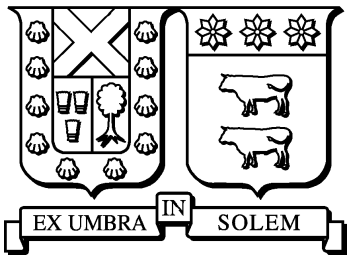
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE




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

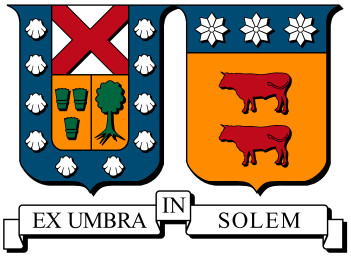
FELIPE NICOLÁS ARAYA BARRERA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: LIOUBOV DOMBROVSKAIA

OCTUBRE 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**

**OCTUBRE 2017**

# Agradecimientos

Se escribirán los agradecimientos una vez este documento haya sido terminado.

# Dedicatoria

Se escribirá la dedicatoria una vez este documento haya sido terminado.

# Resumen

Se completará este capítulo una vez que el cuerpo de este trabajo haya finalizado.

# Abstract

Same thing as the previous section.

# Índice de Contenidos

**Agradecimientos** III

**Dedicatoria** IV

**Resumen** V

**Abstract** VI

**Índice de Contenidos** VII

**Lista de Tablas** X

**Índice de cuadros** X

**Lista de Figuras** XII

**Índice de figuras** XII

**Glosario** XIV

**Introducción** 1

**1. Definición del Problema** 3

1.1. Definición . . . . . 3

1.2. Objetivos . . . . . 4

**2. Estado del Arte** **5**

2.1. Información previa a considerar . . . . . 5

2.1.1. Biomoléculas . . . . . 5

2.1.2. Aminoácidos . . . . . 7

2.2. Secuencias de proteínas . . . . . 9

2.3. Base de datos UniProt . . . . . 13

2.3.1. UniProt - SwissProt . . . . . 14

2.3.2. UniProt - TrEMBL . . . . . 14

2.4. Base de datos EROP-Moscow . . . . . 14

2.5. Técnicas utilizadas en el problema . . . . . 15

2.5.1. Algoritmo de fuerza bruta . . . . . 15

2.5.2. Algoritmos de búsqueda de strings . . . . . 17

2.5.3. Estructuras de datos para texto . . . . . 21

2.5.4. Árbol de sufijos . . . . . 22

2.5.5. Arreglo de sufijos . . . . . 24

2.5.6. Arreglo LCP . . . . . 26

2.5.7. External Memory Suffix Array . . . . . 27

2.5.8. External Memory LCP Array . . . . . 30

2.5.9. Transformación de Burrows-Wheeler . . . . . 30

**3. Implementación** **31**

3.1. Propuesta considerada . . . . . 31

3.1.1. Restando la cantidad máxima posible de diferentes substrings . . . 33

3.1.2. Aumentando la cantidad de diferentes substrings desde 0 considerando determinados tamaños de LCPs consecutivos . . . . . 35

3.2. Cola de prioridad *(priority queue)* . . . . . 37
3.2.1. Algunos comandos de C++ para el *priority queue* . . . . . 38
3.3. Restricciones para la propuesta . . . . . 39
3.4. Algoritmo desarrollado . . . . . 40
3.4.1. Extracción de proteínas desde el archivo .fasta . . . . . 40
3.4.2. Implementación solución para base de datos UniProt-SwissProt . . 43
3.4.3. Implementación solución para base de datos UniProt-TrEMBL . . . 56

**4. Análisis de resultados** **57**
4.1. Resultados obtenidos . . . . . 57
4.2. Análisis detallado . . . . . 57
4.2.1. Análisis matemático . . . . . 57
4.2.2. Análisis biológico . . . . . 57

**Conclusiones** **58**

**Bibliografía** **63**

# Índice de cuadros


2.1. Los 20 aminoácidos existentes y considerados . . . . . 8

2.2. Identificación de macromoléculas según cantidad de aminoácidos . . . . . 9

2.3. Combinaciones posibles a obtener según el tamaño del péptido . . . . . 10

2.4. Número máximo posible de fragmentos que se pueden formar en 5 proteínas 11

2.5. Número máximo posible de fragmentos que se pueden obtener en una proteína de caseína bovina. . . . . 12

2.6. Ejemplo de uso del algoritmo de Knuth-Morris-Pratt . . . . . 18

2.7. Ejemplo de uso del algoritmo de Boyer-Moore . . . . . 19

2.8. Sufijos e índices ordenados según orden alfabético (sector derecho). . . . . 22

2.9. Arreglo de sufijos $A[i]$ de la palabra MISSISSIPPI$ . . . . . 24

2.10. SA de MISSISSIPPI$ y sus sufijos respectivos . . . . . 27

2.11. Arreglo LCP de la palabra MISSISSIPPI$ . . . . . 27

3.1. SA y arreglo LCP de la palabra BANANA$ . . . . . 33

3.2. Arreglo LCP (tabla izquierda) mueve su primer elemento (0) hacia la última posición (tabla derecha). . . . . 35

3.3. SA y arreglo LCP modificado de la palabra BANANA$ . . . . . 35

3.4. 2 secuencias enlazadas en una cadena general utilizando el signo $ como unión. 39

3.5. Ejemplo de cadena encontrada en el archivo destino . . . . . 42

3.6. Formatos de los arreglos SA y LCP . . . . . 49

# Índice de figuras

1.1. Secuencias de varias proteínas en formato **.fasta** . . . . . 4

2.1. Estructura química de los carbohidratos y lípidos. . . . . 6

2.2. Biomoléculas de ADN y péptidos llevadas a cadenas de strings. . . . . 7

2.3. Estructura general de un alfa-aminoácido . . . . . 8

2.4. Página principal del sitio web de UniProt. . . . . 13

2.5. Página principal del sitio web de EROP-Moscow. . . . . 15

2.6. Ejemplos de árboles de sufijos. . . . . 23

2.7. El árbol de sufijo generalizado para las secuencias *GATCG$* y *CTTCG$* [17] 24

2.8. Arreglo de sufijos de un texto $T$ formado por la combinación de arreglos de sufijos de los bloques del texto $T$. . . . . 28

2.9. Combinación de 2 arreglos de sufijos que fueron creados de los fragmentos de texto $Y$ y $Z$. . . . . 29

2.10. Ejemplo de la transformación de Burrows-Wheeler para la cadena acaacg$. 30

3.1. Grafo de muestra del formato de un *priority queue*. . . . . 37

3.2. El *priority queue* con el número 35 extraído. . . . . 38

3.3. El *priority queue* con el número 15 agregado. . . . . 38

3.4. Archivo **.fasta** por defecto con varias proteínas . . . . . 40

3.5. Caso 1 . . . . . 53

3.6. Caso 2 . . . . . 54

3.7. Caso 3 . . . . . 54

3.8. Caso 4 . . . . . 55

3.9. Extracto del archivo de salida “resultados_k1to50.txt” . . . . . 56

# Glosario

# 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.

Hablando de las proteínas, 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-Moskow.

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, que

sigue la regla de combinatoria $20^k$ ya que son 20 los animoácidos base que existen en la actualidad (para dipéptidos serían $20^2 = 400$, para tripéptidos serían $20^3 = 8000$ y así sucesivamente).

# Capítulo 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 base de datos (como UniProt, Genbank) 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
```

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

Una de las principales tareas en la investigación de proteínas consiste es buscar residuos de aminoácidos (secuencias 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 con 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 base de datos, 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 el número máximo de fragmentos de péptidos (AAR) que existen asociado a un valor $k$ determinado, donde $k$ se ubicará en el intervalo entre 2 hasta 50, y en base a aquello obtener y determinar para cada $k$ cuáles son los fragmentos de aminóácidos que más se repiten para posteriormente realizar un análisis tanto matemático y biológico de los resultados obtenidos. Para realizar esta tarea se usará como *dataset* la base de datos de proteínas de SwissProt (550100 proteínas) y de TrEMBL (88032926 proteínas).

A partir de esto las implementaciones de código para trabajar estos archivos serán realizados en los lenguajes de programación **C++** y **Python**, con diferentes finalidades que serán explicados a medida que se avance en el documento.

# Capítulo 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

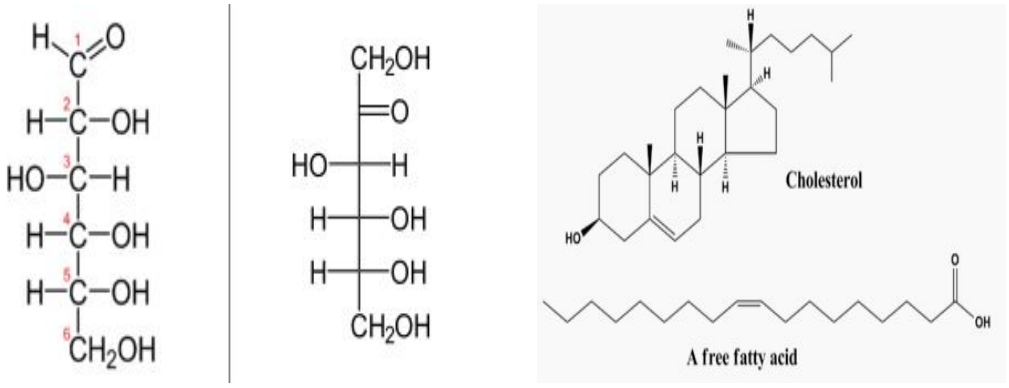
utilizan los seres vivos para realizar sus funciones vitales. Los ejemplos más conocidos son la glucosa, el almidón y el glucógeno.

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.

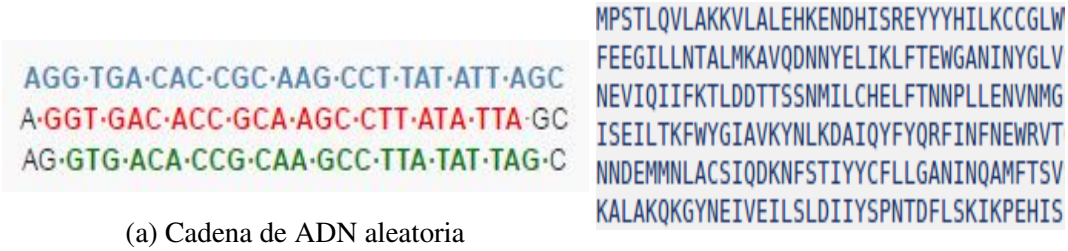
Dentro de estas biomoléculas, el análisis detallado de los carbohidratos y los lípidos depende en demasía de su estructura química (elementos químicos asociados y tipo de enlaces entre ellos), por lo mismo es una materia más ligada a los químicos (ver Figura 1.1).




(a) Glúcidos (glucosa y fructosa) (b) Lípidos (colesterol y un ácido graso)

Figura 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

Figura 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:

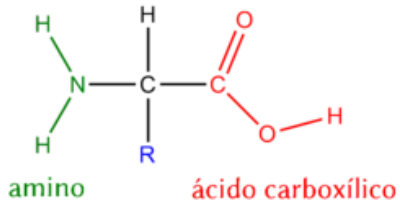



Figura 2.3: Estructura general de un alfa-aminoácido

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:

| 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 |

Cuadro 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 |

Cuadro 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. | $1.024\times10^{13}$ |
| 20 aa. | $1.049\times10^{26}$ |
| 50 aa. | $1.126\times10^{65}$ |

Cuadro 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 (incluye posibles repeticiones) que teóricamente se pueden obtener para una molécula de tamaño $n$, partiendo desde $k = 2$ (dipéptidos) hasta $k = n - 1$, viene dado por:

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

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 $\alpha$ hemoglobina (humanos) | 141 | 9869 |
| 5 | Cadena $\beta$ hemoglobina (humanos) | 146 | 10584 |

Cuadro 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:

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

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, $\alpha - s1$, $\alpha - s2$, $\beta$ y $\kappa$. La siguiente tabla muestra las cantidades teóricas y diferentes de fragmentos obtenidos como dipéptidos y sus sumas totales:

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

Cuadro 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.

Con respecto a las bases de datos de proteínas que se utilizarán, estas corresponderán a UniProt y EROP-Moscow.

## 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.

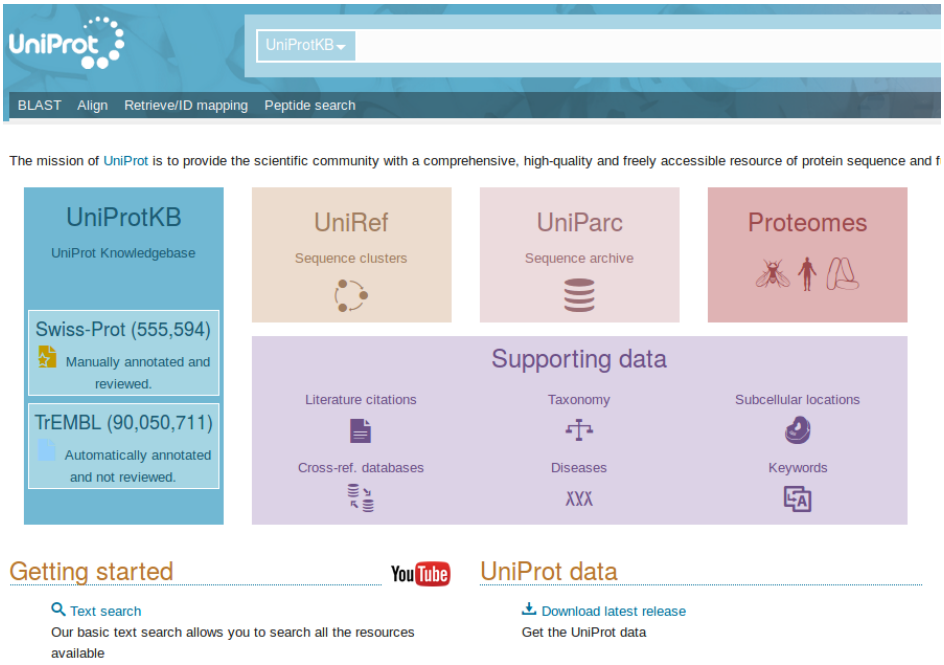



Figura 2.4: Página principal del sitio web de UniProt.

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:

### 2.3.1. UniProt - SwissProt

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.

### 2.3.2. UniProt - TrEMBL

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.

## 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.

Actualmente está compuesta por 14.785 secuencias que equivalen a un texto con peso de 1.4 MB. Aunque son bastante pocas secuencias en función de UniProt, estos péptidos están descritos de manera muy completa y son ideales para realizar diferentes tipos de análisis, ya sea por propiedades físico-químicas, tipo de órgano en el cual se encuentran, entre otros.


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

## 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 conveniente considerando que la cantidad de proteínas que existen el día de hoy son muchas, por consiguiente una herramienta recomendable sería preprocesar 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;
```

Listing 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 en 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 implmentación, 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 $k = 20$). 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 cuáles son los 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 ($\Sigma$) 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:

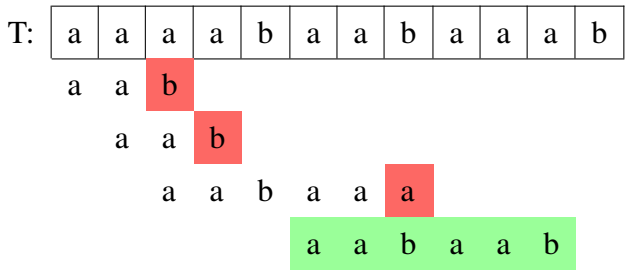
1. Algoritmos de búsqueda de un único patrón (*Single pattern algorithms*)
2. Algoritmos de búsqueda de múltiples patrones (*Multiple pattern algorithm*)

### 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] $p = p_1 p_2 \dots p_m$ en un texto largo $T = T_1 T_2 \dots T_n$, donde $p$ y $T$ son secuencias de caracteres que provienen de set finito de caracteres $\Sigma$. 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”.




Cuadro 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 fue 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 | |

Cuadro 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 $\Sigma$ [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 tiempo y 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. 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, 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$ |

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

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.

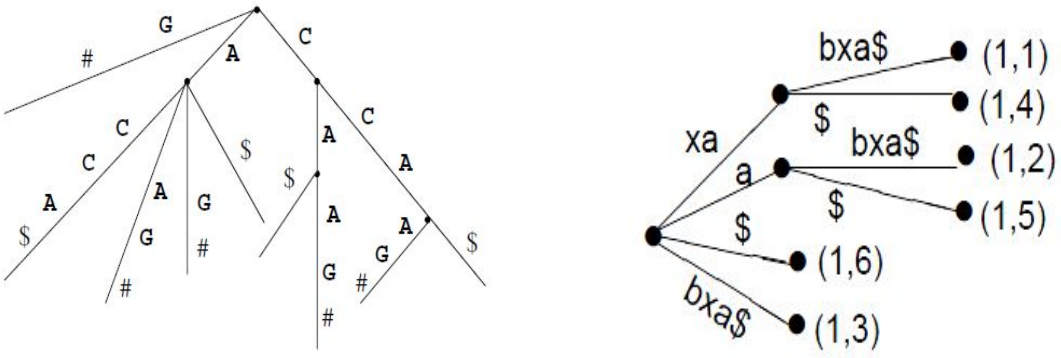
### 2.5.4. Árbol de sufijos

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 $\tau$ 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*

Figura 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*:

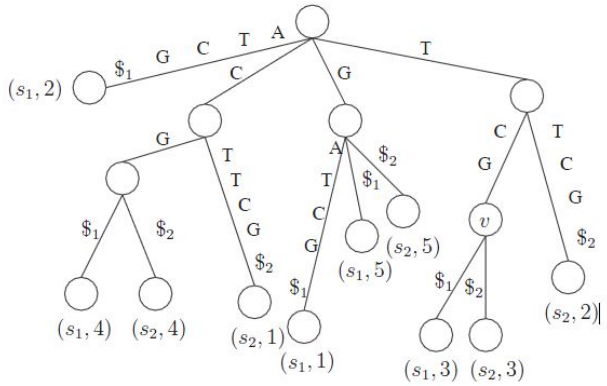



Figura 2.7: El árbol de sufijo generalizado para las secuencias *GATCG$* y *CTTCG$* [17]

### 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 |

Cuadro 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 $O(n)$, 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).

| **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** | | | | | | $ | | | | | | |

Cuadro 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 |

Cuadro 2.11: Arreglo LCP de la palabra MISSISSIPPI$

### 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 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:

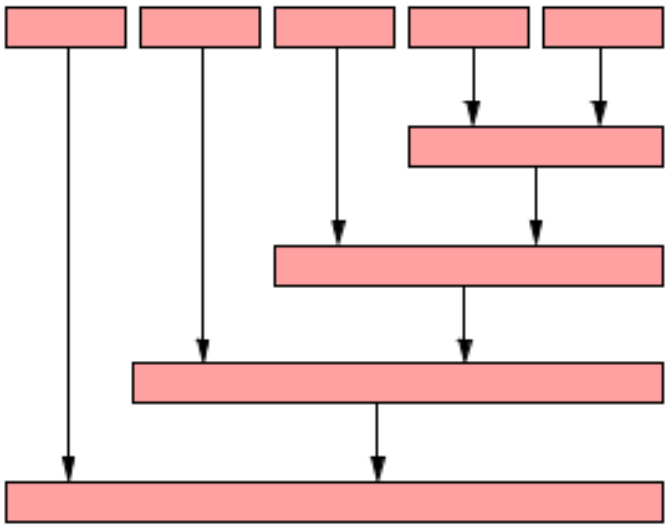


Figura 2.8: Arreglo de sufijos de un texto $T$ formado por la combinación de arreglos de sufijos de los bloques del texto $T$.

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

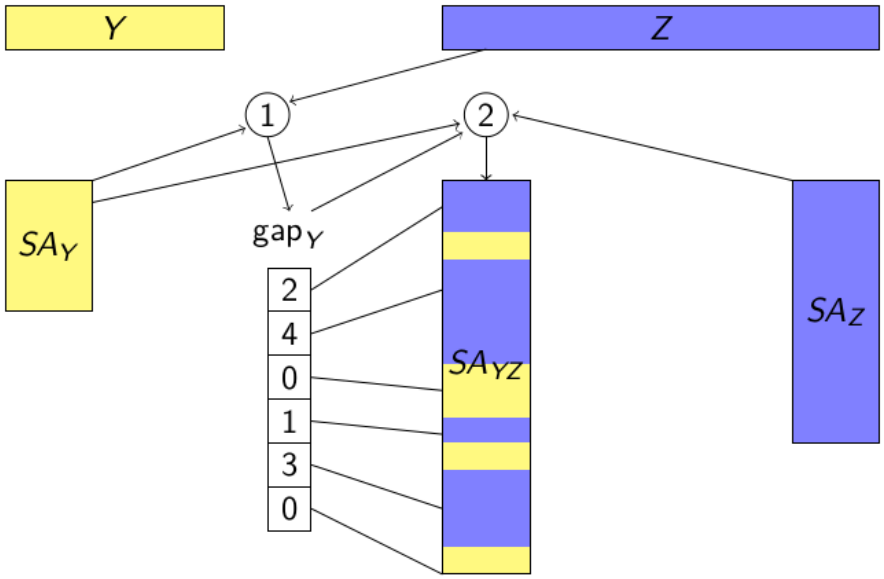



Figura 2.9: Combinación de 2 arreglos de sufijos que fueron creados de los fragmentos de texto $Y$ y $Z$.

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

### 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:

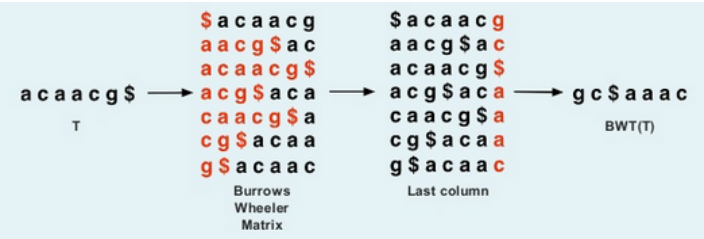



Figura 2.10: Ejemplo de la transformación de Burrows-Wheeler para la cadena acaacg$.

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).

# Capítulo 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 y UniProt-TrEMBL.
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$ |

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$')
   $var = var + 3 - 0 \Rightarrow 16 + 3 = 19$

6. ('NA$','NANA$'): *var*+ = *length*('NANA$') - *LCP*('NA$','NANA$')
   $var = var + 5 - 2 \Rightarrow 19 + 3 = 22$

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**:

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

Cuadro 3.1: 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<br>Substrings totales: 6<br>Elementos LCP ≥ 2: 2<br>DS de tamaño 2: 6 − 2 = 4 | Tamaño 3<br>Substrings totales: 5<br>Elementos LCP ≥ 3: 1<br>DS de tamaño 3: 5 − 1 = 4 | Tamaño 4<br>Substrings totales: 4<br>Elementos LCP ≥ 4: 0<br>DS de tamaño 4: 4 − 0 = 4 |
| Tamaño 5<br>Substrings totales: 3<br>Elementos LCP ≥ 5: 0<br>DS de tamaño 5: 3 − 0 = 3 | Tamaño 6<br>Substrings totales: 2<br>Elementos LCP ≥ 6: 0<br>DS de tamaño 6: 2 − 0 = 2 | Tamaño 7<br>Substrings totales: 1<br>Elementos LCP ≥ 7: 0<br>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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Cuadro 3.2: 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$ |

Cuadro 3.3: 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.

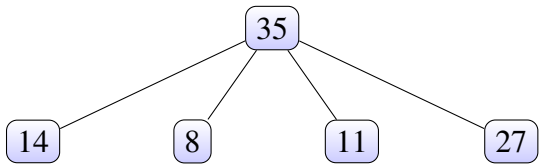



Figura 3.1: Grafo de muestra del formato de un *priority queue*.

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:

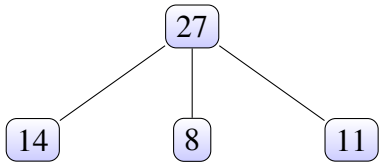



Figura 3.2: El *priority queue* con el número 35 extraído.

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:

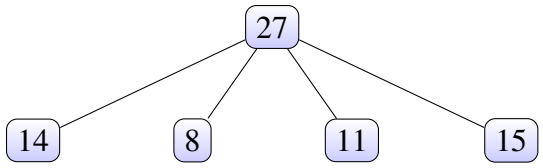



Figura 3.3: El *priority queue* con el número 15 agregado.

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...

Cuadro 3.4: 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, detalle 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 "uniprot_trembl.fasta" está compuesto por 88032926 proteínas con un peso de 40 GB. Para ambos archivos la construcción del arreglo de sufijos y el arreglo LCP serán diferentes pero recibirán la misma cadena enlazada, cuya construcción será explicada en la siguiente sección.

### 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
```

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

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

```
1 ifstream fin("uniprot_sprot.fasta"); //abrir archivo base
```

```
2  if(!fin){
3          cerr << "Couldn't open the input file!";
4          return(1);
5  }
6  ofstream outputfile; //crear archivo destino de cadena
7  outputfile.open("substrings.txt"); //definir nombre de archivo a crear
8  string line;
9  int cantidad_ss = 1;
10
11 getline(fin, line);
12 getline(fin, line); //tomar secuencia (1)
13
14 while(fin){
15         if(line[0] == '>'){ // revisar primer elemento del string (2)
16                 outputfile << "$";
17                 cantidad_ss = cantidad_ss + 1;
18         }
19         else{
20                 outputfile << line;
21         }
22 getline(fin, line); // continuar con la siguiente linea (3)
23 }
24
25 outputfile.close();
```

Listing 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

Cuadro 3.5: 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 esta se almacena en la memoria RAM del ordenador. En primera instancia no hay problema con la base de datos de SwissProt 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

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
10 int cmp(struct suffix a, struct suffix b)
11 {
12         return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
13                 (a.rank[0] < b.rank[0] ?1: 0);
14 }
```

Listing 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 continua 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:

```
1  vector<int> buildSuffixArray(string txt, int n){
2
3          struct suffix *suffixes = new struct suffix [n]; //(1)
4          for (int i = 0; i < n; i++){
5                  suffixes[i].index = i;
6                  suffixes[i].rank[0] = txt[i] - 'a';
7                  suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
8          }
9          sort(suffixes, suffixes+n, cmp);
10         int *ind = new int [n]; //(2)
11         for (int k = 4; k < 2*n; k = k*2){
12                 int rank = 0;
13                 int prev_rank = suffixes[0].rank[0];
14                 suffixes[0].rank[0] = rank;
15                 ind[suffixes[0].index] = 0;//(3)
16                 for (int i = 1; i < n; i++){
17                         if (suffixes[i].rank[0] == prev_rank &&
18                           suffixes[i].rank[1] == suffixes[i-1].rank[1]){ \\(a)
19                                 prev_rank = suffixes[i].rank[0];
20                                 suffixes[i].rank[0] = rank;
21                         }
22                         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  }
```

Listing 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, 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 }
```

Listing 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 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)
18 resultados.open("resultados_sa_lcp.txt");
19 resultados << "Construccion␣SA:␣" << t12 << "␣segundos" << endl;
20 resultados << "␣\n";
21 resultados << "Construccion␣LCP:␣" << t23 << "␣segundos" << endl;
22 resultados << "␣\n";
23 resultados << "Total:␣" << cantidad_ss << "␣proteinas" << endl; \\(5)
24 resultados.close();
```

Listing 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 | ... |

Cuadro 3.6: 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:

```
1  int inicio = cantidad_ss - 1;
2  int n = suffixArr.size(); \\(1)
3  k_utilizados.open("resultados_k1to50.txt");
4  for (int k1 = 1; k1 < 51; k1++){
5      t4 = clock();
6      subcadena arr[1];
7      priority_queue<subcadena, vector<subcadena>, comparador> mypq; \\(2)
8      int activador = 0;
9      int contador;
10     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)
```

Listing 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 };
```

Listing 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 encontrado y `activador` toma el valor de 1 ya que comienza la búsqueda de más repeticiones del substring encontrado hasta que no se cumpla la condición de los caracteres prohibidos; en caso afirmativo se continúa con la siguiente iteración del arreglo.

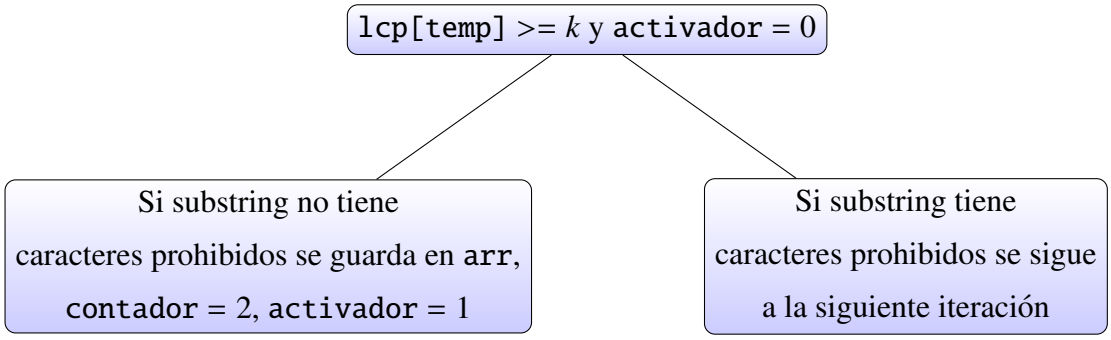



Figura 3.5: Caso 1

*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.

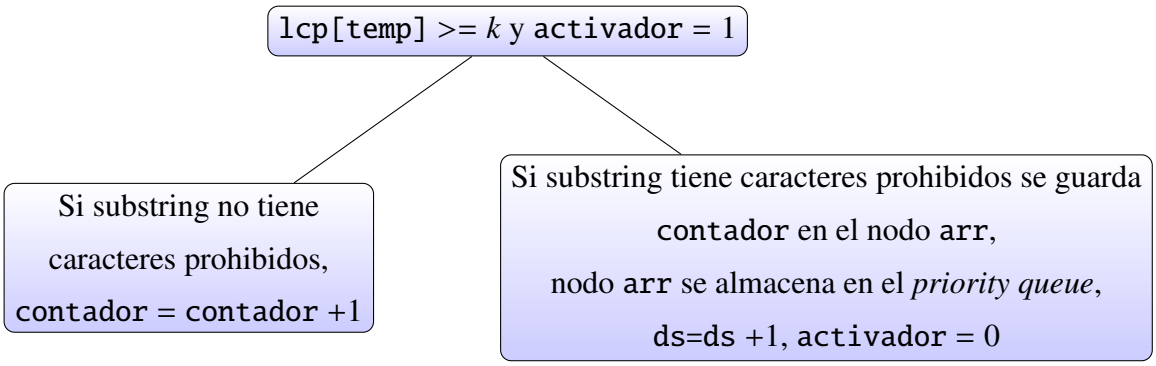



Figura 3.6: Caso 2

*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*.

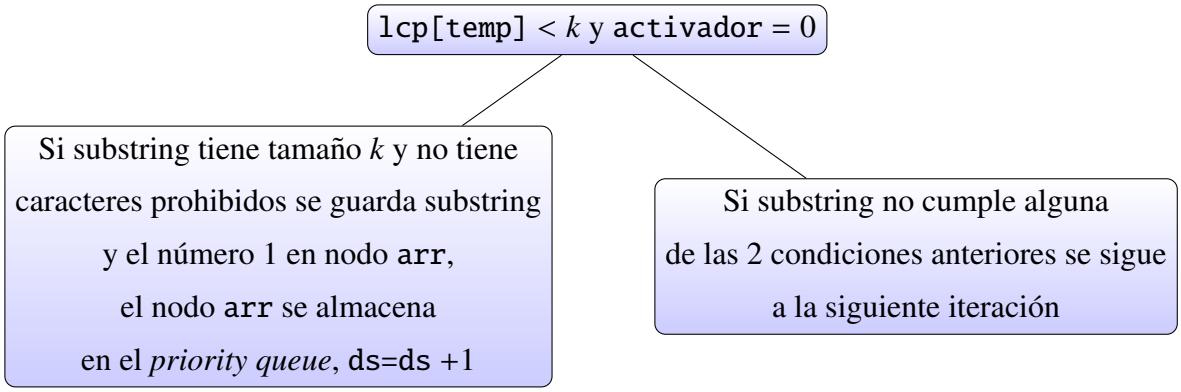



Figura 3.7: Caso 3

*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.

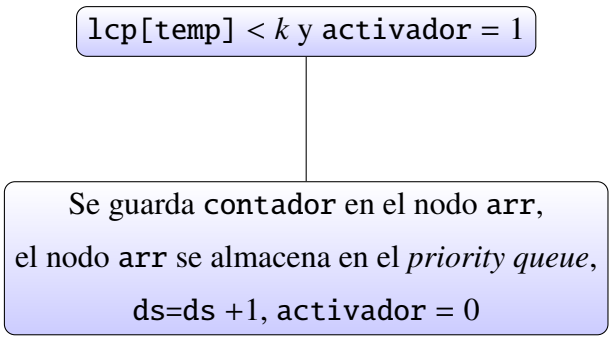



Figura 3.8: Caso 4

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
```

Figura 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

# Capítulo 4

# Análisis de resultados

## 4.1. Resultados obtenidos

## 4.2. Análisis detallado

### 4.2.1. Análisis matemático

### 4.2.2. Análisis biológico

# Conclusiones

# 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
```

# Bibliografía


[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] A. 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] A. 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.