Prácticas de Algorítmica

Búsqueda aproximada de cadenas
(2ª parte proyecto coordinado SAR-ALT)

Curso 2025-2026

Objetivos

El objetivo de esta segunda parte del proyecto conjunto SAR-ALT es:

1. Estudiar e implementar la búsqueda aproximada de una cadena respecto de todas las cadenas de un diccionario.
2. Utilizar esa búsqueda aproximada para ampliar el motor de recuperación de información desarrollado en SAR de forma que acepte consultas con búsqueda aproximada.

Búsqueda aproximada de cadenas

Búsqueda aproximada de cadenas

Hay múltiples perpectivas para abordar la búsqueda aproximada de cadenas. Nosotros nos centraremos en una búsqueda **offline**, a nivel **de palabras**, sin **considerar el contexto** y sobre un diccionario que **no se actualiza**. Esta suele ser la tarea de un corrector ortográfico:

https://en.wikipedia.org/wiki/Spelling_suggestion

Existen bibliotecas Python con esta funcionalidad, por ejemplo: `http://pyenchant.github.io/pyenchant/api/enchant.html`

```
>>> import enchant
>>> d = enchant.Dict("en_US")   # create dictionary for US English
>>> d.suggest("enchnt")
['enchant', 'enchants', 'enchanter', 'penchant', 'incant',
 'enchain', 'enchanted']
```

Búsqueda aproximada de cadenas

Utilizaremos una clase Python llamada `SpellSuggester` que recibe en el constructor un diccionario de términos/palabras y lo preprocesa. El objeto creado dispondrá del método `suggest` que recibe:

- La palabra a buscar.
- Un umbral o nivel de tolerancia (para limitar la búsqueda).

El método `suggest` devuelve una lista que contiene las palabras del diccionario que estén próximas a la buscada (dentro del umbral indicado). Esto se verá con más detalle posteriormente.

Búsqueda aproximada de cadenas

Para elegir las palabras próximas necesitamos evaluar la distancia entre dos palabras. Vamos a limitarnos a estudiar **distancias de edición:**

> *La distancia de edición entre dos cadenas $x, y \in \Sigma^\star$ es el número mínimo de operaciones de edición para convertir la primera en la segunda (o viceversa, la distancia es simétrica/conmutativa).*

Las distancias de edición más utilizadas son:

- Distancia de Levenshtein.
- Distancia de Damerau-Levenstein.

Distancias de edición

Distancias de edición

Las operaciones de edición consideradas para calcular la distancia de Levenshtein entre x e y son:

- Insertar un carácter (denotado $\lambda \to y_j$)
- Borrar un carácter (denotado $x_i \to \lambda$)
- Sustituir un carácter por otro (denotado $x_i \to y_j$)

Recuerda: *λ denota la cadena vacía.*

Aunque hemos definido la distancia de edición como el número de operaciones de edición, es posible considerar el caso **ponderado** donde las operaciones de edición pueden tener costes diferentes. Por ejemplo, tiene sentido que sustituir una consonante por otra diferente tenga un coste mayor a sustituir una vocal por la misma vocal acentuada. En el caso ponderado el coste de cada operación vendría dado por:

- $\gamma(\lambda \to y_j)$
- $\gamma(x_i \to \lambda)$
- $\gamma(x_i \to y_j)$

Distancias de edición

En este proyecto vamos a limitarnos al caso **no ponderado**. Es decir, aunque se pueda contemplar que algunas operaciones de distancia de edición tienen un coste mayor que otras, el coste de todas las inserciones o borrados es el mismo sin importar el símbolo del alfabeto. Por otra parte, todas las operaciones de sustitución dependen únicamente de si el símbolo que se sustituye es el mismo (un acierto, de coste cero) o es diferente (coste mayor que cero):

- $\gamma(\lambda \to y_j) = 1$
- $\gamma(x_i \to \lambda) = 1$
- $\gamma(x_i \to y_j) = \begin{cases} 0 & \text{si } x_i = y_j \\ 1 & \text{si } x_i \neq y_j \end{cases}$

Distancia de Levenshtein

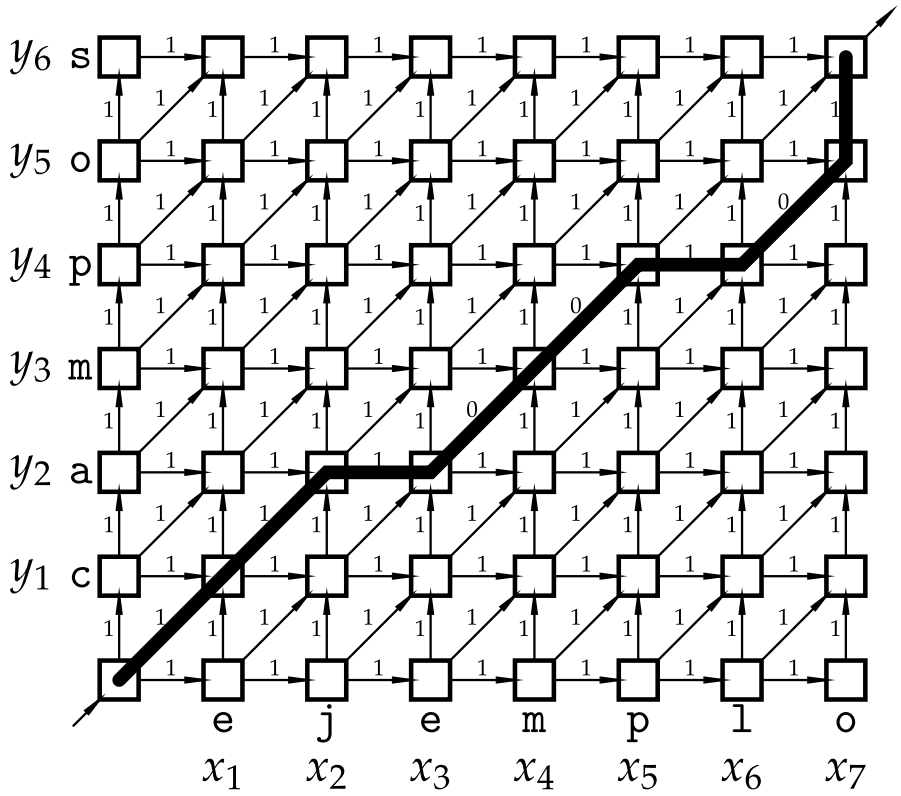
La siguiente ecuación recursiva $D(i,j)$ denota el coste de convertir el prefijo de longitud i de x en el prefijo de longitud j de y:

$$D(i,j) = \text{mín} \begin{cases} 0 & \text{si i=0 } y \text{ j=0} \\ D(i-1,j)+1 & \text{si i>0} \\ D(i,j-1)+1 & \text{si j>0} \\ D(i-1,j-1) & \text{si } i>0, j>0, x_i = y_j \\ D(i-1,j-1)+1 & \text{si } i>0, j>0, x_i \neq y_j \end{cases}$$

De manera que $d(x,y) = D(|x|,|y|)$.

Distancia de Levenshtein

El siguiente gráfico ilustra la matriz de estados cuando se calcula la distancia de Levenshtein entre las cadenas *campos* y *ejemplo*:



Observa que todas las transiciones tienen coste 1 exceptuando las diagonales donde $x_i = y_j$ que tienen coste 0.

Distancia de Damerau-Levenshtein

Se trata de una extensión de la distancia de Levenshtein en la que se añade un nuevo tipo de operación de edición:

- Trasponer o intercambiar dos símbolos adyacentes $ab \leftrightarrow ba$. Con la notación de editar x para obtener y sería $x_{i-1} = y_j$ y $x_i = y_{j-1}$ para $i > 0, j > 0$.

Una observación **obvia** es que, como incluye las operaciones de edición de la distancia de Levenshtein, se cumplirá siempre que:

$$\text{Damerau-Levenshtein}(x, y) \leq \text{Levenshtein}(x, y) \quad \forall x, y \in \Sigma^\star$$

Por ejemplo, la distancia de Levenshtein entre "algoritmo" y "algortimo" es 2 (dos sustituciones "i" $\rightarrow$ "t", "t"$\rightarrow$ "i") mientras que la distancia de Damerau-Levenshtein es 1 (intercambio "it" $\leftrightarrow$ "ti").

Distancia de Damerau-Levenshtein

Existen 2 variantes de esta distancia:

- La versión **no restringida** es la distancia como se entiende de la definición: buscar la forma de aplicar el menor número de operaciones para pasar de una cadena a la otra.
- En la versión **restringida** vamos a suponer que los símbolos utilizados en un intercambio **no** pueden ser utilizados en otras operaciones de edición (antes o después del intercambio).

Resulta que la versión restringida:

1. A veces no da la distancia mínima. Por ejemplo, `d("ba","acb")` es 3 en la versión restringida pero sólo 2 en la no restringida.
2. No cumple la desigualdad triangular, por lo que no es una métrica: Ejemplo: `d("ca","ac") + d("ac","abc") < d("ca","abc")`
3. Es más sencilla y rápida de calcular que la no restringida.

Distancia de Damerau-Levenstein restringida

El cálculo de la versión restringida de Damerau-Levenstein mediante programación dinámica es relativamente sencillo, basta con utilizar la siguiente ecuación recursiva:

$$D(i,j) = \text{mín} \begin{cases} 0 & \text{si i=0 } y \text{ j=0} \\ D(i-1,j)+1 & \text{si i>0} \\ D(i,j-1)+1 & \text{si j>0} \\ D(i-1,j-1) & \text{si } i>0, j>0, x_i = y_j \\ D(i-1,j-1)+1 & \text{si } i>0, j>0, x_i \neq y_j \\ D(i-2,j-2)+1 & \text{si } i>1, j>1, x_{i-1} = y_j, x_i = y_{j-1} \end{cases}$$

Observa que simplemente añade un caso más (última línea) a la ecuación recursiva de la distancia de Levenshtein.

Distancia de Damerau-Levenstein no restringida

En la versión no restringida suponemos un conjunto de operaciones:

$aub \to bva$ *con coste* $1 + |u| + |v|$ *para cualquier* $u, v \in \Sigma^\star$.

ya que para obtener *bva* a partir de *aub* **condicionado a que exista una transposición** $ab \to ba$ deberíamos:

1. Borrar u en aub, lo cual tiene coste $|u|$ borrados,
2. La transposición $ab \to ba$ con coste 1 y, finalmente,
3. Insertar v en ba, con coste $|v|$ inserciones.

Observa que una alternativa a pasar de *aub* a *bva* sin $ab \leftrightarrow ba$ sería:

1. Sustituir $a \to b$ (i.e. $aub \to bub$),
2. Obtener v a partir de u (i.e. $bub \to bvb$),
3. Sustituir $b \to a$ (i.e. $bvb \to bva$).

Esto tiene un coste $2 + d(u, v)$, por lo que sólo conviene transponer si $1 + |u| + |v| < 2 + d(u, v)$ (i.e. si $d(u, v) > |u| + |v| - 1$). El interés de estas operaciones disminuye conforme aumenta $|u| + |v|$.

Es posible implementar la distancia de Damerau-Levenstein no restringida añadiendo un array de longitud $|\Sigma|$, pero resulta complejo.

Distancia de Damerau-Levenstein *intermedia*

A medida que $|u|$ y $|v|$ crecen es más y más improbable que tenga sentido aplicar una transposición $a \leftrightarrow b$ en $aub \rightarrow bva$.
Por eso proponemos una versión llamada *intermedia* que consiste en considerar únicamente los casos donde $|u| + |v| \leq 1$ de modo que únicamente consideraremos las siguientes operaciones de edición donde $a, b, c, d \in \Sigma$:

- $ab \leftrightarrow ba$ coste 1 *(está en la versión restringida)*
- $acb \leftrightarrow ba$ coste 2
- $ab \leftrightarrow bca$ coste 2

No vale la pena contemplar $acb \leftrightarrow bda$ con coste 3 puesto que ese mismo coste se consigue sin trasposición.

Ejercicio:

Añade 2 casos más a la ecuación de recurrencia de Damerau-Levenstein restringido para obtener la versión intermedia.

Tareas a realizar en la parte 1

Tareas a realizar en la parte 1

1. Recuperar la secuencia de operaciones de edición en Levenshtein.
2. Implementar la versión **restringida** de Damerau-Levenstein.
3. Recuperar la secuencia de operaciones de edición en la versión **restringida** de Damerau-Levenstein.
4. Implementar la versión **intermedia** de Damerau-Levenstein.
5. Recuperar la secuencia de operaciones de edición en la versión **intermedia** de Damerau-Levenstein.

Material proporcionado y entrega

- Material proporcionado:
  - Fichero con versión básica de Levenshtein y funciones a completar (fichero `distancias_parte1.py`).
  - Comprobar funciones de distancia (fichero `test_distancias_parte1.py`).
  - Comprobar la recuperación del camino (ficheros `test_edicion.py` y `resultado_test_edicion.py`).
- Entrega:
  - **IMPORTANTE:** Coordinaros para que solamente entregue la tarea uno de los miembros del grupo.
  - La tarea se titula **Proyecto parte 1** y está abierta hasta el 27 de octubre de 2025.
  - No olvidéis poner el nombre de todos los integrantes del equipo/grupo al inicio del fichero `distancias_parte1.py`.
  - Subir únicamente el fichero `distancias_parte1.py` completado.

Material proporcionado

En el fichero `distancias_parte1.py` está la siguiente función:

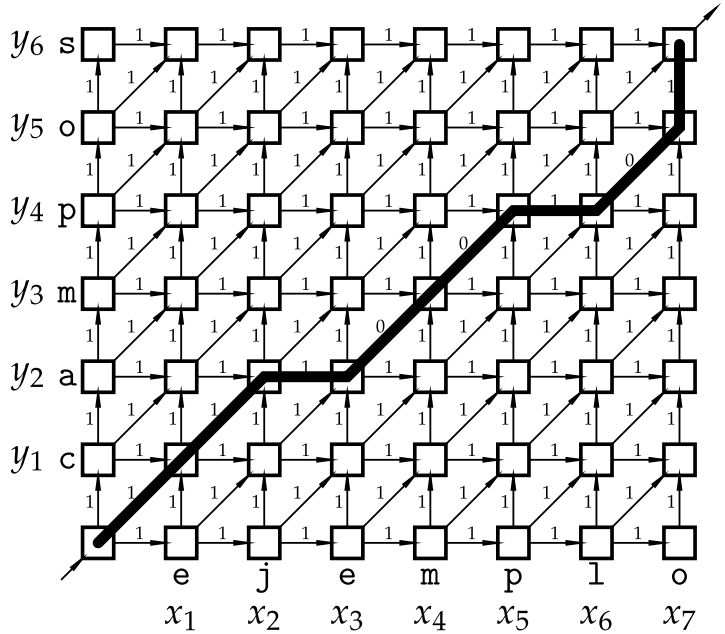
```
def levenshtein_matriz(x, y):
    lenX, lenY = len(x), len(y)
    D = np.zeros((lenX+1,lenY+1), dtype=int)
    for i in range(1, lenX+1):
        D[i][0] = D[i-1][0] + 1
    for j in range(1, lenY+1):
        D[0][j] = D[0][j-1] + 1
        for i in range(1, lenX+1):
            D[i][j] = min(D[i-1][j] + 1,
                          D[i][j-1] + 1,
                          D[i-1][j-1] + (x[i-1] != y[j-1]))
    return D[lenX,lenY]
```

Nota:

En este caso una matriz `numpy` podría resultar algo más lento que utilizar un diccionario Python o una lista de listas Python.

1. Recuperar secuencia de operaciones de edición

Es lo que en programación dinámica se denomina "recuperar el camino":



- **Importante:** tras finalizar el cálculo de la distancia.
- Se parte de la posición `D[lenX,lenY]` y se debe ir retrocediendo.
- Ten en cuenta que al retroceder puedes llegar a "una pared" de la matriz y, en ese caso, hay predecesores que no existen.
- Se obtiene la secuencia de operaciones de edición en sentido inverso. Es más eficiente guardarlos con `append` y hacer un solo `reverse` al finalizar (insertarlos al inicio es más ineficiente).

1. Recuperar secuencia de operaciones de edición

La secuencia devuelta debe estar en el formato utilizado por la siguiente función que convierte una cadena en otra utilizando esa secuencia de operaciones. Aquí tienes un ejemplo de cómo convertir la cadena `'ejemplo'` en la cadena `'campos'` *(es el mismo ejemplo del libro de los apuntes de Andrés Marzal, María José Castro y Pablo Aibar):*

```
aplicar_edicion("ejemplo",
                [('e', ''), ('j', 'c'), ('e', 'a'), ('m', 'm'),
                 ('p', 'p'), ('l', 'o'), ('o', 's')])
```

La edición para Levenshtein es una lista de tuplas, cada tupla son 2 letras donde una de ellas puede ser la cadena vacía si se trata de una inserción o de un borrado. Los aciertos también aparecen explícitamente en la secuencia.

1. Recuperar secuencia de operaciones de edición

El fichero `test_edicion.py` contiene código para lanzar las funciones que calculan operaciones de edición sobre una batería de tests. El resultado tiene este aspecto:

```
Comprobando si levenshtein(ejemplo,campos) == 5
operaciones: [('e', ''), ('j', 'c'), ('e', 'a'), ('m', 'm'),
 ('p', 'p'), ('l', 'o'), ('o', 's')]
 - e       se aplica en |ejemplo para dar |jemplo coste 1
 - j   c   se aplica en |jemplo  para dar c|emplo coste 1
 - e   a   se aplica en c|emplo  para dar ca|mplo coste 1
 - m   m   se aplica en ca|mplo  para dar cam|plo coste 0
 - p   p   se aplica en cam|plo  para dar camp|lo coste 0
 - l   o   se aplica en camp|lo  para dar campo|o coste 1
 - o   s   se aplica en campo|o  para dar campos| coste 1
CORRECTO!
```

2. Damerau-Levenshtein *restringido*

Esta actividad consiste en copiar el código de Levenshtein y modificarlo para dar cuenta de la nueva regla que aparece en la ecuación recursiva (la última línea):

$$D(i,j) = \text{mín} \begin{cases} 0 & \text{si i=0 } y \text{ j=0} \\ D(i-1,j)+1 & \text{si i>0} \\ D(i,j-1)+1 & \text{si j>0} \\ D(i-1,j-1) & \text{si } i>0, j>0, x_i = y_j \\ D(i-1,j-1)+1 & \text{si } i>0, j>0, x_i \neq y_j \\ D(i-2,j-2)+1 & \text{si } i>1, j>1, x_{i-1} = y_j, x_i = y_{j-1} \end{cases}$$

se sugiere tomar como punto de partida la versión de Levenshtein con parada por `threshold` y reducción de coste espacial que utiliza 2 vectores columna. Ahora tendrás que utilizar TRES vectores porque aparece una dependencia `j-2`.

3. Camino en Damerau-Levenshtein *restringido*

Para recuperar el camino, hay que:

- Ver cómo especificar las operaciones de edición de tipo transposición $ab \rightarrow ba$ en la secuencia devuelta. Se trata de generar tuplas `('ab','ba')` donde las letras obviamente serán las adecuadas según el caso.
- Probar que funciona con el fichero `test_edicion.py`.

```
Comprobando si damerau_r(algortimac,algoritmica) == 3
operaciones: [('a', 'a'), ('l', 'l'), ('g', 'g'), ('o', 'o'),
 ('r', 'r'), ('', 'i'), ('t', 't'), ('im', 'mi'), ('ac', 'ca')]
 - a   a   se aplica en |algortimac para dar a|lgortimac coste 0
 - l   l   se aplica en a|lgortimac para dar al|gortimac coste 0
 - g   g   se aplica en al|gortimac para dar alg|ortimac coste 0
 - o   o   se aplica en alg|ortimac para dar algo|rtimac coste 0
 - r   r   se aplica en algo|rtimac para dar algor|timac coste 0
 -     i   se aplica en algor|timac para dar algori|timac coste 1
 - t   t   se aplica en algori|timac para dar algorit|imac coste 0
 - im  mi  se aplica en algorit|imac para dar algoritmi|ac coste 1
 - ac  ca  se aplica en algoritmi|ac para dar algoritmica| coste 1
CORRECTO!
```

4. Damerau-Levenshtein *intermedio*

En este caso se tienen en cuenta una serie de operaciones de transposición descritas anteriormente:

- $ab \leftrightarrow ba$ coste 1 *(ya estaba en restringido)*
- $acb \leftrightarrow ba$ coste 2
- $ab \leftrightarrow bca$ coste 2

donde $a, b, c, d \in \Sigma$. Para ello:

- Inspirándote en la ecuación de recurrencia de la versión restringida y mirando las nuevas operaciones de edición, escribid la ecuación de recurrencia para esta versión.
- Puedes tomar como punto de partida para la implementación la versión Damerau-Levenshtein restringida (seguramente nadie implementará intermedio sin haber hecho restringida) y añadir los nuevos casos que han aparecido en la ecuación de recurrencia.
- Seguramente ahora has de utilizar 4 en lugar de 3 vectores columna.

5. Camino en Damerau-Levenshtein *intermedio*

- Para recuperar la secuencia de operaciones de edición requiere considerar tuplas de tipo `('acb','ba')` y `('ab','bca')`.

```
Comprobando si damerau_i(algoritmo,algortximo) == 2
operaciones: [('a', 'a'), ('l', 'l'), ('g', 'g'), ('o', 'o'),
  ('r', 'r'), ('it', 'txi'), ('m', 'm'), ('o', 'o')]
 - a   a   se aplica en |algoritmo para dar a|lgoritmo coste 0
 - l   l   se aplica en a|lgoritmo para dar al|goritmo coste 0
 - g   g   se aplica en al|goritmo para dar alg|oritmo coste 0
 - o   o   se aplica en alg|oritmo para dar algo|ritmo coste 0
 - r   r   se aplica en algo|ritmo para dar algor|itmo coste 0
 - it  txi se aplica en algor|itmo para dar algortxi|mo coste 2
 - m   m   se aplica en algortxi|mo para dar algortxim|o coste 0
 - o   o   se aplica en algortxim|o para dar algortximo| coste 0
CORRECTO!
```

Tareas a realizar en la parte 2

Tareas a realizar en la parte 2

6. Levenshtein con reducción de coste espacial.
7. Añadir a Levenshtein con reducción de coste espacial un parámetro umbral o `threshold` de modo que se pueda dejar de calcular cualquier distancia mayor a dicho umbral.
8. Cota optimista para ahorrar evaluaciones de distancia Levenshtein e integrarlo en `SpellSuggester` para utilizarlo en el recuperador de SAR.
9. Implementar la versión **restringida** de Damerau-Levenstein con reducción del coste espacial y con un parámetro umbral o `threshold` (para dejar de calcular distancias mayor a dicho umbral).
10. Implementar la versión **intermedia** de Damerau-Levenstein con reducción del coste espacial y con un parámetro umbral o `threshold` (para dejar de calcular distancias mayor a dicho umbral).
11. Completar el método `suggest` en la clase `SpellSuggester`.
12. Modificar el Indexador y el Recuperador de SAR para permitir la búsqueda aproximada de cadenas con la clase `SpellSuggester`.

Material proporcionado

- Fichero con nuevas funciones de distancia a completar (fichero `distancias_parte2.py`).
- Comprobar funciones de distancia (fichero `test_distancias_parte2.py`).
- La clase `SpellSuggester` (fichero `spellsuggester.py`).
- Comprobación de las distancias con `SpellSuggester` (ficheros `test_spellsuggester.py` y `resultado_test_spellsuggester.py`).
- El `pdf` del trabajo de SAR del curso anterior por si alguien quiere consultarlo y no lo tiene a mano.
- Versiones actualizadas `SAR_Indexer.py` y `ALT_Searcher.py`.
- Ficheros `'100.zip'` y `'500.zip'`, `'queries.txt'` y ficheros de referencia para comprobar que la integración con el proyecto de SAR es correcta.

Atención

Si alguien no cursó SAR en los 2 últimos cursos debe utilizar los ficheros de *SAR_BASICO* disponibles en PoliformaT.

6. Reducción del coste espacial

Para reducir el coste espacial es habitual reemplazar la matriz por dos vectores. A partir del código de `levenshtein_matriz` reemplaza la matriz:

```
D = np.zeros((lenX+1,lenY+1), dtype=int)
```

por 2 vectores (es preferible numpy, alternativamente listas Python).

- La función `np.zeros` también puede crear un vector si solamente le pasas un tamaño:

```
>>> np.zeros(10,dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

- Conviene que sean vectores de tamaño `lenX + 1` porque corresponde al bucle más interno.
- La forma *eficiente* de intercambiar los 2 vectores es intercambiar las *referencias* a los mismo:

```
vprev,vcurrent = vcurrent,vprev
```

Atención

Esta parte corresponde a completar la función `levenshtein_reduccion`.

7. Utilización de un parámetro umbral o `threshold`

- Esta parte se debe realizar a partir de la versión con reducción del coste espacial.
- Se trata de añadir un tercer argumento `threshold` de modo que se pueda dejar de calcular una distancia cuando sepamos seguro que el valor final superará dicho umbral.
- **Importante:** Si la distancia calculada es mayor a `threshold` el método deberá devolver `threshold+1` (incluso al final).
- La forma más sencilla consiste en **detener el algoritmo** (vía `return`) si, tras calcular una etapa (columna) se puede asegurar que el coste superará el umbral.

Atención

Esta parte corresponde a completar la función `levenshtein`. Copia primero el código la versión anterior (`levenshtein_reduccion`) y luego añade la modificación para la parada con `threshold`.

8. Cota optimista

Se trata de utilizar una cota optimista para ahorrar evaluaciones de la distancia de Levenshtein (**no sirve** para Damerau-Levenshtein).
La cota optimista propuesta supone olvidar el orden en el que aparecen las letras (como ocurre también con los modelos *bag of words* que puede que vieras en SAR). Veamos unos ejemplos:

- Si se compara `'casa'` con `'saca'`, el nº veces que aparece cada letra es el mismo, así que una cota basada en el conteo devolvería 0.
- Si se compara `'casa'` con `'saco'`, vemos que hay una `'a'` de más en la primera y una `'o'` de más en la segunda. Una cota optimista basándonos en el conteo (ignorando el orden) sólo podría concluir que la distancia es `>=1` por una posible sustitución de la `'a'` por la `'o'`.
- Si se compara `'casa'` con `'abad`, vemos que hay una `'c'` y una `'s'` de más en la primera y una `'b'` y una `'d'` de más en la segunda. Se puede deducir que la distancia es `>=2` para dar cuenta con sustituciones o borrados las letras de más de uno de los lado (interesa el máximo entre ambos, aquí había empate).

8. Cota optimista

- En general, hay que:
  - contar nº veces **de más** que aparece cada letra de una cadena en la otra.
  - sumar todas esas cuentas
  - hacer lo mismo con la otra cadena.
  - devolver el máximo de ambas sumas.

  Por ejemplo, si nos pasan las cadenas `'casa'` y `'abad'` y recorremos la primera sumando 1 obtenemos el diccionario:

  ```
  {'c': 1, 'a': 2, 's': 1}
  ```

  Si ahora recorremos la segunda cadena restando 1 nos queda:

  ```
  {'c': 1, 'a': 0, 's': 1, 'b': -1, 'd': -1}
  ```

  La suma de valores positivos da 2, la de los negativos da -2. El máximo de ambos en valor absoluto es 2 que sería la estimación optimista de `Levenshtein('casa','abad')`. Donde, por *optimista* se entiende que es menor o igual que la distancia real.

8. Cota optimista

- El objetivo de esta actividad es crear una función `levenshtein_cota_optimista` que internamente realice este cálculo y, si el resultado es mayor al `threshold`, devuelva `threshold+1`. En otro caso ya calcula la distancia de Levenshtein de la forma normal (la versión que puede parar por el `threshold` tras calcular cada columna).
- La forma de integrar esta actividad en el código de SAR es muy sencilla: al construir el objeto `SpellSuggester` le pasamos en el diccionario de funciones de distancia una entrada con el nombre (por ejemplo) `'levenshtein_o'` asociada a la función que pruebe la cota optimista y, si no supera el threshold, delegue en la versión "Levenshtein con threshold" anterior (llamada `levenshtein`).

9. Reducción del coste espacial de la versión restringida de Damerau-Levenstein

Se trata de adaptar la implementación de la versión **restringida** de Damerau-Levenstein realizada en la parte anterior:

- La reducción del coste espacial es similar al punto 6. Ten en cuenta que ahora no bastarán dos vectores porque hay una dependencia a la antepenúltima columna.
- Incluir el parámetro threshold es similar al punto 7.

10. Reducción del coste espacial de la versión intermedia de Damerau-Levenstein

Se trata de adaptar la implementación de la versión **intermedia** de Damerau-Levenstein realizada en la parte anterior. Es una actividad muy parecida a la del punto anterior (hacer exactamente lo mismo para la versión **restringida**).

11. Completar el método `suggest` en `SpellSuggester`

```
def suggest(self,term,distance,threshold,flatten=True):
    """
    Args:
        term (str): término de búsqueda.
        distance (str): nombre del alg. de búsqueda a utilizar
        threshold (int): threshold para limitar la búsqueda
    """
    if distance is None:
        distance = self.default_distance
    if threshold is None:
        threshold = self.default_threshold

    ##########################################
    # COMPLETAR
    ##########################################
    resul = []
    if flatten:
        resul = [word for wlist in resul for word in wlist]
    return resul
```

12. Integración del corrector en el proyecto SAR

En el proyecto de SAR se proporcionaban 3 ficheros (`SAR_Indexer.py`, `ALT_Searcher.py` y `SAR_lib.py`) y únicamente se debía modificar `SAR_lib.py`.

- Debéis bajaros un nuevo `SAR_Indexer.py`.
- Ahora se proporciona un fichero `ALT_Searcher` que es una variante de `SAR_Searcher` donde se han añadido esas 3 opciones de la línea de comandos:

  ```
  -d --distance nombre_funcion_distancia
  -t --threshold umbral_distancia
  -s --spell
  ```

  para poder especificar los valores del tipo de distancia a utilizar y el threshold empleado en el `SpellSuggester` (pasándole estos valores en el constructor) y `--spell` para activar la búsqueda con tolerancia.
  Los valores de `-d` y `-t` se usan para llamar al método `suggest` (pero es posible usar el programa **sin** tolerancia (sin usar `suggest`)).
- En la clase `SAR_Indexer` dentro de `SAR_lib.py` debes añadir los atributos `self.use_spelling` y `self.speller` a `False` y `None` respectivamente.

12. Integración del corrector en el proyecto SAR

- También debes añadir el siguiente método que se llama desde `ALT_Searcher`:

```
def set_spelling(self, use_spelling:bool, distance:str=None,
                 threshold:int=None):
    """
    self.use_spelling a True activa la corrección ortográfica
    EN LAS PALABRAS NO ENCONTRADAS, en caso contrario NO utilizará
    corrección ortográfica

    input: "use_spell" booleano, determina el uso del corrector.
           "distance" cadena, nombre de la función de distancia.
           "threshold" entero, umbral del corrector
    """
```

Este método debe asignar a `self.speller` un objeto `SpellSuggester` creado con los valores recibidos (el vocabulario son las claves del `self.index` convertidas a lista, el diccionario de tipos de distancia es `opcionesSpell` que puedes importar del fichero `distancias_parte2.py`).

12. Integración del corrector en el proyecto SAR

Es posible que sea casi suficiente modificar el método `get_posting` para que realice la búsqueda con tolerancia.
Para ello, **únicamente si se activa `use_spell` y el término no está en `self.index`** hay que usar `self.speller` y utilizar `suggest` con dicho término. Si devuelve una lista no vacía de palabras, hay que buscarlas todas (el `get_posting` original solamente tendría que buscar una) y juntar todos los términos invertidos en una sola lista (como se hacía ya cuando se trataba la conectiva lógica `or`). Una vez obtenido el *posting list* juntando todo, lo podemos utilizar como antes y da igual si proviene de un término que estaba en el diccionario o del resultado de usar el corrector ortográfico (es transparente al resto).

- Os hemos dejado los ficheros `'100.zip'` y `'500.zip'`, `'queries.txt'` y ficheros de referencia.
- Podéis utilizar las referencias como en este ejemplo:

```
python ALT_Searcher.py index_100.bin \
       -T result_100_levenshtein_1.txt -s -t 1 -d "levenshtein"
```

Entrega y evaluación

Material a subir en la tarea del proyecto parte 2

- La tarea **Proyecto parte 2** finaliza el 7 noviembre 2025.
- Nuevamente: coordinaros para que solamente suba la tarea un miembro del grupo.
- Cada grupo/equipo debe preparar y subir estos ficheros:
  - Fichero `memoria.pdf`: Es una pequeña **memoria** o informe (bastan 3 a 5 páginas) con una descripción de las tareas desarrolladas, el *reparto de trabajo y decisiones adoptadas*.
  - Fichero `distancias_parte2.py`.
  - Fichero `spellsuggester.py`.
  - Fichero `SAR_lib.py`.
  - Según las actividades realizadas, scripts para medir distancias, resultados, etc.
- Se pueden subir juntos en un zip (no se admite formato rar).
- No olvidéis poner el nombre de todos los integrantes en la memoria y al inicio de `distancias_parte2.py`.
- NO subáis los ficheros que no se han modificado (especialmente NO subáis las carpetas de datos).

Evaluación

- Las dos entregas valen lo mismo (50% cada una).
- La sesión del 14 de noviembre de 2025 se dedica a evaluar (haremos preguntas y probaremos el código).
- Se valorará la organización del código realizado (incluyendo la utilización de módulos/bibliotecas, estilo, eficiencia, documentación/comentarios, etc.).
- Recuperación del proyecto:
  - Para poder recuperar el proyecto es necesario haberlo presentado primero en la fecha estipulada.
  - En el caso de suspenso, os daremos una lista con lo que es necesario subsanar y el plazo para la entrega.