Tema 4: Algorismes per text

Resum del tema 4

- Cerca de cadenes
- Cerca aproximada
- Distància d'edició
    - Levenshtein

Cerca de cadenes de caràcters

Són algorismes crítics en moltes aplicacions importants de la informàtica:

- Editors de text (cerca, ortografia, etc.).
- Bioinformàtica.
- Cercadors d'Internet.
- Bases de dades.
- Compressió.
- Antivirus.
- Etc.

Cerca de cadenes de caràcters

Considerem el següent problema:

> Tenim un string *P* de `m` caràcters (el que volem trobar) i un string *T* de `n` caràcters, `n > m` dins el qual buscar. Aquests strings se solen anomenar **P**atró de cerca i **T**ext on buscar.

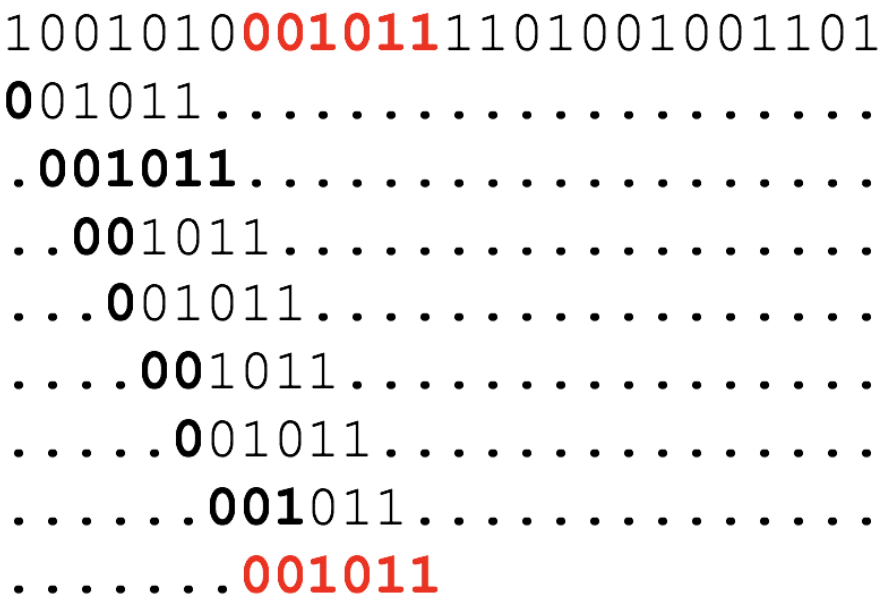
Per exemple:

```
P: 001011

T: 10010101101001100101111010

P: happy

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

P: GATTCAC

T: ATCGGATATCCGGAAACTGGTAGCGTGTAGGAGGTAGCCTGGAAG
```

Cerca de cadenes de caràcters: versió ingènua

```
P: 001011

T: 10010101101001100101111010
```

En una primera instància, podríem comparar tot l'string amb cada possible posició, però fàcilment podem millorar-ho...



Cerca de cadenes de caràcters: versió ingènua

Algorisme de força bruta:

- Alineem el patró al principi del text.
- Ens movem d'esquerra a dreta, comparant cada caràcter del patró amb el caràcter corresponent del text fins que tots els caràcters fan correspondència o trobem una diferència.
- Mentre hi hagi diferències i no haguem recorregut tot el text, realiniem una posició més a la dreta i repetim el pas 2.

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

Cerca de cadenes de caràcters: versió ingènua

La complexitat de l'algorisme es pot analitzar en tres situacions:

- En moltes ocasions, fem una comparació i movem. Aquest és el **millor cas**, i la complexitat si per tots els moviments féssim això seria `O(n)`. Aquest seria el cas, per exemple, de tenir una patró que comença per una lletra que no apareix al text.
- En d'altres, fem totes les comparacions. Aquest és el **pitjor cas**, i la complexitat, si per tots els moviments féssim això, seria `O(n*m)`.
- En un cas real, amb llenguatge natural, la **complexitat mitja** d'aquest algorisme s'acosta a `O(n+m)=O(n)`(l'única manera de calcular aquesta complexitat és de forma empírica: fent experiments).

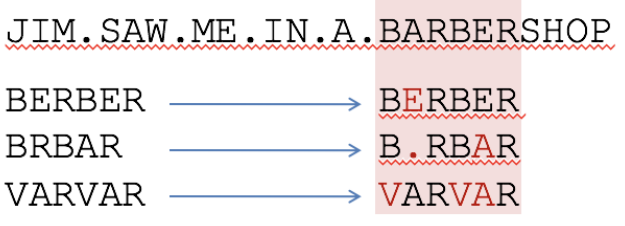
Hi ha algorismes (com l'algorisme de **Boyer-Moore**) que són lleugerament més òptims que la cerca ingènua, tot i que des del punt de vista de la complexitat són també `O(n)`.

Altres problemes

La cerca no és l'únic problema interessant:

- Buscar la subcadena més gran en comú entre dos texts.
- Cerca aproximada.
- Altres

El problema de la **cerca aproximada** és: donat un patró `P[1..m]` i un text `T[1..n]`, trobar la subcadena de `T` amb la distància d'edició mínima respecte a `P`. La **distància d'edició** és el nombre d'operacions primitives per convertir un string en un altre.



En el primer cas de l'exemple hem de fer una *edició* de la paraula `BERBER` per convertir-la en `BARBER`. En el segon en calen 2 i en el tercer cas en calen 3.

Cerca aproximada de cadenes.

Un algorisme basat en la força bruta **calcularia la distància d'edició de `P` a totes les subcadenes de `T`, i llavors escolliria la que té distància mínima**.

Com calculem totes les subcadenes d'una cadena?

```
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]))
```

Els substrings de `hola` són `h`, `o`, `l`, `a`, `ho`, `ol`, `la`, `hol`, `ola` i `hola`

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)`.

Cerca aproximada de cadenes.

Un algorisme basat en la força bruta per fer cerca aproximada de cadenes tindria una complexitat `O(n^3 * m)`, atès que (com veurem més endavant) el càlcul de la distància d'edició té `O(n*m)`.

Hi ha algorismes més òptims per fer-ho?

Càlcul de la distancia d'edició: Levenshtein

Abans de veure com cercar un patró (curt) en un text (llarg), anem a veure com calcular la “distància” `d` entre dos strings (curts).

> Quina és la distància entre `BARBER` i `BRBAR`?

Això es fa amb l’algorisme de Levenshtein:

> В.И. Левенштейн (1965). "Двоичные коды с исправлением выпадений, вставок и замещений символов". Доклады Академий Наук СССР163 (4): 845–8.

Traduït a l'anglès: Levenshtein VI (1966). “Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady 10: 707–10.

Càlcul de la distancia d'edició: Levenshtein

Aquest algorisme (també anomenat *distància d'edició*) calcula el nombre mínim d'operacions d'edició que són necessàries per modificar una cadena `P` i obtenir-ne una altra `T`.

Usualment, les operacions d'edició són:

- inserció (p.e., canviar `cot` per `coat`),
- eliminació (p.e., canviar `coat` per `cot`), i
- substitució (p.e., canviar `coat` per `cost`).

També es podria considerar la transposició: canviar `cost` per `cots`.

Càlcul de la distancia d'edició: Levenshtein

Per fer-ho, va omplint una matriu *d* de manera que la posició `[m,n]` representa la distància d'edició entre el prefix de `m` caràcters d'un patró i el prefix de `n` caràcters d'un text.

patró	L	E	V	E	N	S	H	T	E	I	N
text	M	E	I	L	E	N	S	T	E	I	N

`d[1][1]`, canviar `L` per `M`, val 1 doncs només és una substitució.

`d[1][3]`, canviar `L` per `MEI`, val 3 perque és una substitució i dues insercions.

Càlcul de la distancia d'edició: Levenshtein

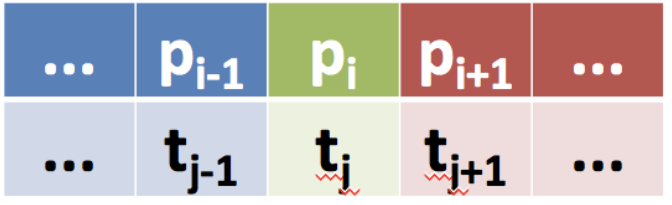
d	.	*t1*	*t2*	*t3*	*t4*
.	0	1	2	3	4
p1	1				
p2	2			d[2,3]	
p3	3				

Aquests valors són evidents

Com calculem aquests valors ?

Càlcul de la distancia d'edició: Levenshtein

Suposem que ja tenim una alineació òptima entre els prefixos `p[0,i-1]` i `t[0,j-1]`. Què podem fer amb `p[i]` i `t[j]` i com calculem `d[i,j]`?

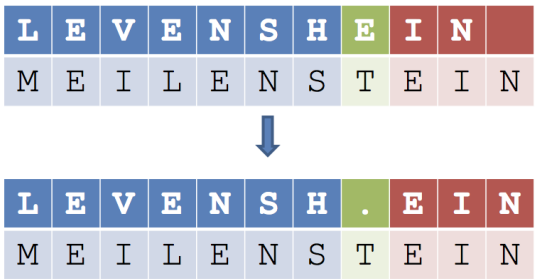


Només podem fer tres coses!

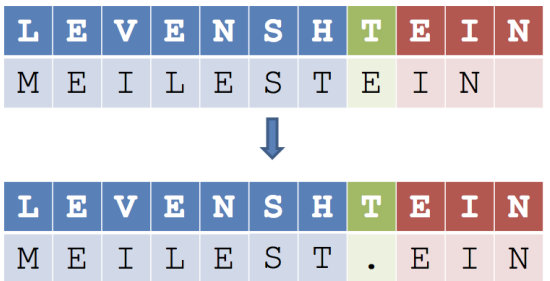
1) Fem que `p[i]` i `t[j]` facin correspondència. Si `p[i]=t[j]` llavors `d[i,j]=d[i-1,j-1]`. Si no substituim i `d[i,j]=d[i-1,j-1]+1`.

Càlcul de la distancia d'edició: Levenshtein

2) Decidim que hi ha un forat al patró, i per tant inserim i `d[i,j]=d[i-1,j]+1`



3) Decidim que hi ha un forat al text, i per tant eliminem i `d[i,j]=d[i,j-1]+1`



Càlcul de la distancia d'edició: Levenshtein

Observació:

`d[i,j]= min{d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + cost}`

Això és podria resoldre amb una crida recursiva, atès que nosaltres volem `d[m,n]` i coneixem `d[0,:]` i `d[:,0]`, però la crida recursiva té massa cost computacional!

Podem seguir la mateixa estratègia que vam fer servir per la seqüència de Fibonacci.

Càlcul de la distancia d'edició: Levenshtein

Observació:

```
d[i,j]= min{d[i-1,j] + 1, d[i,j-1] + 1, d[i-1,j-1] + cost}
```

		G	U	M	B	O
	0	1	2	3	4	5
G	1					
A	2					
M	3					
B	4					
O	5					
L	6					

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0				
A	2	1				
M	3	2				
B	4	3				
O	5	4				
L	6	5				

Càlcul de la distancia d'edició: Levenshtein

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1			
A	2	1	1			
M	3	2	2			
B	4	3	3			
O	5	4	4			
L	6	5	5			

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2		
A	2	1	1	2		
M	3	2	2	1		
B	4	3	3	2		
O	5	4	4	3		
L	6	5	5	4		

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	
A	2	1	1	2	3	
M	3	2	2	1	2	
B	4	3	3	2	1	
O	5	4	4	3	2	
L	6	5	5	4	3	

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

Càlcul de la distancia d'edició: Levenshtein

La matriu es pot omplir seqüencialment:

```
Per cada caràcter de s (i des de 1 fins n):
  Per cada caràcter de t (j des de 1 fins m):
      Si s[i] == t[j]: cost = 0
      Si s[i] != t[j]: cost = 1
      d[i,j] = mínim (d[i-1,j] + 1,
                      d[i,j-1] + 1,
                      d[i-1,j-1] + cost)
```

Això té una complexitat `O(m*n)` equivalent a calcular tots els elements de la matriu.

Càlcul de la distancia d'edició: Levenshtein

El nombre que queda a la **cantonada de baix a la dreta** de la matriu és la distància de Levenshtein, o d'edició, entre les dues paraules.

Si volem saber les operacions d'edició efectuades, hem de buscar el **camí mínim entre els extrems de la matriu o simplement guardar a cada pas la decisió presa respecte a l'edició**.

		m	e	i	l	e	n	s	t	e	i	n
	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

Càlcul de la distancia d'edició: Levenshtein

Pot haver-hi diversos possibles passos de cost mínim:

patró	L	E		V	E	N	S	H	T	E	I	N
	s	=	-	s	=	=	=	+	=	=	=	=
text	M	E	I	L	E	N	S		T	E	I	N

patró	L	E	V		E	N	S	H	T	E	I	N
	s	=	s	-	=	=	=	+	=	=	=	=
text	M	E	I	L	E	N	S		T	E	I	N

Càlcul de la distancia d'edició: Levenshtein

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Càlcul de la distancia d'edició: Levenshtein

```
def levenshtein_distance(first, second):
    if len(first) > len(second):
        first, second = second, first
    if len(second) == 0:
        return len(first)
    first_length = len(first) + 1
    second_length = len(second) + 1
    distance_matrix = [[0] * second_length for x in range(first_length)]
    for i in range(first_length):
        distance_matrix[i][0] = i
    for j in range(second_length):
        distance_matrix[0][j] = j
    for i in range(1, first_length):
        for j in range(1, second_length):
            deletion = distance_matrix[i-1][j] + 1
            insertion = distance_matrix[i][j-1] + 1
            substitution = distance_matrix[i-1][j-1]
            if first[i-1] != second[j-1]:
                substitution += 1
            distance_matrix[i][j] = min(insertion,deletion,substitution)
    return distance_matrix[first_length-1][second_length-1]
```

Càlcul de la distancia d'edició: Levenshtein

```
def levenshtein_distance(first, second):
    if len(first) > len(second):
        first, second = second, first       # el primer sempre més curt
    if len(second) == 0:
        return len(first)
      first_length = len(first) + 1
      second_length = len(second) + 1
      distance_matrix = [[0] * second_length for x in range(first_length)]
      ...
```

`distance_matrix = [[0] * second_length for x in range(first_length)]` és una **comprensió** de Python, que es pot interpretar com:

```
distance_matrix = []
for x in range(first_length):
    distance_matrix.append([0] * second_length)
```

Càlcul de la distancia d'edició: Levenshtein

```
    ...                                     # valors inicials
    for i in range(first_length):
        distance_matrix[i][0] = i
    for j in range(second_length):
        distance_matrix[0][j] = j

    for i in range(1, first_length):   # recorregut resta caselles
        for j in range(1, second_length):

            deletion = distance_matrix[i-1][j] + 1
            insertion = distance_matrix[i][j-1] + 1
            substitution = distance_matrix[i-1][j-1]

            if first[i-1] != second[j-1]:
                substitution += 1      # substitution val 0 o 1

            distance_matrix[i][j] = min(insertion,deletion,substitution)

    return distance_matrix[first_length-1][second_length-1]
```

Cerca aproximada de cadenes

Recordem que el nostre problema era:

> Donat un patró `P[1..m]` i un text `T[1..n]`, trobar la subcadena de `T` amb la distància d'edició mínima respecte a `P`.

Aquest càlcul es pot fer amb l'algorisme de Levenshtein.

Només cal adonar-se que si a la matriu de Levenshtein omplim la primera fila amb zeros (equival a considerar que el cost d'inserir espais en blanc al davant del patró és nul) tindrem una petita variació que ens permetrà trobar les subcadenes de distància mímina!

Cerca aproximada de strings

El càlcul de la matriu té una complexitat de `O(mn)`, mentre que la cerca del camí marxa enrere té una complexitat `O(n+m)`.

		-1	0 C	1 A	2 G	3 A	4 T	5 A	6 A	7 G	8 A	9 G	10 A	11 A
-1		0	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
1	A	2	2	1	1	0	1	1	1	1	0	1	0	1
2	T	3	3	2	2	1	0	1	2	2	1	1	1	1
3	A	4	4	3	3	2	1	0	1	2	2	2	1	1
4	A	5	5	4	4	3	2	(1)	(0)	(1)	2	3	2	(1)

T: `la cassa mes gran que mai ha existit`

P: `casa`

Trobem tres respostes a distància 1: `cas`, `cass`, `cassa`

Possibles preguntes d'exàmen relacionades amb el tema 4

- Completa la següent matriu de Levensthein

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1			
A	2	1	1			
M	3	2	2			
B	4	3	3			
O	5	4	4			
L	6	5	5			

- Quina distància d'edició tenen les següents paraules?

GAT GOS
GAT TIGRE

- En el pitjor cas i amb força bruta quin és el cost de cercar un patró en un text?