Diseño De Software - Métodos De Ordenamiento

Michael Daniel Murillo López

Ingeniería De Sistemas

Corporación Universitaria Minuto De Dios

Como parte de un ejercicio típico de Desarrollo de algoritmos de software, hice un pequeño análisis comparativo de los algoritmos de ordenamiento más populares, buscando estudiar la complejidad de cada uno de estos y como las diferentes formas de resolver un mismo problema pueden afectar los tiempos de ejecución. Quiero aclarar que este es solo un análisis académico muy simple que quiero documentar, el cual tal vez sirva a futuro para otros estudiantes de ciencias de la computación.

El usuario debe poder ingresar una serie de números separados por comas, y estos se entenderán como el conjunto de números del arreglo, y usando un menú en la consola debe poder seleccionar algoritmo utilizar.

1. Bubble Sort:

Este es uno de los algoritmos más simples de ordenamiento. Este algoritmo se basa en la comparación de elementos, particularmente entre parejas adyacentes, y si la pareja a comparar no está ordenada, simplemente se intercambian; el algoritmo concluye cuando al hacer todo el recorrido, y hacer todas las comparaciones entre vecinos adyacentes, no se requieren realizar más intercambios. Este algoritmo no es recomendable para arreglos con gran cantidad de datos, ya que tanto su caso promedio como su peor caso tienen un orden de crecimiento de $\Theta(n2)$.

```
Data: A: Unsorted array of numbers
Result: A*: Sorted array of numbers for
i ← 0 to length(A) − 1 do
    swapped ← false
    for j ← 0 to length(A) − 1 do
        /* compare to adjacent elements */
        if array[j] > array [j + 1] then
            /* swap them */
            auxSwap ← array[j]
            array[j] ← array [j + 1]
            array [j + 1] ← swap
            swapped ← true
        end
    end
    /* if no number was swapped, the array is sorted now */
    if not swapped then
        break
    end
end
```

Algoritmo 1: Bubble Sort

2. Merge Sort:

Este es un algoritmo de ordenamiento que está basado en el paradigma de divide y vencerás. Este es considerado uno de los mejores algoritmos para ordenar elementos de un arreglo, puesto que en el peor de los casos el orden de crecimiento que tiene es de $\Theta(nlogn)$. La estrategia que maneja Merge Sort es simple: el arreglo se divide en dos partes por la mitad, y este proceso se repite hasta que se llegue a arreglos de tamaño

1: luego, cada una de las soluciones se combina de manera ordenada, obteniendo de manera emergente al final el arreglo total completamente ordenado.

```
Data: A: Unsorted array of numbers
Result: A*: Sorted array of numbers if
lenght(A) == 1 then
    /* array is already sorted */
    return A
else
    /* split in two parts */
    left_sub-array ← A [0] . . . A [n / 2]
    right_sub-array ← A [(n / 2) + 1] . . . A[n]
    /* sort each one of the parts */ sortedL
    ← MergeSort (left sub-array) sortedR ←
    MergeSort (right sub-array)
    /* follow the stratefy divide and conquer */
    return Merge (sortedL, sortedR)
end
```

Algoritmo 3: Merge Sort

```
Data: A: Sorted array of numbers, B: Sorted array of numbers
Result: C: Sorted array of numbers that contains all elements of both A and B
l ← length(A) + length(B)
/* create C array */
C → Array of length l
indexA ← 0, indexB ← 0, indexC ← 0
while A and B have elements do
    if A[indexA] < B[indexB] then
        /* add element from A array */
        C[indexC] ← A[indexA]
        indexA ← indexA + 1
        indexC ← indexC + 1
    else
        /* add element from B array */
        C[indexC] ← B[indexB]
        indexB ← indexB + 1
        indexC ← indexC + 1
    end
end
/* one of A or B has still some elements */
while A has elements do
    C[indexC] ← A[indexA]
    indexA ← indexA + 1
    indexC ← indexC + 1
end
while B has elements do
    C[indexC] ← B[indexB]
    indexB ← indexB + 1
    indexC ← indexC + 1
end
return C
```

Algoritmo 4: Merge

3. Quick Sort:

Este es uno de los algoritmos de ordenamiento más eficientes que existe (suele utilizarse con grandes conjuntos de datos, y su eficiencia tanto en casos promedio como en el peor caso es de $\Theta(n\log n)$), el cual consiste en una estrategia de divide y vencerás debido a que siempre parte el arreglo en pequeños sub-arreglos, y este proceso se repite de manera recursiva. Particularmente, en este algoritmo se usa la noción de pivote para definir la construcción de los sub-arreglos, en donde los valores más pequeños que el pivote van al primer sub-arreglo, y los mayores van al segundo sub-arreglo. Tradicionalmente, se selecciona el primer elemento del arreglo como el pivote, y de igual manera se selecciona en los sub-arreglos mientras se está haciendo la recursividad; valga

aclarar que el pivote, luego de hacer el proceso de partición ya se encuentra en el lugar que tendrá finalmente en el conjunto ordenado.

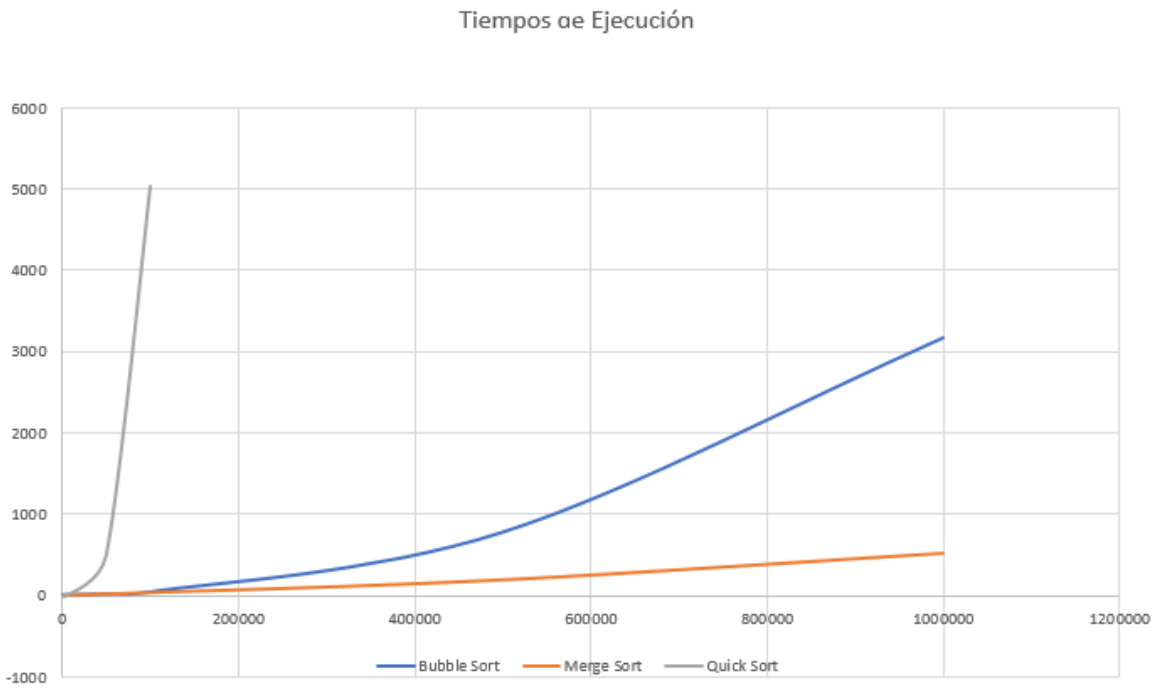
```
Data: A: Unsorted array of numbers
Result: A*: Sorted array of numbers if
length(A) == 1 then
    /* array A is already sorted */
    return A
else
    /* take first set element as a pivot */
    pivot ← A [0]
    for i ← 1 to length(A) do
        /* build both less and greater than pivot subarrays */
        if A[i] < pivot then
            less_subarray.add ← A[i]
        else
            greater subarray.add ← A[i]
        end
        /* call recursion for each one of the subarrays, and concatenate the results */
        return QuickSort (less_subarray) + pivot + QuickSort (greater subarray)
    end
end
```

Algoritmo 5: Quick Sort

4. Conclusiones

Los algoritmos de ordenación rápida y fusión se basan en el algoritmo de división y conquista que funciona de manera bastante similar. La diferencia previa entre la ordenación rápida y la fusión es que en la ordenación rápida el elemento pivote se usa para la ordenación. Por otro lado, la ordenación por fusión no utiliza el elemento pivote para realizar la ordenación. La ordenación rápida es casos más rápidos, pero es ineficiente en algunas situaciones y también realiza muchas comparaciones en comparación con la ordenación por fusión. Aunque la ordenación por fusión requiere menos comparación, necesita un espacio de memoria adicional de 0 (n) para almacenar la matriz adicional, mientras que la ordenación rápida necesita espacio de O (log n).

	Bubble Sort	Merge Sort	Quick Sort
1000	0,55	0,388	0,501
5000	0,143	1,827	4,71
50000	9,558	18,494	498,768
100000	34,606	34,368	5035,547
500000	771,864	191,841	0,001
1000000	3171,462	520,621	0,001



(Bubble Sort): O(nˆ 2)

(Merge Sort): O(n log n)

(Quick sort): O(n log n)