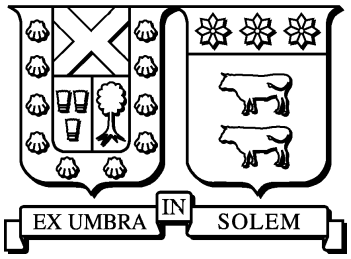
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



"ALGORITMO PARA EL CÁLCULO DE FRAGMENTOS DE PROTEÍNAS EN LOS ORGANISMOS SECUENCIADOS"

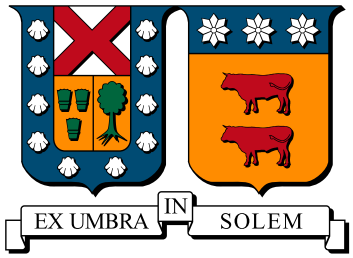
FELIPE NICOLÁS ARAYA BARRERA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: LIOUBOV DOMBROVSKAIA

DICIEMBRE 2017

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



"ALGORITMO PARA EL CÁLCULO DE FRAGMENTOS DE PROTEÍNAS EN LOS ORGANISMOS SECUENCIADOS"

FELIPE NICOLÁS ARAYA BARRERA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: LIOUBOV DOMBROVSKAIA
PROFESOR CORREFERENTE: DIEGO ARROYUELO BILLIARDI

DICIEMBRE 2017

Agradecimientos

Agradezco a todos aquellos que han estado conmigo durante este largo proceso, que comenzó por el año 2011.

Agradezco a mis padres, que me han ayudado, aconsejado, retado y levantado de tropezones, son mi bastión, mi soporte ante la vida y el futuro, los amo mucho.

Agradezco a toda mi familia, en especial a mis abuelos, a Hilda y Juan que están en el cielo y a Carlos y Norma que siguen entregándonos mucho amor. A mis tíos y sus hijos que viven en Santiago, gracias por los 4 años de convivencia y aprendizaje que me brindaron y les pido perdón por los errores que cometí. A mi prima Karla por ayudarme a mí y a mis padres con la obtención de beneficios.

Agradezco a mis amigos y compañeros que he formado en esta hermosa etapa universitaria y con los que he compartido muy lindas vivencias, también a aquellos compañeros que alguna vez han pasado por mi vida ya sea en educación básica y educación media.

Agradezco a mi hermano, por ser alguien tan bueno y puro con los que lo rodean, te quiero mucho.

Agradezco a mis profesores, a Lioubov Dombrovskaia y Diego Arroyuelo por ayudarme a desarrollar este documento, entregándome consejos y buenos ánimos. A Alexander Zamyatnin por prestarme soporte con los resultados.

Por último, agradecer a la vida por estar aquí, por tener a esta linda familia que nos rodea.

Dedicatoria

Dedicado a Víctor Guillermo, Jesica Anabela y Víctor Alonso, los bloques componentes de mi alma.....

..... con mucho amor y cariño les dedico este trabajo.

Resumen

Esta memoria tratará sobre el uso de un tipo de estructura de indexación de textos conocido como el **arreglo de sufijos** y su derivado directo, el **arreglo LCP** para obtener de una cadena de proteínas la cantidad total de diferentes substrings de tamaño *k* que variará entre 1 hasta 50, y para cada *k* encontrar cuáles son los residuos que más se repiten. Para ello los archivos a utilizar para generar esta cadena son 4 que corresponden a las bases de datos de proteínas UniProt-SwissProt, UniProt-TrEMBL, EROP-Moscow y Homosapiens (extraído de UniProt-SwissProt). Se utilizarán 2 algoritmos para lograr este propósito, que construirán el arreglo de sufijos y el arreglo LCP de diferentes maneras y que tendrán en común el uso de la estructura conocida como ***priority queue*** para guardar aquellos residuos que más se repitan.

Abstract

This thesis will deal with the use of a type of text indexing structure known as the **suffix array** and its direct derivative, the **LCP array** to obtain from a chain of proteins the total amount of different substrings of size k that will vary between 1 to 50, and for each k find which are the most repeated residues. For this, the files to be used to generate this chain are 4 which are the UniProt-SwissProt, UniProt-TrEMBL, EROP-Moscow and Homosapiens (extracted from UniProt-SwissProt) protein databases. Two algorithms will be used to achieve this purpose, which will build the suffix array and the LCP array in different ways and that will have in common the use of the structure known as ***priority queue*** to store those residues that are most repeated.

Índice de Contenidos

List of Tables

List of Figures

Glosario

- SA: *Suffix Array* - Arreglo de Sufijos.
- LCP: *Longest Common Prefix* - Prefijo Común Más Largo.
- EM: *External Memory* - Memoria Externa.
- RAM: *Random Access Memory* - Memoria de Acceso Aleatorio.
- BWT: *Burrows-Wheeler Transformation* - Transformación de Burrows-Wheels

Introducción

Motivación

Para el actual documento, las razones que motivaron al alumno presente a realizar la investigación de su tesis son la de comenzar a entrar en una rama que en los últimos años ha tomado bastante importancia en la informática, conocido como la *Bioinformática* [1], la cual aplica las tecnologías computacionales contemporáneas a datos biológicos que pueden pertenecer a estructuras como ADN, proteínas, entre otras estructuras biológicas complejas y los cuales están a la mano del ser humano como archivos de cadenas de secuencias (en varios formatos) y que pueden ser usados a voluntad.

En el caso puntual de esta memoria, se trabajarán con proteínas (compuestas de combinaciones de 20 aminoácidos) que están distribuidas en un *dataset* cuyo formato del archivo está en **.fasta**, donde cada cadena de polipéptido está compuesto por un ID o código identificador, su nombre taxonómico y su posterior secuencia de aminoácidos.

Para este tipo de biomolécula, se han realizado numerosos análisis [2, 3] en bases de datos de secuencias con respecto a la cantidad de aminoácidos que se encuentran en total en este tipo de estructuras, distribuyéndolos según su tamaño o para determinar cuál es el aminoácido que más aparece en este tipo de archivos; también según el año de descubrimiento de las proteínas o el tipo de proteína. Todo esto usando fuentes como UniProt y EROP-Moscow, bases de datos de polipéptidos muy importantes que han recabado mucha información sobre este tipo de estructura.

Como una derivación directa, existe una variabilidad casi infinita de combinaciones llamadas residuos (fragmentos) de aminoácidos (*amino acid residues* o AAR de manera simplificada en inglés) que se determinan según su tamaño k y por las posibles opciones a obtener. Por consiguiente buscar y sumergirse en ese universo de posibilidades de encontrar ciertos residuos considerando la cantidad de proteínas que existen en la actualidad es una motivación muy grande, ya que implica un gran desafío el de meterse a buscar "una aguja en un pajar" si se mira desde un punto más coloquial.

Chapter 1

Definición del Problema

1.1. Definición

Las proteínas desempeñan un papel fundamental para la vida y son las biomoléculas más versátiles y diversas, las cuales realizan una enorme cantidad de funciones diferentes, tales como enzimáticas, estructurales, inmunológicas, entre otras. Para muchos biólogos y científicos especializados en este tipo de biomolécula resulta crucial investigar sobre tales propiedades anteriormente mencionadas, por lo tanto para ellos es necesario saber o conocer la composición básica de cada una de las proteínas en base a su elemento básico, conocido como el aminoácido (existen 20 diferentes en total). Todas las proteínas se componen de aminoácidos, entregando así una cantidad inmensa de polipéptidos que existen y que van apareciendo gracias al trabajo de investigaciones y proyectos que hacen los encargados de este asunto.

Estas proteínas aparecen registradas en bases de datos (como UniProt, Genbank o EROP-Moscow) con su secuencia, su nombre taxonómico y un código en clave también conocido como ID, las cuales están disponibles en archivos con formato **.fasta** (que pueden abrise usando un editor de texto básico) de la siguiente forma:

```
>sp|Q98957|3SA1A_NAJAT Cytotoxin 1a OS=Naja atra PE=3 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLPPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSLLVKYVCCNTDRCN
>sp|P79810|3SA1C_NAJAT Cytotoxin 1c OS=Naja atra PE=3 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLIPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSHLVKYVCCNTDRCN
>sp|P85429|3SA1F_NAJAT Cytotoxin 1f (Fragment) OS=Naja atra PE=1 SV=1
LKCNKLVPLFYKTCP
>sp|P60304|3SA1_NAJAT Cytotoxin 1 OS=Naja atra PE=1 SV=1
MKTLLLTLVVVTIVCLDLGYTLKCNKLIPIASKTCPAGKNLCYKMFMMSDLTIPVKRGCI
DVCPKNSLLVKYVCCNTDRCN
```

Figure 1.1: Secuencias de varias proteínas en formato **.fasta**

La cantidad de información de péptidos que actualmente está guardada es mucha y muy diversa, llegando a un

total de 90 millones de proteínas para la base de datos de UniProt. Descargar esta información al completo y procesarla tomaría una gran cantidad de tiempo ya que el archivo superaría los 40 GB de tamaño, esto implicaría un problema importante para realizar diversas tareas de búsqueda. Una de estas tareas consiste es buscar secuencias de aminoácidos (residuos de aminoácidos o *aminoacid residues (AAR)* en inglés) en el conjunto universo de las polipéptidos para poder localizar cuáles son los residuos de ciertos tamaños que más aparecen en las bases de datos. El problema principal radica en buscar cuántas veces aparece cierto residuo de una base de datos que posean tamaños bastante considerables (por ejemplo, buscar una determinada subsecuencia de 2 aminoácidos en una base de datos de 500 GB o superior) sin saber si este residuo está presente o no en aquella proteína, esto podría provocar gastos innecesarios de tiempo a la hora de realizar este tipo de búsqueda, por consiguiente se desea ocupar el menor tiempo posible para realizar esta tarea.

1.2. Objetivos

Lo que se pretende realizar para esta memoria consiste en obtener de un conjunto predeterminado de proteínas lo siguiente:

1. el número máximo de fragmentos de péptidos (AAR) que existen asociado a un valor k determinado, donde k se ubicará en el intervalo entre 1 hasta 50.
2. En base a lo anterior, obtener y determinar para cada k cuáles son los fragmentos de aminóácidos que más se repiten.

Una vez obtenidos estos datos se realizará un análisis algoritmico y biológico (de manera muy superficial) de los resultados logrados. Para realizar esta tarea se usarán 4 *datasets*:

1. La base de datos de proteínas UniProt-SwissProt [41] (555426 proteínas)
2. La base de datos de proteínas UniProt-TrEMBL [42] (88032926 proteínas).
3. La base de datos de péptidos EROP-Moscow [43] (14875 oligopéptidos).
4. Las proteínas humanas ingresadas manualmente a la base de datos UniProt (datos extraídos de UniProt-SwissProt, 86298 proteínas). Para este caso solo se considerará $k = 1$.

Chapter 2

Estado del Arte

2.1. Información previa a considerar

Para entrar de lleno en la tema, es necesario conocer de antemano varios aspectos básicos de la biología.

2.1.1. Biomoléculas

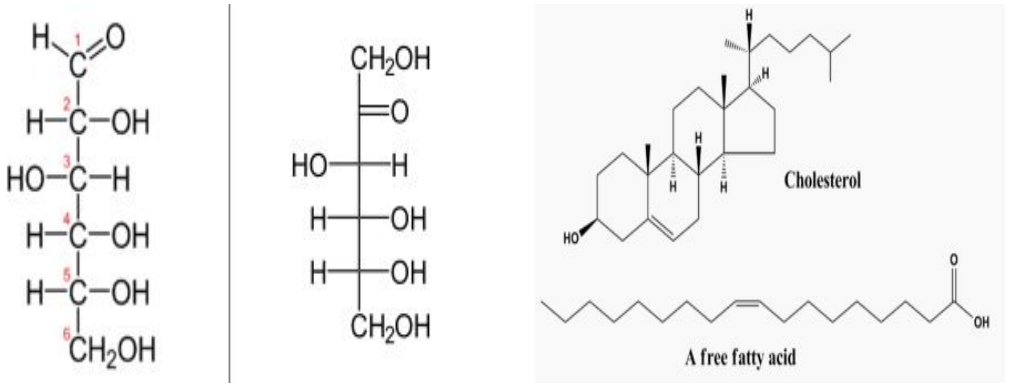
Cada vez que se habla de la biología, este concepto se relaciona directamente con la ciencia que estudia a los seres vivos. Ahora bien, las estructuras o compuestos que constituyen una parte esencial de los seres vivos son conocidas como **biomoléculas**. Estas biomoléculas están principalmente constituidas por elementos químicos como el carbono (C), hidrógeno (H), oxígeno (O), nitrógeno (N), fósforo (P) y azufre (S) [4] y se pueden clasificar en biomoléculas inorgánicas, que se encuentran tanto en seres vivos como en los cuerpos inertes, no obstante son imprescindibles para la vida; y las biomoléculas orgánicas, que son sintetizadas por los seres vivos y tienen una estructura con base en carbono. Estas biomoléculas orgánicas se pueden separar en 4 grandes grupos:

1. Glúcidos (hidratos de carbono o carbohidratos): son la fuente de energía primaria que utilizan los seres vivos para realizar sus funciones vitales. Los ejemplos más conocidos son la glucosa, el almidón y el glucógeno.
2. Lípidos: conforman el principal almacén de energía de los animales y desempeñan funciones reguladores de enzimas y hormonas.
3. Ácidos nucleicos: El ácido desoxirribonucleico y el ácido ribonucleico, mayormente conocidos como ADN (DNA) y ARN (RNA y sus derivados) desarrollan posiblemente la función más importante para la

vida: contener, de manera codificada, las instrucciones necesarias para el desarrollo y funcionamiento de la célula. El ADN tiene la capacidad de replicarse, transmitiendo así dichas instrucciones a las células hijas que heredarán la información.

4. Proteínas: poseen la mayor diversidad de funciones que realizan en los seres vivos; prácticamente todos los procesos biológicos dependen de su presencia y/o actividad. Son proteínas casi todas las enzimas, catalizadores de reacciones metabólicas, hemoglobina, anticuerpos, entre otros. Su unidad base es el *aminoácido*, por el cual se van formando los péptidos según la cantidad de unidades bases enlazadas.

Dentro de estas biomoléculas, el análisis detallado de los carbohidratos y los lípidos depende en demasía de su estructura química (elementos químicos asociados y tipo de enlaces entre ellos), por lo mismo es una materia más ligada a los químicos (ver Figura 1.1).

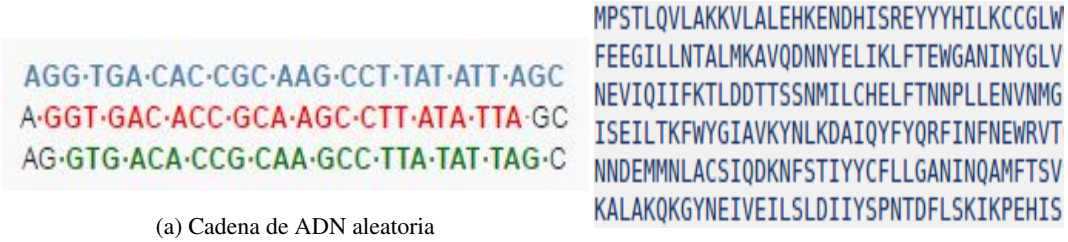


(a) Glúcidos (glucosa y fructosa)

(b) Lípidos (colesterol y un ácido graso)

Figure 2.1: Estructura química de los carbohidratos y lípidos.

Sin embargo, el ADN y los polipéptidos poseen unidades base que pueden ser codificadas como letras, por consiguiente pueden ser secuenciados como *cadenas de strings* y en donde los avances computacionales y la evolución informática toman una importante relevancia (ver Figura 1.2).



(a) Cadena de ADN aleatoria

(b) Cadena aleatoria de una proteína

Figure 2.2: Biomoléculas de ADN y péptidos llevadas a cadenas de strings.

Con respecto a estas 2 últimas estructuras, la diferencia visual más notoria radica en la cantidad de diferentes letras (strings) que las componen, para el ADN son 4 [8] y son denominadas **bases nitrogenadas** que son las siguientes:

1. Adenina
2. Timina
3. Citosina
4. Guanina

Para las proteínas, su elemento básico, como ya se mencionó anteriormente es el **aminoácido**, pero ahora se adentrará en más detalle sobre esta molécula.

2.1.2. Aminoácidos

Los aminoácidos tienen diferentes funciones en el organismo [5] pero ante todo sirven como **las unidades básicas de los péptidos y de las proteínas.** A nivel orgánico el aminoácido es una molécula compuesta con un grupo amino (-NH2) y un grupo carboxilo (-COOH) y que pueden tener distintas distribuciones. Para el caso de los que componen las proteínas se consideran como alfa-aminoácidos:

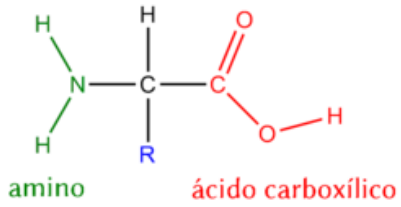


Figure 2.3: Estructura general de un alfa-aminoácido

En la imagen anterior se puede identificar el carbono central (alfa) unido al grupo carboxilo (rojo), grupo amino (verde), un hidrógeno (imagen superior color negro) y el grupo radical (azul) o R. Este grupo radical es el que determina la identidad y las propiedades de cada uno de los diferentes aminoácidos.

El primer aminoácido fue descubierto a principios del siglo XIX, y a partir de ese entonces hasta la actualidad son miles los aminoácidos que han sido descubiertos, pero solo 20 se consideran como los componentes esenciales para las proteínas (y los que se considerarán como parte de esta memoria) que se presentarán a continuación en conjunto con su respectiva abreviación utilizada en las cadenas de proteínas de los archivos FASTA [44]:

Aminoácido - Abreviatura			
Alanina - A	Cisteína - C	Ácido aspártico - D	Ácido glutámico - E
Fenilalalina - F	Glicina - G	Histidina - H	Isoleucina - I
Lisina - K	Leucina - L	Metionina - M	Aspargarina - N
Prolina - P	Glutamina - Q	Arginina - R	Serina - S
Treonina - T	Valina - V	Triptófano - W	Tirosina - Y

Table 2.1: Los 20 aminoácidos existentes y considerados

Existen otras abreviaturas en las cadenas como B, X o J, pero para el alcance de esta memoria no serán considerados como objeto de estudio y análisis posterior.

A partir de este pequeño elemento se forman las macromoléculas que se identifican según la cantidad de aminoácidos (a partir de ahora se mencionarán como aa.) que lo compongan:

Tamaño	Tipo de estructura
2 aa.	Dipéptido
3 aa.	Tripéptido
Entre 2 y 8 aa.	Oligopéptido
Menos de 100 aa.	Péptido
Mayor o igual de 100 aa.	Proteína o polipéptido

Table 2.2: Identificación de macromoléculas según cantidad de aminoácidos

2.2. Secuencias de proteínas

Desde el momento en que se descubrieron los elementos componentes del ADN y las proteínas, se han investigado sobre las posibles combinaciones que se pueden encontrar entre las bases que los conforman y en que cantidad se encuentran. Para el ADN y sus 4 elementos básicos existen millones de seres vivos, parásitos, virus, protozoos y entre otros que se definen por su código genético, por lo cual encontrar los diversos residuos de bases según un determinado tamaño. En el caso puntual para la finalidad de este escrito y según lo mencionado por [6], es posible estudiar de manera teórica y con fórmulas matemáticas la cantidad máxima de fragmentos que puede formar una proteína. Considerando como base que el número posible de estructuras peptídicas naturales P están compuestas de diferentes residuos de aminoácidos (incluyendo repeticiones en cadenas de aminoácidos) sigue la siguiente fórmula:

$$P = A^n \tag{2.1}$$

Donde A es el número de diferentes aminoácidos existentes, y n es la cantidad de aminácidos correspondientes a la estructura estudiada. Por lo mismo y siguiendo esta fórmula (considerando $A = 20$) la cantidad de diferentes combinaciones péptidos de tamaño k que se pueden obtener se aprecian en la siguiente tabla:

Tamaño péptido (k)	Combinaciones posibles (A^k)
2 aa.	400
3 aa.	8000
4 aa.	160000
5 aa.	3200000
10 aa.	1.024×10^{13}
20 aa.	1.049×10^{26}
50 aa.	1.126×10^{65}

Table 2.3: Combinaciones posibles a obtener según el tamaño del péptido

Según lo observado en esta tabla se identificar que a medida que el valor de k va en aumento, las posibles combinaciones que se pueden obtener de fragmentos de proteínas pueden llegar a tener valores inimaginables para el ser humano corriente; no obstante, no todas estas estructuras existen o son capaces de ser encontradas en la naturaleza [3], aun así la diversidad de la búsqueda de estos residuos sigue siendo gigantesca, y por ende difícil de solucionar, y este será uno de los problemas que se intentará solucionar en esta memoria.

Ahora bien, para un polipéptido de tamaño n aminoácidos, el máximo número posible de fragmentos de tamaño k, que teóricamente se podrían obtener (considerando las posibles repeticiones de fragmentos), es descrita mediante la siguiente expresión:

$$N_k^{teorica} = n - k + 1 \tag{2.2}$$

Por consecuencia, el máximo número posible de fragmentos (con repeticiones) que teóricamente se pueden obtener para una molécula de tamaño n, partiendo desde $k = 2$ (dipéptidos) hasta $k = n - 1$, viene dado por:

$$N_{max}^{teorica} = \sum_{2}^{n-1} \frac{k(k-1)}{2} - 1 \tag{2.3}$$

Mediante estas fórmulas, se han calculado la cantidad de posibles fragmentos que se pueden obtener en diferentes oligopéptidos y proteínas:

Número	Oligopéptido/Proteína	n	$N_{suma}^{teorica}$
1	Encefalina (varios tipos biológicos)	5	9
2	Bradiquinina (mamíferos)	9	35
3	ACTH (humanos)	39	740
4	Cadena α hemoglobina (humanos)	141	9869
5	Cadena β hemoglobina (humanos)	146	10584

Table 2.4: Número máximo posible de fragmentos que se pueden formar en 5 proteínas

Considerando que para un polipéptido de largo n aminoácidos, si este valor de n es muy alto, se puede obtener una cantidad muy alta de fragmentos de dipéptidos, pero muchos de estos dipéptidos se pueden repetir varias veces en la cadena, por lo tanto, cuando se desea obtener **el máximo número de fragmentos diferentes** asociado a un valor k determinado, este puede tener un valor muy bajo en comparación con la cantidad total de fragmentos obtenidos. Por medio de las fórmulas descritas anteriormente, se puede obtener el número máximo de diferentes fragmentos (o fragmentos esperables) asociado a un tamaño k:

$$N_k^{diff} = N_k^{teorica} - R_k \tag{2.4}$$

Este valor R_k, se obtiene introduciendo nuevos parámetros i (que es el número de estructuras idénticas para determinado k) y m (el número de diferentes estructuras para el determinado k):

$$R_k = \sum_{1}^{m}(i-1) \tag{2.5}$$

Por lo tanto, el número máximo de fragmentos diferentes que se pueden obtener en una proteína sigue la siguiente fórmula:

$$N_{max}^{diff} = \left[\sum_{2}^{n-1}\frac{k(k-1)}{2} - 1\right] - \sum_{2}^{n-1}\left[\sum_{1}^{m}(i-1)\right] \tag{2.6}$$

Tomando la información de la base de datos de oligopéptidos EROP-Moscow, para mostrar los valores obtenidos con estas fórmulas, se usará como ejemplo la caseína bovina (proteína proveniente de la vaca). Esta proteína se compone de 4 subunidades, $\alpha - s1$, $\alpha - s2$, β y κ. La siguiente tabla muestra las cantidades teóricas y diferentes de fragmentos obtenidos como dipéptidos y sus sumas totales:

Número	Caseína bovina (subunidad)	n	$N_2^{teorica}$	N_2^{diff}	$N_{max}^{teorica}$	N_{max}^{diff}
1	$\alpha - s1$	199	198	134	19700	19621
2	$\alpha - s2$	207	206	131	21320	21216
3	β	209	208	124	21735	21641
4	κ	169	168	118	14195	14138
5	$\alpha - s1 + \alpha - s2 + \beta + \kappa$	784	780	260	76950	76304

Table 2.5: Número máximo posible de fragmentos que se pueden obtener en una proteína de caseína bovina.

Se puede identificar que para los fragmentos de dipéptidos, la cantidad de diferentes fragmentos es bastante menor que la cantidad total de fragmentos obtenidos para las 4 subunidades, pero aún así la cantidad de fragmentos diferentes totales obtenidos es prácticamente la misma que la cantidad de fragmentos totales sin diferenciar. Esto es notorio ya que si k va en progresivo aumento, el universo combinatorio de posibles fragmentos formados se acorta drásticamente, lo que también favorece a la baja formación de fragmentos que se repiten.

2.3. Base de datos UniProt

UniProt (nombre que proviene de ***Uni****versal* ***Prot****ein*) es una base de datos de secuencias de proteínas e información funcional respectiva, accesible de manera gratuita a todo público. Su fuente de investigación la compone un consorcio cuyos participantes son el Instituto Europeo de Bioinformática (EBI - *European Bioinformatics Institute*), el Instituto Suizo de Bioinformática (SIB - *Swiss Institute of Bioinformatics*) y los Recursos de Información de Proteínas (PIR - *Protein Information Resource*) quienes compusieron UniProt en Diciembre del 2003.

Cada uno de estos consorcios está altamente envuelto en las anotaciones y mantenimiento de las bases de datos que corresponden a UniProt, gracias a eso nace **UniProtKB** (UniProt Knowledgebase), que es una base de datos de proteínas comisariada por expertos, que se compone de 2 secciones:

1. **UniProt - SwissProt**: es una base de datos de secuencias de proteínas manualmente anotadas y revisadas. Combina información extraída de la literatura científica y un posterior análisis computacional. Actualmente está compuesto de 555.594 secuencias de proteínas que equivalen a un texto con un peso de 298 MB.

2. **UniProt - TrEMBL**: es una base de datos de secuencias de proteínas automáticamente anotadas y no revisadas. Estas secuencias son registros de alta calidad y computacionalmente analizados. Esta base de datos fue introducida en respuesta para aumentar el flujo de datos que se obtienen de los proyectos relacionados al genoma. Actualmente está compuesto 90.050.711 secuencias de proteínas que equivalen a un texto con un peso de 40 GB.

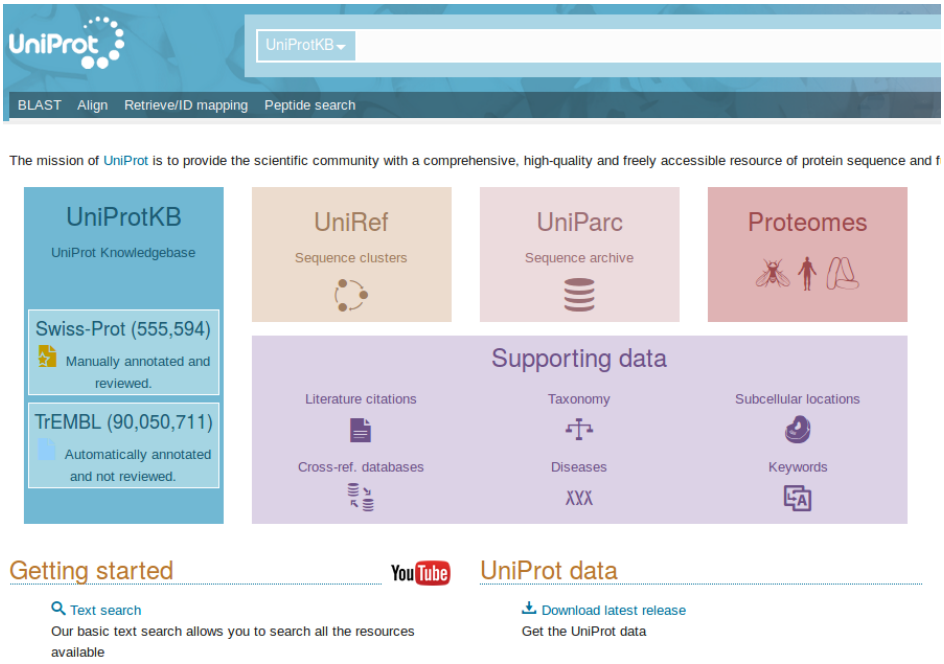


Figure 2.4: Página principal del sitio web de UniProt.

2.4. Base de datos EROP-Moscow

EROP-Moscow es una base de datos de secuencias de oligopéptidos de tamaño que rondan entre 2 a 50 aminoácidos altamente detalladas. Fue creada por Alexander Zamyatnin el año 2003 y de manera constante se le agregan nuevas secuencias investigadas [43].

Figure 2.5: Página principal del sitio web de EROP-Moscow.

Actualmente está compuesta por 14.785 secuencias que equivalen a un texto con peso de 1.4 MB. Aunque son bastante pocas secuencias en comparación con UniProt, estos péptidos están descritos de manera muy completa

y son ideales para realizar diferentes tipos de análisis, ya sea por sus propiedades regulatorias en los seres vivos, propiedades físico-químicas y el tipo de órgano en el cual se encuentran, entre otros.

2.5. Técnicas utilizadas en el problema

En [2] se menciona que buscar secuencias de proteínas en un predeterminado archivo (puede ser de texto o .fasta) es una tarea muy compleja, ya que se formaría un escenario similar al buscar una *aguja en un pajar* y recorrer millones de secuencias en cada búsqueda no sería lo más convieniente considerando que la cantidad de proteínas que existen el día de hoy son muchas, por consiguiente una herramienta recomendable sería pre-procesar la base de datos de proteínas con alguna técnica conocida o implementada. A continuación se hablará de forma general de algunos algoritmos conocidos que han tratado este problema.

2.5.1. Algoritmo de fuerza bruta

Este es el algoritmo más simple posible [8], ya que dado un texto de tamaño n se revisan todas las posiciones posibles de un patrón de tamaño $k < n$ desde el comienzo hasta el final del texto (izquierda a derecha). Para el caso puntual de los archivos .fasta de las proteínas es necesario extraer únicamente las cadenas de secuencias respectivas y adjuntarlas línea a línea (de esa forma es más fácil trabajarlas). Luego, y siguiendo la presunción matemática del número máximo de fragmentos de tamaño k cada una de las cadenas se revisa desde la posición 0 hasta la posición $n-1$ (el valor de n varía según el largo de la cadena de cada proteína) yendo de caracter a caracter un total de $n-k+1$ veces para cada secuencia. Lo siguiente muestra una implementación simple en lenguaje C++ acomodada para obtener los diferentes substrings de largo k en un texto de proteínas:

```
1  set <string> unicos;
2  ifstream proteinas("proteinas.txt"); //nombre generico
3  string secuencia;
4  for (int j = 0; j < cantidad_ss; j++)
5  {
6      getline(proteinas, secuencia);
7      int n = secuencia.size();
8      int maximo = n-k+1; // k entre 1 a 50
9      for (int i = 0; i < maximo; i++)
10     {
11         string pruebas = secuencia.substr(i,k);
12         if (pruebas.find_first_of("BOUXZ")==std::string::npos)
13         {
14             unicos.insert(pruebas);
```

```
15              }
16          }
17  }
18  cout << unicos.size() << endl;
```

Pseudocódigo 2.1: Búsqueda utilizando fuerza bruta en C++

Haciendo un resumen de este código lo que hace es agregar TODOS los substrings totales de tamaño *k* que encuentra en una proteína moviéndose letra por letra en un total de *n-k+1* ocasiones (el getline toma la cadena y la pasa al parámetro *string*, lo hace hasta tomar todas las cadenas de proteínas en el texto), y se verifica si el substring no posee algún aminoácido que no se encuentre dentro de los 20 aa. mencionados anteriormente, en caso afirmativo el substring se ingresa al set (cuya cualidad es no repetir elementos en su conjunto). Una vez realizado todo esto se puede obtener el número de diferentes substrings de tamaño *k* que se encuentran en el texto, que es la gran ventaja que posee esta implementación, por lo que puede servir como un "verificador de resultados" cuando se analicen los resultados de implementaciones más óptimas.

Como desventajas importantes resalta el hecho de que el tiempo de la implementación es muy lento ya que al saltar letra por letra y realizando todas las comparaciones posibles de los substrings se pierde bastante tiempo (la complejidad es de $O(n-k+1)$ por cadena, si se tiene que N es la cantidad total de proteínas la complejidad en el texto sería de $O(N(n-k+1))$), que también es negativamente afectado por el tamaño del texto a trabajar (usar un texto de 40 GB como el "uniprot_trembl.fasta" demoraría casi 20 horas en obtener los diferentes substrings para un $k = 15$). Además para guardar un `set<string>` en C++ requiere de una gran capacidad de memoria RAM, la cual va peligrosamente en aumento si *k* sube su valor. Por otra parte esta implementación tampoco entrega la factibilidad de encontrar una manera de extraer aquellos substrings de tamaño *k* que más se repiten. Por consiguiente esta implementación básica no ayuda a realizar por completo la tarea esperada para este trabajo.

2.5.2. Algoritmos de búsqueda de strings

Esta clase de algoritmos (en inglés conocidos como *string searching algorithm*) tratan de localizar si uno o varios strings solicitados (que también se llaman patrones) aparecen en un string más largo o simplemente un texto como tal, considerando el alfabeto que por el cual está compuesto el string de destino o texto. En la mayoría de las ocasiones este alfabeto (Σ) es el que determina el rendimiento de determinado algoritmo (una variación de este alfabeto puede ayudar o perjudicar la eficiencia de un algoritmo en particular), como también el texto a analizar. Una clasificación básica de estos algoritmos se puede realizar según la cantidad de strings a encontrar:

1. Algoritmos de búsqueda de un único patrón (*Single pattern algorithms*)
2. Algoritmos de búsqueda de múltiples patrones (*Multiple pattern algorithms*)

Algoritmos de búsqueda de un único patrón

Como lo dice el mismo título y hablando de manera más formal, esta clase de algoritmo de búsqueda consiste en encontrar las ocurrencias de un determinado patrón [9] $p = p_1 p_2 \ldots p_m$ en un texto largo $T = T_1 T_2 \ldots T_n$, donde p y T son secuencias de caracteres que provienen de set finito de caracteres Σ. Para este caso se describirán de manera sencilla los algoritmos más reconocidos que han sido desarrollados para este problema, que son el algoritmo de **Knuth-Morris-Pratt** y el algoritmo de **Boyer-Moore**.

a) **Algoritmo de Knuth-Morris-Pratt**

Este algoritmo desarrollado el año 1977 por Donald E. Knuth, James H. Morris, y Vaughan R. Pratt busca como objetivo minimizar la cantidad de comparaciones del patrón p con el texto T manteniendo una pista de información obtenida en informaciones previas, valiéndose de la ayuda de una función de fallo (preproceso del patrón) que indica cuando la última comparación se puede reusar si existe un fallo (revisar [10] para mayores detalles). Las comparaciones de caracteres se realizan de izquierda a derecha buscando el prefijo más largo posible y usarlo como información importante en las iteraciones siguientes (pseudocódigo en sección Apéndice, algoritmo 1). Se puede tomar como ejemplo el texto "aaaabaabaaab" y el patrón "aabaaa".

T:	a	a	a	a	b	a	a	b	a	a	a	b
P1:	a	a	b									
P2:		a	a	b								
P3:			a	a	b	a	a	a				
P4:						a	a	b	a	a	b	

Table 2.6: Ejemplo de uso del algoritmo de Knuth-Morris-Pratt

Apreciando la imagen se puede identificar que el algoritmo KMP realiza un total de 14 comparaciones hasta encontrar que el patrón aparece en el texto, en el caso de la fuerza bruta hubiesen sido necesarias 21 comparaciones hasta identificar que el patrón está en el texto. El tiempo de preprocesamiento del texto va del orden $O(m)$ donde m es el largo del patrón, y el tiempo que demora el patrón en ser ubicado en el texto es aproximadamente del orden $O(n)$ donde n es el largo del texto.

b) **Algoritmo de Boyer-Moore**

Este algoritmo desarrollado por Bob Boyer y J. Strother Moore el año 1977 se desmarca del algoritmo KMP ya que para Boyer-Moore se realiza la comparación entre patrón y texto de derecha a izquierda, por ejemplo si hubiera una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar m posiciones sin realizar ninguna comparación extra. En particular, no es necesario comparar los primeros $m - 1$ caracteres del texto, lo cual indica que podría realizarse una búsqueda

en el texto con menos de n comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios [11]. Esto le entrega una gran ventaja de tiempo y espacio en comparación al algoritmo KMP, en especial cuando el patrón analizado es grande (compuesto por más de 10 caracteres). Para ello se vale de la ayuda de un preprocesamiento del patrón para obtener la posición de ocurrencia de cada uno de los diferentes caracteres involucrados y un localizador de sufijos (pseudocódigo en sección Apéndice, algoritmo 2). Ahora bien, volviendo a realizar la comparación tomando el texto "aaaabaabaaab" y el patrón "aabaaa":

T:	a	a	a	a	b	a	a	b	a	a	a	b
P1:	a	a	b	a	a	a						
P2:			a	a	b	a	a	a				
P3:						a	a	b	a	a	a	

Table 2.7: Ejemplo de uso del algoritmo de Boyer-Moore

Para este ejemplo se puede apreciar que con Boyer-Moore se realizan 9 comparaciones totales hasta finalmente encontrar que el patrón está ubicado en el texto, mejorando el resultado de Knuth-Morris-Pratt. Este algoritmo tiene un tiempo de preprocesamiento del orden de $O(m + k)$ donde m es el largo del patrón y k es cantidad de elementos que componen el alfabeto utilizado; el tiempo que demora en encontrar el patrón en el texto varía entre el orden $O(n/m)$ para el mejor caso y el orden $O(nm)$ para el peor caso, considerando n como el largo del texto en cuestión.

Mirando de una visión macro, pareciera ser que estos algoritmos tuvieran un mejor comportamiento a nivel de tiempo y rendimiento que el algoritmo de fuerza bruta, ¿pero son realmente convenientes para aplicarlos a archivos grandes de proteínas en formatos .fasta? Por una parte, estos algoritmos requieren de un **patrón**, que en este caso serían los posibles residuos de aminoácidos que existen para un tamaño k, por consiguiente, hay que ponerse en el caso de que se buscan los diferentes residuos de proteínas de largo 6, que equivalen a comparar las 64000000 combinaciones posibles y buscarlas en el archivo, que si es grande (por ejemplo superior a los 100 MB) demoraría mucho tiempo porque en ciertos casos podría ocurrir de que se revise para un solo residuo todo el archivo y no encontrarlo en el texto, en consecuencia revisar a cada rato el archivo desde el principio es un objetivo que se desea evitar para este caso del trabajo.

Casi 40 años han pasado desde que estos 2 algoritmos aparecieron para trabajarlos, sin embargo y con el paso del tiempo han aparecido nuevas mejoras de estas implementaciones (como Horsepool) o simplemente nuevos algoritmos que utilizan acercamientos en base a factores intermedios (como el algoritmo BOM - *Backward Oracle Matching*), aunque estas nuevas implementaciones también caen en lo mismo, volver a utilizar como patrones a todos los potenciales residuos de aminoácidos de largo k, y usando valores de k muy largos puede ser una tarea infinita de realizar.

Algoritmos de búsqueda de múltiples patrones

Esta clase de algoritmos es una extensión del punto anterior, ya que aquí se buscan de manera simultánea si un conjunto de patrones $P = \{p_1, p_2, \ldots, p_3\}$ aparece en un texto T dado un set de caracteres Σ [9]. Las implementaciones más reconocidas para este caso son el algoritmo de **Aho-Corsack** (una extensión de algoritmo de Knuth-Morris-Pratt), el algoritmo de **Commentz-Walter** (una extensión del algoritmo de Boyer-Moore) y el algoritmo de **Set-BOM** (una extensión del algoritmo *Backward Oracle Matching*).

Como es lógico, aumentar la cantidad de palabras (patrones) a comparar en el texto determinado toma un proceso más caro a nivel de espacio, pero considerando que se tuviera un gran rango de patrones a buscar, como el caso de todos los potenciales péptidos de tamaño k a encontrar, tomaría una visión mucho más optimista que usando un algoritmo de búsqueda con un solo patrón incluyendo una optimización en los tiempos totales de búsqueda. No obstante, se repetiría el mismo fenómeno con los archivos de gran tamaño, requiriendo de un patrón para localizar y que puede no estar en el texto, lo que supondría un gasto de tiempo innecesario e infructuoso para este trabajo, si esto se traduce en tener muchos substrings para analizarlos como patrones en este caso, por consiguiente utilizar estos algoritmos no es conveniente.

2.5.3. Estructuras de datos para texto

Los casos vistos anteriormente tienen como principal problema que para cada patrón dado, es necesario **revisar el texto desde el inicio hasta llegar a la ubicación donde el patrón existe o simplemente recorrer el texto completo sin ubicar al patrón**, ejemplificando, sería como tener que llevar 20 ovejas de La Serena a Santiago y se va trasladando de una a una generando una pérdida inmensa de tiempo y costo, por ende tiene mucho más sentido agrupar estas ovejas y luego llevarlas todas juntas a su destino. Por lo tanto lo que se busca en definitiva es **organizar todos los patrones en una única estructura de datos**, y para lograr esta tarea, no será necesario tener a mano a los patrones a buscar, sino que al **texto en sí** el cual puede ser agrupado como una larga "cadena de strings".

Llevando a un ejemplo más concreto y sencillo, se tiene la palabra MISSISSIPPI$ que está compuesta por 12 letras o caracteres. Cada caracter se podría representar por medio de un índice (indexar la palabra) y quedaría de la siguiente forma:

0	1	2	3	4	5	6	7	8	9	10	11
M	I	S	S	I	S	S	I	P	P	I	$

El concepto **sufijo** corresponde para este contexto al sector de la palabra desde un punto medio (que puede ser incluso el comienzo de la palabra) hasta el final (el prefijo recurre un sector desde el inicio hasta el punto intermedio), lo cual será útil para el siguiente punto. A partir de la imagen superior es posible obtener los "sufijos" de esta palabra, para posteriormente agrupar todos estos "sufijos" e ir ordenándolos alfabéticamente

(observar Cuadro 2.8 en la siguiente página).

0	MISSISSIPPI$	11	$
1	ISSISSIPPI$	10	I$
2	SSISSIPPI$	7	IPPI$
3	SISSIPPI$	4	ISSIPPI$
4	ISSIPPI$	1	ISSISSIPPI$
5	SSIPPI$	0	MISSISSIPPI$
6	SIPPI$	9	PI$
7	IPPI$	8	PPI$
8	PPI$	6	SIPPI$
9	PI$	3	SISSIPPI$
10	I$	5	SSIPPI$
11	$	2	SSISSIPPI$

Table 2.8: Sufijos (sector izquierdo) e índices (sector derecho) ordenados según orden alfabético.

La clave de este ordenamiento está en el **nuevo orden que tienen los índices**, los cuales tendrán una vital importancia a la hora del desarrollo de esta memoria, y que pueden ser implementados a partir de la palabra o texto entregado para luego guardarlos y registrarlos en un vector.

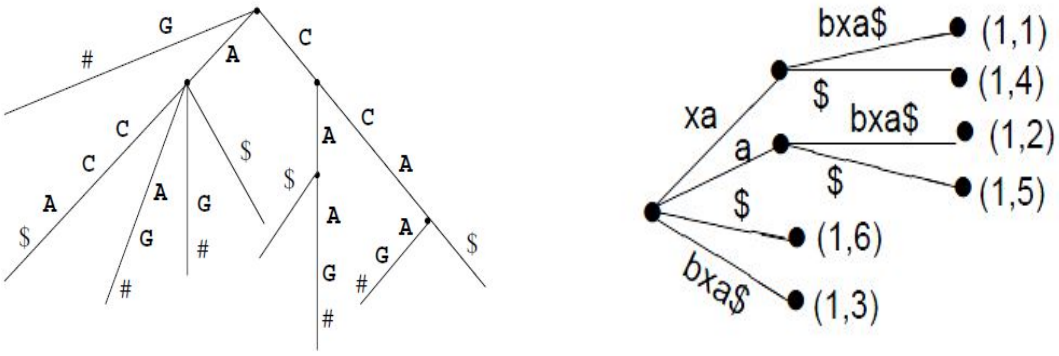
Los algoritmos encargados de trabajar con este tipo de indexación son conocidos como **Estructuras de Datos para Textos** (en inglés se conocen como *Indexed Text Searching*) y de los cuales se describirán los más conocidos.

2.5.4. Árbol de sufijos

Para la lectura de las secuencias de proteínas y ADN, actualmente se considera el uso de los árboles de sufijos [12] o también conocido como el *suffix tree*, el cual es muy usado para leer cadenas de strings, ya que es ideal para desarrollar diversos problemas relativos al uso de strings como hallar substrings en común, comparación de estadísticas, proyectos relacionados con el genoma, entre otros.
Formalmente hablando, el árbol de sufijos [13] es una estructura de datos comprimida que sirve para almacenar una cadena de caracteres con "información pre-procesada" sobre su estructura interna. Según lo definió Weiner en 1973 [14] el árbol de sufijos τ para un string S de largo $m = s_1 s_2 ... s_m$ posee un nodo de inicio, y nodos hoja o hijos (que serán al menos 2). Comienza leyendo s_1, luego lee s_2 (que es s_1 quitándole la primera letra respectiva desde izquierda a derecha) identificando si hay strings que ya formaron una descendencia similar para continuar por la hoja y no crear otra y en caso contrario crear una nueva hoja, hasta así leer todo el string. En ciertas oportunidades, un sufijo de S podría coincidir con el prefijo de algún otro sufijo de S, por lo tanto el camino para el primer sufijo no terminaría en una hoja. Por ello que la solución creada fue añadir un carácter

terminador, siendo usado comúnmente el $ para este caso.

Esto es muy útil, por ejemplo, para resolver el problema de la subcadena en tiempo lineal ya que se tiene este texto S de longitud m el cual se pre-procesa (se construye el árbol) en tiempo de orden $O(m)$ donde se busca una subcadena P de longitud n (cuyo tiempo es de $O(n)$).



(a) Árbol de sufijos de una cadena de ADN aleatoria

(b) Árbol de sufijos de la palabra *xabxa*

Figure 2.6: Ejemplos de árboles de sufijos.

El problema para la secuencia de ADN considera que habrá un conjunto de textos S_i pertenecientes a la cadena de nucleóticos, y en el cual poder verificar si P es subcadena de algún S_i, lo que es llamado *el árbol de sufijos generalizado* (ver Figura 2.7).

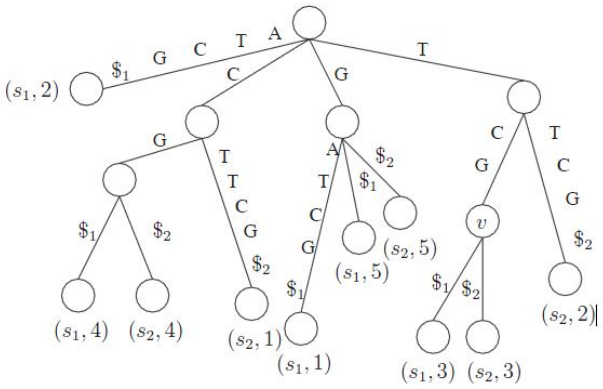


Figure 2.7: El árbol de sufijo generalizado para las secuencias *GATCG$* y *CTTCG$* [17]

2.5.5. Arreglo de sufijos

Los arreglos de sufijos fueron introducidos por Udi Mander y Gene Myers el año 1990 [15] y corresponden a una variante del árbol de sufijos que resulta mucho más eficiente en cuanto al uso de la memoria [16].

Hablando de manera más formal, sea $T = t_1, t_2, \ldots, t_n$ una cadena y sea $T[i, j]$ la subcadena que va del índice i hasta j. El arreglo de sufijos SA de la cadena T va a ser un arreglo de enteros brindando las posiciones iniciales de los sufijos en T en órden lexicográfico. Esto significa que $SA[i]$ contiene la posicion inicial del i-ésimo sufijo más pequeño en T y por tanto se cumple que para todo $1 < i \leq n : T[SA[i-1], n] < T[SA[i], n]$.

Reutilizando el ejemplo de la palabra MISSISSIPPI\$, el arreglo de sufijos que contiene las posiciones iniciales sería el siguiente:

i	0	1	2	3	4	5	6	7	8	9	10	11
$SA[i]$	11	10	7	4	1	0	9	8	6	3	5	2

Table 2.9: Arreglo de sufijos $A[i]$ de la palabra MISSISSIPPI\$

Para este tipo de estructura, y tal como menciona [18], encontrar un patrón P de tamaño m (donde $m < n$) en el texto consiste en buscar el intervalo $[j, k]$ en el arreglo que contiene a todos los sufijos que comienzan con el patrón mencionado. Esto se logra mediante una búsqueda binaria sobre A, buscando lexicográficamente el menor sufijo que parte con P, y otra búsqueda binaria buscando el mayor sufijo que parte con P. En cada comparación de esta búsqueda binaria se compara un sufijo con el patrón, lo que toma tiempo $O(m)$ en el peor caso, luego la búsqueda completa toma $O(m \log n)$. Junto con el arreglo de sufijos es necesario guardar el texto, y ambos juntos ocupan $O(n \log n)$ bits de espacio y un tiempo del orden de $O(n(\log n)^2)$ [20] (pseudocódigo en sección Apéndice, algoritmo 3).

Diferencias entre árbol de sufijos y arreglo de sufijos

Ambas estructuras representan (entregan como salida) el mismo resultado, pero con varias discrepancias. La más obvia, es que el *suffix tree* es un árbol donde al recorrer cada una de sus hojas se obtiene el arreglo de sufijos. No obstante, la diferencia importante entre ambos algoritmos radica en **la cantidad de espacio utilizada**. La construcción del árbol de sufijos toma tiempo lineal en el largo del texto. El árbol de sufijos tiene $O(n)$ nodos. Luego, dado que los substrings de cada rama pueden ser guardados como la posición y el largo de un substring del texto S, un árbol de sufijos de n nodos ocupa $O(n \log n)$ bits. Esto permite realizar varias operaciones sobre los nodos del *suffix tree*. Sin embargo esto se convierte en un problema porque implica un tamaño de almacenamiento de varias veces el texto original [18].

Por el contrario el arreglo de sufijos pierde varias de estas funciones a cambio de una importante mejora en cuanto a los requerimientos en espacio de los árboles de sufijos porque el *suffix array* guarda n **enteros**. Por lo cual si se tiene un arreglo donde un entero requiere 4 *bytes* (32 bits), un arreglo de sufijos en ese caso requeriría un total de implementación de $4n$ *bytes* (si el entero necesitara 8 *bytes* (64 bits), el arreglo tendría un tamaño de $8n$ *bytes*). Y esto es significativamente menor que los $20n$ bytes requeridos en la implementación del árbol de sufijos [19].

2.5.6. Arreglo LCP

LCP es una sigla en inglés que hace referencia al "Prefijo común más largo" (LCP = *Longest Common Prefix*). Este arreglo es una estructura auxiliar al arreglo de sufijos (tamaño igual al del *suffix array*), pero que en sus posiciones guarda las longitudes de los prefijos comunes más largos entre todos los pares de sufijos consecutivos correspondientes al arreglo de sufijos.

Esta estructura fue introducida el año 1990 por Udi Manber y Gene Myers con la finalidad de optimizar el tiempo de ejecución para la búsqueda de strings [15]. Y de manera formal se define como: Se tiene SA que es el arreglo de sufijos del texto $T = t_0 t_1 \ldots t_{n-1}$ y $lcp(v, w)$ es el largo del prefijo común más largo entre 2 strings v y w. Además se sabe que $T[i, j]$ es el substring de S que va desde la posición i hasta j. Por consiguiente el arreglo $LCP[0, n-1]$ es un arreglo compuesto de números enteros de tamaño n donde $LCP[0]$ está indefinido y $LCP[i] = lcp(T[SA[i-1], n], T[SA[i], n])$ para todo $1 < i \leq n$. Por consiguiente $LCP[i]$ almacena la longitud del prefijo común más largo del -lexicográficamente hablando- i-ésimo sufijo más pequeño y su predecesor en el *suffix array*.

Para mostrar un ejemplo, se usará nuevamente la palabra MISSISSIPPI$ en base al SA obtenido anteriormente, mostrando también sus sufijos ordenados (ver Cuadro 2.10). A partir de esto el arreglo LCP es construido comparando lexicográficamente a los sufijos consecutivos para determinar el prefijo común más largo (ver Cuadro 2.11).

Por ejemplo, $LCP[4] = 4$ que es la longitud del prefijo común más largo ISSI que se comparten entre los sufijos $SA[3] = T[4, 11]$ = ISSIPPI$ y $SA[4] = T[1, 11]$ = ISSISSIPPI$. Notar que $LCP[0] = \emptyset$ ya que no hay un sufijo más pequeño lexicográficamente hablando [21].

La construcción del algoritmo del arreglo LCP se puede establecer en 2 categorías, obtener el arreglo LCP como un bi-producto del arreglo de sufijos [15] o simplemente utilizar un *suffix array* previamente construído [22]. Para este segundo caso el arreglo LCP es construido en un tiempo del orden de $O(n \log n)$, y si se asume que cada simbolo de texto pesa un byte y cada valor o elemento del arreglo de sufijos y arreglo LCP pesa 4 bytes, considerando un texto de tamaño n entonces el peso total para guardar los 2 arreglos sería de $8n$ (pseudocódigo en sección Apéndice, algoritmo 4).

2.5.7. External Memory Suffix Array

Como se vio anteriormente, la construcción del *suffix array* (SACA, sigla en inglés de *Suffix Array Construction Algorithm*) se realiza en memoria RAM, donde se almacena el texto de entrada y el posterior arreglo de sufijos. Sin embargo existen textos muy grandes como la base de datos de Wikipedia, bases de datos de genomas, variados textos que superen los 10 GB de tamaño, los cuales son tan grandes que no se podría guardar en la memoria RAM siquiera el texto en sí. Por lo mismo se ha investigado una alternativa para poder construir

i	0	1	2	3	4	5	6	7	8	9	10	11
T[i]	m	i	s	s	i	s	s	i	p	p	i	$
SA[i]	11	10	7	4	1	0	9	8	6	3	5	2
0	$	i	i	i	i	m	p	p	s	s	s	s
1		$	p	s	s	i	i	p	i	i	s	s
2			p	s	s	s	$	i	p	s	i	i
3			i	i	i	s		$	p	s	p	s
4			$	p	s	i			i	i	p	s
5				p	s	s			$	p	i	i
6				i	i	s				p	$	p
7				$	p	i				i		p
8					p	p				$		i
9					i	p						$
10					$	i						
11						$						

Table 2.10: SA de MISSISSIPPI$ y sus sufijos respectivos

i	0	1	2	3	4	5	6	7	8	9	10	11
T[i]	m	i	s	s	i	s	s	i	p	p	i	$
SA[i]	11	10	7	4	1	0	9	8	6	3	5	2
LCP[i]	∅	0	1	1	4	0	0	1	0	2	1	3

Table 2.11: Arreglo LCP de la palabra MISSISSIPPI$

arreglos de sufijos para aquellos textos de gran tamaño.

Ante eso surgió el concepto del "arreglo de sufijos en memoria externa" (traducción en español de *External Memory Suffix Array*, también es conocido como *EM Suffix Array* o su abreviatura, *EMSA*), que consiste en construir el arreglo de sufijos de determinado texto T evitando guardar todo el texto en la memoria RAM. La idea fue introducida por Gastón Gonnet, Ricardo Baeza-Yates y Tim Snider el año 1992 [31] y a partir de ese entonces han aparecido varias implementaciones que se diferencian en cuanto a rendimiento y tiempo ([28], [29], [30]).

Siguiendo lo enunciado por [30], la idea principal del "arreglo de sufijos en memoria externa" es particionar el texto en bloques que son lo suficientemente pequeños para que el arreglo de sufijos de cada bloque pueda ser construido en la memoria RAM. Entonces una vez que se tengan los arreglos de sufijos en bloques, estos se combinan formando el arreglo de sufijos completo del texto siguiendo el siguiente formato:

Se aprecia en la imagen que los bloques de *suffix array* se van combinando desde el final y de manera singular,

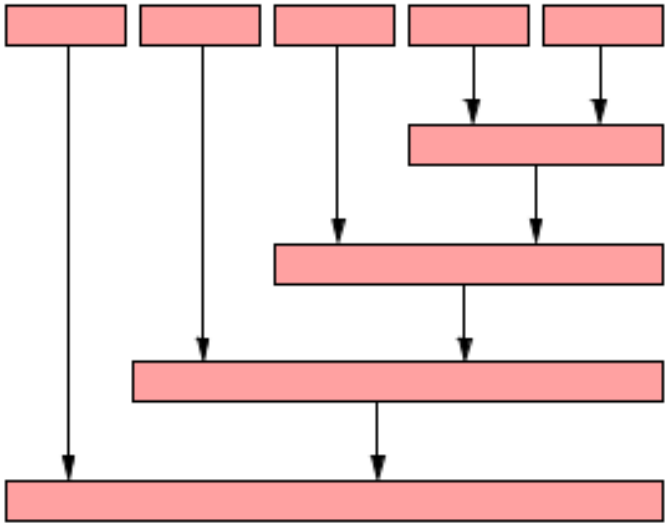


Figure 2.8: Arreglo de sufijos de un texto T formado por la combinación de arreglos de sufijos de los bloques del texto T.

donde cada uno de ellos se va acoplando con el bloque combinado que dará como resultado al arreglo de sufijos completo para el texto T.

Para explicar cómo se realiza la combinación se asumirá que se tiene un string `T[0,...,n-1]` de tamaño n. El texto se divide en bloques de tamaño m, donde m es elegido de tal manera que cada una de las estructuras creadas puedan ajustarse según la memoria RAM de tamaño M y no sobrepasarla ($m < M$). Los bloques se procesan comenzando desde el final del texto. Suponer que hasta ahora se ha procesado `Z = T[i,...,n-1]` y a partir de ese bloque de texto se ha construído el arreglo de sufijos $\texttt{SA}_\texttt{Z}$. Luego se procede a construir el arreglo de sufijos $\texttt{SA}_\texttt{Y}$ del bloque de texto `Y = T[i-m,...,i-1]` y combinarlo con $\texttt{SA}_\texttt{Z}$ para formar $\texttt{SA}_\texttt{YZ}$. De manera muy general la combinación sigue 2 pasos:

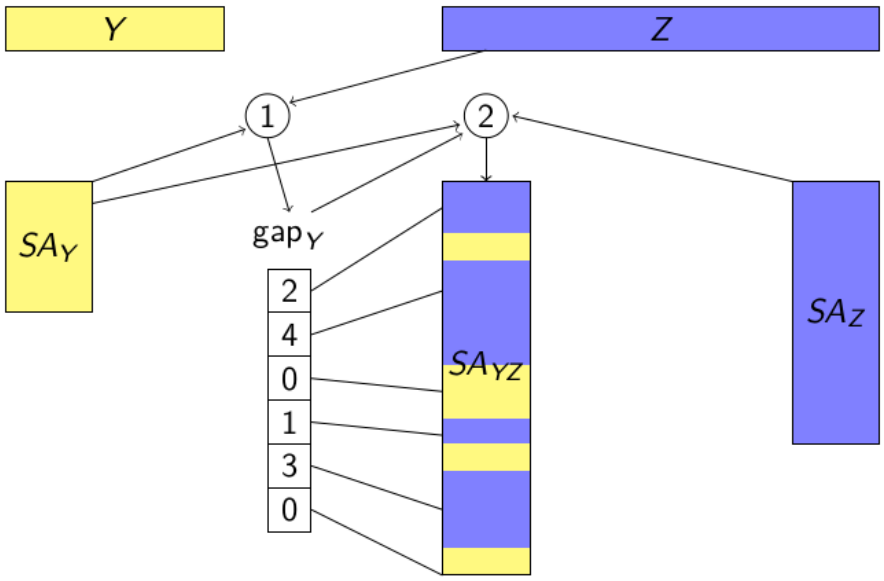


Figure 2.9: Combinación de 2 arreglos de sufijos que fueron creados de los fragmentos de texto Y y Z.

El primer paso es comparar el *suffix array* de Y con el texto Z para formar el arreglo $\texttt{gap}_Y$ que permitirá realizar el segundo paso que es ubicar aquellos sufijos que se ubican entre $\texttt{SA}_Z\texttt{[i]}$ y $\texttt{SA}_Z\texttt{[i+1]}$ y realizar la combinación.

Como son bastantes combinaciones a realizar, la complejidad de lectura y escritura en los archivos (*I/O complexity*) y bloques que se forman va por un orden proporcional de $O(\frac{n^2}{M})$ y un tiempo cuya complejidad va por el orden de $0(\frac{n^2}{M}\log(2+\frac{\log\rho}{\log n}))$. Como se trabajará con este tipo de algoritmo para el problema de la memoria, se dará mayor detalle de su implementación en el capítulo siguiente ("Implementación").

2.5.8. External Memory LCP Array

El arreglo de sufijos, en conjunto con su arreglo LCP son la base de soluciones de diversas aplicaciones [34]. Por consiguiente, cuando comenzó a surgir la implementación del *suffix array* en memoria externa entró a ser materia de investigación la búsqueda de la construcción de un arreglo en LCP también en memoria externa.

Como antes se mencionó, una manera optima de construir el arreglo LCP de un texto T es implementando previamente el arreglo de sufijos del texto T y utilizar este arreglo recien obtenido como *input* para construir el arreglo LCP. Kasai [22] fue quien pudo llevar esta idea a una implementación en memoria RAM (memoria interna) pero no fue hasta que Juha Kärkkäinen y Dominik Kempa [35] pudieron desarrollar una implementación de un arreglo LCP en memoria externa usando como *input* el arreglo de sufijos construido en memoria externa.

Según lo más reciente investigado [36] y para dar un ejemplo sencillo, para obtener el arreglo LCP en memoria externa, se tiene el siguiente texto o string `T` de tamaño n (`T = T[0,...,n-1] = T[0]T[1]...T[n-1]`). A

partir de este texto se obtiene su arreglo de sufijos `SA[0,...,n-1]` que es el arreglo ordenado de todos los sufijos del texto T. Entonces lo que se quiere obtener para todo i donde $0 \leq i < n$ es el arreglo `LCP[0,...,n-1]` (el valor de `LCP[0]` es indefinido) donde `LCP[i]` = *lcp(SA[i],SA[i-1])*, donde *lcp(j,k)* es la longitud del prefijo común más largo entre el sufijo j y k.

Para lograr esto se definen 2 nuevos arreglos, el *arreglo* Φ que consiste en que si se tiene `j = SA[i]`, entonces se cumple que Φ`[j] = SA[i-1]` para todo j donde $1 \leq j \leq n$. En otras palabras el sufijo Φ[i] es el predecesor lexicográfico inmediato del sufijo i y por lo tanto `SA[n-k]` = Φ_k`[SA[n]]` $\forall\ k \in$ [0,...,n].

El segundo arreglo a definir es conocido como el *arreglo LCP permutado* `PLCP[0,...,n-1]` es el arreglo LCP permutado desde el orden lexicográfico hacia el orden del texto, es decir, `PLCP[SA[j]] = LCP[j]` para todo j donde $1 \leq j \leq n$. Por consiguiente `PLCP[i]` = *lcp(i,*Φ*[i])* $\forall\ i \in$ [0,...,n-1]. Se considerará un ejemplo con el string `T = babaabbabbab$`:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	b	a	b	a	a	b	b	a	b	b	a	b	$
SA[i]	12	3	10	1	7	4	11	2	9	0	6	8	5
Φ[i]	9	10	11	12	7	8	0	1	6	2	3	4	-
LCP[i]	-	0	1	2	2	5	0	1	2	3	3	1	4
PLCP[i]	3	2	1	0	5	4	3	2	1	2	1	0	-

Table 2.12: Texto `T = babaabbabbab$` con su respectivos `SA[i]`, Φ`[i]`, `LCP[i]` y `PLCP[i]`.

Por ende los pasos generales para obtener el arreglo LCP en memoria externa tomando como entradas el texto T y el *suffix array* SA (obtenido también por memoria externa) son los siguientes:

1. Particionando el texto en bloques que se puedan ubicar en la memoria RAM, calcular Φ desde el SA obtenido previamente.
2. Calcular el arreglo PLCP usando el arreglo Φ y el texto T como base.
3. Calcular el arreglo LCP usando el arreglo PLCP obtenido, acá el texto se partiona en medio-segmentos (*half-segments*) de tal manera que 2 medio-segmentos puedan ser colocados en la memoria RAM.

Se obviarán por ahora detalles mayores de la explicación del funcionamiento de este algoritmo, ya que será analizado y profundizado en el capítulo “Implementación” cuando se use este algoritmo para obtener parte de los resultados de la memoria con la base de datos TrEMBL.

2.5.9. Transformación de Burrows-Wheeler

La transformación de Burrows-Wheeler [32], (abreviatura *BWT* del inglés *Burrows–Wheeler transform*, también conocida como compresión por ordenación de bloques), es un algoritmo usado en técnicas de compresión de datos. Fue inventado por Michael Burrows y David Wheeler en 1994 mientras trabajaban en el *DEC Systems Research Center* en Palo Alto, California. Esta transformación consiste en que se toma un trozo de texto T y se le realizan todas las rotaciones posibles, para luego ordenarlas de manera lexicográfica, obteniendo como salida la última columna ubicada en la derecha de la matriz:

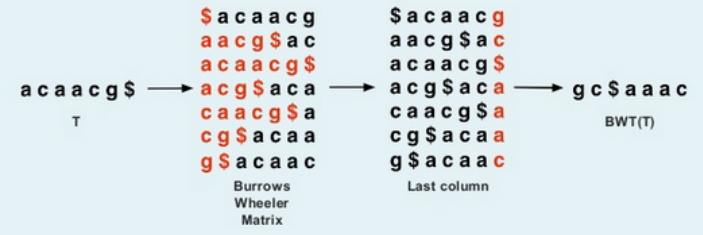


Figure 2.10: Ejemplo de la transformación de Burrows-Wheeler para la cadena acaacg$.

La aplicación más importante que tiene esta transformación es la de compresión de datos (usada por ejemplo la usada en la Alineación de Bowtie [33]) y también mantiene correlación con el arreglo de sufijos, el cual será utilizado en el *External Memory Suffix Array* para este trabajo (en el capítulo "Implementación" se detallará cúal es la función de esta transformación).

Chapter 3

Implementación

3.1. Propuesta considerada

Examinando las técnicas anteriormente revisadas, se llega al punto de que lo más factible es trabajar con las cadenas de secuencias utilizando un arreglo que las encadene una a una. Recordando los objetivos que se tienen para esta memoria, estas son:

1. Obtener la cantidad total de diferentes residuos de aminoácidos de tamaño $k = 1$ hasta 50 que existen para las bases de datos de UniProt-SwissProt, UniProt-TrEMBL, EROP-Moscow y proteínas humanas extraídas de SwissProt.
2. Encontrar para cada caso anterior cuáles son los residuos de aminoácidos que más se repiten.

Para realizar la primera tarea, será necesario construir un *suffix array* el cual será la base del arreglo LCP para realizar este objetivo. Considerando un ejemplo sencillo como la palabra BANANA$:

SA[]	***sufijo***
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Table 3.1: SA de la palabra BANANA$

Se puede apreciar que los números asociados a cada sufijo ya están ordenados como si fuera un arreglo de sufijos. Es posible obtener la cantidad total de diferentes substrings que componen esta palabra utilizando el arreglo LCP de la siguiente forma. Introduciendo los 2 siguientes conceptos:

length('X') = Largo de caracteres de la palabra 'X'.
LCP('Y','Z') = Prefijo más largo en común (*Longest Common Prefix*) entre los substrings 'Y' y 'Z'.

Y partiendo según el orden alfabético dado anteriormente, se hace el siguiente ejercicio:

Largo primer sufijo ordenado ('$') = 1 = *var*
Comienzo de pares de sufijos:

1. ('$','A$'): *var*+ = *length*('$A') - *LCP*('$','$A')
   var = *var* + 2 − 0 => 1 + 2 = 3
2. ('A$','ANA$'): *var*+ = *length*('ANA$') - *LCP*('A$','ANA$')
   var = *var* + 4 − 1 => 3 + 3 = 6
3. ('ANA$','ANANA$'): *var*+ = *length*('ANANA$') - *LCP*('ANA$','ANANA$')
   var = *var* + 6 − 3 => 6 + 3 = 9
4. ('ANANA$','BANANA$'): *var*+ = *length*('BANANA$') - *LCP*('ANANA$','BANANA$')
   var = *var* + 7 − 0 => 9 + 7 = 16
5. ('BANANA$','NA$'): *var*+ = *length*('NA$') - *LCP*('BANANA$','NA$')
   var = *var* + 3 − 0 => 16 + 3 = 19
6. ('NA$','NANA$'): *var*+ = *length*('NANA$') - *LCP*('NA$','NANA$')
   var = *var* + 5 − 2 => 19 + 3 = 22

Cantidad de diferentes substrings que hay en BANANA$: 22.

Lo que se hace en este caso es crear una variable y guardar la longitud del primer sufijo del SA (en este caso $) para luego realizar una comparación entre los sufijos consecutivos *i* y *j* adicionando en cada caso a la variable las longitudes respectivas de los sufijos *j*, y además **se le resta el prefijo común más largo entre estos sufijos consecutivos**. Por tanto el arreglo de sufijos y el arreglo LCP de la palabra BANANA$ queda de la siguiente forma:

SA[]	*LCP[]*	*sufijo*
6	0	$
5	0	A$
3	1	ANA$
1	3	ANANA$
0	0	BANANA$
4	0	NA$
2	2	NANA$

Table 3.2: SA y arreglo LCP de la palabra BANANA$

Entonces, particularizando el problema, ¿cómo sería posible obtener la cantidad total de diferentes substrings de un determinado tamaño? La clave está en el **arreglo LCP** obtenido, el cual se puede utilizar desde 2 perspectivas para realizar esta tarea.

3.1.1. Restando la cantidad máxima posible de diferentes substrings

Primero que todo hay que considerar la cantidad potencial máxima de diferentes substrings de tamaño k que se pueden obtener (la fórmula es $n - k + 1$ donde n es el largo de la palabra) y luego se recorre el arreglo LCP utilizando el valor de k como un comparador. Por ejemplo, de la palabra BANANA$ a simple vista se sabe que los diferentes substrings de tamaño 1 que se encuentran son 4, que son A, B, N y $. Usando la fórmula mencionada en el párrafo anterior se tiene que $7 - 1 + 1 = 7$ es la cantidad máxima de substrings de tamaño 1 de esta palabra. Recorriendo el arreglo LCP es necesario encontrar aquellos valores que sean **mayores o iguales que** k para restarlos a la cantidad máxima de diferentes substrings, porque ese valor indica en el arreglo de sufijos si determinado sufijo se **repite más de una vez**, por consiguiente esto indica que disminuye en una unidad la cantidad total de diferentes substrings de determinado tamaño. Aplicando en el caso anterior:

Máxima cantidad de diferentes substrings de tamaño 1 para BANANA$: $DS = 7$

a) $LCP[0] = 0$ => DS se mantiene => $DS = 7$
b) $LCP[1] = 0$ => DS se mantiene => $DS = 7$
c) $LCP[2] = 1$ => $DS = 7 - 1 = 6$
d) $LCP[3] = 3$ => $DS = 6 - 1 = 5$
e) $LCP[4] = 0$ => DS se mantiene => $DS = 5$
f) $LCP[5] = 0$ => DS se mantiene => $DS = 5$
g) $LCP[6] = 2$ => $DS = 5 - 1 = 4$

Diferentes substrings en total de tamaño 1 en la palabra BANANA$: 4.

Para los tamaños 2 hasta 7 (palabra completa) los diferentes substrings encontrados son los siguientes:

Tamaño 2 Substrings totales: 6 Elementos LCP ≥ 2: 2 DS de tamaño 2: $6-2=4$	Tamaño 3 Substrings totales: 5 Elementos LCP ≥ 3: 1 DS de tamaño 3: $5-1=4$	Tamaño 4 Substrings totales: 4 Elementos LCP ≥ 4: 0 DS de tamaño 4: $4-0=4$
Tamaño 5 Substrings totales: 3 Elementos LCP ≥ 5: 0 DS de tamaño 5: $3-0=3$	Tamaño 6 Substrings totales: 2 Elementos LCP ≥ 6: 0 DS de tamaño 6: $2-0=2$	Tamaño 7 Substrings totales: 1 Elementos LCP ≥ 7: 0 DS de tamaño 7: $1-0=1$

Sumando los DS encontrados entre los tamaños 1 hasta 7 el valor es de $4+4+4+4+3+2+1=22$, obteniendo el mismo valor de antes.

3.1.2. Aumentando la cantidad de diferentes substrings desde 0 considerando determinados tamaños de LCPs consecutivos

Esta segunda perspectiva considera recorrer el arreglo LCP con una pequeña variación, la que sería mover el primer elemento del arreglo (que siempre será 0) y dejarlo en la última posición desde izquierda a derecha:

0	0	1	3	0	0	2	->	0	1	3	0	0	2	0

Table 3.3: Arreglo LCP (tabla izquierda) mueve su primer elemento (0) hacia la última posición (tabla derecha).

El motivo de esto es identificar el prefijo común más largo entre los 2 sufijos consecutivos entre los sufijos x e y considerando al **primero o sufijo *x*** como el sufijo de referencia:

SA[]	***LCP[]***	***sufijo***
6	0	$
5	1	A$
3	3	ANA$
1	0	ANANA$
0	0	BANANA$
4	2	NA$
2	0	NANA$

Table 3.4: SA y arreglo LCP modificado de la palabra BANANA$

Para que sea más entendible, el último valor del arreglo LCP modificado es 0 ya que NANA$ es el último sufijo, y no tiene un sufijo posterior con el cual compararse.

En esta ocasión no se considerará la máxima cantidad de diferentes substrings de tamaño k $(n - k + 1)$ ya que todo se obtendrá del *suffix array* y de su arreglo LCP correspondiente, y el realizar esta modificación en el arreglo LCP permitirá lograr el segundo objetivo para este trabajo, que es el de encontrar a los conjuntos de péptidos que más se repiten para un determinado tamaño. Se puede ejemplificar esto con la búsqueda de los diferentes substrings de tamaño 3 en la palabra BANANA$:

Se inicializa con $DS = 0$.

a) $LCP[0] = 0$ => se mantiene => $DS = 0$ ya que el tamaño del sufijo es menor a 3 ($SA[0] = \$$).
b) $LCP[1] = 1$ => se mantiene => $DS = 0$ ya que ocurre el mismo fenómeno de antes ($SA[1] = A\$$).
c) $LCP[2] = 3$ => $SA[2] = ANA\$$, este sufijo con su siguiente sufijo consecutivo tiene como *longest common prefix* a ANA, por lo tanto se sabe que ANA se repite al menos 2 veces en la palabra; por ahora se seguirá dejando $DS = 0$.

Aquí viene la premisa del LCP consecutivo, ya que si el siguiente valor del arreglo LCP (para este caso $LCP[3]$) fuera mayor o igual que 3, entonces se tendría una nueva repetición del prefijo ANA, por lo tanto ahora serían 3 las veces que este prefijo estaría repetido en la palabra. Entendiéndolo de manera más formal, se tendrían l **valores consecutivos desde la posición** s **del arreglo LCP que serían mayores que** k **(que para este caso es 3)**, entregando un total de $l + 1$ repeticiones del sufijo $SA[s]$ de tamaño k. En caso contrario (valor del arreglo LCP menor que 3), se acabarían las repeticiones de determinado sufijo y se agrega una unidad al total de diferentes substrings encontrados.

d) $LCP[3] = 0$ => Aquí el valor es menor que 3, por lo tanto $DS = 0 + 1 = 1$ y se tiene que "ANA" se repite 2 veces.
e) $LCP[4] = 0$ => Tamaño $SA[4] = 7$, $SA[4, 3] = BAN$, por lo tanto $DS = 1 + 1 = 2$.
f) $LCP[5] = 2$ => Tamaño $SA[5] = 3$, $SA[5, 3] = NA\$$, por lo tanto $DS = 2 + 1 = 3$.
g) $LCP[6] = 0$ => Tamaño $SA[6] = 5$, $SA[6, 3] = NAN$, por lo tanto $DS = 3 + 1 = 4$.

Por consiguiente, se tiene que los diferentes substrings de tamaño 3 para la palabra BANANA$ son 4, ANA que se repite 2 veces, BAN, NA$ y NAN, que se repiten solo una vez. Sumando estas cantidades se tiene un valor de 5, que es el **número total de substrings de tamaño 3** $(n - k + 1)$.

Por ende para la realización del algoritmo se utilizará esta segunda premisa, considerando las restricciones pertinentes para este trabajo, no obstante, es necesario encontrar alguna estructura que permita guardar aquellos residuos que más se repitan para cumplir con el objetivo completo de este trabajo, este punto se explicará en la siguiente sección.

3.2. Cola de prioridad *(priority queue)*

La estructura conocida como *priority queue* [27] es un tipo de estructura contenedora implementada en C++ [26] similar a una lista, vector o arreglo, con su característica principal que al único elemento que se puede acceder es aquel que **sí o solo si tenga la prioridad o valor más alto que los demás elementos**. En otras palabras, se pueden ir agregando varios elementos a esta estructura y dependiendo del valor que tengan el elemento con la prioridad más alta puede variar, de tal forma que extrayendo todos los elementos del *priority queue* estos van siendo removidos desde aquel que tenga la prioridad más alta hasta llegar al elemento con la prioridad más baja. Este contexto es similar a un *heap* [26], donde los elementos pueden ser insertados en cualquier momento, y solamente el elemento con el máximo valor *(max heap)* puede ser obtenido (el elemento en la primera posición en el *priority queue*).

Se puede ejemplificar de la siguiente forma, se crea un *priority queue* de enteros y se insertan los valores: 14, 8, 35, 11 y 27 en ese orden.

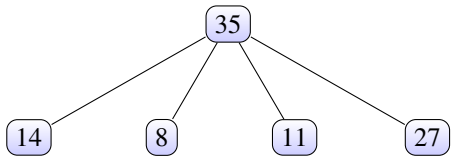


Figure 3.1: Grafo de muestra del formato de un *priority queue*.

El número 35 es el valor más alto y el que tiene la mayor prioridad, por lo tanto es el elemento al que se puede acceder. Ahora si ese valor se extrae, el *priority queue* se reordena y queda de la siguiente forma:

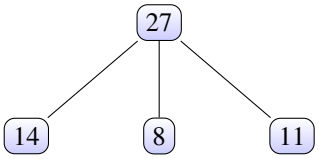


Figure 3.2: El *priority queue* con el número 35 extraído.

En este caso el segundo valor más alto del *priority queue* original (el número 27) es el nuevo elemento con la prioridad mayor y al que se puede acceder.

Ahora si se desea agregar un nuevo número a este arreglo, por ejemplo el 15, ocurre lo siguiente:

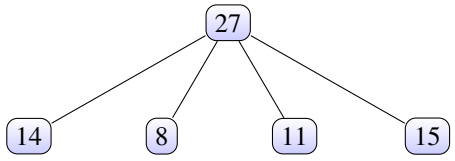


Figure 3.3: El *priority queue* con el número 15 agregado.

Como el número 15 es menor que 27, se mantiene este número como la mayor prioridad.

3.2.1. Algunos comandos de C++ para el *priority queue*

En C++ se define como `priority_queue<int>` "nombre_arreglo" para guardar valores enteros, para este caso se definirá `mypq` como el nombre de ejemplo para esta estructura. Las operaciones más importantes que se pueden realizar para este tipo de estructura son las siguientes:

1. `mypq.push(n)`: Inserta el número *n* en el *priority queue*.
2. `mypq.top()`: Retorna el valor con mayor prioridad del *priority queue*.
3. `mypq.pop()`: Remueve el valor con mayor prioridad del *priority queue*, disminuyendo el tamaño de esta estructura en uno.
4. `mypq.empty()`: Retorna si el *priority queue* está vacío.

A partir de estas operaciones es posible manipular el *priority queue* de manera más fácil, de tal manera que se usará esta estructura para guardar y obtener aquellos residuos de proteínas que más se repiten, sin embargo, para resolver el problema descrito será necesario guardar en este arreglo especial tanto el fragmento del péptido como la cantidad de repeticiones que posea, detalle que será visto en la sección de implementación.

3.3. Restricciones para la propuesta

Primero que todo, se extraerán las secuencias de polipéptidos de los archivos .fasta y se alinearán en una **única gran cadena** donde cada secuencia estará unida por un signo $, con esto será posible identificar en los arreglos si cierto sufijo está compuesto por este signo o no, de esa forma descartarlo dentro de los diferentes residuos de aminoácidos que se cuentan:

...MPSTLQVLAKKVLKENDHISR$EYHILKCWHEAPIILCFNGSKQM...

Table 3.5: 2 secuencias enlazadas en una cadena general utilizando el signo $ como unión.

Para esta cadena grande (de largo *m*), el arreglo de sufijos y el arreglo LCP tendrán tamaño *m*, por lo cual para determinar los diferentes substrings de tamaño *k* y aquellos substrings que más se repiten se debe recorrer el arreglo LCP completo, considerando:

1. Si el largo del sufijo es mayor o igual a *k*, entonces el arreglo LCP puede ser analizado, en caso contrario se omite y se continúa al siguiente valor del arreglo LCP.

2. Si el prefijo del sufijo revisado **solamente esté compuesto por los 20 aminoácidos conocidos** [4]. Otros aminoácidos que no han sido definidos, como B, J, O, U o X serán omitidos para este problema (si son parte del substring del sufijo revisado, se omitirá y se continuará al arreglo LCP siguiente) y considerados como prohibidos [23]. El signo $ también será incluído a este grupo de carácteres prohibidos.

Con respecto a esto es posible obtener los substrings diferentes y la cantidad de repeticiones que posee cada uno de estos, para posteriormente guardarlos en algun tipo de lista o vector. El problema es tratar de acceder a aquellos substrings que más se repiten, punto que se verá en la siguiente sección.

3.4. Algoritmo desarrollado

Para la obtención de los diferentes substrings se realizó un código implementado en lenguaje C++ [26] siguiendo varios puntos, en primera instancia para la base de datos de SwissProt y TrEMBL se realizó una extracción previa de datos.

El archivo "uniprot_sprot.fasta" está compuesto por 555426 proteínas con un peso total de 268 MB, mientras que el archivo "uniprot_trembl.fasta" está compuesto por 88032926 proteínas con un peso de 40 GB, el archivo EROP-Moscow pesa 1.5 MB y el archivo referente a las proteínas humanas pesa 47.2 MB. Para estos archivos la construcción del arreglo de sufijos y el arreglo LCP serán diferentes pero recibirán como *input* una cadena enlazada, cuya construcción será definida en la siguiente subsección.

3.4.1. Extracción de proteínas desde el archivo .fasta

EL archivo **.fasta** entrega cada polipéptido con un código o ID (que comienza con un >), a continuación en la misma línea se tiene al nombre taxativo de la proteína, y en la línea siguiente viene la cadena como tal, para luego repetir el proceso:

```
>sp|Q9BV40|VAMP8_HUMAN Vesicle-associated membrane protein 8 OS=Homo sapiens GN=VAMP8 PE=1 SV=1
MEEASEGGGNDRVRNLQSEVEGVKNIMTQNVERILARGENLEHLRNKTEDLEATSEHFKT
TSQKVARKFWWKNVKMIVLICVIVFIIILFIVLFATGAFS
>sp|Q9NZ42|PEN2_HUMAN Gamma-secretase subunit PEN-2 OS=Homo sapiens GN=PSENEN PE=1 SV=1
MNLERVSNEEKLNLCRKYYLGGFAFLPFLWLVNIFWFFREAFLVPAYTEQSQIKGYVWRS
AVGFLFWVIVLTSWITIFQIYRPRWGALGDYLSFTIPLGTP
>sp|P31151|S10A7_HUMAN Protein S100-A7 OS=Homo sapiens GN=S100A7 PE=1 SV=4
MSNTQAERSIIGMIDMFHKYTRRDDKIEKPSLLTMMKENFPNFLSACDKKGTNYLADVFE
KKDKNEDKKIDFSEFLSLLGDIATDYHKQSHGAAPCSGGSQ
>sp|Q15836|VAMP3_HUMAN Vesicle-associated membrane protein 3 OS=Homo sapiens GN=VAMP3 PE=1 SV=3
MSTGPTAATGSNRRLQQTQNQVDEVVDIMRVNVDKVLERDQKLSELDDRADALQAGASQF
ETSAAKLKRKYWWKNCKMWAIGITVLVIFIIIIVWVVSS
>sp|O70404|VAMP8_MOUSE Vesicle-associated membrane protein 8 OS=Mus musculus GN=Vamp8 PE=1 SV=1
MEEASGSAGNDRVRNLQSEVEGVKNIMTQNVERILSRGENLDHLRNKTEDLEATSEHFKT
TSQKVARKFWWKNVKMIVIICVIVLIIVILIILFATGTIPT
```

Figure 3.4: Archivo **.fasta** por defecto con varias proteínas

Para extraer las cadenas se implementó el siguiente código en C++:

```
ifstream fin("uniprot_sprot.fasta"); //abrir archivo base
if(!fin){
        cerr << "Couldn't open the input file!";
        return(1);
}
ofstream outputfile; //crear archivo destino de cadena
outputfile.open("substrings.txt"); //definir nombre de archivo a crear
string line;
int cantidad_ss = 1;

getline(fin, line);
getline(fin, line); //tomar secuencia (1)

while(fin){
        if(line[0] == '>'){ // revisar primer elemento del string (2)
                outputfile << "$";
                cantidad_ss = cantidad_ss + 1;
        }
        else{
                outputfile << line;
        }
getline(fin, line); // continuar con la siguiente linea (3)
}

outputfile.close();
```

Pseudocódigo 3.1: Creación de cadena de proteínas

Lo que hace este código es crear un nuevo archivo "substrings.txt" en la variable **outputfile** donde se guardará la cadena de proteínas. Con *getline* se lee la primera línea del archivo .fasta que está guardada en la variable **fin**, e inmediatamente después se llama nuevamente a *getline* para leer la segunda línea del archivo .fasta, que es **una secuencia o parte de ella (1)**. Luego se condiciona a realizar una de las 2 tareas siempre y cuando el archivo .fasta no se haya leído completamente:

1. Si el primer elemento de la línea leída del archivo .fasta es >, significa que se llegó al final de una secuencia, por lo tanto es el comienzo de la siguiente (es la linea de definición de la proteína), en ese caso se le agrega el signo $ al archivo destino como separador **(2)**.

2. En caso contrario, se le agrega directamente la línea completa al archivo destino, ya que esta línea solamente está compuesta por **la secuencia en sí**.

Finalmente se vuelve a llamar a *getline* para continuar con la siguiente línea del archivo .fasta **(3)**.

En resumen el formato del nuevo archivo "substrings.txt" es una sola línea que tiene concatenada todas las proteínas:

TSCPGGNHPVCCSTDLCNK$MKTL. . .SDLT$LKCNKLVPLFYKTCP

Table 3.6: Ejemplo de cadena encontrada en el archivo destino

Teniendo esta gran línea ya se puede construir el arreglo de sufijos y el arreglo LCP. Considerar como dato relevante que guardando esta cadena en una variable tipo *string* cada caracter ocupa el tamaño de 1 byte de capacidad [24], y esto determinará de qué manera se construirán los arreglos para la solución de este problema.

Por defecto se tiene que el tamaño de UniProt-SwissProt es de 268 MB y su cadena de proteínas tiene un peso de 199 MB, mientras que para UniProt-TrEMBL su tamaño por defecto es de 40 GB y su cadena de proteínas posee un peso de 30 GB, si se tiene en consideración que para guardar la cadena en una variable ésta se almacena en la memoria RAM del ordenador. En primera instancia no hay problema con la base de datos de SwissProt, de EROP-Moscow y el archivo con las proteínas humanas ya que el espacio ocupado es suficiente si se trabaja en un computador normal (si se guardan los arreglos en una variable tipo *vector*, cada elemento del vector pesa 4 bytes), pero para la cadena creada en función de UniProt-TrEMBL parece ser más complejo, ya que el string para almacenar la cadena requeriría 30 GB de memoria RAM, inclusive ocupando un servidor con 50 GB de capacidad de RAM sería inviable construir los arreglos (en cálculos sencillos se necesitaría de una RAM de al menos 300 GB para realizar esta tarea). Una posible solución para esto se revisará más adelante en la implementación para la base de datos UniProt-TrEMBL.

3.4.2. Implementación solución para base de datos UniProt-SwissProt, EROP-Moscow y Proteínas Humanas

Para la implementación del algoritmo se trabajó por medio del lenguaje C++ en base al algoritmo de Kasai [25], [22] para obtener el arreglo LCP en base al arreglo de sufijos. Antes que nada se define una estructura y una función comparativa que serán de soporte para la construcción del arreglo de sufijos:

```
1  //  Struct para guardar la informacion de un sufijo
2  struct suffix
3  {
4          int index; // Guardar el indice original
5          int rank[2]; // Guarda los ranks y el rank pair siguiente
6  };
7
8  // Una funcion comparativa usada por sort () para comparar 2 sufijos
9  // Compara 2 pares, y retorna 1 si el primer par es mas pequeno
```

```
int cmp(struct suffix a, struct suffix b)
{
        return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
                (a.rank[0] < b.rank[0] ?1: 0);
}
```

Pseudocódigo 3.2: Definición previa de estructuras para construir el arreglo de sufijos.

suffix es una estructura que sirve para almacenar el índice original de determinado sufijo y los *rank* donde se guardan los sufijos y su sufijo siguiente como pares. La función *cmp* es usada para comparar entre los valores de los *rank* entre 2 sufijos, y retorna 1 si el primer par es más pequeño, si son igual se continúa la comparación con el siguiente par, y es el comparador base que será usado por la función *sort* más adelante.

Arreglo de sufijos implementado

Ahora se procederá a explicar la función principal que toma un string "txt" de tamaño n como entrada, y que construye y retorna el arreglo de sufijos para el string dado. Será explicado en secciones que fueron numerados en parte del código (con un //(número)) para aprovechar en mejor manera el formato de este documento:

```
vector<int> buildSuffixArray(string txt, int n){

        struct suffix *suffixes = new struct suffix [n]; //(1)
        for (int i = 0; i < n; i++){
                suffixes[i].index = i;
                suffixes[i].rank[0] = txt[i] - 'a';
                suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
        }
        sort(suffixes, suffixes+n, cmp);
        int *ind = new int [n]; //(2)
        for (int k = 4; k < 2*n; k = k*2){
                int rank = 0;
                int prev_rank = suffixes[0].rank[0];
                suffixes[0].rank[0] = rank;
                ind[suffixes[0].index] = 0;//(3)
                for (int i = 1; i < n; i++){
                        if (suffixes[i].rank[0] == prev_rank &&
                          suffixes[i].rank[1] == suffixes[i-1].rank[1]){ \\(a)
                                prev_rank = suffixes[i].rank[0];
                                suffixes[i].rank[0] = rank;
                        }
                        else
```

```
23                          {
24                                  prev_rank = suffixes[i].rank[0]; \\(b)
25                                  suffixes[i].rank[0] = ++rank;
26                          }
27                          ind[suffixes[i].index] = i;
28                  }
29                  for (int i = 0; i < n; i++){
30                          int nextindex = suffixes[i].index + k/2;
31                          suffixes[i].rank[1] = (nextindex < n)?
32                              suffixes[ind[nextindex]].rank[0]: -1;
33                  }
34                  sort(suffixes, suffixes+n, cmp); //(4)
35          }
36          vector<int>suffixArr;
37          for (int i = 0; i < n; i++){
38          suffixArr.push_back(suffixes[i].index);
39      } //(5)
40          delete [] ind;
41          delete [] suffixes;
42          return suffixArr; //(6)
43  }
```

Pseudocódigo 3.3: Función principal arreglo de sufijos (1)

En **(1)** se crea un estructura que almacena sufijos y sus indices (*suffixes*), que se alojará en memoria dinámica (operador new), esta estructura es necesaria para ordenar los sufijos de manera alfabética y mantener sus antiguos índices mientras se ordena.

Luego con la función *sort* se ordenan los sufijos de acuerdo a sus **2 primeros caracteres**. Una vez realizado esto de manera posterior se ordenan los sufijos de acuerdo a sus primeros 4 caracteres, luego con sus primeros 8 caracteres y asi sucesivamente hasta $k > 2^n$ donde n es el largo del string analizado, para realizar esta tarea se crea un arreglo dinámico *ind* que es necesario para obtener el indice original en la estructura suffixes[] **(2)**.

En cada una de estas iteraciones se le asigna los valores de *rank* e índice al primer sufijo **(3)**, luego se le asigna los *rank* a los siguientes sufijos hasta n siguiendo ciertas reglas:

a) Si el primer *rank* y los siguientes ranks son iguales a aquellos de los sufijos anteriores en el arreglo, asignar el mismo nuevo rank a ese sufijo.
b) En caso contrario aumentar el *rank* y asignar.

Después se le asigna el próximo *rank* a cada sufijo para posteriormente ordenar los sufijos según los primeros k caracteres **(4)**. Se realiza este proceso hasta terminar las iteraciones para la variable k.

Finalmente en **(5)** se crea un vector que permita almacenar los valores obtenidos de los índices ordenados del arreglo *suffixes*. Este vector de enteros *suffixArr* se convertirá en el **arreglo de sufijos** del string *txt*, luego se borran los arreglos dinámicos *suffixes* e *ind* para desocupar el espacio asignado en memoria RAM.

Posteriormente **(6)** se retorna el arreglo de sufijos final, que permitirá construir el arreglo LCP.

Arreglo LCP implementado

La implementación de este arreglo se hizo en base al *suffix array* implementado, siguiendo lo descrito por Kasai [25], [22]. El código realizado es el siguiente:

```
1  vector<int> lcp_str(string txt, vector<int> suffixArr){
2          int n = suffixArr.size();
3          vector<int> lcp(n, 0);
4          vector<int> invSuff(n, 0); \\(1)
5          for (int i=0; i < n; i++)
6                  invSuff[suffixArr[i]] = i;
7
8          int k = 0; \\(2)
9          for (int i=0; i<n; i++){
10                 if (invSuff[i] == n-1){
11                         k = 0;
12                         continue; \\(3)
13                 }
14                 int j = suffixArr[invSuff[i]+1];\\(4)
15                 while (i+k<n && j+k<n && txt[i+k]==txt[j+k]) \\(5)
16                         k++;
17                 lcp[invSuff[i]] = k; \\(6)
18                 if (k>0)
19                         k--;
20         }
21         return lcp; \\(7)
22 }
```

Pseudocódigo 3.4: Función principal arreglo LCP (1)

Lo que se hace en primera instancia es guardar el largo del arreglo, crear un vector para almacenar el arreglo LCP a construir y además crear un arreglo auxiliar $invSuff$ para almacenar el inverso del arreglo de sufijos previamente creado. Por ejemplo si $suffixArr[0]$ es 5, entonces $invSuff[5]$ debiese ser 0. Y esto será usado para obtener el siguiente string del arreglo de sufijos **(1)**.

Luego se llena con valores el arreglo $invSuff$ y se inicializa una variable para guardar la longitud del LCP del

sufijo **(2)**. A partir de acá comienza a revisar todos los sufijos para asignarles su valor en el arreglo LCP.

Un detalle importante se ubica en **(3)**, ya que si el sufijo actual está ubicado en la posicion $n-1$, entonces ya no hay un siguiente substring a considerar, por lo tanto el LCP no es definido para este substring y se hace cero. Esto permitirá enfocar la solución del problema aumentando la cantidad de diferentes substrings desde 0 considerando determinados tamaños de LCPs consecutivos.

Posteriormente en la iteración se define una variable j que contiene el índice del siguiente substring a ser considerado para compararlo con el actual substring, es decir, el siguiente string en el arreglo de sufijos **(4)**. Después la función *while* comienza a revisar el sufijo desde el k-ésimo indice donde al menos $k-1$ caracteres serán similares, mientras esta condición se cumpla, k va aumentando **(5)**.

Posteriormente se asigna el valor del LCP encontrado para el actual sufijo **(6)**, y k se borra para utilizarlo en la siguiente iteración. Finalmente se retorma el arreglo LCP obtenido **(7)**.

Con esto se tienen a disposición los 2 arreglos necesarios como base para resolver el problema de la memoria.

Implementación programa principal

Ahora se procederá a explicar la implementación en la cual cómo se obtuvieron los diferentes substrings totales para cada k entre 1 hasta 50 y la obtención de los residuos que más se repiten para cada caso. En primer lugar cuando la cadena de proteínas fue construída se definió una variable llamada `cantidad_ss`, que guarda la cantidad total de proteínas analizadas. Utilizando las implementaciones del arreglo de sufijos y el arreglo LCP ocurre lo siguiente:

```
1  ifstream file("substrings.txt");
2  stringstream buffer;
3  buffer << file.rdbuf();
4  string str = buffer.str(); \\(1)
5
6  unsigned t0, t1, t2, t3, t4, t5;
7  t0 = clock();
8  vector<int>suffixArr = buildSuffixArray(str, str.length());
9  t1 = clock();
10 double t12 = (double(t1-t0)/CLOCKS_PER_SEC); \\(2)
11
12 t2 = clock();
13 vector<int>lcp = lcp_str(str, suffixArr);
14 t3 = clock();
15 double t23 = (double(t3-t2)/CLOCKS_PER_SEC); \\(3)
16
17 ofstream resultados, k_utilizados; \\(4)
```

```
resultados.open("resultados_sa_lcp.txt");
resultados << "Construccion SA: " << t12 << " segundos" << endl;
resultados << " \n";
resultados << "Construccion LCP: " << t23 << " segundos" << endl;
resultados << " \n";
resultados << "Total: " << cantidad_ss << " proteinas" << endl; \\(5)
resultados.close();
```

Pseudocódigo 3.5: Obtención de los arreglos SA y LCP para la cadena de proteínas.

En primera instancia se debe guardar la cadena de proteínas creada en el archivo "substrings.txt" en el string `str` **(1)**, para luego construir el arreglo de sufijos **(2)** que se guarda en el vector `suffixArr` y el arreglo LCP que se guarda en el vector `lcp` **(3)**. En ambos casos se obtienen los tiempos respectivos que demora en construir estos arreglos.

Después se crean 2 variables para crear archivos, en uno se guardarán los tiempos de construcción de los arreglos, y en otro los diferentes substrings para determinados *k* con sus respectivos residuos que más se repiten **(4)**. A la primera variable se abre el archivo "resultados_sa_lcp.txt" y le agregan al archivo los tiempos de construcción de los 2 arreglos **(5)**, posteriormente este archivo se cierra.

El formato de los arreglos obtenidos es el siguiente:

Arreglo SA	...	34	22	15	89	901	1042	45	4	47	59	...
Arreglo LCP	...	2	0	21	3	8	3	2	0	0	9	...

Table 3.7: Formatos de los arreglos SA y LCP

Recordar que todos los elementos del arreglo de sufijos son diferentes.

Para cada *k* los arreglos se deben recorrer desde el principio hasta el final, con esto será posible obtener los diferentes residuos de péptidos y los que más se repiten. Esto se implementó de la siguiente forma:

```
int inicio = cantidad_ss - 1;
int n = suffixArr.size(); \\(1)
k_utilizados.open("resultados_k1to50.txt");
for (int k1 = 1; k1 < 51; k1++){
    t4 = clock();
    subcadena arr[1];
    priority_queue<subcadena, vector<subcadena>, comparador> mypq; \\(2)
    int activador = 0;
    int contador;
    int ds = 0; \\(3)
```

```
11      for (int temp = inicio; temp < n; temp++){ \\(4)
12          if (lcp[temp] >= k1 && activador == 0){ \\(a)
13              string sc = str.substr(suffixArr[temp], k1);
14              if (sc.find_first_of("$BOUXZ")==std::string::npos){
15                  arr[0].nombre = sc;
16                  contador = 2;
17                  activador = 1;
18              }
19          }else if (lcp[temp] >= k1 && activador == 1){ \\(b)
20              string sc = str.substr(suffixArr[temp], k1);
21              if (sc.find_first_of("$BOUXZ")==std::string::npos){
22                  contador = contador + 1;
23              }else{
24                  arr[0].veces = contador;
25                  if (contador > 0){
26                      mypq.push(arr[0]);
27                  }
28                  ds = ds + 1;
29                  activador = 0;
30              }
31          }else if (lcp[temp] < k1 && activador == 0){ \\(c)
32              string sc = str.substr(suffixArr[temp], k1);
33              int scnumber = sc.size();
34              if (scnumber == k1){
35                  if (sc.find_first_of("$BOUXZ")==std::string::npos){
36                      ds = ds + 1;
37                  }
38              }
39          }else if (lcp[temp] < k1 && activador == 1){ \\(d)
40              arr[0].veces = contador;
41              if (contador > 0){
42                  mypq.push(arr[0]);
43              }
44              ds = ds + 1;
45              activador = 0;
46          }
47      }
48
49      k_utilizados << "Diferentes residuos para " << k1 << " es " << ds << endl;
50      k_utilizados << " \n";
51      for (int posicion = 0; posicion < 20; posicion++){
```

```
52          k_utilizados << mypq.top().nombre << "␣" << mypq.top().veces;
53          mypq.pop();
54          k_utilizados << endl; \\(5)
55      }
56      t5 = clock();
57      double t45 = (double(t5-t4)/CLOCKS_PER_SEC);
58      k_utilizados << "␣\n";
59      k_utilizados << "Tiempo␣utilizado:␣" << t45 << "␣segundos" << endl;
60      k_utilizados << "--------------------------" << endl;
61      k_utilizados << "␣\n";
62  }
63  k_utilizados.close(); \\(6)
```

Pseudocódigo 3.6: Obtención de los diferentes substrings de tamaño k y los 20 substrings que más se repiten para la cadena de proteínas.

Lo primero que se hace es definir 2 variables, una (`n`) es para guardar el largo del arreglo de sufijos (que tiene el mismo tamaño que el arreglo LCP) y la otra variable (`inicio`) guarda el total de proteínas analizadas menos una unidad, que será usada en las iteraciones **(1)**. Luego se abre el archivo "resultados_k1to50.txt" y se deja así hasta que la iteración completa termine, ya que en este archivo se guardarán los resultados obtenidos con el algoritmo.

La iteración principal abarcará desde $k = 1$ hasta 50, es decir que para cada k se recorrerán los arreglos desde principio a fin. Una vez definido k se crea un arreglo `arr[1]` y el posterior *priority queue* que está creado en base a una estructura llamada `subcadena` y una clase llamada `comparador`:

```
1   struct subcadena
2   {
3     string nombre;
4     int veces;
5   };
6
7   class comparador
8   {
9    public:
10     bool operator()(const subcadena& a, const subcadena& b)
11     {
12          return a.veces<b.veces;
13     }
14  };
```

Pseudocódigo 3.7: Implementación de `subcadena` y `comparador` como soportes del *priority queue* a utilizar.

Esta estructura permitirá guardar la cadena de substring y la cantidad de veces que aparezca en la cadena de proteínas, mientras que la clase es usada para comparar entre los substrings agregados al *priority queue* y retornar cuál de ellos **tiene el valor más alto**, gracias a esto es posible obtener cuál es el substring que más se repite y su cantidad **(2)**.

Antes de iterar sobre los arreglos, se definen 3 variables importantes **(3)**, que son:

1. `activador`: Esta variable es usada para verificar si se están sumando repeticiones de un determinado residuo o no. Toma valores entre 0 y 1, donde 0 significa que se están buscando substrings y 1 significa que se están agregando repeticiones de determinado substring encontrado. Esto está asociado directamente con el arreglo creado `arr`, si se guarda en este arreglo solamente el substring y continuan las iteraciones, `activador` toma el valor 1 y si se guarda la cantidad de repeticiones en el arreglo `arr`, `activador` toma el valor 0.
2. `contador`: Esta variable es usada para guardar la cantidad de repeticiones de un determinado residuo.
3. `ds`: Esta variable guarda la cantidad de diferentes substrings encontrados de tamaño *k*.

Con esto definido se puede comenzar a iterar sobre los arreglos. La primera posición denominada como `inicio` está ubicada en `cantidad_ss-1` ya que a cada proteína se le concatena con el signo $, por lo tanto si se concatenan *N* proteínas de un archivo .fasta, se tienen en total *N* veces el signo $, y considerando que el arreglo de sufijos ordena cada sufijo según su orden alfabético, este signo se ubica antes de la letra *a* y además se tiene que este signo pertenece a los carácteres prohibidos, por consiguiente no es necesario revisar estos sufijos **(4)**.

Ahora se comienza a iterar sobre el arreglo LCP (variable `temp` que indica la posición en el arreglo), verificando si el valor es mayor o igual a *k*. Ante esto se tienen 4 casos que se pueden formar:

a) Si `lcp[temp]` >= *k* y `activador` = 0: Si esto ocurre **(a)** se guarda el substring que corresponde al residuo de proteínas en la posición `temp` de tamaño *k* (`substr(suffixArr[temp], k)`) y se revisa si este substring posee alguno de los caracteres prohibidos, en caso negativo se guarda este residuo en la estructura `arr` y se inicializa `contador` con un valor de 2, porque acá se tienen al menos **2 repeticiones** del substring hallado y `activador` toma el valor de 1 ya que comienza la búsqueda de más repeticiones del substring encontrado; en caso afirmativo se continúa con la siguiente iteración del arreglo.

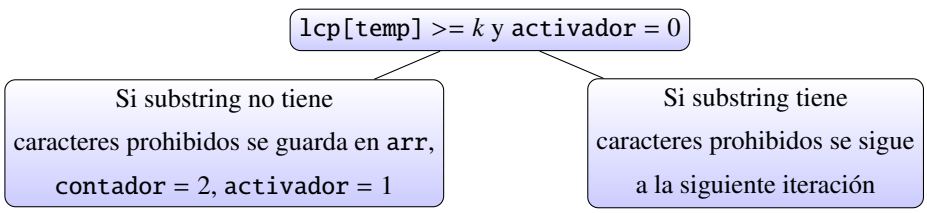


Figure 3.5: Esquema Caso *a*)

b) Si `lcp[temp]` >= *k* y `activador` = 1: Si este caso pasa ahora se deberá comparar el sufijo de esta posición `temp` cumple con la condición de no tener caracteres prohibidos **(b)**, si la cumple a `contador` se le debe adicionar una unidad; si esto no se cumple es decir que se acabaron las repeticiones para el substring y se guarda en la estructura `arr` la cantidad de repeticiones obtenidas hasta ese momento para el substring previamente guardado en esa estructura, para después guardar el substring y sus repeticiones en forma de nodo en el *priority queue*, y posteriormente agregarle una unidad a la variable `ds` (de diferentes substrings) ya que el substring anterior no se volverá a encontrar en el arreglo de sufijos, finalmente la variable `activador` se hace 0.

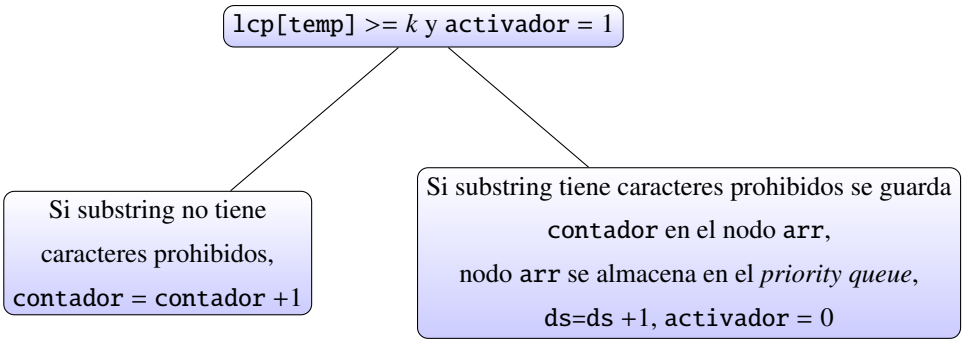


Figure 3.6: Esquema Caso *b*)

c) Si `lcp[temp]` < *k* y `activador` = 0: Como el valor del arreglo LCP en la posición `temp` es menor que *k* acá simplemente se verifica que el substring de tamaño *k* correspondiente a la posición `temp` en el arreglo de sufijos tenga un tamaño igual a *k* y que no tenga caracteres prohibidos **(c)**, si no tiene entonces a la variable `ds` se le agrega una unidad y se guarda el substring con el valor 1 con forma de nodo en el *priority queue*.

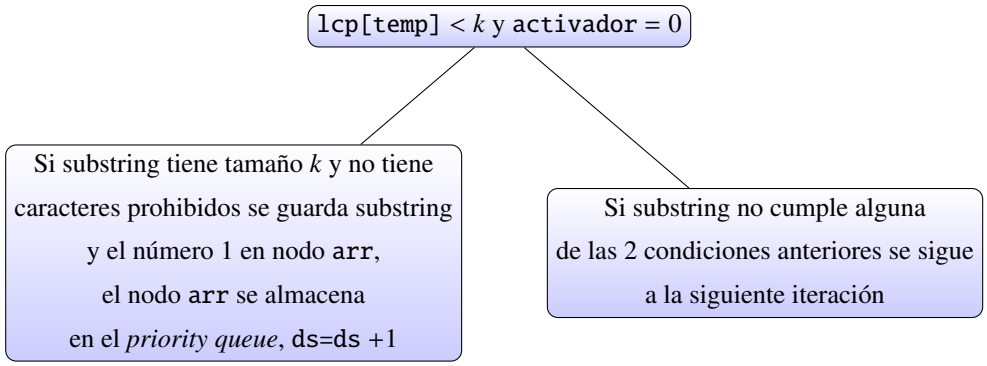


Figure 3.7: Esquema Caso *c*)

d) Si `lcp[temp]` < *k* y `activador` = 1: Acá ya se tiene guardado un substring en el arreglo `arr` porque el activador está con valor 1, por ende el arreglo LCP actual es menor al *k* requerido **(d)** y se guarda la variable `contador` en el arreglo `arr` para luego almacenar esta variable como nodo en el *priority queue*, se le agrega una unidad a la variable `ds` y la variable `activador` se hace 0.

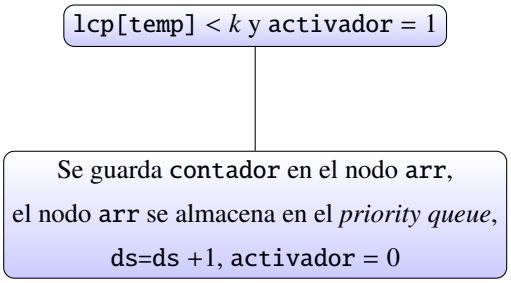


Figure 3.8: Esquema Caso *d*)

Una vez recorrido el arreglo se guarda en el archivo destino ("resultados_k1to50.txt") los diferentes substrings obtenidos (`ds`) y luego utilizando los comandos explicados anteriormente sobre el *priority queue* se extraen los **20 primeros residuos de proteínas que más se repiten junto a su cantidad total** y se guardan en este archivo **(5)**, y luego se reinicia la iteración principal con el siguiente *k* a utilizar.

Una vez usados todos los *k* permitidos **(6)** se cierra el archivo destino y se termina de ejecutar el archivo.

Formato salida archivos

Después de haber ejecutado este algoritmo se obtiene el archivo de salida mencionado anteriormente, llamado "resultados_k1to50.txt" y que tiene el siguiente formato:

```
PNVGKS 1827

Tiempo utilizado: 36.0557 segundos
---------------------------

Diferentes substrings para 7 es 84118859

NNNNNNN 27091
QQQQQQQ 16272
SSSSSSS 6927
EEEEEEE 5342
AAAAAAA 4966
PPPPPPP 4189
GGGGGGG 4140
SDSDSDS 4106
DSDSDSD 4059
HTGEKPY 3252
TTTTTTT 2881
IHTGEKP 2029
ATVITNL 1679
LPWGQMS 1660
HVDHGKT 1659
YVLPWGQ 1651
GYVLPWG 1651
VLPWGQM 1648
TPGHVDF 1648
DTPGHVD 1648

Tiempo utilizado: 39.4517 segundos
---------------------------

Diferentes substrings para 8 es 100896814

NNNNNNNN 23879
QQQQQQQQ 13508
```

Figure 3.9: Extracto del archivo de salida "resultados_k1to50.txt"

Como se puede apreciar en la imagen, se han guardado las cantidades de diferentes substrings obtenidos para cada k (en la imagen se muestra para $k = 7$), los 20 residuos que más repiten con su respectiva cantidad de repeticiones encontradas y el tiempo en recorrer los arreglos para determinado k. Las líneas salteadas indican el cambio hacia el siguiente k (para la imagen es $k = 8$) nuevamente mostrando los diferentes substrings encontrados y los 20 residuos que más se repiten. Este proceso inicia con $k = 1$ hasta llegar a $k = 50$.

3.4.3. Implementación solución para base de datos UniProt-TrEMBL

Antes de comenzar a definir la explicación de la implementación anterior, se dijo que para archivos muy grandes que superen los 1000 MB de capacidad como la base de datos UniProt-TrEMBL era inviable guardar los arreglos (tanto de sufijos como LCP) en memoria RAM ya que el espacio requerido no sería el suficiente ($9n$ donde n es el tamaño de bytes de la cadena de strings). Ante esa necesidad se procedió a buscar una alternativa para trabajar este gran archivo.

En el intertanto de la búsqueda de opciones, se descubrió que existen algoritmos que desarrollan arreglos de sufijos de archivos cuyos tamaños eran superiores a la memoria RAM corriente (computadores de hasta 8 GB de capacidad) y dentro de esas opciones se decidió trabajar con algoritmos en memoria externa.

La implementación realizada se hizo en lenguaje C++ en base al trabajo e investigación efectuada por Juha Kärkkäinen y Dominik Kempa ([30], [36]) para obtener el arreglo de sufijos y el arreglo LCP en memoria externa. Ambos casos usan un archivo MAKEFILE para ejecutar las implementaciones, en consecuencia para cada uno de estos casos se explicarán los pasos más relevantes que permiten obtener los arreglos como resultado final.

Arreglo de sufijos EM implementado

Una vez MAKEFILE se obtiene un archivo

Arreglo LCP EM implementado

Implementación programa principal

En [30]

Chapter 4

Resultados y análisis

Aquí se mostrarán los resultados que se obtuvieron con los algoritmos mencionados en la sección anterior. Haciendo un pequeño recuento, los *datasets* usados son archivos que utilizan un formato **.fasta** y son los siguientes:

Archivo	Proteínas	Tamaño (MB)	Tamaño cadena (MB)
Base de datos SwissProt	555426	268.3	199.5
Base de datos TrEMBL	89396316	40900 (aprox.)	30200 (aprox.)
Base de datos EROP-Moscow	14785	1.5	0.3526
Proteínas humanas	86298	47.2	38.2

Table 4.1: *Datasets* utilizados para la obtención de resultados

4.1. Sistema utilizado y método de experimentación

El trabajo se realizó en un ordenador *Acer Aspire ES14*, no obstante las implementaciones se ejecutaron en un servidor prestado para la ocasión por la Universidad. Los datos relevantes a esta máquina se aprecian en el **Cuadro 4.2**.

Los códigos fueron realizados en C++, mediante el uso del entorno de desarrollo integrado (IDE) "Code::Blocks" [37]. Al momento de ejecutar las implementaciones todo fue realizado en el terminal con los comandos básicos (asumiendo como ejemplo que el archivo base es "prueba.cpp"):

a) `g++ prueba.cpp -o pruebas`: Una vez que se compila el archivo base se crea el ejecutable "pruebas".
b) `./pruebas`: Se activa el ejecutable y comienza a funcionar el código.

Componente	Descripción
Nombre CPU	Intel(R) Xeon(R) CPU E5-2630 @2.30GHz
CPU (s)	12
Caché (capacidad)	15 MB
Memoria RAM (capacidad)	60 GB
Sistema Operativo	Ubuntu Server 16.04 (Linux versión 4.4.0-51)

Table 4.2: Especificaciones del sistema utilizado

Los resultados se guardaron en archivos .txt, y se mostrarán en la siguiente sección.

4.2. Resultados obtenidos

4.2.1. Tiempos obtenidos

Aquí se adjuntan las tablas con los tiempos logrados con las ejecuciones de los algoritmos con las 2 implementaciones explicadas en el capítulo anterior. Siguiendo este esquema se mostrarán en una tabla los tiempos obtenidos utilizando el algoritmo "SwissProt" (primera implementación explicada en el capítulo anterior relativa a alojar el arreglo de sufijos y el arreglo LCP en memoria RAM directamente) y en otra tabla los tiempos utilizando el algoritmo "TrEMBL" (capítulo anterior, segunda implementación relacionada a la obtención de los arreglos en memoria externa) ajustando la cantidad de memoria RAM usada a 20 GB para tanto el arreglo de sufijos como el arreglo LCP en memoria externa.

En esta primera tabla se mostrará el tiempo en construir el arreglo de sufijos, el arreglo LCP y el tiempo empleado para obtener los diferentes substrings y los residuos que más se repiten con k entre 1 a 50 ("Tiempo programa").

Archivo	Tiempo SA (s)	Tiempo LCP (s)	Tiempo programa (s)
SwissProt	2637.18	35.09	2758.82
TrEMBL	-	-	-
EROP-Moscow	1.92	0.03	3.58
Proteínas humanas	422.49	9.06	3.78 (k=1)

Table 4.3: Tiempos de las implementaciones realizadas con los 4 *datasets* utilizando el algoritmo "SwissProt".

Observando los tiempos del Cuadro 4.3 se aprecia que es mucho más rápido obtener el arreglo LCP que el el *suffix array* para los casos analizados ya que la implementación del arreglo de sufijos considera ordenar los primeros p sufijos (donde p aumenta de manera exponencial en la iteración) y ese paso requiere tiempo,

mientras que el arreglo LCP usa el arreglo de sufijos recién creado y la cadena para crear esta nueva estructura, y eso corresponde a analizar los elementos consecutivos del arreglo de sufijos respectivo (ver cuadro 4.3). Para este algoritmo no se pudo obtener los resultados con el archivo de "TrEMBL" ya que la cadena formada tenía un tamaño de 30 GB, con lo cual **ocupaba el 50% de la capacidad disponible de la memoria RAM en el servidor**, en consecuencia generar 2 arreglos con la memoria RAM restante en base a este algoritmo no fue posible (problema tipo `std::bad_alloc` en C++).

Archivo (EM)	**Tiempo SA (s)**	**Tiempo LCP (s)**	**Tiempo guardado elementos (s)**	**Tiempo programa (s)**
SwissProt	17.98	57.23	59.39	4533.09
TrEMBL	7981.08	10785.03	7722.16	8.5 días
EROP-Moscow	0.23	0.19	0.11	7.07
Proteínas humanas	4.10	9.40	9.81	11.02

Table 4.4: Tiempos de las implementaciones realizadas con los 4 *datasets* utilizando el algoritmo "TrEMBL" (EM = *External Memory*).

Con respecto a los tiempos obtenidos en el Cuadro 4.4 se aprecia que la proporción entre tiempos de construcción de los arreglos LCP/SA son muy similares entre sí, a pesar de que el arreglo LCP se obtiene directamente del *suffix array* recién obtenido. El motivo principal radica en que al aplicar los métodos de "memoria externa" (*external memory*) se aplica la complejidad de I/O (entrada/salida, lecturas y escrituras en un archivo) y eso contribuye a **un aumento significativo de los tiempos en la obtención del arreglo LCP** (ver cuadro 4.5). Por otro lado también al cuadro 4.4 se agrega la columna de "Tiempo guardado elementos" el cual es el tiempo que toma guardar los elementos de ambos arreglos en un archivo .txt para utilizarlos en el programa principal.

Proporción tiempos LCP/SA	**Algoritmo SwissProt (%)**	**Algoritmo TrEMBL (%)**
SwissProt	1.33	318.29
TrEMBL	-	135.13
EROP-Moscow	1.56	82.60
Proteínas humanas	2.14	299.26

Table 4.5: Relación entre tiempos de obtención del arreglo de sufijos y el arreglo LCP para los 2 algoritmos.

Apreciando la cantidad de memoria RAM asignada (20 GB), para SwissProt, EROP-Moscow y Proteínas humanas solamente es necesario un bloque de texto, ya que la capacidad de RAM es mucho más grande que los archivos mencionados, mientras que para el archivo de "TrEMBL" ya es necesario particionar el texto en bloques donde se obtienen arreglos de sufijos parciales (*partial suffix arrays*) y de esa forma proseguir con la combinación (*merging phase*) de estos arreglos para obtener el arreglo de sufijos y el arreglo LCP definitivo.

Comparando datos entre los cuadros 4.3 y 4.4, se tiene que con motivos de comodidad es más sencillo trabajar estos archivos con el algoritmo "SwissProt" ya que no se ocupa complejidad I/O a la hora de construir los arreglos, no obstante el utilizar el algoritmo "TrEMBL" usa complejidad I/O y requiere más espacio en el disco duro, a costa de optimizar el tiempo de construcción de los arreglos. Otro punto relevante en este caso son los tiempos de ejecución del programa principal. Para ambos algoritmos la obtención de los diferentes substrings para cada k y aquellos residuos que más se repiten sigue el mismo procedimiento con la importante diferencia que para el algoritmo "TrEMBL" los valores de los arreglos **se guardan en un archivo auxiliar en vez de un vector en memoria interna**, por lo tanto acceder a estos números requiere una mayor cantidad de tiempo de procesamiento.

A nivel general la suma de los tiempos totales de ejecución de los algoritmos se presenta en el siguiente cuadro:

Archivo	**Algoritmo SwissProt (s)**	**Algoritmo TrEMBL (s)**
SwissProt	5431.09	4667.69
TrEMBL	-	8.8 días
EROP-Moscow	9.11	7.6
Proteínas humanas	435.33	34.33

Table 4.6: Tiempos totales comparando entre los 2 algoritmos utilizados

En todos los casos (a excepción del archivo TrEMBL) se aprecia que los tiempos usando el algoritmo de memoria externa son mucho más rápidos que utilizando el algoritmo "SwissProt" ya que, como anteriomente se indicó, existe la correlación de optimización de tiempo y uso de mayor memoria RAM disponible en sacrificio de guardar elementos en disco duro.

Tamaño elementos guardados en disco duro para algoritmo "TrEMBL"

Para el algoritmo de memoria externa, en el proceso de construcción se guardaron archivos a partir de la cadena de strings de tamaño n (el tamaño de la cadena para cada archivo está en el cuadro 4.1). Estos archivos son 4:

1. Archivo de $5n$ bytes para alojar arreglo de sufijos (formato .sa).
2. Archivo de $5n$ bytes para alojar arreglo LCP (formato .lcp).
3. Archivo que guarda los valores accesibles para el arreglo de sufijos (formato .txt).
4. Archivo que guarda los valores accesibles para el arreglo LCP (formato .txt).

Para los 2 primeros *items* estos arreglos usan un formato especial de salida (.sa y .lcp) por lo mismo no es posible acceder a ellos de manera trivial (usando comandos `open` y `getline` en C++). Por consiguiente en el

proceso de construcción de estos arreglos se guardaron estos valores por separado en un archivo .txt para poder usarlos en el programa principal y de esa manera obtener los datos.

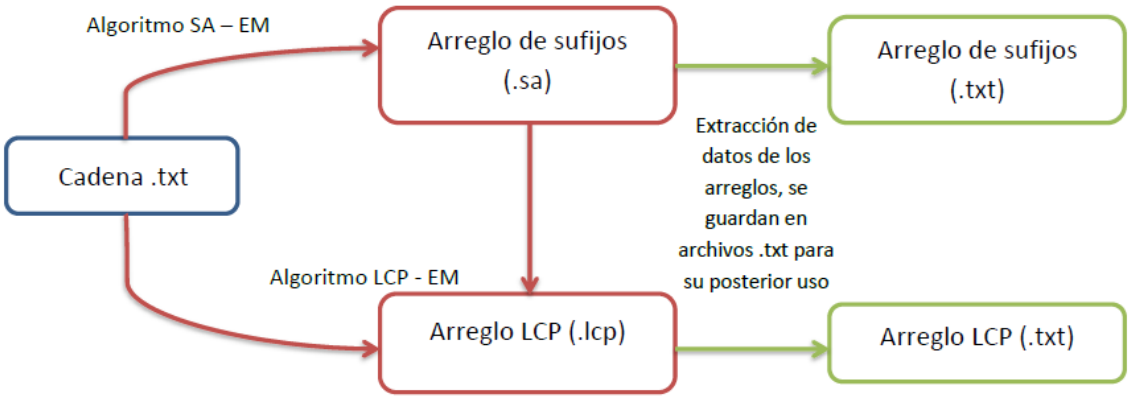


Figure 4.1: Esquema sencillo de transformación de archivos para usarlos en el programa principal

Esto implica un gran uso de disco duro como punto en contra, más aún si se tiene que el archivo "TrEMBL" es muy pesado. Por lo tanto los pesos usados según cada archivo son los siguientes:

Archivo	**Archivo .sa**	**Archivo .lcp**	**Archivo accesible SA**	**Archivo accesible LCP**	**Total**
SwissProt	997.5 MB	997.5 MB	1.9 GB	524.8 MB	4.41 GB
TrEMBL	151 GB	151 GB	351.3 GB	85.3 GB	738.6 GB
EROP-Moscow	1.8 MB	1.8 MB	2.4 MB	807.9 KB	6.80 MB
Proteínas humanas	191 MB	191 MB	332.6 MB	120.8 MB	835.4 MB

Table 4.7: Tamaños de archivos guardados en disco duro utilizando algoritmo "TrEMBL"

En consecuencia los tamaños de los datos se hace gigantesco, en especial con el archivo que guarda los valores del arreglo SA. Recordar que los elementos que componen un arreglo de sufijos son todos números diferentes por ende al ir guardando números más grandes el peso requirido se va haciendo más alto; no obstante el arreglo LCP sí admite valores repetidos.

Así pues el usar este algoritmo de memoria externa implica un importante uso de la capacidad del disco duro, sin embargo en computadores actuales es factible guardar tamaña cantidad de peso, lo cual se mantendría si incluso se desea aumentar la capacidad de memoria RAM disponible para construir los arreglos en memoria externa, descendiendo los tiempos de obtención de estos archivos; aunque es difícil encontrar ordenadores con una memoria RAM superior a 20 GB con excepción de servidores de alta capacidad.

4.2.2. Diferentes substrings entre $k = 1$ hasta $k = 10$

Los resultados obtenidos tomaron un rango de k que va entre 1 hasta 50. En esta subsección se mostrarán los 10 primeros valores de k (desde 1 hasta 10) ya que son los valores más relevantes con respecto a información de péptidos y porcentaje de residuos encontrados (las tablas completas serán colocadas en la sección **Apéndice** si se desean identificar mayores detalles de los valores obtenidos).

Siguiendo la "Tabla 1" que aparece en [2] se mostrarán los diferentes substrings encontrados para cada uno de los 4 archivos analizados, comparando según corresponda entre los algoritmos utilizados.

Archivo SwissProt

Los resultados obtenidos utilizando el archivo de SwissProt son los siguientes:

Largo péptido (N)	**N-péptidos posibles (20^N)**	**Encontrados "SwissProt"**	**Porcentaje encontrado (%)**	**Encontrados "TrEMBL"**	**Porcentaje encontrado (%)**
1	20	20	100	20	100
2	400	400	100	400	100
3	8000	8000	100	8000	100
4	160000	159999	99.99	159998	99.99
5	3200000	3113509	97.29	3113510	97.29
6	64000000	32921109	51.43	32921095	51.43
7	1280000000	84118859	6.57	84118817	6.57
8	25600000000	100896814	0.39	100896768	0.39
9	512000000000	105834330	0.020	105834282	0.020
10	10240000000000	108976567	0.0010	108976518	0.0010

Table 4.8: Resultados de diferentes substrings obtenidos con el archivo SwissProt

Es posible identificar que los todos los potenciales residuos hasta un tamaño $k = 3$ existen en esta base de datos, consideración importante si se tiene que las proteínas que componen este archivo no supera los 600000. A partir de allí a medida que k asciende aparecen más péptidos pero no aumentan al mismo ritmo que los potenciales residuos, por tanto aparece el brusco descenso de porcentaje, a ello se le agrega que los diferentes substrings encontrados se atascan en los 100 millones de proteínas si k sigue en aumento. Con $k = 10$ o superior el porcentaje de péptidos diferentes encontrados en relación a los potenciales péptidos **se hace ínfimo**, a niveles millodecimales. Por esta razón que aunque se presenten diferentes valores de residuos encontrados con los 2 algoritmos, si se lleva eso a porcentaje para ambos casos es lo mismo.

Archivo TrEMBL

Acá solamente se obtuvieron resultados con el algoritmo "TrEMBL" por las razones explicadas anteriormente, estos resultados son los siguientes:

Largo péptido (N)	**N-péptidos posibles (20^N)**	**Encontrados "TrEMBL"**	**Porcentaje encontrado (%)**
1	20	20	100
2	400	400	100
3	8000	8000	100
4	160000	160000	100
5	3200000	3200000	100
6	64000000	63817907	99.71
7	1280000000	1024659629	80.05
8	25600000000	5743538889	22.43
9	512000000000	10114868387	1.97
10	10240000000000	11524607918	0.11

Table 4.9: Resultados de diferentes substrings obtenidos con el archivo TrEMBL

La base de datos "TrEMBL" tiene un total de prácticamente 90 millones de proteínas, casi 164 veces el total de proteínas del archivo "SwissProt", por lo tanto es esperable encontrar una mayor cantidad de diferentes residuos y un porcentaje mayor con respecto al universo completo de posibles péptidos a encontrar para determinado k. Pues bien, los resultados muestran que esa tendencia se cumple a cabalidad, porque en este archivo **se encontraron que existen todos los potenciales residuos desde los aminoácidos hasta los pentapéptidos (péptidos compuestos de 5 aminoácidos)**. Con residuos de mayor tamaño ($k = 6$ en adelante) comienza a bajar el porcentaje de péptidos encontrados, y sucede el mismo fenómeno que con el archivo SwissProt, los residuos diferentes encontrados se estancan en los 11 mil millones y a pesar que siguen en aumento conforme el valor de k se incrementa, no lleva el mismo ritmo que los potenciales residuos a encontrar, tanto es así que para los péptidos compuestos de 10 aminoácidos los residuos encontrados solamente conforman el 0.1% del universo total de residuos posibles.

Archivo EROP-Moscow

Para este archivo los resultados que se obtuvieron con ambos algoritmos son los siguientes:

Largo péptido (N)	**N-péptidos posibles (20^N)**	**Encontrados "SwissProt"**	**Porcentaje encontrado (%)**	**Encontrados "TrEMBL"**	**Porcentaje encontrado (%)**
1	20	20	100	20	100
2	400	400	100	400	100
3	8000	7907	98.83	7907	98.83
4	160000	69218	43.26	69219	43.26
5	3200000	116454	3.63	116457	3.63
6	64000000	125036	0.19	125039	0.19
7	1280000000	125750	0.0098	125752	0.0098
8	25600000000	124054	4.84×10^{-4}	124055	4.84×10^{-4}
9	512000000000	121103	2.36×10^{-5}	121103	2.36×10^{-5}
10	10240000000000	117457	1.14×10^{-6}	117457	1.14×10^{-6}

Table 4.10: Resultados de diferentes substrings obtenidos con el archivo EROP-Moscow

La base de datos EROP-Moscow se compone de 14785 oligopéptidos cuyos tamaños varían entre 2 a 50 aminoácidos, por lo mismo se hace mucho más complejo encontrar un vasto conjunto de diferentes residuos para cada k. No obstante se encuentran **todos los posibles aminoácidos y dipéptidos existentes**, y con residuos de tamaño $k = 3$ en adelante el porcentaje de péptidos encontrados se hace muy pequeño, lo cual es lógico ya que para tal cantidad de péptidos en este archivo es muy difícil obtener una mayor cantidad de diferentes péptidos. Este valor se estanca hasta los 125000 aproximadamente, y a partir de los residuos de tamaño $k = 8$ el valor disminuye. Una explicación aparente para esto es que existen muchos péptidos compuestos de 5 aminoácidos o menos, por lo mismo para buscar péptidos con una magnitud de más de 5 aminoácidos son pocos los oligopéptidos del conjunto completo que cumplen la condición anteriormente mencionada, que se hace más notoria inclusive aumenta el valor de k. Por ejemplo en el caso de la búsqueda de residuos de tamaño $k = 50$ se obtuvieron 116 elementos, que son las 116 proteínas compuestas de 50 aminoácidos en la base de datos de EROP-Moscow.

Archivo Proteínas Humanas

Para este archivo solo se obtuvieron los diferentes substrings para $k = 1$ y los resultados son los siguientes:

Largo péptido (N)	**N-péptidos posibles (20^N)**	**Encontrados "SwissProt"**	**Porcentaje encontrado (%)**	**Encontrados "TrEMBL"**	**Porcentaje encontrado (%)**
1	20	20	100	20	100

Table 4.11: Resultados de diferentes substrings obtenidos con el archivo Proteínas Humanas

En lo que corresponde a las proteínas que existen en el ser humano se puede rescatar que utilizando ambos

algoritmos **se obtienen los 20 aminoácidos buscados (100%)**. Como para este archivo se está trabajando con solo un $k = 1$, se realizará un mayor análisis biológico en la subsección siguiente, para este caso, aquellos aminoácidos que más se repiten.

Diferencias entre resultados obtenidos utilizando el algoritmo "SwissProt" y el algoritmo "TrEMBL"

En los cuadros mostrados anteriormente, para los archivos SwissProt y EROP-Moscow se identifica que según un determinado k varios de los diferentes substrings encontrados entrega diferentes valores usando ambos algoritmos ($k = 4$ o superior para base de datos SwissProt y varios k para EROP-Moscow). ¿Esto implica que uno de los algoritmos utilizados está incorrecto? ¿O acaso los 2 algoritmos están incorrectos?
Cuando se comenzó a trabajar con este tipo de cadenas, destacados investigadores como Gonzalo Navarro [38] han expuesto que utilizar datos compactos es lo más conveniente, porque el procesamiento de texto es más rápido y expedito de comprender. Dentro de esto grupo enorme de estructuras que trabajan con datos tipo cadena se eligió el arreglo de sufijos y el arreglo LCP.
Pues bien, para llevar a tierra todo lo investigado en información es importante obtener **datos concretos de experimentación**, que se han mostrado en cada uno de los cuadros anteriores. Por consiguiente al experimentar con estos algoritmos es bastante factible obtener varias diferencias de resultados obtenidos, que según apreciando el porcentaje de péptidos encontrados, son los mismos, a tal punto que se necesita **llegar a la tercera o cuarta cifra significativa para recíen encontrar diferencias** entre estos porcentajes.

Observando desde el lado de la implementación realizada, los arreglos obtenidos se aplican según esquemas establecidos por investigadores, para el lado del algoritmo desarrollado por Kasai [22] como para los algoritmos de memoria externa ([30], [36]), por ende se hallan diferencias de implementación los cuales definan estas diferencias de resultados. Otra cosa importante es que las cadenas usadas son tan grandes que ocupan tamaños cercanos a los 200 millones de caracteres, a los cuales se le revisa su posición y sufijo al que pertenece para poder extraer aquel residuo con su número (cantidad de veces que aparece en la base de datos) y guardarlo en el *priority queue*, por ende el procedimiento realizado también puede dar con estas diferencias que a la larga son muy pequeñas por los porcentajes que se obtuvieron.

Para intentar corroborar se podría usar una opción poco ortodoxa, que sería obtener los diferentes substrings para cada archivo usando un algoritmo de fuerza bruta, pero realizar esto no es recomendable ya que a nivel de tiempo es el algoritmo más malo que se puede hallar, si se entrega como *input* el archivo de TrEMBL a este algoritmo demoraría meses en obtener los resultados solicitados.

Tomando en consideración esto, se procederá a resumir en un gráfico de Porcentaje v/s k cómo evoluciona el porcentajes de péptidos encontrados según los péptidos posibles a encontrar. Como para cada archivo los porcentajes obtenidos con cada algoritmo utilizado son los mismos se mostrarán 4 líneas, cada una según el archivo al que pertenezca:

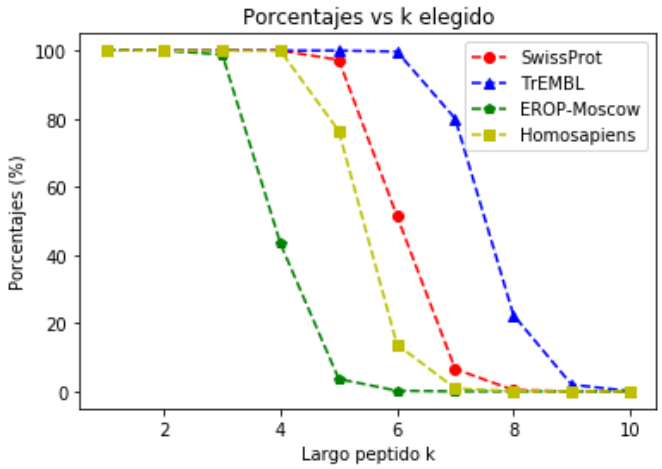


Figure 4.2: Gráfico que muestra los porcentajes de residuos encontrados según el largo de péptido k para los 4 archivos

Observando el gráfico, se aprecia que los valores obtenidos en los porcentajes dependen **directamente de la cantidad de proteínas** que contenga la base de datos (la relación de tamaños de los archivos sigue el siguiente patrón: TrEBML > SwissProt > Homosapiens > EROP-Moscow), es decir, si se tienen más proteínas será mayor la diversidad de residuos a encontrar. Hasta $k = 3$ es posible identificar en todos los archivos que poseen casi todos los residuos posibles, a partir de ese punto comienzan a bajar las probabilidades de localizar un mayor porcentaje de substrings de tamaño k. En todos los casos esta bajada en probabilidades es brusca, tomando en cuenta que los posibles péptidos de tamaño k se rigen según la fórmula 20^k. Al contener esta fórmula como base al 20, el salto exponencial que implica ir de k a $k + 1$ es muy fuerte, tanto es así que por ejemplo el archivo TrEMBL, a pesar de ser 164 veces más grande que el archivo SwissProt, abarca solamente un 20% de los posibles residuos que se pueden encontrar de tamaño 8 ($20^8 = 25600000000$), mientras que SwissProt abarca menos del 0.5% de residuos posibles para este k.
Para $k = 10$ los porcentajes de substrings encontrados para los 4 archivos es prácticamente 0, por esa razón es que se decidió mostrar los el comportamiento de los diferentes substrings localizados hasta un tamaño igual a 10, para k mayores a este valor los resultados de los porcentajes serían muy pequeños, prácticamente a escalas atómicas.

4.2.3. Substrings de determinado k que más se repiten

Ahora se mostrarán los substrings de tamaño k que más se repiten en los 4 *datasets* trabajados. Para cada k se obtuvieron los 20 residuos que más se repiten, no obstante por motivos de espacio se mostrarán en esta sección los 5 substrings que más se repiten variando k entre 1 hasta 10 como se hizo en el análisis anterior (las tablas completas con los substrings que más se repiten variando k entre 1 hasta 50 serán colocadas en la sección **Apéndice**). Primero se presentarán las tablas obtenidas y luego se le hará un análisis a estos resultados.

Archivo SwissProt

k	Algoritmo SwissProt	Algoritmo TrEMBL
1	L - 19206802	L - 19206802
	A - 16435230	A - 16435229
	G - 14086821	G - 14086821
	V - 13664059	V - 13664059
	E - 13406543	E - 13406543
2	LL - 1886886	LL - 1886886
	AA - 1714607	AA - 1714607
	AL - 1683204	AL - 1683204
	LA - 1673336	LA - 1673336
	LS - 1337091	LS - 1337091
3	AAA - 232439	AAA - 232439
	ALA - 185750	ALA - 185750
	LLL - 180299	LLL - 180299
	LAA - 179704	LAA - 179704
	AAL - 177751	AAL - 177751
4	AAAA - 47284	AAAA - 47284
	NNNN - 46321	NNNN - 46321
	SSSS - 39923	SSSS - 39923
	QQQQ - 36956	QQQQ - 36956
	EEEE - 29213	EEEE - 29213
5	NNNNN - 36977	NNNNN - 36977
	QQQQQ - 26232	QQQQQ - 26232
	SSSSS - 17652	SSSSS - 17652
	AAAAA - 17020	AAAAA - 17020
	EEEEE - 13572	EEEEE - 13572

k	Algoritmo SwissProt	Algoritmo TrEMBL
6	NNNNNN - 31269	NNNNNN - 31269
	QQQQQQ - 20197	QQQQQQ - 20197
	SSSSSS - 10379	SSSSSS - 10379
	AAAAAA - 8392	AAAAAA - 8392
	EEEEEE - 8077	EEEEEE - 8077
7	NNNNNNN - 27091	NNNNNNN - 27091
	QQQQQQQ - 16272	QQQQQQQ - 16272
	SSSSSSS - 6927	SSSSSSS - 6927
	EEEEEEE - 5342	EEEEEEE - 5342
	AAAAAAA - 4966	AAAAAAA - 4966
8	NNNNNNNN - 23879	NNNNNNNN - 23879
	QQQQQQQQ - 13508	QQQQQQQQ - 13508
	SSSSSSSS - 5006	SSSSSSSS - 5006
	SDSDSDSD - 3905	SDSDSDSD - 3905
	DSDSDSDS - 3875	DSDSDSDS - 3875
9	NNNNNNNNN - 21325	NNNNNNNNN - 21325
	QQQQQQQQQ - 11488	QQQQQQQQQ - 11488
	SSSSSSSSS - 3791	SSSSSSSSS - 3791
	SDSDSDSDS - 3770	SDSDSDSDS - 3770
	DSDSDSDSD - 3745	DSDSDSDSD - 3745
10	NNNNNNNNNN - 19315	NNNNNNNNNN - 19315
	QQQQQQQQQQ - 9899	QQQQQQQQQQ - 9899
	SDSDSDSDSD - 3653	SDSDSDSDSD - 3653
	DSDSDSDSDS - 3634	DSDSDSDSDS - 3634
	SSSSSSSSSS - 3003	SSSSSSSSSS - 3003

Table 4.12: Residuos más repetidos usando k entre 1 hasta 10 para el archivo SwissProt

Apreciando bien se identifica que para los 2 algoritmos los resultados obtenidos son los mismos, por otra parte es que las cantidades de los valores más repetidos van disminuyendo muy rápidamente (para $k = 1$ los valores repetidos rondan las 10 millones de repeticiones y para $k = 2$ rondan solamente el millón de repeticiones). Otro punto relevante es que muchas repeticiones son residuos compuestos solamente por un aminoácido.

Archivo TrEMBL

k	Algoritmo TrEMBL
1	L - 2966351173
	A - 2715689443
	G - 2175272504
	V - 2065665475
	S - 2030276782
2	AA - 313028096
	LL - 306619288
	LA - 285314569
	AL - 281904750
	AG - 212968976
3	AAA - 45269369
	LAA - 34321839
	ALA - 34225481
	AAL - 33850558
	LLL - 32856884
4	AAAA - 9197193
	SSSS - 5754913
	ALAA - 5176104
	LLLL - 5111818
	AALA - 5085892
5	QQQQQ - 3210070
	AAAAA - 3108174
	SSSSS - 2473516
	GGGGG - 1679232
	PPPPP - 1473476

k	Algoritmo TrEMBL
6	QQQQQQ - 2355101
	AAAAAA - 1496140
	SSSSSS - 1411002
	GGGGGG - 898024
	NNNNNN - 861564
7	QQQQQQQ - 1798936
	SSSSSSS - 922266
	AAAAAAA - 880843
	WTVYPPL - 815253
	GWTVYPP - 814768
8	QQQQQQQQ - 1408675
	GWTVYPPL - 813630
	TGWTVYPP - 798282
	GTGWTVYP - 798215
	MIFFMVMP - 771616
9	QQQQQQQQQ - 1124827
	GTGWTVYPP - 797778
	TGWTVYPPL - 797178
	LLLLSLPVL - 756549
	LLLSLPVLA - 751235
10	QQQQQQQQQQ - 911438
	GTGWTVYPPL - 796679
	LLLLSLPVLA - 749816
	PDMAFPRMNN - 667479
	FPRMNNMSFW - 661041

Table 4.13: Residuos más repetidos usando *k* entre 1 hasta 10 para el archivo TrEMBL

Acá solo se muestran los resultados con los algoritmos de memoria externa ya que con el algoritmo SwissProt no se pudo analizar esta base de datos (los motivos fueron explicados anteriormente).
Los péptidos más repetidos obtenidos para este archivo entregan resultados muy similares a los conseguidos con el archivo SwissProt con respecto a **los residuos**. Este fenómeno será explicado más adelante.

Archivo EROP-Moscow

k	Algoritmo SwissProt	Algoritmo TrEMBL
1	G - 30373	G - 30373
	L - 23867	L - 23867
	K - 21956	K - 21956
	S - 21486	S - 21486
	C - 21365	C - 21365
2	GL - 3307	GL - 3307
	RR - 2904	RR - 2904
	GG - 2872	GG - 2872
	GK - 2558	GK - 2558
	AA - 2493	AA - 2493
3	RRR - 1253	RRR - 1253
	FGL - 483	FGL - 483
	GGG - 479	GGG - 479
	LSG - 469	LSG - 469
	AAK - 451	AAK - 451
4	RRRR - 712	RRRR - 712
	GGGG - 183	GGGG - 183
	GCSC - 172	GCSC - 172
	CCSG - 167	CCSG - 167
	LSGL - 162	LSGL - 162
5	RRRRR - 373	RRRRR - 373
	FDEID - 144	FDEID - 144
	DEIDR - 139	DEIDR - 139
	CGLSG - 128	CGLSG - 128
	GGGGG - 121	GGGGG - 121

k	Algoritmo SwissProt	Algoritmo TrEMBL
6	RRRRRR - 160	RRRRRR - 160
	FDEIDR - 127	FDEIDR - 127
	NFDEID - 114	NFDEID - 114
	CGLSGL - 98	CGLSGL - 98
	GLSGLC - 94	GLSGLC - 94
7	NFDEIDR - 102	NFDEIDR - 102
	CGLSGLC - 88	CGLSGLC - 88
	VCGLSGL - 79	VCGLSGL - 79
	MEHFRWG - 76	MEHFRWG - 76
	FDEIDRS - 72	FDEIDRS - 72
8	VCGLSGLC - 71	VCGLSGLC - 71
	EHFRWGKP - 68	EHFRWGKP - 68
	MEHFRWGK - 68	MEHFRWGK - 68
	SMEHFRWG - 65	SMEHFRWG - 65
	YSMEHFRW - 62	YSMEHFRW - 62
9	MEHFRWGKP - 68	MEHFRWGKP - 68
	SMEHFRWGK - 65	SMEHFRWGK - 65
	YSMEHFRWG - 61	YSMEHFRWG - 61
	EHFRWGKPV - 59	EHFRWGKPV - 59
	GGTCNTPGC - 59	GGTCNTPGC - 59
10	SMEHFRWGKP - 65	SMEHFRWGKP - 65
	YSMEHFRWGK - 61	YSMEHFRWGK - 61
	MEHFRWGKPV - 59	MEHFRWGKPV - 59
	SYSMEHFRWG - 53	SYSMEHFRWG - 53
	EHFRWGKPVG - 46	EHFRWGKPVG - 46

Table 4.14: Residuos más repetidos usando *k* entre 1 hasta 10 para el archivo EROP-Moscow

Con este archivo ya es posible obtener los resultados con los 2 algoritmos sin problemas, además se aprecia que los valores obtenidos son los mismos para cada algoritmo utilizado.

Para esta base de datos los péptidos obtenidos distan de los encontrados en los archivos anteriores, de hecho no aparecen residuos compuestos por un solo aminoácido (a excepción de “R” y “G”), lo que tiene relación con la **base de datos utilizada**. Al ser este un análisis más biológico, se tratará más adelante.

Archivo Proteínas Humanas

k	Algoritmo SwissProt	Algoritmo TrEMBL
1	L - 3979937	L - 3979937
	S - 3123633	S - 3123633
	A - 2631167	A - 2631167
	E - 2467325	E - 2467325
	G - 2448081	G - 2448081
	P - 2336530	P - 2336530
	T - 2248068	T - 2248068
	V - 2229816	V - 2229816
	K - 2082746	K - 2082746
	R - 2012813	R - 2012813
	I - 1863090	I - 1863090
	D - 1723766	D - 1723766
	Q - 1715384	Q - 1715384
	F - 1448281	F - 1448281
	N - 1448115	N - 1448115
	Y - 1078310	Y - 1078310
	H - 997696	H - 997696
	M - 922926	M - 922926
	C - 805709	C - 805709
	W - 527194	W - 527194

Table 4.15: Residuos más repetidos usando $k = 1$ para el archivo Proteínas Humanas

Al ser solamente $k = 1$ a analizar para este archivo (se consideró eso ya que la finalidad era identificar a aquellos aminoácidos que más están presentes en el ser humano) se decide mostrar para ambos algoritmos la cantidad de veces que se repiten cada uno de los 20 aminoácidos.
Se puede identificar que los resultados obtenidos utilizando cualquiera de los 2 algoritmos es el mismo, lo cual entrega veracidad a los resultados logrados.

Análisis de resultados (enfoque algorítmico)

Los resultados conseguidos con los 2 algoritmos en esta sección son similares, por ende otorgan veracidad a la hora de definir que la estrategia de utilizar arreglos de sufijos y arreglos LCP era la indicada. Más importante aún, queda de manifiesto que la opción de uso de la estructura conocida como ***priority queue*** es la correcta

ya que permite guardar substrings enlazados con su respectivo valor para luego extraer aquellos residuos con más altos valores. Su utilización permitió identificar estos datos importantes y construir estos cuadros de información, por consiguiente parece ser la opción indicada a la hora de buscar ordenar números, strings u otro tipo de variables siguiendo un orden de jerarquía determinado con anterioridad.

Análisis de resultados (enfoque biológico)

Para realizar este análisis, al ser un ámbito fuera de la informática se solicitó ayuda y consejos del profesor Alexander Zamyatnin. Este análisis será realizado de manera muy superficial de manera que pueda ser entendido por el lector sin mayor conocimiento de la biología, una mayor profundización a esto será considerado trabajo futuro entre el profesor mencionado y el autor de esta memoria, y publicado en un escrito (*paper*).

Recordando un poco, la estructura de un aminoácido sigue el siguiente formato:

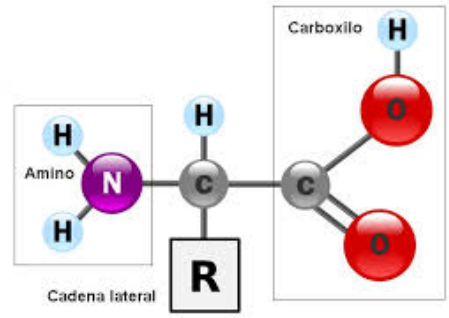


Figure 4.3: Estructura de un aminoácido, donde se identifica el grupo amino (izquierda), grupo carboxilo (derecha) y el grupo radical (R)

Con respecto a la frecuencia de los residuos de aminoácidos encontrados para la base de datos de UniProt (SwissProt, TrEMBL y Homosapiens), los aminoácidos L (Leucina), A (Alanina), G (Glicina), V (Valina) y E (Ácido Glutámico) son los más comunes debido a sus características físico-químicas, en especial para L, A y V, que son **residuos hidrofóbicos** (L es el residuo más hidrofóbico). Cuando se habla de aminoácidos hidrofóbicos [39], su característica principal es que tienden a rechazar el ambiente acuoso (agua) y, por lo tanto, residen predominante dentro de las proteínas. Estos residuos se utilizan para la estabilización de la estructura global de muchas proteínas (en especial aquellas con más de 50 aminoácidos).
Por el contrario, los aminoácidos hidrofílicos tienden a interactuar con el ambiente acuoso, se encuentra ubicados en los sectores extremos de una cadena de polipéptidos y están implicados a menudo en la formación de enlaces de H para la unión entre aminoácidos.

En el caso de los "oligopéptidos reguladores" (base de datos de EROP-Moscow) los aminoácidos G, L, K (Lisina), S (Serina) y C (Cisteína) son los residuos más comunes. Estos oligopéptidos [40] son moléculas flexibles que cambian su configuración física, poseen pocas interacciones moleculares y muy débiles (pocos

residuos hidrofóbicos), tienen fuertes enlaces covalentes (enlaces entre elementos no metálicos de la Tabla Periódica de Elementos Químicos) intramoleculares entre los residuos C-C y participan en variadas interacciones electroestáticas.

En relación a las repeticiones encontradas de un solo aminoácido, muchas de los animoácidos pueden tener carácterísticas ácidas (pH bajo) como N (Aspargarina) o Q (Glutamina) los cuales buscan enlazarse con sus pares ya que poseen los grupos amino y carboxilo polarizados a nivel de carga eléctrica, y de esa forma producir un enlace entre aminoácidos.

Conclusiones

Las proteínas constituyen una parte fundamental de la biología actual, son partícipes de numerosas funciones vitales en los seres vivos, al día de hoy se siguen investigando sobre nuevos polipéptidos que van apareciendo y también se siguen buscando cuáles son los aminoácidos y/o secuencias responsables directamente de estas funciones vitales. Estas proteínas han sido registradas y guardadas en archivos .fasta como cadenas de strings, y pueden ser descargadas de manera gratuita en las páginas encargadas, como UniProt y EROP-Moscow.

Lo que se puso como objetivo en este trabajo es desarrollar en base a estructuras de indexación (Indexed Text Searching) una manera entendible y conveniente para obtener los diferentes residuos de aminoácidos de cantidades que oscilaron entre 1 hasta 50 aminoácidos, y a partir de aquellos extraer las secuencias que más se repetían. Y mirando los resultados logrados en la sección anterior, se puede concluir que el uso del **arreglo de sufijos**, **arreglo LCP** y ***priority queue*** para este trabajo permitió lograr con éxito esta tarea.

Problemas presentados en el desarrollo de la implementación

Aunque en el desarrollo de la implementación, aparecieron variados problemas a la hora de trabajar ciertos aspectos. Primero fue el hecho **de sustraer las cadenas de proteínas de los archivos .fasta y alojarlas en otro archivo encadenándolas en un solo string**, donde el tiempo empleado para realizar esto dependía del tamaño del archivo analizado.
Un segundo problema fue que la cadena de strings obtenidas del archivo de UniProt-TrEMBL era muy grande para formar el vector correspondiente al arreglo de sufijos, lo cual originó **la búsqueda de una nueva alternativa de implementación** de esta estructura en conjunto con su arreglo LCP (algoritmos en memoria externa). Este hecho permitió comparar resultados logrados con los *datasets* usados y de esa manera usar esta "comparación" como ejemplo de veracidad de los resultados obtenidos, en especial para la sección de los residuos que más se repiten.
El tercer problema fue el hecho de identificar en los archivos **cuáles eran los caracteres prohibidos que no se debían tomar en cuenta** para desarrollar este problema. Ante eso las bases de datos pertenecientes a UniProt (SwissProt-TrEMBL-Homosapiens) tenían diferentes letras prohibidas que la base de datos de EROP-Moscow (este archivo tenía cadenas de oligopéptidos en los que se encontraban cadenas de radicales polarizados en los

extremos colocados con los signos + (positivo) y − (negativo)). Estos detalles hicieron modificar ciertas líneas del archivo principal agregando más letras prohibidas a revisar. En consecuencia supuso un aumento (que no fue significativo) en los tiempos al obtener los resultados finales.
Aparte de estas cosas lo demás fue investigar y estudiar nuevos conceptos para el autor como los arreglos de sufijos y derivados, probar con las implementaciones y saber cómo usar el servidor prestado por la Universidad.

Extensión del algoritmo a otros temas

Si bien el algoritmo fue desarrollado con el objetivo de hallar los diferentes substrings de secuencias de proteínas y sus repeticiones, también se puede tomar y modificar este algoritmo para trabajar en:

1. Cadenas de ADN (diferentes substrings de tamaño k y cuáles son los que más se repiten).
2. Todo tipo de textos, ya sean científicos, literarios, de entretención, etc. (buscar cantidad de vocales y/o trozos diferentes de palabras de determinado tamaño que hay en un determinado texto).

Con respecto a la opción de cadenas de ADN, existen variados genomas disponibles a descargar los cuales también se encuentran en formato .fasta, por lo mismo manipularlos a la hora de extraer la cadena tendría el mismo procedimiento que la obtención de la cadena de proteínas, incluso es más sencilla ya que solamente se trabajaría con una cadena genérica de ADN, no habría concatenación con otras cadenas, por otra parte acá se tendrían 4 nucleótidos a revisar y la combinatoria de potenciales substrings de tamaño k a encontrar sería mucho más pequeña.
Por el lado de los textos sería más difícil extraer una cadena de palabras, se tendrían que eliminar los espacios o considerarlos como carácteres prohibidos a igual que los elementos paraverbales, como los signos de interrogación, signos de exclamación, puntos, etc. Por otro lado el universo de letras a considerar serían 27 (las letras del abecedario), de manera que la combinatoria de posibles secuencias de palabras sería bastante más grande que con las proteínas. Lo positivo de trabajar con estos textos es que se utilizan **caracteres ordenados en "palabras"** por lo cual aplicando el algoritmo a estos textos y obteniendo resultados permitiría estudiar el tipo de lenguaje o expresiones que usa un determinado autor para escribir su obra.

Posible trabajo a futuro

La solución entregada en esta memoria con respecto al problema analizado de ninguna manera puede ser catalogado como definitivo, obviamente es una opción muy viable para trabajar pero podrían haber mejores propuestas de implementaciones. El real potencial de esto es el **uso del arreglo de sufijos en una aplicación** demostrando que su utilización es muy importante para trabajar cadenas que para este caso fueron las proteínas. En el caso de los tiempos logrados pueden ser más pequeños si se utilizará un servidor más potente (cosa díficil de lograr si no se tienen los recursos para obtenerlo, por otro lado el servidor de la Universidad es bastante

completo a nivel de velocidad y memoria RAM) o mejorar ciertos pasos de la implementación, por ejemplo cuando **se pasan los datos de los archivos .sa y .lcp a los archivos manipulables .txt para usarlos en el programa principal**. Sería bueno intentar extraer toda esa información sin crear un archivo auxiliar, lo que no solo impondría una mejora notable en los tiempos, sino que también se ocuparía mucho menos espacio en el disco duro.

En lo relativo a investigación, el autor de esta memoria está actualmente en contacto con el profesor ruso Alexander Zamyatnin para trabajar en la publicación en *BIOjournals* de un *paper* en el cual se obtendrán los diferentes substrings de tamaño k y aquellos que más se repiten de ciertos organismos (animales, plantas, bacterias, entre otros) extraídos de las bases de datos de UniProt (SwissProt y TrEMBL) y EROP-Moscow, de los cuáles se realizarán interpretaciones biológicas y fisicoquímicas de los resultados obtenidos lo que permitirá de entrar con más detalles al tema de la Bioinformática y relacionados.

En definitiva siempre será relevante trabajar con proteínas, a cada día, a cada minuto se va descubriendo una nueva proteína, una nueva función correspondiente a un determinado aminoácido, posiblemente se lleguen a descubrir cientos de péptidos en el futuro ya que aún quedan muchas cosas por resolver con respecto a las propiedades de estas estructuras.

Y con respecto a las estructuras de indexación, sus avances y motivación de trabajo han ido a una velocidad tan rápida que a casi 20 años de su creación ya ocupan un sector importante para analizar cadenas de strings. Esto ha ido de la mano directamente con el crecimiento de las tecnologías de información (TI) lo que permitirá en unos años más a seguir optimizando el uso de estas estructuras.

Apéndice

Algorithm 1 Knuth-Morris-Pratt

```
1:  n = length[T];
2:  m = length[P];
3:  F = compute_prefix_function(P);
4:  q = 0;
5:  for i = 1 to n do
6:      while q > 0 and P[q + 1] ≠ T[i] do
7:          q = F[q]
8:      end while
9:      if P[q + 1] == T[i] then
10:         q = q + 1
11:     end if
12:     if q == m then
13:         print "patron ocurre con shift" i − m
14:         q = F[q]
15:     end if
16: end for
```

Algorithm 2 Boyer-Moore

Require: String T (texto de largo n caracteres) y P (patron de largo m caracteres)

Ensure: Indicacion de que si P es un substring de T, o si P no es un substring de T

```
1:  i = m − 1
2:  j = m − 1
3:  repeat
4:      if P[j] == T[i] then
5:          if j == 0 then
6:              return i
7:          else
8:              i = i − 1
9:              j = j − 1
10:         end if
11:     else
12:         i = i + m − j − 1
13:         i = i + max(j − last(T[i]), match(j))
14:         j = m − 1
15:     end if
16: until i > n − 1
17: return "No hay substring en T que sea igual a P"
```

Algorithm 3 Arreglo de sufijos - SA

Require: String A (texto de largo n caracteres)

1: $n = length(A)$
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $S_{0,1}$ = posición de A_i en el arreglo ordenado de las letras de A
4: **end for**
5: $const = 1$
6: **if** $const < 2n$ **then**
7: **for** $k = 1$ **to** n **do**
8: **for** $i = 0$ **to** $n - 1$ **do**
9: $L_i = (S_{k-1,i}, S_{k-1,i+const}, i)$
10: **end for**
11: sort **L**
12: compute $S_{0,1}, i = 0, n - 1$
13: $const = 2 * const$
14: **end for**
15: **end if**

Algorithm 4 Arreglo LCP - Longest Common Prefix

Require: String A de n caracteres, arreglo de sufijos S de largo n y su inverso S^{-1}

Ensure: Arreglo LCP de tamaño $n - 1$

```
1:  l = 0
2:  for i = 0 to n − 1 do
3:      k = S^{-1}[i] // i = S[i]
4:      j = S[k − 1]
5:      while A[i + l] = A[j + l] do
6:          l = l + 1
7:      end while
8:      LCP[k] = l
9:      if l > 0 then
10:         l = l − 1
11:     end if
12: end for
13: return LCP
```

Bibliography

[1] Rafael Lahoz-Beltrá. *BIOINFORMÁTICA, simulación, vida artificial e inteligencia artificial*. Ediciones Díaz de Santos S.A., Madrid, España, 2004.

[2] Bruce C. Orcutt, Winona C. Barker. *Searching the protein sequence database*. National Biomedical Research Foundation, Georgetown University Medical Center, Washington, D.C., *Bulletin of Mathematical Biology*, 46(4):545-552, 1984.

[3] Alexander A. Zamyatnin. *The Features of an Array of Natural Oligopeptides*. Bakh Institute of Biochemistry, Russian Academy of Sciences, Moskow, Russia; Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile, *Neurochemical Journal*, 10(4):249-257, 2016.

[4] Wikipedia, Definición de biomolécula.
`https://es.wikipedia.org/wiki/Biomolécula`.

[5] Jan Koolman, Klaus-Heinrich Röhm. *Bioquímica: Texto y Atlas*. Editorial Médica Panamericana, Madrid, España; Georg Thieme Verlag, Stuttgart, Germany, 2004.

[6] Alexander A. Zamyatnin. *Fragmentomics of Natural Peptide Structures*. Bach Institute of Biochemistry, Russian Academy of Sciences; Universidad Técnica Federico Santa María, Departamento de Informática, Valparaíso, 2009, pp. 405-428.

[7] Alexander A. Zamyatnin. *Fragmentomics of Oligopeptides and Proteins*. Bach Institute of Biochemistry, Russian Academy of Sciences; Universidad Técnica Federico Santa María, Departamento de Informática, Valparaíso, 2nd Asia-Pacific International Peptide Symposium, 2007.

[8] Búsqueda secuencial de texto, Algoritmo Fuerza Bruta.
`https://sites.google.com/site/busquedasecuencialdetexto/home`.

[9] Gonzalo Navarro, Mathieu Raffinot. *Flexible Pattern Matching Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Universidad de Chile; Centre Nationale de Recherche Scientifique, Marne-La-Vallee, Francia, 2002.

[10] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. *Fast pattern matching in strings*. *SIAM Journal on Computing*, 6(2):323-350, 1977.

[11] Boyer, Robert S. and Moore, J. Strother. *A Fast String Searching Algorithm*. *Communications of the ACM*, Nueva York, Estados Unidos, 20(10):762-772, 1977.

[12] Pekka Kilpelanen. *Lecture 8: Applications of Suffix Trees*. University of Kuopio, Finland; Department of Computer Science, Spring 2005, pp. 1-20.

[13] *Introducción a la biología computacional, árboles de sufijos*. `http://webdiis.unizar.es/asignaturas/TAP/material/4.2.sufijos.pdf`, pp. 1-16.

[14] Giulio Pavesi, Giancarlo Mauri, Graziano Pesole. *An algorithm for finding signals of unknown length in DNA sequences*. Department of Computer Science, Systems and Communication, University of Milan-Bicocca, Via Bicocca degli Ancimboldi; Department of General Phisiology and Biochemistry, University of Milan, Via Celoria, April 3rd, 2001, 17:207-214.

[15] Manber, Udi and W. Myers, Eugene. *Suffix Arrays: A New Method for On-Line String Searches*. San Francisco, California, USA, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 319-327, 1990.

[16] Alejandro Deymonnaz. *Arreglos de sufijos para alineamiento de secuencias de ADN con memoria acotada*. Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación, Argentina, 2012.

[17] Pang Ko, Srinivas Aluru. *Suffix Tree Applications in Computational Biology*. Iowa State University, *Handbook of Computational Molecular Biology*, Chapter 6, 2001.

[18] Andrés Abeliuk Kimmerman. *Árboles de sufijos comprimidos para textos altamente repetitivos*. Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación, Chile, 2012.

[19] Kurtz, Stefan. *Reducing the space requirement of suffix trees*. *Software: Practice and Experience*, 29(13):1149-1171, 1999.

[20] Adrian Vladu y Cosmin Negruşeri. *Suffix arrays - a programming contest approach*. Noviembre 2005.

[21] Arreglo LCP, *LCP array*. `https://en.wikipedia.org/wiki/LCP_array`.

[22] Kasai, T.; Lee, G.; Arimura, H.; Arikawa, S.; Park, K. *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*. *Combinatorial Pattern Matching: 12th Annual Symposium*, 2089: 181–192, 2001.

[23] The Twenty Amino Acids. `https://www.cryst.bbk.ac.uk/education/AminoAcid/the_twenty.html`.

[24] Manipulating Strings.
http://teacher.buet.ac.bd/faizulbari/cse201/Strings.pdf.

[25] Kasai's Algorithm for Construction of LCP array from Suffix Array.
http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/.

[26] Juan Soulié. *C++ Language Tutorial*. Junio 2007.

[27] *Chapter 11: Priority Queues and Heaps*.
https://web.engr.oregonstate.edu/~sinisa/courses/OSU/CS261/CS261_Textbook/Chapter11.pdf.

[28] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, Peter Sanders. *Better external memory suffix array construction*. *Journal of Experimental Algorithmics*, Nueva York, Estados Unidos, 12: 1–24, 2008.

[29] Timo Bingmann, Johannes Fischer, Vitaly Osipov. *Inducing Suffix and LCP Arrays in External Memory*. KIT, Institute of Theoretical Informatics, Karlsruhe, Alemania. *Journal of Experimental Algorithmics*, Nueva York, Estados Unidos, 21: 1–15, 2016.

[30] Juha Kärkkäinen, Dominik Kempa. *Engineering a Lightweight External Memory Suffix Array Construction Algorithm*. Department of Computer Science, University of Helsinki, Finlandia. *Mathematics in Computer Science*, 11(2): 137-149, 2017.

[31] Gastón Gonnet, Ricardo Baeza-Yates, Tim Snider. *New indices for text: PAT Trees and PAT arrays*. *Information Retrieval: Algorithms and Data Structures*, Prentice-Hall, Englewood Cliffs, 66-82, 1992.

[32] Michael Burrows y David Wheeler. *A block sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[33] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L. Salzberg. *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome*. Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA; Genome Biology, March 2009.

[34] Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[35] Juha Kärkkäinen, Dominik Kempa. *LCP array construction in external memory*. Department of Computer Science, University of Helsinki, Finlandia. *Proceedings of the* 13^{th} *International Symposium on Experimental Algorithms (SEA 2014)*. 8504: 412–423, Springer, 2014.

[36] Juha Kärkkäinen, Dominik Kempa. *Faster External Memory LCP Array Construction*. Department of Computer Science, University of Helsinki, Finlandia. *Proceedings of the* 24^{th} *Annual European Symposium on Algorithms (ESA 2016)*. 57: 61:1–61:16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[37] Lawrence Goetz, Yedidyah Langsam, Theodore Raphan. *Code::Blocks Student Manual*. Department of Computer and Information Science, Brooklyn College of CUNY. Version 16.01, 2016.

[38] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Departamento de Ciencias de la Computación, Universidad de Chile. *Cambridge University Press*, Nueva York, Estados Unidos, 2016.

[39] Naturaleza Química de los Aminoácidos.
`https://themedicalbiochemistrypage.org/es/amino-acids-sp.php`.

[40] Alexander A. Zamyatnin. *Hemoglobin as a potential source of natural regulatory oligopeptides*. Bach Institute of Biochemistry, Russian Academy of Sciences. *Biochemistry (Moscow)*, 74(2): 201-208, 2009.

[41] UniProt-SwissProt database.
`http://www.uniprot.org/uniprot/?query=reviewed:yes`.

[42] UniProt-TrEMBL database.
`http://www.uniprot.org/uniprot/?query=reviewed:no`.

[43] EROP-Moscow: Endogenous Regulatory OligoPeptide knowledgebase.
`http://erop.inbi.ras.ru/`.

[44] Wikipedia, Definición de archivo FASTA.
`https://es.wikipedia.org/wiki/Formato_FASTA`.