Diseño de Software
Taller No. 1
Metodos de Ordenamiento

Michael Daniel Murillo López
Ingenierι a de Sis-
temas
Corporación Universitaria Minuto de Dios
30 de A

.

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 cual algoritmo utilizar.

1 Bubble Sort:

Este es uno de los algoritmos mas simples de ordenamiento. Este algoritmo se basa en la comparacion de elementos, particularmente entre parejas adyacentes, y si la pareja a comparar no esta 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 mas 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 Θ (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] ar-
            ray[j] ← array[j + 1] ar-
            ray[j + 1] ← swap swapped
            ← true
        end
    end
    /∗ if no number was swapped, the array is sorted now ∗ /
    if not swapped then
        break
    end
end
```

Algorithm 1: BubbleSort

2 Merge Sort:

Este es un algoritmo de ordenamiento que esta basado en el paradigma de divide y venceras. 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 Θ (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 taman˜o 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
```

Algorithm 2: MergeSort

```
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 in-
        dexC ← indexC + 1
    else
        /∗ add element from B array ∗ /
        C[indexC] ← B[indexB]
        indexB ← indexB + 1 in-
        dexC ← 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 in-
    dexC ← indexC + 1
```

```
end
    while B has elements do
    C[indexC] ← B[indexB]
    indexB ← indexB + 1 in-
    dexC ← indexC + 1
end
return C
```

Algorithm 2: Merge

3 Quick Sort:

Este es uno de los algoritmos de ordenamiento mas eficientes que existe (suele utilizarse con grandes conjuntos de datos, y su eficiencia tanto en casos promedio como en el peor caso es de Θ (nlogn)), el cual consiste en una estrategia de divide y venceras debido a que siempre parte el arreglo en pequen˜os sub-arreglos, y este proceso se repite de manera recursiva. Particularmente, en este algoritmo se usa la nocion de pivote para definir la construccion de los sub-arreglos, en donde los valores mas pequen˜os 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 seleciona en los sub-arreglos mientras se esta haciendo la recursividad; valga aclarar que el pivote, luego de hacer el proceso de particion, ya se encuentra en el lugar

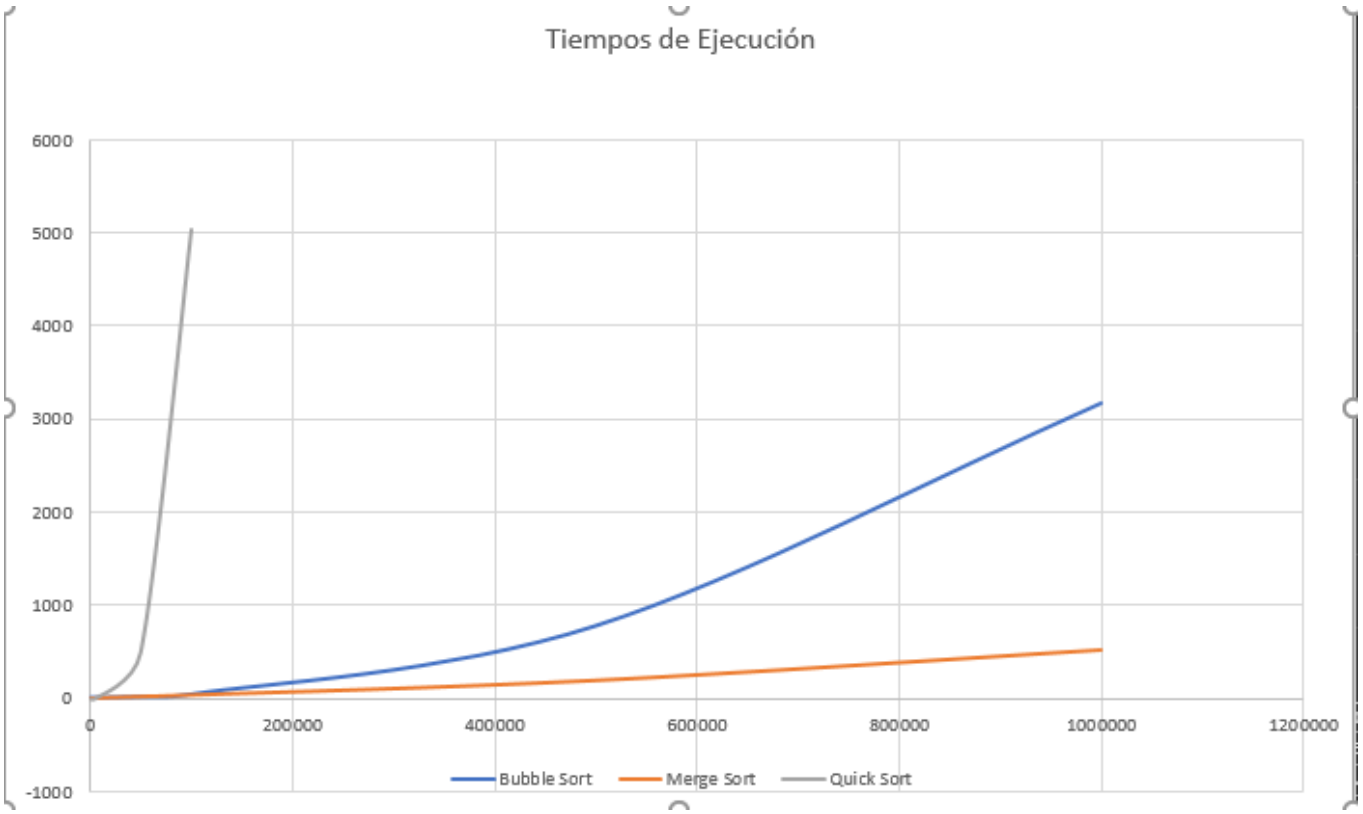
que tendr

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
```

Algorithm 3: QuickSort

(Bubble Sort): O(n^ 2)
(Merge Sort): O(n log n)



(Quicksort): O(n log n)