UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques

Algorísmica

Algorismes per text

Jordi Vitrià

Cerca de *strings*...

Són algorismes crítics en moltes aplicacions importants de la informàtica:

- Editors de text (*search*, *spell*, etc.).
- Bioinformàtica.
- Cercadors d'Internet.
- Bases de dades.
- Compressió.
- Antivirus.
- Etc.

Cerca de *strings*...

Considerem el següent problema:

El patró

Tenim un *string* de m caràcters (el que volem trobar) i un *string* de n caràcters, $n > m$ dins del qual buscar.

El text

```
P: 001011
T: 100101011010011001011110 10

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

P: GATTCAC
T: ATCGGATATCCGGAAACTGGTAGCGTGTAGGAGGTAGCCTGGAAG
```

Cerca de *strings*: la versió ingènua.

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

```
P:  001011
T:  10010101101001100101111010
```

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

Cerca de *strings*: la versió ingènua.

Algorisme de força bruta:

1. Alineem el patró al principi del text.
2. 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.
3. Mentre hi hagi diferències i no haguem recorregut tot el text, re-alineem una posició més a la dreta i repetim el pas 2.

Cerca de *strings*: Algorisme de força bruta.

```
def BFStringMatching(t,p):
    m=len(p)
    n=len(t)
    for i in range(0,n-m+1):
        j=0
        while j<m and p[j]==t[i+j]: j=j+1
        if j == m: return i
    return -1
```

10010100010111101001001101

001011

m-1

n-m

n-1

Cerca de *strings*: la 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 *O(n)*.

Aquest seria el cas 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 *O(nm)*.

- Quan parlem de *llenguatge natural*, la complexitat **mitja** s'acosta més a *O(n+m)=O(n)*.

S'ha de calcular empíricament

Cerca de *strings*: Versions avançades.

Una de les formes que tenim per reduir aquesta complexitat és **pre-processar** l'entrada de l'algorisme per optimitzar el seu funcionament.

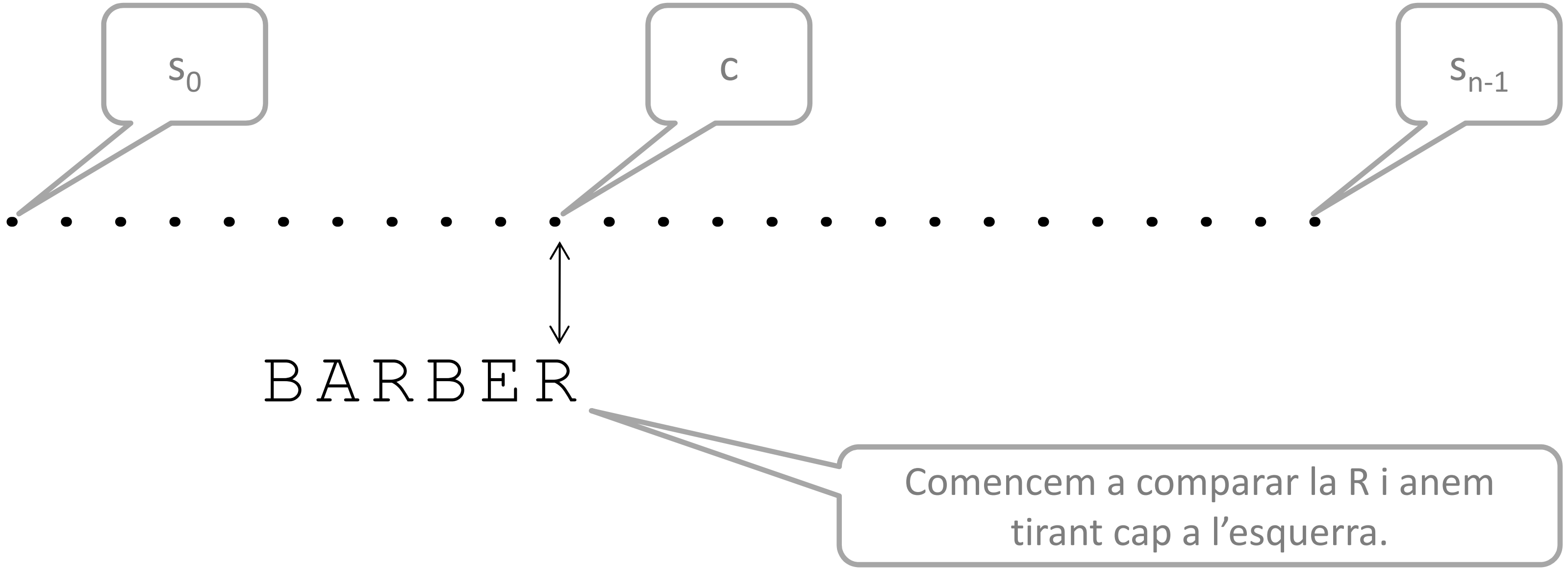
Aquesta estratègia s'usa en molts àmbits de l'algorísmica.

Per exemple, **l'algorisme de Horspool (1980)** (que és una versió millorada de l'algorisme de **Boyer-Moore, 1977**), que **pre-processa** el patró per analitzar el seu contingut, **genera una taula** que li doni informació útil a l'hora de fer desplaçaments i durant el recorregut fa els desplaçaments basant-se en aquesta taula.

Algorisme de Horspool

La majoria de vegades, falla el primer caràcter comparat. Si comencem per l'esquerra només saltem una posició. Per la dreta podem fer un salt més gran!

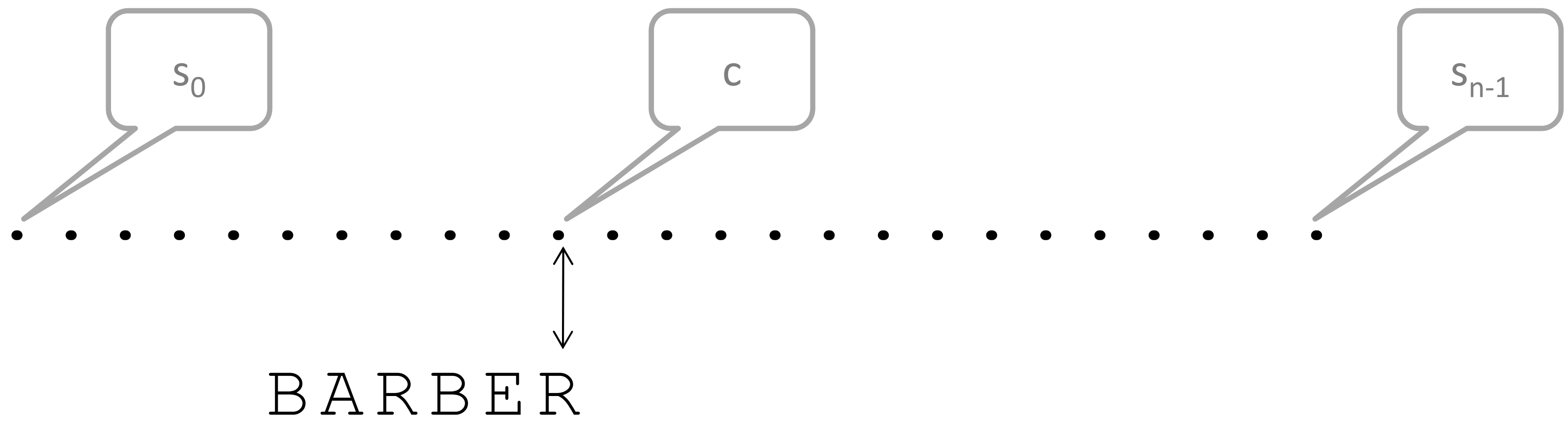
La primera observació és que, després del desplaçament, **començar a comparar text i patró per la dreta** ens proporciona avantatja a l'hora de **precalcular els desplaçaments**!



Algorisme de Horspool

Si tots els caràcters són iguals, hem acabat.
Sinó, desplaçarem el patró cap a la dreta **al màxim**, sense risc de perdre una instància seva al text!

Fins on? Això bàsicament depèn de *c*!



Algorisme de Horspool

Quan falla la comparació entre patró i text, ens podem trobar amb varies situacions:

1. El caràcter *c* no està present al patró:

```
.....S.........................
BARBER
```

és segur desplaçar el patró *m* posicions.

```
.....S.........................
......BARBER
```

Algorisme de Horspool

2. El caràcter *c* està present al patró :

```
.....B........................
BARBER
```

és segur desplaçar fins a la primera aparició del caràcter.

```
.....B........................
..BARBER
```

Algorisme de Horspool

En el cas que *c* no hagi fallat, però <u>falla un altre caràcter</u> més endavant:

3. El caràcter *c* no està present a la resta del patró:

```
...MER.........................
LEADER
```

és segur desplaçar el patró *m* posicions.

```
...MER.........................
......LEADER
```

Algorisme de Horspool

4. El caràcter *c* està present a la resta del patró:

```
....AR........................
BARBER
```

podem desplaçar fins a la primera aparició del caràcter.

```
....AR........................
...BARBER
```

Algorisme de Horspool

És evident que ens estalviem comparacions respecte a l’algorisme basat en força bruta, però també ho és que si hem de fer totes les comprovacions necessàries per saber en quin cas ens trobem en el moment que falla una comparació tampoc hi guanyem res!

Hi ha una instància de *c* a la resta del patró?

El que farem és pre-calcular la taula de desplaçaments.

La taula ens donarà un desplaçament per cada possible lletra de l’alfabet.

Algorisme de Horspool

Els desplaçaments es poden pre-calcular, mirant el patró, amb aquesta fórmula:

- Si *c* no està entre els primer *m-1* caràcters del patró, desplaçament = *m*.
- En tots els altres casos, desplaçament = distància des de la primera aparició *c* (començant per la dreta) al patró.

En el cas de `BAOBAB` tenim:

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

Algorisme de Horspool

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

Distància des de la primera aparició c (començant per la dreta) al patró.

Aquesta instrucció converteix qualsevol seqüència a una tupla.

Algorisme de Horspool: exemple.

```
JIM.SAW.ME.IN.A.BARBERSHOP

BARBER
....BARBER
.....BARBER
...........BARBER
.............BARBER
................BARBER
```

La complexitat en el **pitjor cas** és *O(nm).* En el **cas promig**, *O(n)*, però tot i estar en la mateixa classe de complexitat, és més eficient que l'algorisme de força bruta.

Altres problemes

La cerca no és l'únic problema interessant:

- Buscar el *substring* més gran en comú entre dos texts.
- **Cerca aproximada**.
- Etc.

Cerca aproximada de *strings*.

El **problema** és: donat un patró *P[1..m]* i un text *T[1..n],* trobar el *substring* de *T* amb la **distància d'edició mínima** respecte a *P*.

```
JIM.SAW.ME.IN.A.BARBERSHOP

BERBER      →      BERBER
BRBAR       →      B.RBAR
VARVAR      →      VARVAR
```

> La distància d'edició és el nombre d'operacions primitives per convertir un *string* en un altre.

Un algorisme basat en la **força bruta** calcularia la distància d'edició de *P* a **tots els *substrings*** de *T*, i llavors escolliria el que té distància mínima.

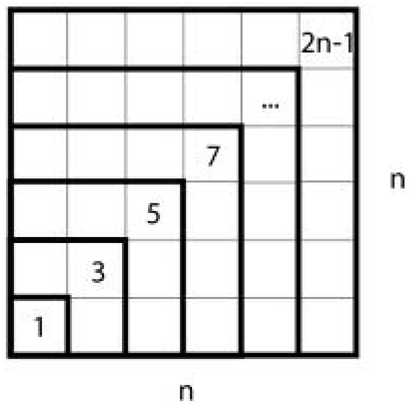
Cerca aproximada de *strings*.

```
1  h
2  ho
3  hol
4  hola
5  o
6  ol
7  ola
8  l
9  la
10 a
```

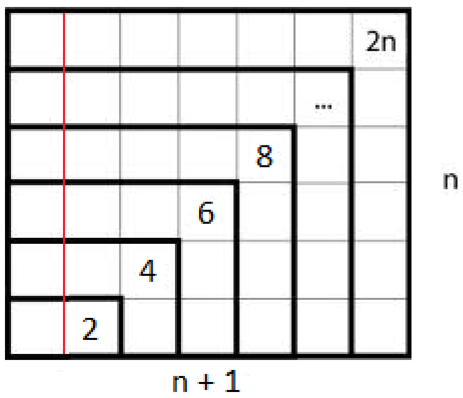
```
a="hola"
cont=0
for j in range(len(a)):
    for i in range(j+1,len(a)+1):
        cont=cont+1
        print cont,(a[j:i])
```

El nombre de substrings és $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ $= O(n^2)$

Un algorisme basat en la força bruta tindria una complexitat ***$O(n^3m)$***, atès que (com veurem) el càlcul de la distància d'edició té ***$O(nm)$***.



La suma dels primers *n* nombres senars (fins el nombre 2n-1) és n^2.



La suma dels primers *n* nombres parells (fins el nombre 2n) és $n\,(n+1)$.

Algorisme de 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).

d ("BARBER" , "BRBAR") =?

d ("BARBER" , "BABRER") =?

. . .

Això es fa amb l'algorisme de **Levenshtein**.

В.И. Левенштейн (1965). "Двоичные коды с исправлением выпадений, вставок и замещений символов". *Доклады Академий Наук СССР* **163** (4): 845–8. Appeared in English as: Levenshtein VI (1966). "Binary codes capable of correcting deletions, insertions, and reversals". *Soviet Physics Doklady* **10**: 707–10.

Algorisme de Levenshtein

Aquest algorisme (també anomenat “**distància d’edició**”) calcula el nombre mínim **d’operacions d’edició** que són necessàries per modificar un *string* *P* i obtenir-ne un altre *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*.

Algorisme de 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]=1, L→M, doncs només és una substitució.
d[1][3]=3, L→MEI, és una substitució i 2 insercions.

Algorisme de 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 ?

Algorisme de 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]*?

...	p_{i-1}	p_i	p_{i+1}	...
...	t_{j-1}	t_j	t_{j+1}	...

Només podem fer tres coses!

Algorisme de Levenshtein

1. Fem que *p[i]* i *t[j]* facin correspondència. Si *p[i]=t[j]* llavors *d[i,j]=d[i-1,j-1]*. Sinó, *d[i,j]=d[i-1,j-1]+1*

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

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

Algorisme de Levenshtein

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

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

L	E	V	E	N	S	H	.	E	I	N
M	E	I	L	E	N	S	T	E	I	N

Algorisme de Levenshtein

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

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

L	E	V	E	N	S	H	T	E	I	N
M	E	I	L	E	N	S	.	E	I	N

Algorisme de Levenshtein

Observació:

0 si són iguals, 1 si són hi ha substitució

$$d[i,j]=$$

$$\min\left\{d[i-1,j]+1,\ d[i,j-1]+1,\ d[i-1,j-1]+\text{cost}\right\}$$

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.

Algorisme de Levenshtein

```
mínim (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				

		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

Algorisme de 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)
```

Eliminació del text

Inserció al text

Substitució

Això té una complexitat *O(mn)* = calcular tots els elements de la matriu.

Algorisme de 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 <u>simplement guardar a cada pas la decisió presa respecte a l’edició</u>.

Algorisme de Levenshtein

		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

Algorisme de 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

Algorisme de 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

Algorisme de 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 xrange(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]
```

Algorisme de Levenshtein (i)

```
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)]
...
```

```
>>> a = [[0] * 3 for x in range (3)]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Algorisme de Levenshtein (ii)

```
....

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]
```

Cerca aproximada de *strings*.

Recordem que el nostre problema era:

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

```
JIM.SAW.ME.IN.A.BARBERSHOP

BERBER              BERBER
BRBAR               B.RBAR
VARVAR              VARVAR
```

Cerca aproximada de *strings*.

Aquest problema el podem reformular així:

1. Per cada posició *j* del text *T*, i cada posició *i* al patró *P*, comprovarem tots els *substrings* de *T* que acaben a *j*, i determinarem quin és el que té distància mínima d'edició als primers *i* caràcters de *P*. Aquesta distància serà ***E(i,j)***.

p_0	…	p_{i-3}	p_{i-2}	p_{i-1}	p_i	p_{i+1}	…	d
					t_j			
				t_{j-1}	t_j			
			t_{j-2}	t_{j-1}	t_j			
		t_{j-3}	t_{j-2}	t_{j-1}	t_j			

$E(i,j)=min\{d\}$

Cerca aproximada de *strings*.

2. Un cop calculada $E(i,j)$ per tots els valors de i i j, podem trobar la solució al problema original: **és el(s) *substring(s)* amb $E(m, j)$ mínima** (essent m la longitud de P).

Però, com calculem $E(m, j)$ de forma eficient?...

Cerca aproximada de *strings*.

El càlcul de $E(m, j)$ es pot fer amb **l'algorisme de Levenshtein**.

L'única diferència és que s'ha d'inicialitzar la **primera fila amb zeros (=considerar que podem inserir tants espais en blanc al davant del patró com sigui necessari)** i guardar com hem calculat $E(m, j)$, és a dir si hem usat $E(i-1,j)$, $\mathrm{E}(i,j-1)$ o $E(i-1,j-1)$ al calcular $E(i, j)$.

Busquem totes les solucions (camins) amb dist<1

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

$$\begin{pmatrix} & & G & A & T & A & A & & & & & & \\ C & A & G & A & T & - & A & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & & G & A & T & A & A & & & & & & \\ C & A & G & A & T & A & - & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & & G & A & T & A & A & & & & & \\ C & A & G & A & T & A & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & - & G & A & T & A & A & & & & & \\ C & A & G & A & T & A & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & & & G & A & T & A & A & & & & & \\ C & A & G & - & A & T & A & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & & G & A & T & A & A & - & & & & \\ C & A & G & A & T & A & A & G & A & G & A & A \end{pmatrix}$$

$$\begin{pmatrix} & & & & & & & G & A & T & A & A \\ C & A & G & A & T & A & A & G & A & G & A & A \end{pmatrix}$$

Cerca aproximada de *strings*.

El càlcul de $E(x, y)$ té una complexitat de $O(mn)$, mentre que la cerca del camí marxa enrere té $O(n+m)$.

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

Trobem tres respostes a distància 1:

```
cas
cass
cassa
```