Tecnológico de Costa Rica

Análisis de Algoritmos | IC4301

Tarea Corta #1

Profesor. Joss Pecou Johnson

Estudiantes:

- **Juan Mora Acuña**
- **Angelo Piedra Castro**
- **Poll Garro Vargas**

Grupo 60

II Semestre | 2025

Parte 1: Implementación y Análisis de Ordenamiento.

Objetivo: Implementar y analizar el algoritmo de ordenamiento que se aplique a una estructura de lista de listas presentando cada lista ordenada en el resultado.

Instrucciones:

- Implementar el algoritmo de ordenamiento que ordene elementos dentro de una lista de listas en el lenguaje de programación Python utilizando estructuras de tamaño nxn.
- Analizar la complejidad temporal del algoritmo para el mejor caso, peor caso y caso medio.
- Proporcionar la función T(n) que describe la complejidad temporal del algoritmo.
- Aplicar el algoritmo para una lista de números aleatorios de 100, 1000, 10000 y 100000 elementos, y presentar los resultados obtenidos en tiempo de ejecución y espacio de memoria utilizado. 5. Comparar la eficiencia del algoritmo implementado con otro algoritmo de ordenamiento de su elección (ej. BubbleSort, QuickSort, InsertionSort, etc.).

Código fuente de la implementación:

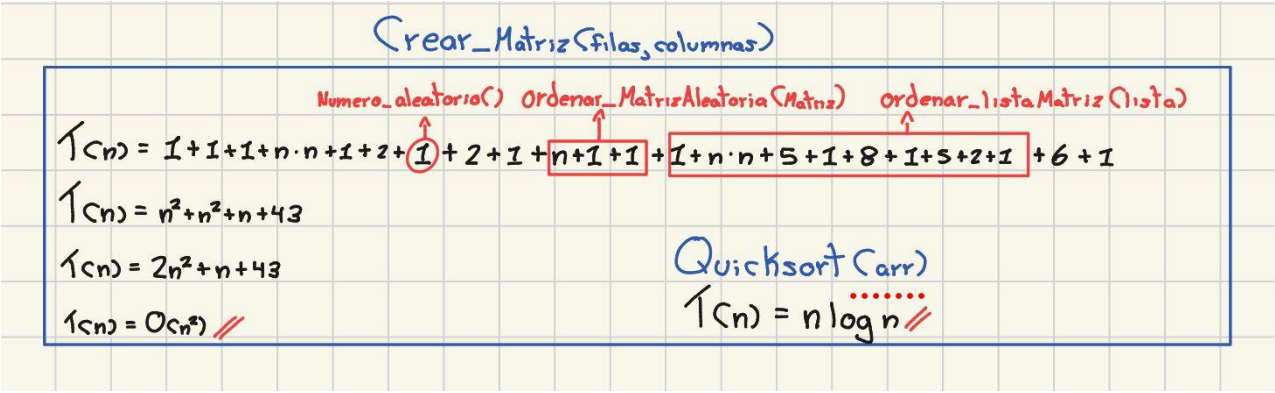
```python
import random # Libreria para generar listas con numeros randoms.
import time # Libreria para visaulizar el tiempo de ejecución del programa.
import tracemalloc # Libreria para visualizar la cantidad de memoria que utiliza la ejecucion del programa.

"""
========================================
        Funciones de Ordenamiento
========================================
"""

# Función que genera números aleatorios para posteriormente agregarlos a las listas de la matriz.
def Numero_aleatorio():
    return random.randint(-100000, 100000)

# Función de ordenamiento Quicksort (Función reutilizada para probar función Quicksort vs función generada por el grupo).
def quicksort(arr):
    # Caso inicial donde las listas pueden estar vacías o si la lista posee un solo elemento.
    if len(arr) <= 1:
        return arr
    else:
        pivote = arr[len(arr) // 2]  # Se selecciona el elemento central de la lista para utilizarlo como pivote.
        izquierda = [x for x in arr if x < pivote]
        centro = [x for x in arr if x == pivote]
        derecha = [x for x in arr if x > pivote]
        return quicksort(izquierda) + centro + quicksort(derecha)

# Función para ordenar cada una de las listas que están dentro de la matriz.
def ordenar_MatrizAleatoria(Matriz):
    for x in range(len(Matriz)):
        Matriz[x] = ordenar_listaMatriz(Matriz[x])
    return Matriz
```

```python
# Función que se encarga de ordenar cada uno de los elementos que están dentro de una lista.
def ordenar_listaMatriz(lista):
    NewLista = [] # Variable donde se almacenará la lista ya ordenada para retornarla al final.
    while lista != []:
        x = lista[0] # Variable que almacenará el primer elemento de la lista.
        lista = lista[1:] # Se recorta la lista sin el primer elemento.
        y = 0
        while y < len(lista):
            if x > lista[0]:
                lista += [x]
                x = lista[0]
                lista = lista[1:]
                y += 1
            else:
                lista += [lista[0]]
                lista = lista[1:]
                y += 1

        NewLista += [x]
    return NewLista

# Función para crear una matriz con la cantidad de filas y columnas recibidas y ordenarla para mostrarla.
def Crear_Matriz(filas, columnas):
    if not isinstance(filas,int) or not isinstance(columnas,int):
        return "Los datos deben de ser enteros"
    elif filas < 1 or columnas < 1:
        return "Los datos deben de ser mayor a 0"
    else:
        inicio = time.time() # Se inicia el tiempo de ejecución del programa.
        tracemalloc.start() # Se inicia el control de memoria durante la ejecución del programa.
        matriz = []
        for _ in range(filas):
            fila = []
            for _ in range(columnas):
                fila += [Numero_aleatorio()] # Fila se rellena con números aleatorios.
            matriz += [fila]

        matriz = ordenar_MatrizAleatoria(matriz)
        final = time.time() # Se finaliza el tiempo de ejecución del programa.
        Duracion = final - inicio # Se resta el tiempo inicial del tiempo final para calcular el tiempo total.
        Actual, Maximo = tracemalloc.get_traced_memory() # Variables para almacenar la memoria actual y el pico máximo de la memoria utilizada.
        tracemalloc.stop() # Se frena el control de memoria de la ejecución del programa.
        # Se muestra el tiempo de la ejecición, la memoria actual, la memoria máxima utilizada y la matriz ordenada.
        return (
                f"Duracion: {Duracion}\n"
                f"Memoria Actual: {Actual}\n"
                f"Memoria Pico: {Maximo}\n"
                #f"Matriz: {matriz}\n"
                )
```

Explicación breve del algoritmo:

La función principal para la funcionalidad del algoritmo solicitado es "Crear_Matriz" la cual lo que realiza es solicitar como parámetro 2 entradas, filas y columnas. Se inicia con validaciones y en una variable se almacena el tiempo y el espacio de la ejecución del programa. Luego se crea una variable que contenga una lista vacía donde se llenara con la cantidad de filas y columnas solicitadas anteriormente. Cada una de las filas va a ser rellenada con números aleatorios utilizando la función "Numero_aleatorio" que selecciona números randoms del -100000 al 100000; y luego de crear una fila, se incluye a la matriz. Luego se modifica la variable "matriz" llamando a la función "ordenar_MatrizAleatoria" la cual ordena cada una de las listas de la matriz con el uso de la función "ordenar_listaMatriz" y devuelve la lista ordenada. Para al final de la ejecución del programa retornar la cantidad de tiempo, espacio y la matriz con todas sus sublistas ordenadas.

Análisis de complejidad temporal incluyendo la función T(n):



Peor caso: El peor caso de la función de ordenamiento es de **O(n^2)**

Mejor caso: La función de ordenamiento presenta como mejor caso **Ω(n^2)**

Caso promedio: El caso medio de la función de ordenamiento es de **Θ (n^2)**

Resultados experimentales para diferentes tamaños de entrada:

```
10x10 = 100
Duracion: 0.0010008811950683594
Memoria Actual: 4736
Memoria Pico: 4848
```

```
32x32 = 1024
Duracion: 0.03000640869140625
Memoria Actual: 42520
Memoria Pico: 42872
```

```
100x100 = 10000
Duracion: 0.8753938674926758
Memoria Actual: 412664
Memoria Pico: 413560
```

```
317x317 = 100489
Duracion: 34.31422805786133
Memoria Actual: 4127152
Memoria Pico: 4129848
```

Comparación de eficiencia con otro algoritmo:

A continuación, se verá el rendimiento del Quicksort con las entradas anteriores.

```
Quicksort 10x10
Duracion: 0.0009999275207519531
Memoria Actual: 5376
Memoria Pico: 6296
```

```
Quicksort 32x32
Duracion: 0.007001638412475586
Memoria Actual: 44944
Memoria Pico: 46896
```

```
Quicksort 100x100
Duracion: 0.06301331520080566
Memoria Actual: 409824
Memoria Pico: 421472
```

```
Quicksort 317x317
Duracion: 0.6637172698974609
Memoria Actual: 4042896
Memoria Pico: 4147120
```

Parte 2: Implementación y Análisis de Búsqueda:

Objetivo: Implementar y analizar un algoritmo de búsqueda que se pueda aplicar en una estructura de tipo lista de lista.

Instrucciones:

- Implementar un algoritmo de búsqueda en una lista de listas utilizando algún algoritmo de búsqueda existente o implementando uno propio.
- Analizar la complejidad temporal del algoritmo en mejor caso, peor caso y caso promedio.
- Proporcionar la función T(n) correspondiente.
- Aplicar el algoritmo sobre diferentes tamaños de matrices (ej. 10x10, 100x100, 500x500) y presentar los resultados en tiempo de ejecución y espacio en memoria utilizado.
- Comparar el comportamiento del algoritmo de búsqueda con otro algoritmo alternativo (ej. búsqueda lineal vs búsqueda binaria).

Código fuente de la implementación:

```python
import random # Libreria para generar listas con numeros randoms.
import time # Libreria para visaulizar el tiempo de ejecución del programa.
import tracemalloc # Libreria para visualizar la cantidad de memoria que utiliza la ejecucion del programa.
from algoritmo_ordenamiento import quicksort # Importa la función de ordenamiento creada por el grupo.


"""
==========================================
        Funciones de Busqueda
==========================================
"""

# Función de busqueda por medio de Busqueda Binaria (Función reutilizada para comparar Búsqueda Binaria vs función generada por el grupo).
def BusquedaBinaria(lista, objetivo):
    if not isinstance(lista,list):
        print("El argumento 'lista' debe ser una lista.")
        return False

    izq = 0 # Índice inicial.
    derecha = len(lista) - 1 # Índice final.
    # Validar que no se crucen los límites izquierda y derecha.
    while izq <= derecha:
        mitad = izq + (derecha - izq) // 2 # Elemento central de la lista.
        if (lista[mitad] == objetivo): # Ver si el elemento del medio es igual al elemento que estamos buscando.
            return True
        elif (lista[mitad] < objetivo): # Si el elemento es mayor a la mitad, buscamos en la parte derecha.
            izq = mitad + 1
        else:
            derecha = mitad - 1 # Si el elemento es menor a la mitad, buscamos en la parte izquierda.
    return False

# Función que recorre cada una de las listas de la matriz y aplica la busqueda binaria.
def BusquedaBinaria_matriz(matriz, objetivo):
    if not isinstance(matriz, list):
        print("El argumento 'matriz' debe ser una lista.")
        return False
    for fila in matriz:
        if BusquedaBinaria(fila, objetivo):
            return True
    return False
```

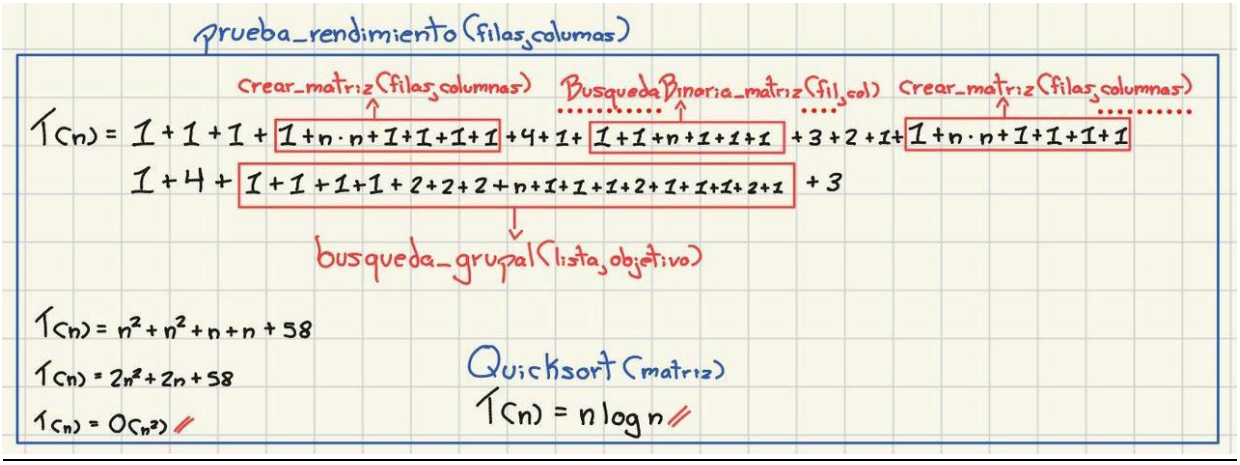
```python
# Función que recorre cada una de las listas de la matriz y aplica la función de busqueda realizada por el grupo.
def busqueda_matriz(matriz, objetivo):
    if not isinstance(matriz, list):
        print("El argumento 'matriz' debe ser una lista.")
        return False

    for lista in matriz:
        if busqueda_grupal(lista, objetivo):
            return True
    return False

# Función de busqueda creada por el grupo.
def busqueda_grupal(lista, objetivo):
    if not isinstance(lista,list):
        print("El argumento 'lista' debe ser una lista.")
        return False
    # Valida que la lista no esté vacía.
    if lista == []:
        return False
    mitad = len(lista) // 2 # Saca el índice central de la lista.
    izq = lista[:mitad] # Variable que almacena la lista de elementos del índice inicial a la derecha.
    derecha = lista[mitad:] # Variable que almacena la lista de los elementos desde el inicio de la lista hasta el índice central.
    # Mientras que ninguno de los extremos de la lista sea vacío se ejecuta.
    while (izq != [] or derecha != []):
        if izq != []:
            if izq[0] == objetivo:
                return True
            izq = izq[1:]

        if derecha != []:
            if derecha[0] == objetivo:
                return True
            derecha = derecha[1:]

    return False

# Función que crea una matriz con las filas y columnas dadas con elementos aleatorios.
def crear_matriz(filas,columnas):
    matriz = []
    for _ in range(filas):
        fila = []
        for _ in range(columnas):
            fila.append(random.randint(1, 100))
        matriz.append(fila)
    return matriz
```

```python
# Función de prueba de rendimiento
def prueba_rendimiento(filas,columnas):
    # Prueba de rendimiento para la función de búsqueda binaria.
    inicio = time.time() # Se inicia el tiempo de ejecución del programa.
    tracemalloc.start() #Se inicia el control de memoria durante la ejecución del programa.

    matriz = crear_matriz(filas,columnas)
    num_busqueda = matriz[random.randint(0, filas-1)][random.randint(0, columnas-1)]
    matriz2 = quicksort(matriz)

    BusquedaBinaria_matriz(matriz2, num_busqueda)

    fin = time.time() # Se finaliza el tiempo de ejecución del programa.
    memoria = tracemalloc.get_traced_memory() # Variable para almacerar la memoria utilizada.
    tracemalloc.stop()

    print(f"\nTiempo de ejecución (Búsqueda Binaria): {fin - inicio} segundos")
    print(f"Uso de memoria (Búsqueda Binaria): {memoria[1] / 10**6} MB\n")

    # Prueba de rendimiento para la función de búsqueda custom.
    inicio = time.time() # Se inicia el tiempo de ejecución del programa.
    tracemalloc.start() #Se inicia el control de memoria durante la ejecución del programa.

    matriz = crear_matriz(filas,columnas)
    num_busqueda = matriz[random.randint(0, filas-1)][random.randint(0, columnas-1)]
    matriz2 = quicksort(matriz)

    busqueda_grupal(matriz, num_busqueda)

    fin = time.time() # Se finaliza el tiempo de ejecución del programa.
    memoria = tracemalloc.get_traced_memory() # Variable para almacerar la memoria utilizada.
    tracemalloc.stop()

    print(f"Tiempo de ejecución (Búsqueda Grupal): {fin - inicio} segundos")
    print(f"Uso de memoria (Búsqueda Grupal): {memoria[1] / 10**6} MB")

prueba_rendimiento(100,100)
```

Explicación breve del algoritmo:

La función principal del algoritmo es "prueba_rendimiento" donde se inicia creando variables para el tiempo y espacio. Además, en otra variable donde se crea matriz con elementos aleatorios con la función "crear_matriz" con las filas y columnas dadas. Luego en esa matriz se escoge un numero para luego utilizarlo como numero de búsqueda en ambas funciones. Y se aplica la búsqueda con la Búsqueda Binaria y la Búsqueda Grupal.

Análisis de complejidad temporal incluyendo la función T(n):



Peor caso: El peor caso de la función de ordenamiento es de **O(n²)**

Mejor caso: La función de ordenamiento presenta como mejor caso **Ω (n log n)**

Caso promedio: El caso medio de la función de ordenamiento es de **Θ (n log n)**

Resultados experimentales para diferentes tamaños de entrada:

10x10

```
Tiempo de ejecución (Búsqueda Grupal): 0.0 segundos
Uso de memoria (Búsqueda Grupal): 0.002344 MB
```

100x100

```
Tiempo de ejecución (Búsqueda Grupal): 0.0 segundos
Uso de memoria (Búsqueda Grupal): 0.097256 MB
```

500x500

```
Tiempo de ejecución (Búsqueda Grupal): 0.2032322883605957 segundos
Uso de memoria (Búsqueda Grupal): 2.134936 MB
```

1000x1000

```
Tiempo de ejecución (Búsqueda Grupal): 0.8834898471832275 segundos
Uso de memoria (Búsqueda Grupal): 8.902144 MB
```

Comparación de eficiencia con otro algoritmo:

A continuación, se verá el rendimiento de Búsqueda Binaria con las entradas anteriores.

10x10

```
Tiempo de ejecución (Búsqueda Binaria): 0.0 segundos
Uso de memoria (Búsqueda Binaria): 0.002608 MB
```

100x100

```
Tiempo de ejecución (Búsqueda Binaria): 0.01072239875793457 segundos
Uso de memoria (Búsqueda Binaria): 0.09944 MB
```

500x500

```
Tiempo de ejecución (Búsqueda Binaria): 0.19716572761535645 segundos
Uso de memoria (Búsqueda Binaria): 2.13308 MB
```

1000x1000

```
Tiempo de ejecución (Búsqueda Binaria): 0.8739025592803955 segundos
Uso de memoria (Búsqueda Binaria): 8.914288 MB
```