Algoritmos de Ordenación en Java

Introducción

Los algoritmos de ordenación son herramientas fundamentales en la programación. Su propósito es organizar una colección de elementos (como números, letras, objetos) en un orden específico (ascendente, descendente, alfabético, etc.). La eficiencia con la que un algoritmo realiza esta tarea incide en el rendimiento de nuestras aplicaciones.

Os presento cinco algoritmos de ordenación básicos, junto con su pseudocódigo y un análisis de su complejidad (aunque este aspecto queda fuera de nuestro estudio). La práctica con estos algoritmos nos ayuda en la programación y comprender las consecuencias de la elección de un algoritmo u otro.

Algoritmos de Ordenación Básicos

Método Burbuja (Bubble Sort)

Es un método sencillo que compara pares de elementos adyacentes y los intercambia si están desordenados. No es el algoritmo de ordenación más eficiente, especialmente para listas grandes.

Pseudocódigo

```
Algoritmo Burbuja(array)
    Para i desde 1 hasta longitud(array) - 1 hacer
        Para j desde 0 hasta longitud(array) - i - 1 hacer
            Si array[j] > array[j+1] entonces
            // Intercambiar elementos
            temp ← array[j]
            array[j] ← array[j+1]
            array[j+1] ← temp
            Fin si
        Fin para
    Fin para
Fin Algoritmo
```

Explicación

El algoritmo de burbuja compara pares de elementos adyacentes en la lista. Si dos elementos están en el orden incorrecto, los intercambia. Este proceso se repite varias veces, "burbujeando" los elementos más grandes hacia el final de la lista en cada pasada.

Análisis de Complejidad

```
Temporal:
    Peor caso: O(n^2) - Cuando la lista está en orden inverso.
```

```
    Caso promedio: O(n^2)
    Mejor caso: O(n) - Cuando la lista ya está ordenada.

Espacial: O(1) - Usa espacio adicional constante.

Estabilidad: Estable - Los elementos con el mismo valor mantienen su orden
relativo.
```

Ventajas y Desventajas

```
Ventajas:
    Fácil de entender e implementar.

Desventajas:
    Ineficiente para listas grandes.
```

Ordenación por Selección (Selection Sort)

Pseudocódigo

```
Algoritmo Selección(lista)
   Para i desde 0 hasta longitud(lista) - 2 hacer
       min_idx ← i
       Para j desde i + 1 hasta longitud(lista) - 1 hacer
           Si lista[j] < lista[min_idx] entonces
               min_idx ← j
           Fin si
       Fin para
       // Intercambiar lista[i] y lista[min_idx]
       temp ← lista[min_idx]
       lista[min_idx] ← lista[i]
       lista[i] ← temp
   Fin para
Fin Algoritmo
```

El algoritmo de selección busca el elemento más pequeño en la lista y lo coloca en su posición correcta al inicio. Luego busca el siguiente más pequeño y lo coloca en la siguiente posición, y así sucesivamente.

Análisis de Complejidad

```
Temporal:
    Peor caso: O(n^2)
    Caso promedio: O(n^2)
    Mejor caso: O(n^2)
```

```
Espacial: O(1)
Estabilidad: No estable
```

Ventajas y Desventajas

```
Ventajas:
    Fácil de entender e implementar.
    Relativamente simple.

Desventajas:

    Ineficiente para listas grandes.
```

Ordenación por Inserción (Insertion Sort)

Pseudocódigo

```
Algoritmo Inserción(lista)
    Para i desde 1 hasta longitud(lista) - 1 hacer
        valor <- lista[i]
        j <- i - 1
        Mientras j >= 0 y lista[j] > valor hacer
            lista[j + 1] <- lista[j]
            j <- j - 1
        Fin Mientras
        lista[j + 1] <- valor
    Fin Para
Fin Algoritmo
```

Explicación

El algoritmo de inserción construye la lista ordenada insertando cada elemento en su posición correcta dentro de la parte ya ordenada de la lista.

Análisis de Complejidad

```
Temporal:
    Peor caso: O(n^2) - Cuando la lista está en orden inverso.
    Caso promedio: O(n^2)
    Mejor caso: O(n) - Cuando la lista ya está ordenada.

Espacial: O(1)

Estabilidad: Estable
```

Ventajas y Desventajas

```
Ventajas:
    Eficiente para listas pequeñas o listas casi ordenadas.

Desventajas:
    Ineficiente para listas grandes.
```

Ordenación Rápida (Quick Sort)

Pseudocódigo

```
ALGORITMO Quicksort(lista, inicio, fin)
    SI inicio < fin ENTONCES
        pivote = particion(lista, inicio, fin)  // Encuentra la posición correcta
del pivote
        Quicksort(lista, inicio, pivote - 1)    // Ordena la parte izquierda del
pivote
        Quicksort(lista, pivote + 1, fin)       // Ordena la parte derecha del
pivote
    FIN SI
FIN ALGORITMO

ALGORITMO particion(lista, inicio, fin)
    pivote = lista[fin]              // Escoge el último elemento como pivote
    i = inicio - 1                   // Índice del menor elemento encontrado
    PARA j DESDE inicio HASTA fin - 1 HACER
        SI lista[j] <= pivote ENTONCES
            i = i + 1
            intercambiar(lista[i], lista[j])  // Coloca el elemento menor que el
pivote en su lugar
        FIN SI
    FIN PARA
    intercambiar(lista[i + 1], lista[fin])   // Coloca el pivote en su posición
correcta
    RETORNAR i + 1                           // Devuelve la posición del pivote
FIN ALGORITMO
```

Explicación

El algoritmo Quick Sort es un algoritmo de divide y vencerás. Divide la lista en sublistas más pequeñas, ordenándolas de forma recursiva y utilizando un elemento pivote.

Análisis de Complejidad

```
Temporal:
    Peor caso: O(n^2)
    Caso promedio: O(n log n)
    Mejor caso: O(n log n)

Espacial: O(log n)

Estabilidad: No estable
```

Ventajas y Desventajas

```
Ventajas:
    Muy eficiente en promedio.

Desventajas:
    Puede ser ineficiente en el peor caso.
    Puede ser más difícil de implementar correctamente que otros.
```

Ordenación por Mezcla (Merge Sort)

Pseudocódigo

```
ALGORITMO Mergesort(lista)
    SI longitud(lista) > 1 ENTONCES
        mitad = longitud(lista) // 2
        izquierda = lista[0:mitad]
        derecha = lista[mitad:longitud(lista)]

        Mergesort(izquierda)
        Mergesort(derecha)

        i = 0, j = 0, k = 0
        MIENTRAS i < longitud(izquierda) Y j < longitud(derecha) HACER
            SI izquierda[i] < derecha[j] ENTONCES
                lista[k] = izquierda[i]
                i = i + 1
            SINO
                lista[k] = derecha[j]
                j = j + 1
            FIN SI
            k = k + 1
        FIN MIENTRAS

        MIENTRAS i < longitud(izquierda) HACER
            lista[k] = izquierda[i]
            i = i + 1
            k = k + 1
```

```
        FIN MIENTRAS

        MIENTRAS j < longitud(derecha) HACER
            lista[k] = derecha[j]
            j = j + 1
            k = k + 1
        FIN MIENTRAS
    FIN SI
FIN ALGORITMO
```

Explicación

El algoritmo Merge Sort también utiliza la técnica de "divide y vencerás". Divide la lista en mitades, las ordena recursivamente y luego las combina en un único array ordenado.

Análisis de Complejidad

```
Temporal:
    Peor caso: O(n log n)
    Caso promedio: O(n log n)
    Mejor caso: O(n log n)

Espacial: O(n)

Estabilidad: Estable
```

Ventajas y Desventajas

```
Ventajas:
    Eficiente y estable.

Desventajas:
    Utiliza memoria adicional.

    Algo más complejo de entender que otros métodos.
```

Conclusión

Cada uno de los algoritmos de ordenación presentados tiene sus propias características en cuanto a complejidad, estabilidad y facilidad de implementación. La elección del algoritmo adecuado dependerá del contexto del problema, el tamaño de los datos, y otros factores.

Por supuesto, no es una lista exhaustiva de algoritmos de ordenación, existen muchos más, pero el conocimiento de estos se considera básico.

Enlaces

https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
https://www.youtube.com/watch?v=pqZ04TT15PQ
https://www.youtube.com/watch?v=hf-_c7DFb3U