Merge Sort: Un Algoritmo de Ordenación Eficiente

1 Introducción

la palabra Merge Sort se podría interpretar como "Ordenamiento por fusión", ya que "merge" significa fusionar y "sort" significa ordenar. Este término describe exactamente cómo funciona el algoritmo: divide la lista en partes más pequeñas, las ordena individualmente y luego las fusiona de forma ordenada. este un algoritmo de ordenación eficiente, general y basado en la comparación. Se fundamenta en la estrategia de *divide y vencerás*, dividiendo un conjunto de datos en mitades, ordenándolas de manera recursiva y luego fusionándolas en una lista ordenada.

2 Explicación de la Notación Big-O

La notación Big-O, representada como $O(f(n))$, se utiliza para describir el crecimiento de la cantidad de operaciones necesarias en un algoritmo en función del tamaño de la entrada n.

2.1 Concepto de Big-O

Big-O nos da una idea de cómo aumenta el tiempo de ejecución cuando n crece, ignorando constantes y términos de menor orden. Algunas de las complejidades más comunes son:

- $O(1)$ - Tiempo constante: El tiempo de ejecución no cambia sin importar el tamaño de la entrada. Ejemplo: acceder a un elemento de un arreglo.
- $O(n)$ - Tiempo lineal: El tiempo de ejecución crece proporcionalmente a n. Ejemplo: recorrer una lista.
- $O(n^2)$ - Tiempo cuadrático: Se usa en algoritmos menos eficientes, como Bubble Sort.
- $O(n \log n)$ - Tiempo subcuadrático: Más rápido que $O(n^2)$, pero más lento que $O(n)$, típico en algoritmos eficientes de ordenamiento como Megesort y Merge Sort.

2.2 Por qué Megesort es $O(n \log n)$

El algoritmo Megesort tiene dos fases principales:

1. **División de la lista**: Se divide la lista en mitades hasta que cada sublista contiene un solo elemento. Como cada división reduce el tamaño de la lista a la mitad, el número total de divisiones es $O(\log n)$.

2. **Fusión de las sublistas**: Una vez divididas, las sublistas se combinan en orden. Cada nivel de la fusión requiere recorrer todos los elementos, lo que toma $O(n)$ operaciones.

Como hay $O(\log n)$ niveles de división y cada nivel requiere $O(n)$ operaciones, la complejidad total es:

$$O(n) \times O(\log n) = O(n \log n)$$

Esto hace que Megesort sea significativamente más rápido que algoritmos con $O(n^2)$ en listas grandes.

3 Características

- **Eficiencia:** Su tiempo de ejecución en el caso promedio y peor caso es de $O(n \log n)$. Esto significa que el número de operaciones necesarias para ordenar una lista de n elementos crece proporcionalmente a n multiplicado por $\log n$.

  En comparación con algoritmos como Bubble Sort o Insertion Sort, que tienen una complejidad de $O(n^2)$ en el peor caso, Megesort es mucho más eficiente en listas grandes.

- **Ordenación estable:** Conserva el orden relativo de los elementos iguales, lo que es útil en aplicaciones donde este orden tiene significado.

- **Paralelización:** Es altamente paralelizable, ya que las diferentes partes del conjunto de datos pueden ordenarse simultáneamente antes de combinarse, aprovechando arquitecturas multiprocesador y mejorando su rendimiento en comparación con algoritmos que dependen de una ejecución secuencial.

4 Historia

El algoritmo Merge Sort fue inventado por John von Neumann en 1945. Un análisis detallado de su implementación ascendente fue publicado por Goldstine y von Neumann en 1948.

5 Funcionamiento del Algoritmo

El algoritmo sigue los siguientes pasos:

1. Dividir el conjunto de datos en mitades.
2. Ordenar recursivamente cada mitad.
3. Fusionar las mitades ordenadas en una sola lista ordenada.

5.1 Ejemplo Manual de Ejecución

Supongamos que tenemos el siguiente arreglo desordenado:

$$[12, 8, 9, 3, 11, 5, 4]$$

Dividimos el arreglo en mitades sucesivas:

$$[12, 8, 9][3, 11, 5, 4]$$

$$[12][8, 9][3, 11][5, 4]$$

$$[12][8][9][3][11][5][4]$$

Luego comenzamos la fusión comparando los elementos:

$$[8, 9][12][3, 11][4, 5]$$

$$[8, 9, 12][3, 4, 5, 11]$$

$$[3, 4, 5, 8, 9, 11, 12]$$

El proceso de división y combinación se realiza de forma recursiva, asegurando que cada subarreglo fusionado mantenga el orden correcto.

6 Implementación en C++

A continuación, se muestra la implementación de Merge Sort en C++ y su explicación.

```
#include <iostream> // Biblioteca para entrada y salida estándar
#include <vector>   // Biblioteca para manejar arreglos dinámicos (vectores)

using namespace std;
// Declaración de la función merge antes de su uso
void merge(vector<int>& arr, int izquierda, int pmedio, int derecha);


// Función recursiva para dividir y ordenar el arreglo usando Merge Sort
void mergeSort(vector<int>& arr, int izquierda, int derecha) {
```

```
    if (izquierda < derecha) { // Si hay más de un elemento en la sección a ordenar
        int pmedio = izquierda + (derecha - izquierda) / 2; // Calcula el punto medio

        // Llamadas recursivas para ordenar cada mitad del arreglo
        mergeSort(arr, izquierda, pmedio);       // Ordena la mitad izquierda
        mergeSort(arr, pmedio + 1, derecha);     // Ordena la mitad derecha

        // Fusiona las dos mitades ordenadas en el arreglo original
        merge(arr, izquierda, pmedio, derecha);
    }
}

// Función para fusionar dos subarreglos ordenados en el arreglo original
void merge(vector<int>& arr, int izquierda, int pmedio, int derecha) {
    // Calcular los tamaños de los subarreglos
    int n1 = pmedio - izquierda + 1; // Tamaño de la primera mitad del arreglo
    int n2 = derecha - pmedio;       // Tamaño de la segunda mitad del arreglo

    // Crear los subarreglos temporales para almacenar las mitades del arreglo original
    vector<int> L(n1), R(n2);

    // Copiar elementos de la mitad izquierda en el subarreglo L
    for (int i = 0; i < n1; i++)
        L[i] = arr[izquierda + i];

    // Copiar elementos de la mitad derecha en el subarreglo R
    for (int i = 0; i < n2; i++)
        R[i] = arr[pmedio + 1 + i];

    // Inicializar índices para recorrer los subarreglos y el arreglo original
    int i = 0, j = 0, k = izquierda;

    // Fusionar los dos subarreglos en el arreglo original de forma ordenada
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) { // Si el elemento en L es menor o igual que el de R
            arr[k++] = L[i++]; // Insertamos el elemento de L en arr y avanzamos en L
        } else {
            arr[k++] = R[j++]; // Insertamos el elemento de R en arr y avanzamos en R
        }
    }

    // Si quedan elementos en L, copiarlos al arreglo original
    while (i < n1)
        arr[k++] = L[i++];

    // Si quedan elementos en R, copiarlos al arreglo original
```

```
    while (j < n2)
        arr[k++] = R[j++];
}

// Función principal para recibir el arreglo desde el usuario y ordenarlo
int main() {
    int n;

    // Solicitar al usuario el número de elementos del arreglo
    cout << "Ingrese el numero de elementos del arreglo: ";
    cin >> n;

    // Validación: el número de elementos debe ser positivo
    if (n <= 0) {
        cout << "El número de elementos debe ser positivo." << endl;
        return 1; // Termina la ejecución del programa si el número no es válido
    }

    // Crear un vector con el tamaño ingresado por el usuario
    vector<int> arr(n);

    // Solicitar los elementos del arreglo al usuario
    cout << "Ingrese los elementos del arreglo: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    // Llamar a la función mergeSort para ordenar el arreglo
    mergeSort(arr, 0, arr.size() - 1);

    // Mostrar el arreglo ordenado
    cout << "Arreglo ordenado: ";
    for (int num : arr)
        cout << num << " ";
    cout << endl;

    return 0; // Termina el programa correctamente
}
```

La función `mergeSort` divide el arreglo en mitades y llama a `merge` para combinar las mitades ordenadas. La función `merge` fusiona los subarreglos comparando los elementos de manera ordenada.

7 Implementación en Python

A continuación, se presenta el código de Merge Sort en Python con explicación.

```
# Función para fusionar dos subarreglos ordenados
def merge(arr, izquierda, medio, derecha):
    # Determina el tamaño de los subarreglos izquierdo (L) y derecho (R)
    n1 = medio - izquierda + 1  # Tamaño del subarreglo izquierdo
    n2 = derecha - medio        # Tamaño del subarreglo derecho

    # Crear los subarreglos L (izquierdo) y R (derecho) copiando elementos desde el arreglo
    L = arr[izquierda:izquierda + n1]  # Subarreglo izquierdo
    R = arr[medio + 1:medio + 1 + n2]  # Subarreglo derecho

    # Inicializar los índices para recorrer los subarreglos (L y R) y el arreglo original
    i = 0  # Índice para recorrer el subarreglo izquierdo (L)
    j = 0  # Índice para recorrer el subarreglo derecho (R)
    k = izquierda  # Índice para insertar valores en el arreglo original

    # Fusionar los subarreglos ordenados en el arreglo original
    while i < n1 and j < n2:  # Mientras haya elementos en ambos subarreglos
        if L[i] <= R[j]:  # Si el elemento en L es menor o igual que el de R
            arr[k] = L[i]  # Coloca el elemento de L en la posición actual del arreglo origi
            i += 1  # Avanza el índice de L
        else:  # Si el elemento en R es menor que el de L
            arr[k] = R[j]  # Coloca el elemento de R en la posición actual del arreglo origi
            j += 1  # Avanza el índice de R
        k += 1  # Avanza al siguiente índice del arreglo original

    # Copiar los elementos restantes de L al arreglo original (si quedan)
    while i < n1:  # Si aún hay elementos en L
        arr[k] = L[i]  # Copia el elemento de L al arreglo original
        i += 1  # Avanza el índice de L
        k += 1  # Avanza el índice del arreglo original

    # Copiar los elementos restantes de R al arreglo original (si quedan)
    while j < n2:  # Si aún hay elementos en R
        arr[k] = R[j]  # Copia el elemento de R al arreglo original
        j += 1  # Avanza el índice de R
        k += 1  # Avanza el índice del arreglo original

# Función recursiva para dividir y ordenar el arreglo
def mergeSort(arr, izquierda, derecha):
    if izquierda < derecha:  # Verifica si la sección del arreglo tiene más de un elemento
        # Encontrar el punto medio del arreglo
        medio = (izquierda + derecha) // 2  # Calcula el índice del punto medio
```

```
        # Llamada recursiva para ordenar la mitad izquierda del arreglo
        mergeSort(arr, izquierda, medio)

        # Llamada recursiva para ordenar la mitad derecha del arreglo
        mergeSort(arr, medio + 1, derecha)

        # Fusionar las dos mitades ordenadas
        merge(arr, izquierda, medio, derecha)  # Se combinan en el arreglo original

# Función principal
if __name__ == "__main__":
    # Solicitar al usuario el número de elementos del arreglo
    n = int(input("Ingrese el número de elementos del arreglo: "))  # Toma el tamaño del arr

    # Crear el arreglo con los valores ingresados por el usuario
    arr = list(map(int, input("Ingrese los elementos del arreglo: ").split()))  # Convierte

    # Llamar a la función mergeSort para ordenar el arreglo
    mergeSort(arr, 0, len(arr) - 1)  # Ordena el arreglo desde el índice 0 hasta el último

    # Mostrar el arreglo ordenado
    print("Arreglo ordenado:", *arr)  # Imprime el arreglo ordenado usando * para desempaque
```

Esta implementación en Python sigue el mismo principio que en C++, dividiendo el arreglo y ordenando cada mitad antes de fusionarlas de nuevo en orden correcto.

8 Conclusión

Merge Sort es un algoritmo eficiente y estable, ideal para ordenar grandes volúmenes de datos y ampliamente utilizado en informática debido a su capacidad de paralelización y consistencia en el rendimiento.