8. Algoritmos de ordenamiento: Ordenamiento por mezcla

Carlos Zerón Martínez

Universidad Nacional Autónoma de México

zeron@ciencias.unam.mx

Jueves 14 de Enero de 2021

Paradigma de algoritmos "Divide y Vencerás"

La estrategia general consiste en tres pasos:

- **Dividir**. Se dividen los datos de entrada original en una o más entradas de menor tamaño.
- **Recurrir**. Resolver recursivamente el problema para las partes obtenidas de la entrada.
- **Vencer**. Tomar las soluciones de las entradas de menor tamaño y combinarlas para obtener una solución para la entrada original.

Paradigma de algoritmos "Divide y Vencerás": *Quicksort*

Quicksort

- **Dividir**. Se selecciona un elemento específico x del arreglo (pivote) y se divide en tres partes:
  - L: elementos menores que x.
  - E: el pivote x.
  - G: elementos mayores o iguales que x.
- **Recurrir**. Ordenar recursivamente las partes del arreglo L y G
- **Vencer**. Trivial, porque quedan ordenados los elementos de L, luego el pivote y posteriormente quedan ordenados los elementos de G

Paradigma de algoritmos "Divide y Vencerás": *Mergesort*

Mergesort (Ordenamiento por mezcla)

Para ordenar un arreglo de objetos comparables, el algoritmo sigue la estrategia de la siguiente manera:

- **Dividir**. Dividir el arreglo de elementos a ordenar en dos mitades
- **Recurrir**. Ordenar cada mitad recursivamente
- **Vencer**. Mezclar ambas mitades en un arreglo ordenado.

Mergesort

- Límite para dividir: al obtener subarreglos de 0 ó 1 elemento (ordenados)
- El proceso de mezcla toma dos mitades del arreglo original ordenadas y las pone en arreglos auxiliares:
  - la primera comprende de la posición *first* a *mid*
  - la segunda comprende de la posición $mid + 1$ a *last*

  donde $first \leq mid < last$.
- En cada paso se comparan el mínimo elemento de la primera mitad con el mínimo de la otra mitad, regresando el mínimo al arreglo original. Acabamos cuando ya no hay más elementos en alguna de las mitades. El orden en que se regresan los elementos al arreglo original deja ordenada la parte de *first* a *last*

Mergesort

Para evitar que en cada iteración se tenga que hacer la verificación de que alguna de las mitades se quedó vacía, al final de cada arreglo temporal se pone un centinela (localidad nula) que representa un ∞, al momento de alcanzarlo en una mitad se determina que siempre se regresen elementos de la otra mitad al arreglo original.

El proceso termina cuando quedan expuestos los centinelas de ambas mitades.

Comparación entre elementos

```
public class MergeSort {
    /**
     * Compara dos objetos comparables. Si uno de los dos es nulo, se asume que
     * el valor es infinito. Regresa:
     * a) un entero negativo si el primero es menor que el segundo;
     * b) 0 si ambos objetos son iguales y
     * c) un entero positivo si el primero es mayor que el segundo.
     */
    private int compare(Comparable x, Comparable y) {
        if (x == null) {
            if (y == null) {
                return 0;  // ambos son infinitos
            }
            return 1;  // x es infinito, y no lo es
        } else if (y == null) {
            return -1; // y es infinito, x no lo es
        }
        return x.compareTo(y); // ninguno es infinito
    }
```

Mezcla de subarreglos (arreglos auxiliares)

```java
/**
 * Mezcla dos mitades del arreglo de elementos comparables considerando como
 * primera mitad, desde la posición first hasta la posición mid y como
 * segunda mitad, desde la posición mid + 1 hasta la posicion last.
 */
private void merge(Comparable[] array, int first, int mid, int last) {
    int n1 = mid - first + 1; // elementos en la primera mitad
    int n2 = last - mid; // elementos en la segunda mitad

    // arreglo temporal correspondiente a la primera mitad
    Comparable[] left = new Comparable[n1 + 1];
    for (int i = 0; i < n1; i++) {
        left[i] = array[first + i];
    }

    // arreglo temporal correspondiente a la segunda mitad
    Comparable[] right = new Comparable[n2 + 1];
    for (int j = 0; j < n2; j++) {
        right[j] = array[mid + j + 1];
    }

    // se hacen infinitos los valores
    // en las ultimas posiciones
    left[n1] = null;
    right[n2] = null;
```

Mezcla de subarreglos (comparaciones)

```
    for (int izq = 0, der = 0, indice = first; indice <= last; indice++) {
        if (compare(left[izq], right[der]) <= 0) {
            array[indice] = left[izq++];
        } else {
            array[indice] = right[der++];
        }
    }
}
```

Código de MergeSort

```
/*
 * Ordena el arreglo que le pasan, de las posiciones first
 * a last, por mezcla (Mergesort).
 */
private void mergeSort(Comparable[] array, int first, int last) {
    if (first < last) {
        int mid = (first + last) / 2;
        mergeSort(array, first, mid);
        mergeSort(array, mid + 1, last);
        merge(array, first, mid, last);
    }
}
```

Ordenando con MergeSort

```
    /**
     * Ordena el arreglo por mezcla (usando mergesort).
     *
     * @param array El arreglo a ordenar.
     */
    public void sort(Comparable[] array) {
        mergeSort(array, 0, array.length - 1);
    }

}
```

Complejidad (Intuición)

- El peor caso de este algoritmo de ordenamiento es $O(n \log n)$.
- Para ordenar n elementos, es necesario ordenar dos subarreglos con $n/2$ elementos cada uno. El número de veces que se puede dividir entre 2 el subarreglo resultante es a lo más $\log_2 n$.
- Se requieren $O(n)$ operaciones para la mezcla de ambos subarreglos.
- Es uno de los algoritmos más eficientes en la teoría, sin embargo tiene el inconveniente de usar más memoria que otros, pues se requiere memoria adicional para el proceso de mezcla de soluciones.