Algoritmos de ordenación

Jorge Mario Londoño Peláez & Varias AI

March 11, 2025

1 Algoritmos de ordenación

1.1 Definición y Aplicaciones

Ordenar un conjunto de datos significa reorganizarlo en una secuencia específica, ya sea ascendente o descendente, según una relación de orden predefinida. Esta relación de orden define un conjunto totalmente ordenado, donde para cualquier par de elementos, se puede determinar cuál es mayor o menor.

Las aplicaciones de la ordenación son vastas:

- **Bases de datos:** La ordenación es fundamental para indexar datos y acelerar las consultas.
- **Generación de reportes:** Los datos ordenados facilitan la creación de informes claros y concisos.
- **Búsqueda:** La búsqueda de elementos en un conjunto ordenado es mucho más eficiente (e.g., búsqueda binaria).
- **Compresión de datos:** Algunos algoritmos de compresión se benefician de datos ordenados.

Características clave:

- **Datos:** Un conjunto de elementos a ordenar.
- **Relación de orden:** Define cómo se comparan los elementos entre sí.

1.2 Comparación de ADTs

Para comparar ADT, los lenguajes de programación ofrecen mecanismos específicos:

- **Java:** La interfaz `Comparable` permite definir un orden natural para los objetos de una clase. La clase debe implementar el método `compareTo()`, que devuelve un valor negativo, cero o positivo si el objeto es menor, igual o mayor que el objeto con el que se compara.
  Interface `Comparable<T>`
- **C#:** La interfaz `IComparable` cumple una función similar a `Comparable` en Java. Las clases que implementan `IComparable` deben proporcionar una implementación del método `CompareTo()`.
  `IComparable<T>` Interface

- **Python:** La comparación de objetos se basa en métodos especiales como `__lt__` (menor que), `__le__` (menor o igual que), `__eq__` (igual a), `__ne__` (no igual a), `__gt__` (mayor que) y `__ge__` (mayor o igual que). Al implementar estos métodos, se define cómo se comparan los objetos de una clase. Por ejemplo, implementar `__le__` permite utilizar el operador `<=` para comparar instancias de la clase.
  Rich comparison methods: `__lt__`, `__le__`, `__gt__`, `__ge__`.

1.3 Ordenación por selección

Algorithm 1 Ordenación por selección

Require: $T[0 \dots n-1]$ es un vector de objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
function SelectionSort(T[0...n−1])
    for i ← 0 to n−2 do
        minj ← i
        minx ← T[i]
        for j ← i+1 to n−1 do
            if T[j] < minx then
                minj ← j
                minx ← T[j]
            end if
        end for
        T[minj] ← T[i]
        T[i] ← minx
    end for
end function
```

Análisis de la eficiencia de la ordenación por selección:
La ordenación por selección tiene una complejidad temporal de $O(n^2)$ en todos los casos (peor, promedio y mejor). Esto se debe a que siempre realiza dos bucles anidados para encontrar el elemento mínimo y colocarlo en su posición correcta.
Características de la ordenación por selección:

- **Cuadrático en comparaciones:** Realiza un número de comparaciones proporcional a n^2.
- **Lineal en intercambios (accesos al arreglo):** Realiza un número de intercambios proporcional a n. Esto la hace útil cuando los intercambios son costosos.
- **No adaptativo:** Su rendimiento no se ve afectado por el orden inicial de los datos.
- **In-place:** No requiere memoria adicional.

1.4 Ordenación por inserción

Análisis de la eficiencia de la ordenación por inserción:

- **Peor caso:** $O(n^2)$. Ocurre cuando el arreglo está ordenado en orden inverso.

Algorithm 2 Ordenación por inserción

Require: $T[0 \ldots n-1]$ es un vector objetos comparables
Ensure: $T[0 \ldots n-1]$ el vector de entrada ordenado ascendentemente

```
function InsertionSort(T[0 ... n − 1])
    for i ← 1 to n − 1 do
        x ← T[i]
        j ← i − 1
        while j > 0 and x < T[j] do
            T[j + 1] ← T[j]          ▷ Los elementos mayores a x se desplazan a la derecha
            j ← j − 1
        end while
        T[j + 1] ← x
    end for
end function
```

- **Mejor caso:** $O(n)$. Ocurre cuando el arreglo ya está ordenado. En este caso, solo realiza una comparación en cada iteración del bucle principal.
- **Caso promedio:** $O(n^2)$.

Características de la ordenación por inserción:

- **Adaptativo:** Su rendimiento mejora si el arreglo está parcialmente ordenado.
- **Estable:** Preserva el orden relativo de los elementos con claves iguales.
- **In-place:** No requiere memoria adicional.
- **Simple de implementar:** Es un algoritmo relativamente fácil de entender e implementar.

1.5 Ordenación shellsort

Análisis de la eficiencia de la ordenación Shellsort:

El análisis de la complejidad temporal de Shellsort es complejo y depende de la secuencia de incrementos utilizada. No se conoce una fórmula exacta para su complejidad en todos los casos.

- **Complejidad empírica:** En la práctica, Shellsort muestra un rendimiento significativamente mejor que la ordenación por selección e inserción, especialmente para arreglos de tamaño mediano. Con la secuencia de incrementos original de Shell $(n/2, n/4, ..., 1)$, la complejidad es $O(n^2)$. Con otras secuencias de incrementos, se puede lograr una complejidad de $O(n^{3/2})$ o incluso mejor.

Características de la ordenación Shellsort:

- **No estable:** No preserva el orden relativo de los elementos con claves iguales.
- **In-place:** No requiere memoria adicional.
- **Adaptativo:** Su rendimiento puede variar dependiendo del orden inicial de los datos y de la secuencia de incrementos utilizada.

Algorithm 3 Ordenación shellsort

Require: $T[0 \ldots n-1]$ es un vector objetos comparables
Ensure: $T[0 \ldots n-1]$ el vector de entrada ordenado ascendentemente

```
function ShellSort(T[0...n−1])
    h ← 1
    while h < N/3 do
        h ← 3h + 1
    end while
    while h ≥ 1 do
        for i ← h to n − 1 do
            x ← T[i]
            j ← i − h
            while j ≥ 0 and x < T[j] do
                T[j + h] ← T[j]        ▷ Los elementos mayores a x se desplazan a la derecha
                j ← j − h
            end while
            T[j + h] ← x
        end for
        h ← h/3
    end while
end function
```

1.6 Ordenación por fusión (mergesort)

Análisis de la eficiencia de la ordenación por fusión:

La ordenación por fusión tiene una complejidad temporal de $O(n \log n)$ en todos los casos (peor, promedio y mejor). Esto se debe a que divide el arreglo en mitades recursivamente hasta que cada subarreglo contiene un solo elemento, y luego fusiona los subarreglos ordenados en un solo arreglo ordenado.

Características de la ordenación por fusión:

- **Complejidad temporal:** $O(n \log n)$.
- **Complejidad espacial:** $O(n)$. Requiere memoria adicional para el arreglo auxiliar utilizado en la fusión.
- **Estable:** Preserva el orden relativo de los elementos con claves iguales.
- **No adaptativo:** Su rendimiento no se ve afectado por el orden inicial de los datos.

1.7 Ordenación rápida (quicksort)

Análisis de la eficiencia de la ordenación rápida:

La eficiencia de Quicksort depende en gran medida de la elección del pivote.

- **Peor caso:** $O(n^2)$. Ocurre cuando el pivote es siempre el elemento más pequeño o el más grande del arreglo. En este caso, la partición divide el arreglo en un subarreglo de tamaño 0 y otro de tamaño $n-1$, lo que lleva a una recursión de profundidad n.

Algorithm 4 Ordenación for fusión

Require: $T[0 \dots n-1]$ es un vector objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
function MERGE(a[0...n-1], aux[0...n-1], lo, mid, hi)
    for k ← lo to hi do
        aux[k] ← a[k]
    end for
    i ← lo
    j ← mid + 1
    for k ← low to hi do
        if i > mid then
            a[k] ← aux[j]
            j ← j + 1
        else if j > hi then
            a[k] ← aux[i]
            i ← i + 1
        else if aux[j] < aux[i] then
            a[k] ← aux[j]
            j ← j + 1
        else
            a[k] ← aux[i]
            i ← i + 1
        end if
    end for
end function

function MERGESORT(T[0...n-1])
    if n ≤ n_0 then
        ahdoc(T)
    else
        U ← T[0...⌊n/2⌋]
        V ← T[⌊n/2⌋ + 1...n-1]
        MERGESORT(U)
        MERGESORT(V)
        T ← MERGE(U, V)
    end if
end function
```

Algorithm 5 Ordenación rápida

Require: $T[0 \dots n-1]$ es un vector objetos comparables
Ensure: $T[0 \dots n-1]$ el vector de entrada ordenado ascendentemente

```
function PARTITION(a[0...n-1], lo, hi)
    i ← lo
    j ← hi + 1
    v ← a[lo]
    while True do
        while a[i] < v do
            i ← i + 1
            if i = hi then
                break
            end if
        end while
        while v < a[j] do
            j ← j - 1
            if j = lo then
                break
            end if
        end while
        if i ≥ j then
            break
        end if
        exchange a[i] ↔ a[j]
    end while
    exchange a[lo] ↔ a[j]
    return j
end function

function QUICKSORT(a[0...n-1], lo, hi)
    if hi ≤ lo then
        return
    end if
    j ← PARTITION(a,lo,hi)
    QUICKSORT(a,lo,j - 1)
    QUICKSORT(a,j + 1,hi)
end function
```

- **Mejor caso:** $O(n \log n)$. Ocurre cuando el pivote divide el arreglo en dos subarreglos de tamaño aproximadamente igual. En este caso, la recursión tiene una profundidad de $\log n$.
- **Caso promedio:** $O(n \log n)$. Con una buena elección de pivote (e.g., elegir un elemento aleatorio), Quicksort tiene un rendimiento promedio muy bueno.

Características de la ordenación rápida:

- **In-place:** No requiere memoria adicional (aparte de la pila de recursión).
- **No estable:** No preserva el orden relativo de los elementos con claves iguales.
- **Adaptativo:** Su rendimiento puede variar dependiendo del orden inicial de los datos y de la elección del pivote.
- **Muy eficiente en la práctica:** A pesar de su complejidad en el peor caso, Quicksort es uno de los algoritmos de ordenación más rápidos en la práctica, especialmente para arreglos grandes.