Συμβολοσειρές

- Σημαντικές λόγω των εφαρμογών τους
  - Επεξεργασία κειμένου
  - Διαδίκτυο
  - Επεξεργασία βιολογικών δεδομένων
    - DNA ως συμβολοσειρές 4 συμβόλων: C (κυτοσίνη), G (γουανίνη), T (θυμίνη), A (αδενίνη)

Ορισμοί

- Σ: αλφάβητο
- Σ*: όλες οι συμβολοσειρές πεπερασμένου μήκους, συμπεριλαμβανομένης της κενής $\in$
- $|x|$ μήκος συμβολοσειράς x
- $x=\alpha\beta$, $y=\gamma\delta\varepsilon$ συμβολοσειρές
  $xy=\alpha\beta\gamma\delta\varepsilon$ συμβολοσειρά συνενώσεώς τους
- $x=\alpha\beta\gamma\delta\varepsilon\zeta$
  - *Πρόθεμα μήκους i: τα i πρώτα σύμβολα, πχ, i=3, αβγ*
  - *Πρόσφυμα μήκους j: τα j τελευταία σύμβολα, πχ, j=4, γδεζ*

Ακριβές Ταίριασμα Προτύπου (Exact Pattern Matching)

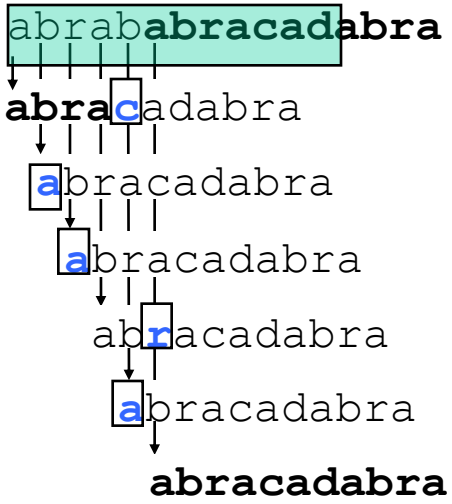
- *«Δοθεισών ενός κειμένου Τ, μήκους n, και ενός προτύπου Ρ, μήκους m, απαιτείται ο εντοπισμός όλων των στιγμιοτύπων του Ρ στο Τ.»*
  - Π.χ. για Τ=ααβαββαββββα και Ρ=βα, έχουμε

    Aα**βα**β**βα**βββ**βα**

- Εφαρμογές:
  - Κειμενογράφοι
  - αναζήτηση υποδομής σε ακολουθία DNA
  - αναζήτηση λέξεως στα περιεχόμενα μιας σελίδας διαδικτύου

Απλοϊκή Λύση

- Αντιπαραβολή του προτύπου, από όλες τις *n*-*m*+1 δυνατές θέσεις εκκινήσεως



Κώδικας

Algorithm bruteForcePM(**char []** T, **char []** P)
Input: Κείμενο T και πρότυπο P
Output: Εντοπισμός του P στο T

```
1. n = T.length; m = P.length;
2. for (i = 1; i = n- m + 1; i++){  // i η τρέχουσα θέση του παραθύρου
3.    j = 1;  // δείκτης θέσεως εντός παραθύρου-προτύπου
4.    while ((j <= m) && (T[i+j-1] == P[j]))
5.       j++;
6.    if (j == m +1)
7.      return i;      // βρέθηκε η αρχική θέση του προτύπου
8.  }
9.  return -1;        // δεν υπάρχει
```

Ανάλυση

- Πολυπλοκότητα $O(nm)$
  - Χειρότερη Περίπτωση

    Σε κάθε αντιπαραβολή να γίνονται m-1 επιτυχημένες συγκρίσεις και να σημειώνεται αποτυχία στην τελευταία. Πχ,

    T = `aaaaaaaaaaaaaaaaaaaaaaaaaaaaa`

    P = `aaaaab`

    n-m+1 αντιπαραβολές των m συγκρίσεων έκαστη

Knuth-Morris-Pratt

- Ανακαλύφθηκε ανεξάρτητα από τους Knuth-Morris και Pratt και, κατόπιν, παρουσιάστηκε ως κοινή εργασία το 1977
- Στηρίζεται στην γραμμική -$O(m)$, δηλ.- επεξεργασία του ***προτύπου και όχι του κειμένου,*** με σκοπό
  - την εξαγωγή βοηθητικού πίνακα f, και
  - την χρήση της πληροφορίας του f για την μείωση του αριθμού των αντιπαραβολών

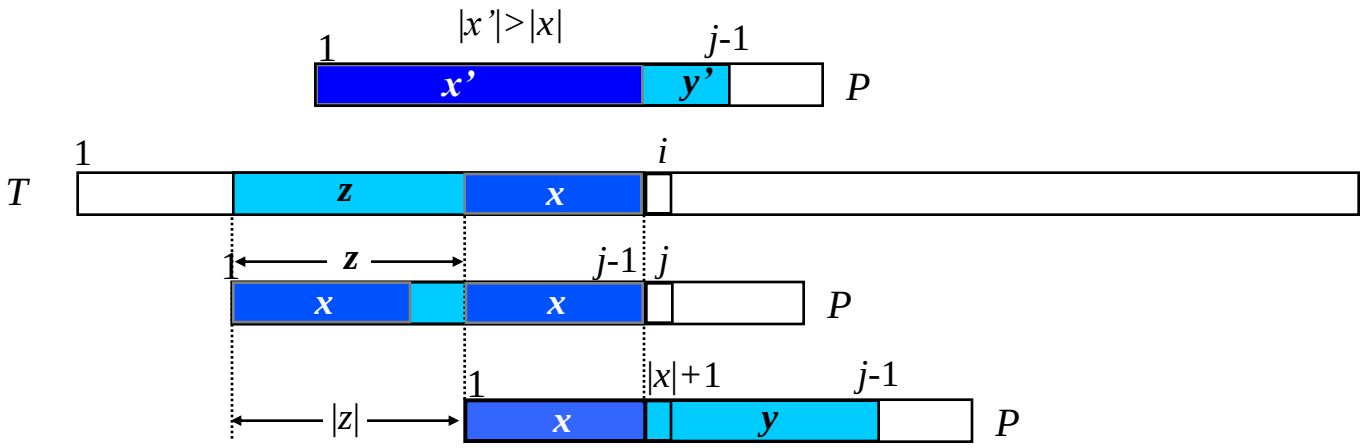
Πίνακας Αποτυχίας f

- $f[j]$ = ***μέγιστο πρόθεμα*** τού $P[1..j]$ που αποτελεί και ***πρόσφυμά του***

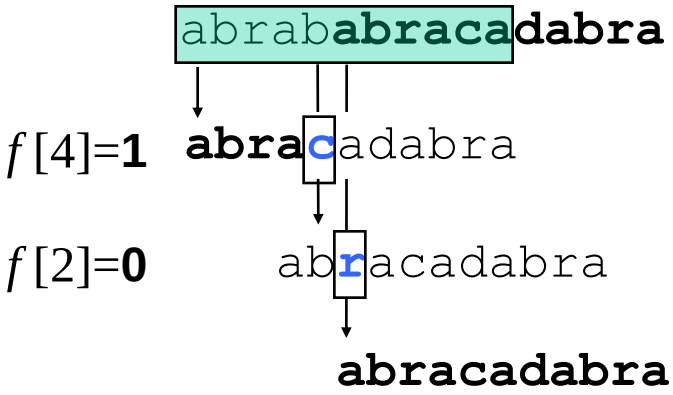
1	2	3	4	5	6	7	8	9	10	11
a										
0										

Ποια η χρήση του *f*

- Εάν έχουν ταιριαστεί οι *j*-1 πρώτοι χαρακτήρες, ενώ υπάρχει διαφορά στον *j*-στό, τότε
  - μπορούμε να ολισθήσουμε την αντιπαραβολή κατά *j*-1-*f*[*j*-1] θέσεις
  - δεν χάνουμε κανένα στιγμιότυπο!

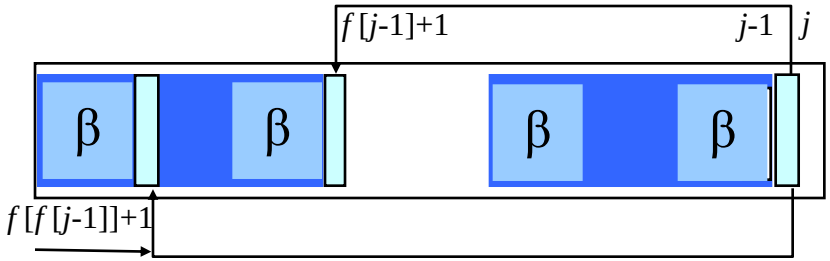


Παράδειγμα


abrababracadabra
f [4]=1
abracadabra
f [2]=0
abracadabra
abracadabra

Τρόπος Υπολογισμού

- Εάν $P[j]=P[f[j-1]+1]$ τότε $f[j] = f[j-1] + 1$
- Διαφορετικά,
  - είτε είναι το μέγιστο κοινό πρόθεμα του $P[1..f[j-1]]$ ($P[1..f[f[j-1]]]$) συν τον επόμενο χαρακτήρα
  - είτε συνεχίζουμε την αναζήτηση αναδρομικά…



```
Algorithm kmpPM(char[] T, char[] P)
1.  n = T.length; m = P.length;
2.  int[] f = failureKMP(P, new int[m]);
3.  int i = 1, j = 1;
4.  while (i < n){
5.    if (P[j] == T[i]){    // ταίριασμα
6.      if (j = m)          // εύρεση!
7.        return i - m;
8.      else{               // μετακίνηση εντός παραθύρου 1 θέση δεξιότερα
9.        i++;
10.       j++;
11.     }
12.   else if (j > 1)       // βρέθηκε ταίριασμα εντός παραθύρου
13.     j = f(j-1)          // οπότε γίνεται χρήση της failure function
14.   else
15.     i++;
16. }
17. return -1;
18.
19. int[] failureKMP(int[] P, int[] f){
20.   int m = P.length, k = 0;
21.   f[1] = 0;
22.   for (j = 2; j <= m; j++){
23.     while ((k > 0) && (P[k+1] != P[j]))  // καθορίζει και την πολυπλοκότητα υπολογισμού
24.         k = f[k];  // P[f[k]+1] ≠ P[j]
25.     if (P[k+1] == P[j])
26.       k++;       // κρίσιμο σημείο!!!
27.     f[j] = k;
28.   }
29.   return f;
```

Πολυπλοκότητες

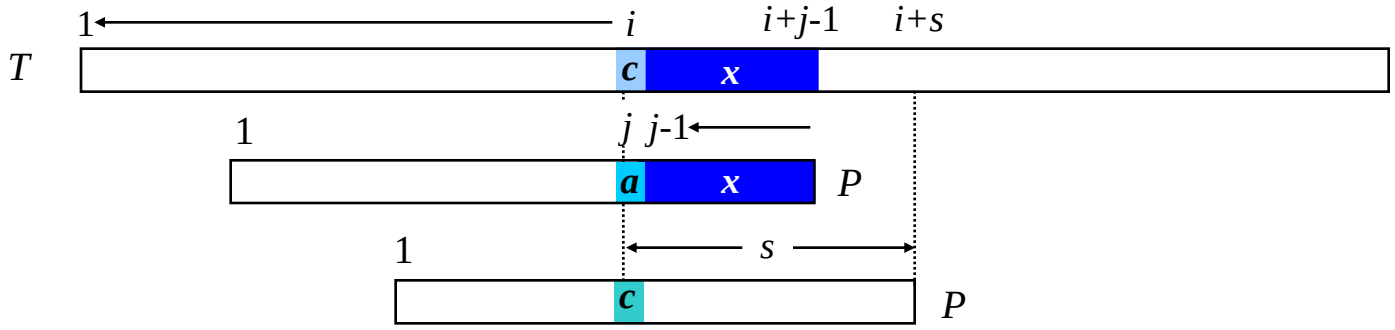
- Ο υπολογισμός της f γίνεται σε χρόνο $O(m)$
  - Η γραμμή 24 του kmpPM καθορίζει την πολυπλοκότητα
- Οι συγκρίσεις είναι μεταξύ n και $2n$ διότι κάθε φορά
  - εάν η σύγκριση είναι επιτυχής, μεταβαίνουμε στον επόμενο χαρακτήρα του κειμένου και ο αντίστοιχος δείκτης i ποτέ δεν μειούται
  - εάν είναι ανεπιτυχής, ολισθαίνουμε τουλάχιστον κατά μία θέση το παράθυρο στο κείμενο, οπότε μία θέση του κειμένου εγκαταλείπεται οριστικά - ο δείκτης j ποτέ δεν μειούται

Boyer-Moore

- Χρησιμοποιείται στο εργαλείο agrep
- Κύρια ιδέα
  - Οι συγκρίσεις να γίνονται *από δεξιά προς τα αριστερά*
  - Όταν σημειώνεται αποτυχία, εφαρμόζονται δύο ευρετικοί κανόνες (heuristics), ώστε βρεθεί η μέγιστη ολίσθηση:
    1. Ευρετικό εμφανίσεως
    2. Ευρετικό ταιριάσματος

Ευρετικό Εμφανίσεως (Occurrence Heuristic)

- Ταίριασμα του χαρακτήρα *c* του κειμένου που προκάλεσε την αποτυχία με την *δεξιότερη εμφάνισή* του στο πρότυπο.
- Εάν αυτή είναι δεξιότερα της *j*, τότε, απλώς, ολισθαίνουμε κατά μία θέση.
- Πολυπλοκότητα O(Σ)



Παράδειγμα

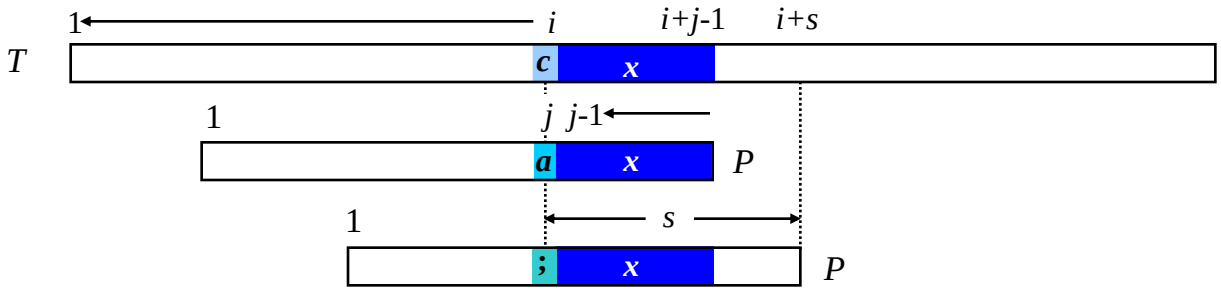
d’	a	b	c	d	r
	11	**9**	**5**	**7**	**10**

d= m-d’	a	b	c	d	r
	0	**2**	**6**	**4**	**1**

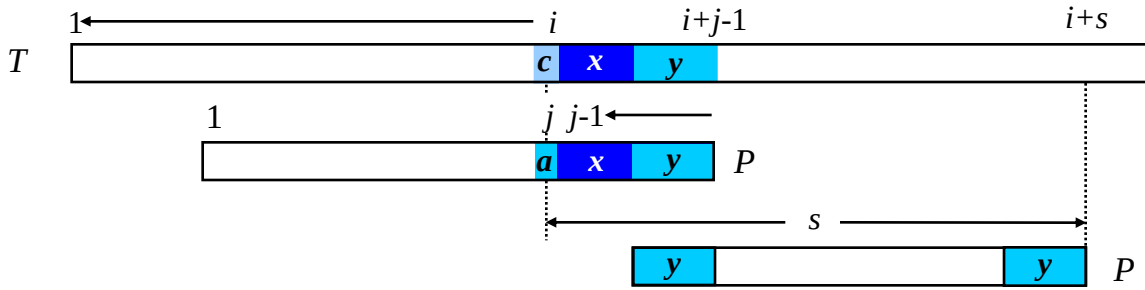
1	2	3	4	5	6	7	8	9	10	11
a	b	r	a	**c**	a	d	a	b	r	a

Ευρετικό Ταιριάσματος (Match Heuristic)

- Δεξιότερο μέγιστο ταίριασμα των χαρακτήρων του κοινού προσφύματος με διαφορετικό χαρακτήρα-«κεφαλή» από αυτόν που προκάλεσε την διαφορά.



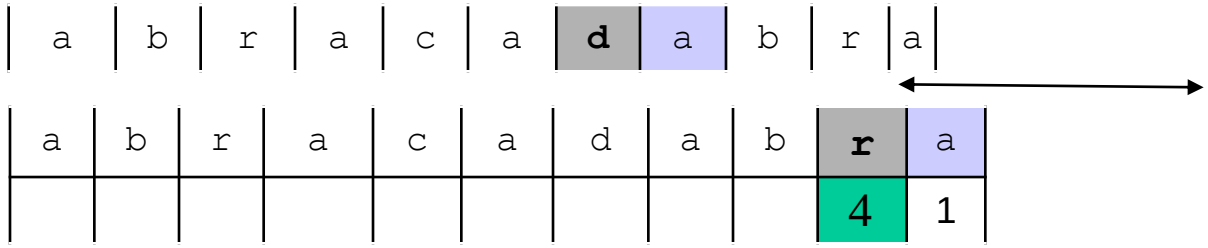
(α)

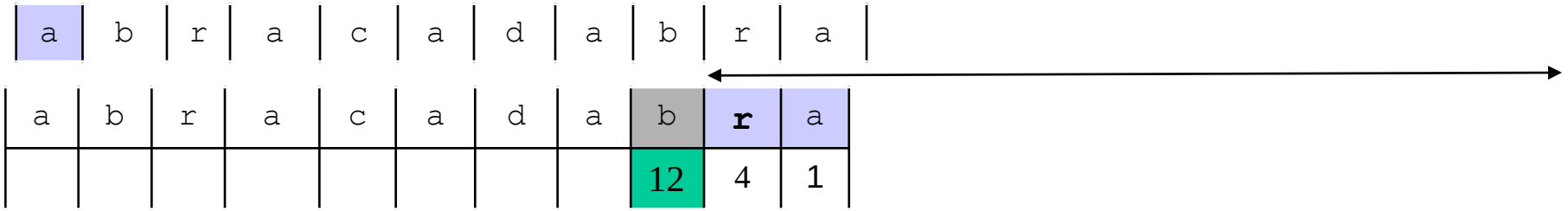


(β)

Ευρετικό Ταιριάσματος (συν.)

- Η υλοποίηση είναι πολύ δύσκολη! Σημειωτέον πως και οι ίδιοι οι ερευνητές είχαν δώσει λάθος κώδικα, τον οποίο διόρθωσε το 1980 ο Rytter!



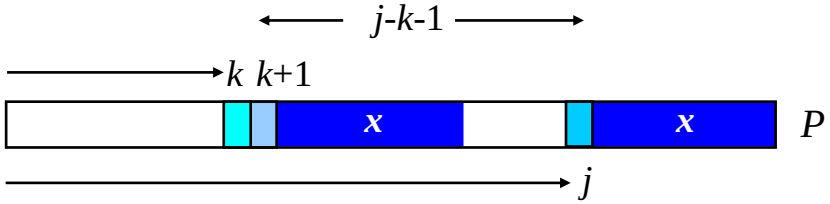


Ευρετικό Ταιριάσματος (συν.)

- Τελικώς…

a	b	r	a	c	a	d	a	b	r	a
17	**16**	**15**	**14**	**13**	**12**	**11**	**13**	**12**	**4**	**1**

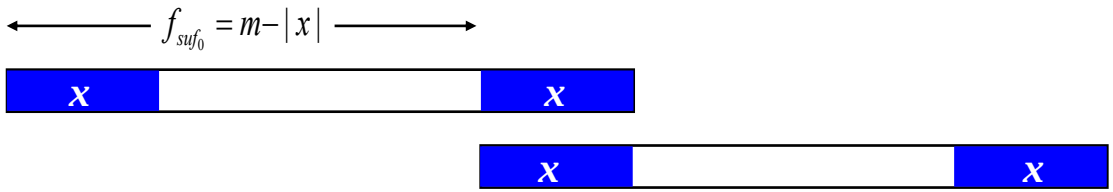
Μία Εξήγηση των Υπολογισμών



(α)

- Έστω ότι υπολογίζουμε «ανάποδα» τον πίνακα KMP, ώστε
  - Κάθε θέση k+1 μας δίδει την θέση j, όπου προηγείται του προσφύματος x
  - $P_j \neq P_{k+1}$, τότε βρήκαμε μία δεξιότερη εμφάνιση του x σε απόσταση j-k-1

Μία Εξήγηση των Υπολογισμών (συν.)



(β)

- Για την δεύτερη περίπτωση:
  - εάν *x* είναι το μεγαλύτερο πρόσφυμα που είναι και πρόθεμα, τότε για τις *m*-|*x*| πρώτες θέσεις η εικονιζόμενη τοποθέτηση είναι ασφαλής
  - Για τις υπόλοιπες του προσφύματος *x*, ο υπολογισμός είναι αναδρομικός...

Παράδειγμα

	0	1	2	3	4	5	6	7	8	9	10	11
		a	b	r	a	c	a	d	a	b	r	a
fsuf	7	8	9	10	11	10	11	10	11	11	11	12
match		12	12	12	12	12	12	12	12	12	**3**	**1**
match		**7**	**7**	**7**	**7**	**7**	**7**	**7**	**10**	**10**	3	1
+		10	9	8	7	6	5	4	3	2	1	0
match		**17**	**16**	**15**	**14**	**13**	**12**	**11**	**13**	**12**	**4**	**1**

- Η πολυπλοκότητα είναι ανάλογη του KMP…

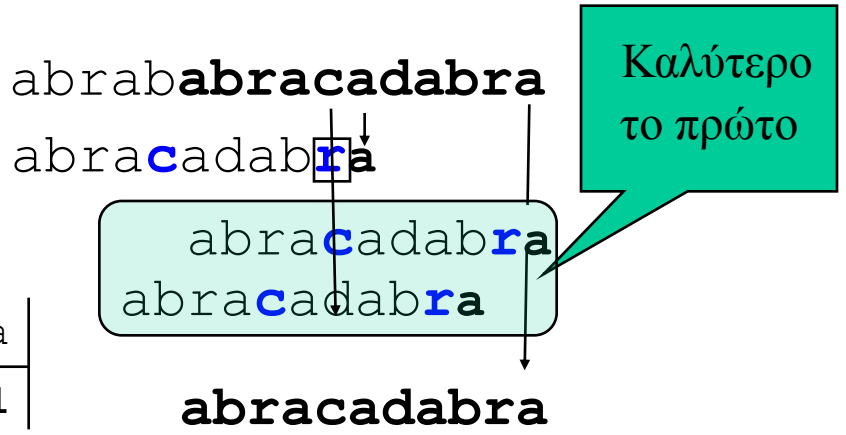
Παράδειγμα Τρεξίματος

d

a	b	c	d	r
0	**2**	**6**	**4**	**1**

m

a	b	r	a	c	a	d	a	b	r	a
17	**16**	**15**	**14**	**13**	**12**	**11**	**13**	**12**	**4**	**1**



- Την πρώτη φορά έχουμε ολίσθηση βάσει ευρετικού εμφανίσεως max{d[c]=6, m[r]=4}=6
- Την δεύτερη, πλήρες ταίριασμα

Πολυπλοκότητα

- Υπολογισμός ευρετικού εμφανίσεως $O(m+|\Sigma|)$ – μειονέκτημα η εξάρτηση από το αλφάβητο
- Υπολογισμός ευρετικού ταιριάσματος
- Συνολικά
  - Χειρότερη Περίπτωση

    $O(nm)$
  - Μέση Περίπτωση υπογραμμικός!

    $O(n\log m/m)$ -δηλαδή, αποφεύγει να διαβάσει όλο το κείμενο

Γενική Συζήτηση

- Στην πράξη, ο Knuth-Morris-Pratt εμφανίζει συμπεριφορά ελάχιστα καλύτερη του απλοϊκού αλγορίθμου
- Ο Boyer-Moore και οι παραλλαγές του είναι η μέθοδος που εφαρμόζεται στην πράξη καίτοι έχει (θεωρητικά) πολύ κακή χειρότερη συμπεριφορά

Ομοιότητα Συμβολοσειρών

- Υπάρχουν εφαρμογές που αιτούν πόσο όμοιες ή ανόμοιες είναι δύο συμβολοσειρές *x*, *y*. Π.χ.,
  - Όταν βρίσκουμε τα ορθογραφικά λάθη και προσπαθούμε να βρούμε την σωστή ορθογραφία
  - Προγράμματα-«αράχνες» που βλέπουν πόσο όμοιες είναι οι σελίδες με ήδη καταχωρημένες, ώστε να αποφασίσουν εάν θα τις καταχωρήσουν ή όχι
  - Ακολουθίες DNA, για τις οποίες ομοιότητα σημαίνει και δομική ή λειτουργική ομοιότητα

Μέγιστη Κοινή Υπακολουθία

- Ορισμός υπακολουθίας συμβολοσειράς

  x=αβγδ**2**τδ**λ**

  x'=**β2λ**, με i_1=2 < i_2=5 < i_3=8

  – Παρατηρήστε πως σεβόμαστε την από αριστερά προς τα δεξιά διάταξη των χαρακτήρων
  – Έτσι, η βαδ ***δεν είναι υπακολουθία***

Μέγιστη Κοινή Υπακολουθία

- Ορισμός

  Εάν x, y είναι δύο συμβολοσειρές, τότε η μέγιστη κοινή υπακολουθία τους ορίζεται ως η μεγαλύτερου μήκους υπακολουθία που εμφανίζεται και στις δύο.

  – Δεν είναι απαραίτητο να υπάρχει μόνο μία. Π.χ., οι

  αβελ **και** βαλε

  έχουν τις εξής μέγιστες, μήκους 2, κοινές υπακολουθίες:

  αλ, αε, βε, βλ

Απλοϊκή Λύση

- Δημιουργία όλων των υπακολουθιών των δύο συμβολοσειρών και σύγκριση μεταξύ τους.
- Κόστος

$$\sum_{i=1}^{\max\{n,m\}} i2^i = O(\max\{n,m\}2^{\max\{n,m\}}), \qquad n = |x|, m = |y|$$

Λύση Δυναμικού Προγραμματισμού

- Παρατήρηση

  Έστω $x = x[1..n]$, $y = y[1..m]$ οι συμβολοσειρές και $z = z[1..k]$ μία μέγιστη κοινή υπακολουθία τους. Τότε, ισχύει:

$$x[n] \begin{cases} = y[m] \Rightarrow\ z[k] = x[n],\ z[1..k\text{-}1] = \text{μκυ}(x[1..n\text{-}1], y[1..m\text{-}1]) \\ \neq y[m] \text{ και } z[k] \neq x[n],\ z[1..k] = \text{μκυ}(x[1..n\text{-}1], y[1..m]) \\ \neq y[m] \text{ και } z[k] \neq y[m],\ z[1..k] = \text{μκυ}(x[1..n], y[1..m\text{-}1]) \end{cases}$$

Λύση Δυναμικού Προγραμματισμού (συν.)

- Επομένως, εάν $c[i,j]$ είναι το μήκος της μέγιστης κοινής υπακολουθίας των $x[1..i]$, $y[1..j]$, τότε ισχύει:

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ ή } j = 0 \\ c[i\text{-}1, j\text{-}1] + 1 & x[i] = y[j] \\ \max\{c[i, j\text{-}1], c[i\text{-}1, j]\} & \text{διαφορετικά} \end{cases}$$

Ψευδοκώδικας

Algorithm largestCommonSubSeq(**char**[] x, **char**[] y)

```
1.  n = x.length; m = y.length;
2.  int[][] c = new int[n+1][m+1];
3.  int[][] aux = new int[n+1][m+1];
4.  for (i = 0; i < = n; i++){     // # O(n)
5.    c[i][0] = 0;
6.    aux[i][0] = NOT X;
7.  }
8.  for (i = 0; i < = m; i++){    // # O(m)
9.     c[0][i] = 0;
10.   aux[0][i] = NOT Y;
11. }
12. for (i = 1; i < = n; i++)      // # O(nm)
13.   for (j = 1; j <= m; j++){
14.     if (x[i] == y[j]) {          // ταίριασμα
15.       c[i][j] = c[i-1][j-1]+1;
16.       aux[i][j] = XY;
17.    }
18.     else if (c[i-1][j] >= c[i][j-1]) // το x[i] δεν ανήκει
18.       c[i][j] = c[i-1][j];
20.       aux[i][j] = NOT_X;
21.     }
22.     else {
19.       c[i][j] = c[i][j-1];              // το y[j] δεν ανήκει
20.       aux[i][j] = NOT_Y;
21.     }
22. }
23. return c[n][m];
```

Παράδειγμα

↑ : απόρριψη από ΒΑΒΕΛ

←: απόρριψη από ΑΒΕΛ

↖: ταίριασμα

		0	A 1	B 2	E 3	Λ 4
	0	0	←0	←0	←0	←0
B	1	↑0	↑0	↖1	←1	←1
A	2	↑0	↖1	←1	←1	←1
B	3	↑0	↑1	↖2	←2	←2
E	4	↑0	↑1	↑2	↖3	←3
Λ	5	↑0	↑1	↑2	↑3	↖4

Ελάχιστη Απόσταση Διαμορφώσεως

- Ορισμοί
  - Πράξεις διαμορφώσεως:
    - delete(*c*): διαγραφή του χαρακτήρα *c*
    - insert(*c*): εισαγωγή του χαρακτήρα *c*
    - replace(*c*,*k*): αντικατάσταση του χαρακτήρα *c* από τον *k*

Ελάχιστη Απόσταση Διαμορφώσεως (συν.)

- Πρόβλημα

  Δοθεισών δύο συμβολοσειρών x, y, ποια είναι η ελάχιστη ακολουθία πράξεων διαμορφώσεως που μετασχηματίζει την x σε y;

- Εφαρμογές
  - Αποθήκευση εκδοχών αρχείων αποθηκεύοντας *μόνο* το πλήθος των αλλαγών τους
  - Σύγκριση αρχείων
  - Σύγκριση ακολουθιών DNA

Παράδειγμα

- 3 πράξεις

  λεί**ε**πει ⇒ λίπ**ε**ι ⇒ λίπ**ι** ⇒ λίπη

  Διαγραφή Διαγραφή Αντικατάσταση

- 4 πράξεις

  λεί**ε**πει ⇒ λί**ί**πει ⇒ λίπ**ε**ι ⇒ λίπη**ι** ⇒λίπη

  Αντικατάσταση Διαγραφή Αντικατάσταση Διαγραφή

Παρατήρηση

- Έστω $c[i,j]$ είναι ο ελάχιστος αριθμός διαμορφώσεως των $x[1..i]$, $y[1..j]$. Τότε ισχύει:

$$c[i,j]=\min\begin{cases} c[i\text{-}1, j] + 1 & \text{del}(x[i]) & // \; x[1..i\text{-}1] \Rightarrow y[1..j] \\ c[i, j\text{-}1] + 1 & \text{ins}(y[j]) & // \; x[1..i] \Rightarrow y[1..j\text{-}1] \\ c[i\text{-}1, j\text{-}1] + 1 & \text{repl}(x[i], y[j]) & // \; x[1..i\text{-}1] \Rightarrow y[1..j\text{-}1] \\ c[i\text{-}1, j\text{-}1] & x[i]=y[j] & // \; x[1..i\text{-}1] \Rightarrow y[1..j\text{-}1] \end{cases}$$

Παράδειγμα

	–	Λ	Ι	Π	Η
–	0	1	2	3	4
Λ	1	0=	←1Ε	←2Ε	←3Ε
Ε	2	↑1Δ	↖1Α	↖2Α	↖3Α
Ι	3	↑2Δ	↖1=	↖2Α	↖3Α
Π	4	↑3Δ	↑ 2 Δ	↖1=	← 2Ε
Ω	5	↑4Δ	↑ 3 Δ	↑ 2Δ	↖2Α

Ψευδοκώδικας

Algorithm minEdtDis**(char**[] x, **char**[] y)

```
1.  n = x.length; m = y.length;
2.  for (i = 0; i <= n; i++)        // # O(n)
3.     c[i][0] = i;
4.  for (i = 0; i <= m; i++)        // # O(m)
5.      c[0][i] = i;
6.  for (i = 1; i <= n; i++)        // # O(nm)
7.     for (j = 1; j <= m; j++)
8.        c[i][j] = min(c[i-1][j]+1,c[i][j-1]+1,(x[i] == y[j] ? c[i-1][j-1] : c[i-1][j-]+1));
9.  return c[n][m];
```

- Πολυπλοκότητα $O(nm)$: ανάλογη, δηλαδή, του μεγέθους του πίνακα c

Αντεστραμμένα Αρχεία

- Αποτελούν την βασική δομή των search engines
- Βασίζονται στην ανάλυση του κειμένου σε λέξεις με:
  - μη διακριτική ικανότητα (άρθρα, σύνδεσμοι κ.ο.κ.), και
  - διακριτική ικανότητα
- Το σύνολο των κοινών, δίχως διακριτική ικανότητα, λέξεων αποτελεί την ***τερματική λίστα* (*stop list*)**
- Οι λέξεις με σημασιολογική σπουδαιότητα καλούνται ***όροι* (*terms*)** και αποτελούν το ***λεξικό***

Περιγραφή

- Με κάθε όρο του λεξικού σχετίζεται μία αντεστραμμένη λίστα με
  - τις εμφανίσεις του όρου –οι δείκτες συνήθως είναι το id του κειμένου–
  - συσχετιζόμενη βοηθητική πληροφορία, π.χ., συχνότητα εμφανίσεως
- Οι όροι είναι ταξινομημένοι κατά αλφαβητική σειρά
- Το μέγεθος του λεξικού είναι ΥΠΟΓΡΑΜΜΙΚΟ $O(n^{\varepsilon})$ στο πλήθος n των λέξεων (νόμος Heaps)

Παράδειγμα

1

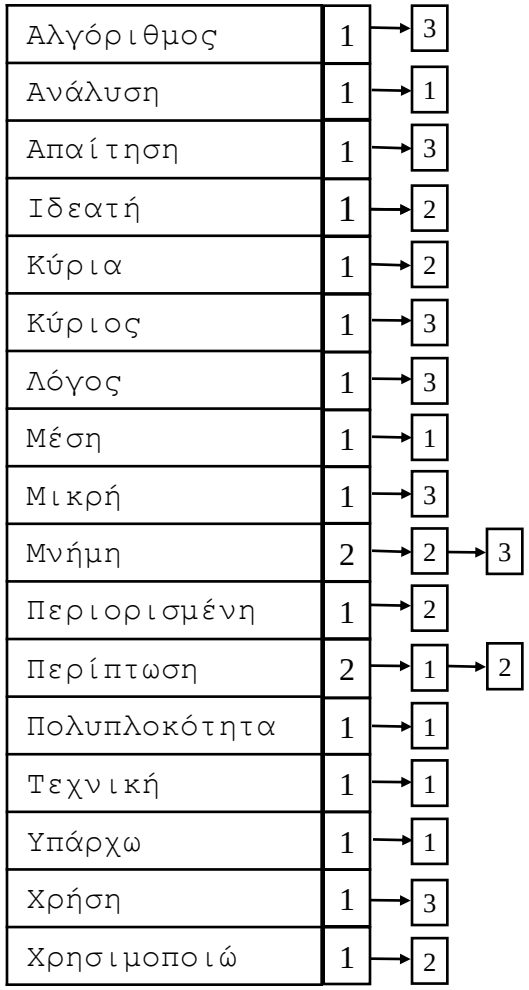
Υπάρχουν δύο τεχνικές ανάλυσης πολυπλοκότητας: η ανάλυση χειρότερης περιπτώσεως και η ανάλυση μέσης περιπτώσεως.

2

Σε περίπτωση που η κύρια μνήμη είναι περιορισμένη, χρησιμοποιείται η ιδεατή μνήμη.

3

Κύριος λόγος χρήσεως του αλγορίθμου αυτού είναι οι μικρές απαιτήσεις σε μνήμη.

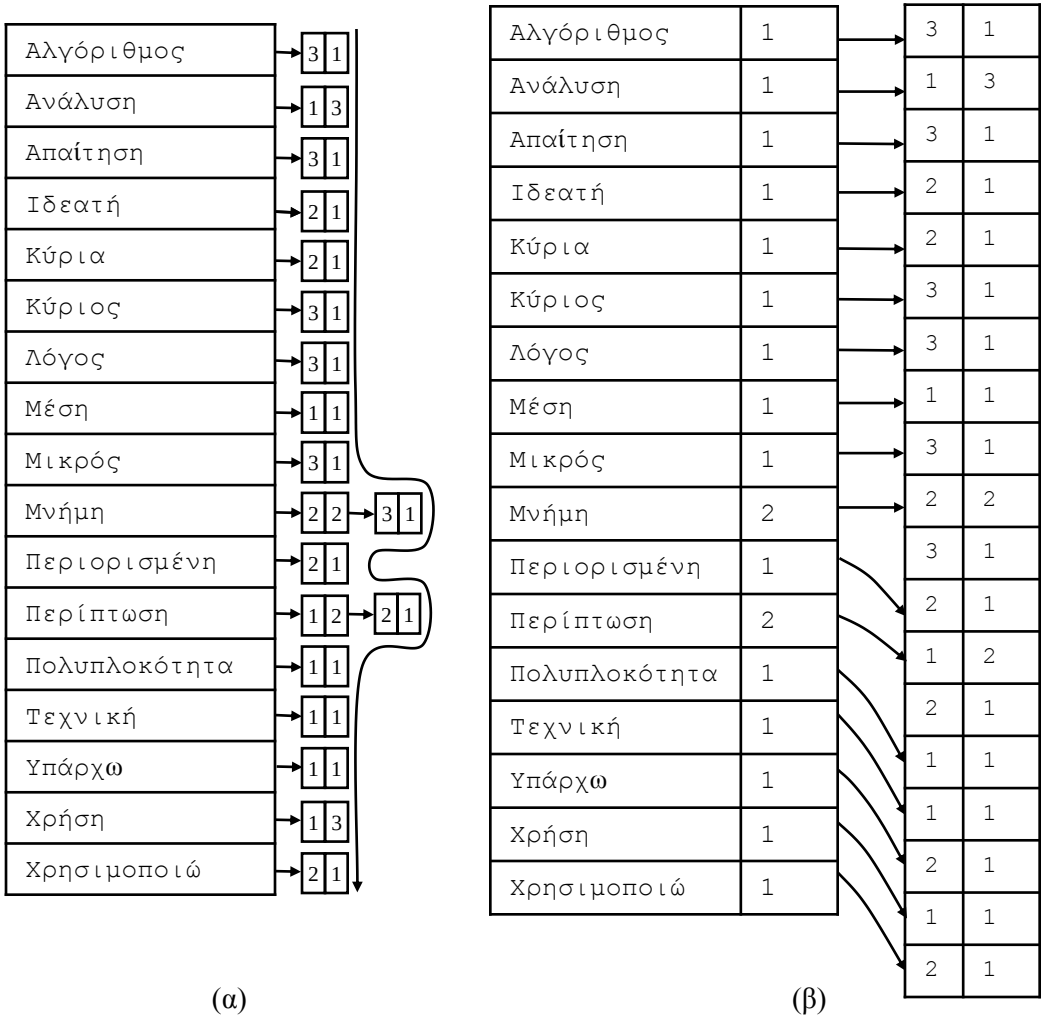


Πώς χρησιμεύει

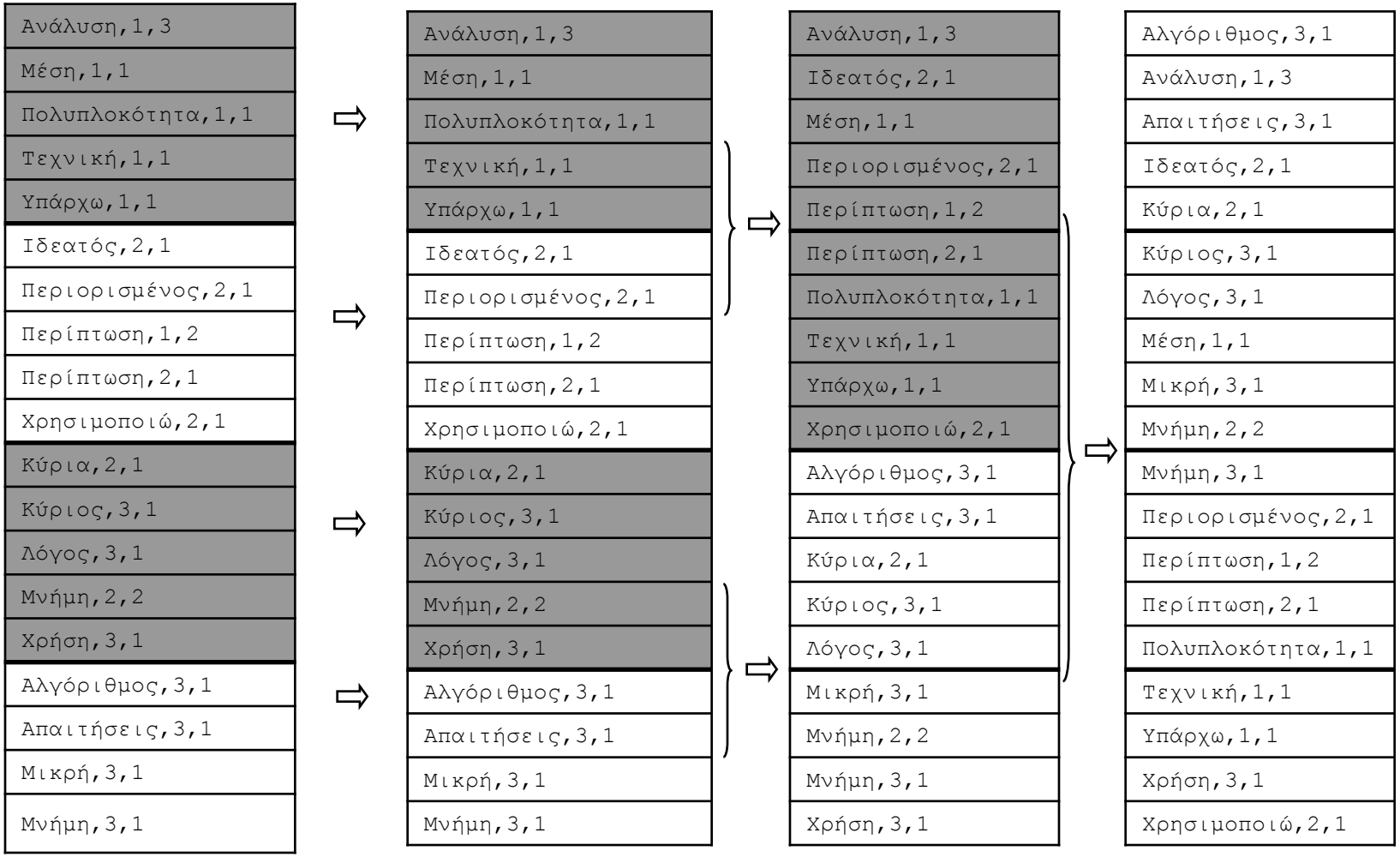
- Μία ερώτηση με ‘και’ (βρες μου όλα τα έγγραφα που περιέχουν ‘πληροφορική’ και ‘πολυτεχνείο’) αντιστοιχεί σε τομή των αντίστοιχων λιστών
- Μία ερώτηση με ‘ή’ (βρες μου όλα τα έγγραφα που περιέχουν ‘πληροφορική’ ή ‘πολυτεχνείο’) αντιστοιχεί σε ένωση των αντίστοιχων λιστών
- Μία ερώτηση με άρνηση (βρες μου όλα τα έγγραφα που δεν περιέχουν την λέξη ‘πληροφορική’) αντιστοιχεί σε αποκλεισμό της αντίστοιχης λίστας
- Μεικτές ερωτήσεις με ‘και’ , ‘ή’ και ‘όχι’ αναλύονται σε αντίστοιχες πράξεις ενώσεως, τομής και συμπλήρωσης (άρνησης) συνόλων με την μορφή λιστών

Δημιουργία

- 1ος τρόπος: στην κύρια μνήμη



- 2ος τρόπος: Δημιουργία στον σκληρό δίσκο με merge sort!



Κωδικοί Huffman

- Συμπίεση Κειμένου

  Η διαδικασία αλλαγής της αναπαραστάσεως ώστε

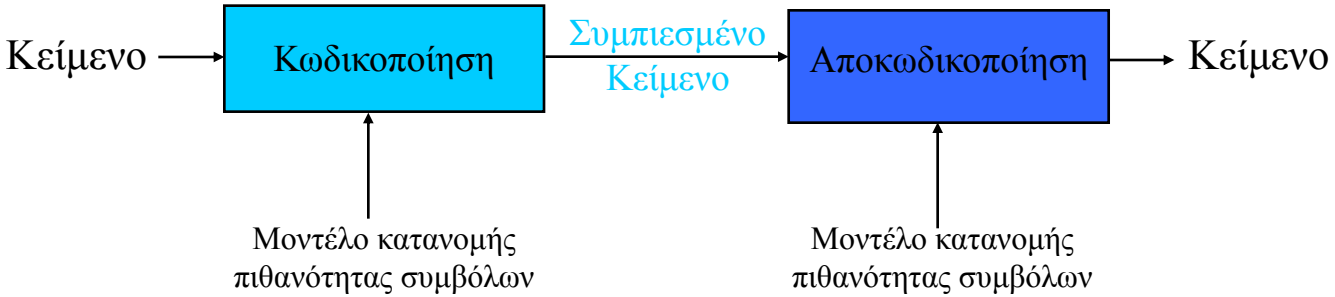
  (α) να καταλαμβάνει μικρότερο χώρο ή να χρειάζεται λιγότερο χρόνο μεταδόσεως και

  (β) να είναι δυνατή η ***πλήρης*** αποκατάσταση της αρχικής μορφής του κειμένου

- Κρίσιμη Παρατήρηση:

  Εν αντιθέσει με την εικόνα ή τον ήχο, όπου τυχόν απώλειες (θόρυβος) είναι αποδεκτές, στα κείμενα ***δεν είναι ανεκτές οι απώλειες***

Γενική Διαδικασία Κωδικοποιήσεως - Αποκωδικοποιήσεως



- Η ανάθεση κωδικών στους χαρακτήρες γίνεται βάσει του πιθανοτικού μοντέλου που διέπει την εμφάνιση των χαρακτήρων του κειμένου

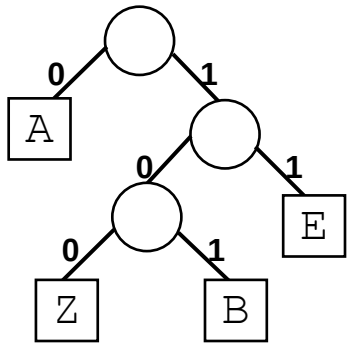
Μέθοδοι Κωδικοποίησης

- **Huffman**
  - Η κύρια μέθοδος μέχρι τα τέλη του ΄70. Την ανακάλυψε ο Huffman στα πλαίσια μίας εργασίας σε μάθημα κατά την διάρκεια των σπουδών του (!)
  - Πετυχαίνει συμπίεση 5 μπιτ ανά χαρακτήρα.
  - Την χρησιμοποιεί η εντολή pack στο unix.
  - Μειονέκτημα ότι πρέπει πρώτα να σαρωθεί όλο το κείμενο.
- **Ziv-Lempel**
  - Συμπιέζει ‘on-the-fly’ βρίσκοντας επαναλαμβανόμενα patterns.
  - Πετυχαίνει 4 μπιτ ανά χαρακτήρα.
  - Σε αυτήν στηρίζονται τα gzip και compress.
- **Αριθμητικοί Μέθοδοι**
  - Συνήθως προτιμούνται για μετάδοση δεδομένων, καθώς πετυχαίνουν κάτι ελάχιστα παραπάνω από 2 μπιτ ανά χαρακτήρα.

Κωδικοί Προθέματος (Prefix Codes)

- **Ορισμός**

  Δυαδικές συμβολοσειρές (bit strings) με την ***κρίσιμη*** ιδιότητα ***καμμία να μην αποτελεί πρόθεμα της άλλης.***
- Αναπαρίστανται μέσω Tries, οπότε είναι εύκολη η αποκωδικοποίηση/κωδικοποίηση με διέλευση του δένδρου. Π.χ.,



Κωδικοί Προθέματος (συν.)

- Κόστος δένδρου

$$E(T)=\sum_{c\in T}\mathbf{Prob}[c]\text{len}(c)$$

Άρα ζητούμενο η εύρεση δένδρου T που ελαχιστοποιεί το κόστος. Γιατί;

$$E(\textit{κειμένου } K) = \sum_{x\in K}\mathbf{Prob}[x]\text{len}(x)$$

$$= |K| \sum_{c\in C}\mathbf{Prob}[c]\text{len}(c),$$

όπου C *το σύνολο των διακεκριμένων χαρακτήρων*

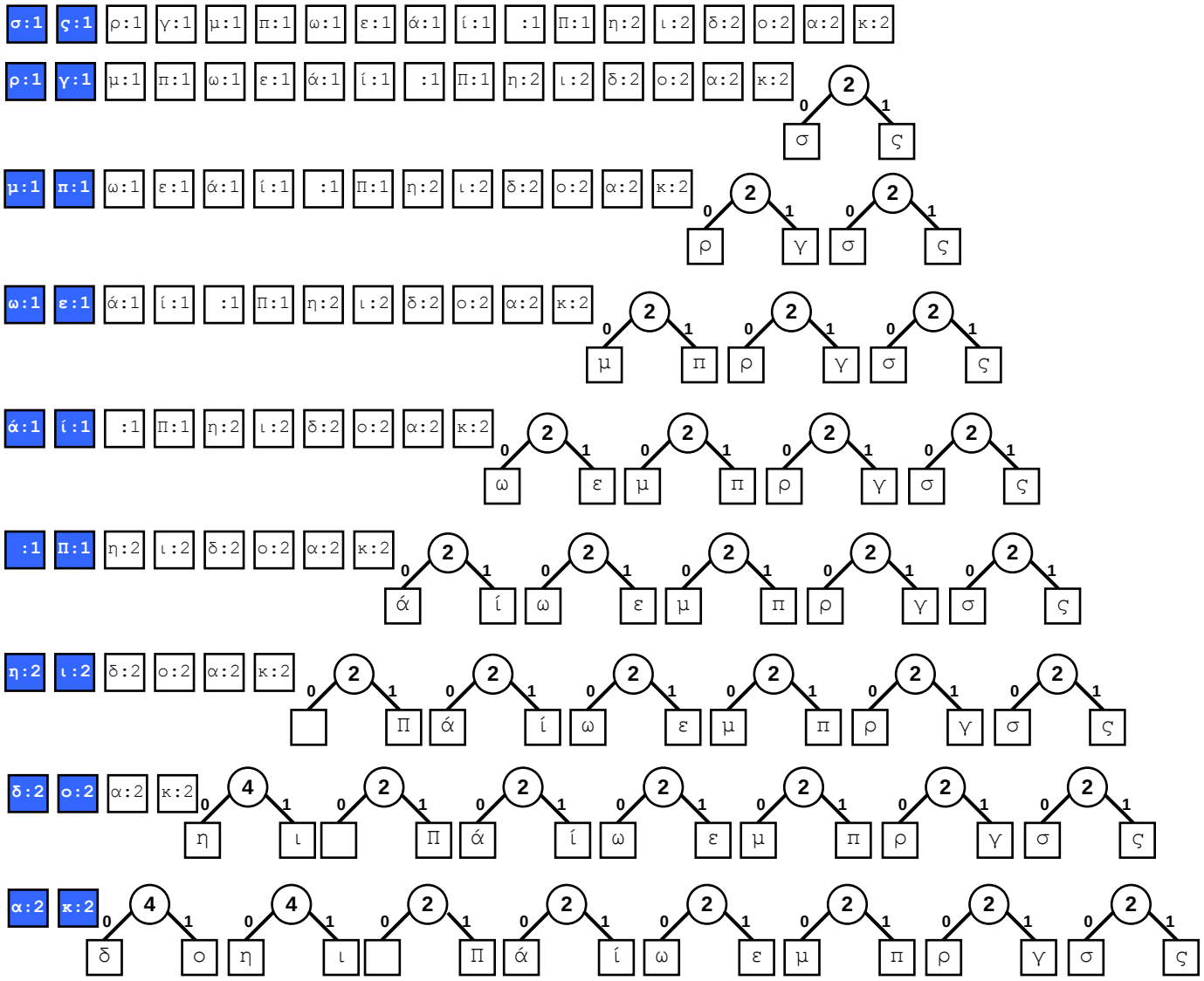
Κεντρική Ιδέα

- Στους χαρακτήρες που έχουν μεγάλη συχνότητα εμφανίσεως (άρα, έχουν μικρή διακριτική αξία) να δοθούν μικρού μήκους κωδικοί, ενώ στους σπάνιους (με μεγάλη σημασιολογική αξία) να δοθούν μεγαλύτερου μήκους κωδικοί.
- Ο Huffman ανακάλυψε έναν ***απλό, βέλτιστο και άπληστο*** αλγόριθμο κατασκευής κωδικών

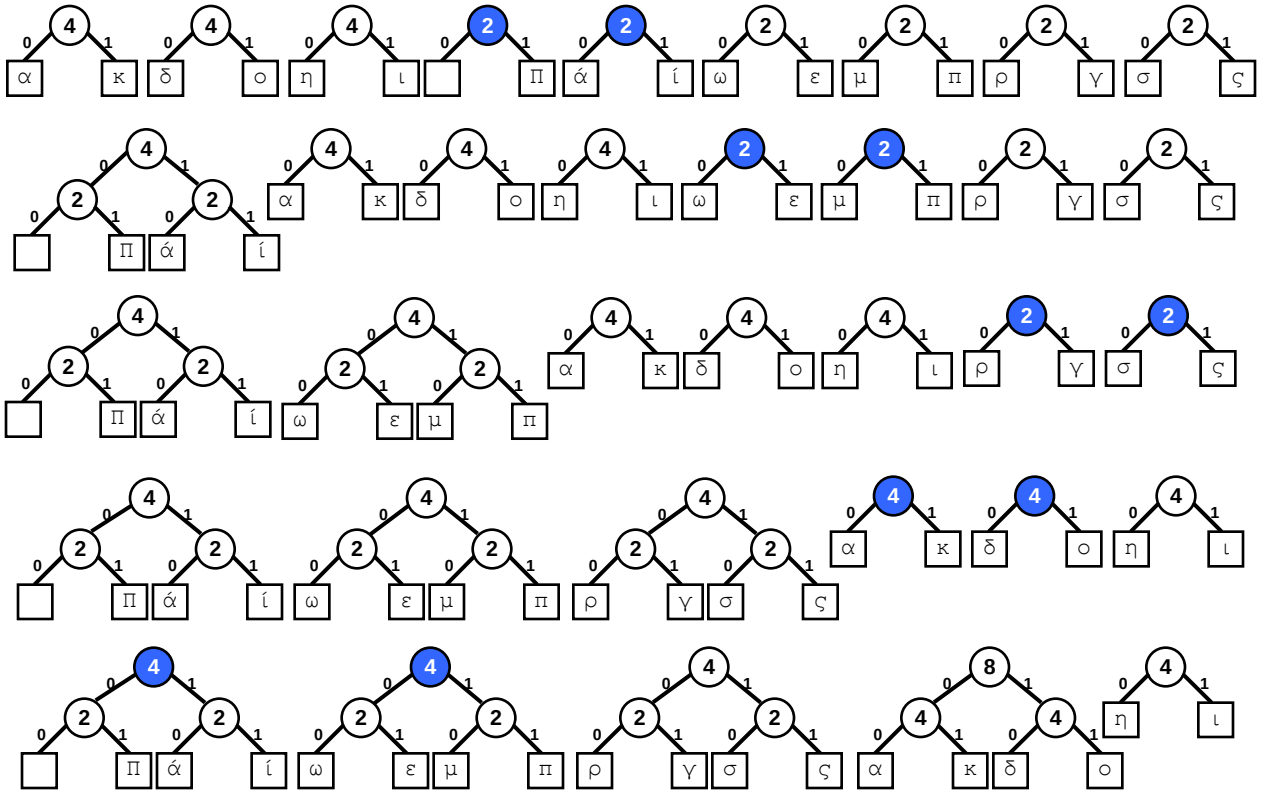
Περιγραφή

- Το αρχικό πρόβλημα κωδικοποιήσεως n χαρακτήρων ανάγεται στην αναδρομική επίλυση ενός προβλήματος n-1 χαρακτήρων. Πώς;
  - Με την επιλογή των δύο χαρακτήρων c_i,c_j μικρότερης συχνότητας f_i, f_j, αντίστοιχα, και την αντικατάστασή τους με έναν τεχνητό χαρακτήρα, συχνότητας f_i+f_j

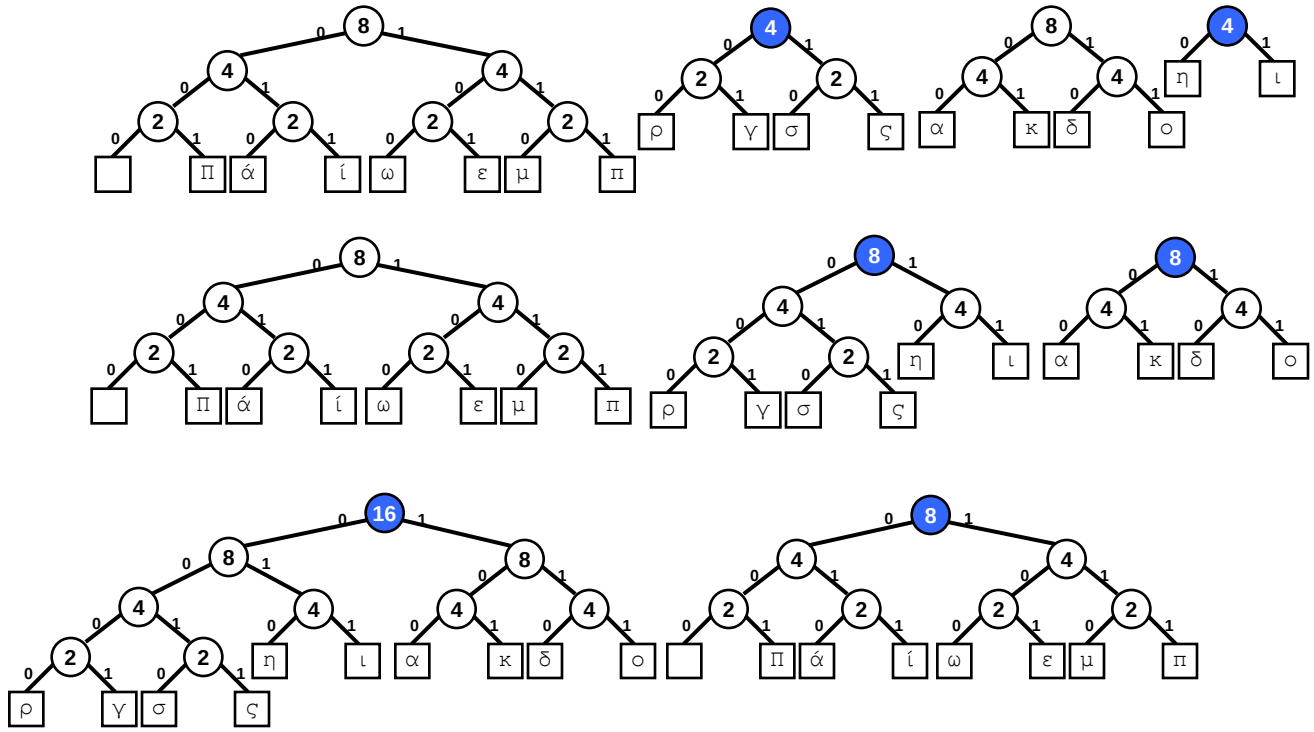
Παράδειγμα


σ:1 ς:1 ρ:1 γ:1 μ:1 π:1 ω:1 ε:1 ά:1 ί:1 :1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
ρ:1 γ:1 μ:1 π:1 ω:1 ε:1 ά:1 ί:1 :1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
μ:1 π:1 ω:1 ε:1 ά:1 ί:1 :1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
ω:1 ε:1 ά:1 ί:1 :1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
ά:1 ί:1 :1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
:1 Π:1 η:2 ι:2 δ:2 ο:2 α:2 κ:2
η:2 ι:2 δ:2 ο:2 α:2 κ:2
δ:2 ο:2 α:2 κ:2
α:2 κ:2

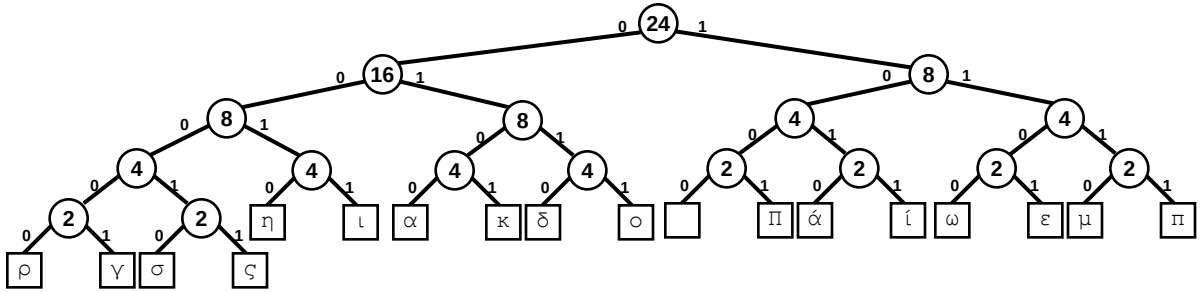
Παράδειγμα (συν.)



Παράδειγμα (συν.)



Παράδειγμα (συν.)



Παράδειγμα κωδικοποιήσεως

ASCII

11010000**11100001**11110001**11011100**11100100
1110010111101001**11100011**11101100**11100001**
10100000**11101010**11111001**11100101**11101001
1110101011101111**11110000**11101111**11011111**
11100111**11110011**11100111**11110010**

Huffman

1001**0100**00000**1010**0110**1101**0011**00001**111**01**
001000**0101**1100**0110**0011**0101**0111**1111**011**10**
110010**00010**0010**00011**

Υλοποίηση

Algorithm huffmanCoding(**char**[] C, **int**[] F)

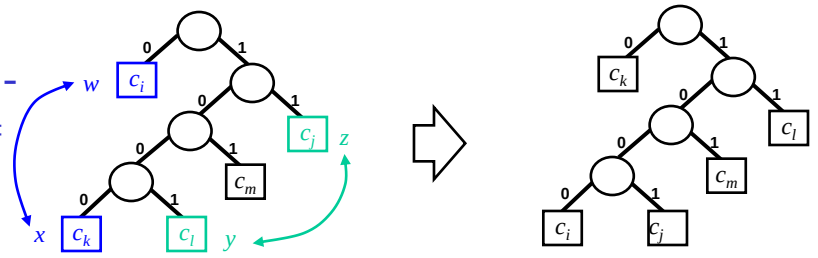
```
1.  pqueue pq = new pqueue();      //ουρά προτεραιότητας
2.  for (i = 0; i < C.length; i++)    // αρχικοποίηση με τα φύλλα – χρόνος O(nlogn)
3.     pq.insKey(new bNode(new Item(C[i],F[i]), F[i]); // χρόνος O(logn)
4.  for (i = 1; i< C.length; i++){   // χρόνος O(nlogn)
5.     bNode t = new bNode();       // νέος εσωτερικός κόμβος
6.      x = pq.delMin();            // χρόνος O(logn)
7.      y = pq.delMin();            // χρόνος O(logn)
8.      t.setLeftSon(x);
9.      t.setRightSon(y);
10.    t.setItem(new Item(O, x.Key+y.Key))  // με στοιχείο τον τεχνητό χαρακτήρα O
11.    qp.insKey(t, t.key);          // και συχνότητα το άθροισμά τους – χρόνος O(logn)
12. }
13. return pq.delMin();             // ρίζα του δένδρου ο εναπομείναν κόμβος
```

- **Πολυπλοκότητα:** $O(n\log n)$ με χρήση δυωνυμικής ουράς

Απόδειξη ορθότητας

- Η άπληστη επιλογή οδηγεί σε βέλτιστη λύση· ήτοι, υπάρχει δένδρο κωδικών με τους σπανιότερους χαρακτήρες να είναι αδέλφια

$-\mathbf{Prob}[c_k]d_x+\mathbf{Prob}[c_k]d_w - \mathbf{Prob}[c_i]d_w+d_x\mathbf{Prob}[c_i] = -\{\mathbf{Prob}[c_k](d_x-d_w)- \mathbf{Prob}[c_i](d_x-d_w)\} \leq 0$



$-\{\mathbf{Prob}[c_l](d_y-d_z)- \mathbf{Prob}[c_j](d_y-d_z)\} \leq 0$

  - Η αλλαγή δίδει δένδρο με μικρότερο ή ίσο κόστος με το παλιό καθώς $\mathbf{Prob}[c_i] \leq \mathbf{Prob}[c_k]$ και $\mathbf{Prob}[c_j] \leq \mathbf{Prob}[c_l]$

Απόδειξη ορθότητας (συν.)

- Το δένδρο που προκύπτει συνδυάζοντας την λύση για τους n-1 χαρακτήρες είναι βέλτιστο.

  Με επαγωγή στο πλήθος n των χαρακτήρων:

  - Με την αντικατάσταση, προκύπτει ένα στιγμιότυπο n-1 χαρακτήρων, όπου ο τεχνητός χαρακτήρας x έχει βάθος d_x, βέλτιστου -βάσει επαγωγής- κόστους $E(T_{n-1})$
  - Με την τοποθέτηση των δύο χαρακτήρων σε βάθος d_x+1, προκύπτει δένδρο κόστους:

    $$E(T_n)=E(T_{n-1})+\mathbf{Prob}[c_i]+\mathbf{Prob}[c_j]$$

  - Αυτό είναι βέλτιστο, γιατί:
    - Εάν δεν ήταν, τότε υπάρχει κάποιο άλλο με καλύτερο κόστος, όπου τα c_i, c_j έχουν μέγιστο βάθος.
    - Αντικαθιστώντας τα με ένα τεχνητό χαρακτήρα, προκύπτει στιγμιότυπο για n-1 χαρακτήρες με καλύτερη λύση (άτοπο)