UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2023/2024

Guion de prácticas

Kmer2

class Profile

Marzo de 2024

Índice

1. Definición del problema

Como ya se indicó con anterioridad, las prácticas tienen como objeto principal trabajar con secuencias de nucleótidos de genomas procedentes de individuos pertenecientes a géneros o especies diferentes. Un ejemplo de género puede ser la *drosophila*, y un ejemplo de especie la *drosophila melanogaster*. Así pues, vamos a desarrollar aplicaciones del ámbito de la bioinformática que nos permitan averiguar automáticamente la especie o el género a la que pertenece un determinado genoma.

En esta práctica vamos a implementar el modelo para una especie o género, que vamos a denominar *perfil* [1], mediante la clase Profile. Un perfil, se construye a partir de un extracto de genoma, contabilizando las frecuencias de todos los k-meros que se han hallado en el genoma fuente. Inicialmente, habrá tantos perfiles como genomas se analicen. No obstante, se pueden crear perfiles *enriquecidos* o más genéricos, una suerte de perfil aglutinado, que integre los perfiles de varios individuos de una misma especie, obtenido por fusión. Pensemos en un perfil de humano como la composición de varios individuos *Homo sapiens*.

De forma más concreta, la clase `Profile` va a contener un vector de objetos de la clase `KmerFreq`, junto con las funcionalidades que se encontraban definidas como funciones externas [2] en la versión anterior de la práctica y a la que le vamos a añadir alguna funcionalidad específica como un método para realizar la fusión de dos objetos `Profile`. Se da así, un paso más en el encapsulamiento de la clase.

Por otro lado, a partir de ahora, vamos a cambiar la forma de ejecución de nuestros programas. Se dejan de realizar lecturas desde teclado (o su equivalencia mediante redireccionamiento). Las lecturas de datos se van a realizar directamente desde ficheros, cuyos nombres, junto con argumentos adicionales, se van a especificar desde la línea de comandos, esto es, a través de los parámetros a nuestro programa.

2. Arquitectura de las prácticas

Como ya se indicó en la práctica anterior, la práctica `Kmer` se ha diseñado por etapas, las primeras contienen estructuras más sencillas, sobre las cuales se asientan otras estructuras más complejas y se van completando con nuevas funcionalidades.

---

[1] Utilizaremos la denominación perfil con la acepción de Recuperación de Información – que representa un conjunto de individuos–, contextualizado a la biología. Un perfil va a representar a un conjunto de individuos que comparten la misma taxonomía, bien nivel de especie o de género. A nivel de especie, varios individuos del género Drosophila –moscas pequeñas, algunos de sus miembros son la mosca de la fruta–; a nivel de género varios individuos de varias especies diferentes pero pertenecientes al mismo género Drosophila: Drosophila yakuba, Drosophila pseudoobscura...

[2] Las funciones externas de la práctica anterior, al convertirse en métodos de la clase `Profile`, cambiarán sus identificadores comenzando con minúscula como todos los métodos de una clase excepto los constructores.

En `Kmer2` se implementa la clase `Profile`, bloque **C** de la Figura 1 y se refactorizan como métodos de la clase `Profile` las funciones externas presentes en el módulo `ArrayKmerFreqFunctions`. Desaparece por tanto, este módulo del proyecto Kmer2.

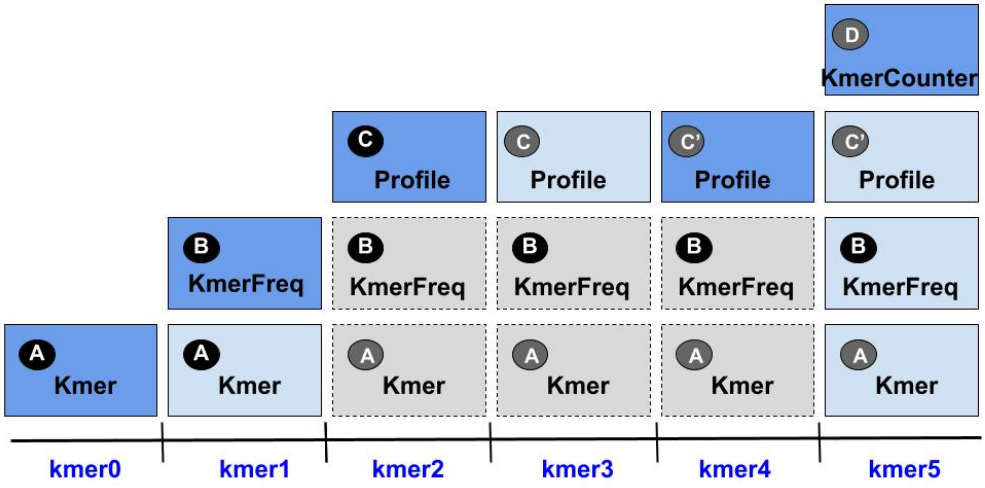

Figura 1: Arquitectura de las prácticas de MP 2024. Los cambios esenciales en las clases (cambio en estructura interna de la clase) se muestran en azul intenso; los que solo incorporan nuevas funcionalidades en azul tenue. En gris se muestran las clases que no sufren cambios en la evolución de las prácticas.

C Profile.cpp
: Implementa la clase Profile, una estructura para almacenar las frecuencias de un conjunto de k-meros. En realidad se trata de un histograma de k-meros. Será nuestro modelo para una especie o un género. En esta práctica usaremos un vector estático para almacenar la lista de pares $\{k$-mero, frecuencia$\}$. A partir de la práctica Kmer4, cambiaremos la implementación para que se use memoria dinámica.

3. Objetivos

El desarrollo de esta práctica `Kmer2` persigue los siguientes objetivos:

- Aprender a leer y escribir datos de/en un fichero de texto mediante los operadores << y >>.
- Practicar con una clase compuesta de un array de objetos y otros datos: la clase `Profile`.
- Conocer cómo leer los parámetros pasados a la función `main()` desde la línea de comandos.

4. Un fichero profile

Un fichero profile es un fichero de texto que contiene una lista de k-meros identificados como frecuentes en un determinado genoma. El fichero profile tiene el siguiente formato.

- La primera línea contiene la cadena "MP-KMER-T-1.0".
- La segunda línea contiene un identificador, una cadena con una o varias palabras, que describe a qué género (nombre simple) o especie (nombre compuesto) pertenece. Ejemplos de identificadores podrían ser *drosophila* y *drosophila melanogaster* para género y especie respectivamente.
- La tercera línea contiene el número de k-meros diferentes y sus frecuencias asociados al profile descrito.
- Las siguientes líneas contienen la lista de k-meros, con sus frecuencias.

Nota: Un fichero profile suele estar ordenado según el criterio establecido en la práctica anterior [3] aunque no es obligatorio [4]. Lo que sí es obligatorio, es la unicidad de los kmers dentro de un profile, esto es, no pueden haber duplicidades como `TTTT 78 ... TTTT 1`. No obstante, sí sería correcto `TTTT 78 ... tttt 1`.

A continuación, se muestra un ejemplo de fichero profile, en el que se pueden ver los 18 k-meros ($k = 5$) más frecuentes para el género *homo sapiens*.

Ejemplo 1 *Ejemplo de fichero profile para homo sapiens*

```
MP-KMER-T-1.0
homo sapiens
18
TGTGT 78
GTGTG 59
CCCAG 47
GGGAG 44
CTGTG 42
GGCTG 41
GGAGG 40
TGTCT 39
CAGGA 38
CCTGG 37
CCAGC 36
CAGCC 35
CCAGG 35
CCTCC 35
CTGGG 35
```

[3] La ordenación es por valor decreciente de frecuencia, y en caso de empate en frecuencia se resuelve por orden alfabético creciente de k-mero.

[4] En la carpeta `Genomes`, casi todos los ficheros profile proporcionados están ordenados según ese criterio, excepto algunos como 4pairsRNA.prf, 5pairsRNA.prf, 6pairsRNA.prf y 7pairsDNA.prf.

```
GTGTC 35
TCTGT 35
GCTGG 34
```

A continuación se muestra un extracto de un fichero profile de la especie *drosophila*.

Ejemplo 2 *Ejemplo de fichero profile para drosophila*

```
MP-KMER-T-1.0
drosophila
1021
TGCTG 42
CTGCT 40
GCTGC 37
AACAA 29
GCTGG 29
GAGGA 28
AAATA 27
GTGGC 27
AAACA 26
GAAAA 25
... hasta 1021 5-meros en total ...
```

5. Práctica a entregar

El programa a desarrollar en esta práctica tiene como objetivo leer y normalizar un número indeterminado de ficheros profile (extensión `.prf`), especificados desde la línea de comandos como argumentos del programa. Los ficheros profile deben fusionarse en un único profile resultado. Tal profile resultado debe ser comprimido, esto es, deben borrarse los k-meros con frecuencia menor o igual a 0. A continuación, debe ordernarse el profile resultado según el orden explicado en la sección 4. Finalmente, el profile resultado se salvará en un fichero `.prf` de salida.

A la hora de comprimir el profile resultado, se usará el método `zip()` de la clase `Profile` de tal forma que no se eliminen los k-meros con algún nucléotido no válido. Solo deben borrarse los k-meros con frecuencia igual a cero.

Para la elaboración de la práctica, dispone de una serie de ficheros que se encuentran en Prado en el fichero `Kmer2_nb.zip` o en el repositorio de `github`. Configure su proyecto en Netbeans de forma similar a cómo se hizo en la práctica anterior.

5.1. La clase Profile

La clase `Profile` representa el modelo para un determinado género o especie biológica. Esta práctica usa un vector estático de pares $\{k$-mero, frecuencia$\}$ para guardar la lista de pares $\{k$-mero, frecuencia$\}$, junto con algún dato adicional.

```
class Profile {
...
private:
    static const int DIM_VECTOR_KMER_FREQ = 2000; ///< The
   capacity of the array _vectorKmerFreq
    static const std::string MAGIC_STRING_T; ///< A const string
    with the magic string for text files

    std::string _profileId; ///< profile identifier
    KmerFreq _vectorKmerFreq[DIM_VECTOR_KMER_FREQ]; ///< array
   of KmerFreq
    int _size; ///< Number of used elements in _vectorKmerFreq
}; // end class Profile
```

Como podemos ver, el vector tiene una capacidad máxima para 2000 pares $\{k$-mero, frecuencia$\}$. El número de pares en un momento concreto lo determina el dato miembre `_size`.

El string `_profileId` define el nombre del género o especie del objeto profile.

La constante `MAGIC_STRING_T`, inicializada con `"MP-KMER-T-1.0"` en el fichero `Profile.cpp`, contiene la cadena de caracteres que aparece al principio de los ficheros profile.

5.2. El módulo main.cpp

Como se ha indicado anteriormente, el programa de esta práctica tiene por objetivo leer varios ficheros profile (extensión `.prf`) de entrada (con el formato indicado en la sección 4), pertenecientes a una misma especie o a un mismo género. Los profiles leídos deben normalizarse y fusionarse con el fin de obtener un único profile. El resultado se comprimirá y ordenará, y finalmente se guardará en un nuevo fichero profile con el mismo formato que los anteriores.

La sintaxis de ejecución del programa desde un terminal es:

```
Linux> dist/Debug/GNU-Linux/kmer2 <outputFile.prf> <file1.prf>
[<file2.prf> ... <filen.prf>]
```

Notación: Los `[]` indican que no es obligatorio, es opcional.

1. Todos los parámetros se pasan al programa desde la línea de órdenes.
2. El programa almacenará el contenido de cada fichero profile de entrada, en un objeto de la clase Profile, tal y como se describe en la sección 5.1.
3. Recuerde que los profiles de entrada deben normalizarse una vez leídos.
4. El programa debe recibir por la línea de órdenes, al menos dos ficheros profile: `<outputFile.prf>` y `<file1.prf>`. El primero, `<outputFile.prf>`, es de escritura, mientras que el segundo,

`<file1.prf>`, y los siguientes (caso de haberlos) son de lectura. En el primero es dónde se salva el objeto profile resultante de la fusión. Todos los ficheros profile tienen extensión **.prf**, con el formato descrito anteriormente.

5. Los ficheros profile de entrada suelen estar ordenados según el orden descrito en la sección 4 (orden decreciente por frecuencia y en caso de empate, se ordena por orden alfabético de k-mero), pero no es obligatorio que así sea.

6. Para hacer fusión, cada fichero de entrada debe tener la misma etiqueta o identificador, esto es, estar asociado a una misma especie o a un mismo género [5], aunque los k-meros que contengan puedan ser diferentes de un fichero a otro, por ejemplo, porque se han obtenido a partir de distintos genomas fuentes. El profile de salida tendrá como identificador la etiqueta de la especie o del género correspondiente al `<file1.prf>`, el primero de los ficheros de lectura. Si un profile de entrada tiene una etiqueta diferente a la de `<file1.prf>`, entonces ese profile no se incluirá en la fusión.

7. El programa debe leer todos y cada uno de los ficheros profile de entrada, indicados en la línea de órdenes y fusionarlos en un único profile. La fusión a realizar se descompone en fusión de pares de profiles, según el siguiente procedimiento:

   a) Si el k-mero leído de un fichero nuevo ya existe en el profile de salida, se suman las dos frecuencias.

   b) Si el k-mero nuevo no existe en el profile de salida, se añade al final de este profile con la frecuencia leída.

8. Antes de salvar el objeto profile de salida en un fichero `.prf`, tal objeto debe **comprimirse** y **ordenarse** (en orden decreciente de frecuencia y en caso de empate, por orden alfabético de k-mero) con el método *sort()* de la clase Profile.

5.3. Ejemplos de ejecución

En la carpeta `Genomes` del repositorio puede encontrar varios ficheros `.prf` con los que podrá hacer diferentes pruebas. Se detallan aquí algunos ejemplos.

Recuerde también que en cada práctica, se proporcionan una serie de ficheros de tests con extensión `.test` que contienen tests de integración, y que pueden usarse con la script `runTests.sh` para ejecutar automáticamente tales tests de integración.

Ejemplo 3 *Faltan argumentos para la ejecución del programa:*

```
Linux>dist/Debug/GNU-Linux/kmer2
Linux>dist/Debug/GNU-Linux/kmer2 ../Genomes/6pairsDNA.prf
```

[5] Especie = apellido1 apellido2 mientras que Género = apellido1.

La salida del programa será la siguiente:

```
ERROR in Kmer2 parameters
Run with the following parameters:
kmer2 <outputFile.prf> <file1.prf> [<file2.prf> ... <filen.prf>]

Parameters:
<outputFile.prf>: output file where the join profile will be saved
<file1.prf> [<file2.prf> ... <filen.prf>]: input profile files

This program obtains a join profile (<outputFile.prf>) using the provided
↪  input profiles <file1.prf> [<file2.prf> ... <filen.prf>]
```

De ahora en adelante en este guion, prescindiremos del path completo: `dist/Debug/GNU-Linux/` para la ejecución del programa kmer2 y lo mostramos de forma abreviada como `kmer2`.

Ejemplo 4 *Un solo fichero profile de entrada (no hay fusión) que no estaba normalizado:*

```
Linux>kmer2 /tmp/6pairsDNA.prf.out ../Genomes/6pairsDNA.prf
```

El contenido del fichero `6pairsDNA.prf` *es el siguiente:*

```
1 MP-KMER-T-1.0
2 red bug
3 6
4 Cc 3
5 GC 3
6 cu 2
7 ug 2
8 cG 1
9 gu 1
```

El contenido del fichero `6pairsDNA.prf.out` *obtenido al ejecutar el programa sería el siguiente:*

```
MP-KMER-T-1.0
red bug
6
CC 3
GC 3
C_ 2
_G 2
CG 1
G_ 1
```

Como puede ver en el anterior ejemplo, el fichero profile obtenido como salida (`6pairsDNA.prf.out`) es similar al de entrada, excepto que el resultado ha sido normalizado, lo que hace que aparezca el carácter '_' en algún k-mero. El profile de entrada ya estaba comprimido y ordenado, por lo que estos pasos no han tenido efecto en el resultado.

Ejemplo 5 *Un solo fichero profile de entrada que no estaba normalizado, ni ordenado:*

```
Linux>kmer2 /tmp/7pairsDNA.prf.out ../Genomes/7pairsDNA.prf
```

El contenido del fichero `7pairsDNA.prf` *es el siguiente:*

```
MP-KMER-T-1.0
red bug
7
Ct 5
CG 4
aG 2
CC 0
Gt 1
cT 4
at 4
```

El contenido del fichero `7pairsDNA.prf.out` *obtenido al ejecutar el programa sería el siguiente:*

```
MP-KMER-T-1.0
red bug
5
CT 9
AT 4
CG 4
AG 2
GT 1
```

Como puede verse, el resultado ha sido normalizado y ordenado. También, la compresión ha hecho que se elimine el par $\{$CC 0$\}$ al tener frecuencia igual a cero.

Ejemplo 6 *Se hace fusión de dos ficheros profile del mismo género (red bug), donde todos los k-meros resultantes de normalizar son comunes:*

```
Linux>kmer2 /tmp/6pairs_5pairsDNA.prf ../Genomes/6pairsDNA.prf
↪  ../Genomes/5pairsDNA.prf
```

Proceso seguido:

```
Abre el fichero 6pairsDNA.prf
Profile detectado: red bug
Lee 6 k-meros
Normaliza
Abre el fichero 5pairsDNA.prf
Profile detectado: red bug
Lee 5 k-meros
Normaliza
Realiza fusión. Total k-meros: 6
Realiza zip
Ordena
Salva en fichero 6pairs_5pairsDNA.prf
```

El fichero obtenido `6pairs_5pairsDNA.prf` tendría el siguiente contenido:

```
MP-KMER-T-1.0
red bug
6
C_ 7
CG 5
CC 4
_G 4
GC 3
G_ 2
```

El resultado, es un profile de `red bug` que contiene el mismo número de bímeros y dónde se han acumulado las frecuencias.

Ejemplo 7 *Se hace fusión de dos ficheros profile del mismo género (red bug), donde hay 6 bímeros no comunes:*

```
Linux>kmer2 /tmp/12pairsDNA.prf ../Genomes/6pairsDNA.prf
↪  ../Genomes/6pairsbisDNA.prf
```

El proceso que sigue el programa para obtener el profile resultado es el siguiente:

```
Abre el fichero 6pairsDNA.prf
Profile detectado: red bug
Lee 6 k-meros
Normaliza
Abre el fichero 6pairsbisDNA.prf
Profile detectado: red bug
Lee 6 k-meros
Normaliza
Realiza fusión. Total k-meros: 12
Realiza zip
Ordena
Salva en fichero 12pairsDNA.prf
```

El fichero obtenido `12pairsDNA.prf` tendría el siguiente contenido:

```
MP-KMER-T-1.0
red bug
12
T_ 6
AA 4
AT 4
TA 4
CC 3
GC 3
CT 2
C_ 2
_G 2
CG 1
G_ 1
_T 1
```

Los k-meros no comunes son `T_,AA,AT,TA,CT,_T`, cuyos respectivos valores de frecuencia coinciden con los valores que tenían en el archivo de profile origen de cada k-mero.

Ejemplo 8 *Se pretende hacer fusión de varios ficheros profile pero que tienen género diferente, por lo que solo los dos primeros se incluirán en la fusión:*

```
Linux>kmer2 /tmp/what.prf ../Genomes/6pairsDNA.prf
↪  ../Genomes/6pairsbisDNA.prf ../Genomes/human01.prf
```

Los ficheros `6pairsDNA.prf` *y* `6pairsbisDNA.prf` *tienen red bug como identificador. El fichero* `human01.prf` *sin embargo tiene homo sapiens, por lo que no será incluido en la fusión*

En este ejemplo, el profile resultado es el mismo que el del ejemplo 7.

5.4. Para la entrega

La práctica deberá ser entregada en Prado, en la fecha que se indica en cada entrega, y consistirá en un fichero ZIP del proyecto, `Kmer2.zip`. Se procederá como en prácticas anteriores.

6. Código para la práctica

6.1. Profile.h

```
/*
 * Metodología de la Programación: Kmer2
 * 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 29 January 2023, 11:00
 */

#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.
     */
    Profile();


    /**
     * @brief It builds a Profile object with "unknown" as
     * identifier, and a vector with a size of @p size pairs Kmer-frequency.
     * 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 or @p size > @p DIM_VECTOR_KMER_FREQ
     * @param size The size for the vector of kmers in this Profile.
     * Input parameter
     */
    Profile(int size);


    /**
     * @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 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
     * @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[])

    /**
     * @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
     */
    void join(Profile profile);

private:
    static const int DIM_VECTOR_KMER_FREQ = 2000; ///< The capacity of the array _vectorKmerFreq
    static const std::string MAGIC_STRING_T; ///< A const string with the magic string for text files

    std::string _profileId; ///< profile identifier
    KmerFreq _vectorKmerFreq[DIM_VECTOR_KMER_FREQ]; ///< array of KmerFreq
    int _size; ///< Number of used elements in _vectorKmerFreq
};

#endif /* PROFILE_H */
```

6.2. main.cpp

```cpp
/*
 * Metodología de la Programación: Kmer2
 * Curso 2023/2024
 */

/*
 * File:   main.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 31 October de 2023, 14:30
 */

#include <iostream>

#include "Profile.h"

using namespace std;


/**
 * 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 Kmer2 parameters" << endl;
    outputStream << "Run with the following parameters:" << endl;
    outputStream << "kmer2 <outputFile.prf> <file1.prf> [<file2.prf> ... <filen.prf>] " << endl;
    outputStream << endl;
    outputStream << "Parameters:" << endl;
    outputStream << "<outputFile.prf>: output file where the join profile"
            << " will be saved" << endl;
    outputStream << "<file1.prf> [<file2.prf> ... <filen.prf>]: input profile files" << endl;
    outputStream << endl;
    outputStream << "This program obtains a join profile (<outputFile.prf>)"
            << " using the provided input profiles <file1.prf> [<file2.prf> ... <filen.prf>]" << endl;
    outputStream << endl;
}

/**
 * This program reads and normalizes an undefined number of Profile objects
 * from the input files passed as parameters to main(). These normalized Profile
 * objects are used to obtain the union of them, the join Profile. The join
 * Profile is then zipped (pairs with frequency less or equals to zero are
 * deleted) and sorted by decreasing order of frequency. In that order,
 * if there is any tie in frequencies, then alphabetical order of kmers is
 * applied. Finally, the resulting sorted Profile is saved in the
 * output file.
 * The program must receive at least an input file and an output file.
 * The output Profile will have as profile identifier, the one of the first
 * input file (<file1.prf>).
 * If an input file <file*.prf> has a profile identifier different from the one
 * of the first file (<file1.prf>), then it will not be included in the union.
 *
 * Running sintax:
 * > kmer2 <outputFile.prf> <file1.prf> [<file2.prf> ... <filen.prf>]
 *
 * Running example:
 * > kmer2 /tmp/outputFile.prf ../Genomes/5pairsRNA.prf ../Genomes/6pairsRNA.prf
 *
 * > cat /tmp/outputFile.prf
MP-KMER-T-1.0
red bug
7
C_ 6
AG 4
CC 4
CG 4
GC 2
G_ 2
_G 2
 */
int main(int argc, char* argv[]) {
    // This string contains the list of nucleotides that are considered as
    // valid within a genetic sequence. The rest of characters are considered as
    // unknown nucleotides
    const string VALID_NUCLEOTIDES = "ACGT";


    // Check if the number of running arguments is correct, otherwise call to
    // showEnglishHelp(cerr) and end main()

    // Load and normalize the first input Profile file

    // Use a loop to load, normalize and obtain the join with the rest of Profiles

    // Zip the resulting Profile

    // Sort the zipped Profile

    // Save the final Profile to the output file

    return 0;
}
```