UNIVERSIDAD AMERICANA

Algoritmos y estructuras de datos

Documentación QuickSort.

Integrantes:

Franco Xavier Aguilera Ortez

David Joel Sánchez Acevedo

Alicia Massiel Estrada Acevedo

Sara Alejandra Zambrana Taylor

Andrea Johanna Duarte Guerrero

Docente:

Ing. Silvia Ticay.

25 de Junio del 2025

data_generator.py

```
import random
```

Se importa el módulo random para generar datos aleatorios, necesario en varias funciones.

```
def generate_random_list(size, max_value=None):
    """Genera una lista de numeros aleatorios"""

    if max_value is None:
        max_value = size * 10
    return [random.randint(0, max_value) for _ in range(size)]
```

Si no se especifica max_value, se define como 10 veces el tamaño de la lista.
Genera una lista con size enteros aleatorios entre 0 y max_value.

```
def generate_ordered_list(size):
    """Genera una lista ordenada de numeros"""

    return list(range(size))
```

Devuelve una lista ordenada de 0 hasta size - 1.
Representa un caso "ordenado ascendente".

```
def generate_reverse_ordered_list(size):
    """Genera una lista ordenada de numeros en orden inverso"""

    return list(range(size - 1, -1, -1))
```

- Devuelve una lista ordenada en orden descendente.
- Va desde size - 1 hasta 0, paso -1.

```
def generate_duplicates_list(size, num_unique_elements=10):
    """Genera una lista con numeros duplicados"""

    return [random.randint(0, num_unique_elements - 1) for _ in
range(size)]
```

- Genera una lista con solo num_unique_elements diferentes.
- Cada elemento es aleatorio dentro de ese conjunto.
- Ideal para probar QuickSort con datos repetidos.

main.py

```
import sys
import io
import re
from memory_profiler import memory_usage
```

- memory_usage se usa para medir el consumo de memoria del algoritmo.
- Se importan funciones propias para ordenar, generar datos, medir rendimiento y graficar resultados.

```
from quicksort_alg import quicksort
from data_generator import generate_random_list, generate_ordered_list,
generate_reverse_ordered_list, generate_duplicates_list
from performance_metrics import measure_time, calculate_average,
print_results
from plotter import plot_performance

def quicksort_profiled(arr):
    return quicksort(arr)
```

Función de envoltorio para poder pasar quicksort como argumento a memory_usage, que requiere una función con argumentos empaquetados.

```
def parse_memory_profiler_output(output_string):
    #Funcion para analizar la salida de memory_profiler

    for line in output_string.splitlines():
        if "MiB" in line and "Increment" not in line and "Line" not in
line:
            parts = line.split()
            try:
                if "Total memory:" in line:
                    idx = parts.index("memory:") + 1
                else:
                    idx = 0
```

```
                    for i, part in enumerate(parts):
                        if "MiB" in part:
                            idx = i-1
                            break
                    if idx < 0: continue

                mem_value = float(parts[idx].replace('MiB',
'').strip())
                return mem_value #Retorna en MiB
            except (ValueError, IndexError):
                continue
        return None
```

Esta función intenta extraer manualmente el valor de memoria desde la salida de texto de memory_profiler.

```
def run_analysis_for_data_type(data_sizes, num_repetitions,
data_generation_func, title_prefix):
    """Ejecuta el análisis para un tipo de dato específico. (Por
ejemplo, lista aleatoria, ordenada, etc.)"""
    test_results = {}
    print(f"\n----Ejecutando análisis para {title_prefix}----")
    for size in data_sizes:
        print(f"\n----Probando Quicsort con {size} elementos
({title_prefix})----")
        #--- Medicion de Uso de Memoria ---
        print(f"Midiendo uso de memoria para {size} elementos
({title_prefix})...")
        data_for_memory = data_generation_func(size)  #Genera los datos
segun el tipo
        mem_usage = memory_usage((quicksort_profiled,
(list(data_for_memory),)), interval=0.01)
        # Calcular el pico real de memoria (diferencia máxima entre
cualquier punto y el mínimo)
        if mem_usage:
            peak_memory_mib = max([m - min(mem_usage) for m in
mem_usage])
        else:
            peak_memory_mib = None
        if peak_memory_mib is not None and peak_memory_mib > 0:
            print(f"Uso de memoria pico: {peak_memory_mib:.2f} MiB")
```

```
        else:
            print("No se pudo medir el uso de memoria o el valor fue
0.")
        #--- Medicion de Tiempo de Ejecucion ---
        execution_times_seconds = []
        print(f"Midiendo tiempo de ejecución para {size} elementos
({title_prefix})...")
        for i in range(num_repetitions):
            data_for_time = data_generation_func(size) #Genera los
datos segun el tipo
            time_taken_seconds = measure_time(quicksort,
list(data_for_time))
            execution_times_seconds.append(time_taken_seconds)
            print(f"Repetición {i+1}: {time_taken_seconds:.6f}
segundos")
        average_time_seconds =
calculate_average(execution_times_seconds)
        print(f"Tiempo promedio de ejecución:
{average_time_seconds:.6f} segundos")
        test_results[size] = {
            'peak_memory_mib': peak_memory_mib,
            'average_time': average_time_seconds
        }
    return test_results
```

- Ejecuta pruebas de rendimiento para un tipo de datos (aleatorio, ordenado, descendente, etc.).
- Por cada tamaño de lista (data_sizes):
    - Genera datos con la función proporcionada (data_generation_func).
    - Mide uso de memoria con memory_usage.
    - Mide tiempo de ejecución con measure_time, varias veces.
    - Calcula el promedio de tiempos.
- Retorna un diccionario con resultados por tamaño de entrada.

```


if __name__ == "__main__":
        print("¡El script se está ejecutando!")
        data_sizes = [100, 10000, 100000]
```

```
        num_repetitions_per_size = 5

        #Ejecutar para el Caso Promedio (Aleatorio)

        print("----Analizando Caso Promedio (Datos Aleatorios)----")
        average_case_results = run_analysis_for_data_type(data_sizes,
num_repetitions_per_size, generate_random_list, "Caso Promedio
(Aleatorio)")

        print_results(average_case_results, "Caso Promedio
(Aleatorio)")
        plot_performance(average_case_results, "QuickSort. Caso
Promedio (Aleatorio)")

        """
         --- Ejecutar para el Peor Caso (Ordenado Ascendente) ---
    # NOTA: Para nuestra implementación de QuickSort (pivote central),
una lista ya ordenada
    # NO es el peor caso. De hecho, a menudo es eficiente.
    # El verdadero peor caso para pivote central es más complejo (ej.
patrón específico de medianas en sub-particiones).
    # Sin embargo, para demostrar "peor caso" para otras
implementaciones de QS (pivote fijo),
    # o para ilustrar un caso "no-óptimo" para el nuestro, podemos
usarlo.
        """

        print("----Analizando Caso Peor (Datos Ordenados)----")

        results_ordered_case = run_analysis_for_data_type(data_sizes,
num_repetitions_per_size, generate_ordered_list, "Caso Peor (Datos
Ordenados Ascendente)")

        print_results(results_ordered_case, "Caso Peor (Datos Ordenados
Ascendente)")
        plot_performance(results_ordered_case, "QuickSort. Caso Peor
(Datos Ordenados Ascendente)")

        #Ejecutar para el Peor Caso (Ordenado Descendente)
        print("----Analizando Caso Peor (Datos Ordenados
Descendente)----")
        results_reverse_ordered_case =
run_analysis_for_data_type(data_sizes, num_repetitions_per_size,
```

```
generate_reverse_ordered_list, "Caso Peor (Datos Ordenados
Descendente)")

        print_results(results_reverse_ordered_case, "Caso Peor (Datos
Ordenados Descendente)")
        plot_performance(results_reverse_ordered_case, "QuickSort. Caso
Peor (Datos Ordenados Descendente)")
```

performance_metrics.py

```
import time
import pandas as pd
from memory_profiler import profile as memory_profile
```

- **time:** para medir duración de ejecuciones.
- **pandas:** para tabular y mostrar resultados.
- **memory_profiler**: no usado directamente aquí, pero podría usarse para decoradores.

```
def measure_time(func, *args, **kwargs):
    """
    Mide el tiempo de ejecucion de una funcion
    :param func: La función a medir.
    :param args: Argumentos posicionales para la función.
    :param kwargs: Argumentos de palabra clave para la función.
    :return: El tiempo de ejecución en segundos.
    """

    start_time = time.perf_counter()
    func(*args, **kwargs)
    end_time = time.perf_counter()
    return (end_time - start_time)
```

- Mide tiempo de ejecución usando perf_counter.
- Recibe cualquier función con sus argumentos.

```
def calculate_average(data_points):
    """
    Calcula el promedio de una lista de puntos de datos.
    :param data_points: Lista de puntos de datos.
    :return: El promedio de los puntos de datos.
    """
```

```
    if not data_points:
        return 0
    return sum(data_points) / len(data_points)
```

- Calcula el promedio de una lista de números.
- Si está vacía, devuelve 0 para evitar errores.

```
def print_results(results, title="Resultados de Medición"):
    """
    Imprime los resultados en formato tabular.
    :param results: Diccionario con los resultados a imprimir.
    """

    print(f"\n{title}")
    data = []

    for size, metrics in results.items():
        row = {
            "Tamaño de Datos (N)": size,
            "Tiempo Promedio (s)":f"{metrics['average_time']:.6f}",
            "Uso de Memoria Pico (MiB)":
f"{metrics['peak_memory_mib']:.2f}" if metrics['peak_memory_mib'] is
not None else "N/A"
        }
        data.append(row)

    df = pd.DataFrame(data)
    print(df.to_string(index=False))
    print("\n" + "=" * 50)
```

- Muestra resultados en tabla usando pandas.
- Incluye tiempo promedio y memoria pico.

plotter.py

```
import matplotlib.pyplot as plt

def plot_performance(results, title="Quicksort Performance"):
    """
```

```
    Genera graficos de tiempo de ejecucion y uso de memoria.
    """

    sizes = sorted(results.keys())
    # Usar la clave correcta para los tiempos
    times = [results[size]['average_time'] if 'average_time' in
results[size] else results[size].get('average_time_seconds', 0) for
size in sizes]

    memories = [results[size]['peak_memory_mib'] for size in sizes if
results[size]['peak_memory_mib'] is not None and
results[size]['peak_memory_mib'] > 0]
    sizes_for_memory = [size for size in sizes if
results[size]['peak_memory_mib'] is not None and
results[size]['peak_memory_mib'] > 0]

    # Filtrar para evitar ceros o negativos en log
    sizes_log = [s for s, t in zip(sizes, times) if s > 0 and t > 0]
    times_log = [t for s, t in zip(sizes, times) if s > 0 and t > 0]

    #Grafico de Tiempo de Ejecucion vs. Tamaño de Datos
    if sizes_log and times_log:
        plt.figure(figsize=(10, 5))
        plt.plot(sizes_log, times_log, marker='o', linestyle='-',
color='blue')
        plt.xscale('log') # Escala logaritmica para N
        plt.yscale('log') # Escala logaritmica para tiempo
        plt.title(f"{title} - Tiempo de Ejecución vs. Tamaño de Datos")
        plt.xlabel("Tamaño de Datos (N)")
        plt.ylabel("Tiempo Promedio (s)")
        plt.grid(True, which='both', linestyle='--')
        plt.tight_layout()
        plt.show()
    else:
        print("No hay datos positivos para graficar tiempo en escala
logarítmica.")

    #Grafico de Uso de Memoria vs. Tamaño de Datos
    if memories and all(m > 0 for m in memories):
        plt.figure(figsize=(10, 5))
        plt.plot(sizes_for_memory, memories, marker='o', linestyle='-',
color='red')
        plt.xscale('log') # Escala logaritmica para N
```

```
        plt.title(f"{title} - Uso de Memoria vs. Tamaño de Datos")
        plt.xlabel("Tamaño de Datos (N)")
        plt.ylabel("Uso de Memoria Pico (MiB)")
        plt.grid(True, which='both', linestyle='--')
        plt.tight_layout()
        plt.show()
    else:
        print("No hay datos positivos para graficar memoria en escala
logarítmica.")
```

Se importa la biblioteca matplotlib.pyplot, para la creación de gráficos y visualizaciones científicas. Dentro del archivo hay una única función llamada plot_performance, la cual toma como argumentos un diccionario con los resultados experimentales (results) y un título opcional (title) para personalizar los encabezados de los gráficos.

El diccionario de resultados que se recibe tiene como claves los tamaños de entrada evaluados (por ejemplo, 1000, 10000, 100000) y como valores otro diccionario que incluye dos métricas principales: el tiempo promedio de ejecución (average_time) y el uso máximo de memoria durante la ordenación (peak_memory_mib). La función comienza extrayendo y ordenando las claves del diccionario (es decir, los tamaños de entrada), y luego construye listas con los valores de tiempo y memoria correspondientes. En el caso del tiempo, también considera una alternativa (average_time_seconds) por si los resultados fueron etiquetados de forma distinta.

Para evitar errores en los gráficos, la función filtra todos los valores inválidos (como ceros o negativos) que no pueden representarse en escalas logarítmicas. Luego, se genera un primer gráfico que representa el tiempo promedio de ejecución frente al tamaño de los datos. Este gráfico utiliza una escala logarítmica en ambos ejes para resaltar mejor cómo se comporta el algoritmo conforme aumenta la entrada. El trazado es una línea azul con marcadores circulares en cada punto, y se le añaden etiquetas a los ejes, un título y una cuadrícula de fondo para facilitar su lectura.

Después, si existen datos válidos de memoria, se genera un segundo gráfico que muestra cómo varía el uso máximo de memoria según el tamaño de entrada. En este caso, se utiliza escala logarítmica únicamente en el eje horizontal. El gráfico utiliza una línea roja con marcadores, y también incluye título, etiquetas de ejes, cuadrícula y ajustes visuales para asegurar que los elementos gráficos no se solapen.

En caso de que no haya datos positivos o suficientes para construir alguno de los gráficos, se imprime un mensaje en consola indicando el motivo y se omite esa visualización. Al final, ambos gráficos se muestran en pantalla utilizando la función plt.show() y se aplican mejoras visuales con plt.tight_layout() para optimizar la presentación general. Esta función no devuelve ningún valor, ya que su propósito es exclusivamente visual. Es especialmente útil para interpretar los resultados de forma intuitiva y comparar el rendimiento de QuickSort bajo diferentes condiciones de entrada.

quicksort_alg.py

```python
def quicksort(arr):
    """Implementacion del algoritmo QUICKSORT"""

    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

- Caso base: si la lista tiene 0 o 1 elemento, no se ordena.
- El pivote es el elemento central de la lista.
- Se crean tres listas:
    - left: menores al pivote
    - middle: iguales al pivote
    - right: mayores al pivote
- Se aplica QuickSort recursivamente a left y right, y se unen los tres segmentos.