UNIVERSIDAD TECNOLOGICA NACIONAL

TRABAJO PRÁCTICO INTEGRADOR

Análisis de Algoritmos

Materia: Programación 1

Alumnas: Andrea Julia Ayala y Estefanía Ávalos

Comisión 24

Profesor: Bruselario Sebastián / Tutora: Gubiotti Flor

Índice:

Introducción:

Hemos decidido abordar el análisis de algoritmos por la relevancia que este tema adquiere al momento de decidir cómo encaminar la resolución de un problema y buscar la solución más eficiente. Se trata de una habilidad fundamental para cualquier programador, especialmente cuando se trabaja con aplicaciones que manejan grandes volúmenes de datos.

Nos proponemos explorar tanto el enfoque teórico como el práctico, analizando las ventajas y desventajas de cada uno. Abordaremos conceptos clave como la eficiencia temporal y espacial, aspectos imprescindibles en este tipo de análisis.

Además, nos planteamos una duda central: ¿es suficiente con medir el tiempo de ejecución, o es posible evaluar un algoritmo de forma independiente de las características específicas de la máquina en la que se ejecuta?

Marco Teórico:

Un algoritmo es un conjunto de instrucciones bien definidas, diseñadas para resolver un problema específico. Evaluar la eficacia de esa solución es un concepto clave.

¿Qué puntos son importantes de verificar? Para que sea **correcto**, debe resolver el problema, y lo hace si devuelve una salida correcta para cada entrada posible. Esto nos lleva a hablar de **robustez**: un algoritmo robusto puede manejar situaciones inesperadas sin fallar. Finalmente, llegamos a la **eficiencia**, una característica esencial que nos indica si los recursos se utilizan de la mejor manera para alcanzar el objetivo de forma óptima, tanto en tiempo como en memoria. Cuando hablamos del tiempo de ejecución, nos referimos a la eficiencia temporal; cuando hablamos del uso de memoria, a la eficiencia espacial.

Para poder definir si un algoritmo es correcto, robusto y eficiente, podemos hacer distintos tipos de análisis y, según los resultados obtenidos, hacer ajustes en caso de ser necesarios. A continuación veremos en qué consisten el Análisis Empírico y el Análisis Teórico.

- ***Análisis empírico:***

Es el que se basa en la **experimentación y observación real de los resultados**. En este contexto, el algoritmo tiene que ser implementado en un lenguaje de programación y ejecutado en una computadora.

Para esto, podemos usar herramientas que nos permitan medir la eficiencia temporal (tiempo de ejecución) y el uso de memoria, que son los parámetros clave. Además, también podemos aprovechar esta instancia para verificar que el algoritmo resuelve correctamente el problema planteado, incluso ante situaciones no esperadas, con distintos casos de prueba.

Sin embargo, nos parece relevante aclarar que, a pesar de que elegimos optar por la función time (ya que fue presentada en las clases de guía y queríamos aplicar lo aprendido), sabemos que existe otra función llamada timeit (que suele ser más precisa para pequeñas operaciones, aunque tiene otras limitaciones).

La función time tiene la ventaja de medir el tiempo de cualquier bloque de código, incluso partes específicas de un programa y es ideal para perfilar secciones de un script o aplicación completa. Además, es útil para calcular las duraciones entre eventos (ej. inicio/fin de un proceso).

Lo negativo es que esta función depende del reloj del sistema y la medición puede verse afectada dependiendo de otras tareas en ejecución: como mide el tiempo "real", no compensa por ejemplo delays del SO.

Como resultado podemos pensar que para este trabajo práctico puede ser de mucha utilidad para dar un pantallazo general de cómo funciona cada bloque del código, pero entendemos que el tiempo puede variar según las condiciones de ejecución y esto afectará levemente a los resultados obtenidos.

Por último, es importante mencionar que al trabajar con un volumen de datos reducido, en principio las mediciones pueden no parecer tan dispares entre sí (con relación a los resultados numéricos obtenidos en la ejecución de los distintos bloques de código), pero esta diferencia se hará más evidente y notoria al trabajar con grandes cantidades de datos o en ejecuciones más largas.

- ***Análisis teórico:***

A diferencia del análisis empírico, éste busca evaluar el comportamiento de un código sin haberlo puesto en marcha. Se hacen hipótesis respecto a lo conocido como **Complejidad Algorítmica** (Big O), en la que se evalúan el tiempo que puede tardar en ejecutarse el código (en función del tamaño de la entrada) y el espacio en memoria necesario (dependiendo de si se usan variables simples o si se requiere mayor espacio para estructuras nuevas).

Además, se tiene en cuenta la **correctitud** (cuáles son las precondiciones, las postcondiciones e invariantes para el algoritmo funcione); las **estructuras de datos** elegidas según el caso (listas, conjuntos, etc) y su impacto en la eficiencia; los **paradigmas y patrones** (evaluar si es iterativo, recursivo, etc); y **casos de prueba teóricos** (cuándo el código es más/menos eficiente, cuál es la media de eficiencia, etc).

Caso Práctico

Para poner a prueba ambos enfoques, queremos mostrar cómo se comportan diferentes formas de resolver un mismo problema. Para eso, vamos a aplicar distintos métodos de resolución a los mismos casos y así poder compararlos. De esta manera, vamos a poder medir el tiempo que tarda cada uno y cómo es el uso de memoria en cada situación, logrando así analizar cuál resulta más eficiente en cuanto a estos puntos.

Para darle coherencia y sentido a las pruebas, creamos un contexto ficticio que nos sirve como hilo conductor, ayudándonos a mantener una línea clara en los distintos ejemplos que analizamos.

- ***Contexto:***

Empezamos a trabajar en un local de venta al público. Nos está yendo muy bien pero no damos a basto con todo el trabajo.
Necesitamos un programa que resuelva los siguientes problemas:

1) Ver si un cliente específico hizo un pedido;
2) Sumar el total de los productos pedidos;

3) Reordenar los pedidos antes de entregarlos (del último al primero).

Para crear el programa que realice estas tareas tenemos infinitas opciones, pero en esta oportunidad vamos a comparar las siguientes alternativas:

1) Analizar si el cliente buscado está en una lista usando For vs. buscarlo en un conjunto usando Set();
2) Sumar los artículos mediante otro bucle For vs. usando Sum(lista) vs. aplicar una función recursiva;
3) Reordenar la lista mediante un Slicing ([::-1]) vs. el uso de un Reversed() vs. a través de un bucle manual.

• *Código y análisis Empírico de la Propuesta:*

En esta sección vamos a centrarnos en las herramientas que nos da Python para medir el tiempo y memoria consumidos en la ejecución de una función o la creación de un elemento (lista o conjunto). Herramientas como time y tracemalloc serán clave.

A nivel código, tenemos dos funciones que serán reutilizadas en varias partes del trabajo y que son las que efectivamente miden tiempo y memoria de ejecución/creación para cada función/dato que pasamos como parámetros. Vamos a explicarlo detenidamente.

Primero, la función `medir_tiempo_y_memoria` sirve para ejecutar cualquier función que le pasemos y al mismo tiempo medir cuánto tarda y cuánta memoria consume durante esa ejecución.

Más específicamente:

1. Inicia el monitoreo de uso de memoria.

2. Anota el momento en que empieza la ejecución de la función.

3. Ejecuta la función con los parámetros que se le pasan.

4. Anota el momento en que termina.

5. Calcula cuántos segundos tardó.

6. Mide cuánta memoria está usando en ese instante y cuál fue el pico máximo que se usó durante toda la ejecución.

7. Detiene el monitoreo de memoria.

8. Finalmente, devuelve el resultado de la función, el tiempo que tardó, la memoria usada y el pico de memoria.

```
# Función para medir tiempo y memoria de la ejecución de cada función
generada anteriormente
def medir_tiempo_y_memoria(func, *args):
    tracemalloc.start()
    tiempo_de_inicio = time.time()
    resultado_de_la_funcion = func(*args)
    tiempo_de_finalización = time.time()
    tiempo_transcurrido = tiempo_de_finalización - tiempo_de_inicio
    memoria_actual, memoria_maxima = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return resultado_de_la_funcion, tiempo_transcurrido, memoria_actual,
memoria_maxima
```

La segunda función, `medir_estructura`, está pensada para que se pueda probar estructuras como listas o conjuntos (por ejemplo, pasarle `set` o `list`), y ver qué tanto tardan y cuánto consumen al crearse con un conjunto de datos.

Usa la función explicada anteriormente para medir la creación de esa estructura, mostrar por consola un mensaje con el tiempo y los valores de memoria, y devolver la estructura creada, por si se necesita.

La función además muestra en consola si la estructura que se está midiendo es un conjunto o una lista, y le agrega una etiqueta descriptiva para que se entienda mejor en los resultados, y se sepa qué es lo que estamos midiendo.

Esto es muy útil para comparar, por ejemplo, si te conviene usar un `set` o una `list` cuando querés buscar elementos muchas veces, y ver el impacto en tiempo y memoria en distintos volúmenes de datos a la hora de la creación de los mismos.

```
# Función para medir tiempo y memoria de la creación de un conjunto o lista
def medir_estructura(tipo_estructura, datos, etiqueta):
    estructura, tiempo, memoria_actual, memoria_maxima =
medir_tiempo_y_memoria(tipo_estructura, datos)
    nombre = "conjunto" if tipo_estructura == set else "lista"
    print(f"Crear {nombre} de {etiqueta}: {tiempo:.6f} seg, la memoria
actual es: {memoria_actual} bytes y el pico de memoria: {memoria_maxima}
bytes")
    print("_" * 40)
    return estructura
```

- *Problema 1: Cómo funciona la lista (usando for) vs un conjunto (usando set):*
    - Para comparar el rendimiento entre listas y conjuntos al buscar un elemento, definimos dos funciones:
        - buscar_en_lista: recorre la lista de clientes y devuelve True si encuentra un nombre que coincida con el que estamos buscando.
        - buscar_en_conjunto: utiliza la operación in para verificar si el nombre está presente en el conjunto de clientes.

```
# Parte 1: Diferenciar entre buscar en una lista y un conjunto
def buscar_en_lista(nombre, lista_clientes):
    for nombre_cliente in lista_clientes:
        if nombre == nombre_cliente:
            return True

def buscar_en_conjunto(nombre, conjunto_clientes):
    return nombre in conjunto_clientes
```

    - Al momento de la ejecución vamos a medir tiempo y memoria en cada caso, para diferentes cantidades.

- Se define una variable llamada `nombre` que contiene el valor que vamos a buscar en los distintos grupos de datos (listas y conjuntos).
- Se crea una lista llamada grupos_de_datos. Cada elemento de esta lista es una tupla que contiene:
  - una lista de clientes,
  - un conjunto con los mismos clientes, y
  - una etiqueta (string) que indica cuántos elementos contiene ese grupo (por ejemplo, "1000").
- Se recorre cada tupla del grupo de datos. En cada iteración se obtienen:
  - lista_clientes: la lista de nombres de clientes,
  - conjunto_clientes: el mismo grupo de clientes pero en un set (conjunto),
  - etiqueta: el texto que identifica cuántos elementos hay.
- Se llama a la función medir_tiempo_y_memoria pasándole como parámetros:
  - la función buscar_en_lista,
  - el nombre que se quiere buscar,
  - y la lista_clientes.
- Esta función retorna:
  - si el nombre fue encontrado (resultado_lista),
  - cuánto tardó en ejecutarse (tiempo_lista),
  - cuánta memoria usó en total (memoria_actual_lista) y cuál fue el pico (memoria_maxima_lista).
- Después, se imprime el resultado de la búsqueda en la lista, junto con el tiempo y la memoria usados.
- Se repite el mismo proceso, pero ahora con:
  - la función buscar_en_conjunto,
  - el mismo nombre,
  - y el conjunto de clientes.
- Se imprime si el nombre fue encontrado en el conjunto y también se muestran el tiempo de ejecución y la memoria utilizada.

```
nombre = "Lucía Fernández"
```

```
# Creamos un grupo de datos definidos por la cantidad de elementos, por cada
uno hay: lista, conjunto, etiqueta
grupos_de_datos = [
    (lista_clientes_1000, conjunto_clientes_1000, "1000"),
    (lista_clientes_10000, conjunto_clientes_10000, "10000"),
    (lista_clientes_100000, conjunto_clientes_100000, "100000")
]

# Recorremos cada elemento (lista, conjunto, etiqueta) del grupo de datos
for lista_clientes, conjunto_clientes, etiqueta in grupos_de_datos:
    # Medimos la lista de cada uno de los grupos de datos
    resultado_lista, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(buscar_en_lista, nombre,
lista_clientes)
    print(f"{nombre} {'está' if resultado_lista else 'no está'} en la lista
de {etiqueta}")
    print(f"Búsqueda con lista de {etiqueta}: {tiempo_lista:.6f} seg, pico
de memoria: {memoria_maxima_lista} bytes")

    # Medimos los conjuntos de cada uno de los grupos de datos
    resultado_conjunto, tiempo_conjunto, memoria_actual_conjunto,
memoria_maxima_conjunto = medir_tiempo_y_memoria(buscar_en_conjunto, nombre,
conjunto_clientes)
    print(f"{nombre} {'está' if resultado_conjunto else 'no está'} en el
conjunto de {etiqueta}")
    print(f"Búsqueda con conjunto de {etiqueta}: {tiempo_conjunto:.6f} seg,
pico de memoria: {memoria_maxima_conjunto} bytes")

    print("_" * 40)
```

- Como resultado, observamos que la búsqueda en una lista tarda significativamente más que en un conjunto. Además, la ejecución con `for` genera un pico de uso de memoria que no se presenta al utilizar la operación `in` con un `set`.

```
________________________________________
Lucía Fernández está en la lista de 1000
Búsqueda con lista de 1000: 0.000011 seg, pico de memoria: 48
bytes
Lucía Fernández está en el conjunto de 1000
Búsqueda con conjunto de 1000: 0.000004 seg, pico de memoria: 0
bytes

________________________________________
```

```
Lucía Fernández está en la lista de 10000
Búsqueda con lista de 10000: 0.000010 seg, pico de memoria: 48
bytes
Lucía Fernández está en el conjunto de 10000
Búsqueda con conjunto de 10000: 0.000002 seg, pico de memoria: 0
bytes

_____________________________________
Lucía Fernández está en la lista de 100000
Búsqueda con lista de 100000: 0.000010 seg, pico de memoria: 48
bytes
Lucía Fernández está en el conjunto de 100000
Búsqueda con conjunto de 100000: 0.000002 seg, pico de memoria: 0
bytes
```

- A partir de la investigación sobre el comportamiento de los conjuntos (`set`) y las listas (cuyo análisis se presenta más adelante en este documento), decidimos también medir el tiempo y la memoria que consume crear cada estructura.
- Para hacer una comparación justa —es decir, con la misma cantidad de elementos únicos en ambas estructuras—, transformamos nuestras listas originales en listas sin elementos duplicados Esto nos asegura que tanto los conjuntos como las listas que comparamos tengan la misma cantidad de elementos, lo cual es fundamental para que la evaluación sea equitativa:

```python
# Obtener listas sin duplicados para cada tamaño
lista_clientes_unicos_1000 = list(set(lista_clientes_1000))
lista_clientes_unicos_10000 = list(set(lista_clientes_10000))
lista_clientes_unicos_100000 = list(set(lista_clientes_100000))
```

- Ahora si, los medimos:

```python
# Medir memoria para listas sin duplicados
medir_estructura(list, lista_clientes_unicos_1000, "1000 únicos")
medir_estructura(list, lista_clientes_unicos_10000, "10000 únicos")
medir_estructura(list, lista_clientes_unicos_100000, "100000 únicos")

# Medir memoria para sets con los mismos elementos que la lista
medir_estructura(set, lista_clientes_unicos_1000, "1000 únicos")
medir_estructura(set, lista_clientes_unicos_10000, "10000 únicos")
```

```
medir_estructura(set, lista_clientes_unicos_100000, "100000 únicos")
```

- Obtenemos como resultado que el conjunto consume bastante más memoria que la lista:

```
Crear lista de 1000 únicos: 0.000073 seg, la memoria actual es:
856 bytes y el pico de memoria: 856 bytes

_____________________________________________
Crear lista de 10000 únicos: 0.000007 seg, la memoria actual es:
856 bytes y el pico de memoria: 856 bytes

_____________________________________________
Crear lista de 100000 únicos: 0.000006 seg, la memoria actual es:
856 bytes y el pico de memoria: 856 bytes

_____________________________________________
Crear conjunto de 1000 únicos: 0.000014 seg, la memoria actual es:
8408 bytes y el pico de memoria: 10504 bytes

_____________________________________________
Crear conjunto de 10000 únicos: 0.000012 seg, la memoria actual
es: 8408 bytes y el pico de memoria: 10504 bytes

_____________________________________________
Crear conjunto de 100000 únicos: 0.000011 seg, la memoria actual
es: 8408 bytes y el pico de memoria: 10504 bytes
```

Podemos concluir que, aunque los conjuntos (sets) son más rápidos que las listas al momento de buscar elementos, su creación consume más memoria que la de las listas.

- *Problema 2: Cómo funciona la suma al analizar una lista mediante un For vs Sum(lista) vs Recursividad*

- Probaremos hacer sumas de distintas cantidades de números con for, sum y recursividad. Para eso creamos las funciones:
  - bucle for: recorre cada elemento de la lista para ir acumulando el resultado en la variable `suma`. Es un enfoque explícito, que permite entender claramente cómo se construye la suma paso a paso.
  - sum(): utilizamos la función nativa `sum()` de Python, que está optimizada internamente en C para ser muy

rápida y eficiente. Le pasamos como parámetro nuestra lista.
    - recursividad: cada llamada toma el primer elemento de la lista y suma el resultado de aplicar la misma función al resto de la lista. Corta cuando ya no hay elementos en la lista.

```
# Parte 2: Diferenciar entre sumar con bucle, sum o recursividad
def suma_bucle(lista):
    suma = 0
    for cantidad in lista:
        suma += cantidad
    return suma

def suma_con_sum(lista):
    return sum(lista)

def suma_con_recursividad(lista):
    if len(lista) == 0:
        return 0
    else:
        return lista[0] + suma_con_recursividad(lista[1:])
```

- Posteriormente (similar a lo ya visto) agrupamos las listas de productos junto con una etiqueta que indica cuántos elementos tiene cada lista. Esto nos permite recorrerlas fácilmente en un bucle y mostrar resultados más claros.
- Empieza un bucle que va a recorrer cada grupo (lista + etiqueta). Por ejemplo, primero recorre la lista de 100 elementos, luego la de 1000, y así.
- Llamamos a la función medir_tiempo_y_memoria, pasándole suma_bucle y la lista actual. Esta función devuelve:
    - el resultado de la suma,
    - el tiempo que tardó en ejecutarse,
    - cuánta memoria se usó durante la ejecución,
    - y cuánta fue el pico máximo de memoria usada.
- Solo ejecuta la función recursiva si la lista tiene 100 elementos. ¿Por qué? Porque en listas más grandes, la recursividad puede provocar un desbordamiento de pila (`RecursionError`), ya que cada llamada a la función ocupa memoria y Python tiene un límite.

```
# Creamos un grupo de datos definidos por la cantidad de elementos, por cada
uno hay: lista, etiqueta
grupos_de_datos = [
    (lista_cantidad_productos_100, "100"),
    (lista_cantidad_productos_1000, "1000"),
    (lista_cantidad_productos_10000, "10000"),
    (lista_cantidad_productos_100000, "100000")
]

# Recorremos cada elemento (lista, etiqueta) del grupo de datos
for lista_cantidad_productos, etiqueta in grupos_de_datos:
    # Medimos la lista de cada uno de los grupos de datos
    resultado_funcion, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(suma_bucle,
lista_cantidad_productos)
    print(f"El resultado de la suma con bucle es: {resultado_funcion}")
    print(f"Las medidas con {etiqueta} elementos: {tiempo_lista:.6f} seg,
pico de memoria: {memoria_maxima_lista} bytes")

    resultado_funcion, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(suma_con_sum,
lista_cantidad_productos)
    print(f"El resultado de la suma con sum es: {resultado_funcion}")
    print(f"Las medidas con {etiqueta} elementos: {tiempo_lista:.6f} seg,
pico de memoria: {memoria_maxima_lista} bytes")

    #La suma con recursividad la podemos hacer solo con esta cantidad, ya
que con más obtenemos el error de sobrecarga de pila
    if(etiqueta == "100"):
        resultado_funcion, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(suma_con_recursividad,
lista_cantidad_productos)
        print(f"El resultado de la suma con recursividad es:
{resultado_funcion}")
        print(f"Las medidas con {etiqueta} elementos: {tiempo_lista:.6f}
seg, pico de memoria: {memoria_maxima_lista} bytes")

    print("_" * 40)
```

- El resultado podemos verlo acá:

```
El resultado de la suma con bucle es: 313
Las medidas con 100 elementos: 0.000028 seg, pico de memoria: 112
bytes
```

```
El resultado de la suma con sum es: 313
Las medidas con 100 elementos: 0.000009 seg, pico de memoria: 48
bytes
El resultado de la suma con recursividad es: 313
Las medidas con 100 elementos: 0.000283 seg, pico de memoria:
44304 bytes

_____________________________________________
El resultado de la suma con bucle es: 3079
Las medidas con 1000 elementos: 0.000353 seg, pico de memoria: 112
bytes
El resultado de la suma con sum es: 3079
Las medidas con 1000 elementos: 0.000008 seg, pico de memoria: 48
bytes

_____________________________________________
El resultado de la suma con bucle es: 30256
Las medidas con 10000 elementos: 0.004235 seg, pico de memoria:
112 bytes
El resultado de la suma con sum es: 30256
Las medidas con 10000 elementos: 0.000075 seg, pico de memoria: 48
bytes

_____________________________________________
El resultado de la suma con bucle es: 299826
Las medidas con 100000 elementos: 0.048199 seg, pico de memoria:
112 bytes
El resultado de la suma con sum es: 299826
Las medidas con 100000 elementos: 0.000633 seg, pico de memoria:
48 bytes

_____________________________________________
```

Estos resultados demuestran, en primer lugar, que la recursividad presenta limitaciones en cuanto a la cantidad de ejecuciones permitidas, debido al riesgo de desbordamiento de pila. Además, se observa que este enfoque es considerablemente más lento y consume más memoria que las otras alternativas.

Por otro lado, el uso de un bucle (`for`) resulta más eficiente que la recursividad, tanto en tiempo como en consumo de memoria.

Sin embargo, ninguno de los dos métodos supera la eficiencia de la función `sum()`, que se destaca como la opción más rápida y con menor uso de recursos para realizar este tipo de operaciones.

- *Problema 3: Cómo funciona el Slicing ([::-1]) vs reversed() vs bucle manual.*
    - Usamos la función incorporada reversed() de Python, que devuelve un iterador que recorre la lista en orden inverso.
        - reversed(lista) no devuelve una lista, sino un iterador que recorre los elementos desde el final al principio.
        - list() convierte ese iterador en una lista nueva.
        - Se devuelve esa nueva lista con los elementos en orden invertido.
    - Utilizamos la técnica de slicing (corte de listas) para crear una copia invertida de la lista original.
        - lista[::-1] es una notación de slicing que significa: inicio:fin:paso
        - En este caso no se especifica ni inicio ni fin, solo el paso = -1, lo que indica que se recorre la lista de atrás hacia adelante.
        - Se devuelve directamente la nueva lista invertida.
    - Con el bucle:
        - Se crea una lista vacía llamada nueva_lista.
        - range(len(lista)-1, -1, -1) genera los índices desde el último (len(lista)-1) hasta el primero (0), en orden descendente.
        - En cada iteración del bucle: se accede al elemento de la lista original en esa posición (lista[i]), se agrega ese elemento a nueva_lista, al final, se retorna la lista con el orden invertido.

```
# Parte 3: Diferenciar entre invertir orden con reversed, slicing o bucle
def invertir_orden_reversed(lista):
    return list(reversed(lista))

def invertir_orden_slicing(lista):
    return lista[::-1]

def invertir_con_bucle(lista):
    nueva_lista = []
    for i in range(len(lista)-1, -1, -1):
        nueva_lista.append(lista[i])
    return nueva_lista
```

    - Importamos tres listas de pedidos con diferente cantidad de elementos: 1000, 10000 y 100000.

- Usamos la función medir_tiempo_y_memoria para evaluar el rendimiento (tiempo y memoria) de cada método de inversión de lista.
- Se recorre cada lista y aplica:
  - invertir_orden_reversed.
  - invertir_con_bucle.
  - invertir_orden_slicing.
- Imprime los resultados indicando:
  - Tiempo que tardó en invertirse la lista.
  - Pico de memoria alcanzado durante la operación.

```
# Creamos un grupo de datos definidos por la cantidad de elementos, por cada
uno hay: lista, etiqueta
grupos_de_datos = [
    (lista_pedidos_1000, "1000"),
    (lista_pedidos_10000, "10000"),
    (lista_pedidos_100000, "100000")
]

# Recorremos cada elemento (lista y etiqueta) del grupo de datos
for lista_pedidos, etiqueta in grupos_de_datos:
    # Medimos la lista de cada uno de los grupos de datos
    resultado_lista, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(invertir_orden_reversed,
lista_pedidos)
    print(f"El reordenamiento con reversed de una lista con {etiqueta}
elementos conlleva: {tiempo_lista:.6f} seg, pico de memoria:
{memoria_maxima_lista} bytes")

    resultado_lista, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(invertir_con_bucle,
lista_pedidos)
    print(f"El reordenamiento con bucle de una lista con {etiqueta}
elementos conlleva: {tiempo_lista:.6f} seg, pico de memoria:
{memoria_maxima_lista} bytes")

    resultado_lista, tiempo_lista, memoria_actual_lista,
memoria_maxima_lista = medir_tiempo_y_memoria(invertir_orden_slicing,
lista_pedidos)
    print(f"El reordenamiento con slicing de una lista con {etiqueta}
elementos conlleva: {tiempo_lista:.6f} seg, pico de memoria:
{memoria_maxima_lista} bytes")

    print("_" * 40)
```

- Los resultados nos indican lo siguiente:

```
El reordenamiento con reversed de una lista con 1000 elementos
conlleva: 0.000033 seg, pico de memoria: 8104 bytes
El reordenamiento con bucle de una lista con 1000 elementos
conlleva: 0.001046 seg, pico de memoria: 8832 bytes
El reordenamiento con slicing de una lista con 1000 elementos
conlleva: 0.000012 seg, pico de memoria: 8000 bytes

_____________________________________________
El reordenamiento con reversed de una lista con 10000 elementos
conlleva: 0.000121 seg, pico de memoria: 80104 bytes
El reordenamiento con bucle de una lista con 10000 elementos
conlleva: 0.004317 seg, pico de memoria: 85216 bytes
El reordenamiento con slicing de una lista con 10000 elementos
conlleva: 0.000048 seg, pico de memoria: 80000 bytes

_____________________________________________
El reordenamiento con reversed de una lista con 100000 elementos
conlleva: 0.001588 seg, pico de memoria: 800104 bytes
El reordenamiento con bucle de una lista con 100000 elementos
conlleva: 0.053704 seg, pico de memoria: 801024 bytes
El reordenamiento con slicing de una lista con 100000 elementos
conlleva: 0.001489 seg, pico de memoria: 800000 bytes
```

A partir de los resultados obtenidos, podemos concluir que el método más eficiente para invertir listas es el slicing, ya que en todos los tamaños evaluados fue el que menor tiempo y menos memoria consumió. El siguiente, por poco,es el método `reversed`, que también es rápido y bastante eficiente en memoria. En cambio, el bucle manual resulta ser el menos eficiente, ya que requiere más tiempo y memoria que los otros dos métodos.

• *Análisis Teórico de la Propuesta:*

Para esta parte nos vamos a centrar en la evaluación de la Complejidad Algorítmica con relación a la eficiencia en tiempo y memoria requerida para ejecutar los códigos propuestos. Primero presentaremos la comparación de proyección de tiempos y luego la proyección de uso de memoria.

Antes de analizar cada porción de código, es relevante entender cómo funcionan las hipótesis gracias al Big O. Para esto, a modo de ejemplo tomamos un fragmento para mostrar cómo lo evaluamos:

```
def buscar_en_lista(nombre, lista_clientes):
    for nombre_cliente in lista_clientes:
        if nombre == nombre_cliente:
            return True
     return False
```

Como veremos más adelante, cuando realizamos búsquedas a través de bucles tenemos una "mejor opción" (cuando el elemento está primero), la "peor opción" (cuando el elemento está al final o directamente no se encuentra en la lista) y casos intermedios (cuando el elemento está en una posición aleatoria en la lista).

Para nuestro análisis de Big O, nos vamos a basar en el peor escenario, donde ocurrirá lo siguiente:

- El bucle recorre todos los elementos (**n veces**).
- Cada iteración incluye:
  Comparación (if nombre == nombre_cliente): **1 operación.**
  No entra al return True (porque es el peor caso donde el elemento no está o está al final).
- Al final, va a haber un return False: +**1 operación.**

Para una mejor visualización, lo podemos presentar de la siguiente manera:

```
def buscar_en_lista(nombre, lista_clientes):
    for nombre_cliente in lista_clientes:  # O(n) en peor caso
        if nombre == nombre_cliente:       # O(1) por comparación
            return True                    # O(1) (solo si se
ejecuta)
   return False                           # O(1) (peor caso)
```

Entonces, el total se puede calcular como: n * 1 + 1

En Big O, las constantes se descartan, por eso n + 1 se simplifica a O(n) = n + 1 → **O(n).**

Esto quiere decir que el análisis va a estar sujeto a la cantidad de n y que tendrá una forma lineal.

Como información adicional podemos decir que en el mejor escenario (cuando el elemento está primero en la lista), este análisis puede pensarse así:

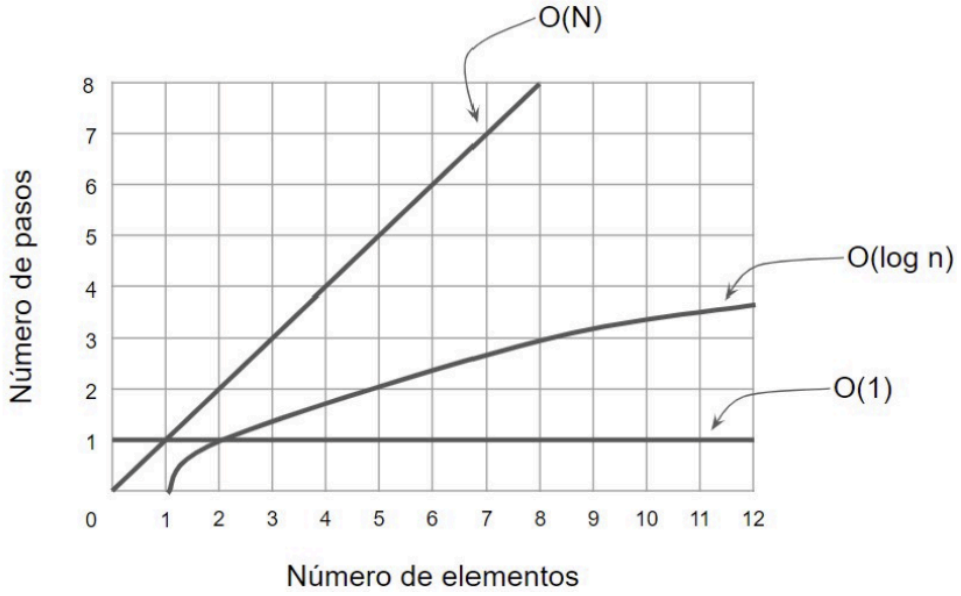
Como solo será necesario ejecutarse una vez, habrá:

- 1 iteración.
- 1 comparación.
- 1 return True.

Total: 1 * 1 + 1 = 2 → **O(1).**

En este caso, al no ser dependiente de n (porque se evalúa una vez) actúa como una constante.

Podemos ver en la siguiente imagen cómo se ven las asíntotas del Big O para O(1), O(n) y O (log n) a modo de referencia:



Proyección de tiempo de resultados:

- *Problema 1: Como funciona la lista (usando for) vs un conjunto (usando set):*

Cuando se usa un bucle para buscar un elemento, en este caso mediante un **For**, estamos recorriendo una lista en el orden en el que está armada.

El mejor escenario es que el ítem buscado se encuentre en la posición 0 (que sería equivalente a O(1)) y el peor escenario es que el ítem no se encuentre en la lista. En este último caso, para llegar a esta conclusión, será necesario recorrer toda la lista completa y recién ahí se podrá decir que efectivamente ese elemento no pertenece a esa lista (equivalente en tiempo a O(n)).

Sin embargo, cuando analizamos este caso mediante un **Set()**, es decir, evaluamos los elementos del grupo como un conjunto, no es necesario hacer un recorrido lineal (un ítem a la vez), si no que la función tiene acceso a los elementos de forma directa, lo que se traduce en mayor eficiencia a nivel tiempo de ejecución.

Los conjuntos en Python están implementados con tablas hash, una estructura que permite acceder a los elementos en tiempo constante(O (1)), independientemente del tamaño del conjunto.

Esto implica que en orden de eficiencia para buscar un elemento, debería ser más eficiente el SET que el FOR, ya que el segundo se ve afectado directamente por la cantidad de veces que haya que iterar (n) dependiendo de la cantidad de elementos contenidos en la lista.

Sin embargo, es importante hacer algunas aclaraciones respecto al comportamiento de Set(), ya que al momento de crear las tablas hash, deja de ser constante y pasa a ser O(n). Esto sucede porque el conjunto se arma siguiendo estos pasos:

- Se calcula el hash de cada elemento (que es un proceso que transforma un dato en un valor numérico fijo llamado hash, que actúa como una "huella digital" única para ese elemento) —> (O(1) por elemento).
- Se inserta en la tabla hash —> (O(1) por elemento en promedio, pero puede requerir redimensionamiento y manejo de colisiones).

- Este circuito hay que repetirlo para los n elementos —> el tiempo total es O(n).

Para poder comparar estos tiempos con la versión de este código en listas, hicimos una función que creara listas con nombres únicos (es decir, donde no se repita la misma combinación de nombre y apellido).

- *Problema 2: Como funciona la suma al analizar una lista mediante un For vs Sum(lista) vs Recursividad*

El bucle **For,** similar al caso anterior, va a recorrer la lista posición a posición analizando cada elemento. El valor de ese elemento se irá guardando y sumando a una variable que llamaremos suma, donde se irán agregando sistemáticamente estos valores hasta llegar al final del bucle.

En cuanto a la función nativa de Python conocida como **Sum()**, analiza posición a posición de forma optimizada y da el resultado directamente.

Por último, el método de la **recursividad** consiste en crear una función que se llame a sí misma tantas veces como sea necesario hasta que se cumpla una condición que actúa como corte. En este caso, la condición es que no haya más elementos en la lista.

En estos tres casos, el tiempo necesario para terminar la ejecución va a estar estrechamente relacionado con la cantidad de elementos en la lista a evaluar, por lo cual podemos pensar que el consumo de tiempo será O(n).

Sin embargo, podemos predecir que la opción Sum() será más eficiente por tener que iterar sin ejecutar otras líneas de código cada vez, además de estar implementado en C.

Por esto también será más rápida que un bucle for en Python puro (creado manualmente) y habrá aún una mayor diferencia con relación a la recursividad que requiere muchos recursos para volver a ejecutar una función repetidas veces.

- *Problema 3: Cómo funciona el Slicing ([::-1]) vs reversed() vs bucle manual.*

El **slicing** en este caso está pensado para ir analizando la lista de atrás hacia adelante (esto está denotado por el ":-1", donde el negativo muestra el sentido en el que se analiza y el 1 es el paso). Esto quiere decir que recorre la lista y va creando una copia (también conocida como shallow copy) de la secuencia original, pero invertida.

El **reversed()** es una función nativa de Python que retorna un iterador a la vez al recorrer una secuencia en orden inverso. Por esto, es necesario crear una variable con una lista nueva donde se pueda guardar el resultado obtenido mediante un list(reversed(lista_original)).

Lo último que probamos fue un bucle usando un **For** (similar al caso del Problema 1). En cada iteración se fue analizando de a un elemento y se hizo el recorrido desde el último ítem de la lista hasta el primero (similar a lo descrito en el slicing). Se evaluó el largo de longitud de la lista (mediante un len()) y se le restó 1 (para que no quedara fuera de rango, ya que las posiciones en las listas empiezan en 0 por lo cual, de lo contrario, estaría faltando un elemento); como punto para finalizar el recorrido pusimos -1, ya que python no considera el extremo y queríamos que terminara cuando estuviera en la posición 0; y pusimos -1 en el paso, ya que queríamos que analizara un elemento a la vez, de atrás hacia adelante.

En los tres casos, el consumo de tiempo en notación Big O se puede pensar como O(n), ya que va a depender de la cantidad de elementos de la lista, como mencionamos antes.

Se puede presumir que el slicing será la mejor alternativa ya que está implementado directamente en C en el núcleo de Python (CPython), lo que lo hace mucho más eficiente que cualquier código en Python puro.

El reversed() también es una función nativa de Python como dijimos anteriormente, pero debería ser menos eficiente ya que devuelve un iterador a la vez (en vez de la lista completa) y luego ese dato debe guardarse en una lista nueva.

Respecto al bucle manual, el tiempo debería ser mayor, ya que hay que evaluar posición a posición de la lista y ejecutar mayor cantidad de instrucciones.

¿Y qué sucede con los casos anteriores respecto al uso de memoria?

- *Problema 1: Como funciona la lista (usando for) vs un conjunto (usando set):*

El bucle **For** respecto a la búsqueda en una lista tiene un consumo de memoria adicional de O(1), ya que solo recorre la lista original sin crear estructuras intermedias. Es decir, no almacena datos extra y evalúa cada elemento secuencialmente hasta encontrar (o no) el valor buscado.

Cómo mencionamos anteriormente, la función **Set()** con lo que respecta a la búsqueda analiza el conjunto ya armado como un bloque accediendo rápidamente al elemento. No se requiere almacenamiento extra durante la búsqueda, por lo cual también es O(1) (constante).

¿Pero qué sucede con relación a la memoria cuando se están creando las tablas hash?

Al usar **Set()**, la memoria requerida para crear una tabla hash es O(n), donde n es el número de elementos a almacenar. Además, hay que tener en cuenta que la función tiene que seguir estos pasos:

- Reservar espacio en memoria para todos los elementos.
- Calcular el hash de cada elemento y guardarlo en una posición específica.
- Manejar colisiones (cuando dos elementos tienen el mismo hash), lo que puede requerir espacio adicional.

Podemos predecir entonces que el consumo de memoria será alto en comparación con la lista, ya que los conjuntos requieren crear la tabla hash y dejar espacios vacíos para evitar colisiones.

- *Problema 2: Como funciona la suma al analizar una lista mediante un For vs Sum(lista) vs Recursividad*

El bucle **For** tiene una sola variable acumuladora, por lo cual ocupa memoria constante O (1);

La función **Sum()** es similar en ese sentido al bucle For (también O (1)), pero está optimizado en C. En este caso Python usa un acumulador como con el For, pero sin el overhead de interpretación de bytecode (que es la sobrecarga computacional y de tiempo asociada con la interpretación del código de bytes por parte de una máquina virtual o un intérprete).

La **recursividad** consume memorial lineal O (n), debido a la pila de llamadas recursivas. Cada llamada añade un marco a la pila, que es como un bloque de memoria temporal que se crea cada vez que se llama a una función en Python y que guarda: las variables locales de esa función; los argumentos que se le pasaron; dónde debe volver cuando termine (puntero de retorno).

En conclusión, Sum () y For son óptimos en memoria y se debería evitar el uso de la recursión por su alto consumo

- *Problema 3: Cómo funciona el Slicing ([::-1]) vs reversed() vs bucle manual.*

El **slicing** ocupa espacio de memoria adicional (O (n)), ya que genera una copia completa de la lista.

La función **reversed()** es eficiente en memoria (O (1), constante) mientras se use como iterador, ya que no almacena la secuencia completa. Sin embargo, en este caso como queremos obtener una lista invertida, el resultado también va a estar sujeto a n, por lo cual el consumo de memoria es O (n).

El **bucle manual** también tiene consumo de memoria lineal (O (n)) ya que crea una nueva lista, con la desventaja de que es deficiente respecto al tiempo requerido en comparación con las opciones anteriores.

Metodología Utilizada

En el análisis práctico trabajamos sobre varios puntos clave:

- Creamos algoritmos aplicados a problemas reales.
- Hicimos tests con distintos casos, incluyendo casos extremos, para verificar que el algoritmo se mantenga robusto ante entradas inesperadas.

- Medimos el tiempo de ejecución de cada una de las soluciones usando el módulo time.
- Verificamos el uso de memoria con la herramienta tracemalloc, observando tanto el consumo actual como el pico de uso.

En el análisis teórico vimos los siguientes puntos:

- Proyectamos cuales estructuras serían más eficientes a nivel tiempo y requerimiento de memoria
- Utilizamos lo aprendido respecto al análisis de Complejidad Algorítmica.
- Desarrollamos cómo varía la eficiencia de la función Set() respecto al punto de ejecución (cuando se está creando la tabla hash vs. cuando se está realizando una búsqueda).
- Analizamos por qué funciones como Sum() o Reversed() pueden ser mejores o peores que otras alternativas teniendo en cuenta el caso específico en el que se implementa.

Resultados Obtenidos

Para cerrar este proyecto, tomamos los resultados obtenidos al ejecutar las distintas porciones de código y comparamos estos valores entre sí.

Por un lado, evaluamos cómo se comportó un mismo código al variar la cantidad de elementos de entrada (tanto a nivel tiempo como a nivel memoria).

Por otro lado, comparamos los tiempos y luego el uso de memoria entre la ejecución de los distintos códigos para poder determinar cuál era el más eficiente en cada caso.

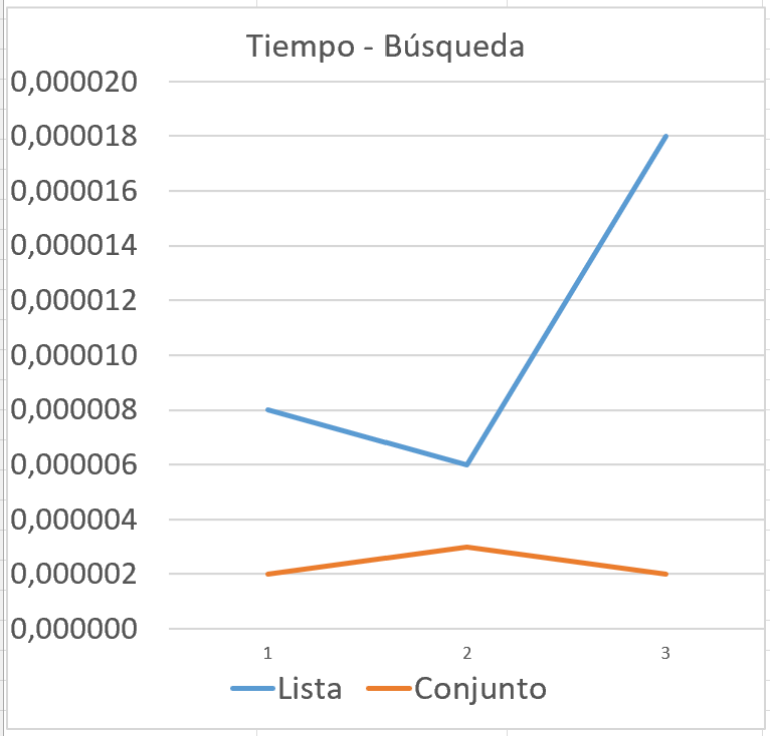
- *Problema 1: Como funciona la lista (usando for) vs un conjunto (usando set):*

Comparación de Búsqueda en Listas / Conjuntos:

Búsqueda de cliente con For (en lista)		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000008	48
10000	0,000006	48
100000	0,000018	48
Búsqueda de cliente con Set() (en conjunto)		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000002	0
10000	0,000003	0
100000	0,000002	0

Tiempos de Búsqueda:

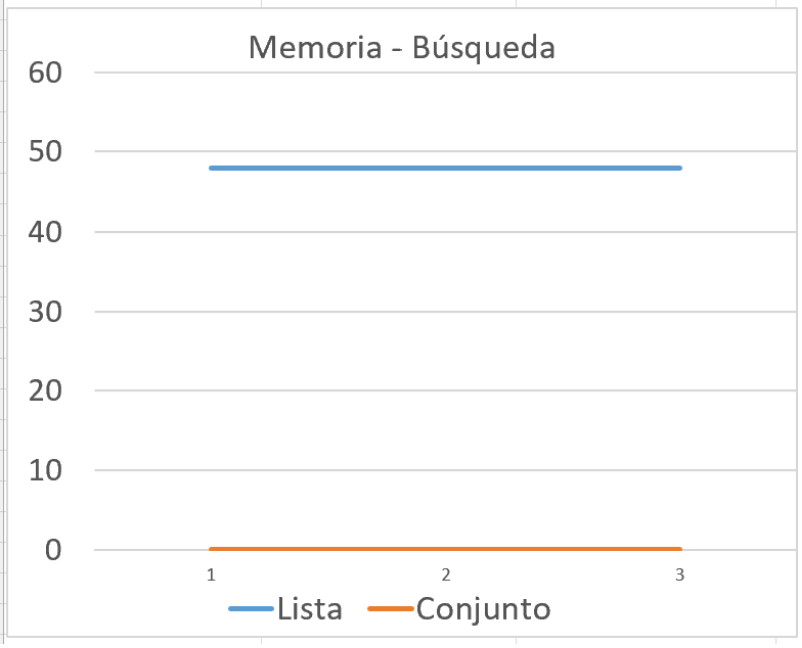
Elementos	Lista	Conjunto
1000	0,000008	0,000002
10000	0,000006	0,000003
100000	0,000018	0,000002



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

Memoria en búsqueda:

Elementos	Lista	Conjunto
1000	48	0,000000
10000	48	0,000000
100000	48	0,000000



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

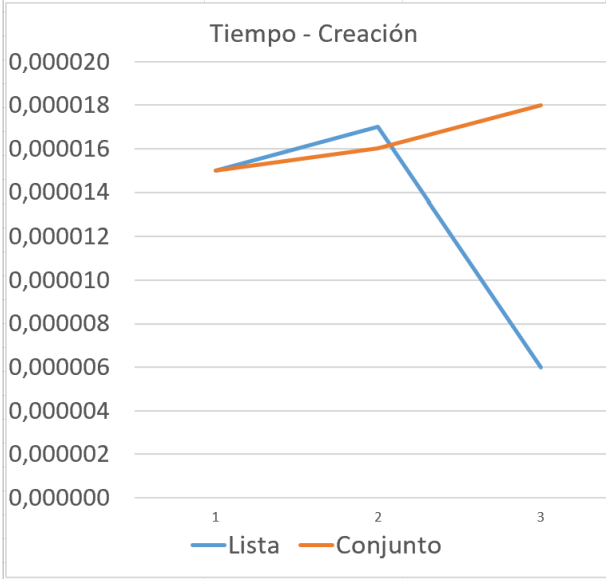
Comparación de Creación de Listas / Conjuntos:

Crear Lista con Nombres Únicos		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000015	856
10000	0,000017	856
100000	0,000006	856

Crear Conjunto con Nombres Únicos		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000015	10504
10000	0,000016	10504
100000	0,000018	10504

Tiempos de Creación:

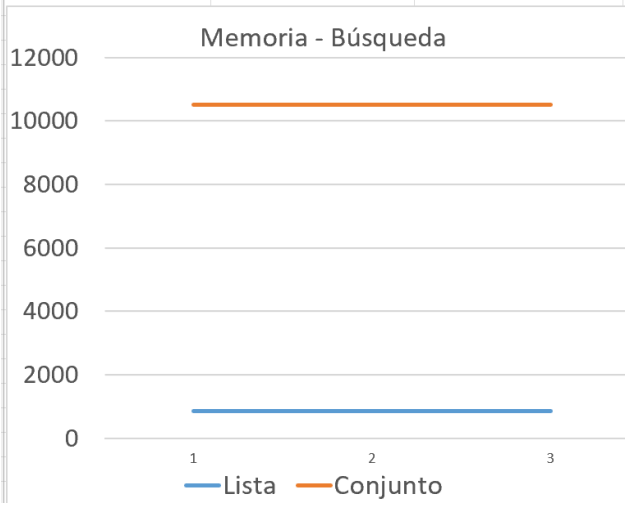
Elementos	Lista	Conjunto
1000	0,000015	0,000015
10000	0,000017	0,000016
100000	0,000006	0,000018



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

Memoria en Creación:

Elementos	Lista	Conjunto
1000	856	10504
10000	856	10504
100000	856	10504



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

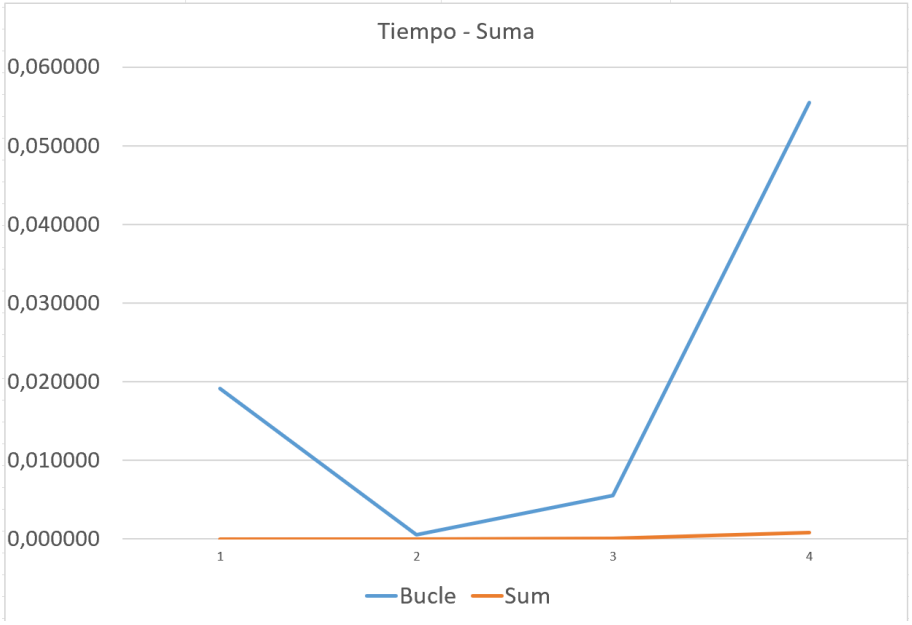
- *Problema 2: Como funciona la suma al analizar una lista mediante un For vs Sum(lista) vs Recursividad*

Análisis individual según código ejecutado:

Suma con Bucle		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
100	0,019156	112
1000	0,000522	112
10000	0,005550	112
100000	0,055495	112
Suma con Sum		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
100	0,000014	48
1000	0,000010	48
10000	0,000092	48
100000	0,000809	48
Suma con Recursividad		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
100	0,002815	44304
1000	---	---
10000	---	---
100000	---	---

Comparación de tiempos:

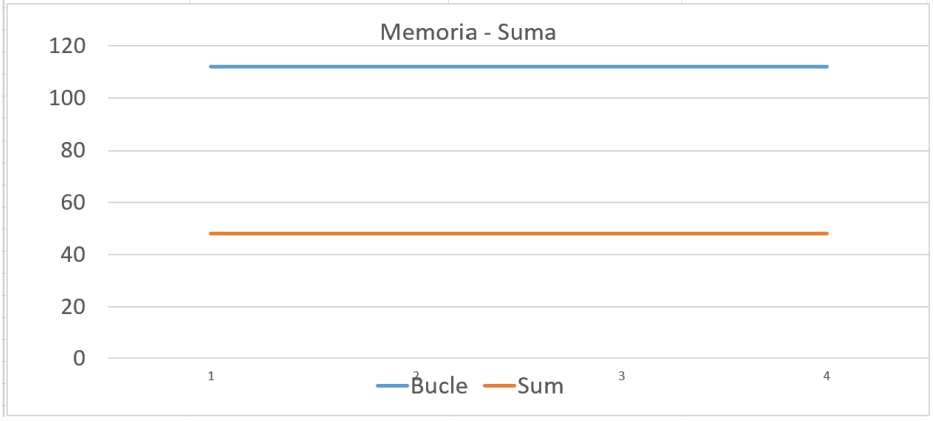
Elementos	Bucle	Sum	Recursividad
100	0,019156	0,000014	0,002815
1000	0,000522	0,000010	
10000	0,005550	0,000092	
100000	0,055495	0,000809	



**Siendo en X: 1 = 100 elementos; 2 = 1000; 3 = 10000; 4 = 100000*

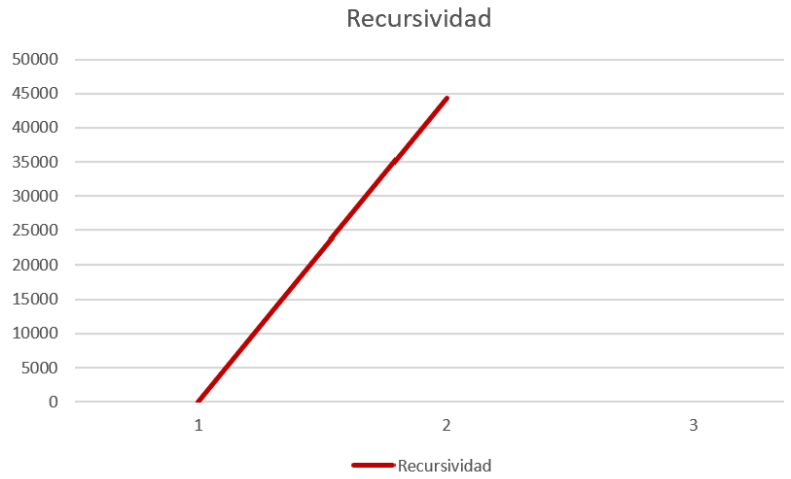
Comparación de memoria de Bucle Vs. Sum

Elementos	Bucle	Sum	Recursividad
100	112	48	44304
1000	112	48	
10000	112	48	
100000	112	48	



**Siendo en X: 1 = 100 elementos; 2 = 1000; 3 = 10000; 4 = 100000*

Vemos como el uso de memoria de Bucle y Sum se mantienen constantes mientras que con recursividad irá crecimiento rápidamente (por estar sujeta a n se considera lineal) y habrá error de ejecución. La función tenderá a verse como la siguiente:



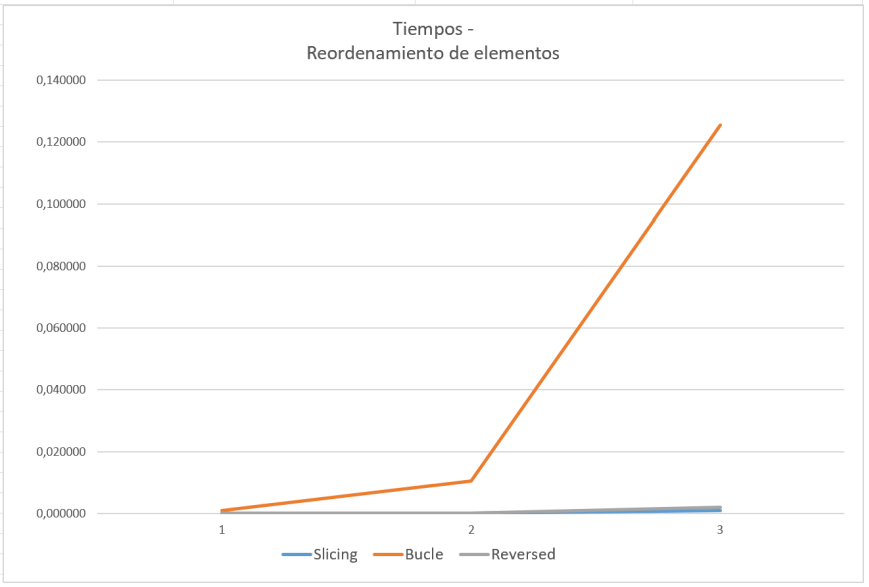
- *Problema 3: Cómo funciona el Slicing ([::-1]) vs reversed() vs bucle manual.*

Análisis individual según código ejecutado:

Reordenamiento con Reversed		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000000	8104
10000	0,000000	80104
100000	0,001896	800104
Reordenamiento con Bucle		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000998	8832
10000	0,010539	85216
100000	0,125559	801024
Reordenamiento con Slicing		
Elementos	**Tiempo de Ejecución**	**Pico de Memoria**
1000	0,000000	8000
10000	0,000000	80000
100000	0,000954	800000

Comparación de tiempos:

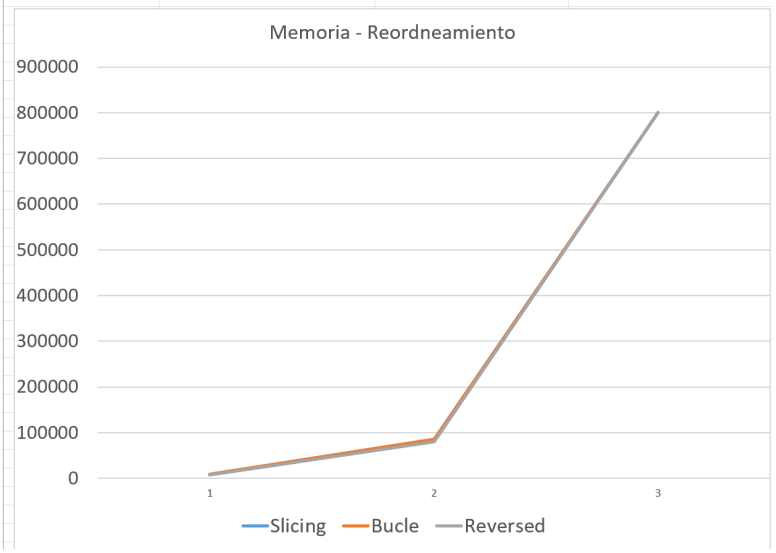
Elementos	Slicing	Bucle	Reversed
1000	0,000000	0,000998	0,000000
10000	0,000000	0,010539	0,000000
100000	0,000954	0,125559	0,001896



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

Comparación de memoria:

Elementos	Slicing	Bucle	Reversed
1000	8000	8832	8104
10000	80000	85216	80104
100000	800000	801024	800104



**Siendo en X: 1 = 1000 elementos; 2 = 10000; 3 = 100000*

Como vemos al analizar los números obtenidos mediante la función medir_tiempo_y_memoria y medir_estructura y contrastarlos con lo pensado a nivel teórico, podemos ver que hay una correlación entre ambos criterios.

Además, aunque algunos valores no crecen o decrecen de forma lineal *(atribuimos en parte estas oscilaciones al uso de la función time que puede presentar pequeñas alteraciones en los resultados)*, se puede notar que hay bloques de código más eficientes con respecto a otros.

También, como anticipamos al inicio, podemos ver que al tratar con volúmenes de entrada pequeños a veces parece haber una diferencia ínfima entre las distintas ejecuciones de bloques de código. Sin embargo, los márgenes de tiempo/memoria ganados cuando hay procesamiento de muchos datos sí son significativos, por lo cual este análisis es fundamental.

Como resumen final, quisimos mostrar de forma más sencilla nuestra percepción sobre la eficiencia de los distintos códigos según lo analizado de forma empírica y teórica:

RESULTADO FINAL	Tiempo	Memoria
Suma con Bucle	✅✅	✅✅✅
Suma con Sum	✅✅✅	✅✅✅
Suma con Recursividad	❌	❌
RESULTADO FINAL	**Tiempo**	**Memoria**
Reordenamiento con Reversed	✅✅	✅✅
Reordenamiento con Bucle	✅	✅
Reordenamiento con Slicing	✅✅✅	✅✅
RESULTADO FINAL	**Tiempo**	**Memoria**
Búsqueda Lista Nombres Únicos	✅✅	✅✅
Búsqueda Conjunto Nom. Únicos	✅✅✅	✅✅✅
RESULTADO FINAL	**Tiempo**	**Memoria**
Creación Lista Nombres Únicos	✅	✅✅
Creación Conjunto Nom. Únicos	✅	✅

Conclusión

A lo largo del trabajo nos propusimos analizar distintos aspectos de los algoritmos, tanto desde una mirada teórica como empírica. Esto nos permitió entender no solo cómo deberían comportarse idealmente, sino también cómo se comportan realmente al momento de implementarlos.

Pudimos ver que medir el tiempo o el uso de memoria no siempre cuenta toda la historia: hay algoritmos que, si bien parecen simples, se comportan de forma muy distinta dependiendo de cómo estén construidos o del tipo de estructura que usen. Así como también en términos prácticos, dependiendo donde corramos nuestros algoritmos, los números son variables. Por lo tanto, la complementación de esta investigación con herramientas de análisis teórico es esencial.

Queda claro, entonces, que combinar teoría y práctica es clave para lograr soluciones más eficientes y robustas.

Bibliografía

- https://es.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/verifying-an-algorithm#:~:text=Un%20an%C3%A1lisis%20%22emp%C3%ADrico%22%20es%20el,y%20ejecutado%20en%20una%20computadora.

- https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011
- https://chat.deepseek.com/a/chat/ - https://chatgpt.com/ - Información general

Anexos

- https://github.com/andyjayala/TP-Final - Repositorio en Github
- https://drive.google.com/drive/folders/1wRg2QbQn6peRehYQuNvTYDXk5RJjlYmH?usp=sharing - Carpeta con todos los archivos
- https://drive.google.com/file/d/1prxS1fvDpTXq0edL-nYUSIlfLl_b7vcC/view?usp=sharing - Video explicativo