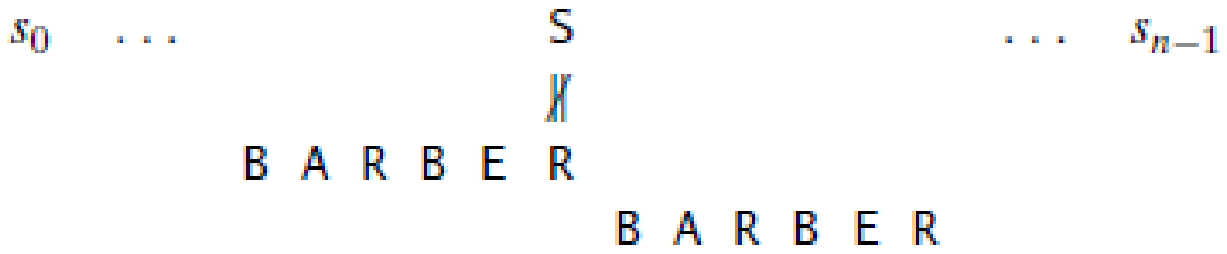
- Αν βρούμε κάποια διαφορά, τότε το P μετακινείται προς τα δεξιά
- Ο αλγόριθμος προσπαθεί να βρει τη μεγαλύτερη δυνατή μετατόπιση
- Κοιτάζει το χαρακτήρα c ο οποίος έχει ευθυγραμμιστεί με τον τελευταίο χαρακτήρα του P
- Ισχύει ακόμα και αν ο c είναι όμοιος με την τελευταία θέση του P



String Matching (25/42)

The Horspool's Algorithm

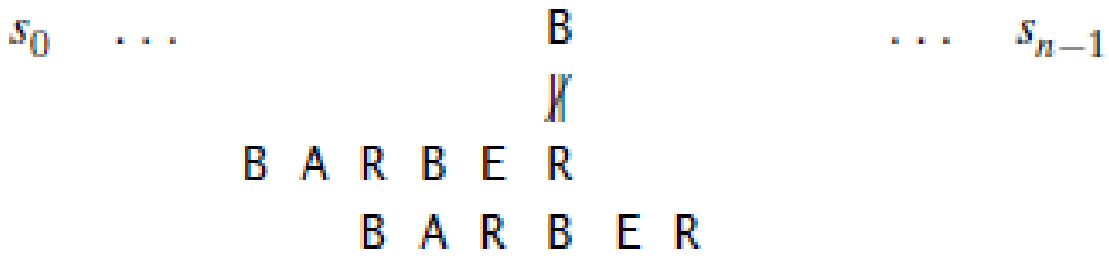
- Περίπτωση 1: αν το c δεν ταιριάζει / δεν υπάρχει, τότε μπορούμε να μετακινήσουμε το P ως προς ολόκληρο το μήκος του



String Matching (26/42)

The Horspool's Algorithm

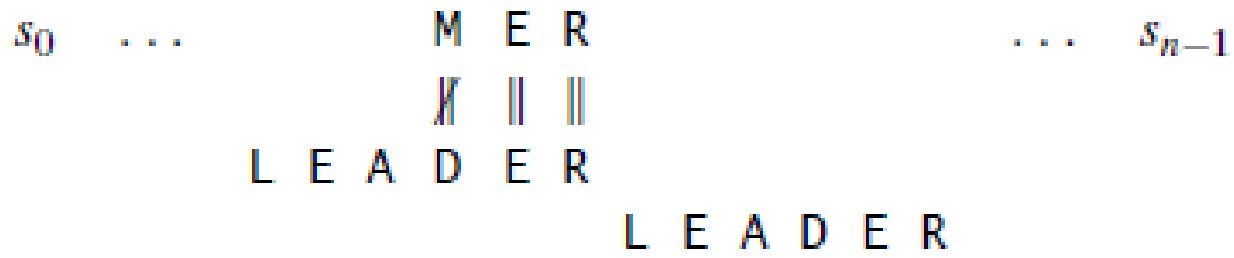
- Περίπτωση 2: Αν υπάρχουν εμφανίσεις του c στο P αλλά δεν είναι στην τελευταία θέση, η μετατόπιση μπορεί να γίνει ως προς την πιο δεξιά εμφάνιση του c στο P



String Matching (27/42)

The Horspool's Algorithm

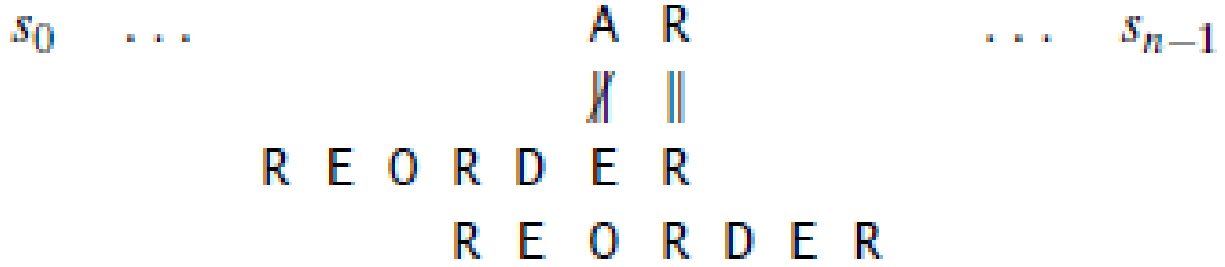
- Περίπτωση 3: Αν το c υπάρχει στην τελευταία θέση του P αλλά δεν υπάχρει το c στις υπόλοιπες θέσεις (m-1), έχουμε όμοια περίπτωση με την 1η.



String Matching (28/42)

The Horspool's Algorithm

- Περίπτωση 4: αν το c υπάρχει στην τελευταία θέση αλλά και σε κάποιες άλλες ενδιάμεσες θέσεις, τότε έχουμε όμοια περίπτωση με τη 2η



String Matching (29/42)

The Horspool's Algorithm

- Μπορούμε να υπολογίσουμε σε προγενέστερο χρόνο το μήκος των μετατοπίσεων και να τις αποθηκεύσουμε σε ένα πίνακα
- Τα στοιχεία του πίνακα δείχνουν το μήκος των μετατοπίσεων ως ακολούθως:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

String Matching (30/42)

The Horspool's Algorithm

- Αλγόριθμος εύρεσης μήκους μετατοπίσεων

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

String Matching (31/42)

The Horspool's Algorithm

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//         or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else  i ← i + Table[T[i]]
return −1
```

String Matching (32/42)

The Horspool's Algorithm

- Παράδειγμα

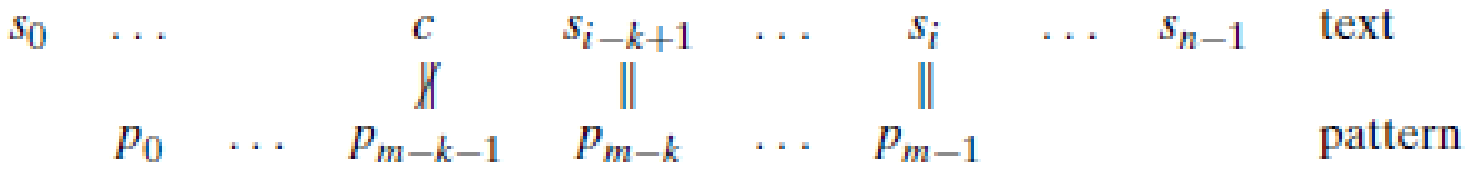
character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                 B A R B E R
```

String Matching (33/42)

The Boyer-Moore Algorithm

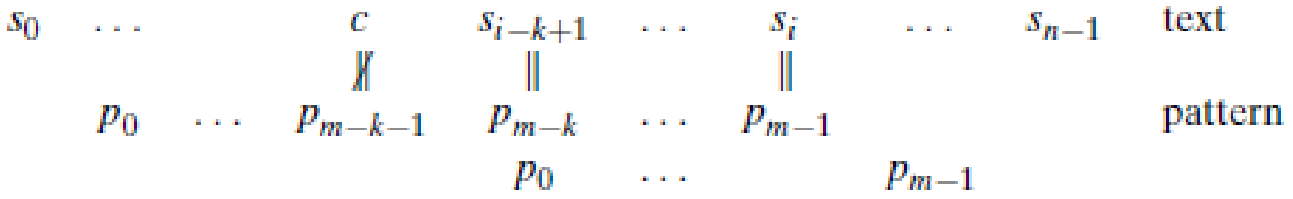
- Αν η σύγκριση του χαρακτήρα που ευθυγραμμίζεται με το c (τελευταίος χαρακτήρας του P), ο αλγόριθμος κάνει τις ίδιες λειτουργίες με τον Horspool
- Μετακινεί το P τόσες θέσεις όσες δείχνει ο πίνακας που έχει ήδη προ-υπολογιστεί
- Οι δύο αλγόριθμοι όμως συμπεριφέρονται διαφορετικά όταν μετά από k χαρακτήρες που έχουν ταιριάξει πριν παρατηρηθεί κάποια διαφορά



String Matching (34/42)

The Boyer-Moore Algorithm

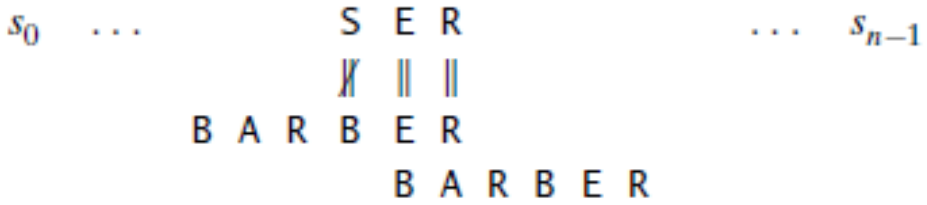
- Σε αυτές τις περιπτώσεις ο αλγόριθμος λαμβάνει υπόψιν του δύο ποσότητες:
  - Η πρώτη εξαρτάται από το c που προκάλεσε μια διαφορά (bad symbol shift). Αν το c δεν είναι στο P, μετατοπίζουμε το P ώστε να ξεπεράσει το c – το μήκος της μετατόπισης μπορεί να υπολογιστεί από το $t_1(c) – k$, όπου $t_1(c)$ είναι η τιμή στον προ-υπολογισμένο πίνακα όπως αυτός υπολογίζεται από τον αλγόριθμο Horspool και k είναι το πλήθος των χαρακτήρων που ταιριάζουν



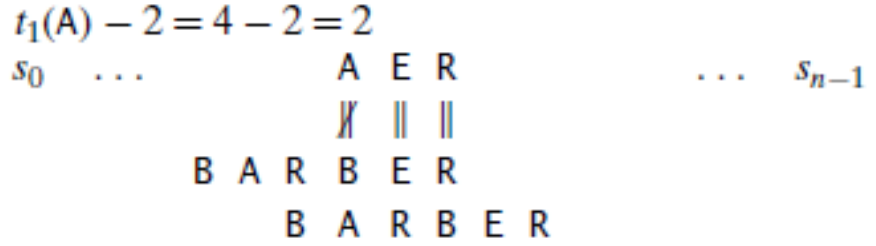
String Matching (35/42)

The Boyer-Moore Algorithm

- Παράδειγμα:
- Αν ψάχνουμε το BARBER και ταιριάξουμε τους τελευταίους 2 χαρακτήρες, μπορούμε να μετατοπίσουμε το P κατά $t_1(S)-2=6-2=4$ θέσεις



- Ο ίδιος τύπος μπορεί να υιοθετηθεί όταν ο c υπάρχει στο P δεδομένου ότι $t_1(c) - k > 0$
- Παράδειγμα



String Matching (36/42)

The Boyer-Moore Algorithm

- Όταν $t_1(c) - k < 0$, τότε φυσικά δεν θέλουμε να κάνουμε μετατόπιση με αρνητικό αριθμό θέσεων, όποτε υιοθετούμε μια brute force προσέγγιση και μετατοπίζουμε κατά 1
- Γενικά, η μετατόπιση για ένα bad symbol d_1, υπολογίζεται με βάση το πρόσημο της παράστασης $t_1(c) - k$

$$d_1 = \max\{t_1(c) - k, 1\}$$

- Ο δεύτερος τύπος μετατόπισης εξαρτάται από το επιτυχές ταίριασμα των τελευταίων k χαρακτήρων του P
- Συμβολίζουμε με suff(k) το τερματικό τμήμα του P που αποτελεί ένα επίθημα
- Ονομάζεται good suffix shift

String Matching (37/42)

The Boyer-Moore Algorithm

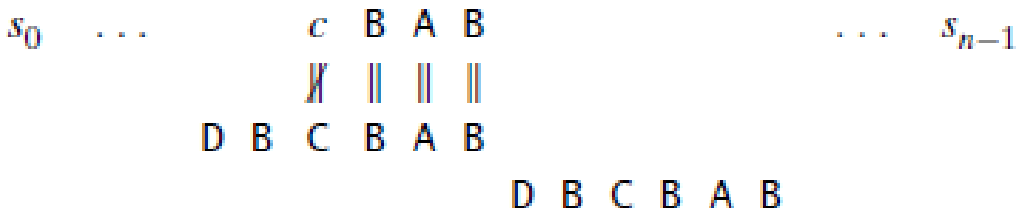
- Ας θεωρήσουμε την περίπτωση όπου υπάρχει και άλλο suff(k) στο P στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως για το πιο δεξιά suff(k)
- Σε αυτή την περίπτωση μπορούμε να μετατοπίσουμε το P κατά μια απόσταση d_2 ανάμεσα στα δύο suff(k)
- Παράδειγμα

k	pattern	d_2
1	ABC$\overline{\text{B}}$AB	2
2	$\overline{\text{AB}}$CBAB	4

String Matching (38/42)

The Boyer-Moore Algorithm

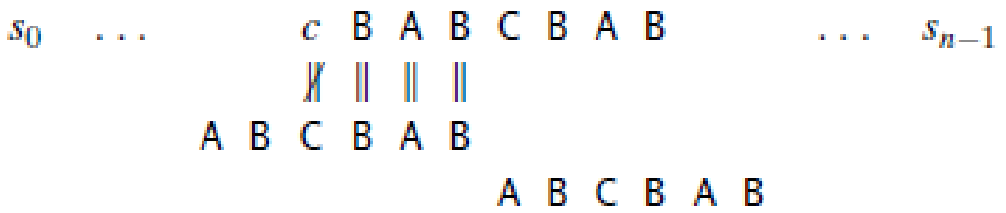
- Τι γίνεται όμως στην περίπτωση όπου δεν υπάρχει άλλη εμφάνιση του suff(k) στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως στο πιο δεξιά suff(k)?
- Σε αυτή την περίπτωση μπορούμε να μετατοπίσουμε το P ως προς το συνολικό μέγεθός του
- Για παράδειγμα, αν k = 3 και P=DBCBAB μπορεί να μετατοπισθεί κατά 6 χαρακτήρες



String Matching (39/42)

The Boyer-Moore Algorithm

- Δυστυχώς όμως η μετατόπιση ως προς το συνολικό μήκος του P όταν δεν υπάρχει άλλη εμφάνιση του suff(k) στην οποία δεν προηγείται ο ίδιος χαρακτήρας όπως στο πιο δεξιά suff(k), δεν είναι πάντα σωστή
- Για το P=ABCBAB και k=3, η μετατόπιση κατά 6 χαρακτήρες, θα χάσει μια υπο-συμβολοσειρά που ξεκινά με το AB ευθυγραμμισμένο με τους τελευταίους δύο χαρακτήρες του T



String Matching (40/42)

The Boyer-Moore Algorithm

- Για να αποφύγουμε το πρόβλημα αυτό, πρέπει να βρούμε το μεγαλύτερο prefix μεγέθους l < k που ταιριάζει με το suffix ιδίου μήκους l
- Αν υπάρχει τέτοιο πρόθεμα, το μήκος μετατόπισης d_2 υπολογίζεται ως η απόσταση του prefix και του suffix
- Εναλλακτικά, το d_2 τίθεται ίσο με το μήκος του P
- Παράδειγμα

k	pattern	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4
3	$\overline{\text{AB}}$C$\underline{\text{BAB}}$	4
4	$\overline{\text{AB}}\underline{\text{C}}$BAB	4
5	$\overline{\text{A}}\underline{\text{BC}}$BAB	4

String Matching (41/42)

The Boyer-Moore Algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

String Matching (42/42)

The Boyer-Moore Algorithm

- Παράδειγμα εύρεσης BAOBAB

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

k	pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5