# INFORME: SARS CoV-2 Challenge (Drapythron)

Los autores del SARS CoV-2 Challenge y miembros del grupo Drapythron somos:

- Sergio Beltrán Guerrero
- Jaume Guasch Llobera
- Martí Serratosa Sedó
- Sebastian Bampton Blasco

El programa utilizado para programar el código han sido PyCharm y Visual Studio Code donde hemos trabajado con clases. Hay clases que corresponden exactamente con partes de la práctica, otras,  tienen más de una clase. Para el redactado en markdown hemos utilizado Typora, Visual Studio Code y Github tanto en el informe como el readme.

**Tabla de contenidos:**



# 1-Introducción

Hoy en día, unos de los grandes objetivos a nivel internacional de la comunidad científica es hallar una vacuna para el virus SARS-CoV-2. Uno de los problemas más significantes del coronavirus, es su rápida mutación. Expertos en computación quieren clasificar las diferentes muestras de ARN para poder crear un árbol genealógico.

**El objetivo de esta práctica, reto (challenge) es clasificar las diferentes muestras del virus que hay en el mundo y mostrar su árbol genealógico.**

La ejecución del programa se realiza en el sarscovhierarchy.py amará otras clases para poder realizar las funciones deseadas. El orden de uso de la diferentes clases es el siguiente:

1. LecturaCSV
2. UrlSequence
3. NeedlemanWunsch / NeedlemanWunschRust
4. Clustering

# 2-Pre-procesamiento

La primera parte de la práctica consiste en la lectura de un archivo csv con las secuencias para poder calcular la mediana de cada país. Esta parte se ha realizado íntegramente en **la clase *LecturaCSV***.

## Función newLocation

En esta función lo único que hace es renombrar el nombre del país, dividiendo el nombre del país cuando nos encontramos con ':'. Ejemplo: `"Spain:Valencia".split(':') --> Spain`

## Lectura del archivo *sequences.csv*

Primero de todo abrimos el archivo .csv para poder leer las secuencias de los países.

Luego leeremos y almacenaremos línea a línea los datos más relevantes: accession, length y geoLocation.

El coste de este proceso es **O(n)**, donde n es el número de líneas del archivo csv.

Seguidamente realizaremos la mediana de estos países. Para realizarla, iniciaremos un bucle con todos los países. En este bucle tenemos la función mediana, que nos devolverá la tupla "mediana" de el país.

Este proceso tiene un coste de **O(n)**, donde n es el número de países.

## Método mediana(geoLocation)

En este método, analizaremos todas las tuplas de la secuencia. Iremos comparando una a una con la geoLocation pasada. Si la tupla coincide, la almacenaremos la tupla en la lista 'analysisList' y el length en 'lengths'. Una vez hemos comparado todas las tuplas de la secuencia, realizaremos la mediana de la lista 'idems'.

Una vez tenemos el resultado tenemos la mediana, podemos realizar dos opciones:

1. Sí la mediana esta contenida en la lista, devolveremos la posición de la mediana en la lista.
2. Sí la mediana no esta contenida en la lista, buscaremos el valor que más se acerque a la mediana con el siguiente código:

```
result = analysisList[min(range(len(idems)), key = lambda i: abs(idems[i] - median))]
```

Una vez tenemos la posición de la mediana en la lista, devolveremos la tupla en la posición de 'analysisList'

Este proceso tiene un coste de ***O(n)***, donde n es el tamaño de 'sequences'.

En definitiva, el proceso de lectura del archivo csv y la devolución de la mediana de cada país tiene un coste de $O(n^3)$.

## Función median

Esta función nos sirve para encontrar la mediana de los valores de la lista. La mediana devuelve el valor que se encuentra en medio de la lista.

Para ello primero tendremos que ordenar la lista, nosotros hemos utilizamos el método de *Divide y Vencerás* **Merge Sort** (explicado en el siguiente apartado).

Según la longitud de la lista (si finaliza en impar o par) la mediana puede encontrarse en un único número (impar), o entre dos posiciones (par).

Si la longitud de la lista es impar, significa que podremos encontrar la mediana en un único resultado, el valor del medio al tener la lista ordenada. Por el otro lado, en caso de longitud de la lista par, la mediana la obtendremos entre dos números, el de delante y atrás del valor del medio, y por tanto tendremos que hacer la media de los valores en conflicto. El coste de este proceso es **O(1)**, ya que solo contiene un if-else.

## Función mergeSort

En la función mergeSort pretendemos ordenar las longitudes de las secuencias de menor a mayor en una lista, siendo el objetivo final poder obtener el elemento que forma la mediana de la lista.

El algoritmo empleado es del tipo "**divide y vencerás**", con un coste de **O(n·log(n))** y se emplea de forma recursiva.

En el algoritmo dividimos la lista proporcionada en dos partes, siendo la única condición que la longitud de las listas a dividir sea mayor que dos. El objetivo ahora es seguir dividiendo las dos listas obtenidas sucesivamente hasta obtener listas con elementos individuales o pares.

Una vez hemos obtenido listas formadas por uno/dos elemento(s), compararemos los valores y ordenaremos las listas, una vez realizada la ordenación de los elementos en las listas, concatenamos las listas para reducir todos los elementos a una sola lista ordenada, la cual devolvemos.

# 3-Alineamiento secuencial

La segunda parte de la práctica consiste en realizar dos tareas.

- La primera tarea es la lectura del ARN de cada nucleotido de la mediana del país encontrada en la primera parte. Para realizar esta tarea usaremos la **Clase UrlSequence**.
- La segunda tarea consiste en alinear las secuencias de ARN y compararlas. Esta tarea la hemos implementado de dos formas. Usando el lenguaje de programación Python y usando el lenguaje de programación Rust.
    - La **Clase NeedlemanWunsch** es la clase que hemos implementado en Python.
    - La **Clase NeedlemanWunschRust** es la clase que hemos implementado en Rust.

## UrlSequence

La clase UrlSequence **obtiene los datos ARN de la base de datos de la NCBI cuando le pasamos un *accession***.

Para proceder a la obtención de la secuencia ARN primero de todo usando el *accession* descargaremos la secuencia ARN en formato .fasta. Para conseguirlo usamos el método `urllib.request.urlretrieve(url, 'sequencesFASTA.fast')` que nos creará un fichero temporal con los datos de la secuencia.

A continuación, procederemos a su lectura. Iremos recorriendo todo el fichero con *.readLine()* borrando los espacios y saltos de linea. Las secuencias ya listas las añadiremos en una lista junto a accession, length y geolocation. Tal que así **['Accession', 'Length', 'GeoLocation', 'ARN']**.

El coste de esta clase es de **O(n)**, n es el número de lineas de cada secuencia.

## NeedlemanWunsch

Este algoritmo nos ayudará a obtener una puntuación en base al alineamiento de dos códigos genéticos de distintos países.
El alineamiento de secuencias es una forma de comparar dos secuencias haciendo hincapié en las zonas donde hay similitudes.
NeedlemanWunsch es el algoritmo más utilizado para comparar código genético.
Se trata de un algoritmo de orden ***O(n·m)***.

Como ya hemos comentado anteriormente, hemos realizado dos implementaciónes de este algoritmo.

Primero de todo, vamos a aclarar las cosas que hay en común en las dos implementaciones.

- Los pesos para comparar son: **gap** = -1, **match** = 1 y **mismatch** = -1.
- Hemos remplazado los ácidos nucleicos que no son estándar por los estándar. El criterio que hemos seguido es el siguiente, hay ácidos nucleicos que pueden significar dos o más ácidos, por lo tanto, hemos escogido uno de los ácidos que puede significar. La siguiente tabla lo deja todo claro.

| Código de ácido nucleico | Significado | Ácido escogido |
|---|---|---|
| N | A, G, C, T | A |
| B | G, T, C | G |
| Y | T, C | T |

## Needleman-Wunsch en Python

Para realizar esta implementación nos hemos basado en el pseudo-código de Wikipedia. El código lo hemos dividido en tres partes, la primera que nos inicia la matriz con sus pesos. Una vez tenemos la matriz creada, llamamos al método `__alignment`, para que nos realice la alineación de las secuencias usando programación dinámica. Una vez acaba el algoritmo obtenemos dos String, que estos son las secuencias alineadas. Y finalmente, para obtener la puntuación, compararemos uno a uno los ácidos de las dos secuencias, para devolver su puntuación.
A la hora de puntuar, hemos decidido que el rango de puntuación sea de [0, 100], 0 significara que son iguales y 100 que son totalmente distintas. El criterio que hemos utilizado para devolver esta puntuación es el siguiente:

1. Como hemos escogido que, gap = -1, match = 1 y mismatch = -1, tendremos que el rango de puntuación seá [-len(secuencia_alienada), len(secuencia_alineada)].
2. Desplazaremos a la derecha el rango para que todos los valores sean positivos, [0, len(secuencia_alineada)*2] y también desplazaremos la puntuación obtenida tal que así: `puntuación + len(secuencia_alineada)`.
3. Realizaremos una simple regla de 3.

```
(puntuación_desplazada / len(secuencia_alineada)*2) * 100 = nueva_puntuación

nueva_puntuación = 100 - nueva_puntuación
```

## Needleman-Wunsch en Rust

Para mejorar la velocidad del algoritmo hemos implementado el algoritmo anterior en el lenguaje de Programación Rust. Hemos creado una **librería** en Rust con las funciones de alinear y puntuación. Una vez creada la hemos compilado y importado a nuestro código en Python.
A la hora de implementar el código en Rust, hemos importado una librería que se llama *Simple_Matrix*, la cual nos crea una matriz de enteros, con la que trabajaremos. Las demás implementaciones son igual que en python pero traducido a Rust.

# 4-Clustering

En la **clase clustering** realizaremos una clasificación de los países según sus puntuaciones.

Primero de todo, dejar claro que la función principal de esta clase es recursiva. Para realizar esta clasificación, primero de todo, compararemos los centros actuales con los anteriores. Sí los dos centros son iguales significará que ya no se puede clasificar más esta lista de países.

En cuanto al algoritmo de clasificación lo podemos dividir en 2 partes:

1. Búsqueda de los clusters, en este proceso compararemos todas las muestras con los centros y la muestra que sea mínima la añadiremos al cluster del centro. Ejemplo aclaratorio:

   | | **0** | **1** | **2** |
   |---|---|---|---|
   | **0** | 0 | 4 | 7 |
   | **1** | 1 | 0 | 3 |
   | **2** | 4 | 3 | 0 |

   Centros = 0, 1

   Cluster = [[0], [1]]

   El cluster inicial esta formado por los centros ya que representa que son idénticos y luego buscamos el peso mínimo entre las muestras que faltan. En nuestro ejemplo solo falta comprara la muestra 2.

   min(02 = 4 y 12 = 3) --> res 12 = 3

   **Cluster = [[0], [1, 2]]**

2. Una vez tenemos los clusters, buscamos un nuevo centro, si existe. En cada cluster realizaremos la suma de cada muestra y la que sea menor será el nuevo centro, ya que representa que es el que esta más al centro.

   En el ejemplo anterior ([1,2]), la suma de 1 es 7 y la suma de 2 es 10, por lo tanto el centro sigue siendo 1 ya que 7 es menor que 10.

El coste de esta clase es de $\mathbf{O(n + m \cdot n + n \cdot p^2)}$, donde n es el número de centros que siempre es igual a k, m el número de muestras y p el número de muestras en cada cluster.

# 5-Árbol de Clusters

Para que podamos apreciar mejor el resultado del programa, hemos creado una clase que construye un árbol cuyas ramas son los clusters. Esta sería una pequeña demostración del árbol.

ADJUNTAR IMAGEN

El coste de esta clase es de $\mathbf{O(n \cdot m)}$, donde n es el numero de clusters y m es el número de países en cada cluster. La m ira variando según los países que pueda tener cada cluster.

# 6-Tests

A la hora de realizar los test del programa, solo hemos podido realizar los de NeedlemanWunsch, ya que, es el único algoritmo que podemos acceder y sabemos lo que devolverá al 100%. En la clase de Lectura CSV no hemos podido acceder ya que necesitábamos pasar un archivo csv. Y en la clase clustering no podemos saber con certeza cual sera el cluster que nos devolverá el programa, ya que, los primero centros son aleatorios.

---