

# Algorismes de text

**Algorísmica**

Grau d'Enginyeria Informàtica

Mireia Ribera, [ribera@ub.edu](mailto:ribera@ub.edu)

Daniel Ortiz, [daniel.ortiz@ub.edu](mailto:daniel.ortiz@ub.edu)

# Algorismes de text: Conceptes clau

- 1 El cost del treball amb dades textuales
- 2 Cercar un text
- 3 Cercar un text aproximat
- 4 Levenshtein

# El cost de treballar amb cadenes de text

Quan treballem amb cadenes de text, qualsevol operació es complica perquè hem de fer el tractament caràcter a caràcter.

Veiem per exemple el següent codi:

```
def is_unique(paraula):
    caracters_vistos = []
    for car in paraula:
        if car in caracters_vistos:
            return False
        caracters_vistos.append(car)
    return True
```

# El cost de treballar amb cadenes de text (II)

```
def is_unique(paraula):
    caracters_vistos = [] # 1 pas
    for car in paraula: # per n, on n = nombre caràcters a paraula
        if car in caracters_vistos: # in té un cost m, m = len (caracters_vistos)
            return False # 1 pas
        caracters_vistos.append(car) # 1 pas
    return True
```

Caràcters \_vistos tindrà 1 caràcter el primer cop, 2 el segon, 3... fins a n-1 caràcters en el pitjor cas. El cost d'iterar sobre aquest in és una sèrie aritmètica:

$$\sum_{i=1}^q i = \frac{q(q + 1)}{2} \approx q^2$$

El cost és  $1 + n^2 + n + 1 \Rightarrow O(n^2)$

# El cost de treballar amb cadenes de text (III)

Un petit algorisme com el que s'ha vist ja té un cost  $O(n^2)$  on n és el nombre de caràcters del text a processar.

Si hem de processar una pàgina web o un llibre, els costos són molt grans!!

# Cercar un text

Un dels algorismes de text d'ús més generalitzat és buscar una paraula dins un text gran. Aquest algorisme es fa servir en:

- Editors de text
- Bioinformàtica
- Cercadors d'internet
- Bases de dades
- Antivirus
- LLMs

# Cercar un text: definició del problema

Tenim una cadena  $P$  (patró) de  $m$  caràcters (la que volem trobar) i una cadena  $T$  (*text*) de  $n$  caràcters (en la que busquem el patró), amb  $n > m$ .

Per exemple:

P: 001011            T: 10010101101001100101111010

P: happy            T: It is never too late to have a happy childhood.

P: GATTACAC        T: ATCGGATATCCGGAAACTGGTAGCGTGTAGGAGGTAGCCTGGAAG

# Cercar un text: solució ingènua

Una solució ingènua al problema seria anar buscant el patró d'esquerra a dreta a cada posició:

```
10010100010111101001001101  
001011.....  
..001011.....  
..001011.....  
...001011.....  
....001011.....  
.....001011.....  
.....001011.....  
.....001011
```

# Cercar un text: solució ingènua. Codi

1. Alineem el patró a l'inici del text
2. Ens movem d'esquerra a dreta comparant cada caràcter del patró amb el caràcter corresponent del text. Repetim fins que tots els caràcters fan correspondència o trobem diferència.
3. Si hem trobat diferència anem al pas 2, realineant patró una posició a la dreta. Fins que no s'acabi el text.

```
def CercaTextForcaBruta(t:str,p:str) -> int: # coneぐut com BFStringMatching
    m:int = len(p)
    n:int = len(t)
    for i in range(n-m+1):                      # i és la posició inicial del patró
        j:int = 0
        while j < m and p[j] == t[i+j]:           # j són els caràcters que coincideixen
            j += 1
        if j == m:
            return i
    return -1
```

# Cercar un text: solució ingènua. Complexitat codi

```
def CercaTextForcaBruta(t:str,p:str) -> int:  
    m:int = len(p)                                # 1 pas  
    n:int = len(t)                                # 1 pas  
    for i in range(n-m+1):                         # el de dintre per n-m vegades  
        j:int = 0                                    # 1 pas  
        while j < m and p[j] == t[i+j]:            # en el cas pitjor, el de dintre per m vegades  
            j += 1                                    # 2 passos  
        if j == m:                                  # 1 pas  
            return i                                # 1 pas (només s'executarà un cop)  
    return -1                                      # 1 pas (s'executarà aquest o l'anterior)
```

# cost:  $2 + (n-m)*(1 + (m-1)*2 + 1) + 1 \Rightarrow$  si només tenim en compte els components de més pes  $O(n*m)$

## Cercar un text. Complexitat codi. Cas millor i cas mig

A la diapositiva anterior hem vist el cas pitjor. Però quin seria el cas millor?

I pots endevinar quina és la complexitat en textos reals de llenguatge natural?

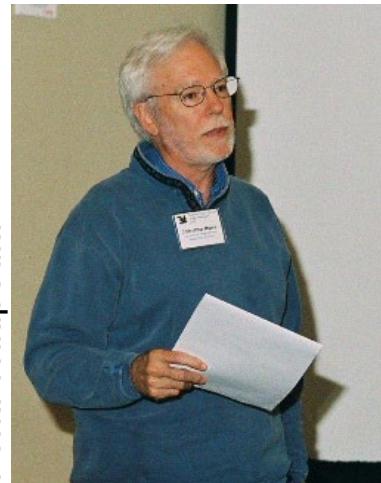
# Cercar un text: Boyer-Moore-Horspool

Font: Universitat de Texas



Robert Boyer

Font: Wikipedia



J. Strother Moore

Robert S. Boyer i J. Strother Moore, dos matemàtics d'Estats Units, l'any 1977 van desenvolupar un algorisme, l'algorisme Boyer-Moore per cercar cadenes que millora una mica la versió ingènua gràcies a preprocessar l'entrada.

Nigel Horspool, un informàtic d'Estats Units, va fer una versió millorada d'aquest algorisme, que tot i tenir el mateix ordre de complexitat, simplifica l'algorisme original de Boyer-Moore.



Nigel Horspool

Font: Universitat de Victòria

# Cercar un text: Boyer-Moore-Horspool

En aquest algorisme comencem pel final del patró i si és diferent a la lletra del text, desplaçem el patró a la dreta tantes posicions com faci falta fins que ambdues lletres coincideixin..

JIM.SAW.ME.O.IN.A.BARBERSHOP

BARBER

..... BARBER

..... BARBER

..... BARBER

..... BARBER

..... BARBER

# Cercar un text: Boyer-Moore-Horspool

Per a poder fer els salts en el patró (de  $m$  caràcters) hem de tenir una taula en la que, donada una lletra, ens indiqui quin salt hem de fer ( $m$  si no és al patró, la mínima distància al darrer caràcter altrament)

Per exemple, si el patró és BAOBAB, la **taula de desplaçaments** seria la següent:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6

Si el patró és BARBER, la **taula de desplaçaments** seria la següent:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	X	Y	Z
4	2	6	6	1	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6

# Cercar un text: Horspool. Codi

```
def BoyerMooreHorspool(patro:str, text:str)->int:
    m:int = len(patro)
    n:int = len(text)
    if m > n:
        return -1
    taula_desplacaments = []
    for k in range(256):
        taula_desplacaments.append(m)
    for k in range(m - 1):
        taula_desplacaments[ord(patro[k])] = m - k - 1
    taula_desplacaments = tuple(taula_desplacaments)
    k = m - 1
    while k < n:
        j = m - 1
        i = k
        while j >= 0 and text[i] == patro[j]:
            j -= 1
            i -= 1
        if j == -1:
            return i + 1
        k += taula_desplacaments[ord(text[k])]
    return -1
```

# Cercar un text: Horspool. Complexitat codi

A la complexitat de l'algorisme ingenu se li afegeix el cost de generar la taula de desplaçaments.

I en el cas pitjor l'algorisme té el mateix cost  $O(n*m)$ .

El cas promig és d' $O(n)$ , però és més eficient que el codi ingenu.

# Cerca aproximada d'un text: definició del problema

Tenim una cadena  $P$  (patró) de  $m$  caràcters (la que volem trobar) i una cadena  $T$  (*text*) de  $n$  caràcters (en la que busquem el patró), amb  $n > m$ .

El que busquem ara no és exactament el patró sinó la **subcadena de  $T$**  que s'hi assembla més, és a dir, **amb la distància d'edició menor**.

Per exemple:

P: BERBER	T: JIM SAW ME IN A BARBERSHOP	Té una distància d'1
P: B.RBAR	T: JIM SAW ME IN A BARBERSHOP	Té una distància de 2
P: VARVAR	T: JIM SAW ME IN A BARBERSHOP	Té una distància de 3

El primer que veurem és un **algoritme per calcular la distància entre el patró i el text**.

# Distància d'edició entre dues paraules: Algorisme de Levenshtein



Font: Wikipedia

Vladimir I. Levenshtein

Vladimir Iosifovich Levenshtein, un científic rus que feia recerca sobre computació i combinatòria va desenvolupar aquest algoritme l'any 1965.

# Distància d'edició: conceptes preliminars

Si volem **calcular la distància entre el patró i el text**, hem de veure quants canvis hem de fer al patró per a que sigui igual que el text. Considerarem 3 operacions, cadascuna de cost 1:

- Inserció (I): afegir una lletra al patró
- Deletion (D): treure una lletra al patró
- Substitució (S): canviar una lletra al patró.

Considerem també la possibilitat de que dos caràcters coincideixin (C), i llavors el cost és 0

Per exemple:

P: BERBER T: **BARBER**      Transformar BERBER a BARBER comporta 1 canvi, distància 1. Substituïm E per A

P: B.RBAR T: **BARBER**      Transformar B.RBAR a BARBER comporta 2 canvis, distància 2.

Afegim A al patró. Substituïm A per E

P: VARVAR T: **BARBER**      Transformar VARVAR a BARBER comporta 3 canvis, distància 3.

Substituïm V->B, V->B, A->E

# Distància d'edició: procediment per calcular-la

	text	M	E	I	L	E	N	S	T	E	I	N
patró	0	1	2	3	4	5	6	7	8	9	10	11
L	1	1	2	3	3	4	5	6	7	8	9	10
E	2	2	1	2	3	3	4	5	6	7	8	9
V	3	3	2	2	3	4	4	5	6	7	8	9
E	4	4	3	3	3	3	4	5	6	6	7	8
N	5	5	4	4	4	4	3	4	5	6	7	7
S	6	6	5	5	5	5	4	3	4	5	6	7
H	7	7	6	6	6	6	5	4	4	5	6	7
T	8	8	7	7	7	7	6	5	4	5	6	7
E	9	9	8	8	8	7	7	6	5	4	5	6
I	10	10	9	8	9	8	8	7	6	5	4	5
N	11	11	10	9	9	9	8	8	7	6	5	4

Per calcular la distància anirem calculant la distància de cada part del text amb cada part del patró, començant pel cas trivial i avançant pas a pas.

Quan tinguem tot el text i tot el patró, ja tindrem la distància final. En el dibuix, **4!**

# Distància d'edició: procediment per calcular-la

text	M	E	I	L	E	N	S	T	E	I	N	
patró	0	1	2	3	4	5	6	7	8	9	10	11

Anem omplint una matriu, si el patró fos buit la distància seria inserir una lletra per cada lletra del text.

# Distància d'edició: procediment per calcular-la

	text	M	E	I	L	E	N	S	T	E	I	N
patró	0											
L	1											
E	2											
V	3											
E	4											
N	5											
S	6											
H	7											
T	8											
E	9											
I	10											
N	11											

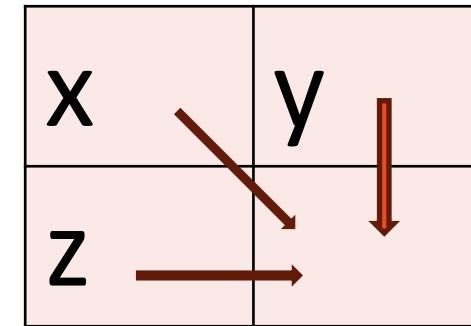
Si el text fos buit però el patró no, la distància seria esborrar cadascuna de les lletres del patró.

# Distància d'edició: procediment per calcular-la

T

text	M	E	I	L	E	N	S	T	E	I	N	
patró	0	1	2	3	4	5	6	7	8	9	10	11
L	1	1,1										
E	2											
V	3											
E	4											
N	5											
S	6											
H	7											
T	8											
E	9											
I	10											
N	11											

P

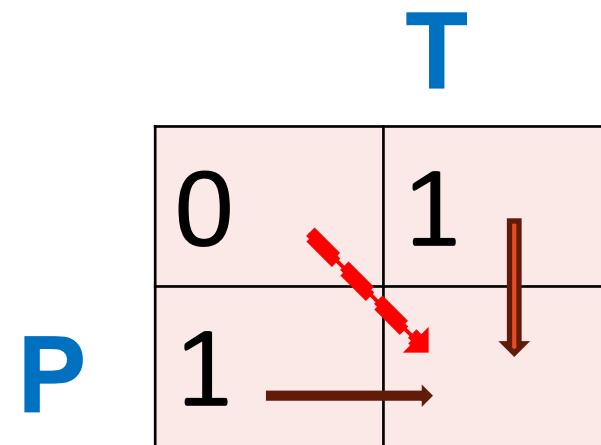


I per les altres caselles avancem pel camí òptim, determinats pel nou caràcter en el text (a la imatge "M") i pel nou caràcter en el patró (a la imatge "L"). I per les caselles del voltant.

Segons el cas ens tocarà: inserir un caràcter a P (avançar a la dreta), esborrar (delete) un caràcter a P (avançar avall) o coincidir o substituir un caràcter a P (avançar en diagonal). Quin triem? El que correspon als caràcters i que té el menor cost! El cost és el de la casella d'on partim + 1 (o en el cas de coincidència, + 0)

# Distància d'edició: procediment per calcular-la

text	M	E	I	L	E	N	S	T	E	I	N	
patró	0	1	2	3	4	5	6	7	8	9	10	11
L	1	1										
E	2											
V	3											
E	4											
N	5											
S	6											
H	7											
T	8											
E	9											
I	10											
N	11											



En el cas de la imatge la única opció és substituir la L per la M. Per tant avancem en diagonal.

El cost és el de la casella precedent (0) + 1.

# Distància d'edició: procediment per calcular-la

D	↓	Deletion, eliminem en el patró, ens desplaçem avall											
I	→	Insertion, inserim al patró, ens desplaçem a la dreta											
	text	M	E	I	L	E	N	S	T	E	I	N	
patró	0	1	2	3	4	5	6	7	8	9	10	11	
L	1	1	2	3	3	4	5	6	7	8	9	10	
E	2	2	1	2	3	3	4	5	6	7	8	9	
V	3	3	2	2	3	4	4	5	6	7	8	9	
E	4	4	3	3	3	3	4	5	6	6	7	8	
N	5	5	4	4	4	4	3	4	5	6	7	7	
S	6	6	5	5	5	5	4	3	4	5	6	7	
H	7	7	6	6	6	6	5	4	4	5	6	7	
T	8	8	7	7	7	6	5	4	4	5	6	7	
E	9	9	8	8	8	7	7	6	5	4	5	6	
I	10	10	9	8	9	8	8	7	6	5	4	5	
N	11	11	10	9	9	9	8	8	7	6	5	4	

Les operacions realitzades han estat:

S	
C	
S	I
C	
C	
C	
D	
C	
C	
C	
C	
C	

Clau:

D – Deletion

I – Insertion

S – Substitution

C – Coincidence

S i C són un moviment avall i a la dreta (en diagonal).

# Distància d'edició: codi

```
def distancia_levenshtein(patro, text):
    if len(text) == 0:
        return len(patro)
    llargada_patro = len(patro) + 1
    llargada_text = len(text) + 1
    matriu_distancies = [[0] * llargada_text for x in range(llargada_patro)]
    for i in range(llargada_patro):
        matriu_distancies[i][0] = i
    for j in range(llargada_text):
        matriu_distancies[0][j] = j
    for i in range(1, llargada_patro):
        for j in range(1, llargada_text):
            deletion = matriu_distancies[i-1][j] + 1
            insercio = matriu_distancies[i][j-1] + 1
            substitucio = matriu_distancies[i-1][j-1]
            if patro[i-1] != text[j-1]:
                substitucio += 1
            matriu_distancies[i][j] = min(insercio,deletion,substitucio)
    return matriu_distancies[llargada_patro-1][llargada_text-1]
```

# Cercar un text aproximat: solució ingènua

Amb Levenshtein hem vist com calcular la distància entre un text i un patró. Però per resoldre el problema plantejat de la **Cerca aproximada d'un text** caldrà calcular la distància del patró a totes les subcadenes de T, i llavors escollir la subcadena que té una distància menor:

```
a="hola"  
cont=0  
for j in range(len(a)):  
    for i in range(j+1,len(a)+1):  
        cont=cont+1  
    print (cont,(a[j:i]))
```

## Cercar un text: solució ingènua. Cost

Si  $n$  és la longitud de la cadena, el nombre de subcadenes és  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  que té una complexitat  $O(n^2)$ . I, com hem vist, el cost de calcular la distància per cada subcadena és d' $O(n*m)$ .

Per tant la complexitat total seria de  $O(n^3 * m)$

# Cercar un text aproximat: optimització

Només cal adonar-se que si a la matriu de Levenshtein omplim la primera fila amb zeros tindrem una petita variació que ens permetrà trobar la subcadena de distància mínima!

	text	M	E	I	L	E	N	S	T	E	I	N
patró	0	0	0	0	0	0	0	0	0	0	0	0
L	1	1	1	1	0	1	1	1	1	1	1	1
E	2	2	1	2	1	0	1	2	2	1	2	2
V	3	3	2	2	2	1	1	2	3	2	2	3
E	4	4	3	3	3	2	2	2	3	3	3	3
N	5	5	4	4	4	3	2	3	3	4	4	3
S	6	6	5	5	5	4	3	2	3	4	5	4
H	7	7	6	6	6	5	4	3	3	4	5	5
T	8	8	7	7	7	6	5	4	3	4	5	6
E	9	9	8	8	8	7	6	5	4	3	4	5
I	10	10	9	8	9	8	7	6	5	4	3	4
N	11	11	10	9	9	9	8	7	6	5	4	3

# Cercar un text aproximat: optimització. Un altre cas

	text	C	A	G	A	T	A	A	G	A	G	A	A
patró	0	0	0	0	0	0	0	0	0	0	0	0	0
G	1	1	1	0	1	1	1	1	0	1	0	1	1
A	2	2	1	1	0	1	1	1	1	0	1	0	1
T	3	3	2	2	1	0	1	2	2	1	1	1	1
A	4	4	3	3	2	1	0	1	2	2	2	1	1
A	5	5	4	4	3	2	1	0	1	2	3	2	1

En aquest exemple la distància mínima és 0, però veiem que, a diferència de Levenshtein, aquí es podrien donar diverses solucions. A l'exemple es mostra que hi ha diverses subcadenes que tenen distància 1.

# Camí enrere

Quan usem 0 a la primera fila la mínima distància és el mínim de la darrera fila, però a quin text es correspon? Per saber-ho hem de desfer el camí fet.

	text	C	A	G	A	T	A	A	G	A	G	A	A
patró	0	0	0	0	0	0	0	0	0	0	0	0	0
G	1	1	1	0	3	1	1	1	1	0	1	0	1
A	2	2	1	1	0	1	1	1	1	0	1	0	1
T	3	3	2	2	1	0	1	2	2	1	1	1	1
A	4	4	3	3	2	1	0	1	2	2	2	1	1
A	5	5	4	4	3	2	1	0	7	1	2	3	2

	text	C	A	G	A	T	A	A	G	A	G	A	A
patró	S	I	I	I	I	I	I	I	I	I	I	I	I
G	D	S	S	C	S	S	S	S	C	S	C	S	S
A	D	S	C	D	C	I	C	C	D	C	I	C	C
T	D	S	D	S	D	C	I	S	S	D	S	D	S
A	D	S	C	S	C	D	C	C	I	C	S	C	C
A	D	S	C	S	C	D	C	C	I	C	S	C	C

A mida que anem avançant a la matriu de Levenshtein anirem apuntant el canvi fet i així el podem desfer. C o S ens movem en diagonal, I ens movem cap a l'esquerra, D cap amunt.

# Levenshtein. Exercici

Pots omplir la matriu de Levenshtein del patró “sitting” i del text “kitten”?

	k	i	t	t	e	n
s						
i						
t						
t						
i						
n						
g						

# Levenshtein. Exercici 2

Ara que tens la solució, pots desfer el camí fet?

		<b>k</b>	<b>i</b>	<b>t</b>	<b>t</b>	<b>e</b>	<b>n</b>
	0	1	2	3	4	5	6
<b>s</b>	1	1	2	3	4	5	6
<b>i</b>	2	2	1	2	3	4	5
<b>t</b>	3	3	2	1	2	3	4
<b>t</b>	4	4	3	2	1	2	3
<b>i</b>	5	5	4	3	2	2	3
<b>n</b>	6	6	5	4	3	3	2
<b>g</b>	7	7	6	5	4	4	3

# Subcadenes amb condicions en algorismes de text

Un altre problema habitual en cadenes de text és buscar una subcadena que compleixi una propietat: sense caràcters repetits, amb tantes vocals com consonants, que sigui palíndrom, etc.

Aquest algorisme s'aplica a sumes màximes, anàlisi de fluxos de dades, cerca d'expressions regulars...

## Solució ingènua

Com hem vist en els problemes anteriors recórrer totes les subcadenes és molt car computacionalment, d'ordre  $O(n^2)$  o pitjor.

# Finestres lliscants

Una solució òptima a la cerca de subcadenes que compleixin certa condició és l'ús de finestres lliscants. Una finestra és un fragment de text delimitat per dos punters:

- iniciFinestra (punter esquerre)
- finalFinestra (punter dret)

La finestra pot créixer o reduir-se segons la propietat que busquem.

Només llegim el text una sola vegada (recorregut linial,  $O(n)$ )

## Solució ingènua

Com hem vist en els problemes anteriors recórrer totes les subcadenes és molt car computacionalment, d'ordre  $O(n^2)$  o pitjor.

## Exemple: subcadena més llarga sense caràcters repetits

a b c a b d c b b Finestra

i f

[a]

i f

[a b]

i f

[a b c]

i f

**!a repetida**

i f

[b c a]

i f

**!b repetida**

i f

[c a b]

Subcadena més llarga

[a]

[a b]

[a b c]

[a b c]

[a b c]

# Finestres lliscants. Exercicis

Pots continuar l'algorisme i definir la finestra actual i la subcadena més llarga 3 illetres més?

Quin és el resultat final?

# Notebook de suport



Observa el codi de la subcadena més llarga al notebook dels codis de teoria

# Gràcies i recapitulació



MIREIA RIBERA | ACCESSIBILITAT DIGITAL  
| EXPERIÈNCIA D'USUARI  
| VISUALITZACIÓ DE DADES



UNIVERSITAT DE  
BARCELONA

# Recapitulació

-  El text és un format molt costós de processar amb algoritmes
-  Els algorismes de Boyer-Moore-Horspool i Levenshtein ens ajuden a optimitzar la cerca de text exacte o aproximat.
-  Les finestres lliscants són un mètode eficient de trobar subcadenes que compleixin una condició.