# Programación II - Trabajo Práctico Final

Bertoni Juan Ignacio

# Idea

El objetivo del programa es utilizar la información recopilada por la lectura de los archivos dados como dato para conjeturar una predicción acerca de cuál palabra es la más adecuada para completar las distintas frases a ser completadas.

Mi idea fue dividir la búsqueda en tres posibles casos: primeramente, intento buscar "coincidencias exactas", basándome en los pares de palabras (tanto por izquierda como por derecha) que están al lado de la palabra que se quiere averiguar. De esta manera, si una oración incompleta fue sacada directamente de los textos de ejemplo, es muy probable que sea encontrada.

No obstante, existen frases a completar que no cumplen estas características, entonces se va al segundo caso: búsqueda por frecuencia. Cada palabra tiene asociada un grupo de palabras que aparecen a su lado a lo largo del texto que se tiene como dato (grupos de palabras tanto a la izquierda como a la derecha). Asimismo, cada una de las palabras de estos grupos tiene asignada una frecuencia. Entonces primero se intenta buscar la palabra mas frecuente por la izquierda y, en caso de que ninguna palabra se frecuente más de una vez, se busca la palabra mas frecuente por la derecha. Esto se decidió así ya que, me di cuenta mientras hacía algunas pruebas, que los resultados eran mejores priorizando las elecciones por la izquierda (aunque claramente si una palabra es mucho mas frecuente por derecha, y por izquierda o hay ninguna que aparezca más de una vez, se elegirá la derecha)

Los dos casos de arriba cubren la gran mayoría de las posibles situaciones, pero si existiera una oración a completar tal que no fuera cubierta por ninguno de los casos de arriba, para no dejar el espacio vacío se busca una palabra aleatoria del texto para rellenar. No es lo más elegante, pero dada mi implementación, me pareció que dejarlo así era lo más adecuado.

# Estructuras de datos empleadas

Para el primer caso, usé las siguiente estructura:

`dictsBigramas: (bigramasI, bigramasD)`

De donde:

```
bigramasI: {(palIzq2, palIzq1): set(pal)}
bigramasD: {(palDer1, palDer2): set(pal)}
```

Donde palIzq1, palIzq2, palDer1, palDer2 y pal son strings (str).

dictsBigramas es una tupla que contiene a los diccionarios bigramasI y bigramasD. El diccionario bigramasI se trata de un diccionario cuya claves son tuplas de la forma (palIzq2, palIzq1) donde palIzq2 es la string que se encuentra 2 palabras a la izquierda de pal y palIzq1 la string que se encuentra exactamente a la izquierda de pal, siendo pal un elemento del conjunto de valores asociados a cada clave. Análogamente funciona el diccionario bigramasD, pero hacia la derecha. En la parte de tests se pueden ver ejemplos del funcionamiento.

Decidí el uso de esta estructura ya que, conociendo las dos palabras anteriores (posteriores) a una cierta palabra a averiguar, se puede hacer una búsqueda eficiente en los diccionarios (respectivamente por izquierda o por derecha) de las claves a través del operador **in**, además del indexado por clave de los diccionarios, otro de sus puntos fuertes. Con la búsqueda de dicha clave, podemos acceder a su valor asociado. El valor asociado será un conjunto de palabras acordes a las características buscadas. Decidí que sea un conjunto ya que no interesa el órden de las palabras adecuadas ni la repetición de ellas. De esta manera podemos encontrar matches exactos en las frases a predecir, que se hallan en los textos conocidos.

Para el segundo caso, use la estructura:

```
dictFrecuencias: {str: (dictI, dictD)}
dictI: {str: int}
dictD: {str: int}
```

Diccionario en el cual las claves son palabras, y el valor asociado a cada una de las claves son tuplas de la forma (dictI, dictD) donde el dictI corresponde a un diccionario donde las claves son palabras a la izquierda de la palabra original y sus valores asociados son numeros enteros correspondientes a la cantidad de veces que las palabras aparecen juntas (en ese orden) en el texto. DictD se comporta de manera analoga pero almacenando las palabras a la derecha de la palabra original. En la parte de tests hay ejemplos del funcionamiento.

Decidí el uso de estructura nuevamente pensando en las búsquedas de claves de diccionario a través de **in** y el indexado por clave de los diccionarios. En este caso, conociendo la palabra anterior (posterior), podremos buscar en las claves del diccionario dicha cadena, para encontrar la palabra con mayor frecuencia asociada. En este caso sí se recorren los diccionarios dictI (dictD) a través de bucles, pero estos van a ser de un tamaño considerablemente menor al del diccionario dictFrecuencias, por lo cual la eficiencia no varía tanto a que si se usara in (que, para obtener la mayor frecuencia, era complicado).

# Estructura del proyecto

Las carpetas son las siguientes:

- Entradas: donde se almacenan los textos de los artistas de la forma **artista**.txt en un formato ya compatible para que el programa en python los lea.
- Frases: donde se encuentran los archivos también de la forma **artista**.txt de extractos de canciones de artistas incompletas. Otra de las carpetas que el programa en python debe leer.
- Salidas: donde se encuentran las mismas oraciones del respectivo artista de la carpeta Frases, pero con las frases completadas, reemplazando el caracter '_' por una palabra acorde.
- Tests: esta carpeta fue creada por mí. Acá se encuentran los archivos de texto/carpetas utilizados en los programas de testeo.
- Textos: donde se encuentran carpetas con los respectivos nombres de los artistas. Cada carpeta contiene canciones escritas por cada artista, en formato de texto (pudiendo contener saltos de línea, coma, punto y coma, etc).

Después se encuentran los archivos sueltos: **main.c** y **main.py** correspondientes a los respectivos programas en C y en python; y **archivos.txt**, utilizado por el programa en C para detectar los archivos presentes dentro de un directorio específico. También están los archivos de testeo **tests.c** y **tests.py**.

Después está el archivo de encabezado **funciones.h** que contiene los prototipos de las funciones usadas en el programa en C, junto con la inclusión de librerías y declaración de constantes. Después, las funciones son implementadas en **funciones.c**. Decidí separar el código en C de esta manera, ya que me pareció la manera más sencilla para hacer tests después, ya que el archivo de **tests.c** y **main.c** no dependen entre sí, sino que ambos dependen de **funciones.h**.

# Correr el programa

## Compilar y correr

Para compilar el programa en C, usando gcc:

```
gcc funciones.c main.c -Wall -gstabs -o main
```

Y como el código en C llama internamente al programa en Python, bastará con correr el ejecutable generado:

```
./main Nombre_Artista
```

## Correr los tests

Para correr los tests en C:

```
gcc funciones.c tests.c -gstabs -Wall -o tests
```

```
./tests
```

Para correr los tests en Python, se necesita tener la utilidad pytest.

```
pip install pytest
```

Y lo corremos:

```
python3 -m pytest tests.py
```