

Trabajo Práctico Integrador – Programación I

Algoritmos de búsqueda y ordenamiento

Alumnos

Maximiliano Niemiec - maximiliano.niemiec@tupad.utn.edu.ar

Paola Pasallo - paola.pasallo@tupad.utn.edu.ar

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Docente Titular

Julieta Trapé

Docente Tutor

Marcos Vega

9 de junio de 2025

Índice

Introducción	3
Marco teórico	5
Algoritmos de búsqueda.....	6
Algoritmos de ordenamiento.....	8
Caso práctico.....	10
Metodología utilizada.....	10
Aplicación de algoritmos	13
Conclusiones	20
Bibliografía	21
Anexo I.....	22

Introducción

El ácido desoxirribonucleico (ADN) contiene nuestra información genética, es decir, la información necesaria para determinar el color de nuestro pelo, de nuestros ojos, tono de piel y muchas más características fenotípicas. Pero no solo se limita a ello, en él también encontramos la información de cómo se conforma nuestro complejo mayor de histocompatibilidad (como nos defendemos ante agentes extraños), como “fabricar” la insulina (esencial para el metabolismo de lípidos y azúcares), como fabricar ciertas proteínas que nos permiten ver e infinidad de otras características que podríamos seguir analizando. No solo los seres humanos poseemos esta información genética: aves, insectos, peces, hongos, virus y bacterias también están “codificados” con este lenguaje natural.

El ADN está compuesto por cuatro moléculas: Adenina, Timidina, Citosina y Guanina. Nos referiremos a ellas como ATCG. El ordenamiento y disposición de estas en el espacio nos aporta, en lenguaje natural, la información necesaria para prácticamente la vida. “H O L A” en idioma español significa un saludo, ATCGGTAGCTCCCATG en idioma “natural” podría significar insulina. Muchos investigadores, equipos médicos, bioquímicos, agrónomos y demás personal de ciencias biológicas usan estas secuencias en investigación y desarrollo de nuevas metodologías para detección de agentes biológicos (como por ejemplo la carrera mundial que hubo por determinar el genoma completo (es decir, el orden y disposición de estas moléculas) del COVID-19. A partir de saberlo y saber que “herramientas” tiene el virus, poder desarrollar estrategias para combatirlo, desarrollar cultivos transgénicos y hasta editarlos para curar enfermedades (por ejemplo con la tecnología CRISPR-Cas). Elegimos este tema porque hay infinidad de bases de datos donde en ocasiones tenemos que buscar secuencias de ADN que significan algo: insulina, proteína del COVID, etc. para realizar diferentes trabajos. Descargamos secuencias de miles o millones de caracteres (ATCG) donde tenemos que buscar específicamente una parte y analizar en qué posición se encuentra, si se repite y cuantas veces,

contar cuantas moléculas que conforman el ácido desoxirribonucleico lo componen al fragmento y con qué frecuencia se repite. Ordenarlas de acuerdo con el “peso” biológico que tienen, etc. para esto la programación resulta una herramienta esencial que nos puede ayudar a alivianar el trabajo, automatizar tareas repetitivas, acortar tiempos de búsqueda y facilitar el ordenamiento de secuencias.

Marco teórico

En el campo de la genética, los algoritmos de búsqueda y ordenamiento juegan un papel crucial para analizar las vastas secuencias de ADN que contienen información esencial para entender enfermedades y características biológicas. La capacidad de estos algoritmos permite manejar datos de dimensiones colosales, desde decenas de miles hasta millones de pares de bases, lo que facilita procesar y comparar información genética con una precisión que excede las posibilidades humanas.

Para ilustrar la utilidad de estos algoritmos, consideremos el caso de la diabetes mellitus tipo 1 (DM1). Esta condición, que afecta a millones de personas en el mundo, está relacionada con mutaciones en ciertas regiones genéticas que podrían predisponer a su desarrollo. Por ejemplo, la región HLA, conocida por su papel en la regulación del sistema inmune, contiene diversos genes que se han asociado con la patogenia de la DM1. Entre estos, el gen CTLA4, que codifica una proteína reguladora en los linfocitos T, ha mostrado variaciones genéticas vinculadas al riesgo de autoinmunidad. Mutaciones como 6230G>A, 319C>T o 1661A>G son ejemplos de cómo un cambio en las moléculas del ADN puede alterar la función biológica.

El análisis de estas mutaciones y su relación con enfermedades como la diabetes se ha beneficiado enormemente de los avances en algoritmos computacionales. Estos algoritmos permiten automatizar el procedimiento de análisis, realizar comparaciones exactas entre secuencias de ADN y ahorrar tiempo y recursos. Además, dada la magnitud de los datos genéticos, los algoritmos especializados son esenciales para manejar el volumen creciente de información, que supera ampliamente las capacidades de procesamiento manual.

Así, la utilización de algoritmos de búsqueda y ordenamiento no solo abre nuevas posibilidades en la investigación genética, sino que también permite abordar problemas biológicos complejos con una eficiencia y una precisión sin precedentes.

Algoritmos de búsqueda

Los algoritmos de búsqueda se utilizan ampliamente en la mayoría de los sistemas informáticos. Los investigadores han estudiado varios algoritmos de búsqueda relacionados con conjuntos de datos ordenados.

- **Búsqueda lineal:** también conocida como búsqueda secuencial, es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado, puede ser una matriz o una lista. Es fácil de implementar, pero puede ser lento para conjuntos de datos grandes.

La complejidad temporal de la búsqueda lineal es $O(n)$, donde n es el número de elementos del array. En el peor de los casos, cuando el elemento objetivo no está presente en el array o se encuentra en la última posición, el algoritmo debe recorrer todo el array, lo que resulta en una complejidad temporal lineal.

- **Búsqueda binaria:** Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.

La complejidad temporal de la búsqueda binaria es $O(\log n)$, donde n es el número de elementos del array. Cada iteración reduce a la mitad el intervalo de búsqueda, lo que resulta en una complejidad temporal logarítmica. Esto hace que la búsqueda binaria sea mucho más rápida que la búsqueda lineal, especialmente para arrays grandes y ordenados.

Las diferencias entre la búsqueda lineal y la búsqueda binaria son:

- La búsqueda lineal busca secuencialmente desde el principio y la binaria divide la matriz por la mitad.
- La búsqueda lineal funciona tanto en matrices ordenadas como no ordenadas en cambio la binaria sólo en ordenadas.
- La complejidad temporal de la búsqueda lineal es $O(n)$ y la de la binaria $O(\log n)$

- La búsqueda lineal es más fácil de implementar mientras que la binaria requiere una implementación más compleja.

- La búsqueda lineal es más adecuada para matrices pequeñas o datos sin clasificar mientras que la binaria es más adecuada para matrices ordenadas de gran tamaño.

- Búsqueda de interpolación: Este algoritmo mejora la búsqueda binaria al estimar la posición del elemento deseado según su valor. Es especialmente eficiente para conjuntos de datos grandes y con valores distribuidos de manera uniforme.

Buscar un elemento en datos ordenados es esencial en el procesamiento de datos, y tanto la búsqueda binaria como la búsqueda por interpolación son herramientas clave para ello. Sin embargo, la eficiencia de la búsqueda por interpolación tradicional depende de que los datos estén distribuidos uniformemente; de lo contrario, su rendimiento puede disminuir notablemente.

La búsqueda binaria interpolada (IBS) está diseñada para trabajar con datos de distribución desconocida o que cambian dinámicamente, adaptándose mejor a estos escenarios.

La principal diferencia entre ambos métodos radica en cómo determinan el punto de división, lo que influye directamente en el rendimiento de cada algoritmo según el tipo de datos que se maneje.

- Búsqueda de hash: Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

El hash es un concepto esencial en Python debido a su eficiencia y versatilidad. Permite la rápida recuperación y comparación de datos, lo que lo hace ideal para tareas como la búsqueda, la indexación y la validación de datos. El método `hash()` de Python proporciona una forma práctica de generar valores hash para objetos, lo que permite una manipulación y organización eficientes de los datos.

Algoritmos de ordenamiento

Los algoritmos de ordenamiento desempeñan un papel fundamental en la organización de datos dentro de estructuras como arreglos u otros formatos. La correcta disposición de la información es esencial para muchas de las aplicaciones con las que interactuamos diariamente, ya que permite realizar búsquedas, análisis y otras operaciones de manera más rápida y eficiente.

Entre los beneficios de los algoritmos de ordenamiento se encuentran:

- **Búsqueda más eficiente:** Los datos ordenados facilitan y aceleran la localización de elementos específicos.
- **Análisis simplificado:** La estructura organizada de la información permite identificar patrones y tendencias con mayor facilidad. Por ejemplo, en el ámbito de la genética, un análisis de secuencias ordenadas podría revelar que una población con mayor prevalencia del gen de la insulina con la base "A" en lugar de "T" tiene una predisposición más alta a desarrollar resistencia a la insulina, lo que incrementa el riesgo de padecer diabetes tipo 2.

Dentro de los algoritmos de ordenamiento más utilizados, se encuentran diversas propuestas que optimizan la organización de datos según las necesidades y características específicas del conjunto de información.

- **Ordenamiento por burbuja (bubble sort):** Es un algoritmo de ordenamiento simple y fácil de implementar. Se basa en recorrer el vector de datos comparando los elementos adyacentes entre sí, intercambiándolos si el elemento de la derecha es mayor o menor según el criterio que se desee emplear. Esto debe repetirse una y otra vez sobre el vector hasta dejarlo ordenado.

- **Ordenamiento por selección:** Es otro algoritmo de ordenamiento simple que funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista estén ordenados.

- Ordenamiento por inserción (insertion sort): A diferencia del burbujeo, en este algoritmo se recorre el vector y se toma el valor evaluado y se lo compara con cada uno de los elementos anteriores hasta insertarlo en el lugar correcto. Si bien, esta es una mecánica que sería la tendencia a aplicar para un ser humano, es bastante más compleja de programar.

- Ordenamiento rápido: Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes y luego ordenando cada parte de forma recursiva.

- Ordenamiento por mezcla: Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas.

- Ordenamiento por onda (ripple sort): Este es una variante de bubble sort para mejorar el tiempo de ejecución al evitar repetir constantemente y es relativamente sencillo de idear. Se basa en un sistema de pivot, donde se ordena una posición a la vez. Luego, esta se va comparando con las posiciones siguientes hasta alcanzar el último elemento ($p+1$, $p+2$, $p+3$, ..., $p+(N-1)$), intercambiando los valores del vector (swap) si cumplen el criterio de ordenamiento (ascendente o descendente).

Caso práctico

Para llevar a cabo el caso práctico decidimos simular ser un grupo de médicos que realizarán análisis genéticos a dos poblaciones: una urbana y una rural, y dentro de ellas las diferenciamos por sexo (masculino o femenino) y edades (0-10 años, 11 a 50 años y >50 años). Vamos a estudiar la composición molecular de las bases “A,T,C,G” y en base a ello proporcionar una probabilidad de desarrollar la enfermedad o no. Algunas de las condiciones clínicas expresadas en la metodología utilizada no son reales en la vida. Fueron inventadas a fines de implementar algoritmos de búsqueda y ordenamiento y poder mostrar cómo estos funcionan podrían ayudarnos en un contexto como el planteado.

Metodología utilizada

1. Generación de datos:

- Simulamos una muestra de 50 pacientes ($n = 50$), divididos equitativamente en dos poblaciones:
 - Zona urbana
 - Zona rural
- Dentro de cada grupo, segmentamos:
 - Entre 0-10 años
 - Entre 11-50 años
 - Mayores de 50 años
- A cada paciente se le asigna aleatoriamente una secuencia de ADN simulada de bases (letras A, T, C, G que variarán entre 1.000, 5.000, 10.000).

2. Definición de patrones genéticos de riesgo:

- Definimos mutaciones simuladas.

Por ejemplo: "ATCGCGAA"

"TTCGTGC"

"ATTCGTCGA"

Imaginemos que estas tres secuencias son predisponentes de X enfermedad. Para eso haremos uso de la búsqueda lineal.

Establecemos reglas simples para interpretar riesgo en base a:

- La frecuencia relativa de cada base (A, T, C, G) influyen en la probabilidad de desarrollo de la enfermedad.

Si esos pacientes tienen más de un 25% de "A", las probabilidades de desarrollar "X" enfermedad son aún mayores. Pero si la cantidad de "G" supera el 25% es un buen indicio de protección y baja probabilidad de manifestar la enfermedad a pesar de tener una secuencia mutada. También puede ser indeterminada su condición clínica si ninguna de las anteriores mencionadas se cumplen.

3. Aplicación de algoritmos:

- **Búsqueda:**

- Usamos algoritmos como búsqueda lineal para detectar:
 - Mutaciones específicas
 - Frecuencias moleculares (ejemplo: cuántas veces aparece "A, T, C o G")

- **Ordenamiento:**

- Ordenamos a los pacientes por rango de edades para;
 - Facilitar la visualización de los rangos etarios con los que se trabajan.
- Se ordenó de menor a mayor

Poder visualizar dónde está la mayor incidencia de la mutación buscada, de forma más rápida.

Generación de datos

Paso 1: Importamos librerías que utilizaremos. Estas son time y random

```
import random
import time
```

Paso 2: Generamos una lista con strings. Estas serán nuestras bases A, T, C y G a utilizar

```
# Bases nitrogenadas posibles en modo de lista
bases = ['A', 'T', 'C', 'G']
```

Paso 3: Definimos varias longitudes para generar distintas secuencias. Así, podremos ver como nuestra búsqueda lineal a medida que aumenta el n sobre el cual buscar, aumenta el tiempo que necesita

```
# Longitudes posibles para las secuencias de ADN que elegimos
longitudes = [1000, 5000, 10000, 100000, 250000, 1000000]
```

Paso 4: Definimos rangos de edad

```
# Rangos de edad (niños, adultos, mayores)
def generar_edad():
    prob = random.random() #Esto me generará un float aleatorio
    entre 0.0 y 1.0. En base a eso evaluo:
    if prob < 0.25: #Si el aleatorio generado es menor a 0.25, me
    generara una edad aleatoria en el rango (0,10) años
        return random.randint(0, 10) #Y así con los otros valores
    elif prob < 0.75:
        return random.randint(11, 50)
    else:
        return random.randint(51, 90)
```

Paso 5: Generamos una secuencia de ADN aleatoria

```
# Función para generar una secuencia de ADN aleatoria
def generar_adn(longitud):
    return ''.join(random.choices(bases, k=longitud))
```

En esta función se ingresa como argumento una longitud (int) de "ATCG" que se desee y el código buscará aleatoriamente desde la lista "bases" alguno de sus elementos. En k=longitud indicamos cuantos elementos queremos que nos devuelva. Luego lo unimos con join ya que el resultado de ella es una lista ['A','T','T','C'] que necesitamos que pase a ser 'ATTC'.

Paso 6: Generamos nuestra base de datos sobre la cual trabajar

```
# Generación de base de datos
n = 50 → Nuestra población de 50 personas
base_de_datos = [] → Nuestra base de datos será una lista que contiene diccionarios
for i in range(n): → Iteramos de 0 a 49
    persona = {
        'id': i + 1, a*
        'sexo': random.choice(['Masculino', 'Femenino']), b*
        'edad': generar_edad(), c*
        'zona': random.choice(['Rural', 'Urbana']), d*
        'longitud_adn': random.choice(longitudes), e*
    }
    persona['adn'] = generar_adn(persona['longitud_adn']) f*
    base_de_datos.append(persona) g*
```

a* → Se asigna una identificación al paciente. Va tomando el valor de i+1 (en el primer caso 0+1 = 1)

b* → Hacemos uso del código random.choice para elegir aleatoriamente el sexo (masculino o femenino)

c* → Generamos aleatoriamente la edad del paciente

d* → Hacemos uso del código random.choice para elegir aleatoriamente la zona (rural o urbana)

e* → Definimos la longitud del ADN utilizando random.choice y le damos la lista “longitudes”

f* → persona[‘adn’] agregará la clave ‘adn’ al diccionario y como valor recibe la función “generar_adn” que esta, a su vez, recibe como argumento la longitud del ADN generada en el apartado “e*”

g* → Agregamos cada paciente(diccionario) a la lista base_de_datos

Aplicación de algoritmos

Paso 1: Búsqueda lineal a partir de las secuencias inventadas “ATCGCGAA”, “TTCGTGC” y “ATTCTCGA”

El siguiente código tiene dos parámetros: base_de_datos (que recibe como argumento una base de datos. Por ejemplo, la que creamos anteriormente) y objetivo que es la secuencia mutada que queremos estudiar.

f* En caso de que se haya recorrido todo el diccionario y no se haya encontrado la mutación en ningún paciente, se imprimirá por pantalla que no se ha encontrado coincidencia.

File Edit Selection View Go Run ... Search

tp-integrador.py x

C:\Users\Usuario\Desktop > Maxi > Programación > Tecnicatura UTN > Primer año > Programación I > TP_Integrador > tp-integrador.py ...

31 1 1 1

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

Imaginemos que estas tres secuencias son predisponentes de X enfermedad. Para eso haremos uso de la búsqueda lineal

Buscaremos: ATCGCGAA, TTCTGCG y ATTCGTGCA

Primera búsqueda: ATCGCGAA

Mutación encontrada en la secuencia del paciente ID: 4

Mutación encontrada en la secuencia del paciente ID: 5

Mutación encontrada en la secuencia del paciente ID: 6

Mutación encontrada en la secuencia del paciente ID: 7

Mutación encontrada en la secuencia del paciente ID: 8

Mutación encontrada en la secuencia del paciente ID: 10

Mutación encontrada en la secuencia del paciente ID: 11

Mutación encontrada en la secuencia del paciente ID: 12

Mutación encontrada en la secuencia del paciente ID: 13

Mutación encontrada en la secuencia del paciente ID: 14

Mutación encontrada en la secuencia del paciente ID: 15

Mutación encontrada en la secuencia del paciente ID: 16

Mutación encontrada en la secuencia del paciente ID: 17

Mutación encontrada en la secuencia del paciente ID: 19

Mutación encontrada en la secuencia del paciente ID: 21

Mutación encontrada en la secuencia del paciente ID: 23

Mutación encontrada en la secuencia del paciente ID: 25

Mutación encontrada en la secuencia del paciente ID: 27

Mutación encontrada en la secuencia del paciente ID: 29

Mutación encontrada en la secuencia del paciente ID: 31

Mutación encontrada en la secuencia del paciente ID: 33

Mutación encontrada en la secuencia del paciente ID: 34

Mutación encontrada en la secuencia del paciente ID: 35

Mutación encontrada en la secuencia del paciente ID: 36

Mutación encontrada en la secuencia del paciente ID: 41

Mutación encontrada en la secuencia del paciente ID: 42

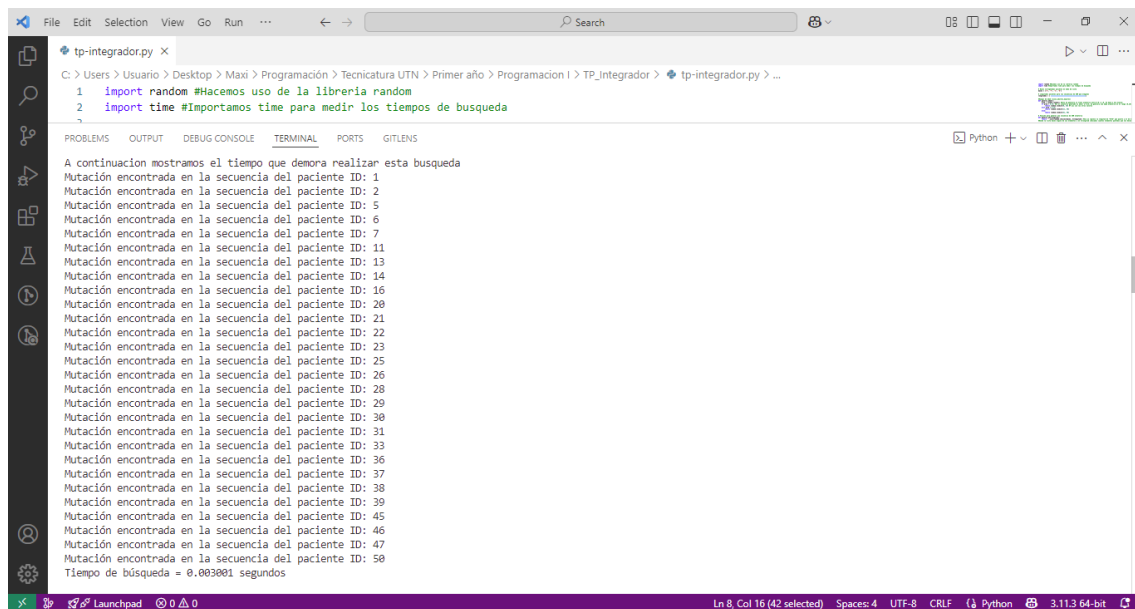
Mutación encontrada en la secuencia del paciente ID: 43

Mutación encontrada en la secuencia del paciente ID: 44

C:\Program Files\Python\Python311\Scripts\python.exe 3113.64 KB

Tiempo de búsqueda:

La siguiente imagen muestra cuánto demoró el programa en encontrar las secuencias mutadas después de pasar por los 50 pacientes

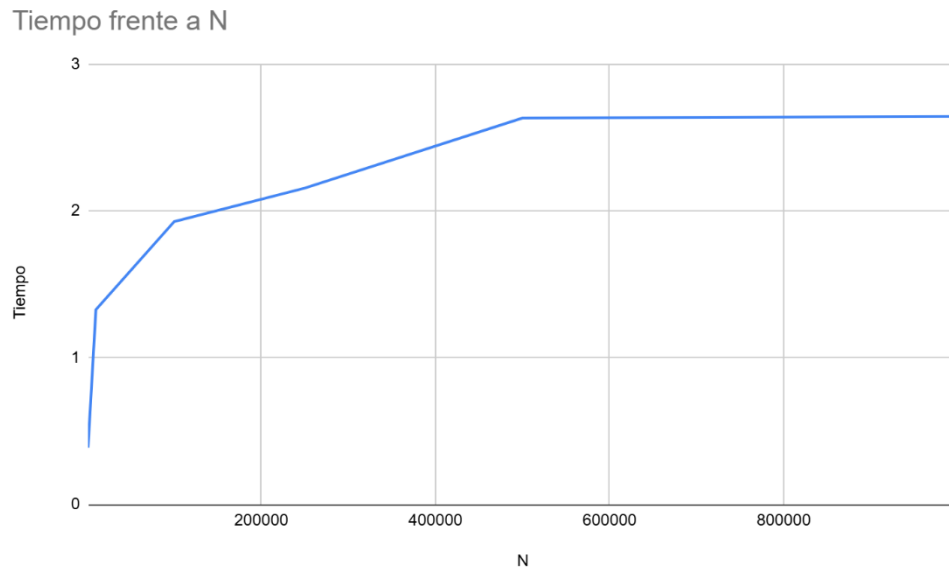


```

tp-integrador.py
C:\Users\Usuario\Desktop>Maxi>Programación>Tecnatura UTN>Primer año>Programación I>TP_Integrador>tp-integrador.py > ...
1 import random #Hacemos uso de la libreria random
2 import time #Importamos time para medir los tiempos de busqueda
3
A continuación mostramos el tiempo que demora realizar esta búsqueda
Mutación encontrada en la secuencia del paciente ID: 1
Mutación encontrada en la secuencia del paciente ID: 2
Mutación encontrada en la secuencia del paciente ID: 5
Mutación encontrada en la secuencia del paciente ID: 6
Mutación encontrada en la secuencia del paciente ID: 7
Mutación encontrada en la secuencia del paciente ID: 11
Mutación encontrada en la secuencia del paciente ID: 13
Mutación encontrada en la secuencia del paciente ID: 14
Mutación encontrada en la secuencia del paciente ID: 16
Mutación encontrada en la secuencia del paciente ID: 20
Mutación encontrada en la secuencia del paciente ID: 21
Mutación encontrada en la secuencia del paciente ID: 22
Mutación encontrada en la secuencia del paciente ID: 23
Mutación encontrada en la secuencia del paciente ID: 25
Mutación encontrada en la secuencia del paciente ID: 26
Mutación encontrada en la secuencia del paciente ID: 28
Mutación encontrada en la secuencia del paciente ID: 29
Mutación encontrada en la secuencia del paciente ID: 30
Mutación encontrada en la secuencia del paciente ID: 31
Mutación encontrada en la secuencia del paciente ID: 33
Mutación encontrada en la secuencia del paciente ID: 36
Mutación encontrada en la secuencia del paciente ID: 37
Mutación encontrada en la secuencia del paciente ID: 38
Mutación encontrada en la secuencia del paciente ID: 39
Mutación encontrada en la secuencia del paciente ID: 45
Mutación encontrada en la secuencia del paciente ID: 46
Mutación encontrada en la secuencia del paciente ID: 47
Mutación encontrada en la secuencia del paciente ID: 50
Tiempo de búsqueda = 0.003001 segundos
  
```

A medida que las secuencias son mayores, más tiempo tarda en realizar esta búsqueda lineal siguiendo una complejidad de tipo $O(n)$. Para eso fuimos cambiando en la lista “longitudes” los diferentes valores que se muestran en la tabla a continuación:

n	tiempo (ms)
1000	0,387
10000	1,327
100000	1,928
250000	2,156
500000	2,633
1000000	2,644



Paso 2: Vamos a buscar a los pacientes que tienen el gen mutado y filtrarlos por su edad, sexo y zona.

```
def pacientes_mutados(base_de_datos, objetivo):
    edades = []
    sexo = []
    zonas = []
    adn = []
    for persona in base_de_datos:
        if objetivo in persona['adn']:
            edades.append(persona['edad'])
            sexo.append(persona['sexo'])
            zonas.append(persona['zona'])
            adn.append(persona['adn'])
    return edades, sexo, zonas, adn

pacientes_mutados(base_de_datos, "TTCGTGC")
```

Paso 3: Vamos a filtrar por edades los pacientes mutados. Con esto podremos facilitar la evaluación visual del rango etario con el cual trabajamos.

```
pacientesMutadosPorEdad = pacientes_mutados(base_de_datos,
"TTCGTGC") [0] → Con [0] nos quedamos con la primera lista, la de edades
```


Paso 4: Ordenamos la lista con dos algoritmos diferentes: sorted y quick sort

Definimos Quick sort

```
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]

    return quick_sort(menores) + [pivote] + quick_sort(mayores)

print(f"Lista ordenada con sorted: {sorted(pacientesMutadosPorEdad)}")

print(f'Lista ordenada con Quick sort: {quick_sort(pacientesMutadosPorEdad)}')
```

Paso 5: Evaluamos los tiempos que demora el ordenamiento con ambas metodologías

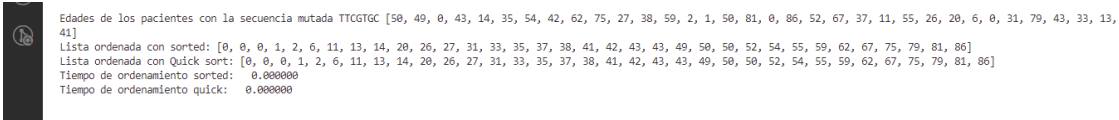
```
inicio_sorted = time.time()
listaDeSorted = sorted(pacientesMutadosPorEdad)
fin_sorted = time.time()
tiempo_sorted = fin_sorted - inicio_sorted

inicio_quick = time.time()
lista_quick = quick_sort(pacientesMutadosPorEdad)
fin_quick = time.time()
tiempo_quick = fin_quick - inicio_quick

print(f"Tiempo de ordenamiento sorted: {tiempo_sorted:10f}")
print(f"Tiempo de ordenamiento quick: {tiempo_quick:10f}")
```

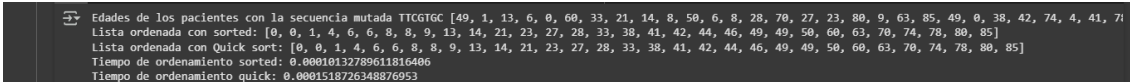
Utilizando visual studio code podemos ver que no hay diferencias entre ambos métodos para orden, pero al ejecutar con Coolab si vemos diferencias y en este caso Sorted es más eficiente.

Ejecución en VSC



```
Edades de los pacientes con la secuencia mutada TTCGTGC [50, 49, 0, 43, 14, 35, 54, 42, 62, 75, 27, 38, 59, 2, 1, 50, 81, 0, 86, 52, 67, 37, 11, 55, 26, 20, 6, 0, 31, 79, 43, 33, 13, 41]
Lista ordenada con sorted: [0, 0, 0, 1, 2, 6, 11, 13, 14, 20, 26, 27, 31, 33, 35, 37, 38, 41, 42, 43, 43, 49, 50, 50, 52, 54, 55, 59, 62, 67, 75, 79, 81, 86]
Lista ordenada con Quick sort: [0, 0, 0, 1, 2, 6, 11, 13, 14, 20, 26, 27, 31, 33, 35, 37, 38, 41, 42, 43, 43, 49, 50, 50, 52, 54, 55, 59, 62, 67, 75, 79, 81, 86]
Tiempo de ordenamiento sorted: 0.000000
Tiempo de ordenamiento quick: 0.000000
```

Ejecución en Google Coolab



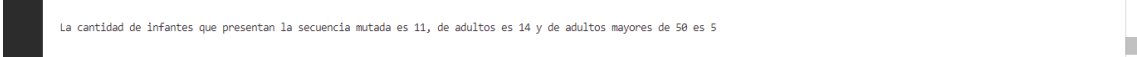
```
Edades de los pacientes con la secuencia mutada TTCGTGC [49, 1, 13, 6, 0, 60, 33, 21, 14, 8, 50, 6, 8, 28, 70, 27, 23, 80, 9, 63, 85, 49, 0, 38, 42, 74, 4, 41, 71, 41]
Lista ordenada con sorted: [0, 0, 1, 4, 6, 6, 8, 8, 9, 13, 14, 21, 23, 27, 28, 33, 38, 41, 42, 44, 46, 49, 49, 50, 60, 63, 70, 74, 78, 80, 85]
Lista ordenada con Quick sort: [0, 0, 1, 4, 6, 6, 8, 8, 9, 13, 14, 21, 23, 27, 28, 33, 38, 41, 42, 44, 46, 49, 49, 50, 60, 63, 70, 74, 78, 80, 85]
Tiempo de ordenamiento sorted: 0.00010132789611816406
Tiempo de ordenamiento quick: 0.0001518726348876953
```

Las diferencias podrían deberse a:

- Diferencias en las secuencias (ya que cada ejecución genera aleatoriamente diferentes secuencias. Puede ser que trabajando con VSC éstas hayan sido más chicas y el ordenamiento le resultó más fácil)
- Diferencias en hardware: Quizá la máquina virtual que nos “presta” Google sea un poco menos eficiente que nuestra computadora personal.

Paso 5: Visualmente vemos que las personas adultas son las que mayormente poseen la secuencia mutada, pero con el siguiente código llegamos a valores exactos.

```
def mutados_por_edad(secuencia):
    infantes = 0
    adultos = 0
    adultos_mayores_50 = 0
    for edad in pacientes_mutados(base_de_datos, secuencia)[0]:
        if edad > 0 and edad < 10:
            infantes += 1
        elif edad > 11 and edad < 50:
            adultos += 1
        else:
            adultos_mayores_50 += 1
    print(f"La cantidad de infantes que presentan la secuencia mutada es {infantes}, de adultos es {adultos} y de adultos mayores de 50 es {adultos_mayores_50}")
```



Paso 6: Por último, veremos como la frecuencia relativa de cada base influye en la probabilidad de desarrollo de la enfermedad según el criterio clínico propuesto. En los pacientes que presentan la secuencia mutada necesitamos buscar cuantas A tienen y cuantas G tienen. También sacar el porcentaje de cada una.

```
secuencias_adn = pacientes_mutados(base_de_datos, "TTCGTGC")[3] a*

for idx, adn in enumerate(secuencias_adn): b*
    total_bases = len(adn)
    cantidad_A = adn.count('A') c*
    cantidad_G = adn.count('G')

    porcentaje_A = (cantidad_A / total_bases) * 100 d*
    porcentaje_G = (cantidad_G / total_bases) * 100

    print(f"Paciente {idx + 1}:")
```

```
print(f"Total de bases: {total_bases}")

if porcentaje_A > 25 and porcentaje_G > 25: e*
    print("Este paciente tiene riesgo de desarrollar la
enfermedad. A supera el %25")
elif porcentaje_G < 25:
    print("Paciente con baja probabilidad de desarrollar la
enfermedad. G es inferior a %25")

else:
    print("Condicion clinica indeterminada en base a la
proporcion de bases A y G")

print(f"  A: {cantidad_A} ({round(porcentaje_A,2)}%) ")
print(f"  G: {cantidad_G} ({round(porcentaje_G,2)}%) \n")
```

a* → A partir de la función `pacientes_mutados`, filtramos las secuencias de ADN que ellos tienen y las guardamos en una variable

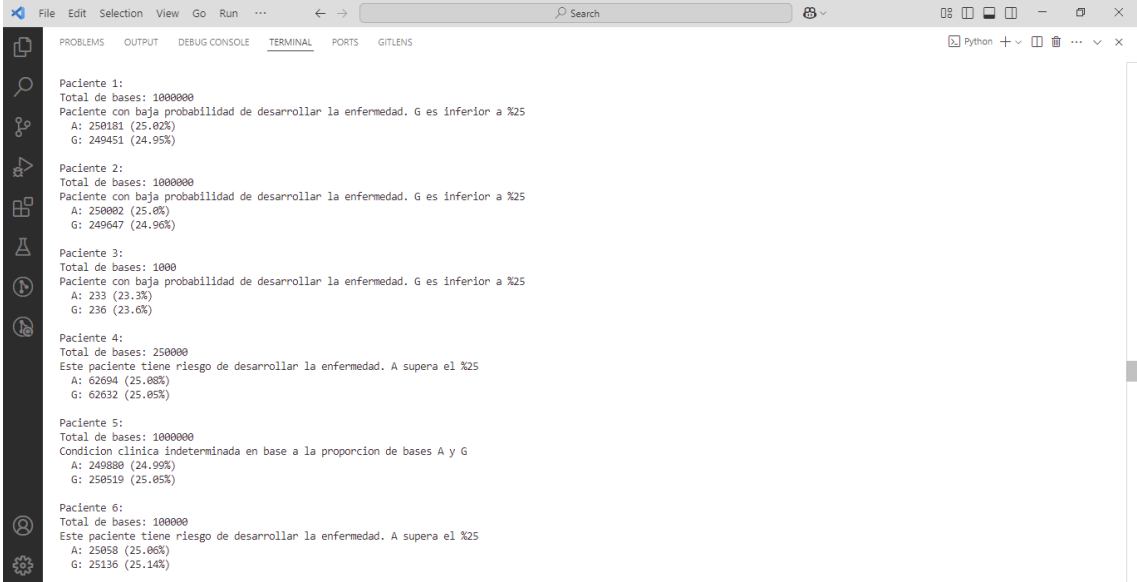
b* → Usamos “enumerate” para recorrer una lista y a cada elemento de ella lo vamos enumerando para que los datos queden más presentables en la impresión por pantalla.

c* → Contamos las “A” y las “G” de las secuencias

d* → Sacamos el porcentaje que hay de cada base en la secuencia de cada paciente

e* → Establecemos una condición clínica

Ejecución del código



```
Paciente 1:
Total de bases: 1000000
Paciente con baja probabilidad de desarrollar la enfermedad. G es inferior a %25
A: 250181 (25.02%)
G: 249451 (24.95%)

Paciente 2:
Total de bases: 1000000
Paciente con baja probabilidad de desarrollar la enfermedad. G es inferior a %25
A: 250002 (25.0%)
G: 249647 (24.96%)

Paciente 3:
Total de bases: 1000
Paciente con baja probabilidad de desarrollar la enfermedad. G es inferior a %25
A: 233 (23.3%)
G: 236 (23.6%)

Paciente 4:
Total de bases: 250000
Este paciente tiene riesgo de desarrollar la enfermedad. A supera el %25
A: 62694 (25.08%)
G: 62632 (25.05%)

Paciente 5:
Total de bases: 1000000
Condicion clinica indeterminada en base a la proporcion de bases A y G
A: 249880 (24.99%)
G: 250519 (25.05%)

Paciente 6:
Total de bases: 100000
Este paciente tiene riesgo de desarrollar la enfermedad. A supera el %25
A: 25058 (25.06%)
G: 25136 (25.14%)
```

Conclusiones

En nuestro equipo quisimos plantear en este trabajo práctico cómo a través de algoritmos de búsqueda y ordenamiento podemos analizar bases de datos y extraer información valiosa de manera automatizada, rápida y con certeza de no equivocarnos en cuestiones inherentes a lo humano. El caso práctico plantea el estudio de secuencias de ADN que al ojo de las personas son caracteres muy fáciles de confundir, pero sumamente importantes para la biología su orden y posición. Pudimos utilizar una base de datos para analizar en milisegundos el ADN de 50 pacientes y en base a ciertos criterios ficticios planteados encontrar mutaciones, diferenciar aquellos individuos que presentaban cambios en sus bases “ATCG” según el rango etario y definir de acuerdo con la composición de su ADN si tenían mayores o menores probabilidades de desarrollar cierta enfermedad. El uso de estos algoritmos resulta fundamental en análisis e investigaciones de la índole planteada y como futuros profesionales en el área de la programación consideramos que nuestra formación nos dará la capacidad de llevar a cabo las tareas requeridas, colaborar en el ámbito de las ciencias y poder proporcionar información de calidad.

Bibliografía

- J. C. Wiebe, A.M. Wagner, F. J. Novoa Mogollón, 2011. Genética de la diabetes mellitus. Disponible en: <https://revistanefrologia.com/es-genetica-diabetes-mellitus-articulo-X2013757511002452>
- B. Kumar, 2024. Difference between linear search and binary search in python. [Citado 7 de junio 2025]. Disponible en: <https://pythonguides.com/python-binary-search/>
- A. D. Mohammed, S. E. Amrahov, F. V. Celebi, 2021. Interpolated binary search: An efficient hybrid search algorithm on ordered datasets. Disponible en: <https://www.sciencedirect.com/science/article/pii/S221509862100046X>
- N. Tiwari, 2024. Comprehensive Guide on Python hash () Method. [Citado 7 de junio 2025]. Disponible en: <https://www.analyticsvidhya.com/blog/2024/01/comprehensive-guide-on-python-hash-method/>
- M. S. Ávalos, 2025. Taller de programación: Búsqueda y ordenamiento. Disponible en: <https://tute-avalos.com/static/python/10%20-%20Algoritmos%20de%20b%C3%BAsqueda%20y%20ordenamiento.pdf>

Anexo I

Video Explicativo. Disponible en: <https://www.youtube.com/watch?v=hWxIsnwladv>