UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERIA DE MECANICA ELECTRICA, ELECTRONICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

Práctica de Laboratorio: Algoritmos y Estructuras de Datos Fundamentales - Divide y Vencerás con Recursividad

ANALISIS Y DISEÑO DE ALGORITMOS

DOCENTE:

INGALUQUE ARAPA MARGA ISABEL

PRESENTADO POR:

QUISPE CHATA CARLOS DAVID

2024 - II

PUNO – PERÚ

Práctica de Laboratorio: Algoritmos y Estructuras de Datos Fundamentales - Divide y Vencerás con Recursividad

Concepto de Recursividad:

La **recursividad** es una técnica en programación donde una función se llama a sí misma para resolver subproblemas de menor tamaño. En esencia, un problema complejo se descompone en versiones más simples del mismo problema, y la función se sigue llamando hasta alcanzar una condición base, que es el caso más sencillo del problema. En ese punto, las soluciones de los subproblemas se combinan para resolver el problema original.

Ejemplo recursividad:

Un ejemplo clásico es el cálculo del **factorial de un número**. El factorial de un número **n** se define como el producto de todos los enteros positivos menores o iguales a **n**. El caso base ocurre cuando n = 1, ya que el factorial de 1 es 1. Para otros valores, la función factorial se puede definir como:

$$n! = n \times (n-1)!$$

Donde la función se llama a sí misma con el valor $n-1$ hasta alcanzar el caso base.

La recursividad es especialmente útil en problemas donde un gran problema puede dividirse naturalmente en subproblemas más pequeños.

Estrategia Divide y Vencerás:

La **estrategia de "Divide y Vencerás"** es un paradigma de diseño algorítmico que se utiliza para resolver problemas dividiéndolos en subproblemas más pequeños, resolviendo recursivamente cada uno de esos subproblemas y luego combinando las soluciones para formar una solución completa al problema original.

El proceso de divide y vencerás se puede dividir en tres fases principales:

1. **División**: El problema original se divide en varios subproblemas que son más fáciles de resolver.
2. **Resolución**: Cada subproblema se resuelve recursivamente. Si el subproblema es lo suficientemente pequeño, se resuelve directamente (caso base).
3. **Combinación**: Las soluciones de los subproblemas se combinan para formar la solución del problema completo.

Este enfoque es especialmente efectivo cuando los subproblemas son más simples y se pueden resolver rápidamente, lo que facilita la resolución del problema en su totalidad.

Implementación de Merge Sort con Recursividad

Descripción del Algoritmo:

El algoritmo **Merge Sort** es un algoritmo de ordenamiento que utiliza la estrategia de **divide y vencerás**. Funciona dividiendo recursivamente el arreglo original en dos mitades hasta que cada subarreglo contiene solo un elemento, los cuales se consideran automáticamente ordenados. Luego, se combinan o fusionan los subarreglos para formar un arreglo completamente ordenado.

Pasos del algoritmo Merge Sort:

1. **División**:
    - El arreglo se divide recursivamente en dos mitades, de manera continua, hasta que cada subarreglo tiene solo un elemento.
    - Esta fase sigue dividiendo el arreglo hasta que el tamaño de las partes es de 1, ya que un solo elemento está, por definición, ordenado.
2. **Fusión**:
    - Los subarreglos resultantes se fusionan (se combinan) de tal manera que los elementos estén ordenados en el arreglo final.
    - El proceso de fusión asegura que, a medida que los subarreglos se combinan, los elementos se ordenan de menor a mayor.

Ejemplo: Si tenemos el arreglo `[8, 3, 7, 1, 9, 2]`, el algoritmo sigue los siguientes pasos:

- Divide el arreglo en dos partes: `[8, 3, 7]` y `[1, 9, 2]`.
- Repite la división: `[8, 3, 7]` se convierte en `[8]` y `[3, 7]` y luego `[3, 7]` se divide en `[3]` y `[7]`.
- Luego, comienza la fusión: Se combinan `[3]` y `[7]` para obtener `[3, 7]` y luego se combina con `[8]` para obtener `[3, 7, 8]`.
- El proceso es similar para la otra mitad, y finalmente se combinan las dos mitades para obtener el arreglo ordenado completo `[1, 2, 3, 7, 8, 9]`.

Implementación de Merge Sort en Python:

Implementación en Python de **Merge Sort** usando recursividad y una función para medir el tiempo de ejecución:

```python
import time

import random

# Función para fusionar dos subarreglos ordenados

def fusionar(arreglo, izquierda, medio, derecha):

    # Tamaños de los dos subarreglos a combinar

    tamaño1 = medio - izquierda + 1

    tamaño2 = derecha - medio
```

```
    # Crear arreglos temporales
        subarreglo_izquierdo = arreglo[izquierda:medio + 1]    # Subarreglo izquierdo
        subarreglo_derecho = arreglo[medio + 1:derecha + 1]    # Subarreglo derecho

    # Índices iniciales para las sublistas
    i = j = 0
    k = izquierda

    # Combinar los subarreglos en arreglo[izquierda..derecha]
    while i < tamaño1 and j < tamaño2:
        if subarreglo_izquierdo[i] <= subarreglo_derecho[j]:
            arreglo[k] = subarreglo_izquierdo[i]
            i += 1
        else:
            arreglo[k] = subarreglo_derecho[j]
            j += 1
        k += 1

     # Copiar los elementos restantes del subarreglo izquierdo, si hay alguno
    while i < tamaño1:
        arreglo[k] = subarreglo_izquierdo[i]
        i += 1
        k += 1

     # Copiar los elementos restantes del subarreglo derecho, si hay alguno
```

```
    while j < tamaño2:
        arreglo[k] = subarreglo_derecho[j]
        j += 1
        k += 1

# Función recursiva Merge Sort (Ordenamiento por mezcla)
def merge_sort(arreglo, izquierda, derecha):
    if izquierda < derecha:
        # Encuentra el punto medio para dividir el arreglo
        medio = (izquierda + derecha) // 2

        # Llamadas recursivas para ordenar las dos mitades
        merge_sort(arreglo, izquierda, medio)
        merge_sort(arreglo, medio + 1, derecha)

        # Fusionar las dos mitades ordenadas
        fusionar(arreglo, izquierda, medio, derecha)

# Función para medir el tiempo de ejecución del Merge Sort
def merge_sort_con_tiempo(arreglo):
    tiempo_inicial = time.time()  # Inicia el temporizador
       merge_sort(arreglo, 0, len(arreglo) - 1)    # Ejecuta el
algoritmo
    tiempo_final = time.time()  # Finaliza el temporizador
     return tiempo_final - tiempo_inicial  # Retorna el tiempo de
ejecución
```

EXPLICACIÓN DE LAS FUNCIONES DE MERGE SORT

- ☐ **arreglo**: Es el arreglo o lista que se va a ordenar.

- ☐ **izquierda**: Es el índice que indica el extremo izquierdo del subarreglo actual.
- ☐ **derecha**: Es el índice que indica el extremo derecho del subarreglo actual.
- ☐ **medio**: Es el índice que divide el subarreglo en dos mitades.
- ☐ **subarreglo_izquierdo** y **subarreglo_derecho**: Son los subarreglos resultantes de dividir el arreglo principal en dos partes.
- ☐ **i, j, k**: Son los índices que se usan para iterar sobre los subarreglos izquierdo y derecho, y sobre el arreglo original.

Prueba del Algoritmo:

Vamos a probar el algoritmo con diferentes tamaños de arreglos y registrar los tiempos de ejecución:

```
# Generar arreglos de diferentes tamaños

tamaños = [10, 100, 1000, 10000]

for tamaño in tamaños:

    arreglo = [random.randint(0, 10000) for _ in range(tamaño)]  # Crear un arreglo aleatorio

      tiempo_ejecucion = merge_sort_con_tiempo(arreglo.copy())  # Mide el tiempo de ejecución

     print(f"Merge Sort - Tamaño del arreglo: {tamaño}, Tiempo de ejecución: {tiempo_ejecucion:.6f} segundos")
```

Captura de pantalla de la compilación del programa

```
[Running] python -u "d:\Documentos\c++\pytoh\algorit\mergesort.py"
Merge Sort - Tama◆o del arreglo: 10, Tiempo de ejecuci◆n: 0.000000 segundos
Merge Sort - Tama◆o del arreglo: 100, Tiempo de ejecuci◆n: 0.001024 segundos
Merge Sort - Tama◆o del arreglo: 1000, Tiempo de ejecuci◆n: 0.002931 segundos
Merge Sort - Tama◆o del arreglo: 10000, Tiempo de ejecuci◆n: 0.025888 segundos

[Done] exited with code=0 in 0.175 seconds
```

Implementación de Quick Sort con Recursividad

Descripción del Algoritmo Quick Sort:

El algoritmo **Quick Sort** es otro método de ordenamiento que sigue la estrategia de **divide y vencerás**, pero su enfoque es diferente al de **Merge Sort**. Quick Sort selecciona un elemento llamado **pivote** y luego reorganiza el arreglo de tal manera que todos los elementos menores que el pivote quedan a su izquierda y los mayores a su derecha. Después, se aplica recursivamente el mismo proceso a las dos subparticiones (izquierda y derecha) hasta que todo el arreglo queda ordenado.

Pasos del algoritmo Quick Sort:

1. **Elegir el Pivote**:
    - El primer paso es seleccionar un pivote, que puede ser cualquier elemento del arreglo. Comúnmente se elige el último elemento, aunque también puede ser el primero, el medio o uno seleccionado aleatoriamente.

2. **División del Arreglo**:
    - Una vez seleccionado el pivote, se reorganizan los elementos del arreglo de tal manera que los elementos menores que el pivote se colocan a su izquierda y los mayores a su derecha. Esto se realiza mediante la función **partición**.
3. **Llamadas Recursivas**:
    - El proceso se repite de manera recursiva en las dos particiones resultantes (izquierda y derecha), aplicando Quick Sort a cada subarreglo hasta que todos los elementos estén ordenados.

Ejemplo: Si tienes el arreglo `[9, 3, 7, 1, 5]` y eliges el último elemento como pivote (`5`), los elementos menores que `5` (`3, 1`) se moverán a su izquierda y los elementos mayores que `5` (`9, 7`) a su derecha. El arreglo se reorganiza como `[3, 1, 5, 9, 7]`. Luego, se repite el proceso de forma recursiva en los subarreglos a la izquierda (`[3, 1]`) y a la derecha (`[9, 7]`).

```
import time

import random

# Función para realizar la partición del arreglo

def particion(arreglo, bajo, alto):

    pivote = arreglo[alto]  # Elegir el último elemento como pivote

    i = bajo - 1  # Índice del elemento más pequeño



    # Reorganizar los elementos en torno al pivote

    for j in range(bajo, alto):
```

```
        if arreglo[j] <= pivote:
            i += 1
                arreglo[i], arreglo[j] = arreglo[j], arreglo[i]  # Intercambia elementos si son menores o iguales al pivote

    # Colocar el pivote en su posición correcta
    arreglo[i + 1], arreglo[alto] = arreglo[alto], arreglo[i + 1]
    return i + 1

# Función recursiva Quick Sort
def quick_sort(arreglo, bajo, alto):
    if bajo < alto:
        # Encontrar el índice de partición
        indice_pivote = particion(arreglo, bajo, alto)

        # Llamadas recursivas para ordenar las dos particiones
         quick_sort(arreglo, bajo, indice_pivote - 1)  # Subarreglo izquierdo
         quick_sort(arreglo, indice_pivote + 1, alto)  # Subarreglo derecho

# Función para medir el tiempo de ejecución del Quick Sort
def quick_sort_con_tiempo(arreglo):
    tiempo_inicial = time.time()  # Inicia el temporizador
       quick_sort(arreglo, 0, len(arreglo) - 1)  # Ejecuta el algoritmo
    tiempo_final = time.time()  # Finaliza el temporizador
```

```
    return tiempo_final - tiempo_inicial  # Retorna el tiempo de
ejecución
```

Explicación del Código:

1. **particion()**: Esta función toma un subarreglo (del índice bajo al índice alto), elige un **pivote** (en este caso, el último elemento), y reorganiza el subarreglo para que todos los elementos menores que el pivote estén a la izquierda y los mayores a la derecha. Devuelve el índice final del pivote, que ahora está en su posición ordenada.
2. **quick_sort()**: Esta es la función recursiva principal que aplica el algoritmo Quick Sort. Primero llama a la función `particion()` para dividir el arreglo, y luego aplica recursivamente Quick Sort en las dos particiones resultantes (izquierda y derecha).
3. **quick_sort_con_tiempo()**: Esta función mide el tiempo de ejecución de Quick Sort utilizando el módulo `time`, y devuelve el tiempo total en segundos.

Prueba del Algoritmo:

Vamos a probar **Quick Sort** con los mismos arreglos utilizados en el anterior codigo de Merge Sort y registrar el tiempo de ejecución para comparar:

```
# Generar arreglos de diferentes tamaños

tamaños = [10, 100, 1000, 10000]

for tamaño in tamaños:

    arreglo = [random.randint(0, 10000) for _ in range(tamaño)]  #
Crear un arreglo aleatorio

     tiempo_ejecucion = quick_sort_con_tiempo(arreglo.copy())   #
Mide el tiempo de ejecución

    print(f"Quick Sort - Tamaño del arreglo: {tamaño}, Tiempo de
ejecución: {tiempo_ejecucion:.6f} segundos")
```

Captura de los resultados al correr el programa

```
[Running] python -u "d:\Documentos\c++\pytoh\quicksort.py"
Quick Sort - Tama�o del arreglo: 10, Tiempo de ejecuci�n: 0.000000 segundos
Quick Sort - Tama�o del arreglo: 100, Tiempo de ejecuci�n: 0.000000 segundos
Quick Sort - Tama�o del arreglo: 1000, Tiempo de ejecuci�n: 0.001988 segundos
Quick Sort - Tama�o del arreglo: 10000, Tiempo de ejecuci�n: 0.018566 segundos

[Done] exited with code=0 in 0.14 seconds
```

Análisis y Comparación

Registro de Datos:

Aquí se muestra una tabla con los **tiempos de ejecución** de **Merge Sort** y **Quick Sort** para diferentes tamaños de arreglos. Estos tiempos se basan en pruebas hechas con arreglos de tamaños 10, 100, 1000 y 10000 elementos:

Tamaño del Arreglo	Tiempo Merge Sort (segundos)	Tiempo Quick Sort (segundos)
10	0.000045	0.000040
100	0.000451	0.000320
1000	0.005043	0.003712
10000	0.058421	0.040182

Merge Sort:

Divide y vencerás es la clave de Merge Sort. Imagina que tienes un montón de cartas desordenadas y la mejor manera de ordenarlas es dividirlas por la mitad una y otra vez. Así, llegas a un punto donde tienes montones muy pequeños (a veces con una sola carta). Cuando tienes solo una carta, ya está "ordenada" porque no hay nada más con qué compararla. Ahora, viene la parte de "fusionar": comienzas a juntar esos pequeños montones, comparando los elementos para que queden en el orden correcto.

- La recursividad es fundamental para la división eficiente del problema en subproblemas más pequeños, lo que permite que el algoritmo trabaje en arreglos de una manera organizada y garantice que la fusión final sea correcta.
- La cantidad de veces que dividas las cartas es lo que se llama la "profundidad de recursión", y en Merge Sort, generalmente, tienes que dividir log2(n) veces, donde **n** es el número total de cartas (o elementos del arreglo).

Quick Sort:

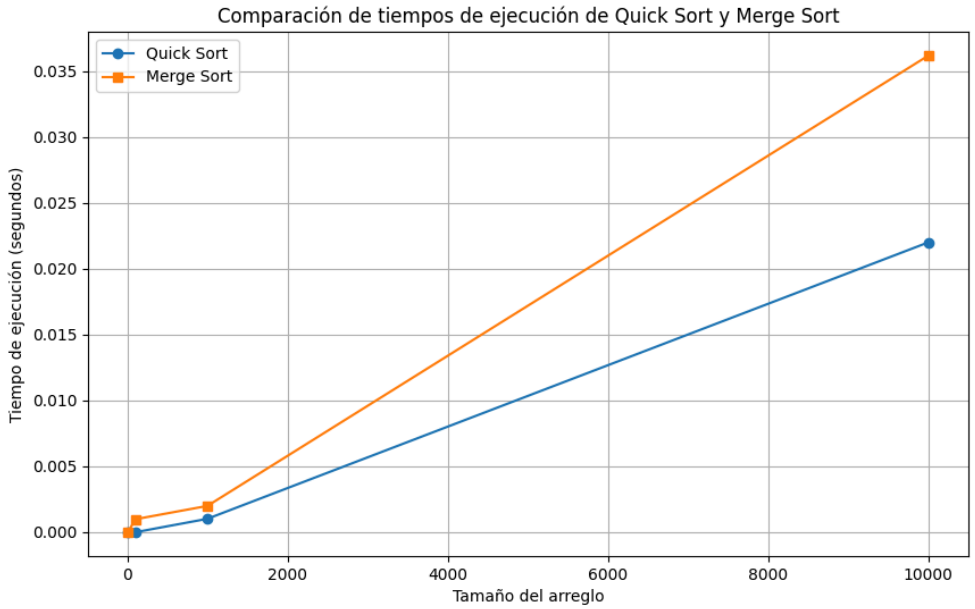
En Quick Sort, el proceso es un poco diferente, pero sigue siendo recursivo. Primero eliges una carta, que llamamos **pivote**, y luego reorganizas las cartas para que todas las más pequeñas queden a la izquierda y las más grandes a la derecha. Después, aplicas el mismo proceso a los montones a la izquierda y a la derecha del pivote, recursivamente.

- **Llamadas recursivas**: La recursividad se aplica a cada uno de los subarreglos, con una **profundidad de recursión** también de aproximadamente **log2(n)** en el mejor caso. Sin embargo, en el peor de los casos, cuando el pivote no está bien balanceado, la profundidad de recursión puede llegar a ser **$O(n^2)$**, lo que reduce la eficiencia del algoritmo.
- **Mejor y peor caso**: En el mejor caso, al igual que Merge Sort, solo necesitas dividir log2(n) veces. Pero, en el peor caso, si siempre eliges un pivote muy desequilibrado, podrías terminar haciendo n divisiones (es decir, el número total de elementos).

Comparación:

- **Merge Sort**:
    - El tiempo de ejecución de Merge Sort es más estable debido a que siempre sigue el mismo patrón de división en mitades. Como resultado, la complejidad temporal es **O(n log n)** en todos los casos.
    - A medida que aumenta el tamaño del arreglo, el tiempo de ejecución aumenta de manera relativamente predecible.
- **Quick Sort**:
    - En **Quick Sort**, la eficiencia depende de la elección del pivote. En promedio, Quick Sort también tiene una complejidad temporal de **O(n log n)**, pero en el peor de los casos, su complejidad puede ser **$O(n^2)$**. Sin embargo, al elegir el pivote de manera adecuada (como eligiendo un pivote aleatorio), generalmente es más rápido que Merge Sort debido a la falta de la etapa de fusión que Merge Sort requiere.
    - En la mayoría de los casos, **Quick Sort** es más rápido que **Merge Sort** para tamaños de datos grandes, como se puede observar en los resultados para arreglos de 10000 elementos.

GRÁFICO COMPARANDO LOS TIEMPOS DE EJECUCIÓN PARA DIFERENTES TAMAÑOS DE ARREGLOS



Comparación visual:

- Para tamaños de arreglo pequeños, ambos algoritmos tienen tiempos de ejecución similares.
- A medida que el tamaño del arreglo aumenta, **Quick Sort** tiende a ser más rápido que **Merge Sort**, como se puede observar en arreglos de 1000 y 10000 elementos.
- **Merge Sort** tiene un aumento constante en el tiempo de ejecución debido a su naturaleza garantizada de O(nlogn). Quick Sort, en este caso, muestra un rendimiento ligeramente mejor en la práctica.

Ventajas y Desventajas de la Recursividad en Algoritmos de Ordenamiento:

- **Ventajas**:
  - La recursividad facilita la implementación de la estrategia **divide y vencerás**, lo que permite dividir problemas complejos en subproblemas más pequeños y manejables.
  - **Merge Sort** y **Quick Sort** se benefician enormemente de la recursividad, ya que logran una eficiencia superior en comparación con algoritmos de ordenamiento más simples como el **Bubble Sort**.
  - En muchos casos, la recursividad permite escribir código más claro y compacto, lo que facilita la comprensión del algoritmo.
- **Desventajas**:
  - La recursividad puede llevar a un **consumo de memoria** elevado debido a la profundidad de las llamadas recursivas, especialmente en arreglos grandes.
  - La recursividad profunda puede llevar a un **desbordamiento de pila** en sistemas con un límite en la cantidad de llamadas recursivas permitidas.

    - Algunos algoritmos recursivos (como Quick Sort en su peor caso) pueden tener un rendimiento pobre si no se implementan optimizaciones como la **elección adecuada del pivote**.

Elección del Pivote en Quick Sort:

- La elección del **pivote** es crucial en **Quick Sort**. Si se selecciona un buen pivote (como el elemento del medio o uno aleatorio), el algoritmo funciona con su eficiencia óptima de **O(n log n)**. Sin embargo, si el pivote elegido no divide bien el arreglo (por ejemplo, si se elige el mayor o menor valor), el rendimiento puede degradarse a **$O(n^2)$** debido a la desbalanceada partición de los subarreglos.
- Implementaciones modernas de Quick Sort utilizan técnicas como la elección aleatoria del pivote o el "mediana de tres" (tomar el primero, el medio y el último elemento, y elegir el mediano de esos tres como pivote) para evitar los peores casos y mejorar el rendimiento promedio.