Análisis de Algoritmos

Alumnos:
Matías Farfán **–** m.farfan06@gmail.com
Lucas Desiderio Silva **–** lucasdesiderio000@gmail.com

Materia: Programación I.

Profesor: AUS Bruselario, Sebastián.

Fecha de Entrega: 9 de junio de 2025.

Índice:

1. Introducción

El análisis de algoritmos es un componente esencial en el estudio de la informática, ya que permite evaluar la eficiencia de las soluciones desarrolladas para resolver problemas computacionales. Analizar el comportamiento de un

algoritmo en cuanto a tiempo de ejecución y uso de memoria resulta clave en sistemas donde la optimización de recursos es fundamental.

El objetivo de este trabajo es aplicar conceptos teóricos y empíricos del análisis algorítmico, combinando el estudio matemático con la experimentación práctica. Para ello, se implementarán y compararán distintos algoritmos utilizando el lenguaje Python. Se emplearán herramientas como el módulo time para medir tiempos de ejecución (Sweigart, 2015) y se tomarán como base fundamentos de diseño y estructura algorítmica (Downey, 2015).

2. Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es una secuencia finita y bien definida de pasos diseñada para resolver un problema o llevar a cabo una tarea específica. Se caracteriza por ser preciso, sistemático y ejecutable tanto por una persona como por una computadora. Downey (2015) resalta que los algoritmos permiten automatizar procesos sin requerir razonamiento complejo, lo que los convierte en la base de la programación.

¿Por qué analizar algoritmos?

El análisis de algoritmos permite estimar su comportamiento antes de ser ejecutados, lo que ayuda a predecir su eficiencia en diferentes contextos de uso. Esta práctica es indispensable para resolver problemas de manera óptima y escalable en la industria del software, especialmente al trabajar con grandes volúmenes de datos (Downey, 2015, p. 159).

Tipos de análisis algorítmico

Análisis teórico: Se basa en una evaluación matemática del algoritmo para determinar su comportamiento en función del tamaño de entrada n. En el enfoque teórico se analiza la cantidad de pasos computacionales mediante una función de complejidad (*Análisis Teórico de Algoritmos*, p. 2).

Análisis empírico: Se realiza mediante la ejecución del algoritmo, midiendo variables como el tiempo de respuesta o el consumo de recursos. Sweigart (2015) muestra cómo utilizar el módulo time de Python para registrar el tiempo que tarda una función en ejecutarse (p. 363).

Notación Big-O

La notación Big-O describe el comportamiento asintótico de un algoritmo, es decir, cómo cambia su rendimiento a medida que crece el tamaño de la entrada. Permite comparar algoritmos ignorando constantes y enfocándose en el crecimiento dominante (*Notación Big-O*, p. 1).

Notación	Complejidad	Ejemplo
O(1)	Constante	Acceso directo a un elemento de lista
O(n)	Lineal	Recorrido de todos los elementos
O(n²)	Cuadrática	Bucles anidados
O(log n)	Logarítmica	Búsqueda binaria
O(n log n)	Lineal-logarítmica	Merge Sort, Quick Sort

Complejidad y estructuras de control

Las estructuras de control que componen un algoritmo influyen directamente en su complejidad. Por ejemplo, los bucles y las decisiones condicionales determinan cuántas operaciones se realizan, lo que impacta en su eficiencia general (*Análisis Teórico de Algoritmos*, s.f.).

3. Caso Práctico

Ejemplo 1: Búsqueda Lineal vs Búsqueda Binaria

Descripción

Este ejemplo compara dos algoritmos utilizados para buscar un valor dentro de una lista:

La búsqueda lineal recorre la lista elemento por elemento hasta encontrar el objetivo.

La búsqueda binaria, en cambio, parte la lista ordenada por la mitad en cada paso, descartando la mitad que no contiene el valor buscado. Esta diferencia en la estrategia tiene un impacto significativo en la eficiencia cuando se trabaja con grandes volúmenes de datos.

Código Python

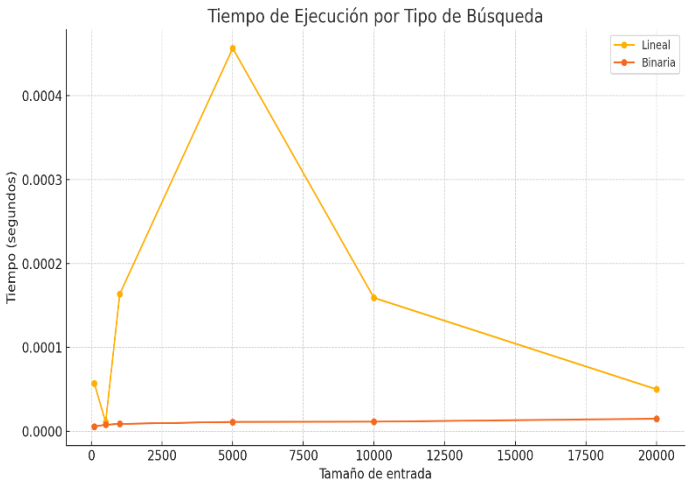
```
import time
import random

#Función de búsqueda lineal
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):          #Recorre la lista desde el inicio hasta el final
        if lista[i] == objetivo:         #Si encuentra el objetivo, devuelve su índice
```

```
            return i
    return -1                       #Si no lo encuentra, devuelve -1

#Función de búsqueda binaria (requiere lista ordenada)
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:                    #Mientras haya elementos para buscar
        medio = (inicio + fin) // 2         #Calcula el punto medio
        if lista[medio] == objetivo:        #Si lo encuentra, retorna su posición
            return medio
        elif lista[medio] < objetivo:       #Si el valor medio es menor al objetivo
            inicio = medio + 1              #Busca en la mitad derecha
        else:
            fin = medio - 1                 #Busca en la mitad izquierda
    return -1                               #Si no lo encuentra, retorna -1

#Tamaños de entrada a evaluar
tamaños = [100, 500, 1000, 5000, 10000, 20000]

#Evaluamos una única vez por cada tamaño de lista
for n in tamaños:
    lista = list(range(n))                  #Lista ordenada de 0 a n-1
    objetivo = random.choice(lista)         #Elegimos un número al azar dentro de
la lista

    print(f"\nTamaño de entrada: {n}")

    #Medición de búsqueda lineal
    inicio = time.time()
    busqueda_lineal(lista, objetivo)
    fin = time.time()
    tiempo_lineal = fin - inicio
    print(f"  Tiempo búsqueda lineal:  {tiempo_lineal:.8f} segundos")

    #Medición de búsqueda binaria
    inicio = time.time()
    busqueda_binaria(lista, objetivo)
    fin = time.time()
    tiempo_binaria = fin - inicio
```

```
print(f"  Tiempo búsqueda binaria: {tiempo_binaria:.8f} segundos")
```

Resultados obtenidos:

Tamaño entrada	Lineal en segundos	Binaria en segundos
100	0,00005698	0,00000572
500	0,00001097	0,00000787
1000	0,00016356	0,00000882
5000	0,00045633	0,00001121
10000	0,00015903	0,00001144
20000	0,00005007	0,00001502



Análisis

En este experimento, se evaluó el tiempo de ejecución de los algoritmos de búsqueda lineal y binaria en listas ordenadas de diferentes tamaños (desde 100 hasta 20.000 elementos). Para cada tamaño, se ejecutó una única búsqueda de un elemento elegido al azar dentro de la lista.

Los resultados muestran que la **búsqueda binaria es notablemente más rápida a medida que crece la cantidad de elementos**, ya que divide el espacio de búsqueda a la mitad en cada paso, lo que se traduce en una complejidad logarítmica **O(log n)**. En cambio, la **búsqueda lineal recorre uno por uno cada elemento**, lo cual implica una complejidad **O(n)**, haciendo que su rendimiento se deteriore proporcionalmente al tamaño de la lista.

Es importante remarcar que **la búsqueda binaria requiere que la lista esté ordenada** para funcionar correctamente. En este caso, se garantiza ese orden desde el inicio con list(range(n)). En situaciones reales donde los datos no

estén ordenados, la búsqueda binaria podría fallar o arrojar resultados incorrectos. En esos casos, la búsqueda lineal sigue siendo una alternativa funcional, aunque más lenta.

Este análisis práctico reafirma cómo el conocimiento de las características de un algoritmo y del tipo de datos que se manejan resulta clave para elegir la mejor solución posible.

Ejemplo 2: Bubble Sort vs Quick Sort

Descripción

En este ejemplo se comparan dos algoritmos de ordenamiento:

- **Bubble Sort**: un método simple que compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto.
- **Quick Sort**: un método más avanzado que divide y conquista, eligiendo un elemento como pivote para ordenar los demás a su alrededor.

Ambos algoritmos logran ordenar una lista, pero la forma en que lo hacen y el tiempo que tardan es muy distinto.

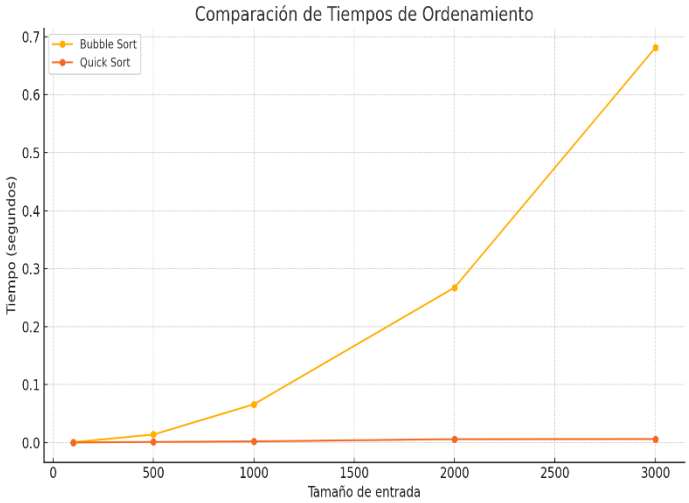
Código Python

```
import time
import random

#Algoritmo de ordenamiento Bubble Sort
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):                #Recorre la lista desde el inicio hasta el penúltimo elemento no ordenado
            if lista[j] > lista[j + 1]:              #Si el elemento actual es mayor que el siguiente
                lista[j], lista[j + 1] = lista[j + 1], lista[j]  #Intercambia los elementos
    return lista

#Algoritmo de ordenamiento Quick Sort (versión recursiva)
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]                                #Se elige el primer elemento como pivote
    menores = [x for x in lista[1:] if x <= pivote]     #Lista de elementos menores o iguales al pivote
    mayores = [x for x in lista[1:] if x > pivote]      #Lista de elementos mayores al pivote
```

```
        return quick_sort(menores) + [pivote] + quick_sort(mayores)    #Ordena recursivamente y combina

#Tamaños de entrada a evaluar
tamaños = [100, 500, 1000, 2000, 3000]

#Se mide una sola vez por cada tamaño
for n in tamaños:
    lista_original = random.sample(range(n * 10), n)     #Lista aleatoria de n elementos sin repetir

    lista1 = lista_original.copy()                  #Copia para Bubble Sort
    lista2 = lista_original.copy()                  #Copia para Quick Sort

    print(f"\nTamaño de entrada: {n}")

    #Medición de tiempo para Bubble Sort
    inicio = time.time()
    bubble_sort(lista1)
    fin = time.time()
    tiempo_bubble = fin - inicio
    print(f"  Tiempo Bubble Sort: {tiempo_bubble:.8f} segundos")

    #Medición de tiempo para Quick Sort
    inicio = time.time()
    quick_sort(lista2)
    fin = time.time()
    tiempo_quick = fin - inicio
    print(f"  Tiempo Quick Sort:  {tiempo_quick:.8f} segundos")
```

Resultados obtenidos:

Tamaño entrada	Bubble Sort	Quick Sort
100	0,00090551	0,00035119
500	0,01411223	0,00141144
1000	0,06621647	0,0022943
2000	0,26749372	0,00614905
3000	0,68082523	0,00638366



Análisis

En este experimento se midió el rendimiento de los algoritmos **Bubble Sort** y **Quick Sort** aplicados a listas desordenadas de distintos tamaños (100, 500, 1000, 2000 y 3000 elementos). Para cada tamaño, se generó una lista aleatoria sin elementos repetidos, y ambos algoritmos trabajaron sobre copias idénticas de esa lista.

Los resultados confirmaron que **Quick Sort es mucho más eficiente que Bubble Sort**, especialmente a medida que crece el tamaño de la entrada. Esto se debe a que Quick Sort divide recursivamente la lista usando un pivote, lo que en el caso promedio y en el mejor escenario le permite operar en tiempo **O(n log n)**. En contraste, **Bubble Sort realiza múltiples comparaciones e intercambios en cada pasada**, y su rendimiento es cuadrático, es decir, **O(n²)**.

Aunque Bubble Sort es útil para enseñar los conceptos básicos de ordenamiento, **no se recomienda su uso en aplicaciones reales con grandes volúmenes de datos**, ya que su ineficiencia se vuelve evidente rápidamente. Quick Sort, por su parte, es ampliamente utilizado en entornos productivos por su velocidad y versatilidad.

Este análisis refuerza la importancia de conocer la complejidad algorítmica y su impacto real en la ejecución, especialmente cuando se trabaja con estructuras de datos grandes.

Ejemplo 3: Fibonacci Recursivo vs Memoizado

Descripción

La secuencia de Fibonacci es una sucesión numérica donde cada número es la suma de los dos anteriores. Este ejemplo compara dos formas de calcularla:

- **Fibonacci recursivo simple**: llama repetidamente a la misma función, recalculando los mismos valores muchas veces.

- **Fibonacci memoizado**: guarda los resultados intermedios para no repetir cálculos.

La diferencia de rendimiento entre ambos enfoques es muy notoria cuando el número a calcular es alto.

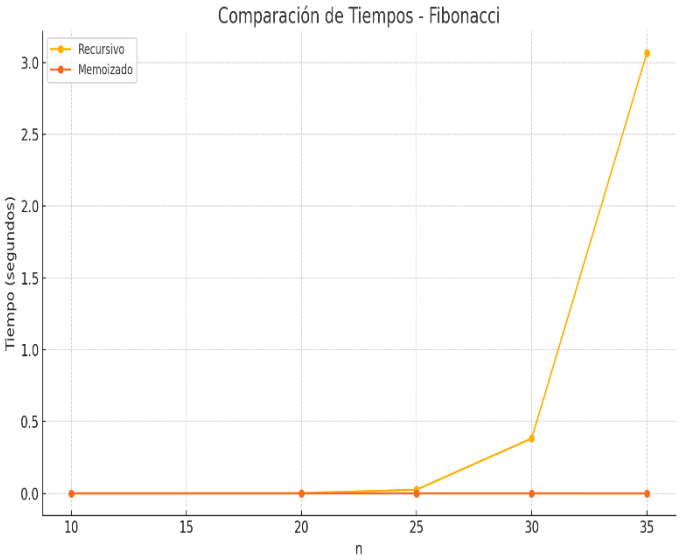
Código Python

```
import time

#Versión recursiva de Fibonacci (sin optimización)
def fibonacci_recursivo(n):
    if n <= 1:
        return n
    return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2)

#Versión optimizada con memoización (almacena resultados ya calculados)
memo = {}       #Diccionario para guardar resultados
def fibonacci_memo(n):
    if n in memo:                   #Si ya está calculado, se retorna directamente
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    return memo[n]

#Valores de entrada para evaluar
valores = [10, 20, 25, 30, 35]

#Se mide una sola vez por cada valor
for n in valores:
    print(f"\nFibonacci de n = {n}")

    #Medición de tiempo para versión recursiva
    inicio = time.time()
    resultado_rec = fibonacci_recursivo(n)
    fin = time.time()
    tiempo_rec = fin - inicio
    print(f"  Recursivo:  {resultado_rec} | Tiempo: {tiempo_rec:.8f} segundos")

    #Medición de tiempo para versión memoizada
    inicio = time.time()
```

```
resultado_memo = fibonacci_memo(n)
fin = time.time()
tiempo_mem = fin - inicio
print(f"  Memoizado: {resultado_memo} | Tiempo: {tiempo_mem:.8f} segundos")
```

Resultados obtenidos:

n	Recursivo	Memoizado
10	0,00004888	0,00001645
20	0,00186443	0,00000668
25	0,02599335	0,0000217
30	0,3834641	0,00002527
35	3,06617451	0,00000715



Análisis

En este ejemplo se compararon dos implementaciones del cálculo de Fibonacci: una versión recursiva tradicional y otra optimizada mediante **memoización**, que reutiliza los resultados ya calculados. Se evaluaron distintos valores de entrada (n = 10, 20, 25, 30 y 35), midiendo el tiempo de ejecución de cada enfoque.

Los resultados muestran una **diferencia de rendimiento significativa** a medida que crece el valor de n. La versión recursiva clásica **recalcula la misma operación muchas veces**, provocando una explosión exponencial en el número de llamadas (complejidad **$O(2^n)$**). Como resultado, el tiempo de ejecución crece rápidamente y se vuelve impráctico para valores grandes.

Por otro lado, la versión memoizada **almacena cada resultado en un diccionario** y lo reutiliza cuando es necesario. Esto reduce drásticamente la cantidad de llamadas recursivas, logrando una complejidad mucho menor (**O(n)**)

y un tiempo de respuesta prácticamente inmediato, incluso para valores altos de n.

Este análisis resalta cómo una **optimización simple como la memoización puede transformar un algoritmo ineficiente en uno altamente eficaz**, especialmente cuando se trata de problemas recursivos con subproblemas repetidos.

4. Metodología Utilizada

Para el desarrollo de este trabajo integrador se recurrió a diversas fuentes bibliográficas previamente citadas, entre ellas los textos de **Downey (2015)** y **Sweigart (2015)**, así como los apuntes de la cátedra y videos explicativos disponibles en el aula virtual. Estos materiales sirvieron para comprender conceptos clave como la eficiencia algorítmica, la notación Big-O y las diferencias entre análisis teórico y práctico.

El entorno utilizado fue **Visual Studio Code**, con la versión **Python 3.13**. Se emplearon únicamente bibliotecas nativas: time para medir los tiempos de ejecución y random para generar datos de entrada aleatorios. No se utilizó control de versiones en esta instancia.

Los ejemplos fueron seleccionados por su representatividad respecto de los temas abordados durante la cursada:

- Comparación entre búsqueda lineal y binaria.
- Contraste entre los algoritmos de ordenamiento Bubble Sort y Quick Sort.
- Evaluación de eficiencia entre las versiones recursiva y memoizada del cálculo de Fibonacci.

El desarrollo consistió en implementar los algoritmos, validar su correcto funcionamiento mediante pruebas intermedias, y luego ejecutar los experimentos con distintos tamaños de entrada, midiendo los tiempos de ejecución una sola vez por cada caso. Esto permitió observar el comportamiento de cada algoritmo sin necesidad de promediar múltiples mediciones, lo que facilitó la visualización de su desempeño.

El trabajo fue realizado de forma colaborativa por dos integrantes, sin una división rígida de tareas. Ambas partes participaron activamente en la codificación, el análisis, la documentación y la redacción del informe.

5. Resultados Obtenidos

El trabajo logró cumplir con los objetivos planteados: aplicar conceptos teóricos y realizar comparaciones empíricas de rendimiento entre distintos

algoritmos. La práctica permitió observar con claridad cómo influyen la estructura y el enfoque de un algoritmo en su desempeño frente a distintos tamaños de entrada.

Ejemplo 1 – Búsqueda Lineal vs Búsqueda Binaria

Se verificó que la búsqueda binaria es **notablemente más eficiente** en listas ordenadas, gracias a su estrategia de dividir el problema en mitades. A medida que el tamaño de la lista aumenta, la diferencia de rendimiento respecto a la búsqueda lineal se hace más evidente. Sin embargo, se remarcó que la búsqueda binaria solo es válida cuando los datos están previamente ordenados, mientras que la búsqueda lineal es aplicable en cualquier caso, aunque con mayor costo en tiempo de ejecución.

Ejemplo 2 – Bubble Sort vs Quick Sort

Este ejemplo dejó en evidencia la **ventaja clara de Quick Sort** frente a Bubble Sort, especialmente en listas más grandes. Quick Sort mantuvo un rendimiento estable gracias a su naturaleza recursiva y estrategia de partición eficiente. Por el contrario, Bubble Sort, aunque funcional, resultó lento debido a su estructura de comparación elemental. El análisis reforzó la idea de que algoritmos simples pueden ser útiles con fines educativos, pero no son apropiados para contextos con grandes volúmenes de datos.

Ejemplo 3 – Fibonacci Recursivo vs Memoizado

La diferencia entre ambas versiones fue significativa. La versión recursiva clásica mostró tiempos crecientes y poco eficientes conforme aumentaba el valor de n, debido a la repetición innecesaria de cálculos. En cambio, la versión **memoizada** redujo drásticamente los tiempos de ejecución al **almacenar y reutilizar resultados previos**, incluso para valores elevados como n = 35. Esto evidenció cómo una optimización estructural puede transformar un algoritmo ineficiente en uno altamente eficaz.

Dificultades

Las principales dificultades surgieron en la implementación de la versión memoizada de Fibonacci y en la comprensión de la lógica recursiva de Quick Sort. Fue necesario consultar ejemplos adicionales y realizar pruebas exhaustivas para garantizar un funcionamiento correcto. Estas instancias consumieron más tiempo que el resto del trabajo, pero aportaron un gran valor al aprendizaje.

6. Conclusiones finales

La realización de este trabajo resultó sumamente **didáctica y enriquecedora**. Permitió aplicar conocimientos teóricos en un contexto práctico, facilitando una comprensión más profunda del análisis de algoritmos. Se cumplieron los objetivos propuestos, incluyendo la aplicación de la notación Big-O y la evaluación empírica de rendimiento.

Uno de los principales aprendizajes fue la **importancia de elegir el algoritmo adecuado según el contexto y tipo de datos**. Aunque todos los algoritmos resuelven sus respectivas tareas, sus tiempos de ejecución y consumo de recursos varían de manera significativa. Esta experiencia reforzó la necesidad de contar con criterios sólidos para la elección de soluciones eficientes en programación.

Este tipo de trabajos no solo consolidan el conocimiento, sino que estimulan el pensamiento crítico y la capacidad de análisis en el campo de la informática.

7. Bibliografía

Downey, A. (2015). *Piensa en Python: Cómo pensar como un científico informático* (2.ª ed.). Versión en español. Green Tea Press. Recuperado de [thinkpython2-spanish.pdf]

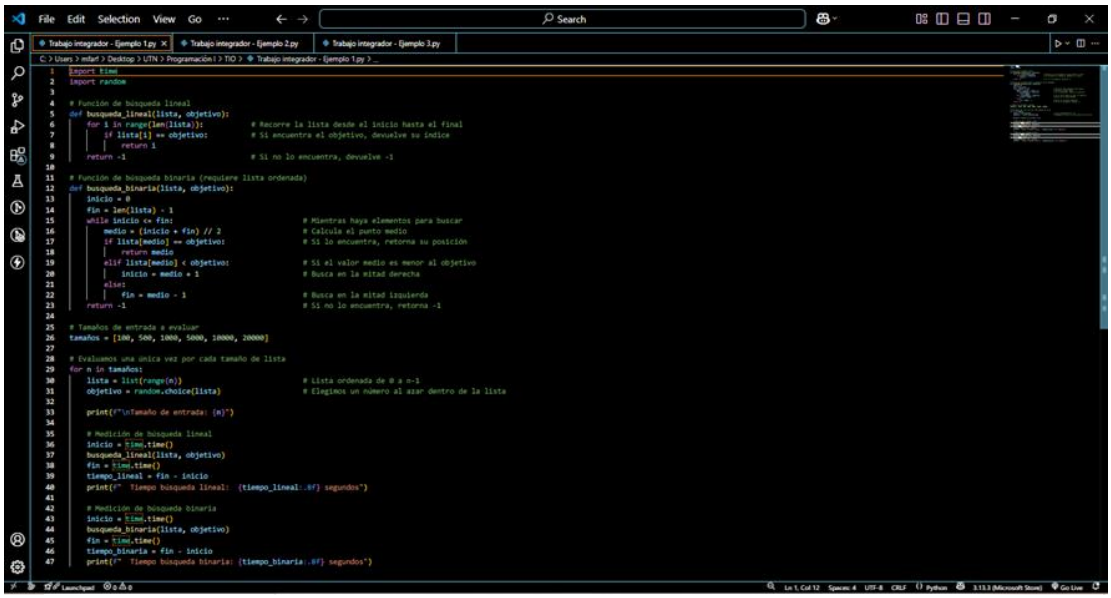
Sweigart, A. (2015). *Automatiza tareas aburridas con Python*. No Starch Press. Recuperado de [Automate the Boring Stuff with Python.pdf]

Análisis Teórico de Algoritmos (s.f.). Material académico en PDF. [Análisis-Teórico-de-Algoritmos.pdf]

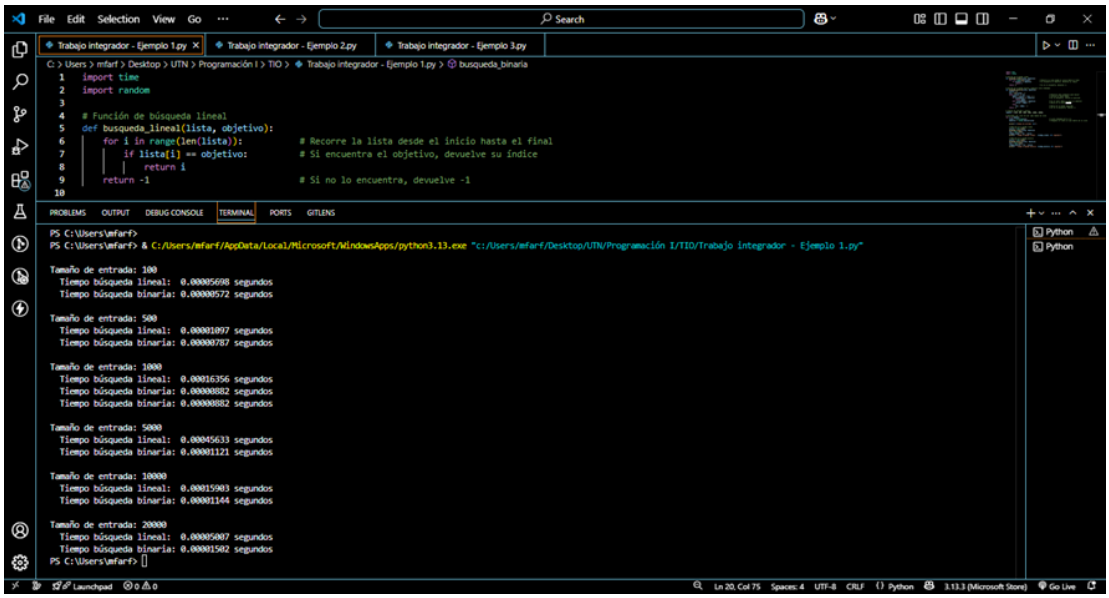
Notación Big-O (s.f.). Material académico en PDF. [Notacion-Big-O.pdf]

8. Anexos

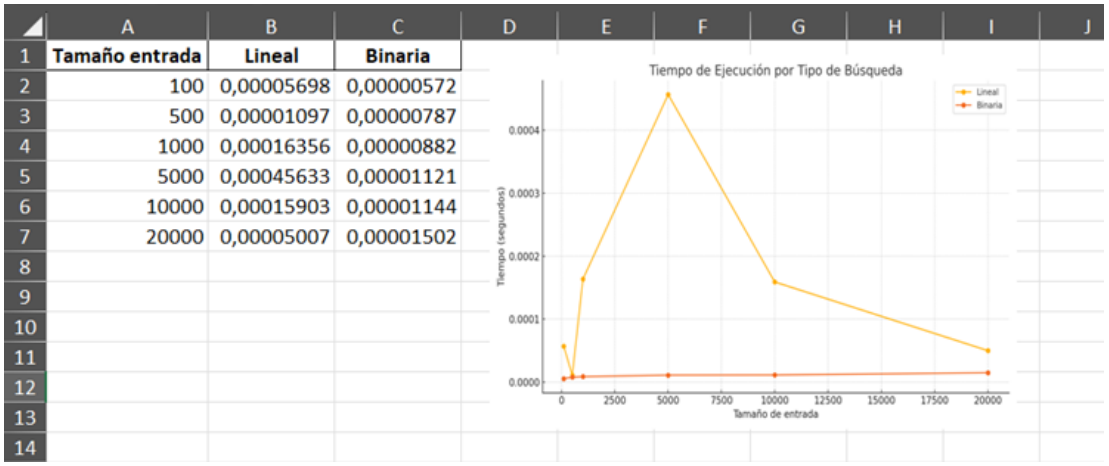
Capturas ejemplo 1:



```python
import time
import random

# Función de búsqueda lineal
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):          # Recorre la lista desde el inicio hasta el final
        if lista[i] == objetivo:         # Si encuentra el objetivo, devuelve su índice
            return i
    return -1                            # Si no lo encuentra, devuelve -1

# Función de búsqueda binaria (requiere lista ordenada)
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:                 # Mientras haya elementos para buscar
        medio = (inicio + fin) // 2      # Calcula el punto medio
        if lista[medio] == objetivo:     # Si lo encuentra, retorna su posición
            return medio
        elif lista[medio] < objetivo:    # Si el valor medio es menor al objetivo
            inicio = medio + 1           # Busca en la mitad derecha
        else:
            fin = medio - 1              # Busca en la mitad izquierda
    return -1                            # Si no lo encuentra, retorna -1

# Tamaños de entrada a evaluar
tamaños = [100, 500, 1000, 5000, 10000, 20000]

# Evaluamos una única vez por cada tamaño de lista
for n in tamaños:
    lista = list(range(n))               # Lista ordenada de 0 a n-1
    objetivo = random.choice(lista)      # Elegimos un número al azar dentro de la lista

    print(f"\nTamaño de entrada: {n}")

    # Medición de búsqueda lineal
    inicio = time.time()
    busqueda_lineal(lista, objetivo)
    fin = time.time()
    tiempo_lineal = fin - inicio
    print(f"  Tiempo búsqueda lineal:  {tiempo_lineal:.8f} segundos")

    # Medición de búsqueda binaria
    inicio = time.time()
    busqueda_binaria(lista, objetivo)
    fin = time.time()
    tiempo_binaria = fin - inicio
    print(f"  Tiempo búsqueda binaria: {tiempo_binaria:.8f} segundos")
```

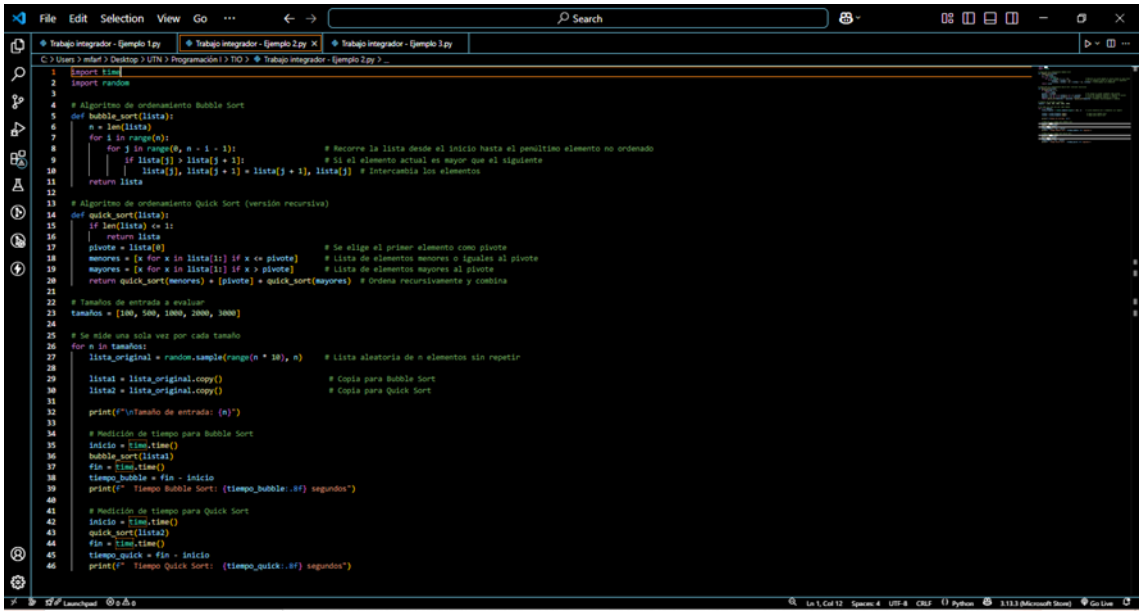


```
PS C:\Users\mfarf>
PS C:\Users\mfarf> & C:/Users/mfarf/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/mfarf/Desktop/UTN/Programación I/TIO/Trabajo integrador - Ejemplo 1.py"

Tamaño de entrada: 100
  Tiempo búsqueda lineal:  0.00005698 segundos
  Tiempo búsqueda binaria: 0.00000572 segundos

Tamaño de entrada: 500
  Tiempo búsqueda lineal:  0.00001097 segundos
  Tiempo búsqueda binaria: 0.00000787 segundos

Tamaño de entrada: 1000
  Tiempo búsqueda lineal:  0.00016356 segundos
  Tiempo búsqueda binaria: 0.00000882 segundos
  Tiempo búsqueda binaria: 0.00000882 segundos

Tamaño de entrada: 5000
  Tiempo búsqueda lineal:  0.00045633 segundos
  Tiempo búsqueda binaria: 0.00001121 segundos

Tamaño de entrada: 10000
  Tiempo búsqueda lineal:  0.00015903 segundos
  Tiempo búsqueda binaria: 0.00001144 segundos

Tamaño de entrada: 20000
  Tiempo búsqueda lineal:  0.00005007 segundos
  Tiempo búsqueda binaria: 0.00001502 segundos
PS C:\Users\mfarf>
```

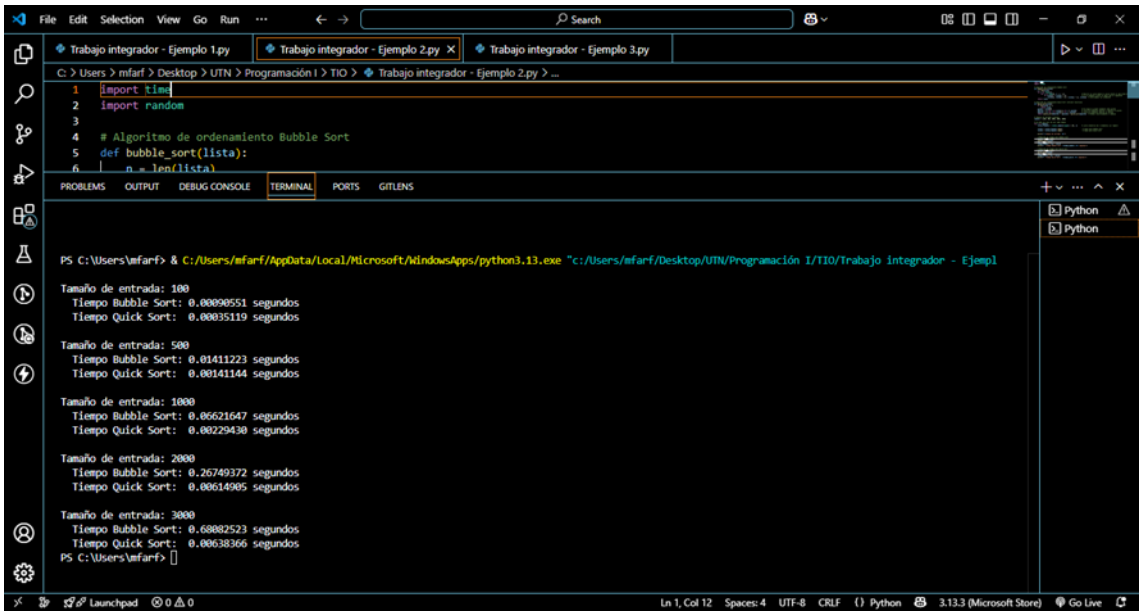


Tamaño entrada	Lineal	Binaria
100	0,00005698	0,00000572
500	0,00001097	0,00000787
1000	0,00016356	0,00000882
5000	0,00045633	0,00001121
10000	0,00015903	0,00001144
20000	0,00005007	0,00001502

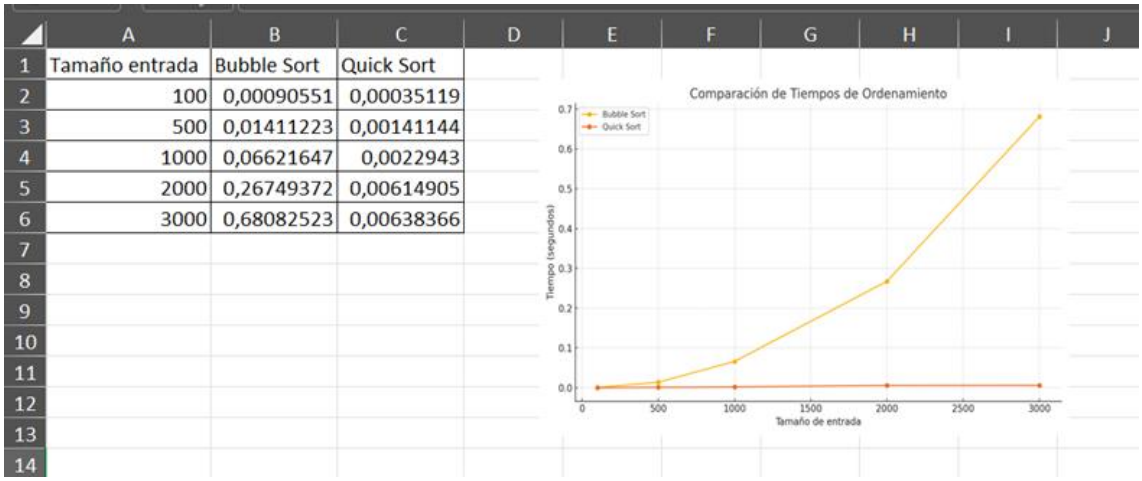
Capturas ejemplo 2:



```python
import time
import random

# Algoritmo de ordenamiento Bubble Sort
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):                # Recorre la lista desde el inicio hasta el penúltimo elemento no ordenado
            if lista[j] > lista[j + 1]:              # Si el elemento actual es mayor que el siguiente
                lista[j], lista[j + 1] = lista[j + 1], lista[j]  # Intercambia los elementos
    return lista

# Algoritmo de ordenamiento Quick Sort (versión recursiva)
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]                                # Se elige el primer elemento como pivote
    menores = [x for x in lista[1:] if x <= pivote]  # Lista de elementos menores o iguales al pivote
    mayores = [x for x in lista[1:] if x > pivote]   # Lista de elementos mayores al pivote
    return quick_sort(menores) + [pivote] + quick_sort(mayores)  # Ordena recursivamente y combina

# Tamaños de entrada a evaluar
tamaños = [100, 500, 1000, 2000, 3000]

# Se mide una sola vez por cada tamaño
for n in tamaños:
    lista_original = random.sample(range(n * 10), n)    # Lista aleatoria de n elementos sin repetir

    lista1 = lista_original.copy()                      # Copia para Bubble Sort
    lista2 = lista_original.copy()                      # Copia para Quick Sort

    print(f"\nTamaño de entrada: {n}")

    # Medición de tiempo para Bubble Sort
    inicio = time.time()
    bubble_sort(lista1)
    fin = time.time()
    tiempo_bubble = fin - inicio
    print(f"  Tiempo Bubble Sort: {tiempo_bubble:.8f} segundos")

    # Medición de tiempo para Quick Sort
    inicio = time.time()
    quick_sort(lista2)
    fin = time.time()
    tiempo_quick = fin - inicio
    print(f"  Tiempo Quick Sort:  {tiempo_quick:.8f} segundos")
```

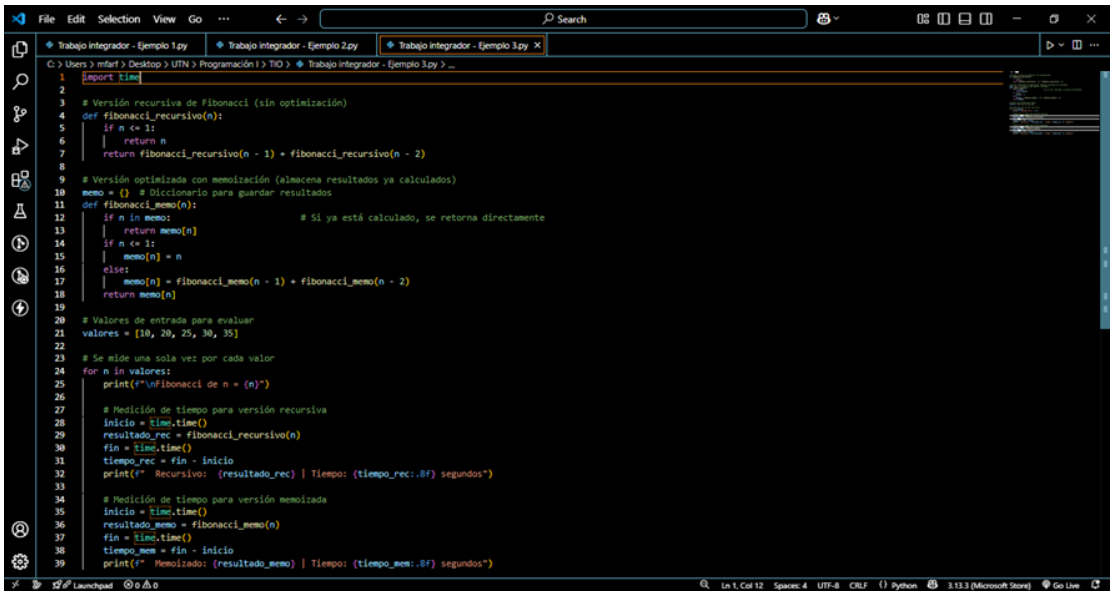


```
PS C:\Users\mfarf> & C:/Users/mfarf/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/mfarf/Desktop/UTN/Programación I/TIO/Trabajo integrador - Ejempl

Tamaño de entrada: 100
  Tiempo Bubble Sort: 0.00090551 segundos
  Tiempo Quick Sort:  0.00035119 segundos

Tamaño de entrada: 500
  Tiempo Bubble Sort: 0.01411223 segundos
  Tiempo Quick Sort:  0.00141144 segundos

Tamaño de entrada: 1000
  Tiempo Bubble Sort: 0.06621647 segundos
  Tiempo Quick Sort:  0.00229430 segundos

Tamaño de entrada: 2000
  Tiempo Bubble Sort: 0.26749372 segundos
  Tiempo Quick Sort:  0.00614905 segundos

Tamaño de entrada: 3000
  Tiempo Bubble Sort: 0.68082523 segundos
  Tiempo Quick Sort:  0.00638366 segundos
PS C:\Users\mfarf>
```



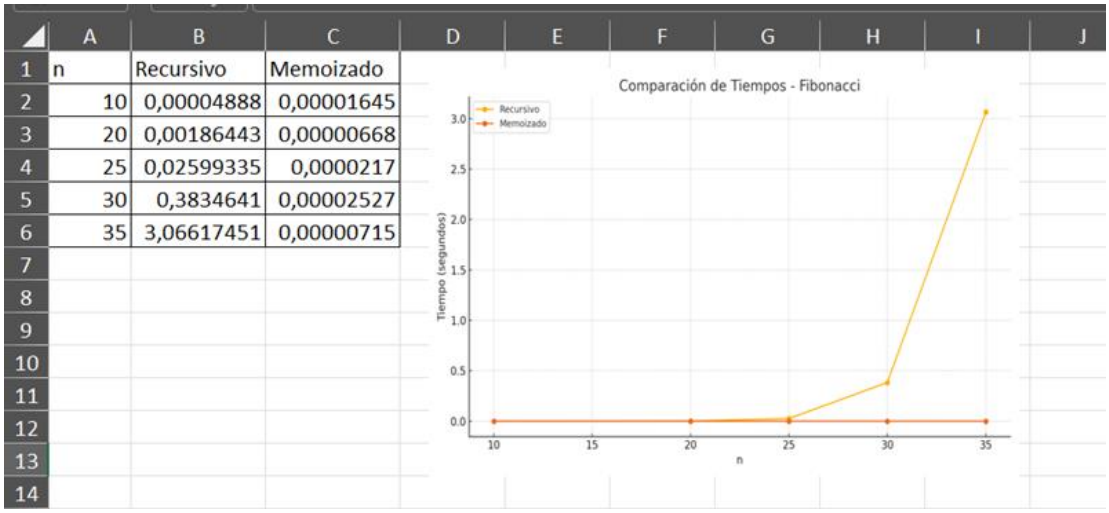
	A	B	C
1	Tamaño entrada	Bubble Sort	Quick Sort
2	100	0,00090551	0,00035119
3	500	0,01411223	0,00141144
4	1000	0,06621647	0,0022943
5	2000	0,26749372	0,00614905
6	3000	0,68082523	0,00638366

Capturas ejemplo 3:



```python
import time

# Versión recursiva de Fibonacci (sin optimización)
def fibonacci_recursivo(n):
    if n <= 1:
        return n
    return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2)

# Versión optimizada con memoización (almacena resultados ya calculados)
memo = {}  # Diccionario para guardar resultados
def fibonacci_memo(n):
    if n in memo:                    # Si ya está calculado, se retorna directamente
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    return memo[n]

# Valores de entrada para evaluar
valores = [10, 20, 25, 30, 35]

# Se mide una sola vez por cada valor
for n in valores:
    print(f"\nFibonacci de n = {n}")

    # Medición de tiempo para versión recursiva
    inicio = time.time()
    resultado_rec = fibonacci_recursivo(n)
    fin = time.time()
    tiempo_rec = fin - inicio
    print(f"  Recursivo:  {resultado_rec} | Tiempo: {tiempo_rec:.8f} segundos")

    # Medición de tiempo para versión memoizada
    inicio = time.time()
    resultado_memo = fibonacci_memo(n)
    fin = time.time()
    tiempo_mem = fin - inicio
    print(f"  Memoizado: {resultado_memo} | Tiempo: {tiempo_mem:.8f} segundos")
```

```
Fibonacci de n = 10
  Recursivo:  55 | Tiempo: 0.00004888 segundos
  Memoizado: 55 | Tiempo: 0.00001645 segundos

Fibonacci de n = 20
  Recursivo:  6765 | Tiempo: 0.00186443 segundos
  Memoizado: 6765 | Tiempo: 0.00000668 segundos

Fibonacci de n = 25
  Recursivo:  75025 | Tiempo: 0.02599335 segundos
  Memoizado: 75025 | Tiempo: 0.00002170 segundos

Fibonacci de n = 30
  Recursivo:  832040 | Tiempo: 0.38346410 segundos
  Memoizado: 832040 | Tiempo: 0.00002527 segundos

Fibonacci de n = 35
  Recursivo:  9227465 | Tiempo: 3.06617451 segundos
  Memoizado: 9227465 | Tiempo: 0.00000715 segundos
PS C:\Users\mfarf>
```



n	Recursivo	Memoizado
10	0,00004888	0,00001645
20	0,00186443	0,00000668
25	0,02599335	0,0000217
30	0,3834641	0,00002527
35	3,06617451	0,00000715

Link del repositorio en GitHub:

https://github.com/Lkssssx/Trabajo-Practico---Analisis-De-Algoritmos

9. **Video explicativo**
   https://youtu.be/Hj7su7HCbGY?si=5Jjn6_N4tq6RjI_Z