UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2023/2024

Guion de prácticas

Kmer5

Práctica final

Junio de 2024

Índice

1. Definición del problema

Por fin, con esta práctica alcanzamos el propósito final de la aplicación que consiste en que dada una determinada secuencia genómica de origen desconocido, se quiere predecir el género o especie al que corresponde, lo que se conoce como *clasificar*. Para ello, previamente es necesario *aprender* un modelo para cada género o especie con los que se va a comparar el nuevo genoma desconocido. Hasta ahora, hemos elaborado utilidades varias para trabajar con un modelo de especie sin preocuparnos seriamente del recuento de las frecuencias de los k-meros del modelo. No obstante, tanto los genomas de partida de los que se conoce su especie como el genoma desconocido, los tenemos almacenados en ficheros que contienen una única cadena genómica, larga, muy larga [1]. Estos ficheros de genomas tienen extensión (`dna`) o (`rna`) correspondientes a secuencias de DNA o RNA (donde cambia alguno de los nucleótidos válidos en la cadena). Por otro lado, tendremos los ficheros de perfiles (extensión `.prf`) [2], que contienen los k-meros y frecuencias de las secuencias genómicas usadas como modelo.

En esta práctica final se deben implementar dos programas que se podrán ejecutar de forma independiente. La ejecución combinada de estos logrará nuestro objetivo final de predicción de la especie o género de un genoma desconocido. Las tareas están modularizadas, de modo que las salidas de un programa podrán ser entradas del otro. Los programas son: **LEARN** y **CLASSIFY**.

En la Figura 1 podemos identificar: los ficheros de genoma de entrada, las ejecuciones del software a desarrollar para la realización de una clasificación completa y por último, como salida del proceso de clasificación, el identificador más adecuado para el genoma `unknown.dna`, que ha sido seleccionado entre los identificadores candidatos. Sombreado en azul, podemos reconocer una lista de genomas en los que se conoce su género [3], que se van a utilizar como entrada del proceso de aprendizaje. Sombreado en color malva, podemos observar las ejecuciones sucesivas de cada uno de los programas de aplicación que van a ser necesarios para llevar a cabo la predicción del genoma *drosophila* para el fichero `unknown.dna`. Pero, vamos a detallar cada etapa del proceso.

El primer paso consiste en utilizar el programa **LEARN**. Dicho programa construye un fichero profile (`.prf`) a partir del genoma almacenado en ficheros (`.dna` o `.rna`), contabilizando las frecuencias de todos los k-meros que se han hallado en la cadena del genoma. En la figura 2 podemos ver cómo se aplica **LEARN** sobre diversos ficheros correspondientes a diferentes genomas del mismo o de varias especies diferentes. Así pues, mediante este programa obtendremos tantos ficheros profiles como genomas de partida.

Adicionalmente, mediante el programa **LEARN**, se pueden crear profi-

[1] Las cadenas de mayor longitud con las que vamos a trabajar contienen hasta 500.000 nucléotidos.

[2] Los mismos ficheros utilizados en las prácticas anteriores.

[3] Los ficheros que mencionamos los puede encontrar en Genomes, donde hay disponibles múltiples individuos pertenecientes a diferentes especies.

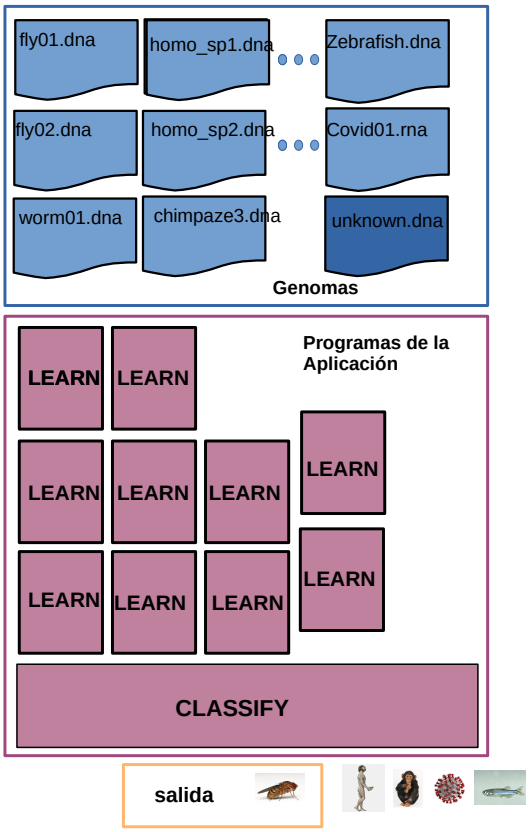

Figura 1: Componentes de la aplicación: Programas y Ficheros de entrada, Salida y predicción final.

les enriquecidos, una suerte de profile aglutinado, que integre en un único perfil secuencias genómicas pertenecientes a individuos de la misma especie. Por ejemplo, el fichero `homo_sapiens.prf` contiene un profile más representativo de *homo sapiens*. Tal fichero ha sido obtenenido a partir de dos secuencias genómicas, correspondientes a dos individuos diferentes, almacenadas en los ficheros `homo_sp1.dna` y `homo_sp2.dna` (vea en la Figura 2, última ejecución de **LEARN**).

Por otro lado, el programa **CLASSIFY** (ver figura 3) toma como entrada el fichero `.prf` de una secuencia genómica desconocida y otro conjunto de ficheros `.prf` que contienen los modelos candidatos (perfiles) de los géneros o especies de referencia. Este programa mostrará por la salida estándar el identificador del perfil más plausible de entre los perfiles candidatos de la secuencia genómica desconocida. Indicar que la clasificación se hace en base a la medida de distancia que se introdujo en la práctica Kmer3. El identificador resultante será aquel cuyo valor de distancia sea menor de entre las distancias calculadas a cada perfil conocido.

2. Arquitectura de las prácticas

Como ya sabemos, la práctica `Kmer` se ha diseñado por etapas. En esta práctica final es dónde se desarrolla la última clase necesaria y, dónde

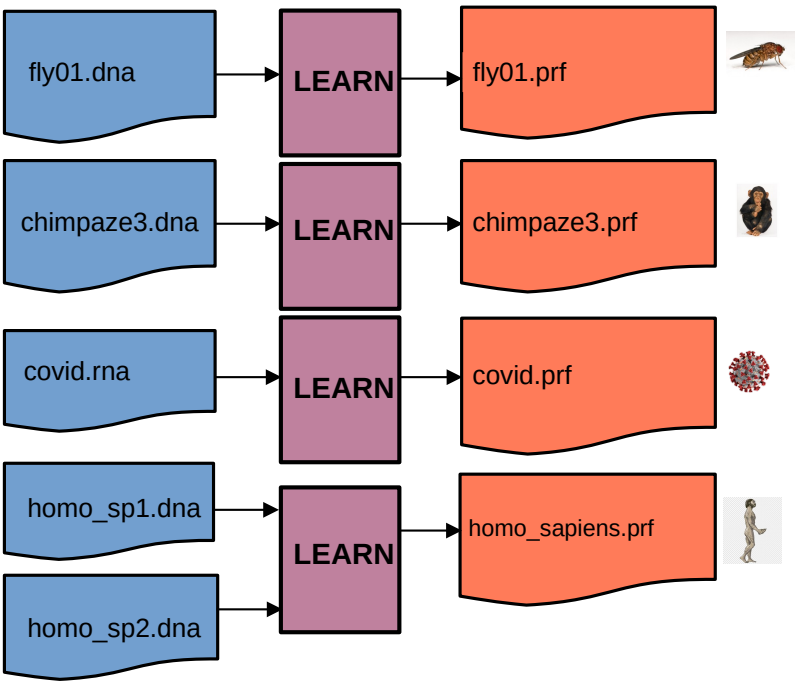

Figura 2: Diferentes ejecuciones del programa **LEARN** para la obtención de cuatro perfiles diferentes.

se revisitan las clases anteriores (vea en la Figura 4 las clases que van a cambiar). Como puede apreciarse, aparece la nueva clase `KmerCounter` y se añaden funcionalidades nuevas a las clases `Kmer`, `KmerFreq` y `Profile`, con la finalidad de permitir usar operadores con los objetos de tales clases de la misma forma que con tipos elementales, mediante la sobrecarga de operadores.

D KmerCounter.cpp
: Implementa la clase KmerCounter, una estructura que nos será de utilidad para llevar a cabo el conteo de cada k-mero de forma eficiente en el proceso de aprendizaje de un modelo. Internamente, esta clase usará una matriz bidimensional en memoria dinámica

Además de la aportación principal de la clase `KmerCounter`, se van a revisitar las clases anteriores.

A' Kmer.cpp
: Implementa la clase `Kmer`, incorporando algunos métodos adicionales como la sobrecarga de operadores << y >> .

B KmerFreq.cpp
: Implementa la clase `KmerFreq`, una composición de un k-mero y un entero para el registro de su frecuencia. Se incorporan algunos métodos adicionales, como la sobrecarga de operadores << y >> y todos los operadores relacionales <, <=, ==, >, >=, !=.

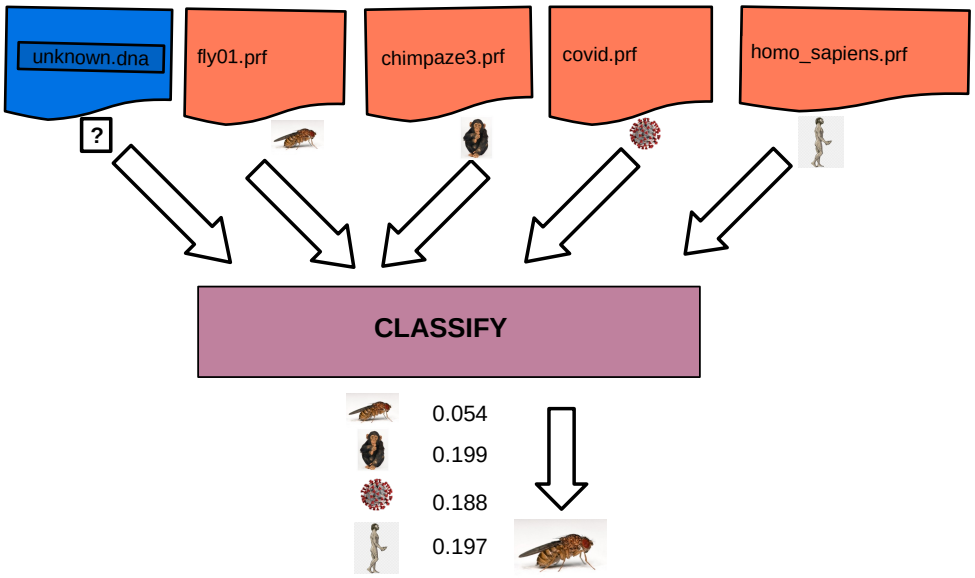

Figura 3: Ejecución del programa **CLASSIFY**.

C' Profile.cpp

Implementa la clase `Profile`, una estructura para almacenar las frecuencias de un conjunto de k-meros utilizando memoria dinámica. Se incorporan algunos métodos adicionales, como la sobrecarga de los operadores `[]`, << y >>. Se deben refactorizar los métodos `load()` y `save()` para que hagan uso **obligatoriamente** de << y >>; además deben extenderse para admitir ficheros binarios. Se debe refactorizar también **obligatoriamente** el método `sort()` para que use alguno de los operadores relacionales recién definidos en `KmerFreq`. Por último, se incorpora la sobrecarga del operador += con un Profile como parámetro. Eso hace que ya no sea necesario el método `join()`. Aunque se mantiene, lo declaramos como `deprecated` [4].

Este trabajo progresivo, se ha planificado en hitos sucesivos y podrá comprobar que parte del contenido de esta práctica final coincide con el trabajo desarrollado en prácticas anteriores, por lo que podrá reutilizar material ya elaborado y depurado. De esta forma el tiempo para su elaboración se verá reducido de forma significativa.

3. Objetivos

El desarrollo de la práctica Kmer5 persigue los siguientes objetivos:

- Practicar con punteros y memoria dinámica dentro de una clase para la implementación de una matriz 2D.

[4] Método obsoleto (deprecated) que no se debe usar desde el exterior de la clase, ya que podría ser eliminado en futuras versiones de la clase.

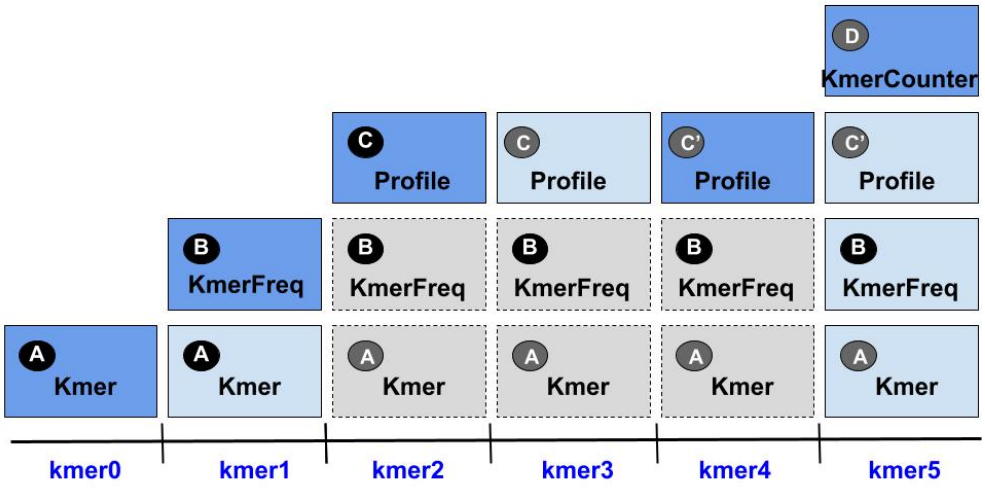

Figura 4: Arquitectura de las prácticas de MP 2024. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso; los que solo incorporan nuevas funcionalidades en azul tenue. En gris se muestran las clases que no sufren cambios en la evolución de las prácticas.

- Profundizar en el desarrollo de los métodos básicos de una clase con memoria dinámica: constructor de copia, destructor y operador de asignación.
- Practicar la sobrecarga de operadores tanto monarios como binarios.
- Practicar con ficheros binarios para su lectura y escritura.
- Elaborar un proyecto con múltiples `main()`.
- Practicar con herramientas como el depurador de Netbeans y `valgrind` para rastrear errores en tiempo de ejecución.

4. Práctica a entregar

Como se ha indicado anteriormente, en esta práctica vamos a trabajar con un proyecto que generará dos programas ejecutables: **LEARN** y **CLASSIFY**. En la sección 5 puede encontrar detalles de cómo hacer esto en `Netbeans`.

Para la elaboración de la práctica, dispone en el repositorio de la asignatura de una plantilla de proyecto en Netbeans ya configurado para trabajar con los dos ejecutables anteriores. Debe completar este proyecto con el código que hizo en prácticas anteriores para las clases `Kmer`, `KmerFreq` y `Profile`, tanto en los ficheros `.h` como `.cpp`. Además debe añadir a cada módulo los nuevos métodos y funciones que se introducen en esta práctica. Finalmente, debe completar el módulo de la

nueva clase `KmerCounter` y los `main()` de los programas **LEARN** y **CLASSIFY**.

4.1. Programa LEARN. Aprendizaje de un modelo profile a partir de un genoma

A partir de uno o varios ficheros de ADN o ARN de entrada, el programa **LEARN** genera un fichero profile de salida con la lista de k-meros diferentes, junto con sus frecuencias, hallados en la lista de ficheros de entrada. El programa permitirá definir cualquier valor para k, a la hora de buscar los k-meros.

4.1.1. La matriz bidimensional de `KmerCounter`

La clase `KmerCounter` es la última que necesitamos para resolver el problema del aprendizaje. En la sección 6.4 puede encontrar la declaración de la clase. La funcionalidad de esta clase permitirá:

1. Realizar el conteo de frecuencias de los k-meros presentes en una secuencia genómica.

2. Una vez realizado el conteo anterior, podrá obtenerse un objeto `Profile`, que posteriormente podrá ser comprimido (eliminación de aquellos k-meros con algún nucleótido no válido o con frecuencia nula) y ordenado por frecuencia de mayor a menor, para obtener el modelo de una especie o género.

La clase encapsula una matriz 2D dinámica de enteros, con tantas casillas como k-meros distintos puedan formarse usando k nucleótidos. El conjunto de nucleótidos que pueden usarse dentro de un k-mero está formado por el conjunto (`KmerCounter::_validNucleotides`) de nucleótidos válidos, más el carácter (`Kmer::MISSING_NUCLEOTIDE = '_'`) que representa un nucleótido desconocido. Este conjunto se almacena en el dato miembro `KmerCounter::_allNucleotides`. Así, cada casilla de la matriz contabilizará la frecuencia de un k-mero concreto, el que venga definido por su fila y columna, según se explica más adelante. De forma general, el número de casillas, n, de esta matriz bidimensional viene determinado por el tamaño del conjunto `_allNucleotides` y del valor k:

$$n = m^k \tag{1}$$

siendo $m = |\_allNucleotides|$ el número de nucléotidos en $\_allNucleotides$, $\_allNucleotides = \{\_\} \cup \{A, C, G, T\}$ para ADN [5] y k, la longitud de los k-meros.

Un ejemplo de matriz que permitiría contar los k-meros ($k = 2$) de una secuencia genómica de ADN sería el que aparece en la Tabla 1. La tabla está inicialmente con el valor 0 en todas las casillas.

[5] Valor por defecto, caso de tratarse de ARN el nucleótido T se cambia por U.

F	_	A	C	G	T	Fila
_	0	0	0	0	0	0
A	0	0	0	0	0	1
C	0	0	0	0	0	2
G	0	0	0	0	0	3
T	0	0	0	0	0	4
Columna	0	1	2	3	4	

Cuadro 1: Matriz bidimensional F para conteo de frecuencias de bimeros de ADN.

posición	0	1	2	3	4
nucleótido	_	A	C	G	T

Cuadro 2: Posición de cada nucléotido en $\_allNucleotides$

Declarado en `KmerCounter.h` y definido en tiempo de ejecución, el atributo `_allNucleotides` es un string que va a cobrar gran relevancia a la hora de indexar dentro de la matriz. Cada carácter de este string ocupa una posición dentro de él. Suponemos que las posiciones empiezan en cero, y van numeradas de izquierda a derecha como muestra la Tabla 2 para el caso de ADN.

4.1.2. Obtención de fila y columna para un $k-mero$ ($k = 2$)

Cuando nos llega un $k-mero$ debemos sumar 1 en una de las casillas de la matriz bidimensional. El primer paso que necesitamos hacer es determinar la fila y columna de la casilla que corresponde a tal $k-mero$. Veamos en primer lugar, cómo realizar este cálculo mediante un ejemplo para el caso en que $k = 2$.

Ejemplo 1 *Se quiere montar una matriz bidimensional para contabilizar $k-meros$ con $k = 2$. El genoma de partida es el siguiente:*

```
AAACGTTAAAGT
```

Para cada par de caracteres consecutivos en la cadena de entrada, la lista de bímeros que se obtiene es:

```
AA
 AA
  AC
   CG
    GT
     TT
      TA
       AA
        AA
         AG
          GT
```

Indicar que, sea cual sea la cadena de partida, para cada bímero extraído, se debe comprobar si cada carácter que lo compone es un nucleótido válido. En caso de encontrarse un nucleótido no válido, el proceso de normalización del k-mero sustituye tal carácter por el carácter '_'.

Una vez normalizado cada k-mero podemos contabilizar su frecuencia de aparición. Para realizar este conteo de forma eficiente, vamos a ayudarnos de la matriz bidimensional, `F`, con $n = 25$ casillas, que puede verse en la tabla 3. En la posición $F[i][j]$ se almacenará la frecuencia con la que aparece el k-mero que tiene como primer carácter el elemento i-ésimo y como segundo carácter el elemento j-ésimo en el conjunto `_allNucleotides`, con valores $0, \ldots, 4$ para i y j respectivamente [6]. Así, al bímero AA le corresponde el contador $F[1][1]$, al bímero AC el contador $F[1][2]$, y al bímero TT el contador $F[4][4]$. Por tanto, dado un bímero, podemos actualizar su frecuencia fácilmente en la matriz, cuando hallamos la fila y la columna a que corresponde cada carácter del bímero. O sea, calcularemos la posición pos de cada carácter del bímero en `_allNu cleotides` (tabla 2) para obtener la fila y columna en la matriz e incrementamos en 1 el valor del contador de la casilla encontrada.

F	_	A	C	G	T	Fila
_	0	0	0	0	0	0
A	0	4	1	1	0	1
C	0	0	0	1	0	2
G	0	0	0	0	2	3
T	0	1	0	0	1	4
Columna	0	1	2	3	4	

Cuadro 3: Matriz bidimensional F correspondiente al conteo de frecuencias del ejemplo 1.

4.1.3. Obtención de fila y columna para un $k-mero$ (cualquier k)

Para bímeros, ha sido muy intuitivo montar una matriz bidimensional, pero como vamos a ver, usaremos siempre una matriz `KmerCounter` de dimensión 2, sea cual sea el valor de k de los k-meros que se vayan a contabilizar. Para montarla, se va a fraccionar cada k-mero en dos fragmentos (sub$k$-meros). Así, la parte izquierda tendrá una longitud de $l_f = (k+1)/2$ caracteres y la parte derecha una longitud de $l_c = k - l_f$ caracteres. Por ejemplo, en el caso de un 5-mero ($k = 5$) se parte en dos sub$k$-meros de longitudes $l_f = 3$ y $l_c = 2$. El primero servirá para indexar la fila de la matriz y el segundo para indexar la columna.

Al igual que en el caso $k = 2$, el cálculo de n se hace mediante la fórmula (1). El número de filas a reservar en la matriz depende de l_f, y se calcula con:

$$nfilas = m^{l_f}$$

[6] Se busca un carácter en el string $\_allNucleotides$, la posición devuelta es el índice a utilizar para filas o columnas de la matriz.

mientras que el número de columnas sería:

$$ncolumnas = m^{l_c}$$

Ejemplo 2 *Se quiere montar una matriz bidimensional para contabilizar* $k-meros$ *con* $k = 3$*. El genoma de partida es el mismo que en el ejemplo anterior:*

```
AAACGTTAAAGT
```

F	_	A	C	G	T	Fila
__	0	0	0	0	0	0
_A	0	0	0	0	0	1
_C	0	0	0	0	0	2
_G	0	0	0	0	0	3
_T	0	0	0	0	0	4
A_	0	0	0	0	0	5
⋮	⋮	⋮	⋮	⋮	⋮	⋮
TG	0	0	0	0	0	23
TT	0	1	0	0	0	24
Columna	0	1	2	3	4	

Cuadro 4: Matriz bidimensional F correspondiente al conteo de frecuencias del ejemplo 2.

La matriz `F` *tendría en este caso* $n = m^k = 125$ *casillas. Se usarán* $l_f = 2$ *nucleótidos de cada* k*-mero para indexar la fila y* $l_c = 1$ *nucleótido para indexar la columna. El número de filas de la matriz será de* $nfilas = m^2 = 25$*; el número de columnas* $ncolumnas = m^1 = 5$*. La matriz resultante aparece parcialmente en la Tabla 4.*

Dado el 3*-mero* $\_\_A$*, este se fragmenta como* $\_\_$ *y A, que corresponde en la matriz* F *con la casilla con* $fila = 0$ *y* $columna = 1$*.*

De forma general, para hallar el valor del índice (fila o columna) de un fragmento $kmer$ de longitud l de un k-mero usando un determinado conjunto de nucleótidos $\_allNucleotides$ usaremos la siguiente expresión:

$$index = \sum_{i=0...l-1} m^i \cdot pos(kmer[l-i-1]) \qquad (2)$$

donde la notación $kmer[i]$ se refiere al nucleótido i del fragmento $kmer$, y $pos(nucleotido)$ sería la posición que ocupa el nucleótido en $\_allNucleotides$. De esta forma, $pos(kmer[l-i-1])$ sería la posición del nucleótido $l-i-1$ del fragmento $kmer$ en $\_allNucleotides$ (vea en la Tabla 2 las posiciones de cada posible nucleótido en $\_allNucleotides$).

Ejemplo 3 *Llegan los* 3*-meros AAA y CGT.*

Continuando con el ejemplo 2, si ahora llega el 3-mero AAA, se fragmentará como AA y A lo que se corresponde con la casilla de F situada en $fila = m^0 \cdot 1 + m^1 \cdot 1 = 6$ y $columna = m^0 \cdot 1 = 1$.

Realizando los mismos cálculos, dado el 3-mero CGT, se fragmenta como CG y T, lo que se corresponde con la casilla de F situada en $fila = m^0 \cdot 3 + m^1 \cdot 2 = 13$ y $columna = m^0 \cdot 4 = 4$.

Ejemplo 4 *Ejercicio con 5-mero.*

Compruebe que, dado k=5, las coordenadas para el 5-mero AAACG son $fila = 31$ y $columna = 13$, y para AACGT son $fila = 81$ y $columna = 19$.

El cálculo del índice (fila o columna) de un fragmento se realizará mediante el método privado `int getIndex(fragmento)` de la clase `KmerCounter`.

4.1.4. Obtención del k-mero asociado a una casilla de la matriz de `KmerCounter`

Para concluir con los métodos clave de la clase `KmerCounter`, una vez se ha contabilizado la frecuencia de aparición de cada uno de los k-meros mediante la matriz F, esta se tiene que traducir a un objeto de la clase `Profile`. En el ejemplo 1, el contenido del objeto `Profile` que se obtiene a partir de la tabla 3, se muestra a continuación. Como puede verse, no se han incluido k-meros con frecuencia igual a cero. El `Profile` no está ordenado por frecuencia todavía.

```
unknown
7
AA 4
AC 1
AG 1
CG 1
GT 2
TA 1
TT 1
```

Partiendo de la tabla 3, vamos a convertir cada casilla $F[i, j]$ no nula [7] de F en un objeto `KmerFreq`, compuesto de un k-mero y una frecuencia. Para ello, se recorre sistemáticamente cada posición de la tabla (i, j) para almacenar en el profile el k-mero y su frecuencia.

Veamos cómo proceder para la obtención del k-mero. Mediante el método privado `Kmer getKmer(fila, columna)`, dadas las coordenadas `fila` y `columna`, se ha de componer el k-mero al que corresponde esa casilla. Recordemos que, un k-mero se fragmenta en dos sub$k$-meros, el correspondiente a la fila y el correspondiente a la columna. El método `Kmer getKmer(fila, columna)` va a hacer la operación inversa, que consiste en componer el k-mero final a partir de los dos sub$k$-meros obtenidos por separado, uno correspondiente a la fila y otro a la columna en

[7] Por eficiencia, solo las casillas con frecuencia distinta de cero, son las que tienen interés de ser almacenadas en el profile.

la matriz bidimensional de `KmerCounter`. Este método se ayudará del método privado `string getInvertedIndex(index, l)` para componer el k-mero de la siguiente forma:

$$kmero = getInvertedIndex(fila, l_f) + getInvertedIndex(columna, l_c)$$

donde l_f y l_c son el número de nucleótidos que usamos en el k-mero para indexar las filas y columnas respectivamente (vea cómo obtener l_f y l_c en la sección 4.1.3).

El método `string getInvertedIndex(index, l)`, devuelve una cadena de caracteres de longitud l con el l-mero correspondiente al índice (una fila o columna de la matriz) proporcionado. Este método hace la operación inversa a lo que hace `int getIndex(fragmento)` en el proceso de obtención (indexación) de la fila o columna para un fragmento de k-mero.
Este método comienza creando una cadena $kmer$ de longitud l, con todas sus posiciones con el carácter `_`. Luego, calcula cada carácter de $kmer$ de la siguiente forma:

$$kmer[i] = \_allNucleotides[index\,\%m];\ index = index/m,\ \ i = l-1, \ldots, 0$$

donde $\%$ es la operación de módulo y $/$ la operación de división entera.

Ejemplo 5 *Obtención del fragmento de un k-mero dada la fila en que aparece.*

En el ejemplo 4 teníamos el k-mero AAACG. El fragmento correspondiente a las filas sería AAA que iría a la fila 31. Veamos que efectivamente, la cadena devuelta por `string getInvertedIndex(31, 3)` *sería AAA.*

- *En la primera iteración* $i = 2$, $\_allNucleotides[31\,\%m] =$ `A`, $kmer =$`__A` *y se actualiza* $index$ *al valor* $index/m = 6$.
- *En la segunda iteración* $i = 1$, $\_allNucleotides[6\,\%m] =$ `A`, $kmer =$`_AA` *y se actualiza* $index$ *al valor* $index/m = 1$.
- *En la tercera y última iteración* $i = 0$, $\_allNucleotides[1\,\%m] =$ `A`, $kmer =$`AAA` *y se actualiza* $index$ *al valor* $index/m = 0$.

Ejemplo 6 *Obtención del fragmento de un k-mero dada la fila en que aparece.*

En el ejemplo 3 teníamos el k-mero CGT. El fragmento correspondiente a las filas sería CG que iría a la fila 13. Veamos que efectivamente, la cadena devuelta por `string getInvertedIndex(13, 2)` *sería CG.*

- *En la primera iteración* $i = 1$, $\_allNucleotides[13\,\%m] =$ `G`, $kmer =$`_G` *y se actualiza* $index$ *al valor* $index/m = 2$.
- *En la segunda y última iteración* $i = 0$, $\_allNucleotides[2\,\%m] =$ `C`, $kmer =$`CG` *y se actualiza* $index$ *al valor* $index/m = 0$.

Si consideramos por separado los valores anteriores de (31,13) como las coordenadas de `F` con $k = 5$, el $kmer$ final que obtendrá el método `Kmer getKmer(fila, columna)` sería AAACG.

4.1.5. Ejecución de LEARN

La sintaxis para ejecutar el programa **LEARN** es como sigue:

```
Linux> LEARN [-t|-b][-p profileId][-k kValue][-n nucleotidesSet] [-o
↪  ficheroSalida] fichero1.dna [fichero2.dna fichero3.dna ...]
```

Como en prácticas anteriores, la notación `[]` indica que el parámetro no es obligatorio, es opcional y | indica alternativa.
Los argumentos son:

-t | **-b**: modo texto o modo binario para el fichero de salida, siendo `-t` el valor por defecto.
-p `profileId`: define el nombre del género o de la especie que se aprende, siendo `unknown` el valor por defecto si no se usa este parámetro.
Nota: La cadena `profileId` debe darse entre comillas simples (') si contiene espacios en blanco.
-k `kValue`: longitud de los k-meros del modelo a aprender. `kValue = 5` por defecto si no hay parámetro -k.
-n `nucleotidesSet`: cadena con los caracteres de los nucleótidos válidos del modelo a aprender, siendo `nucleotidesSet = ACGT` el valor por defecto si no se usa este parámetro.
-o `ficheroSalida`: nombre del fichero de salida <**.prf**> siendo `output.prf` el valor por defecto, si no se usa este parámetro.

Nota: Los parámetros se pueden introducir en cualquier orden, pero solo los que comienzan con $-$.

El programa **LEARN** acepta un número indeterminado de ficheros de entrada <.dna>, siempre que al menos aparezca uno en la llamada.

Para el uso correcto de los parámetros opcionales del programa, detallaremos algunos ejemplos de llamadas al programa **LEARN**.

Ejemplo 7 *Faltan argumentos para la ejecución del programa. El programa necesita al menos un fichero de entrada (un genoma).*

```
Linux> LEARN
```

La salida del ejemplo anterior debe dar el siguiente mensaje por la salida estándar de error:

```
ERROR in LEARN parameters
Run with the following parameters:
LEARN [-t|-b][-k kValue][-n nucleotidesSet][-p profileId][-o outputFilename]
↪  <file1.dna>[<file2.dna> <file3.dna>...]

Parameters:
-t|-b: text mode or binary mode for the output file (-t by default)
-k kValue: number of nucleotides in a kmer (5 by default)
-n nucleotidesSet: set of possible nucleotides in a kmer (ACGT by default).
↪  Note that the characters should be provided in uppercase
-p profileId: profile identifier (unknown by default)
-o outputFilename: name of the output file (output.prf by default)
<file1.dna> <file2.dna> <file3.dna> ....: names of the input files (at least
↪  one is mandatory)

This program learns a profile model from a set of input DNA files
↪  <file1.dna><file2.dna><file3.dna> ....
```

Ejemplo 8 *Sea la siguiente llamada con argumentos correctos. Se quiere aprender del profile homo sapiens a partir del genoma* `human1.dna`*, guardando el modelo aprendido en el fichero* `human1.prf`*.*

```
Linux> LEARN -p 'homo sapiens' -o output/human1.prf ../Genomes/human1.dna
```

El resultado de esta ejecución es el fichero `human1.prf` en disco, con la lista de 5-meros y las frecuencias halladas en el genoma de `human1.dna`. El formato del fichero `human1.prf` es el que hemos estado usando en las prácticas anteriores al guardarse en formato texto.

Ejemplo 9 *Aprendizaje del género homo sapiens a partir de los genomas* `human1.dna` *y* `human2.dna`*, guardando el modelo aprendido en el fichero* `human.prf` *en formato binario.*

```
Linux> LEARN -b -p 'homo sapiens' -o output/human.prf ../Genomes/human1.dna
↪  ../Genomes/human2.dna
```

En la carpeta `Genomes` podrá encontrar una serie de ficheros de genomas de ejemplo en el formato indicado, procedentes de al menos 6 especies diferentes.

4.2. Programa CLASSIFY. Predicción del género o de la especie de un genoma

El programa **CLASSIFY** permite obtener el género o especie de un fragmento de genoma de entrada comparándolo con un conjunto de profiles de referencia $P_1, P_2, \ldots, P_n$, que han sido aprendidos a partir de otros fragmentos de genoma cuyo identificador de especie o género era conocido. La forma de comparar el genoma de entrada con los profiles que sirven de modelo, es construir un profile para el genoma de entrada y medir la distancia que hay a cada uno de los profiles $P_1, P_2, \ldots, P_n$ para determinar aquel con menor distancia. Al genoma de género o especie desconocida, se le asignará el identificador del profile de menor distancia. En las prácticas `Kmer3` y `Kmer4` ya resolvimos el problema de medir la distancia entre dos profiles.

Resumiendo, el proceso de clasificación se descompone en tres pasos:

1. Aprender el profile P_x para el fragmento de genoma desconocido.
2. Calcular la distancia del profile P_x a cada P_j de referencia para hallar el de menor distancia P_{min}.
3. Asignarle la etiqueta del P_{min} al genoma desconocido.

En el paso 1, a partir del genoma de entrada se ha de generar un profile cuyo identificador es `unknown`; este profile contiene la lista de los k-meros hallados con sus frecuencias. El conteo de las frecuencias

de los k-meros hallados se realiza mediante los métodos de la clase `KmerCounter` desarrollados para el programa LEARN. Finalmente, se obtiene un profile P_x y ya estamos en las mismas condiciones de `Kmer4`.

En el paso 2, se calculan las distancias $distance(P_x, P_1)$, $distance(P_x, P_2)$, $distance(P_x, P_3)$, ..., $distance(P_x, P_n)$.

En el paso 3, la predicción de identificador consiste en asignarle al genoma de entrada, el género o la especie de aquel profile cuya distancia sea menor.

La sintaxis para ejecutar el programa **CLASSIFY** es como sigue:

```
Linux> CLASSIFY <individuo.dna> <especie1.prf> [<especie2.prf> <especie3.prf>...]
```

El programa recibe al menos dos ficheros de entrada [8]: `<individuo.dna>` y `<especie.prf>`. El primero, `<individuo.dna>`, es el genoma o fragmento de entrada del que se va a obtener un objeto profile en memoria. El segundo, `<especie1.prf>`, y el resto de ficheros (`.prf`) son los profiles con los que se van a calcular las distancias para buscar el más cercano. Finalmente, la salida muestra por pantalla el del identificador del perfil seleccionado.

Ejemplo 10 *Faltan argumentos para la ejecución del programa*

```
Linux> CLASSIFY
```

Ejemplo 11 *Faltan argumentos para la ejecución del programa*

```
Linux> CLASSIFY ../Genomes/chimpanzee_chr6_s1_1500000.dna
```

La salida de los dos ejemplos anteriores debe dar el siguiente mensaje por la salida estándar de error:

```
ERROR in CLASSIFY parameters
Run with the following parameters:
CLASSIFY [-k kValue][-n nucleotidesSet]<file.dna><profile1.prf>[<profile2.prf>
↪  <profile3.prf> ....]

Parameters:
-k kValue: number of nucleotides in a kmer (5 by default)
-n nucletiodesSet: set of possible nucleotides in a kmer (ACGT by default). It
↪  is used when learning a model for <file.dna>. Note that the characters
↪  should be provided in uppercase
<profile1.prf>[<profile2.prf><profile3.prf>....] ....: names of the Profile
↪  models (at least one is mandatory)

This program obtains the identifier of the closest profile to the input DNA
↪  file
```

Ejemplo 12 *Clasificación del genoma 'desconocido' contenido en el fichero* `drosophila_chr2L_s1_1500000.dna` *como drosophila melanogaster, pan troglodyte, severe acute respiratory syndrome coronavirus 2, o como homo sapiens.*

[8] Mínimo dos ficheros para que no dé error de sintaxis, pero no ha lugar a ninguna predicción, pues le asignará siempre el identificador del segundo. ¿Por qué?.

```
Linux> CLASSIFY ./Genomes/drosophila_chr2L_s1_l1500000.dna
↪  ../Genomes/drosophila_chr3L_s1_l1500000.prf
↪  ../Genomes/chimpanzee_chr6_s1_l1500000.prf
↪  ../Genomes/covidFullGenomeRNA.prf ../Genomes/human_chr6_s60000_l1500000.prf
```

Puede comprobar que se clasifica como especie *drosophila melanogaster*, como era de esperar. La salida obtenida es:

```
Distance to ../Genomes/drosophila_chr3L_s1_l1500000.prf (drosophila
↪  melanogaster): 0.0546589
Distance to ../Genomes/chimpanzee_chr6_s1_l1500000.prf (pan troglodytes):
↪  0.19194
Distance to ../Genomes/covidFullGenomeRNA.prf (severe acute respiratory
↪  syndrome coronavirus 2): 0.449932
Distance to ../Genomes/human_chr6_s60000_l1500000.prf (homo sapiens): 0.189238

Final decision: drosophila melanogaster with a distance of 0.0546589
```

Ejemplo 13 *Clasificación del genoma* `human1.dna` *de humano como severe acute respiratory syndrome coronavirus 2, drosophila melanogaster, o como pan troglodyte.*

```
Linux> CLASSIFY ../Genomes/human1.dna ../Genomes/covidFullGenomeRNA.prf
../Genomes/drosophila_chr3L_s1_l1500000.prf
↪  ../Genomes/chimpanzee_chr6_s1_l1500000.prf
```

Se clasifica como *pan troglodytes* (chimpacé) lo cual puede parecer sorprendente, pero fijémonos que no se proporcionó ningún profile de humano con el que comparar, por lo que la menor distancia es con el profile `chimpanzee_chr6_s1_l1500000.prf`, uno de los modelos para chimpacé. A continuación se muestra la salida que se obtendría:

```
Distance to ../Genomes/covidFullGenomeRNA.prf (severe acute respiratory
↪  syndrome coronavirus 2): 0.462093
Distance to ../Genomes/drosophila_chr3L_s1_l1500000.prf (drosophila
↪  melanogaster): 0.302083
Distance to ../Genomes/chimpanzee_chr6_s1_l1500000.prf (pan troglodytes):
↪  0.180438

Final decision: pan troglodytes with a distance of 0.180438
```

No se puede predecir una especie para un genoma que a priori no se haya aprendido y con el que se compare. Con esto constatamos una de las muchas limitaciones que tiene toda clasificación, *si no se tiene un modelo apropiado con el que comparar, en el caso anterior, un humano, la clasificación será forzosamente errónea.*

4.3. Ficheros profile en formato binario

Como sabemos, los datos pueden guardarse en un fichero en formato texto y en formato binario. Hasta ahora hemos venido usando formato texto para nuestros ficheros profile (extensión `.prf`). En esta práctica final se pide que los ficheros profile permitan que los datos estén guardados en formato texto o bien en formato binario según se elija al crear cada fichero.

El formato de los ficheros profile (de texto) que teníamos hasta ahora es el siguiente:

- La primera línea contiene siempre la cadena “MP-KMER-T-1.0”.
- La segunda línea contiene una cadena que indica el identificador de especie o género. Es la cadena que se introdujo al crear el profile con el programa `LEARN` usando el parámetro `-p identificador`. El valor sería `unknown` si no se usó tal parámetro.
- La tercera línea contiene el número de k-meros diferentes que están asociados al profile descrito.
- Las siguientes líneas contienen la lista de k-meros y sus frecuencias, de longitud el número especificado en la línea tercera.

En el caso de ficheros profile binarios, usaremos la cadena mágica “MP-KMER-B-1.0” en la primera línea del fichero en lugar de “MP-KMER-T-1.0”. Concretamente, el formato de los ficheros profile binarios será el siguiente:

- La primera línea contiene la cadena “MP-KMER-B-1.0”.
- La segunda línea coincide con el contenido en formato de texto: una cadena de texto que indica el identificador de especie o género.
- La tercera línea también coincide con el contenido en formato de texto: número (en formato texto) de k-meros diferentes que están asociados al profile descrito.
- A continuación aparece en formato binario la lista de parejas k-mero y frecuencia.

Cada pareja k-mero-frecuencia aparece en un fichero de salida binario de la siguiente forma:

- Para el k-mero, aparecen cada uno de sus caracteres, más el carácter `'\0'` que sirve para indicar que el k-mero no tiene más caracteres.
- La frecuencia aparece como un entero guardado en forma binaria. O sea, en el fichero aparece con los bytes que usa la representación interna del entero de la frecuencia.

Si se visualizase un fichero profile binario con un editor de texto, solo la cabecera (tres primeras líneas) sería legible. La lista de parejas k-mero y frecuencia no lo será ya que las frecuencias están guardadas siempre usando los 4 bytes de la representación interna del entero en memoria principal, y no dígito a dígito como pasaría si apareciese en modo texto.

Para leer en modo binario usaremos el método `std::istream::read()` de la clase `istream` y para escribir en modo binario usaremos el método `std::ostream::write()` de la clase `ostream` [9].

Para que nuestros programas sean capaces de leer ficheros profile binarios, debe modificar el método `Profile::load(fileName)` que hizo en prácticas anteriores, para que sea capaz de leer tales ficheros. Este método tendrá que empezar leyendo la cadena mágica y según sea esta, leer el fichero como si fuese texto o como si fuese binario.

[9] Se remite al lector a revisar los ficheros binarios en la parte de teoría.

- En caso de que sea de texto, se debe delegar (es obligatorio hacerlo según se indica en la sección 2) el resto de lectura del fichero en el operador >> de la clase `Profile`.
- Para ficheros binarios, se debe leer el resto de la cabecera del fichero, y cargar a continuación las parejas k-mero y frecuencia ayudándose del método `KmerFreq::read(inputStream)`. Este método debe implementarlo en la clase `KmerFreq`, el cual se ayudará a su vez del método `Kmer::read(inputStream)` que implementará en la clase `Kmer`.

En el caso de la escritura de ficheros profile, debe modificar el método `Profile::save(fileName)` que hizo en prácticas anteriores. En este caso, es necesario añadir un parámetro adicional a este método para poder elegir si se quiere escribir el fichero en formato texto o binario. En el fichero `Profile.h` de Kmer5, puede ver el prototipo y documentación para la nueva versión de este método:

```
/**
  * @brief Saves this Profile object in the given file
  * Query method
  * @param fileName A c-string with the name of the file where this Profile
  * object will be saved. Input parameter
  * @param mode The mode to use to save this Profile object: 't' for text
  * mode and 'b' for binary mode. Input parameter
  * @throw std::invalid_argument Throws a std::invalid_argument exception
  * if the given @mode is not valid ('t' or 'b')
  * @throw std::ios_base::failure Throws a std::ios_base::failure exception
  * if the given file cannot be opened or if an error occurs while writing
  * to the file
  */
void save(char *fileName, char mode = 't');
```

Como siempre, debe revisar el prototipo del método anterior, y modificarlo en su caso. En la implementación de este método, según sea el valor del parámetro `mode`, se escribirá el fichero en formato texto o binario.

- En caso de que sea de texto, se debe delegar (es obligatorio hacerlo según se indica en la sección 2) el resto de escritura del fichero en el operador << de la clase `Profile`.
- Para ficheros binarios, se debe escribir el resto de la cabecera del fichero, y escribir a continuación las parejas k-mero y frecuencia ayudándose del método `KmerFreq::write(outputStream)`. Este método debe implementarlo en la clase `KmerFreq`, el cual se ayudará a su vez del método `Kmer::write(outputStream)` que implementará en la clase `Kmer`.

Note en los comentarios del método `save()`, que este debe lanzar una nueva excepción en caso de que el modo de escritura proporcionado no sea correcto.

5. Netbeans. Un proyecto con varios ejecutables

Netbeans elabora un solo ejecutable por cada proyecto. Así que, para evitar tener que crear dos proyectos, uno por cada programa, se ha

preparado el proyecto Kmer5 con la configuración específica, con la estructura habitual de directorios. En la carpeta `src` además de la definición de cada una de las clases se encuentran 2 ficheros fuentes: `LEARN.cpp` y `CLASSIFY.cpp`. Cada uno tiene su función `main()` correspondiente para la gestión de sus parámetros de entrada. Para la compilación y ejecución de los programas por separado, utilizará un fichero denominado `metamain.cpp` ya completo, que nos permitirá seleccionar el programa principal del proyecto. El contenido se muestra a continuación.

```
#ifdef LEARN
    #include "LEARN.cpp"
#elif CLASSIFY
    #include "CLASSIFY.cpp"
#endif
```

Para seleccionar en Netbeans el fuente que se quiere compilar y ejecutar, hemos de seleccionar el ejecutable que se desea. Para ello, seleccionamos una de las configuraciones alternativas a DEBUG y RELEASE que han sido añadidas en este proyecto: LEARN y CLASSIFY. En la figura 5 puede verse que se tiene seleccionada la configuración LEARN en la lista desplegable. Otra forma alternativa de hacerlo es, situados en el proyecto Kmer5 $->$ Properties $->$ Debug $->$ Configuration y se despliega la lista indicada para la selección.

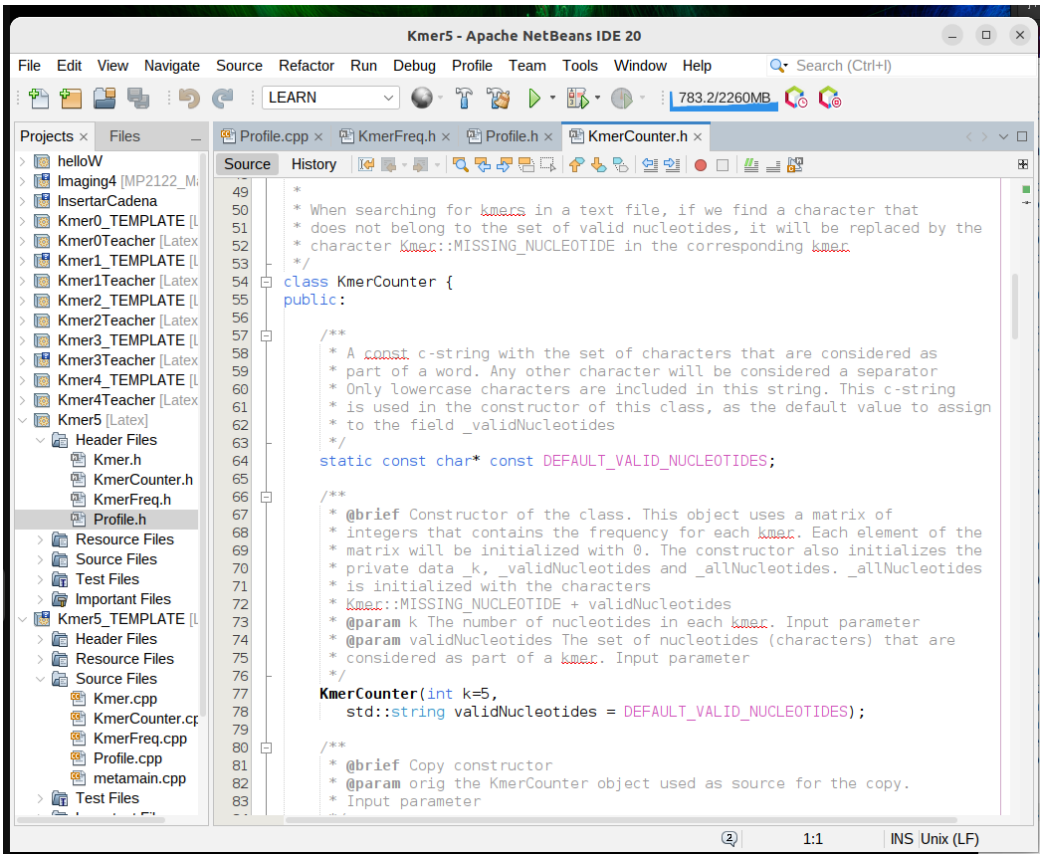

Figura 5: Seleccionando el ejecutable

6. Código para la práctica

6.1. Kmer.h

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/**
 * @file Kmer.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 17 November 2023, 10:15
 */

#ifndef KMER_H
#define KMER_H

#include <iostream>
#include <string>

/**
 * @class Kmer
 * @brief It represents a list of k consecutive nucleotides of a DNA or RNA
 * sequence. Each nucleotide is represented with a character like
 * 'A', 'C', 'G', 'T', 'U'.
 */
class Kmer {
public:
    /**
     * A static const character representing an unknown nucleotide. It is used
     * when we do not known which nucleotide we have in a given position of
     * a Kmer
     */
    static const char MISSING_NUCLEOTIDE = '-';

    /**
     * @brief It builds a Kmer object using a string with @p k characters
     * (nucleotides). Each character will be set to the value
     * @p MISSING_NUCLEOTIDE.
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if @p k is less or equal than zero
     * @param k the number of nucleotides in this Kmer. It should be an integer
     * greater than zero. Input parameter
     */
    Kmer(int k=1);

    /**
     * @brief It builds a Kmer object with the characters in the string @p text
     * representing the list of nucleotides of the new Kmer.
     * @throw std::invalid_argument Throws an std::invalid_argument exception
     * if the given text is empty
     * @param text a string with the characters representing the nucleotides for
     * the kmer. It should be a string with at least one character. Input parameter
     */
    Kmer(const std::string& text);

    /**
     * @brief Returns the number of nucleotides in this Kmer.
     * Query method
     * @return the number of nucleotides in this Kmer
     */
    int getK() const;

    /**
     * @brief Returns the number of nucleotides in this Kmer.
     * Query method
     * @return the number of nucleotides in this Kmer
     */
    int size() const;

    /**
     * @brief Returns a string with a list of characters, each one representing
     * a nucleotide of this Kmer.
     * Query method
     * @return The text of this Kmer as a string object
     */
    std::string toString() const;

    /**
     * @brief Gets a const reference to the character (nucleotide) at the given
     * position.
     * Query method
     * @param index the position to consider. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * index is not in the range from 0 to k-1 (both included).
     * @return A const reference to the character at the given position
     */
    const char& at(int index) const;

    /**
     * @brief Gets a reference to the character (nucleotide) at the given
```

```
     * position.
     * Modifier method
     * @param index the position to consider. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * index is not in the range from 0 to k-1 (both included).
     * @return A reference to the character at the given position
     */
    char& at(int index);

    /**
     * @brief Converts uppercase letters in this Kmer to lowercase
     * Modifier method
     */
    void toLower();

    /**
     * @brief Converts lowercase letters in this Kmer to uppercase
     * Modifier method
     */
    void toUpper();

    /**
     * @brief Normalizes this Kmer. That is, it converts all the characters to
     * uppercase. Then, invalid characters are replaced by the
     * MISSING_NUCLEOTIDE value.
     * Modifier method
     * @param validNucleotides a string with the list of characters
     * (nucleotides) that should be considered as valid. Input parameter
     */
    void normalize(const std::string& validNucleotides);

    /**
     * @brief Returns the complementary of this Kmer. For example, given the Kmer
     * "TAGAC", the complementary is "ATCTG" (assuming that we use.
     * @p nucleotides="ATGC" and @p complementaryNucleotides="TACG").
     * If a nucleotide in this object is not in @p nucleotides, then that
     * nucleotide remains the same in the returned kmer.
     * Query method
     * @param nucleotides A string with the list of possible nucleotides. Input parameter
     * @param complementaryNucleotides A string with the list of complementary
     * nucleotides. Input parameter
     * @throw std::invalid_argument Throws an std::invalid_argument exception if
     * the sizes of @p nucleotides and @p complementaryNucleotides are not
     * the same
     * @return The complementary of this Kmer
     */
    Kmer complementary(const std::string& nucleotides,
        const std::string& complementaryNucleotides) const;

    /**
     * @brief Writes this object to the given output stream. All the characters in
     * the string of this Kmer (including '\0') are sent to the ouput stream.
     * Query method
     * @param outputStream An output stream where this object will be written
     */
    void write(std::ostream outputStream);

    /**
     * @brief Reads this object from the given input stream. It reads characters
     * from the given input stream and put them in the text of this Kmer
     * Modifier method
     * @param inputSstream An input stream from which this object will be read
     */
    void read(std::istream inputStream);

private:
    /**
     * A string with a list of characters representing the nucleotides in
     * this Kmer.
     */
    std::string _text;
}; // end class Kmer

/**
 * @brief Checks if the given nucleotide is contained in @p validNucleotides.
 * That is, if the given character can be considered as part of a genetic
 * sequence.
 * @param nucleotide The nucleotide (a character) to check. Input parameter
 * @param validNucleotides The set of characters that we consider as possible
 * characters in a genetic sequence. Input parameter
 * @return true if the given character is contained in @p validNucleotides;
 * false otherwise
 */
bool IsValidNucleotide(char nucleotide, const std::string& validNucleotides);

/**
 * @brief Converts to lowercase the characters (nucleotides) of the given Kmer
 * @deprecated This function could go away in future versions
 * @param kmer A Kmer object. Output parameter
 */
void ToLower(Kmer& kmer);

/**
 * @brief Converts to uppercase the characters (nucleotides) of the given Kmer
 * @deprecated This function could go away in future versions
 * @param kmer A Kmer object. Output parameter
 */
void ToUpper(Kmer& kmer);

/**
```

```
 * @brief Overloading of the stream insertion operator for Kmer class. It
 * inserts the characters (nucleotides) of the given Kmer in the output string.
 * @param os The output stream to be used. Output parameter
 * @param kmer the Kmer object. Input parameter
 * @return @p os A reference to the output stream
 */
std::ostream operator<<(std::ostream os, Kmer kmer);

/**
 * @brief Overloading of the stream extraction operator for Kmer class. It
 * reads a list of characters from the input string that will set the
 * list of nucleotides of the given Kmer.
 * @param is The input stream to be used
 * @param kmer the Kmer object
 * @return @p is the input stream
 */
std::istream operator>>(std::istream is, Kmer kmer);

#endif /* KMER_H */
```

6.2. KmerFreq.h

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/*
 * @file   KmerFreq.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 17 November 2023, 10:15
 */


#ifndef KMER_FREQ_H
#define KMER_FREQ_H

#include "Kmer.h"

/**
 * @class KmerFreq
 * @brief A pair formed by a Kmer object and a frequency (an int),
 * that gives the frequency of a Kmer (times it appears) in a genoma.
 */
class KmerFreq {
public:
    /**
     * @brief Base constructor.
     * It builds a KmerFreq object containing a Kmer with one nucleotide, the
     * unknown nucleotide (Kmer::UNKNOWN_NUCLEOTIDE) and 0 as its frequency
     */
    KmerFreq();

    /**
     * @brief Gets a const reference to the Kmer of this KmerFreq object
     * Query method
     * @return A const reference to the Kmer of this KmerFreq object
     */
    Kmer getKmer();

    /**
     * @brief Gets the frequency of this KmerFreq object
     * Query method
     * @return The frequency of this KmerFreq object
     */
    int getFrequency();

    /**
     * @brief Sets the Kmer of this KmerFreq object.
     * Modifier method
     * @param kmer The new Kmer value for this object. Input parameter
     */
    void setKmer(Kmer kmer);

    /**
     * @brief Sets the frequency of this KmerFreq object
     * Modifier method
     * @throw std::out_of_range if @p frequency is negative
     * @param frequency the new frequency value for this KmerFreq object.
     * Input parameter
     */
    void setFrequency(int frequency);

    /**
     * @brief Obtains a string with the string and frequency of the kmer
     * in this object (separated by a whitespace).
     * Query method.
     * @return A string with the nucleotide and frequency of the kmer
     * in this object
     */
    std::string toString();
```

```
    /**
     * @brief Writes this object to the given output stream. It first writes
     * the kmer of this object (using method Kmer::write(ostream&))
     * and them the bytes of the frequency (an int value) in binary format
     * (using method ostream::write(const char* s, streamsize n))
     * Query method
     * @param outputStream An output stream where this object will be written
     */
    void write(std::ostream outputStream);

    /**
     * @brief Reads this object from the given input stream. It first reads
     * the Kmer of this object (using method Kmer::read(std::istream&) and
     * then the bytes of the frequency (an int value) in binary format (using
     * method istream::read(char* s, streamsize n))
     * Modifier method
     * @param inputStream An input stream from which this object will be read
     */
    void read(std::istream inputStream);

private:
    Kmer _kmer; ///< the Kmer object
    int _frequency; ///< the frequency
}; // end class KmerFreq

/**
 * @brief Overloading of the stream insertion operator for KmerFreq class
 * @param os The output stream to be used. Output parameter
 * @param kmerFreq the KmerFreq object. Input parameter
 * @return @p os A reference to the output stream
 */
std::ostream operator<<(std::ostream os, KmerFreq kmerFreq);

/**
 * @brief Overloading of the stream extraction operator for KmerFreq class
 * @param is The input stream to be used. Output parameter
 * @param kmerFreq the KmerFreq object. Input parameter
 * @return @p is A reference to the input stream
 */
std::istream operator>>(std::istream is, KmerFreq kmerFreq);

/**
 * @brief Overloading of the relational operator > for KmerFreq class
 * @param kmerFreq1 The first object to be compared. Input parameter
 * @param kmerFreq2 The second object to be compared. Input parameter
 * @return true if the frequency of @p kmerFreq1 is greater than that of
 * @p kmerFreq2 or if both frequencies are equals and the text of
 * @p kmerFreq1 is minor than the text of @p kmerFreq2; false otherwise
 */
bool operator>(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

/**
 * @brief Overloading of the operator < for KmerFreq class
 * @param kmerFreq1 a Kmer object. Input parameter
 * @param kmerFreq2 a Kmer object. Input parameter
 * @return true if kmerFreq1 < kmerFreq2; false otherwise
 */
bool operator<(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

/**
 * @brief Overloading of the operator == for Kmer class
 * @param kmerFreq1 a KmerFreq object. Input parameter
 * @param kmerFreq2 a KmerFreq object. Input parameter
 * @return true if the two kmers contains the same pair Kmer-frequency;
 * false otherwise
 */
bool operator==(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

/**
 * @brief Overloading of the operator != for KmerFreq class
 * @param kmerFreq1 a Kmer object. Input parameter
 * @param kmerFreq2 a Kmer object. Input parameter
 * @return true if the two kmerFreq1 are not equals (see operator==);
 * false otherwise
 */
bool operator!=(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

/**
 * @brief Overloading of the operator <= for KmerFreq class
 * @param kmerFreq1 a Kmer object. Input parameter
 * @param kmerFreq2 a Kmer object. Input parameter
 * @return true if kmerFreq1 <= kmerFreq2; false otherwise
 */
bool operator<=(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

/**
 * @brief Overloading of the operator >= for KmerFreq class
 * @param kmerFreq1 a Kmer object. Input parameter
 * @param kmerFreq2 a Kmer object. Input parameter
 * @return true if kmerFreq1 >= kmerFreq2; false otherwise
 */
bool operator>=(KmerFreq kmerFreq1, KmerFreq kmerFreq2);

#endif /* KMER_FREQ_H */
```

6.3. Profile.h

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/**
 * @file Profile.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 17 November 2023, 10:15
 */

#ifndef PROFILE_H
#define PROFILE_H


#include <iostream>
#include "KmerFreq.h"

/**
 * @class Profile
 * @brief It defines a model (profile) for a given biological species. It
 * contains a vector of pairs Kmer-frequency (objects of the class KmerFreq)
 * and an identifier (string) of the profile.
 */
class Profile {
public:

    /**
     * @brief Base constructor. It builds a Profile object with "unknown" as
     * identifier, and an empty vector of pairs Kmer-frequency. The vector will
     * have Kmer::INITIAL_CAPACITY as initial capacity.
     */
    Profile();


    /**
     * @brief It builds a Profile object with "unknown" as
     * identifier, and a vector with a size of @p size pairs
     * Kmer-frequency. The vector will also have @p size as initial capacity.
     * Each pair will be initialized as Kmer::MISSING_NUCLEOTIDE for the Kmer
     * and 0 for the frequency.
     * @throw std::out_of_range Throws a std::out_of_range exception if
     * @p size <0
     * @param size The size for the vector of kmers in this Profile.
     * Input parameter
     */
    Profile(int size);

    /**
     * @brief Copy constructor
     * @param orig the Profile object used as source for the copy. Input
     * parameter
     */
    Profile(Profile orig);

    /**
     * @brief Destructor
     */
    ~Profile();

    /**
     * @brief Overloading of the assignment operator for Profile class.
     * Modifier method
     * @param orig the Profile object used as source for the assignment. Input
     * parameter
     * @return A reference to this object
     */
    Profile operator=(Profile orig);

    /**
     * @brief Returns the identifier of this profile object.
     * Query method
     * @return A const reference to the identifier of this profile object
     */
    std::string getProfileId();

    /**
     * @brief Sets a new identifier for this profile object.
     * Modifier method
     * @param id The new identifier. Input parameter
     */
    void setProfileId(std::string id);


    /**
     * @brief Gets a const reference to the KmerFreq at the given position
     * of the vector in this object.
     * Query method
     * @param index the position to consider. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if the
     * given index is not valid
     * @return A const reference to the KmerFreq at the given position
     */
```

```
KmerFreq at(int index);

/**
 * @brief Gets a reference to the KmerFreq at the given position of the
 * vector in this object
 * Query and modifier method
 * @param index the position to consider. Input parameter
 * @throw std::out_of_range Throws an std::out_of_range exception if the
 * given index is not valid
 * @return A reference to the KmerFreq at the given position
 */
KmerFreq at(int index);

/**
 * @brief Gets the number of KmerFreq objects.
 * Query method
 * @return The number of KmerFreq objects
 */
int getSize();

/**
 * @brief Gets the capacity of the vector of KmerFreq objects.
 * Query method
 * @return The capacity of the vector of KmerFreq objects
 */
int getCapacity();

/**
 * @brief Gets the distance between this Profile object (\f$P_1\f$) and
 * the given argument object @p otherProfile (\f$P_2\f$).
 * The distance between two Profiles \f$P_1\f$ and \f$P_2\f$ is
 * calculated in the following way:
 *
 * \f$d = \frac{ \sum_{kmer_i(P_1)} | rank_{kmer_i(P_1)}^{P_1} -
 * rank_{kmer_i(P_1)}^{P_2} | }{size(P_1) * size(P_2) }\f$,
 *
 * where \f$kmer_i(p_j)\f$ is the kmer \f$i\f$ of the Profile \f$p_j,
 * j \in \{1, 2\}\f$ and \f$rank_{kmer_i(p_j)}^{p_k}\f$ is the ranking
 * of the kmer \f$i\f$ of the Profile \f$p_j, j \in \{1, 2\}\f$ in the
 * Profile \f$p_k\f$.
 *
 * The rank of a kmer is the position in which it
 * appears in the list of KmerFreq. We consider 0 as the
 * first position (rank equals to 0). When calculating
 * \f$rank_{kmer_i(P_1)}^{P_2}\f$, if the kmer \f$kmer_i(P_1)\f$
 * does not appears in the Profile \f$P_2\f$ we consider that the rank
 * is equals to the size of Profile \f$P_2\f$.
 * Query method
 * @param otherProfile A Profile object. Input parameter
 * @pre The list of kmers of this and otherProfile should be ordered in
 * decreasing order of frequency. This is not checked in this method.
 * @throw Throws a std::invalid_argument exception if the implicit object
 * (*this) or the argument Profile object are empty, that is, they do not
 * have any kmer.
 * @return The distance between this Profile object and the given
 * argument @p otherProfile.
 */
double getDistance(Profile otherProfile);

/**
 * @brief Searchs the given kmer in the list of kmers in this
 * Profile, but only in positions from initialPos to finalPos
 * (both included). If found, it returns the position where it was found.
 * If not, it returns -1. We consider that position 0 is the first kmer in
 * the list of kmers and this->getSize()-1 the last kmer.
 * Query method
 * @param kmer A kmer. Input parameter
 * @param initialPos initial position where to do the search. Input parameter
 * @param finalPos final position where to do the search. Input parameter
 * @return If found, it returns the position where the kmer
 * was found. If not, it returns -1
 */
int findKmer(Kmer kmer, int initialPos, int finalPos);

/**
 * @brief Searchs the given kmer in the list of kmers in this
 * Profile. If found, it returns the position where it was found. If not,
 * it returns -1. We consider that position 0 is the first kmer in the
 * list of kmers and this->getSize()-1 the last kmer.
 * Query method
 * @param kmer A kmer. Input parameter
 * @return If found, it returns the position where the kmer
 * was found. If not, it returns -1
 */
int findKmer(Kmer kmer);

/**
 * @brief Obtains a string with the following content:
 * - In the first line, the profile identifier of this Profile
 * - In the second line, the number of kmers in this Profile
 * - In the following lines, each one of the pairs kmer-frequency
 * (separated by a whitespace).
 * Query method
 * @return A string with the number of kmers and the list of pairs of
 * kmer-frequency in the object
 */
std::string toString();

/**
 * @brief Sorts the vector of KmerFreq in decreasing order of frequency.
```

```
     * If two KmerFreq objects have the same frequency, then the alphabetical
     * order of the kmers of those objects will be considered (the object
     * with a kmer that comes first alphabetically will appear first).
     * Modifier method
     */
    void sort();

    /**
     * @brief Saves this Profile object in the given file.
     * Query method
     * @param fileName A c-string with the name of the file where this Profile
     * object will be saved. Input parameter
     * @param mode The mode to use to save this Profile object: 't' for text
     * mode and 'b' for binary mode. Input parameter
     * @throw std::invalid_argument Throws a std::invalid_argument exception
     * if the given @mode is not valid ('t' or 'b')
     * @throw std::ios_base::failure Throws a std::ios_base::failure exception
     * if the given file cannot be opened or if an error occurs while writing
     * to the file
     */
    void save(char fileName[], char mode = 't');

    /**
     * @brief Loads into this object the Profile object stored in the given
     * file. Note that this method should remove any Kmer-frequency pairs that
     * this object previously contained.
     * Modifier method
     * @param fileName A c-string with the name of the file where the Profile
     * will be stored. Input parameter
     * @throw std::out_of_range Throws a std::out_of_range exception if the
     * number of kmers in the given file, cannot be allocated in this Profile
     * because it exceeds the maximum capacity or if the number of kmers read
     * from the given file is negative.
     * @throw std::ios_base::failure Throws a std::ios_base::failure exception
     * if the given file cannot be opened or if an error occurs while reading
     * from the file
     * @throw throw std::invalid_argument Throws a std::invalid_argument if
     * an invalid magic string is found in the given file
     */
    void load(char fileName[]);

    /**
     * @brief Appends a copy of the given KmerFreq to this Profile object.
     * If the kmer is found in this object, then its frequency is increased
     * with the one of the given KmerFreq object. If not, a copy of the
     * given KmerFreq object is appended to the end of the list of
     * KmerFreq objects in this Profile.
     * Modifier method
     * @thow std::out_of_range Throws a std::out_of_range exception in case
     * that a new element must be appended to the end of the array and the
     * number of elements in the array of KmerFreq is equals to the capacity
     * of that array. In that case, the array is full, and no more elements
     * can be appended to the array.
     * @param kmerFreq The KmerFreq to append to this object. Input paramether
     */
    void append(KmerFreq kmerFreq);

    /**
     * @brief Normalizes the Kmers of the vector of KmerFreq in this object.
     * That is, for each Kmer in the vector, all its characters are converted
     * to uppercase. Then, invalid characters are replaced by the
     * MISSING_NUCLEOTIDE value.
     *
     * If after the previous normalization process of every kmer, identical kmers
     * are obtained, these will be merged into the first identical kmer by
     * adding their frequencies.
     * For example, suposse the following list of kmers:
4
Ct 5
hG 4
nG 1
cT 4
     *
     * After the process of normalization of every kmer, we obtain the following
     * list of kmers:
     *
4
CT 5
_G 4
_G 1
CT 4
     *
     * The final step will transform the list into:
2
CT 9
_G 5
     *
     * Modifier method
     * @param validNucleotides a string with the list of characters (nucleotides)
     * that should be considered as valid. Input parameter
     */
    void normalize(std::string validNucleotides);

    /**
     * @brief Deletes the KmerFreq object from the vector of KmerFreq in this
     * object at the position @p pos. We consider that the first element has
     * position 0, and the last element position size()-1.
     * Modifier method
     * @param pos The index of the position to be deleted. Input parameter
     * @throw std::out_of_range Throws an std::out_of_range exception if @p pos
```

```
     * is not in the range from 0 to size()-1 (both included).
     */
    void deletePos(int pos);

    /**
     * @brief Deletes the KmerFreq objects from the vector of KmerFreq in this
     * object which verifies one the following two criteria:
     *  -# The argument deleteMissing is true and the Kmer contains an unknown
     * nucleotide
     *  -# The frequency is less or equals to @p lowerBound.
     *
     * Note that the number of elements in the argument array could be modified.
     * Modifier method
     * @param deleteMissing A bool value that indicates whether kmers with any
     * unknown nucleotide should be removed. This parameter is false by default.
     * Input parameter
     * @param lowerBound An integer value that defines which KmerFreq objects
     * should be deleted from the vector of KmerFreq in this object.
     * KmerFreq objects with a frequency less or equals to this value, are
     * deleted. This parameter has zero as default value.
     * Input parameter
     */
    void zip(bool deleteMissing=false, int lowerBound = 0);

    /**
     * @brief Appends to this Profile object, the list of pairs
     * kmer-frequency objects contained in the Profile @p profile. This
     * method uses the method append(const KmerFreq& kmerFreq) to
     * append the pairs kmer-frequency contained in the argument
     * Profile @p profile
     * Modifier method
     * @param profile A Profile object. Input parameter
     * @deprecated This method could be removed in future versions of this
     * class. Use the operator += instead.
     */
    void join(Profile profile);

    /**
     * @brief Overloading of the [] operator for Profile class
     * @param index index of the element. Input parameter
     * Query method
     * @return A const reference to the KmerFreq object at position @p index
     */
    KmerFreq operator[](int index);

    /**
     * @brief Overloading of the [] operator for Profile class
     * @param index index of the element. Input parameter
     * Query and modifier method
     * @return A reference to the KmerFreq object at position @p index
     */
    KmerFreq operator[](int index);

    /**
     * @brief Overloading of the += operator with a KmerFreq parameter.
     * It appends to this Profile object a copy of the given KmerFreq.
     * If the kmer is found in this object, then its frequency is increased
     * with the one of the given KmerFreq object. If not, a copy of the
     * given KmerFreq object is appended to the end of the list of
     * KmerFreq objects in this Profile.
     * Modifier method
     * @param kmerFreq The KmerFreq object to append to this object.
     * Input parameter
     * @return A reference to this object.
     */
    Profile operator+=(KmerFreq kmerFreq);

    /**
     * @brief Overloading of the += operator with a Profile parameter.
     * For each kmer in the given Profile @p profile, if that kmer is
     * found in this object, then its frequency is increased with the one in
     * @p profile. If not, a copy of the kmer-pair is appended to the end
     * of the list of KmerFreq objects in this Profile.
     * Modifier method
     * @param profile A Profile object. Input parameter
     * @return A reference to this object.
     */
    Profile operator+=(Profile profile);

private:
    std::string _profileId; ///< Profile identifier
    KmerFreq* _vectorKmerFreq; ///< Dynamic array of KmerFreq
    int _size; ///< Number of used elements in the dynamic array _vectorKmerFreq
    int _capacity; ///< Number of reserved elements in the dynamic array _vectorKmerFreq

    static const int INITIAL_CAPACITY=10; ///< Default initial capacity for the dynamic array
      _vectorKmerFreq. Should be a number >= 0
    static const int BLOCK_SIZE=20; ///< Size of new blocks in the dynamic array _vectorKmerFreq

    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files
    static const std::string MAGIC_STRING_B; ///< A const string with the magic string for binary files
};

/**
 * @brief Overloading of the stream insertion operator for Profile class
 * @param os The output stream to be used. Output parameter
 * @param profile the Profile object. Input parameter
 * @return @p os A reference to the output stream
 */
std::ostream operator<<(std::ostream os, Profile profile);
```

```
/**
 * @brief Overloading of the stream extraction operator for Profile class.
 * Note that this operator should remove any Kmer-frequency pairs that
 * the argument Profile object previously contained.
 * @throw std::out_of_range Throws a std::out_of_range if the number of kmers
 * read from the file is negative.
 * @param is The input stream to be used. Output parameter
 * @param profile the Profile object. Output parameter
 * @return @p is A reference to the input stream
 */
std::istream operator>>(std::istream is, Profile profile);

#endif /* PROFILE_H */
```

6.4. KmerCounter.h

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/*
 * @file:   KmerCounter.h
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 7 November 2023, 14:00
 */

#ifndef KMER_COUNTER_H
#define KMER_COUNTER_H

#include <string>

#include "Profile.h"

/**
 * @class KmerCounter
 * @brief It is a helper class used to calculate the frequency of each kmer in
 * a text file.
 * It consists of a matrix of integers. Each element in the matrix contains
 * the frequency of the kmer that is defined by its row and column: the kmer formed
 * taking the nucleotides defined by the row and column of that element.
 *
 * This class has a private data member string _validNucleotides to contain the set of
 * possible nucleotides in a kmer and a private data member
 * string _allNucleotides that contains the character that define the missing
 * nucleotide (Kmer::MISSING_NUCLEOTIDE) plus the characters in
 * _validNucleotides. Also, it contains a private data member int _k that
 * defines the number of nucleotides in each kmer.
 *
 * For example, if "ACGT" are the valid nucleotides, then _validNucleotides will
 * be "ACGT", _allNucleotides will be "_ACGT".
 * If the number of nucleotides in each kmer is 5, then _k will be 5. The first
 * 3 nucleotides of each kmer will be associated to index the rows of the
 * frequency matrix and the last 2 nucleotides to index the columns. For
 * example, for the kmer "ACATG", "ACA" will be used to index the row and
 * "TG" to index the column.
 * In this example, the first row corresponds to "___", the second to "__A",
 * the third to "__C" and so on. The first column corresponds to "__",
 * the second to "_A", the third to "_C" and so on.
 *
 *
 * When searching for kmers in a text file, if we find a character that
 * does not belong to the set of valid nucleotides, it will be replaced by the
 * character Kmer::MISSING_NUCLEOTIDE in the corresponding kmer
 */
class KmerCounter {
public:

    /**
     * A const c-string with the set of characters that are considered as
     * part of a word. Any other character will be considered a separator
     * Only lowercase characters are included in this string. This c-string
     * is used in the constructor of this class, as the default value to assign
     * to the field _validNucleotides
     */
    static const char* const DEFAULT_VALID_NUCLEOTIDES;

    /**
     * @brief Constructor of the class. This object uses a matrix of
     * integers that contains the frequency for each kmer. Each element of the
     * matrix will be initialized with 0. The constructor also initializes the
     * private data _k, _validNucleotides and _allNucleotides. _allNucleotides
     * is initialized with the characters
     * Kmer::MISSING_NUCLEOTIDE + validNucleotides
     * @param k The number of nucleotides in each kmer. Input parameter
     * @param validNucleotides The set of nucleotides (characters) that are
     * considered as part of a kmer. Input parameter
     */
    KmerCounter(int k=5,
        std::string validNucleotides = DEFAULT_VALID_NUCLEOTIDES);
```

```
    /**
     * @brief Copy constructor
     * @param orig the KmerCounter object used as source for the copy.
     * Input parameter
     */
    KmerCounter(KmerCounter orig);

    /**
     * @brief Destructor
     */
    ~KmerCounter();

    /**
     * @brief Returns the number of nucleotides that can be part of a
     * kmer, that is, the number of characters in the private data
     * _allNucleotides. For example, if "_ACGT" are the set of all nucleotides,
     * then this method will return 5.
     * Query method
     * @return The number of nucleotides that can be part of a kmer
     */
    int getNumNucleotides();

    /**
     * @brief Returns the number of nucleotides in each kmer
     * Query method
     * @return The number of nucleotides in each kmer
     */
    int getK();

    /**
     * @brief Returns the number of different kmers that can be built using
     * @p _k nucleotides (including the missing nucleotide)
     * Query method
     * @return The number of different kmers that can be built using
     * @p _k nucleotides
     */
    int getNumKmers();


    /**
     * @brief Gets the number of kmers with a frequency greater than 0
     * Query method
     * @return the number of kmers with a frequency greater than 0
     */
    int getNumberActiveKmers();

    /**
     * @brief Obtains a string with the following content:
     * - In the first line, the content of the private data _allNucleotides and
     * the value of _k (number of nucleotides in each kmer) separated by a
     * whitespace.
     * - In the following lines, each one of the rows in the frequency matrix
     * (frequencies separated by a whitespace).
     * Query method
     * @return A string with the content of this object
     */
    std::string toString();

    /**
     * @brief Increases the current frequency of the given kmer using the value
     * provided by @p frequency. If the argument @p frequency is not
     * provided, then 1 is added to the current frequency of the kmer.
     * Modifier method
     * @throw std::invalid_argument This method throws an
     * std::invalid_argument exception if the given kmer contains any
     * invalid nucleotide.
     * @param kmer The kmer in which the frequency will be modified.
     * Input parameter
     * @param frequency The quantity that will be added to the current
     * frequency. Input parameter
     */
    void increaseFrequency(Kmer kmer, int frequency = 1);

    /**
     * @brief Overloading of the assignment operator.
     * Modifier method
     * @param orig the KmerCounter object used as source for the assignment.
     * Input parameter
     * @return A reference to this object
     */
    KmerCounter operator=(KmerCounter orig);

    /**
     * @brief Overloading of the operator +=. It increases the current
     * frequencies of the kmers of this object with the frequencies of the
     * kmers of the given object.
     * Modifier method
     * @param kc a KmerCounter object. Input parameter
     * @throw std::invalid_argument This method throws an
     * std::invalid_argument exception if the given argument @p kc has a different
     * set of nucleotides or a different K (number of nucleotides in
     * kmers).
     * @return A reference to this object
     */
    KmerCounter operator+=(KmerCounter kc);

    /**
     * @brief Reads the given text file and calculates the frequencies of each
     * kmer in that file. This method normalizes each found Kmer and then
```

```
     * sum 1 at the corresponding element of the frequency matrix.
     * This method sets to zero the frequency of each kmer before starting to
     * calculate frequencies. In this way, if this method is called twice
     * consecutively, then this KmerCounter will contain only the frecuencies
     * calculated in the last call.
     * Modifier method
     * @param fileName The name of the file to process. Input parameter
     * @throw std::ios_base::failure Throws a std::ios_base::failure if the
     * given file cannot be opened
     */
    void calculateFrequencies(char* fileName);

    /**
     * @brief Builds a Profile object from this KmerCounter object. The
     * Profile will contain the kmers and frequencies for those one with
     * a frequency greater than 0.
     * Note that the resulting Profile could contain kmers with the
     * missing nucleotide. If you want to eliminate them, you must zip
     * (using Profile::zip(true) method) the returned Profile after calling
     * to this method.
     * Also note that this method does not obtain a Profile with an ordered
     * vector of kmers. If you need an ordered vector of kmers, you must sort
     * (using Profile::sort() method) the returned Profile after calling to
     * this method.
     * Query method
     *
     * @return A Profile object from this KmerCounter object
     */
    Profile toProfile();

private:
    int** _frequency; ///< 2D matrix with the frequency of each kmer

    int _k; ///< Value of K (number of nucleotides in each kmer)

    /**
     * Set of characters that define the possible nucleotides of a kmer. Any other
     * character will be considered an invalid nucleotide. Only uppercase
     * characters are included in this string
     */
    std::string _validNucleotides;

    /**
     * Set with the character representing a missing (unknown) nucleotide
     * (Kmer::MISSING_NUCLEOTIDE) and the set of valid nucleotides.
     */
    std::string _allNucleotides;

    /**
     * @brief Returns the numbers of rows of the matrix in this object.
     * Query method
     * @return The numbers of rows of the matrix in this object
     */
    int getNumRows();

    /**
     * @brief Returns the numbers of columns of the matrix in this object.
     * Query method
     * @return The numbers of columns of the matrix in this object
     */
    int getNumCols();

    /**
     * Returns the index (a row or a column in the frequency matrix)
     * corresponding to the given argument string. Note that this method
     * will be used calling it with a string that will contain a half of a Kmer.
     * Query method
     * @param kmer A string with a set of nucleotides. Input parameter
     * @return Returns the index (a row or a column in the frequency matrix)
     * corresponding to the given argument string.
     * It returns -1 if some of the nucleotides in @p kmer does not belong to
     * the set _allNucleotides
     */
    int getIndex(const std::string& kmer) const;

    /**
     * Returns a string with the nucleotides that define the given index (a
     * row or a column of the frequency matrix).
     * Query method
     *
     * @param index An integer, row or column of the frequency matrix.
     * Input parameter
     * @param nCharacters Number of characters that will be in the returned
     * string. If defines the number of nucleotides associated with a row or
     * column. Input parameter
     * @return a string with the nucleotides that define the given index
     * (row or a column of the frequency matrix).
     */
    std::string getInvertedIndex(int index, int nCharacters) const;

    /**
     * @brief Obtains the row and column for the frequency of the given
     * argument Kmer. -1 will be assigned to row or column if some of the
     * nucleotides in @p kmer does not belong to the set _allNucleotides
     * Query method
     * @param kmer A Kmer. Input parameter
     * @param row The row in the frequency matrix corresponding to Kmer @p kmer.
     * Output parameter
     * @param column The column in the frequency matrix corresponding to
     * Kmer @p kmer. Output parameter
```

```
     */
    void getRowColumn(Kmer kmer, int row, int column);

    /**
     * @brief Returns the Kmer that is defined by the provided row and column
     * Query method
     * @param row A row in the frequency matrix (int value from 0 to nRows-1).
     * Input parameter
     * @param column A column in the frequency matrix (int value from 0 to
     * nColumns-1). Input parameter
     * @throw std::invalid_argument This method throws an
     * std::invalid_argument exception if the given row or column are out
     * of the correct bounds
     * @return the Kmer that is defined by the provided row and column
     */
    Kmer getKmer(int row, int column);

    /**
     * @brief Sets the frequency of each kmer to 0, that is, it fills with 0 the
     * matrix of frequencies
     * Modifier method
     */
    void initFrequencies();

    /**
     * @brief Overloading of the () operator to access to the element at a
     * given position.
     * Query method
     * @param row Row of the element. Input parameter
     * @param column Column of the element. Input parameter
     * @return A const reference to the element at the given position
     */
    int operator()(int row, int column);

    /**
     * @brief Overloading of the () operator to access to the element at a
     * given position.
     * Query and modifier method
     * @param row Row of the element. Input parameter
     * @param column Column of the element. Input parameter
     * @return A reference to the element at the given position
     */
    int operator()(int row, int column);
};

#endif /* KMER_COUNTER_H */
```

6.5. LEARN.cpp

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/**
 * @file LEARN.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 22 December 2023, 10:00
 */


/**
 * Shows help about the use of this program in the given output stream
 * @param outputStream The output stream where the help will be shown (for example,
 * cout, cerr, etc)
 */
void showEnglishHelp(ostream& outputStream) {
    outputStream << "ERROR in LEARN parameters" << endl;
    outputStream << "Run with the following parameters:" << endl;
    outputStream << "LEARN [-t|-b] [-k kValue] [-n nucleotidesSet] [-p profileId] [-o outputFilename] <file1
      .dna> [<file2.dna> <file3.dna> .... ]" << endl;
    outputStream << endl;
    outputStream << "Parameters:" << endl;
    outputStream << "-t|-b: text mode or binary mode for the output file (-t by default)" << endl;
    outputStream << "-k kValue: number of nucleotides in a kmer (5 by default)" << endl;
    outputStream << "-n nucleotidesSet: set of possible nucleotides in a kmer (ACGT by default). "
                 << "Note that the characters should be provided in uppercase" << endl;
    outputStream << "-p profileId: profile identifier (unknown by default)" << endl;
    outputStream << "-o outputFilename: name of the output file (output.prf by default)" << endl;
    outputStream << "<file1.dna> <file2.dna> <file3.dna> ....: names of the input files (at least one is
      mandatory)" << endl;
    outputStream << endl;
    outputStream << "This program learns a profile model from a set of "<<
            "input DNA files <file1.dna> <file2.dna> <file3.dna> ...." << endl;
    outputStream << endl;
}

/**
 * This program learns a Profile model from a set of input DNA files (file1.dna,
 * file2.dna, ...). The learned Profile object is then zipped (kmers with any
 * missing nucleotide or with frequency equals to zero will be removed)
```

```
 * and ordered by frequency and saved in
 * the file outputFilename (or output.prf if the output file is not provided).
 *
 * Running sintax:
 * > LEARN [-t|-b] [-k kValue] [-n nucleotidesSet] [-p profileId] [-o outputFilename] <file1.dna> [<file2.
     dna> <file3.dna> ....]
 *
 * Running example:
 * > LEARN -k 2 -p bug -o /tmp/unknownACGT.prf ../Genomes/unknownACGT.dna
 *
 * > cat /tmp/unknownACGT.prf
MP-KMER-T-1.0
bug
7
GG 2
AC 1
AG 1
AT 1
CC 1
GA 1
TA 1
 *
 * @param argc The number of command line parameters
 * @param argv The vector of command line parameters (cstrings)
 * @return 0 If there is no error; a value > 0 if error
 */
int main(int argc, char *argv[]) {
    // Process the main() arguments

    // Loop to calculate the kmer frecuencies of the input genome files using
    // a KmerCounter object

    // Obtain a Profile object from the KmerCounter object

    // Zip the Profile object

    // Sort the Profile object

    // Save the Profile object in the output file
}
```

6.6. CLASSIFY.cpp

```
/*
 * Metodología de la Programación: Kmer5
 * Curso 2023/2024
 */

/**
 * @file CLASSIFY.cpp
 * @author Silvia Acid Carrillo <acid@decsai.ugr.es>
 * @author Andrés Cano Utrera <acu@decsai.ugr.es>
 * @author Luis Castillo Vidal <L.Castillo@decsai.ugr.es>
 * @author Javier Martínez Baena <jbaena@ugr.es>
 *
 * Created on 22 December 2023, 10:00
 */

/**
 * Shows help about the use of this program in the given output stream
 * @param outputStream The output stream where the help will be shown (for example,
 * cout, cerr, etc)
 */
void showEnglishHelp(ostream& outputStream) {
    outputStream << "ERROR in CLASSIFY parameters" << endl;
    outputStream << "Run with the following parameters:" << endl;
    outputStream << "CLASSIFY [-k kValue] [-n nucleotidesSet] <file.dna> <profile1.prf> [<profile2.prf> <
      profile3.prf> ....]" << endl;
    outputStream << endl;
    outputStream << "Parameters:" << endl;
    outputStream << "-k kValue: number of nucleotides in a kmer (5 by default)" << endl;
    outputStream << "-n nucletiodesSet: set of possible nucleotides in a kmer (ACGT by default). "
                 << "It is used when learning a model for <file.dna>. "
                 << "Note that the characters should be provided in uppercase" << endl;
    outputStream << "<profile1.prf> [<profile2.prf> <profile3.prf> ....] ....: "
                 << "names of the Profile models (at least one is mandatory)" << endl;
    outputStream << endl;
    outputStream << "This program obtains the identifier of the closest profile to the input DNA file" <<
      endl;
    outputStream << endl;
}

/**
 * This program prints the profile identifier of the closest profile model
 * for an input DNA file (<file.dna>) among the set of provided models:
 * <profile1.prf>, <profile2.prf>, ...
 * The program uses the KmerCounter class to obtain a Profile for the input
 * file <file.dna>. That Profile should be zipped, to eliminate kmers with
 * any missing nucleotide, and sorted in decreasing order of frequency of
 * kmers. After that, the program compares the learned Profile with the ones
 * provided by the arguments <profile1.prf> [<profile2.prf> <profile3.prf> ....]
 * It classifies the input DNA file with the identifier of the Profile with
 * a minor distance.
 *
 * This program assumes that the profile files are already normalized and
```

```
 * sorted by frequency. This is not checked in this program. Unexpected results
 * will be obtained if those conditions are not met.
 *
 * Running sintax:
 * > CLASSIFY [-k kValue] [-n nucleotidesSet] <file.dna> <profile1.prf> [<profile2.prf> <profile3.prf> ....]
 *
 * Running example:
 * > CLASSIFY ../Genomes/human_chr6_s60000_l500000.dna ../Genomes/brewers_yeast_chrVII.s1_l500000.prf ../
      Genomes/chimpanzee_chr9_s1_l500000.prf ../Genomes/covidFullGenomeDNA.prf ../Genomes/
      drosophila_chr2L_s1_l500000.prf ../Genomes/ebolaFullGenomeDNA.prf ../Genomes/human_chr9_s10000_l500000
      .prf ../Genomes/monkeypoxFullGenomeDNA.prf ../Genomes/mouse_chr6_s3050050_l500000.prf ../Genomes/
      nematode_chrI_s1l500000.prf ../Genomes/rat_chr6_s1l500000.prf ../Genomes/zebrafish_chr6_s1l500000.prf
Distance to ../Genomes/brewers_yeast_chrVII.s1_l500000.prf (saccharomyces cerevisiae): 0.20294
Distance to ../Genomes/chimpanzee_chr9_s1_l500000.prf (pan troglodytes): 0.0643864
Distance to ../Genomes/covidFullGenomeDNA.prf (severe acute respiratory syndrome coronavirus 2): 0.194633
Distance to ../Genomes/drosophila_chr2L_s1_l500000.prf (drosophila melanogaster): 0.189238
Distance to ../Genomes/ebolaFullGenomeDNA.prf (ebolavirus zaire): 0.179686
Distance to ../Genomes/human_chr9_s10000_l500000.prf (homo sapiens): 0.0557804
Distance to ../Genomes/monkeypoxFullGenomeDNA.prf (monkey pox virus): 0.262987
Distance to ../Genomes/mouse_chr6_s3050050_l500000.prf (mus musculus): 0.088129
Distance to ../Genomes/nematode_chrI_s1l500000.prf (caenorhabditis elegans): 0.221075
Distance to ../Genomes/rat_chr6_s1l500000.prf (rattus norvegicus): 0.111126
Distance to ../Genomes/zebrafish_chr6_s1l500000.prf (danio rerio): 0.145231

Final decision: homo sapiens with a distance of 0.0557804
 *
 * @param argc The number of command line parameters
 * @param argv The vector of command line parameters (cstrings)
 * @return 0 If there is no error; a value > 0 if error
 */
int main(int argc, char *argv[]) {
    // Process the main() arguments

    // Calculate the kmer frecuencies of the input genome file using
    //    a KmerCounter object

    // Obtain a Profile object for the input genome from the KmerCounter object

    // Zip the for the input genome Profile object

    // Sort the for the input genome Profile object

    // Use a loop to print the distance from the input genome to
    //   each one of the provided profile models

    // Print the identifier and distance to the closest profile



}
```