INTRODUCTION TO NLP

Session 3

David Buchaca Prats

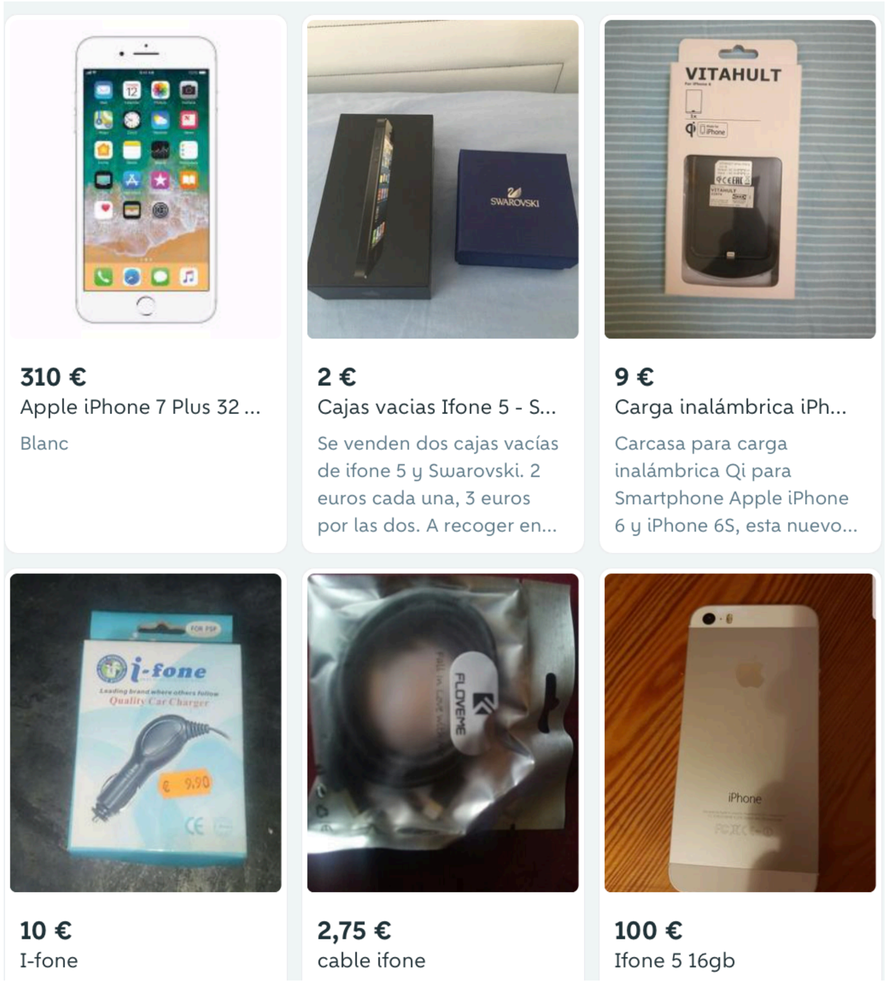
2022

PROBLEMS WORKING WITH STRINGS

- Strings treated as elements within a vocabulary are problematic.
  - A string might be in the Vocabulary, but the same string with a single character change might not be in the Vocabulary.
  - Small changes in the input string might have dramatic consequences:
    - **‘motorbike’** in Vocabulary
    - **‘motorvike’** not in Vocabulary
- To deal with this type of issues many NLP applications process the data using edit distances.

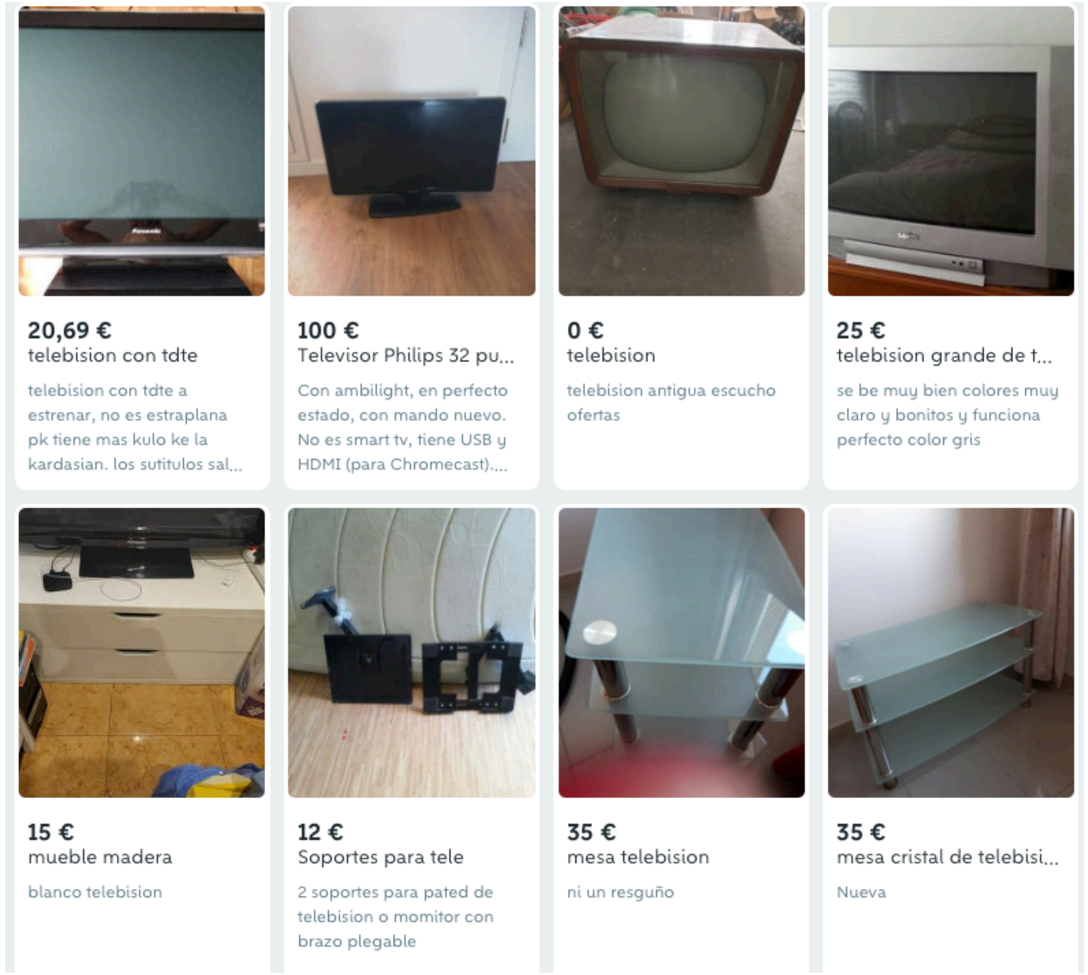
EXAMPLE: ‘IFONE’

- Consider users writting **‘ifone’**
- Users will get information from documents containing **‘ifone’** but not **‘iphone’** (unless both **‘ifone’** an **‘iphone’** appear in the text).



EXAMPLE: ‘TELEBISION’

No string distances used



String distances are used

String distances allow searching for similar strings

STRING DISTANCE INTUITION

- The objective of a string distance is to measure how different two strings are.
  - If two strings are exactly the same the edit distance has to be zero.
  - If two strings differ from a single character the edit distance has to be the distance provided by the different character.
  - If two strings differ from two characters the edit distance has to be the distance provided by the different characters.
- In other words, we want to know the minimum number of edit operations between two strings, where operations are:
  - Insert: Add a character in a given position.
  - Delete: Delete a character in a given position.
  - Substitute: Substitute a character in a given position by another character.

JACCARD DISTANCE

- Jaccard similarity: $s_{jaccard}(x, y) = \frac{|x \cap y|}{|x \cup y|}$

```
def jaccard_similarity(s1,s2):
    return len(s1.intersection(s2)) / len(s1.union(s2))
```

```
jaccard_similarity(set("exponential"),set("exponentia"))
```

executed in 3ms, finished 16:36:48 2020-03-01

```
0.8888888888888888
```

```
def jaccard_distance(s1,s2):
    return 1- jaccard_similarity(s1,s2)
```

```
jaccard_similarity(set("exponential"),set("polynomial"))
```

executed in 2ms, finished 16:36:30 2020-03-01

```
0.5454545454545454
```

- Jaccard distance: $d_{jaccard}(x, y) = 1 - \frac{|x \cap y|}{|x \cup y|}$

JACCARD DISTANCE

- The Jaccard distance does not fit very well with the sequential nature of strings

```
set1 = set('panmpi')
set2 = set('mapping')
jaccard_distance(set1,set2)
```

executed in 3ms, finished 16:27:53 2021-03-07

```
0.1666666666666663
```

panmpi is similar to **mapping** ?

```
set1 = set('mapping')
set2 = set('mappin')
jaccard_distance(set1,set2)
```

executed in 3ms, finished 16:27:57 2021-03-07

```
0.1666666666666663
```

mapping is similar to **mappin** ?

JACCARD DISTANCE

- Not taking into account order in strings makes the jaccard distance misleading:

```
query = set('guardin')
distances = compute_distances(query,words)
print(f"the closest word to query={query} is {words[np.argmin(distances)]}")
```

executed in 219ms, finished 16:17:26 2021-03-08

```
the closest word to query={'u', 'r', 'a', 'd', 'i', 'n', 'g'} is guardian
```

```
closest_words = [words[d] for d in np.argsort(distances)]
closest_words[0:10]
```

executed in 66ms, finished 16:17:27 2021-03-08

```
['unniggard',
 'gurniad',
 'guarding',
 'undaring',
 'guardian',
 'indiguria',
 'ungrained',
 'antidrug',
 'unarraigned',
 'underguardian']
```

SIMPLE CASE: CONSIDER ALL OPERATIONS HAVE THE SAME COST

```
def edit_distance_recursive(x,y):
    if len(x) ==0:
        return len(y)
    if len(y) == 0:
        return len(x)

    delta = 0 if x[-1] == y[-1] else 1
    return min(edit_distance_recursive(x[:-1],y[:-1]) + delta,
               edit_distance_recursive(x[:-1],y)      + 1,
               edit_distance_recursive(x,y[:-1])      + 1)
```

RECURSIVE EDIT DISTANCE

- The previous recursive implementation is correct but slow

```
n = 0
def edit_distance_recursive(x,y):
    global n
    if len(x) ==0:
        return len(y)
    if len(y) == 0:
        return len(x)

    if x =="super" and y=="sup":
        n += 1

    delta = 0 if x[-1] == y[-1] else 1
    return min(edit_distance_recursive(x[:-1],y[:-1]) + delta,
               edit_distance_recursive(x[:-1],y)      + 1,
               edit_distance_recursive(x,y[:-1])      + 1)

```

```
edit_distance_recursive("superman", "supermaniac")
n
```

```
833
```

STANDARD EDIT DISTANCE IMPLEMENTATION (WIKIPEDIA)

Computation [edit]

The first algorithm for computing minimum edit distance between a pair of strings was published by Damerau in 1964.[6]

Common algorithm [edit]

Main article: Wagner–Fischer algorithm

Using Levenshtein's original operations, the edit distance between $a = a_1 \dots a_n$ and $b = b_1 \dots b_m$ is given by d_{mn}, defined by the recurrence[2]

$$d_{i0} = \sum_{k=1}^{i} w_{\text{del}}(b_k), \qquad \text{for } 1 \le i \le m$$

$$d_{0j} = \sum_{k=1}^{j} w_{\text{ins}}(a_k), \qquad \text{for } 1 \le j \le n$$

$$d_{ij} = \begin{cases} d_{i-1,j-1} & \text{for } a_j = b_i \\ \min \begin{cases} d_{i-1,j} + w_{\text{del}}(b_i) \\ d_{i,j-1} + w_{\text{ins}}(a_j) \\ d_{i-1,j-1} + w_{\text{sub}}(a_j, b_i) \end{cases} & \text{for } a_j \neq b_i \end{cases} \qquad \text{for } 1 \le i \le m, 1 \le j \le n.$$

This algorithm can be generalized to handle transpositions by adding another term in the recursive clause's minimization.[3]

OVERVIEW OF THE LEVENSHTEIN DISTANCE

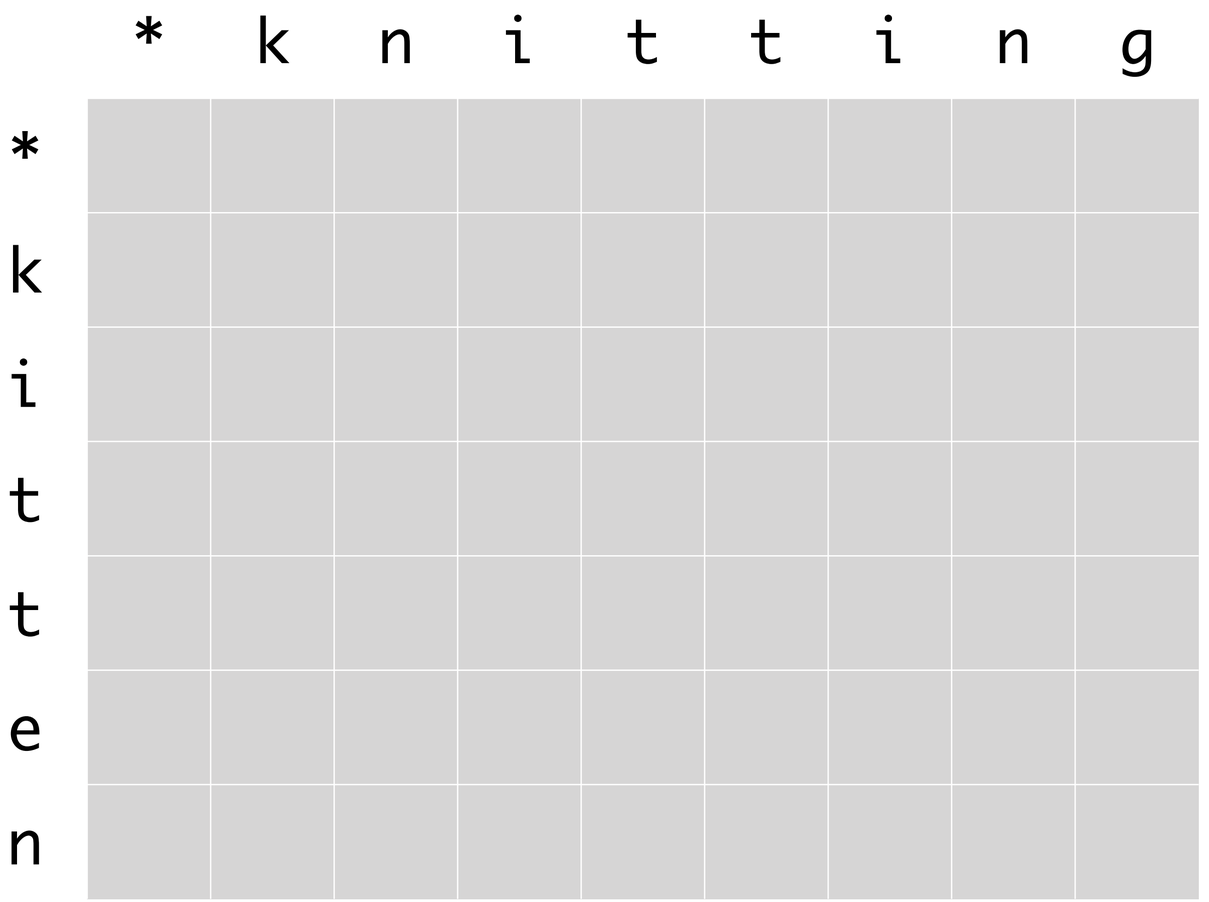
- Given **x**, **y** strings we want to compute **d(x,y)**.
- This edit distance consist on finding the cost associated to the minimum number of edits needed to go from **x** to **y**
- In order to make the computation efficiently we will reuse pre-computed substring distances.
  - We will compute **d(x,y) = d(x[:-1], y[:-1]) + something**
- We define **x[1:i]** as the first i characters from **x**
- We define **D[i,j]** as the distance between **x[:i]** and **y[:j]**
- We define **D** a matrix of shape **(len(x), len(y))** containing **D[i,j]** at position **i,j**

OVERVIEW OF THE LEVENSHTEIN DISTANCE

- Initialization
  - We start from an empty string **‘*’**
  - Since **d(‘*’, ‘c’) = 1** for any character **‘c’** we know that the cost of going from an empty string to any character is 1.

EXAMPLE: INITIALIZATION

- Given two strings s1 and s2
  - We start creating an empty array X
  - X.shape = (len(s1), len(s2))



EXAMPLE: INITIALIZATION

- Given two strings s1 and s2
  - We start creating an empty array X
  - X.shape = (len(s1), len(s2))

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1								
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[1,1]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,1} = C(0,1) + c_{del} = ? + 1$$

$$sub_{1,1} = C(0,0) + c_{sub} = ? + 1$$

$$ins_{1,1} = C(1,0) + c_{ins} = ? + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1								
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[1,1]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,1} = C(0,1) + c_{del} = ! + 1$$

$$sub_{1,1} = C(0,0) + c_{sub} = ! + 1$$

$$ins_{1,1} = C(1,0) + c_{ins} = ! + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0							
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[1,2]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,2} = C(0,2) + c_{del} = ? + 1$$

$$sub_{1,2} = C(0,1) + c_{sub} = ? + 1$$

$$ins_{1,2} = C(1,1) + c_{ins} = ? + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	→						
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[1,2]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{1,2} = C(0,2) + c_{del} = 2 + 1$$

$$sub_{1,2} = C(0,1) + c_{sub} = 1 + 1$$

$$ins_{1,2} = C(1,1) + c_{ins} = 0 + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1						
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,1]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,1} = C(1,1) + c_{del} = ? + 1$$
$$sub_{2,1} = C(1,0) + c_{sub} = ? + 1$$
$$ins_{2,1} = C(2,0) + c_{ins} = ? + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2								
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,1]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,1} = C(1,1) + c_{del} = 0 + 1$$
$$sub_{2,1} = C(1,0) + c_{sub} = 1 + 1$$
$$ins_{2,1} = C(2,0) + c_{ins} = 2 + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1							
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,2]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,2} = C(1,2) + c_{del} = ? + 1$$
$$sub_{2,2} = C(1,1) + c_{sub} = ? + 1$$
$$ins_{2,2} = C(2,1) + c_{ins} = ? + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1 →							
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,2]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,2} = C(1,2) + c_{del} = 1 + 1$$
$$sub_{2,2} = C(1,1) + c_{sub} = 0 + 1$$
$$ins_{2,2} = C(2,1) + c_{ins} = 1 + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1						
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,3]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,3} = C(1,3) + c_{del} = ? + 1$$
$$sub_{2,3} = C(1,2) + c_{sub} = ? + 1$$
$$ins_{2,3} = C(2,2) + c_{ins} = ? + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1						
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute **C[2,3]**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

$$del_{2,3} = C(1,3) + c_{del} = ! + 1$$

$$sub_{2,3} = C(1,2) + c_{sub} = ! + 1$$

$$ins_{2,3} = C(2,2) + c_{ins} = ! + 1$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1					
t	3								
t	4								
e	5								
n	6								

EXAMPLE: C COMPUTATION

- Let us compute the rest of the table….

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1	2	3	4	5	6
t	3	2	2	2	1	2	3	4	5
t	4	3	3	3	2	1	2	3	4
e	5	4	4	4	3	2	2	3	4
n	6	5	4	5	4	3	3	2	3

EXAMPLE: C COMPUTATION

- The result of **d('knitting','kitten') = C[-1,-1] = 3**

$$C(i,j) = \begin{cases} C(i-1,j-1) & \textbf{if } a_j = b_i \\ min(ins_{i,j}, del_{i,j}, sub_{i,j}) & \textbf{if } a_j \neq b_i \end{cases}$$

	*	k	n	i	t	t	i	n	g
*	0	1	2	3	4	5	6	7	8
k	1	0	1	2	3	4	5	6	7
i	2	1	1	1	2	3	4	5	6
t	3	2	2	2	1	2	3	4	5
t	4	3	3	3	2	1	2	3	4
e	5	4	4	4	3	2	2	3	4
n	6	5	4	5	4	3	3	2	3