Facultad de Ingeniería
UNER

Trabajo Práctico N° 1

ALUMNOS:

Biasoli, Ana Inés

Taborda, Veronica

Problema 1
Análisis de ordenamiento Burbuja

El ordenamiento Burbuja (o *Bubble Sort*) es un algoritmo de ordenamiento sencillo que compara pares de elementos adyacentes en una lista y los intercambia si están en el orden incorrecto. Este proceso se repite varias veces hasta que la lista queda completamente ordenada. Su nombre proviene del modo en que los elementos más grandes "flotan" hacia el final de la lista, como burbujas en el agua. Aunque es fácil de implementar, no es eficiente para listas grandes, ya que su complejidad en el peor caso esO(n^2).

```
import random
import matplotlib.pyplot as plt
import time

def OrdenamientoBurbuja(lista1):
    for extremo in range(len(lista1)-1,0,-1):
        for i in range(extremo):
            if lista1[i]>lista1[i+1]:
                temp = lista1[i]
                lista1[i] = lista1[i+1]
                lista1[i+1] = temp
    return lista1

lista1 = [random.randint(10000,99999) for _ in range(500)]
#print("Lista 1:", lista1)
listaordenadaburb = OrdenamientoBurbuja(lista1)
#print ("lista ordenada:",listaordenadaburb)

#if __name__=="__main__":
    #lista1 = [random.randint(10000,99999) for _ in range(500)]
    #print("Lista 1:", lista1)
    #listaordenadaburb = OrdenamientoBurbuja(lista1)
    #print ("lista ordenada:",listaordenadaburb)

contador=0
ordenada = True
while contador < len(listaordenadaburb) - 1:
    if listaordenadaburb[contador] > listaordenadaburb[contador + 1]:
        ordenada = False
        break
    contador += 1
if ordenada:
    print("La lista está correctamente ordenada.")
else:
    print("La lista NO está ordenada.")
```

Este programa genera una lista de 500 números aleatorios entre 10.000 y 99.999, la ordena utilizando el algoritmo de burbuja (implementado en la función `OrdenamientoBurbuja`) y luego verifica si la lista quedó correctamente ordenada de menor a mayor. El algoritmo recorre repetidamente la lista comparando pares de elementos adyacentes e intercambiándolos si están en el orden incorrecto, lo que hace que los valores más grandes "floten" hacia el final. Después de ordenar, se recorre la lista con un bucle `while` para comprobar que cada elemento sea menor o igual al siguiente; si se encuentra un par desordenado, se marca como incorrecto y se detiene la verificación. Finalmente, el programa imprime un mensaje indicando si la lista está correctamente ordenada.

Análisis de ordenamiento por residuos

El ordenamiento por residuos (también conocido como ordenamiento por resto o ordenamiento modular) es una técnica que organiza los elementos de una lista basándose en el residuo que deja cada uno al ser dividido por un número fijo (el *módulo*). Por ejemplo, si usamos el módulo 3, agrupamos los elementos según si su residuo es 0, 1 o 2. Este método no es un algoritmo de ordenamiento clásico como Burbuja o Quicksort, pero puede utilizarse en situaciones específicas, como para clasificar datos en grupos según ciertas propiedades numéricas. Es útil cuando se busca una agrupación previa o una forma particular de ordenar basada en divisiones.

```
def ordenamiento_porconteo(lista, exp):
    n = len(lista)
    lista_datos_ordenados = [0] * n    #esta lista va a contener los
datos ordenados
    conteo = [0] * 10    #esta lista es la que va a contar la cantidad
de veces que aparece cada dígito en la lista original

    # Contar cuántos números tienen cada dígito en la posición 'exp'
    for i in range(n):
        indice = (lista[i] // exp) % 10
        conteo[indice] += 1

    # Acumular los conteos para saber las posiciones correctas
    for i in range(1, 10):
        conteo[i] += conteo[i - 1]

    # Construir la lista ordenada según el dígito actual
    for i in reversed(range(n)):
        indice = (lista[i] // exp) % 10
        lista_datos_ordenados[conteo[indice] - 1] = lista[i]
        conteo[indice] -= 1

    return lista_datos_ordenados
```

```
# Función principal: Ordenamiento Radix
def ordenamiento_radix(lista):
    maximo = max(lista)        # Encontrar el número más grande
    exp = 1                    # Comenzar por las unidades

    # Repetir para cada dígito (unidades, decenas, centenas, etc.)
    while maximo // exp > 0:
        lista = ordenamiento_porconteo(lista, exp)
        exp *= 10

    return lista
```

Este programa implementa el algoritmo de ordenamiento Radix Sort, que ordena una lista de números enteros comparando sus dígitos de menor a mayor (unidades, decenas, centenas, etc.). La función principal `ordenamiento_radix` comienza identificando el número más grande de la lista para determinar cuántas cifras tiene, y luego aplica repetidamente el ordenamiento por conteo (`ordenamiento_porconteo`) en cada posición decimal. Esta función auxiliar cuenta cuántas veces aparece cada dígito en la posición actual (`exp`), acumula esos conteos para determinar las posiciones correctas de los elementos, y finalmente construye una nueva lista ordenada según ese dígito. El proceso se repite aumentando la potencia de 10 hasta haber ordenado por todas las cifras del número más grande. Así, al finalizar, la lista queda completamente ordenada de menor a mayor.

Análisis de ordenamiento Quicksort

El **ordenamiento Quicksort** es un algoritmo eficiente de ordenamiento que utiliza el enfoque de **divide y vencerás**. Consiste en seleccionar un elemento llamado *pivote* y dividir la lista en dos sublistas: una con los elementos menores al pivote y otra con los mayores. Luego, se aplica recursivamente el mismo procedimiento a cada sublista. Finalmente, se combinan las sublistas ordenadas con el pivote en el medio.

```
import random
import time
import matplotlib.pyplot as plt

# Implementación de quicksort
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivote = arr[len(arr) // 2]
        menores = [x for x in arr if x < pivote]
        iguales = [x for x in arr if x == pivote]
        mayores = [x for x in arr if x > pivote]
        return quicksort(menores) + iguales + quicksort(mayores)
```

```
# Medición de tiempos para listas de tamaño 1 a 1000
tamanios = list(range(1, 1001))
tiempos_quick = []

for n in tamanios:
    lista = [random.randint(10000, 99999) for _ in range(n)]
    inicio = time.time()
    quicksort(lista)
    tiempos_quick.append(time.time() - inicio)

# Graficar los resultados
plt.figure(figsize=(10, 6))
plt.plot(tamanios, tiempos_quick, label='Quicksort', color='blue')
plt.xlabel('Tamaño de la lista')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title('Tiempo de ejecución de Quicksort (listas con números de 5
dígitos)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Se implementa algoritmo quicksort. Este es un algoritmo de ordenamiento que divide una lista de datos en sublistas más pequeñas, para luego ordenarlas individualmente.

```
def quicksort(arr):
    if len(arr) <= 1:              (si la lista tiene de 0a 1 elemento, ya está ordenada)
        return arr
    else:
        pivote = arr[len(arr) // 2]                (elige como pivote el elemento del medio)
        menores = [x for x in arr if x < pivote]
        iguales = [x for x in arr if x == pivote]
        mayores = [x for x in arr if x > pivote]
        return quicksort(menores) + iguales + quicksort(mayores)
```

Ordena recursivamente menores y mayores y une todo en un nuevo orden

Sobre la medición del tiempo en la ejecucion
tamanios: genera una lista de tamaños de 1 a 1000 (ej: [1, 2, ..., 1000]).
tiempos_quick: lista vacía donde se guardarán los tiempos de ejecución.

```
for n in tamanios:
    lista = [random.randint(10000, 99999) for _ in range(n)]
```

```
inicio = time.time()
quicksort(lista)
tiempos_quick.append(time.time() - inicio)
```

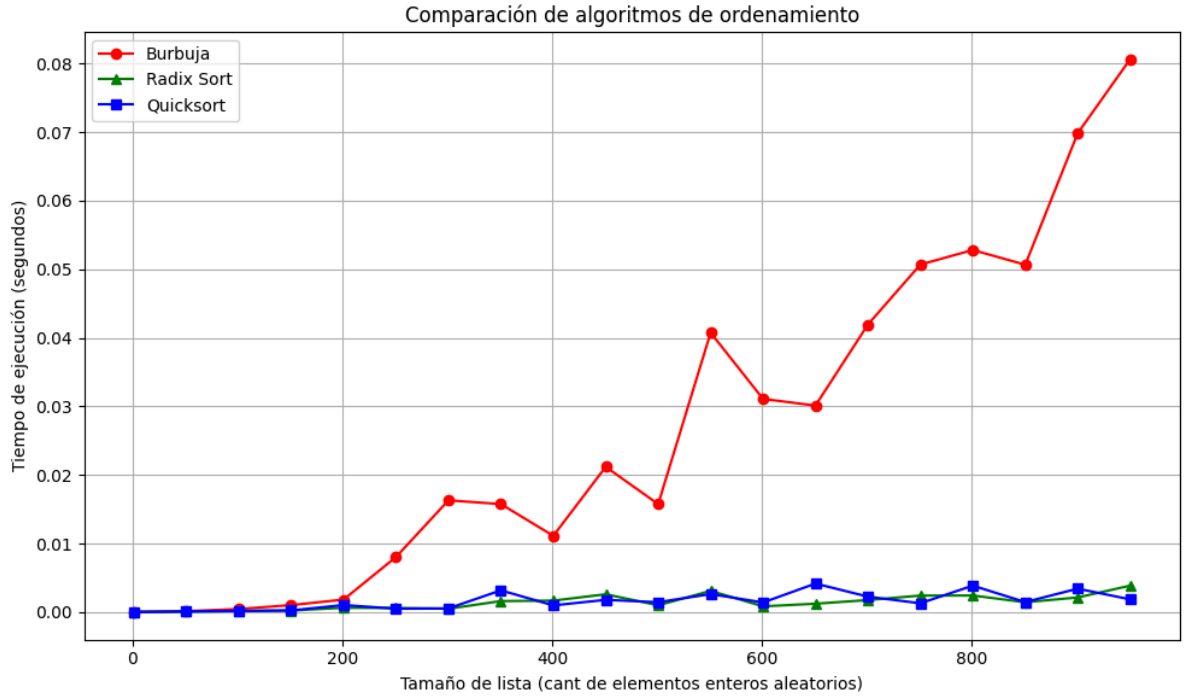
Se genera una lista con n números aleatorios de 5 dígitos

inicio=time.time(): guarda el tiempo justo antes de ordenar
quicksort (lista) ordena la lista
tiempos_quick.append(...): calcula cuánto tiempo tardó y lo guarda en la lista tiempos_quick

Se graficó el tiempo de ejecución en función del tamaño de la lista. En la imágen puede observarse que, aunque hay cierta variabilidad, el tiempo de ejecución tiende a crecer de forma **logarítmico-lineal**, lo que es esperable para este tipo de algoritmo.

Para la misma se crea una figura de tamaño 10x6 pulgadas, se grafica el tiempo (tiempos_quick) en función del tamaño de la lista (tamanios). Finalmente, se ponen etiquetas a los ejes y se muestra la leyenda, se activa la cuadrícula del gráfico, se ajustan los márgenes y se muestra la gráfica en la pantalla.

En la siguiente imagen mostramos la gráfica que cuenta con los tres tipos de ordenamiento, los cuales varían por cada ejecución ya que son listas aleatorias.



En cada algoritmo, el orden de complejidad de O son:

- Bubble Sort: $O(n^2)$: este algoritmo compara un par de elementos adyacentes y los intercambia si están en orden incorrecto, lo que implica n^2 comparaciones en el peor de los casos. En la gráfica podemos ver como la curva roja crece de forma cuadrática, concordando con orden de complejidad O. Podemos ver que para listas de 0 a 200 elementos tiene un comportamiento similar a algoritmos como QuickSort

o RadixSort, pero siendo mayores a 200 elementos el tiempo de funcionamiento crece de forma considerable, lo que demuestra su ineficiencia en estos casos.

- Radix Sort: $O(nk)$: en este caso, k es el número de dígitos del número más grande de la lista. Este algoritmo ordena los números por dígitos: primero por las unidades, luego decenas, centenas, etc. Por cada dígito se hace una pasada completa sobre la lista, usando un algoritmo estable como Counting Sort (que es $O(n)$).Si hay k dígitos, se realizan k pasadas de $O(n)$ cada una, dando un total de $O(nk)$. Es por esto que este algoritmo es eficiente para enteros y cuando k es pequeño y constante. En la gráfica podemos observar que la curva es casi plana o crece muy poco, lo cual es consistente con un crecimiento lineal; y se ve que es muy eficiente para los tamaños de listas grandes que utilizamos.
- Quick Sort: tiene una complejidad teórica promedio de $O(n \cdot log(n))$: este algoritmo divide la lista en 2 partes y las ordena recursivamente; eligiendo un buen pivote el número de comparaciones es cercano a $n \cdot log(n)$, en el caso de no elegir un buen pivote el número de comparaciones es n^2, siendo en ese caso la complejidad teórica tipo $O(n^2)$. En la gráfica podemos observar un crecimiento lento, lo que indica que a mayor cantidad de elementos este método tarda más tiempo en ordenar la lista, pero no tanto como el ordenamiento Burbuja, lo que demuestra que es más eficiente que este último pero un que Radix Sort en cuanto a tiempo es el más eficiente de todos.

La función *sorted* usa el algoritmo **Timsort**, que es una combinación de Merge Sort y de Insertion Sort. Es eficiente en datos reales, sobre todo si ya están parcialmente ordenados. Su complejidad en el mejor de los casos, que es cuando la lista se encuentra casi ordenada, es $O(n)$ y en el peor de los casos es $O(n \cdot log(n))$.

La función sorted detecta patrones ya ordenados y los aprovecha con el fin de ahorrar trabajo; divide la lista en subsecuencias ordenadas y luego las fusiona eficientemente; mantiene el orden de los elementos iguales y está optimizada en C, lo que hace que funcione mucho más rápido.

Haciendo un cuadro comparativo podemos comparar complejidad, estabilidad y rendimiento de cada uno de los algoritmos.

Algoritmo	Complejidad	Estabilidad	Rendimiento
Bubble Sort	$O(n^2)$	Sí	Malo
Radix Sort	$O(nk)$	A veces	Bueno en el caso de enteros
Quick Sort	$O(n \cdot log(n))$	No	Muy bueno
Sorted	$O(n)$ o $O(n \cdot log(n))$	Sí	Excelente