

Universidad Autónoma del Estado de México
Centro Universitario UAEM Texcoco
Ingeniería en Computación



Comparación empírica del número de operaciones
de diferentes algoritmos de ordenamiento

TESIS

Que para obtener el Título de
Ingeniero en Computación

Presentan

David García Estrada

Daniel González Mendoza

Director de tesis

Dr. Farid García Lamont

Texcoco, Estado de México, Noviembre de 2023

Contenido

Lista de algoritmos.....	4
Lista de tablas	5
Lista de figuras.....	6
Introducción.....	8
Justificación	13
Planteamiento del problema	14
Hipótesis	15
Objetivos	16
Objetivo general	16
Objetivos específicos	16
Marco teórico	17
Métodos de ordenamiento existentes	17
Notación asintótica	29
Series matemáticas	30
Propiedades de los logaritmos.....	33
Solución de recurrencias	34
Análisis de algoritmos y experimentos	35
Complejidad de algoritmos.....	35
Experimentos realizados	54
Resultados y discusión	57
Experimentos utilizando lenguaje C/C++.....	57
Experimentos utilizando lenguaje Java.....	61
Comparación asintótica de resultados obtenidos utilizando C/C++	64
Comparación asintótica de resultados obtenidos utilizando Java	72
Conclusiones.....	80
Referencias	81
Apéndice.....	83
Códigos fuentes en C/C++.....	83
Códigos fuentes en Java	95

Lista de algoritmos

Algoritmo 1 Método de Heapsort	26
Algoritmo 2 Seudocódigo del método de ordenamiento de la burbuja	35
Algoritmo 3 Seudocódigo del método de ordenamiento de Stooge.....	38
Algoritmo 4 Seudocódigo del método de ordenamiento de Shell.....	39
Algoritmo 5 Seudocódigo del método de ordenamiento de Comb	44
Algoritmo 6 Algoritmo Mergesort.....	46
Algoritmo 7 Seudocódigo del método de ordenamiento de Merge	47
Algoritmo 8 Algoritmo Quicksort.....	48
Algoritmo 9 Seudocódigo del método de partición	49

Lista de tablas

Tabla 1 Complejidades de los algoritmos de ordenamiento para los mejores, peores y casos promedios	17
Tabla 2 Resultado de las pasadas restantes	21
Tabla 3 Valores graficados método de la burbuja.....	64
Tabla 4 Valores graficados método Stoooge.....	65
Tabla 5 Valores graficados método Shell.....	66
Tabla 6 Valores graficados método Comb	67
Tabla 7 Valores graficados método Merge.....	69
Tabla 8 Valores graficados método Quicksort.....	70
Tabla 9 Valores graficados método de la burbuja.....	72
Tabla 10 Valores graficados método Stoooge	73
Tabla 11 Valores graficados método Shell.....	74
Tabla 12 Valores graficados método Comb	75
Tabla 13 Valores graficados método Merge.....	77
Tabla 14 Valores graficados método Quicksort.....	78
Tabla 15 Tabla de los promedios de intercambios en 30 corridas con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Merge, Shell, Burbuja, Stoooge, Quicksort y Comb.....	93
Tabla 16 de los promedios de comparaciones en 30 corridas con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Merge, Shell, Burbuja, Stoooge, Quicksort y Comb.....	94
Tabla 17 de los promedios de intercambios en 30 ejecuciones con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Burbuja, Shell, Merge, Quicksort, Stoooge y Comb.	107
Tabla 18 de los promedios de comparaciones en 30 ejecuciones con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Burbuja, Shell, Merge, Quicksort, Stoooge y Comb.	108

Lista de figuras

Fig. 1 Montículo. Imagen tomada de la referencia	24
Fig. 2 arreglo unidimensional.....	24
Fig. 3 Ejemplo de iteraciones del método de la burbuja durante el ordenamiento	37
Fig. 4 Primera pasada de ordenamiento del método de Shell.....	40
Fig. 5 Segunda pasada de ordenamiento del método de Shell.....	41
Fig. 6 Tercera pasada de ordenamiento del método de Shell	41
Fig. 7 Cuarta pasada de ordenamiento del método de Shell.....	41
Fig. 8 Ejemplo de intercambio del método Comb	43
Fig. 9 Ejemplo de iteraciones del método de mergesort durante el ordenamiento	45
Fig. 10 Ejemplo de ordenamiento	51
Fig. 11 Ubicación del resto de los elementos en el arreglo.....	51
Fig. 12 Comparación de cantidades de intercambios con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooage, Quicksort y Comb.....	57
Fig. 13 Comparación de métodos con las cantidades de intercambios más bajas.....	58
Fig. 14 Comparación de métodos con las cantidades de intercambios más altas.....	58
Fig. 15 Comparación de cantidades de comparaciones con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooage, Quicksort y Comb.....	59
Fig. 16 Comparación de métodos con las cantidades de comparaciones más bajas.....	59
Fig. 17 Comparación de métodos con las cantidades de comparaciones más altas.....	60
Fig. 18 Método Stooage con las cantidades de comparación más altas.....	60
Fig. 19 Comparación de cantidades de intercambios con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooage, Quicksort y Comb.....	61
Fig. 20 Comparación de métodos con las cantidades de intercambios más bajas.....	61
Fig. 21 Comparación de métodos con las cantidades de intercambios más altas.....	62
Fig. 22 Comparación de cantidades de comparaciones con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooage, Quicksort y Comb implementado en java.....	62
Fig. 23 Numero de comparaciones realizadas con los métodos Shell, Merge y Quicksort implementados en Java.....	63

Fig. 24	Número de comparaciones con los métodos menos eficientes implementados en Java.	63
Fig. 25	Comparación de pasos obtenidos para el método de la burbuja y una función n^2	65
Fig. 26	Comparación de pasos obtenidos para el método de Stooge y una función $O(n\log 1.53)$...	66
Fig. 27	Comparación de pasos obtenidos para el método de Shell y una función $O(n\log n)$	67
Fig. 28	Comparación de pasos obtenidos para el método de Comb y una función $O(n^2)$	68
Fig. 29	Comparación de pasos obtenidos para el método de Merge y una función $O(n\log n)$	70
Fig. 30	Comparación de pasos obtenidos para el método de Quicksort y una función $O(n\log n)$...	71
Fig. 31	Comparación de pasos obtenidos para el método de la Burbuja y una función $O(n^2)$	73
Fig. 32	Comparación de pasos obtenidos para el método de Stooge y una función $O(n\log 1.53)$...	74
Fig. 33	Comparación de pasos obtenidos para el método de Shell y una función $O(n\log n)$	75
Fig. 34	Comparación de pasos obtenidos para el método de Comb y una función $O(n^2)$	76
Fig. 35	Comparación de pasos obtenidos para el método de Merge y una función $O(n\log n)$	78
Fig. 36	Comparación de pasos obtenidos para el método de Quicksort y una función $O(n\log n)$...	79

Introducción

El desarrollo de algoritmos de ordenamiento de números es muy importante ya que son necesarios para diferentes tareas en el área de ciencias de la computación. Uno de los criterios que se consideran para seleccionar un algoritmo de ordenamiento es su complejidad computacional. Aunque existen varios análisis de complejidad de diferentes algoritmos de ordenamiento, estos no han sido comprobados de forma empíricamente práctica, ya que estos análisis son teóricos. De aquí que se propone implementar y correr diferentes algoritmos de ordenamiento en donde se calculan la cantidad de intercambios de valores que le toma a cada algoritmo ordenar una lista de números. Esto con el fin de comparar los valores obtenidos de forma práctica y compararlo con los valores que se obtienen de forma teórica con los análisis de complejidad; de esta forma se pueda confirmar si tales análisis teóricos de complejidad son correctos.

Los ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

La eficiencia es el factor que mide la calidad y rendimiento de un algoritmo. En los métodos de ordenación, existen dos criterios para decidir que algoritmo es el más eficiente:

- tiempo menor de ejecución en computadora
- menor número de instrucciones.

Es difícil efectuar estas medidas ya que dependen del lenguaje de programación y del estilo de cada programador, por ello el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza.

Al ordenar los elementos de un vector número de comparaciones será función del número de elementos (n) del vector (array). Así que, se puede expresar el número de comparaciones en términos de n (por ejemplo, $n+4$), o bien, n^2 en lugar de números enteros (por ejemplo, 220).

Existen actualmente varios algoritmos de ordenamiento, al revisar el estado de la técnica se encontraron los siguientes:

1. Bead Sort.
2. Binary Insertion Sort.
3. Bitonic Sort.
4. Bogo Sort.
5. Bubble Sort.
6. Bucket Sort.
7. Circle Sort.
8. Cocktail Shaker Sort.
9. Comb Sort.
10. Counting Sort.
11. Cycle Sort.
12. Double Sort.
13. Dutch National Flag Sort.
14. Exchange Sort.
15. External Sort.
16. Gnome Sort.
17. Heap Sort.
18. Insertion Sort.
19. Intro Sort.
20. Iterative Merge Sort.
21. Merge Insertion Sort.
22. Msd Radix Sort.
23. Natural Sort.
24. Odd Even Sort.
25. Odd Even Transposition Parallel.
26. Odd Even Transposition Single Threaded.
27. Pancake Sort.
28. Patience Sort.
29. Pigeon Sort.

30. Pigeonhole Sort.
31. Quick Sort.
32. Quick Sort 3 Partition.
33. Radix Sort.
34. Recursive Bubble Sort.
35. Recursive Insertion Sort.
36. Recursive Mergesort Array.
37. Recursive Quick Sort.
38. Selection Sort.
39. Shell Sort.
40. Shrink Shell Sort.
41. Slowsort.
42. Stooge Sort.
43. Strand Sort.
44. Tim Sort.
45. Topological Sort.
46. Tree Sort.
47. Unknown Sort.
48. Wiggle Sort.
49. Alpha Numerical Sort.
50. Find Second Largest Element.
51. Fisher Yates Shuffle.
52. Flash Sort.
53. Heap Sort V 2.
54. Quick Sort Recursive.
55. Simplified Wiggle Sort.
56. Swap Sort.
57. Cocktail Selection Sort.
58. Count Inversions.
59. Counting Sort String.
60. Dnf Sort.

61. Library Sort.
62. Non Recursive Merge Sort.
63. Numeric String Sort.
64. Quick Sort 3.
65. Radix Sort 2.
66. Random Pivot Quick Sort.
67. Selection Sort Iterative.
68. Selection Sort Recursive.
69. Shell Sort 2.
70. Wave Sort.
71. Bubble Sort Recursion.
72. Dual Pivot Quick Sort.
73. Introspective Sort.
74. Link List Sort.
75. Merge Sort No Extra Space.
76. Merge Sort Recursive.
77. Simple Sort.
78. Sort Algorithm.
79. Sort Utils.
80. Sort Utils Random Generator.
81. Bubble Sort 2.
82. Cocktail Sort.
83. Heap Sort 2.
84. Insertion Sort Recursive.
85. Merge Sort Nr.
86. Multikey Quick Sort.
87. Partition Sort.
88. Random Quick Sort.
89. Shaker Sort.
90. Quick Sort 3 Ways.
91. Sleep Sort.

92. Jort Sort.
93. Permutation Sort.
94. Count Sort.
95. Insert Sort.
96. Select Sort
97. Sort Color.
98. Array Keys Sort.
99. Brick Sort.
100. Is Sorted.
101. Quickselect Median of Medians.
102. Stabilize.
103. Three Way Partition.

Sin embargo, para este estudio se han seleccionado los algoritmos que mejor desempeño tienen actualmente como lo son Shell, Mergesort, Quicksort y Comb. Pero por otra parte, se consideró estudiar dos algoritmos que son ejemplos de algoritmos con alta complejidad, los cuales son el método de la Burbuja y el método de Stooge.

En este caso, se ordenaron vectores con valores asignados aleatoriamente. Haciendo foco en los métodos más populares, analizando el tiempo que demora y revisando el código, escrito en C ++ y Java, de cada algoritmo.

Justificación

Actualmente existen varios trabajos donde se analiza y reportan la complejidad algorítmica de diferentes algoritmos de ordenamiento. Sin embargo, no se conocen trabajos donde mencionen haber hecho experimentos que sirvan para validar los análisis de complejidad algorítmica. Por un lado, es importante realizar esta experimentación para dar certeza y validar los análisis de complejidad de diferentes algoritmos de ordenamiento; por otro lado, esto puede permitir seleccionar mejor los algoritmos de ordenamiento para tareas específicas en donde el tiempo de ejecución sea un factor importante, y además verificar que tanto influye los lenguajes de programación y el hardware para correr algoritmos de ordenamiento.

Planteamiento del problema

Los principales problemas de los algoritmos de ordenamiento consisten en: 1) el tamaño de la lista de números a ordenar, 2) que tan desordenada se encuentra la lista de números. La cantidad de operaciones que requiere cualquier algoritmo de ordenamiento dependerá del tamaño de la lista de números y por otra, en qué orden se encuentre la lista. Es decir, no es lo mismo ordenar una lista con 100 número a una lista con 10000 números; pero también, no es lo mismo ordenar una lista de 10000 números que tiene una tercera parte de números desordenados a una lista de ese mismo tamaño que se encuentre ordenada de forma inversa, esto es, que la lista se encuentre ordenada de mayor a menor.

Otra cuestión importante que se debe verificar, es si los lenguajes de programación y el hardware de la computadora donde se ejecutan los códigos de los algoritmos, influyen en la cantidad de operaciones que deban hacer los algoritmos para ordenar las listas de números. De esta forma, se plantea hacer experimentos con diferentes algoritmos de ordenamientos, codificados en diferentes lenguajes de programación y en diferentes procesadores, en donde se cuenten la cantidad de intercambios de valores que requieren los algoritmos para ordenar listas de números de diferentes tamaños, y de aquí comparar los valores obtenidos con los valores que se obtienen de las funciones de complejidad algorítmica de los algoritmos de ordenamiento propuestos.

Hipótesis

Verificar si los lenguajes de programación y el hardware de la computadora donde se ejecutan los códigos de los algoritmos, influyen en la cantidad de operaciones (intercambios y comparaciones) que realizan los algoritmos para ordenar listas de números. Para lo cual, se implementan los algoritmos, que abajo se enlistan, en donde se cuentan la cantidad de intercambios y comparaciones que realizan; así como comparar los valores obtenidos con los órdenes de crecimiento o de complejidad de los algoritmos que se enlistan a continuación:

- Burbuja: $O(n^2)$
- Shell: $O(n \log n)$
- Merge: $O(n \log n)$
- Quicksort: $O(n \log n)$
- Stooge: $O(n^{\log_{1.5} 3})$
- Comb: $O(n \log n)$

Objetivos

Objetivo general

Comprobar la complejidad algorítmica de diferentes algoritmos de ordenamiento al contar la cantidad de intercambios y comparaciones que requieren los algoritmos para ordenar una lista de números.

Objetivos específicos

- Implementar en dos lenguajes diferentes, los siguientes algoritmos de ordenamiento:
 - Método de la burbuja
 - Método Shell
 - Método Merge
 - Método Quicksort
 - Método Stooge
 - Método Comb
- Contar la cantidad de intercambios de números que se hacen para ordenar la lista en cada método.
- Obtener la complejidad algorítmica de cada método.
- Ejecutar los algoritmos en dos computadoras y lenguajes con características diferentes.

Marco teórico

Métodos de ordenamiento existentes

Como ya se mencionó anteriormente, existen actualmente varios métodos de ordenamiento, pero en este estudio nos enfocamos a los métodos con las mejores complejidades se ha documentado. Por otra parte, también se seleccionaron los métodos de la burbuja y de Stooage, que son dos métodos que han mostrado tener alta complejidad. En la Tabla 1 se muestran las complejidades de los algoritmos de ordenamiento, de acuerdo con lo que se encontró en trabajos anteriores.

Método	Mejor	Promedio	Peor
Burbuja	$O(n)$	$O(n^2)$	$O(n^2)$
Stooage	$O(n^{\log_{1.5} 3})$	$O(n^{\log_{1.5} 3})$	$O(n^{\log_{1.5} 3})$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Shell	$O(n \log n)$	$O(n^{4/3})$	$O(n^{3/2})$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Comb	$O(n \log n)$	$O(n^2)$	$O(n^2)$

Tabla 1 Complejidades de los algoritmos de ordenamiento para los mejores, peores y casos promedios.

En este trabajo nos enfocaremos en hacer corridas con diferentes tamaños de arreglos y contar la cantidad de intercambios y comparaciones que hace cada algoritmo y así corroborar que las complejidades de estos algoritmos corresponden a las funciones que se reportan.

Ordenación por el método de la sacudida (shaker sort)

“El método de shaker sort, más conocido como el método de la sacudida, es una optimización del método de intercambio directo (Burbuja). La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método de la burbuja.

En este algoritmo cada pasada tiene dos etapas. En la primera etapa, de derecha a izquierda, se trasladan los elementos más pequeños hacia la parte izquierda de arreglo, almacenando en una variable la posición del último elemento intercambiado. En la segunda etapa, de izquierda

a derecha, se trasladan los elementos más grandes hacia la parte derecha del arreglo, almacenando en otra variable la posición del último elemento intercambiado. Las sucesivas pasadas trabajan con los componentes del arreglo comprendidos entre las posiciones almacenadas en las variables auxiliares. El algoritmo termina cuando en una etapa no se producen intercambios, o bien cuando el contenido de la variable que guarda el extremo izquierdo del arreglo es mayor que el contenido de la variable que almacena el extremo derecho.”

“Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A utilizando el método de la sacudida.

A: 15 67 08 16 44 27 12 35

PRIMERA PASADA

Primera etapa (de derecha a izquierda)

$A[7] > A[8]$ (12 > 35) no hay intercambio

$A[6] > A[7]$ (27 > 12) sí hay intercambio

$A[5] > A[6]$ (44 > 12) sí hay intercambio

$A[4] > A[5]$ (16 > 12) sí hay intercambio

$A[3] > A[4]$ (08 > 12) no hay intercambio

$A[2] > A[3]$ (67 > 08) sí hay intercambio

$A[1] > A[2]$ (15 > 08) sí hay intercambio

Ultima posición de intercambio de izquierda a derecha: 8.

Luego de la segunda etapa de la primera pasada, el arreglo queda así:

A: 08 15 12 16 44 27 35 67

SEGUNDA PASADA

Primera etapa (de derecha a izquierda)

$A[6] > A[7]$ $(27 > 35)$ no hay intercambio
 $A[5] > A[6]$ $(44 > 27)$ sí hay intercambio
 $A[4] > A[5]$ $(16 > 27)$ no hay intercambio
 $A[3] > A[4]$ $(12 > 16)$ no hay intercambio
 $A[2] > A[3]$ $(15 > 12)$ sí hay intercambio

Ultima posición de intercambio de derecha a izquierda: 3.

A: 08 12 15 16 27 44 35 67

Segunda etapa (de izquierda a derecha)

$A[3] > A[4]$ $(15 > 16)$ no hay intercambio
 $A[4] > A[5]$ $(16 > 27)$ no hay intercambio
 $A[5] > A[6]$ $(27 > 44)$ no hay intercambio
 $A[6] > A[7]$ $(44 > 35)$ sí hay intercambio

Ultima posición de intercambio de izquierda a derecha: 7.

A: 08 12 15 16 27 34 44 67

Al realizar la primera etapa de la tercera pasada se observa que no se realizaron intercambios; por lo tanto, la ejecución del algoritmo se termina.” (Cairó y Guardati, 2006).

Ordenación por inserción directa (Método de la Baraja)

“El método de ordenación por inserción directa es el que utiliza generalmente los jugadores de cartas cuando las ordenan, de ahí que también se conozca con el nombre de método de la baraja.

La idea central de este algoritmo consiste en insertar un elemento del arreglo en su parte izquierda, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el n-ésimo elemento.”

“Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A utilizando el método de inserción directa:

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[2] > A[1]$ (67 > 15) no hay intercambio

A: 15 67 08 16 16 44 27 12 35

SEGUNDA PASADA

$A[3] > A[2]$ (08 > 67) sí hay intercambio

$A[2] > A[1]$ (08 > 15) sí hay intercambio

A: 08 15 67 16 44 27 12 35

TERCERA PASADA

$A[4] > A[3]$ (16 > 67) sí hay intercambio

$A[3] > A[2]$ (16 > 15) no hay intercambio

A: 08 15 16 67 44 27 12 35

Observe que una vez que se determina la posición correcta del elemento, se interrumpen las comparaciones. Por ejemplo, para el caso anterior no se realizó la comparación $A[2] < A[1]$.

En la siguiente tabla se presenta el resultado de las pasadas restantes.” (Cairó y Guardati, 2006).

4a. pasada:	08	15	16	44	67	27	12	35
5a. pasada:	08	15	16	27	44	67	12	35
6a. pasada:	08	12	15	16	27	44	67	35
7a. pasada:	08	12	15	16	27	35	44	67

Tabla 2 Resultado de las pasadas restantes

Ordenación por el método de inserción binaria

“El método de ordenación por inserción binaria es una mejora del método de inserción directa presentando anteriormente. La mejora consiste en realizar una búsqueda binaria en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado. El proceso, al igual que en el método de inserción directa, se repite desde el segundo hasta n-ésimo elemento. Analicemos un ejemplo.” (Cairó y Guardati, 2006).

“Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A utilizando el método de inserción binaria.

A: 15 67 08 16 44 27 12 35

A continuación, se presentan las comparaciones que se llevan a cabo:

PRIMERA PASADA

$A[2] > A[1]$ (67 > 15) no hay intercambio

A: 15 67 08 16 44 27 12 35

SEGUNDA PASADA

$A[3] > A[1]$ (08 > 15) sí hay intercambio

A: 08 15 67 16 44 27 12 35

TERCERA PASADA

$A[4] > A[2]$ (16 > 15) no hay intercambio

$A[4] > A[3]$ (16 > 67) sí hay intercambio

A: 08 15 16 67 44 27 12 35

CUARTA PASADA

$A[5] > A[2]$ (44 > 15) no hay intercambio

$A[5] > A[3]$ (44 > 16) no hay intercambio

$A[5] > A[4]$ (44 > 67) sí hay intercambio

A: 08 15 16 44 67 27 12 35

QUINTA PASADA

$A[6] > A[3]$ (27 > 16) no hay intercambio

$A[6] > A[4]$ (27 > 44) sí hay intercambio

A: 08 15 16 27 44 67 12 35

SEXTA PASADA

$A[7] > A[3]$ (12 > 16) sí hay intercambio

$A[7] > A[1]$ (12 > 08) no hay intercambio

$A[7] > A[2]$ (12 > 15) sí hay intercambio

A: 08 12 15 16 27 44 67 35

SEPTIMA PASADA

$A[8] > A[4]$ (35 > 16) no hay intercambio

$A[8] > A[6]$ (35 > 44) sí hay intercambio

$A[8] > A[5]$ (35 > 27) no hay intercambio

A: 08 12 15 16 27 35 44 67”

(Cairó y Guardati, 2006).

Ordenación por el método heapsort (montículo)

“El método de ordenación heapsort se conoce también como montículo. Su nombre se debe a su autor, J. W. Williams, quien lo llamo así. Es el más eficiente de los métodos de ordenación que trabaja con árboles. La idea central de este algoritmo se basa en dos operaciones:

1. Construir un montículo.
2. Eliminar la raíz del montículo en forma repetida.

Es importante señalar que un montículo se define como: Para todo nodo de árbol se debe cumplir que su valor sea mayor o igual que el valor de cualquier de sus hijos.

Ilustracion1 Se muestra un montículo. Allí se puede observar que para cada nodo K del árbol, su valor es mayor o igual que el valor de cualquiera de sus hijos. Ahora bien, para representar un montículo en un arreglo lineal se debe tener en cuenta para todo nodo K lo siguiente:

1. El nodo K se almacena en la posición K correspondientes del arreglo.
2. El hijo izquierdo del nodo K se almacena en la posición $2*K$.
3. El hijo derecho del nodo K se almacena e la posición $2*K+1$.

Ilustracion2 Contiene la representación del montículo de la figura 8.4 en un arreglo unidimensional A .

Observe que si se desea obtener el hijo izquierdo del nodo $A[4]$, cuyo valor es 28, se hace $A[4 * 2] = A[8]$ y su contenido es 27. Si deseamos obtener en cambio el hijo derecho de $A[4]$, hacemos $A[4 * 2 + 1] = A[9]$ y su contenido es 16.

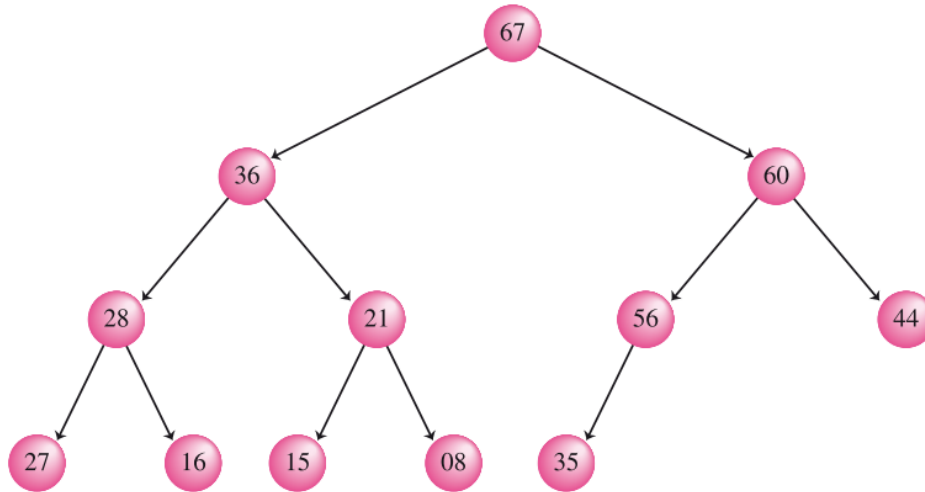


Fig. 1 Montículo. Imagen tomada de la referencia

A	67	36	60	28	21	56	44	27	16	15	08	35
	1	2	3	4	5	6	7	8	9	10	11	12

Fig. 2 arreglo unidimensional

A su vez, es posible calcular el padre de un nodo no raíz K cualquiera, tomando la parte entera de $(K \text{ entre } 2)$. Así, por ejemplo, si se desea obtener el padre del nodo $A[11]$, cuyo valor es 8, se hace $A[\text{parte entera}(11 \text{ entre } 2)] A[5]$ y su contenido es 21.”

Algoritmo de ordenación

```
void filtrado_desc (Dato * A, int i, int N){
    /* queremos que se respete el orden MAX del montículo */
    Dato tmp = A[i];
    int hijo = 2 * i;

    if ((hijo < N) && (A[hijo + 1] > A[hijo]))
        hijo++;
    while ((hijo <= N) && (tmp < A[hijo])){
        /* elijo bien el hijo */
        if ((hijo < N) && (A[hijo + 1] > A[hijo]))
            hijo++;
        A[i] = A[hijo];
        i = hijo;
        hijo = 2 * i;
    }
    A[i] = tmp;
}

Void intercambiar (Dato * A, int i, int j){
    Dato tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
}

Void heapsort (Dato * A, int N){
    int i;
    /* meto los datos en el montículo (ordeno) */
    for (i = N / 2; i >= 0; i--){
        filtrado_desc (A, i, N);
        /* saco los datos y los meto al final para obtener el array
        ordenado */
    }
    for (i = N - 1; i > 0; i--){
        intercambiar (A, 0, i);
    }
}
```

```
filtrado_desc (A, 0, i - 1);  
}  
}
```

Algoritmo 1 Método de Heapsort

Ordenación por mezcla directa

“El método de ordenación por mezcla directa es probablemente el más utilizado por su fácil comprensión. La idea central de este algoritmo consiste en la realización sucesiva de una partición y una fusión que produce secuencias ordenadas de longitud cada vez mayor. En la primera pasada, la partición es de longitud 1 y la fusión o mezcla produce secuencia ordenadas de longitud 2. En la segunda pasada, la partición es de longitud 2 y la fusión o mezcla produce secuencias ordenadas de longitud 4. Este proceso se repite hasta que la longitud de secuencia para la partición sea:

$$\text{Parte entera } ((n + 1)/2)$$

Donde *n* representa el número de elementos del archivo original.

Supongamos que se desea ordenar las claves del archivo F. Para realizar tal actividad se utilizan dos archivos auxiliares a los que se les denominan F1 y F2.

F: 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

PRIMERA PASADA

Partición en secuencias de longitud 1.

F1: 09' 75' 14 68' 17 29' 25 31' 04 05' 13 18' 42 72' 61'

F2: 75' 68' 17' 25' 05' 18' 46'

Fusión en secuencias de longitud 2.

F: 09 75' 14 68' 17 29' 25 31' 04 05' 13 18' 46 72' 61'

SEGUNDA PASADA

Partición en secuencias de longitud 2.

F1: 09 14 68 75' 17 25 29 31' 04 05 13 18' 46 61 72'

TERCERA PASADA

Partición en secuencias de longitud 4.

F1: 09 14 68 75' 04 05 13 18'

F2: 17 25 29 31' 46 61 72'

Fusión en secuencias de longitud 8.

F: 09 14 17 25 29 31 68 75' 04 05 13 18 46 61 72'

CUARTA PASADA

Partición en secuencias de longitud 8.

F1: 09 14 17 25 29 31 68 75'

F2: 04 05 13 18 46 61 72'

Fusión en secuencias de longitud 16.

F: 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75"

(Cairó y Guardati, 2006).

Ordenación por el método de mezcla equilibrada

“El método de ordenación por mezcla equilibrada, conocido también como mezcla natural, es una optimización del método de mezcla directa.

La idea central de este algoritmo consiste en realizar las particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias de tamaño fijo previamente determinadas. Luego se realiza la fusión de las secuencias ordenadas, en forma alternada, sobre dos archivos. Aplicando estas acciones en una forma repetida, se logra que el archivo original quede ordenado. Para la realización de este proceso de orientación se necesitará cuatro archivos. El archivo original *F* y tres archivos auxiliares a los que se denominara *F1*, *F2*, *F3*. De estos archivos, dos serán considerados de entrada y dos de salida; esto, de manera

alternada, con el objeto de realizar la fusión-partición. El proceso termina cuando en la realización de una fusión-partición el segundo archivo queda vacío.

Supongamos que se desea ordenar las claves del archivo F utilizando el método de mezcla equilibrada.

F: 09 75 14 68 29 17 31 25 04 05 13 18 72 46 61

Los pasos que se realizan son:

PARTICIPACIÓN PRINCIPAL

*F*₂: 09 75' 29' 25' 46 61'

*F*₃: 14 68' 17 31' 04 05 13 18 72'

PRIMERA FUSIÓN-PARTICIÓN

F: 09 14 68 75' 04 05 13 18 25 46 61 72'

*F*₁: 17 29 31'

SEGUNDA FUSIÓN-PARTICIÓN

*F*₂: 09 14 17 29 31 68 75'

*F*₃: 04 05 13 18 25 46 61 72'

TERCERA FUSIÓN-PARTICIÓN

F: 04 05 09 13 14 17 18 25 29 31 46 61 68 72 75

*F*₁:

Observe que al realizar la tercera fusión-partición el segundo archivo queda vacío; por lo tanto, se puede afirmar que el archivo ya se encuentra ordenado.”

Notación asintótica

La notación que se emplea para describir el tiempo de ejecución asintótica de un algoritmo está definida en termino de funciones cuyo dominio e imagen son los números naturales \mathbb{N} y los números reales \mathbb{R} , respectivamente. La notación es conveniente para describir el tiempo de ejecución como una función $T(n)$, el cual esta usualmente definida por tamaños de entrada de tipo entero. Se puede extender la notación en el dominio de números reales o restringirlo a un subconjunto de números naturales. La cota superior asintótica de una función $g(n)$ se escribe $O(f(n))$ como el conjunto de funciones:

$$O(f(n)) = \{g(n) | \exists c > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow g(n) \leq cf(n)\} \quad (1)$$

Es decir, $O(f(n))$ es el conjunto de funciones con dominio e imagen en los números naturales, tales que existe una constante c mayor a cero, donde a partir de un número natural n_0 el crecimiento de $g(n)$ es menor o igual al de la función $cf(n)$.

La cota inferior asintótica de una función $g(n)$ se escribe $\Omega(f(n))$ como el conjunto de funciones:

$$\Omega(f(n)) = \{g(n) | \exists c > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow g(n) \geq cf(n)\} \quad (2)$$

$\Omega(f(n))$ Es el conjunto de funciones con dominio e imagen en los números naturales, tales que existe una constante c mayor a cero, donde a partir de un número natural n_0 el crecimiento de $g(n)$ es mayor o igual al de la función $cf(n)$. Para representar que una función $g(n)$ tiene el mismo orden de crecimiento que un conjunto de funciones se escribe como:

$$\Theta(f(n)) = \{g(n) | \exists c_1, c_2 > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow c_1f(n) \leq g(n) \leq c_2f(n)\} \quad (3)$$

Esto significa que $g(n) = \Theta(f(n))$ si existen constantes positivas c_1 y c_2 tales que la “mantienen” entre $c_1f(n)$ y $c_2f(n)$ para un n suficientemente grande. Otra forma de establecer que $g(n) = \Theta(f(n))$ es que se cumpla la siguiente condición:

$$g(n) = O(f(n)) \text{ y } g(n) = \Omega(f(n)) \quad (4)$$

Series matemáticas

Una **serie matemática** es la generalización de la noción de suma, aplicada a los términos de una sucesión de números a_1, a_2, a_3, \dots , que suele escribirse con el símbolo de sumatoria.

$$\sum_{i=1}^{\infty} a_i = a_1 + a_2 + a_3 + \dots \quad (5)$$

Una **suma parcial o finita** de n números se escribe como:

$$\sum_{i=1}^n a_i = a_1 + a_2 + a_3 + \dots + a_n \quad (6)$$

La propiedad de linealidad aplica para las series:

$$\sum_{i=1}^n (\alpha a_i + b_i) = \alpha \sum_{i=1}^n a_i + \sum_{i=1}^n b_i \quad (7)$$

Una **serie aritmética** es la suma de los términos de una sucesión aritmética; esto es, una **sucesión aritmética** es una lista de números cuya diferencia entre dos términos sucesivos cualquiera es constante. Por ejemplo, en la sucesión 3,5,7,9,11, ... la diferencia común es 2.

Sumas importantes

$$\sum_{i=1}^n c = cn \quad (8)$$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (9)$$

$$\sum_{k=i}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (10)$$

$$\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4} \quad (11)$$

Para cualquier número real $a > 1$ la suma se dice ser una **serie geométrica** si tiene la forma:

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \dots + a^n \quad (12)$$

Un resultado importante de este tipo de serie geométrica es:

$$\sum_{k=0}^n a^k = \frac{a^{n+1} - 1}{a - 1} \quad (13)$$

Nótese que si $|a| < 1$ entonces $\lim_{n \rightarrow \infty} a^{n+1} = 0$; por lo tanto, la suma queda como:

$$\sum_{k=0}^{\infty} a^k = \frac{1}{1 - a} \quad (14)$$

Una serie se dice que es una **suma telescópica** si tiene la forma:

$$\sum_{i=1}^n (a_i - a_{i-1}) = a_n - a_0 \quad (15)$$

De acuerdo a la fórmula del binomio de Newton, tenemos para cada i :

$$(n + 1)^m - 1^m = \sum_{k=0}^{m-1} C_k^m s_{kn} \quad (16)$$

En donde $C_k^m = \frac{m!}{k!(m-k)!}$.

Resulta el sistema de ecuaciones

$$(n + 1) - 1 = s_{0n} \quad (17)$$

$$(n + 1)^2 - 1 = s_{0n} + 2s_{1n} \quad (18)$$

$$(n + 1)^3 - 1 = s_{0n} + 3s_{1n} + 3s_{2n} \quad (19)$$

$$(n + 1)^4 - 1 = s_{0n} + 4s_{1n} + 6s_{2n} + 4s_{3n} \quad (20)$$

$$(n + 1)^5 - 1 = s_{0n} + 5s_{1n} + 10s_{2n} + 10s_{3n} + 5s_{4n} \quad (21)$$

Al resolverlo para los primeros valores s_{mn} obtenemos:

$$s_{0n} = n \quad (22)$$

$$s_{1n} = \frac{1}{2}n(n + 1) \quad (23)$$

$$s_{2n} = \frac{1}{6}n(n + 1)(2n + 1) \quad (24)$$

$$s_{3n} = \frac{1}{4}n^2(n + 1)^2 \quad (25)$$

$$s_{4n} = \frac{1}{30}n(n+1)(2n+1)(3n^2+3n-1) \quad (26)$$

$$s_{5n} = \frac{1}{12}n^2(n+1)^2(2n^2+2n-1) \quad (27)$$

Propiedades de los logaritmos

El logaritmo de un número es el exponente al cual hay que elevar la base para obtener dicho número. Esto es, al despejar x de $b^x = n$ se tiene:

$$x = \log_b n \quad (28)$$

Esto se lee como “ x es igual al logaritmo de base b de n ”.

Los logaritmos tienen las siguientes propiedades, siempre y cuando tanto la base b como x e y sean mayores a cero, es decir, que $b, x, y > 0$:

$$\log(x \cdot y) = \log x + \log y \quad (29)$$

$$\log\left(\frac{x}{y}\right) = \log x - \log y \quad (30)$$

$$\log x^n = n \log x \quad (31)$$

$$y^{\log x} = x^{\log y} \quad (32)$$

$$\log 1 = 0 \quad (33)$$

$$\log_b x = \frac{1}{\log_x b} \quad (34)$$

$$\log_b b = 1 \quad (35)$$

Solución de recurrencias

El método maestro que se utiliza para resolver recurrencias de la forma:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad (36)$$

Donde $a \geq 1$, $b > 1$ son números naturales constantes y $f(n)$ es una función asintóticamente positiva. Una recurrencia de esta forma describe un algoritmo que divide un problema de tamaño n en a subproblemas, cada uno de tamaño n/b . Los a subproblemas se resuelven recursivamente, cada uno en tiempo $T(n/b)$. La función $f(n)$ comprende el costo de dividir el problema y combinar los resultados de los subproblemas.

En el método maestro existen tres casos para establecer las cotas asintóticas, dependiendo de las características de la recurrencia. Entonces $T(n)$ tiene las siguientes cotas asintóticas:

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ para algún $\epsilon > 0$ constante, entonces $T(n) = \Theta(n^{\log_b a})$.
2. Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante $\epsilon > 0$, y si $af(n/b) \leq cf(n)$ para alguna constante $0 < c < 1$ y todo n suficientemente grande, entonces $T(n) = \Theta(f(n))$.

Análisis de algoritmos y experimentos

Complejidad de algoritmos

Método de la burbuja

El Ordenamiento de burbuja (BubbleSort en inglés) es un algoritmo de ordenamiento simple. Funciona revisando cada elemento de la lista a ordenar con el siguiente, cambiándolos de posición si están en un orden incorrecto ($n > n + 1$). Es necesario repetir este proceso varias veces hasta que no se necesiten más cambios, lo que significa que la lista quedó ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

El Algoritmo 2 muestra el pseudocódigo del método de la burbuja.

Entrada:	Lista de números b de tamaño n
Salida:	Lista de números b ordenada
1:	Burbuja (b){
2:	for $i = 1$ to n do
3:	for $j = 1$ to $n - i$ do
4:	if $b_j > b_{j+1}$ then intercambiar(b_j, b_{j+1});
5:	end
6:	end
7:	return b ;
8:	}

Algoritmo 2 Pseudocódigo del método de ordenamiento de la burbuja.

“Supongamos que se desea ordenar las siguientes claves del arreglo unidimensional A. transportando en cada pasada el menor elemento hacia la parte izquierda del arreglo.

A: 15 67 08 16 44 27 12 35

Las comparaciones que se realizan son:

PRIMERA PASADA

$A[7] > A[8]$	$(12 > 35)$	no hay intercambio
$A[6] > A[7]$	$(27 > 12)$	sí hay intercambio
$A[5] > A[6]$	$(44 > 12)$	sí hay intercambio
$A[4] > A[5]$	$(16 > 12)$	sí hay intercambio
$A[3] > A[4]$	$(08 > 12)$	no hay intercambio
$A[2] > A[3]$	$(67 > 08)$	sí hay intercambio
$A[1] > A[2]$	$(15 > 08)$	sí hay intercambio

Luego de la primera pasada el arreglo queda así:

A: 08 15 67 12 16 44 27 35

Observe que el elemento más pequeño, en este caso el 08, fue situado en la parte izquierda del arreglo.

SEGUNDA PASADA

$A[7] > A[8]$	$(27 > 35)$	no hay intercambio
$A[6] > A[7]$	$(44 > 27)$	sí hay intercambio
$A[5] > A[6]$	$(16 > 27)$	no hay intercambio
$A[4] > A[5]$	$(12 > 16)$	no hay intercambio
$A[3] > A[4]$	$(67 > 12)$	sí hay intercambio
$A[2] > A[3]$	$(15 > 12)$	sí hay intercambio

Luego de la segunda pasada el arreglo queda así:

A: 08 12 15 67 16 27 44 35

Y el segundo elemento más pequeño del arreglo, en este caso 12 fue situado en la segunda posición.

En la tabla se representa el resultado de las pasadas restantes.” (Cairó y Guardati, 2006).

3a. pasada:	08	12	15	16	67	27	35	44
4a. pasada:	08	12	15	16	27	67	35	44
5a. pasada:	08	12	15	16	27	35	67	44
6a. pasada:	08	12	15	16	27	35	44	67
7a. pasada:	08	12	15	16	27	35	44	67

Fig. 3 Ejemplo de iteraciones del método de la burbuja durante el ordenamiento

Análisis de Eficiencia del método Burbuja

“El número de comparaciones que se realizan en el método de la burbuja se pueden contabilizar fácilmente. En la primera pasada se realizan $(n - 1)$ comparaciones, en la segunda pasada $(n - 2)$ comparaciones, en la tercera pasada $(n - 3)$ comparaciones y así sucesivamente hasta llegar a 2 y 1 comparaciones entra claves, siendo n el número de elementos del arreglo. Por lo tanto, tenemos que el número de comparaciones es:

$$T(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1 \quad (37)$$

$$= \sum_{k=1}^n (n - k) \quad (38)$$

$$= n \sum_{k=1}^n 1 - \sum_{k=1}^n k = n(n) - \frac{n(n + 1)}{2} \quad (39)$$

$$= O(n^2) + O(n^2) \quad (40)$$

$$= O(n^2) \quad (41)$$

El tiempo necesario para ejecutar el algoritmo de la burbuja es proporcional a $n^2, O(n^2)$, donde n es el número de elementos del arreglo.”

VENTAJAS:

1. Fácil implementación.
2. No requiere memoria adicional.

DESVENTAJAS:

1. Muy lento.
2. Realiza numerosas comparaciones.

- Realiza numerosos intercambios.

Método de Stooge

El Algoritmo 3 muestra el pseudocódigo del método de Stooge.

Entrada:	Lista de números b de tamaño n , i inicio de la lista, j fin de la lista
Salida:	Lista de números b ordenada
1:	Stooge (b, i, j) {
2:	if $b_i > b_j$ then intercambiar(b_i, b_j);
3:	if $i + 1 \geq j$ then return ;
4:	$k \leftarrow \lceil \frac{j-i+1}{3} \rceil$;
5:	Stooge ($b, i, j - k$);
6:	Stooge ($b, i + k, j$);
7:	Stooge ($b, i, j - k$);
8:	return b ;
9:	}

Algoritmo 3 Pseudocódigo del método de ordenamiento de Stooge

Esta función es recursiva, en donde se puede apreciar que en los llamados recursivos la lista de números se divide en $2/3$ partes. Esto es, desde la línea 2 hasta la línea 4 se pueden asumir como una cantidad constante de operaciones c , mientras que la línea 5 se representa como $T(2n/3)$, lo mismo para los llamados recursivos de las líneas 6 y 7. Por lo tanto, la cantidad de intercambios se puede establecer de la siguiente forma:

$$T(n) = f\left(\frac{2}{3}n\right) + f\left(\frac{2}{3}n\right) + f\left(\frac{2}{3}n\right) + c \quad (42)$$

$$= 3f\left(\frac{2}{3}n\right) + c \quad (43)$$

Resolviendo por el método maestro, se observa que se cumple el primer caso. Esto es, $a = 3$, $b = 2/3$ y $f(n) = c$. Es fácil ver que esta ecuación recurrente se resuelve con el primer caso del método maestro, es decir, $c = O(n^{\log_{1.5} 3 - \epsilon})$, para algún $\epsilon > 0$; por lo tanto, $T(n) = \Theta(n^{\log_{1.5} 3})$. De esta manera, la complejidad del algoritmo de ordenamiento por el método de Stooge es $\Theta(n^{\log_{1.5} 3})$.

Método Shell

“El método de Shell es una versión mejorada del método de inserción directa. Recibe ese nombre en honor de su autor, Donald L. Shell, quien lo propuso en 1959. Este método también se conoce como inserción con incrementos decrecientes.

Entrada:	Lista de números b de tamaño n
Salida:	Lista de números b ordenada
1:	Shell (b){
2:	$p \leftarrow \lfloor n/2 \rfloor$;
3:	while $p > 0$ do
4:	for $i = p$ to n do
5:	$j \leftarrow i - p$;
6:	while $j \geq 1$ do
7:	$k \leftarrow j + p$;
8:	if $b_j \leq b_k$ then $j \leftarrow 0$;
9:	else intercambiarse(b_j, b_k);
10:	$j \leftarrow j - p$;
11:	end
12:	end
13:	$p \leftarrow \lfloor p/2 \rfloor$;
14:	end
15:	return b ;
16:	}

Algoritmo 4 Seudocódigo del método de ordenamiento de Shell

En el método de ordenación por inserción directa cada elemento se compara para su ubicación correcta en el arreglo con los elementos que se encuentran en su parte izquierda. Si el elemento a insertar es el más pequeño que el grupo de elementos que se encuentran a su izquierda, será necesario efectuar varias comparaciones antes de su ubicación.

Shell propone que las comparaciones entre elementos se efectúen con saltos de mayor tamaño, pero con incrementos decrecientes; así, los elementos quedarán ordenados en el arreglo más rápidamente. Para comprender mejor este algoritmo analice el siguiente caso.

Consideremos un arreglo que contenga 16 elementos. En primer lugar, se dividirán los elementos del arreglo en ocho grupos, teniendo en cuenta los elementos que se encuentran a ocho posiciones de distancia entre sí, y se ordenarán por separado. Quedarán en el primer grupo los elementos ($A[1], A[9]$); en el segundo, ($A[2], A[10]$); en el tercero, ($A[3], A[11]$), y así sucesivamente. Después de este primer paso se dividirán los elementos del arreglo en

cuatro grupos, teniendo en cuenta ahora los elementos que se encuentren a cuatro posiciones de distancia entre sí, y se les ordenara por separado. Quedarán en el primer grupo los elementos $(A[1], A[5], A[13])$; en el segundo $(A[2], A[6], A[10], A[14])$, así sucesivamente. En el tercer paso se dividirán los elementos del arreglo en grupos, tomando en cuenta los elementos que se encuentran ahora a dos posiciones de distancia entre sí y nuevamente se les ordenara por separado.

En el primer grupo quedara $(A[1], A[3], A[5], A[7], A[9], A[11], A[13], A[15])$ y en el segundo $(A[2], A[4], A[8], A[10], A[12], A[14], A[16])$.

Finalmente se agruparán y ordenarán los elementos de manera normal; es decir, de uno en uno.

Supongamos que se desea ordenar los elementos que se encuentran en el arreglo unidimensional A utilizando el método de Shell.

A : 15 67 08 16 44 27 12 35 56 21 13 28 60 36 07 10

PRIMERA PASADA

Se dividen los elementos en 8 grupos:

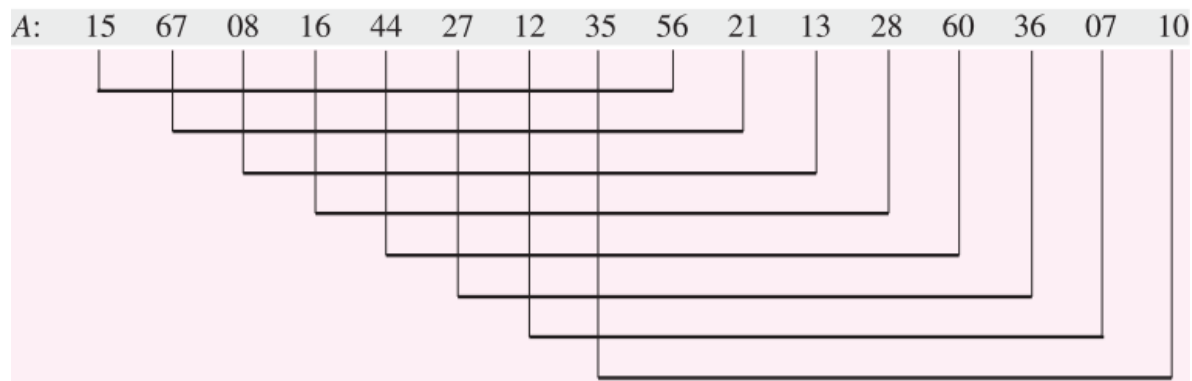


Fig. 4 Primera pasada de ordenamiento del método de Shell

La ordenación produce:

A : 15 21 08 16 44 27 07 10 56 67 13 28 60 36 12 35

SEGUNDA PASADA

Se dividen los elementos en 4 grupos:

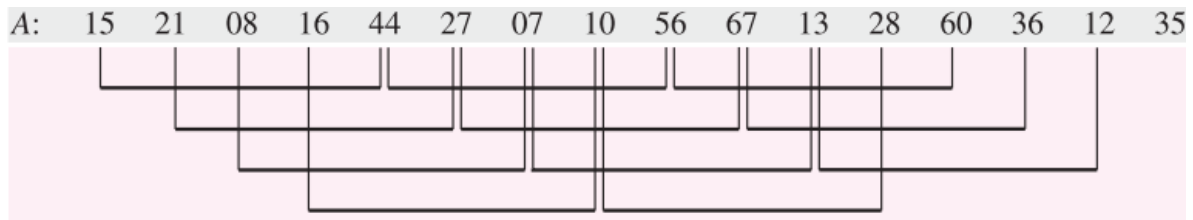


Fig. 5 Segunda pasada de ordenamiento del método de Shell

La ordenación produce:

A: 15 21 07 10 44 27 08 16 56 36 12 28 60 67 13 35

TERCERA PASADA

Se dividen los elementos en 2 grupos:



Fig. 6 Tercera pasada de ordenamiento del método de Shell

La ordenación produce:

A: 07 10 08 16 12 21 13 27 15 28 44 35 56 36 60 67

CUARTA PASADA

Se dividen los elementos en un solo grupo:

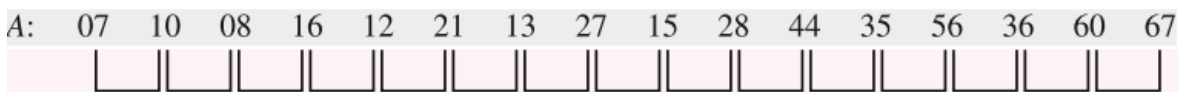


Fig. 7 Cuarta pasada de ordenamiento del método de Shell

La ordenación produce:

A: 07 08 10 12 13 15 16 21 27 28 35 36 44 56 60 67 “

En este método las comparaciones e intercambio de valores se hacen con un paso que disminuye conforme se van ordenando los números de la lista. El primer paso es del tamaño de la mitad de la lista, cada par de números son comparados con un paso de ese tamaño. Una vez recorrida la lista, se calcula el nuevo paso de comparación, que es la mitad del paso anterior; nuevamente, cada par de números son comparados con ese nuevo tamaño de paso, que para ese momento es un cuarto del tamaño de la lista, se recorre toda la lista. Posteriormente se vuelve a calcular el paso, dividiendo el a la mitad el paso anterior y se comparan los números con el nuevo paso, y así sucesivamente.

La cantidad de comparaciones se puede establecer como la suma de las diferencias entre el tamaño de la lista y el tamaño del paso en cada iteración. De esta forma, se puede establecer lo siguiente:

$$T(n) = \left(n - \frac{n}{2}\right) + \left(n - \frac{n}{2^2}\right) + \left(n - \frac{n}{2^3}\right) + \dots + \left(n - \frac{n}{2^k}\right) \quad (44)$$

$$= \sum_{i=1}^k \left(n - \frac{n}{2^i}\right) \quad (45)$$

La suma termina hasta cuando $n/2^k \leq 1$; por lo tanto, al despejar k se tiene que $\log_2 n \leq k$. De aquí, se tiene lo siguiente:

$$T(n) = \sum_{i=1}^k n - \sum_{i=1}^k \frac{n}{2^i} \quad (46)$$

$$= n \sum_{i=1}^k 1 - n \sum_{i=1}^k \frac{1}{2^i} \quad (47)$$

$$\leq n(k) - n(2), \text{ sustituyendo } k \geq \log n \quad (48)$$

$$\leq n \log n - 2n \quad (49)$$

$$= O(n \log n) + O(n) \quad (50)$$

$$= O(n \log n) \quad (51)$$

Este análisis se hace considerando que en las líneas 6 a 11 no cumplen con todo el recorrido de valores de la variable j . Es decir, la condición de la línea 6 deja de cumplirse en unas cuantas iteraciones, lo que representaría en un coeficiente constante que multiplica el término $n \log n$ de la ecuación anterior, lo que no afecta significativamente en la complejidad.

Método Comb

En 1991, Stephen Lacey y Richard Box propusieron el mismo algoritmo bajo el nombre de Combsort en su artículo A fast, easy sort Byte, vol. 16, issue 4, Abril de 1991.

La idea básica del algoritmo combsort es que el espacio pueda ser mucho mayor de uno.

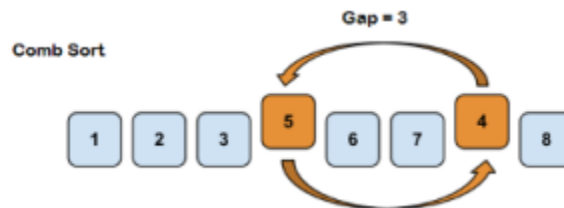


Fig. 8 Ejemplo de intercambio del método Comb

“El espacio se inicia como la longitud de la lista a ordenar dividida por el factor de encogimiento (generalmente 1,3), y la lista se ordena con este valor (redondeado al piso para obtener el entero si es necesario) para el espacio. Después el espacio se vuelve a dividir entre el factor de encogimiento (Gap), y la lista se ordena con este nuevo espacio, y el proceso se repite hasta que el espacio es 1. En esta fase, el algoritmo combsort continúa usando un espacio de 1 hasta que la lista está completamente ordenada.”.

Características:

1. Es un algoritmo comparativo.
2. Puede ordenar números negativos.
3. Puede ordenar números reales.
4. Puede ordenar grandes cantidades.

Este método es similar al método Shell, en donde las comparaciones e intercambio de valores se hacen con un paso que disminuye conforme se van ordenando los números de la lista. En el método Shell el paso se va reduciendo a la mitad, mientras que en el método Comb el paso se reduce por un factor de 1.3.

Entrada:	Lista de números b de tamaño n
Salida:	Lista de números b ordenada
1:	Comb (b) {
2:	$p \leftarrow n$;
3:	$so \leftarrow \text{false}$;
4:	while $so = \text{false}$ do
5:	$p \leftarrow \lfloor p/1.3 \rfloor$;
6:	if $p \leq 1$ then
7:	$p \leftarrow 1$;
8:	$so \leftarrow \text{true}$;
9:	end
10:	$i \leftarrow 1$;
11:	while $i + p \leq n$ do
12:	if $b_i > b_{i+p}$ then
13:	intercambiarse(b_i, b_{i+p});
14:	$so \leftarrow \text{false}$;
15:	end
16:	$i \leftarrow i + 1$;
17:	end
18:	end
19:	return b ;
20:	}

Algoritmo 5 Seudocódigo del método de ordenamiento de Comb

La cantidad de comparaciones se puede establecer como la suma de las diferencias entre el tamaño de la lista y el tamaño del paso en cada iteración. De esta forma, se puede establecer lo siguiente:

$$T(n) = \left(n - \left\lfloor \frac{n}{1.3} \right\rfloor\right) + \left(n - \left\lfloor \frac{n}{1.3^2} \right\rfloor\right) + \left(n - \left\lfloor \frac{n}{1.3^3} \right\rfloor\right) + \dots + \left(n - \left\lfloor \frac{n}{1.3^k} \right\rfloor\right) \quad (52)$$

$$= \sum_{i=1}^k \left(n - \left\lfloor \frac{n}{1.3^i} \right\rfloor\right) \quad (53)$$

La suma termina hasta cuando $n/1.3^k \leq 1$; por lo tanto, al despejar k se tiene que $k \geq \log_{1.3} n$. De aquí, se tiene lo siguiente:

$$T(n) = n \sum_{i=1}^k 1 - n \sum_{i=1}^k \left\lfloor \frac{1}{1.3^i} \right\rfloor \quad (54)$$

Para facilitar las operaciones, $1.3 = 13/10$, por lo tanto $1 \div 13/10 = 10/13$, de aquí tenemos:

$$T(n) = n \sum_{i=1}^k 1 - n \sum_{i=1}^k \left(\frac{10}{13}\right)^i = n(k) - n\left(\frac{13}{3}\right) \quad (55)$$

$$\leq n(k) - nO(1), \text{ sustituyendo } k \geq \log n \quad (56)$$

$$\leq n \log n + O(n) \quad (57)$$

$$= O(n \log n) + O(n) \quad (58)$$

$$= O(n \log n) \quad (59)$$

Método Mergesort

“Este método de ordenación divide el vector por la posición central, ordena cada una de las mitades y después realiza la mezcla ordenada de las dos mitades. El caso base es aquel que recibe un vector con ningún elemento, o con 1 solo elemento, ya que obviamente está ordenado.

El algoritmo anterior funciona de la siguiente manera. Tenemos un arreglo de enteros no ordenados de 10 posiciones:

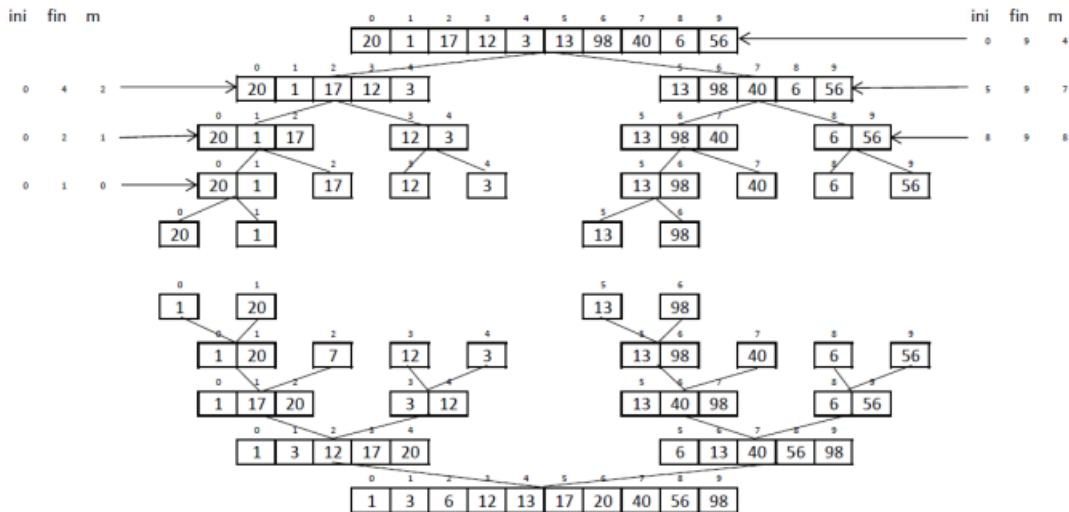


Fig. 9 Ejemplo de iteraciones del método de mergesort durante el ordenamiento

Estudio de la complejidad

El algoritmo es recursivo esa razón que se quiere determinar el tiempo empleado por cada una de las 3 fases del algoritmo divide y vencerás. Cuando se llama a la función mezclista se deben mezclar las dos listas más pequeñas en una nueva lista con n elementos. la función hace una pasada a cada una de las sublistas. Por consiguiente, el número de operaciones realizadas será como máximo el producto de una constante multiplicada por n . si se consideran las llamadas recursivas se tendrá entonces el número de operaciones: constante $\cdot n^k$ (profundidad de llamadas recursivas). El tamaño de la lista a ordenar se divide por dos en cada llamada recursiva, de modo que el número de llamadas es aproximadamente igual al número de veces que no se puede dividir por 2, parándose cuando el resultado es menor o igual a 1. Por consiguiente, la ordenación por mezcla se ejecutará, aproximadamente, el número de operaciones: alguna constante multiplicada por n y después por $(\log n)$, Resumiendo hay dos llamadas recursivas, y una iteración que tarda tiempo n por lo tanto la recurrencia es: $T(n) = n + 2T(n/2)$ si $n > 1$ y 1 en otro caso. De esta forma, aplicando expansión de recurrencia se tiene: $T(n) = n + 2T(n/2) = n + 2n/2 + 4T(n/4) = \dots = n + n + n + \dots n(k = \log n \text{ veces}) = n \log n$ por lo tanto la ordenación por mezcla tiene un tiempo de ejecución de $O(n \log n)$.” (Fager, Pantoja, Villacrés, 2014).

El método de mezcla o “merge” divide la lista en mitades de forma recursiva hasta que está ya no se puede dividir más. Los números de estas sublistas se ordenan y van regresando la recursividad para ir integrándolas, en cada paso que se unen estas las sublistas se ordenan los números. Al final, cuando se integra toda la lista de números, estos están parcialmente ordenados, pero se requieren pocos intercambios para terminar de ordenar la lista completa.

Entrada:	Lista de números b de tamaño n , i inicio de la lista, j fin de la lista
Salida:	Lista de números b ordenada
1:	Mergesort (b, i, j) {
2:	if $i < j$ then
3:	$m = \lfloor \frac{i+j}{2} \rfloor$;
4:	Mergesort (b, i, m);
5:	Mergesort ($b, m + 1, j$);
6:	Merge (b, i, m, j);
7:	end
8:	return b ;
9:	}

Algoritmo 6 Algoritmo Mergesort

Entrada:	Lista de números b de tamaño n , l inicio de la lista, m punto medio de la lista, h fin de la lista
Salida:	Lista de números b parcialmente ordenada
1:	Merge (b, l, m, h) {
2:	$p \leftarrow l; i \leftarrow l; j \leftarrow m + 1; c \in \mathbb{R}^n;$
3:	while $p \leq m$ and $j \leq h$ do
4:	if $b_p \leq b_j$ then
5:	$c_i \leftarrow b_p;$
6:	$p \leftarrow p + 1;$
7:	else
8:	$c_i \leftarrow b_j;$
9:	$j \leftarrow j + 1;$
10:	end
11:	$i \leftarrow i + 1;$
12:	end
13:	if $p > m$ then
14:	for $k = j$ to h do
15:	$c_i \leftarrow b_k;$
16:	$i \leftarrow i + 1;$
17:	end
18:	else
19:	for $k = p$ to m do
20:	$c_i \leftarrow b_k;$
21:	$i \leftarrow i + 1;$
22:	end
23:	end
24:	for $k = l$ to h do
25:	$b_k \leftarrow c_k;$
26:	end
27:	}

Algoritmo 7 Seudocódigo del método de ordenamiento de Merge

De esta forma, la ecuación que describe la complejidad de este método es:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \frac{5}{2}n \quad (60)$$

$$= 2T\left(\frac{n}{2}\right) + \frac{5}{2}n \quad (61)$$

Es fácil ver en el seudocódigo del método Mergesort que se invoca dos veces el método para arreglar la mitad de la lista, líneas 4 y 5. El término $5n/2$ se obtiene del método Merge que se invoca en la línea 6. Al revisar el seudocódigo de este último método se observa que las líneas 3-12 se repiten n veces, ya que se recorre toda la lista de números; en las líneas 13-17

y 19-23 se recorre la mitad de la lista, por lo tanto, se repiten $n/2$ veces. Las líneas 24-26 recorren toda la lista, por lo tanto, se repiten n veces. De aquí que al sumar todo tenemos $5n/2$.

Esta ecuación se resuelve con el método maestro, en donde $a = 2$, $b = 2$ y $f(n) = 5n/2$, que se puede resolver con el caso 2 del método maestro, donde se debe comprobar que $f(n) = \Theta(n^{\log_b a})$. Entonces $\log_b a = \log_2 2 = 1$, de esta forma se puede verificar fácilmente que $5n/2 = \Theta(n)$. Por lo tanto, la solución y complejidad de este algoritmo es $T(n) = \Theta(n \log n)$.

Quicksort

“El método de ordenación Quicksort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es también conocido como un método rápido y de ordenación por partición. Este método es una mejora sustancial del método de intercambio directo y se denomina Quicksort rápido por la velocidad con que la ordena los elementos del arreglo.

Entrada:	Lista de números b de tamaño n , i inicio de la lista, j fin de la lista
Salida:	Lista de números b ordenada
1:	Quicksort (b, i, j){
2:	if $i < j$ then
3:	$p \leftarrow \text{partition}(b, i, j)$;
4:	Quicksort ($b, i, p - 1$);
5:	Quicksort ($b, p + 1, j$);
6:	end
7:	return b ;
8:	}

Algoritmo 8 Algoritmo Quicksort

Entrada:	Lista de números b de tamaño n , l inicio de la lista, h fin de la lista
Salida:	Lista de números b parcialmente ordenada
1:	Partition (b, l, h) {
2:	$pv \leftarrow b_h$;
3:	$i \leftarrow l$;
4:	for $j = l$ to h do
5:	if $b_j < pv$ then
6:	intercambiarse(b_i, b_j);
7:	$i \leftarrow i + 1$;
8:	end
9:	end
10:	intercambiarse(b_i, b_h);
11:	return i ;
12:	}

Algoritmo 9 Seudocódigo del método de partición

Su autor, C. A. Hoare, lo llamo así. La idea central de este algoritmo consiste en lo siguiente:

1. Se toma el elemento X de una posición cualquiera del arreglo.
2. Se trata de ubicar a X en la posición correcta del arreglo, de tal forma que todos los elementos que se encuentren a su izquierda sean menores o iguales a X y todos los que se encuentren a su derecha sean mayores o iguales a X .
3. Se repiten los pasos anteriores, pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición de X en el arreglo.
4. El proceso termina cuando todos los elementos se encuentran en su posición correcta en el arreglo.

Se deben seleccionar, entonces, un elemento X cualquiera. En este caso se seleccionará $A[l]$. Se empieza a recorrer el arreglo de derecha a izquierda comparando si los elementos son mayores o iguales a X . Si un elemento no cumple con esta condición, se intercambian aquellos y se almacena una variable la posición del elemento intercambiado se acota el arreglo por la derecha. Se inicia nuevamente el recorrido, pero ahora de izquierda a derecha, comparando si los elementos son menores o iguales a X .

Si un elemento no cumple con esta condición, entonces se intercambian aquellos y se almacena en otra variable la posición del elemento intercambiado se acota el arreglo por la

izquierda. Se repiten los pasos anteriores hasta que el elemento X encuentre su posición correcta en el arreglo.

Supongamos que se desea ordenar los elementos que se encuentran en el arreglo A utilizando el método.

A : 15 67 08 16 44 27 12 35

Se selecciona $A[1]$, por lo tanto, $X \leftarrow 15$.

Se lleva a cabo las comparaciones que se muestran a continuación:

PRIMERA PASADA

Recorrido de derecha a izquierda

$A[8] \geq X$ $(35 \geq 15)$ no hay intercambio

$A[7] \geq X$ $(12 \geq 15)$ Sí hay intercambio

A : 12 67 08 16 44 27 15 35

Recorrido de izquierda a derecha

$A[2] \geq X$ $(67 \leq 15)$ Sí hay intercambio

A : 12 15 08 16 44 27 35

SEGUNDA PASADA

Recorrido de derecha a izquierda

$A[6] \geq X$ $(27 \geq 15)$ no hay intercambio

$A[5] \geq X$ $(44 \geq 15)$ no hay intercambio

$A[4] \geq X$ $(16 \geq 15)$ no hay intercambio

$A[3] \geq X$ $(08 \geq 15)$ Sí hay intercambio

A : 12 08 15 16 44 27 67 35

Como el recorrido de izquierda a derecha se debería iniciar en la misma posición donde se encuentra el elemento X , el proceso termina ya que se detecta que el elemento X se encuentra en la posición correcta. Observe que los elementos que forman parte del primer conjunto son menores o iguales a X , y los del segundo conjunto son mayores o iguales a X .



Fig. 10 Ejemplo de ordenamiento

Este proceso de particionamiento aplicado para localizar la posición correcta de un elemento X en el arreglo se repite cada vez que queden conjuntos formados por dos o más elementos. El método se puede aplicar de manera iterativa o recursiva.

En la tabla se representa la ubicación del resto de los elementos en el arreglo.”

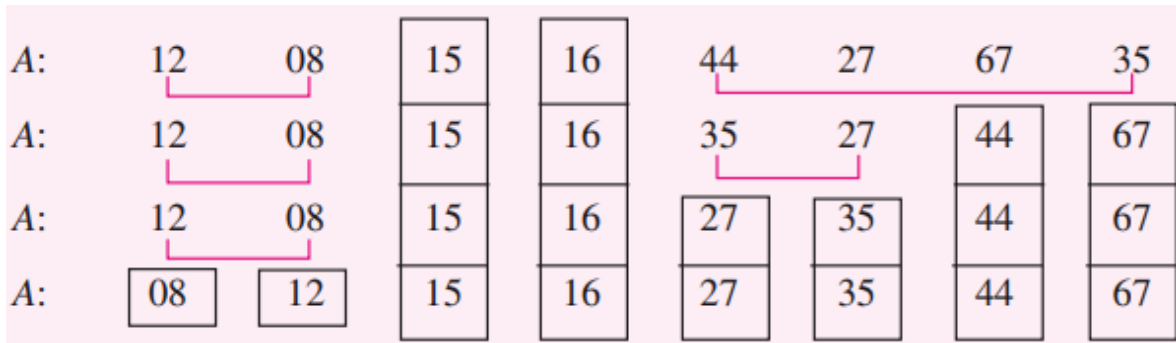


Fig. 11 Ubicación del resto de los elementos en el arreglo

Para calcular la complejidad de este método depende de la forma en que se escoja el pivote. En este pseudocódigo que se presenta se selecciona siempre al último elemento de la lista. Se puede ver en el método Partición que recorre toda la lista, por lo tanto, las líneas 4-9 se repiten n veces. Mientras que el método Quicksort se puede representar de la siguiente forma:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad (62)$$

$$= 2T\left(\frac{n}{2}\right) + n \quad (63)$$

Esta recurrencia se puede resolver con el método maestro, en donde $a = 2$, $b = 2$ y $f(n) = n$, que se puede resolver con el caso 2 del método maestro, donde se debe comprobar que $f(n) = \Theta(n^{\log_b a})$. Entonces $\log_b a = \log_2 2 = 1$, de esta forma se puede verificar fácilmente que $n = \Theta(n)$. Por lo tanto, la solución y complejidad de este algoritmo es $T(n) = \Theta(n \log n)$.

Es importante señalar que, con muy mala suerte, con este pivote y dependiendo como se encuentre la lista de número a ordenar, puede darse el caso que por cada número que se ordene se requiera recorrer toda la lista de número. Por lo que la recurrencia sería de la forma:

$$T(n) = T(n - 1) + n \quad (64)$$

Para resolverla se utiliza el método iterativo, teniendo:

$$T(n) = T(n - 1) + n \quad (65)$$

$$= T(n - 1 - 1) + n + n - 1 = T(n - 2) + 2n - 1 \quad (66)$$

$$= T(n - 1 - 2) + n - 1 + 2n - 1 = T(n - 3) + 3n - 2 \quad (67)$$

$$= T(n - 1 - 3) + n - 1 + 3n - 2 = T(n - 4) + 4n - 3 \quad (68)$$

⋮

$$= T(n - k) + kn - \sum_{i=0}^{k-1} i \quad (69)$$

$$= T(n - k) + kn - \frac{k(k - 1)}{2} \quad (70)$$

Las iteraciones se detienen hasta que $n = k$, y considerando que $T(0) = c$, entonces:

$$T(n) = T(0) + n^2 - \frac{n(n - 1)}{2} \quad (71)$$

$$= O(1) + O(n^2) + O(n^2) \quad (72)$$

$$= O(n^2) \quad (73)$$

VENTAJAS:

1. Es muy rapido.
2. No requiere memoria adicional.

DESVENTAJAS:

1. Implementación un poco más complicada.
2. Demasiada diferencia entre el peor y el mejor caso.

Experimentos realizados

Los algoritmos se implementan en dos lenguajes de programación diferentes y se ejecutan 30 veces cada uno para listas de tamaños desde 500 hasta 10000 elementos, con incrementos de 500 elementos. Es decir, 500, 1000, 1500, 2000,..., 10000. Se cuentan la cantidad de intercambios y de comparaciones de valores que hace cada algoritmo, en cada corrida. Se promedia la cantidad de intercambio de valores y se gráfica. Se grafican las funciones de complejidad de cada algoritmo y se comparan con las gráficas obtenidas al contar la cantidad de intercambio de valores. Este procedimiento se hace en dos computadoras con características diferentes de hardware.

Lenguaje de programación C/C++

Una de las computadoras que se utilizó para el desarrollo de los experimentos de estos algoritmos se programó en C/C++. Es un lenguaje de programación que proviene de la extensión del lenguaje C para que pudiese manipular objetos. A pesar de ser un lenguaje con muchos años, su gran potencia lo convierte en uno de los lenguajes de programación más demandados.

“Fue diseñado a mediados de los años 80 por el danés Bjarne Stroustrup. Su intención fue la de extender el lenguaje de programación C (con mucho éxito en ese momento) para que tuviese los mecanismos necesarios para manipular objetos. Por lo tanto, C++ contiene los paradigmas de la programación estructurada y orientada a objetos, por lo que se le conoce como un lenguaje de programación multiparadigma.” (Robledano, 2019).

Esta Computadora cuenta con las siguientes características:

Fabricante:	HP
Modelo:	HP Pavilion Laptop 15-cw0xxx
Procesador:	AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx 2.00 GHz
RAM:	12.0 GB (10.9 GB utilizable)

Tipo de sistema: Sistema operativo de 64 bits, procesador x64

Edición: Windows 11 Home Single Lenguaje

Lenguaje de programación Java

“James Gosling, un programador de Sun Microsystems, crea en 1995 el lenguaje Java, que, en sus orígenes, no se llamó de esa forma y, tras ciertos cambios, evolucionó hasta llegar a lo que hoy conocemos. Al ser un lenguaje de programación multipropósito con el que podemos realizar cualquier tipo de programa, su uso se ha extendido enormemente. Java se ha hecho muy famoso por una de sus principales características: es un lenguaje independiente de la plataforma. Eso quiere decir que, si hacemos un programa en Java, podrá funcionar en cualquier computadora del mercado. Esto es una ventaja significativa para los desarrolladores de software, pues antes tenían que crear un programa para cada sistema operativo, como Windows, Linux, IOS, etcétera. Java es independiente de la plataforma porque tiene una máquina virtual para cada sistema; esta máquina funciona como puente entre el sistema operativo y el programa en Java, y posibilita que este último se lea y se ejecute a la perfección.

En sus comienzos, Java fue concebido para ser utilizado en todo tipo de electrodomésticos; sin embargo, esta idea inicial no llegó a buen puerto. En ese momento, uno de los programadores, en vez de abandonar el proyecto, definió una nueva dirección, pero esta vez con un objetivo más ambicioso: orientarlo a Internet, lo que para esa época sonaba como una locura, pues se trataba de una tecnología que daba sus primeros pasos. A partir de entonces, logró ser integrado al navegador Netscape (el más importante del momento), donde se encargaba de ejecutar programas o applets dentro de una página web. Con el paso de los años, Java se convirtió en un lenguaje potente, seguro y universal, gracias a la posibilidad de usarlo en cualquier plataforma y por su carácter gratuito. Actualmente, Java se utiliza para un amplio abanico de posibilidades, y casi todo lo que se puede hacer en cualquier otro lenguaje es posible lograrlo también en Java, muchas veces, con más ventajas. Para lo que aquí nos interesa, con Java podemos programar páginas web dinámicas con accesos a bases de datos, utilizar XML y aprovechar diversos tipos de conexión de red entre cualquier

sistema, entre otras funciones. En general, Java será útil para cualquier aplicación que deseemos desarrollar, por ejemplo, programas con acceso mediante la Web, y apps para sistemas móviles, como Android, lo que en la actualidad se encuentra en pleno auge. Podemos concluir que Java es un lenguaje que llegó para quedarse, de eso no nos cabe ninguna duda; siempre ha sabido responder a los incesantes cambios tecnológicos y ha vuelto sobre sus pasos cuando fue necesario. Ahora apunta al mundo de los sistemas móviles; esto es importante pues las estadísticas indican que el uso de Android llega a casi un 70% de los teléfonos de este tipo en el mundo.”

Se utilizo NetBeans como IDE (entorno de desarrollo integrado) en una laptop de las siguientes características:

Fabricante:	ASUSTeK COMPUTER INC.
Modelo:	ASUS TUF Gaming A15
Procesador:	AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
RAM:	16.0 GB (15.4 GB usable)
Tipo de sistema:	Sistema operativo de 64 bits, procesador basado en x64
Edición:	Windows 11 Home Single Language

Resultados y discusión

En este capítulo se muestran los resultados obtenidos de las corridas de los métodos de ordenamiento mencionados anteriormente utilizando los lenguajes de programación C/C++ y Java en dos diferentes procesadores. Posteriormente se comparan los resultados obtenidos con las funciones de complejidad obtenidos.

Experimentos utilizando lenguaje C/C++

En la siguiente Fig. 12 gráfica podemos ver los datos de intercambios de los algoritmos Merge, Shell, Burbuja, Stooje, Quicksort, Comb.

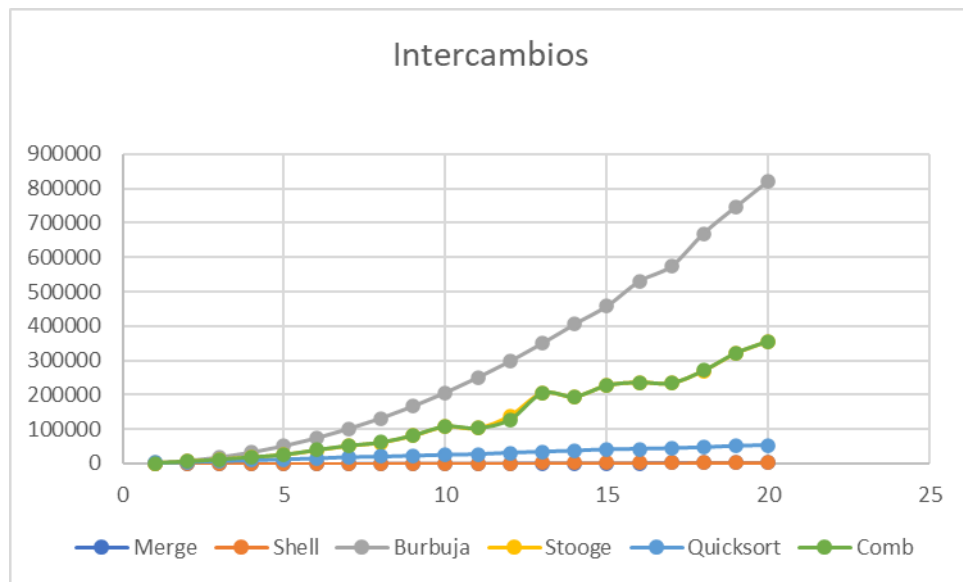


Fig. 12 Comparación de cantidades de intercambios con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooje, Quicksort y Comb.

En la siguiente Fig. 13 podemos ver los datos de intercambios de los algoritmos más eficientes.

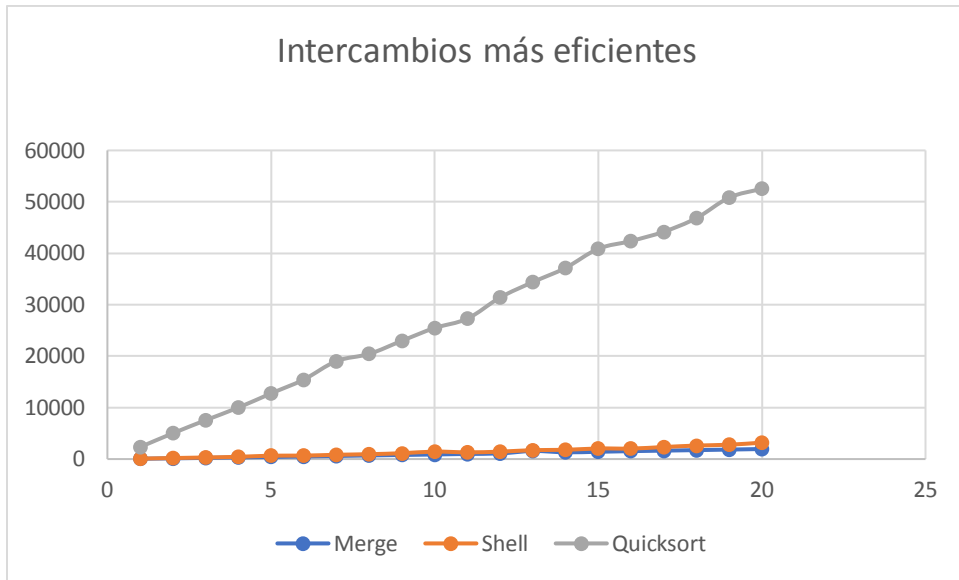


Fig. 13 Comparación de métodos con las cantidades de intercambios más bajas.

En la siguiente gráfica podemos ver los datos de intercambios de los algoritmos menos eficientes.

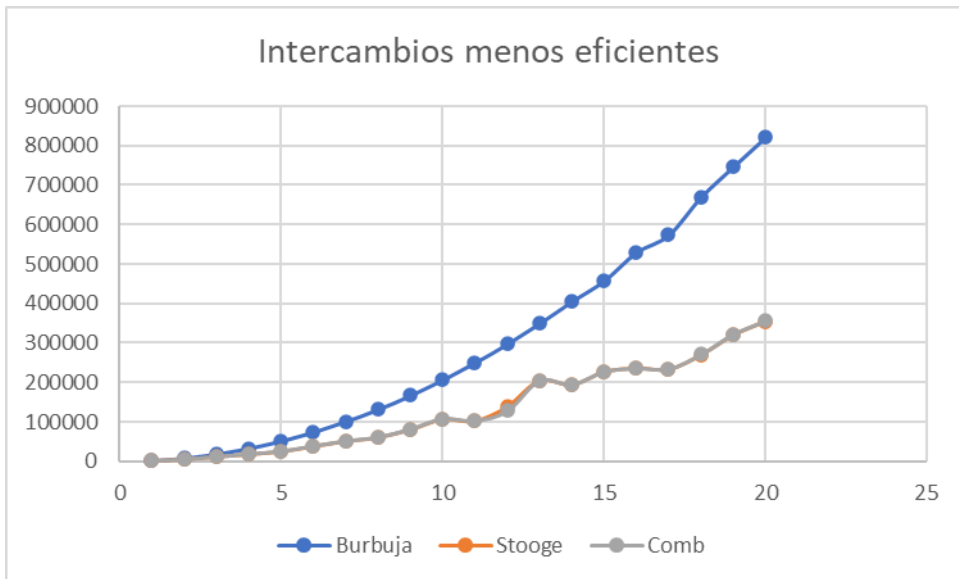


Fig. 14 Comparación de métodos con las cantidades de intercambios más altas

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos Merge, Shell, Burbuja, Stooge, Comb.

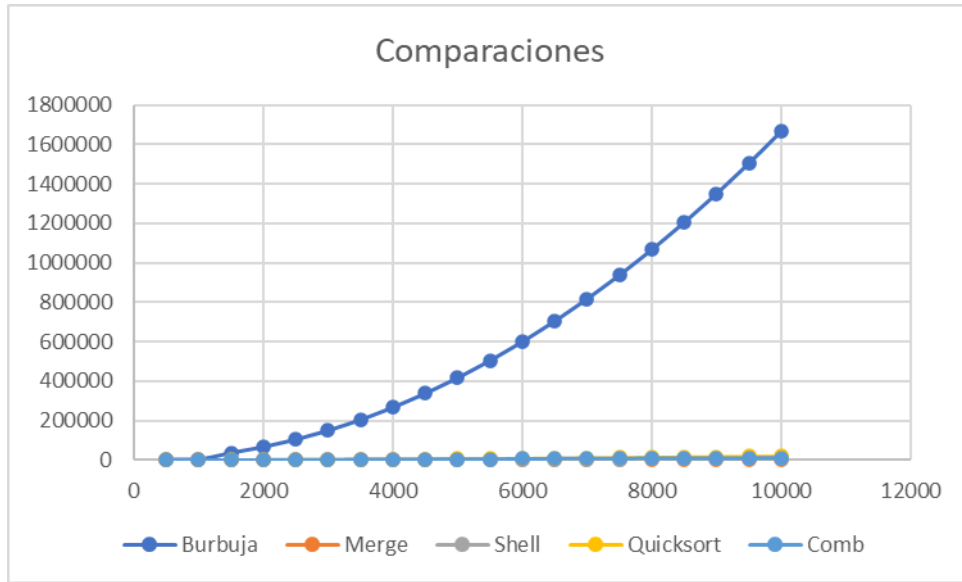


Fig. 15 Comparación de cantidades de comparaciones con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooze, Quicksort y Comb.

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos más eficientes.

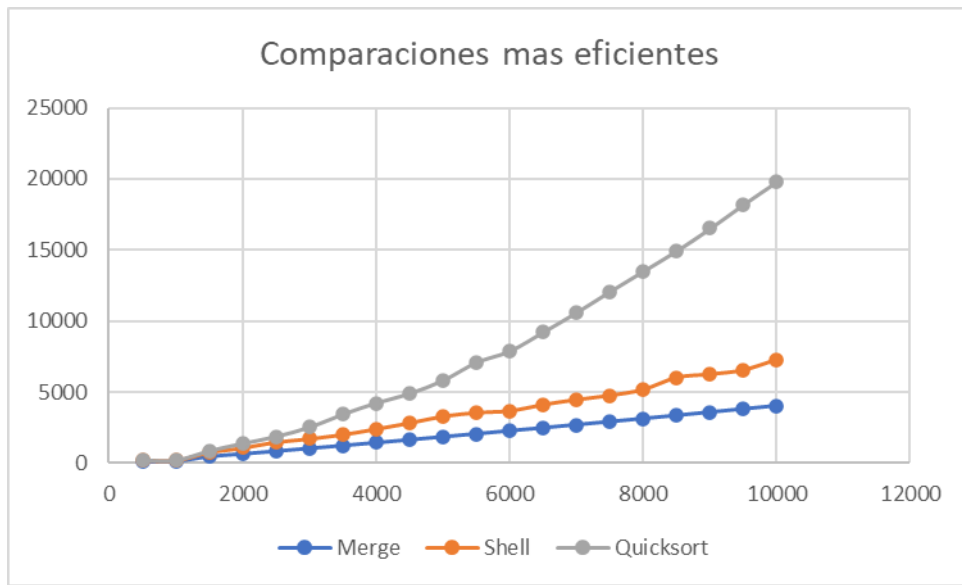


Fig. 16 Comparación de métodos con las cantidades de comparaciones más bajas.

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos menos eficientes.

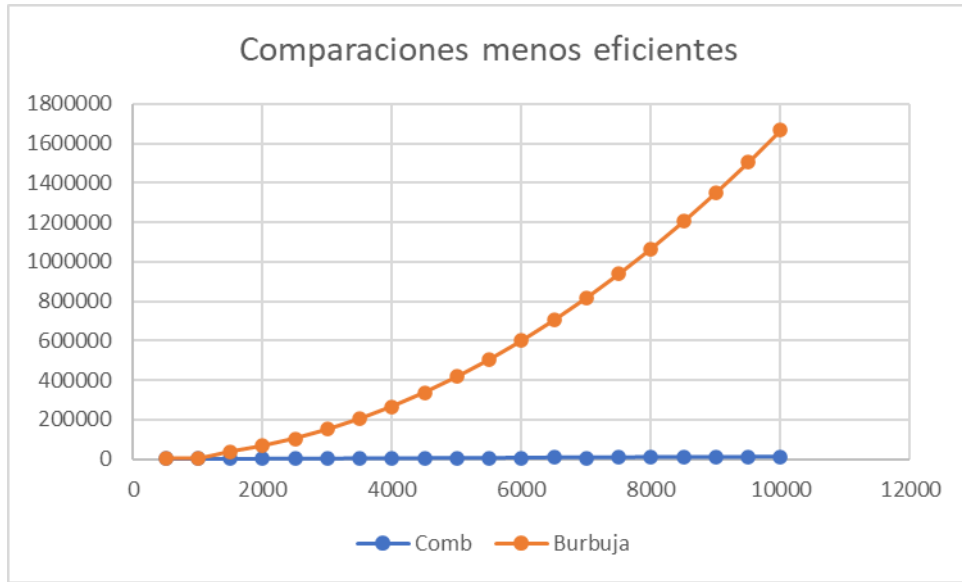


Fig. 17 Comparación de métodos con las cantidades de comparaciones más altas

En la siguiente gráfica podemos ver los datos de comparaciones de Stooge.

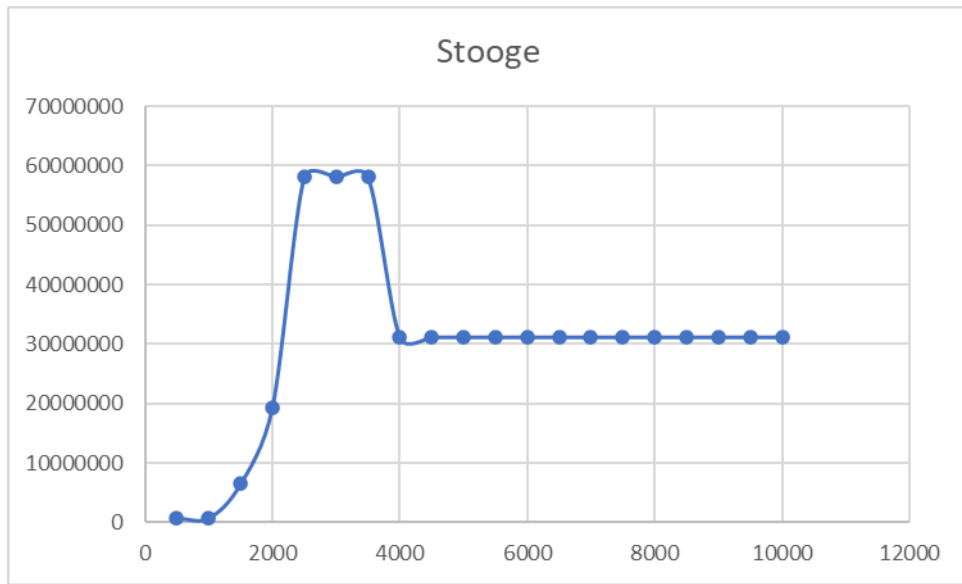


Fig. 18 Método Stooge con las cantidades de comparación más altas

Experimentos utilizando lenguaje Java

En la siguiente gráfica podemos ver los datos de intercambios de los algoritmos Merge, Shell, Burbuja, Stooge, Quicksort, Comb.

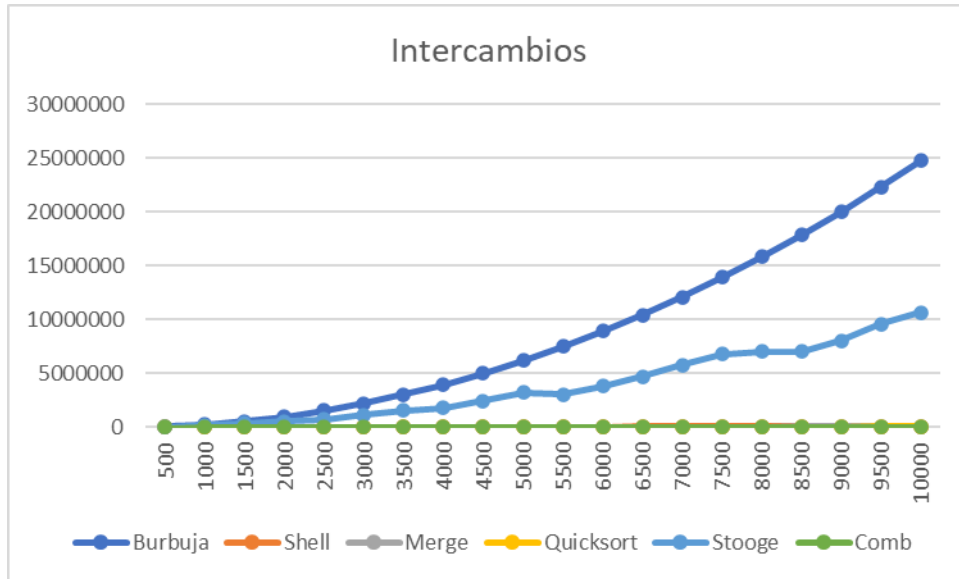


Fig. 19 Comparación de cantidades de intercambios con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooge, Quicksort y Comb.

En la siguiente podemos ver los datos de intercambios de los algoritmos más eficientes.

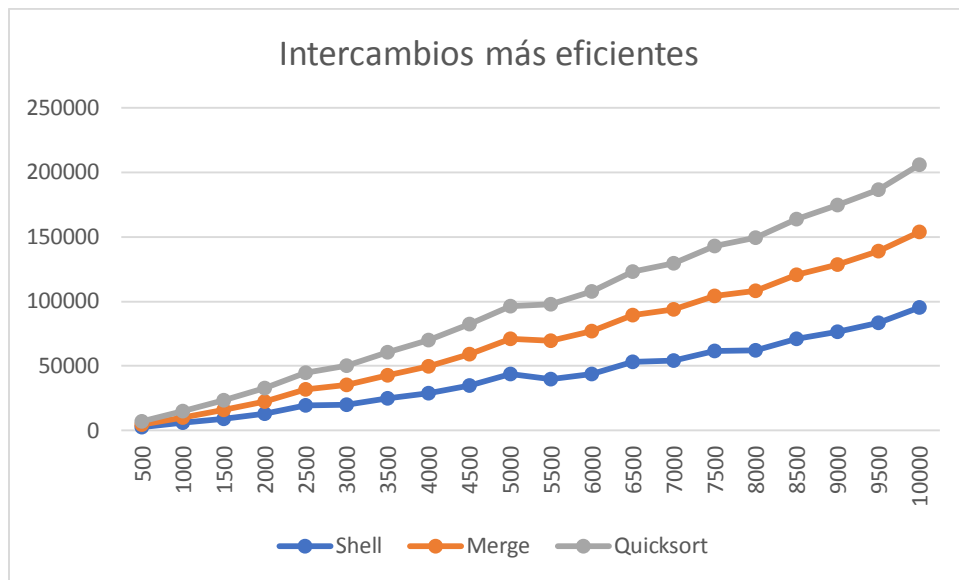


Fig. 20 Comparación de métodos con las cantidades de intercambios más bajas.

En la siguiente gráfica podemos ver los datos de intercambios de los algoritmos menos eficientes.

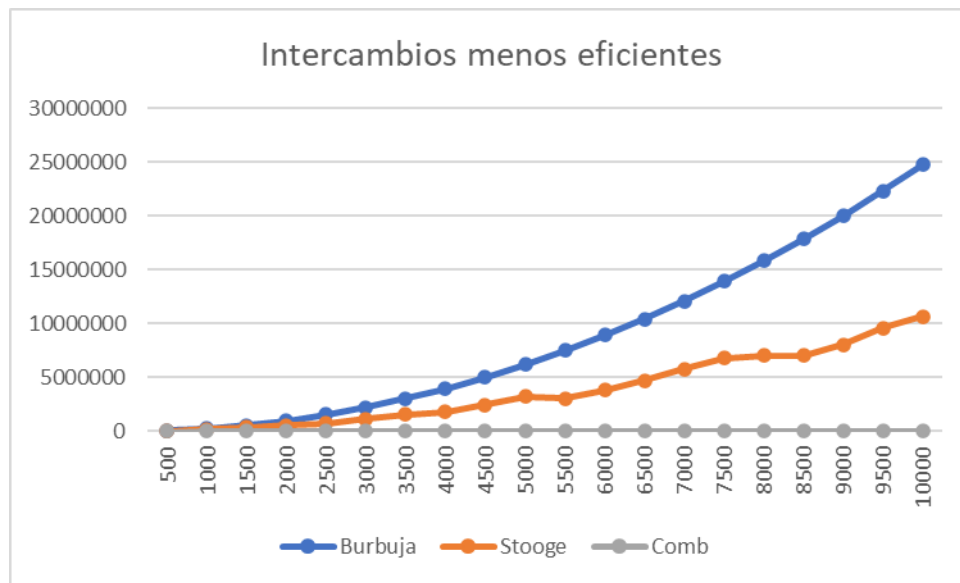


Fig. 21 Comparación de métodos con las cantidades de intercambios más altas

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos Merge, Shell, Burbuja, Stooge, Comb.

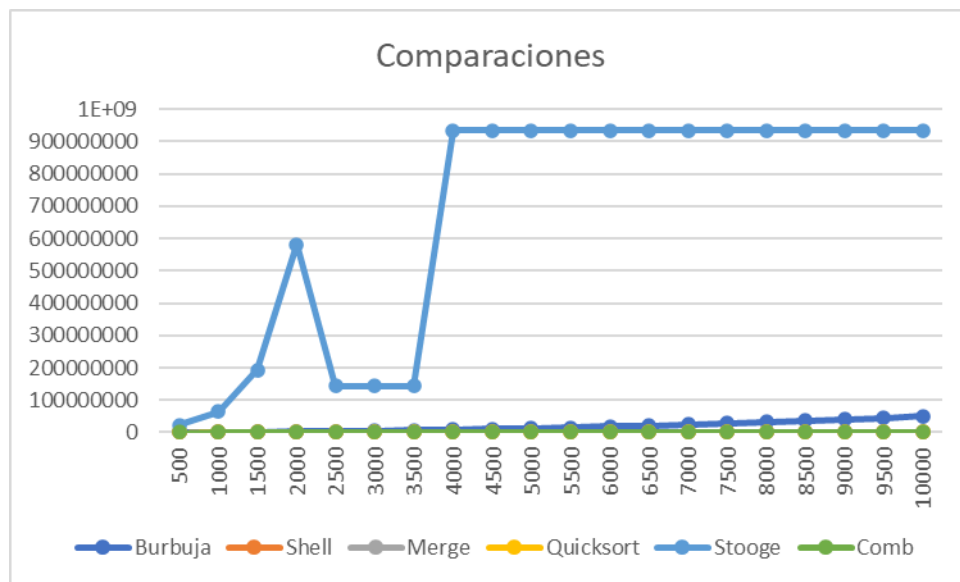


Fig. 22 Comparación de cantidades de comparaciones con diferentes tamaños de arreglos de números utilizando los métodos Merge, Shell, Burbuja, Stooge, Quicksort y Comb implementado en java.

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos más eficientes.

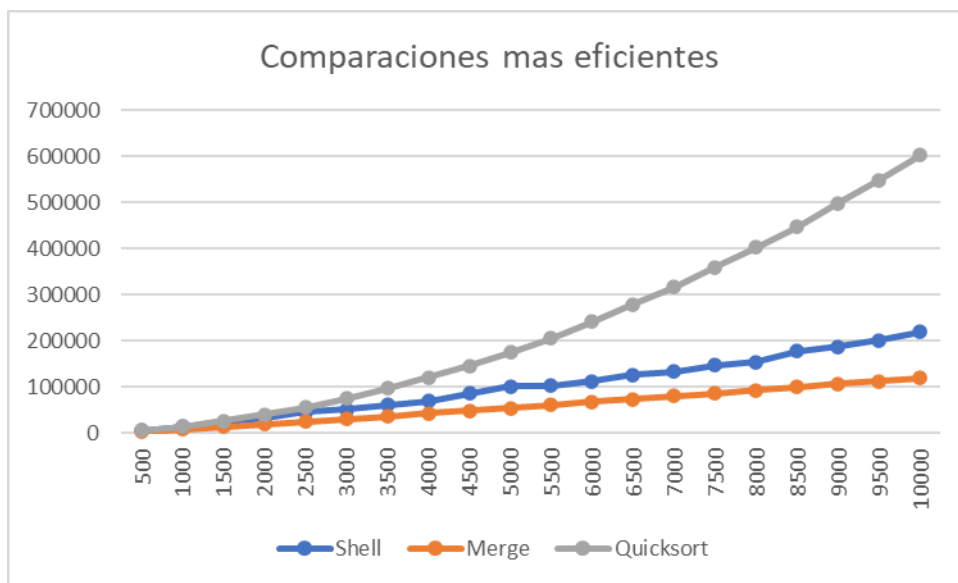


Fig. 23 Numero de comparaciones realizadas con los métodos Shell, Merge y Quicksort implementados en Java.

En la siguiente gráfica podemos ver los datos de comparaciones de los algoritmos menos eficientes.

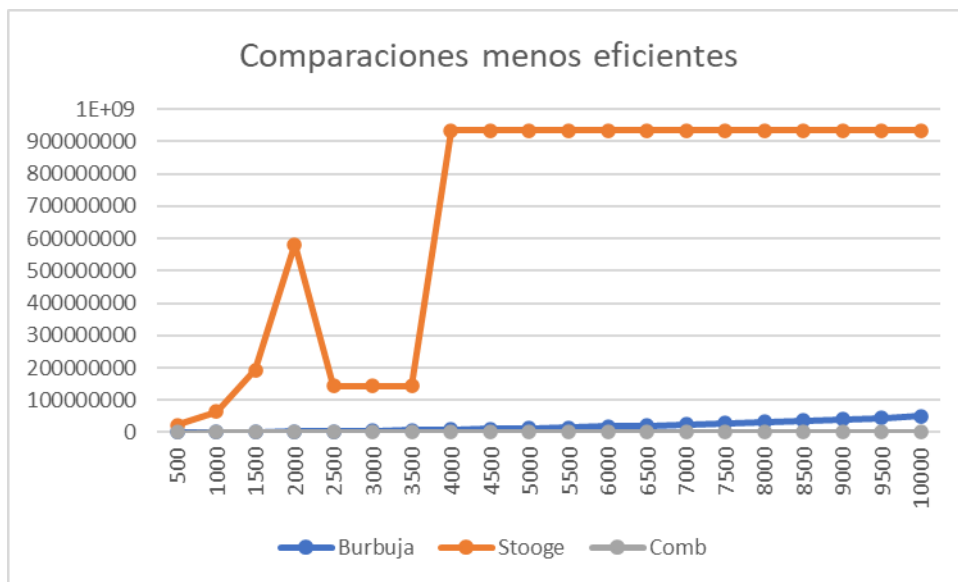


Fig. 24 Número de comparaciones con los métodos menos eficientes implementados en Java.

Comparación asintótica de resultados obtenidos utilizando C/C++

Tabla 3 Valores graficados método de la burbuja

Burbuja		
n	f(n)	O(g(n))
500	2053.73557	187500
1000	8253.79334	750000
1500	18541.0732	1687500
2000	32820.2757	3000000
2500	51454.0025	4687500
3000	74137.4229	6750000
3500	101208.843	9187500
4000	131655.364	12000000
4500	166749.235	15187500
5000	206153.718	18750000
5500	249878.12	22687500
6000	297276.97	27000000
6500	349326.645	31687500
7000	404980.899	36750000
7500	457523.565	42187500
8000	529409.259	48000000
8500	573559.988	54187500
9000	669348.235	60750000
9500	746361.558	67687500
10000	821040.895	75000000

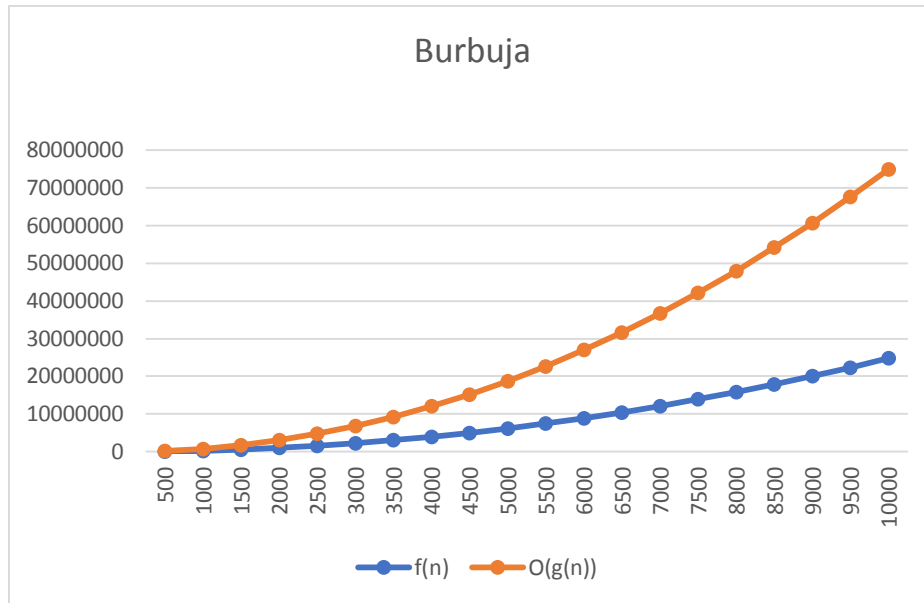


Fig. 25 Comparación de pasos obtenidos para el método de la burbuja y una función n^2

Tabla 4 Valores graficados método Stooze

Stooze		
n	f(n)	O(g(n))
500	1399.97666	1027.68426
1000	5687.60555	6722.08312
1500	12084.7944	20166.2494
2000	18451.7745	43969.148
2500	25290.751	80487.2011
3000	38572.2632	131907.444
3500	51284.9831	200291.439
4000	60663.349	287602.213
4500	80830.7945	395722.332
5000	106729.551	526466.81
5500	102454.289	681592.723
6000	138667.67	862806.638
6500	204416.525	1071770.5
7000	194067.175	1310106.38
7500	226693.452	1579400.43

8000	235843.244	1881206.17
8500	233090.286	2217047.27
9000	269959.79	2588419.91
9500	320065.547	2996794.94
10000	354867.539	3443619.59

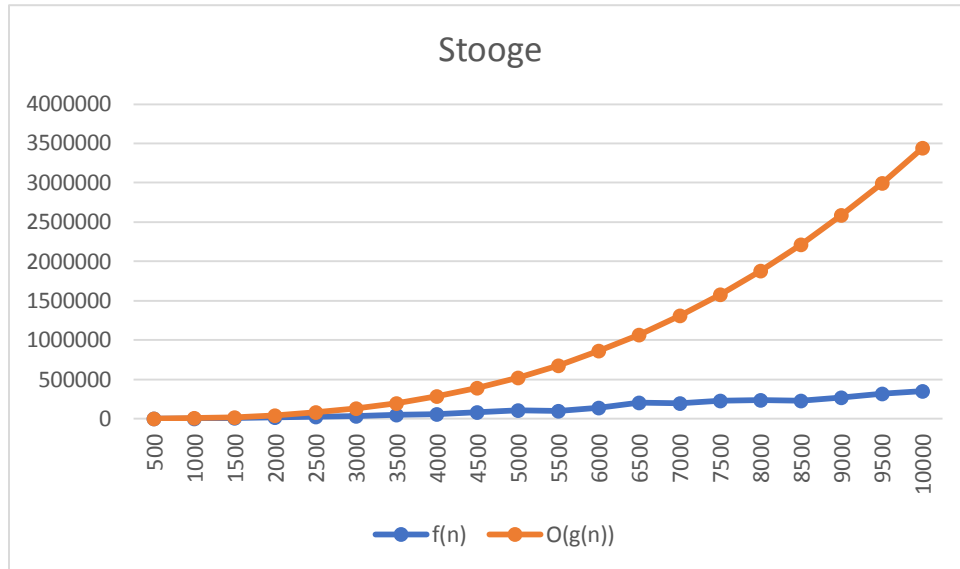


Fig. 26 Comparación de pasos obtenidos para el método de Stooge y una función $O(n^{\log_{1.5} 3})$

Tabla 5 Valores graficados método Shell

Shell		
n	f(n)	O(g(n))
500	85.5277777	54.8944744
1000	198.485555	151.501845
1500	308.513334	255.115508
2000	434.69111	380.956629
2500	658.249988	616.28601
3000	673.233331	632.50021
3500	834.395553	809.746478
4000	948.812219	938.373271
4500	1137.57223	1154.83403

5000	1448.08335	1520.47722
5500	1311.12111	1357.87375
6000	1441.16414	1512.21627
6500	1735.09345	1867.09895
7000	1794.46313	1939.69559
7500	2058.58487	2265.97437
8000	2038.65233	2241.17206
8500	2342.2674	2621.86153
9000	2625.14113	2981.68236
9500	2791.67335	3195.60459
10000	3192.11132	3715.71182

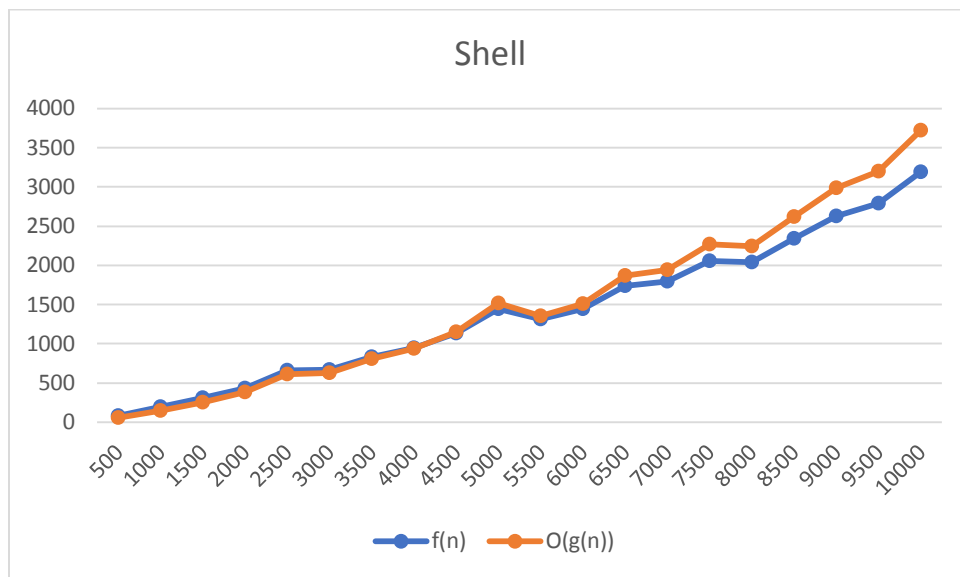


Fig. 27 Comparación de pasos obtenidos para el método de Shell y una función $O(n \log n)$

Tabla 6 Valores graficados método Comb

Comb		
n	f(n)	O(g(n))
500	1505.63555	1589.37251
1000	5913.40111	7409.35701
1500	12207.5722	16572.3758

2000	18917.9611	26877.6345
2500	25876.5889	37933.4227
3000	39276.3414	59941.0893
3500	52110.331	81653.1461
4000	61592.2993	97996.1961
4500	81895.3078	133665.371
5000	107910.868	180421.367
5500	103038.01	171587.313
6000	127777.378	216752.253
6500	204241.836	360280.941
7000	194539.238	341799.637
7500	227201.365	404273.294
8000	236060.613	421339.815
8500	233951.887	417273.131
9000	270879.282	488863.711
9500	321650.289	588463.496
10000	355601.704	655726.082

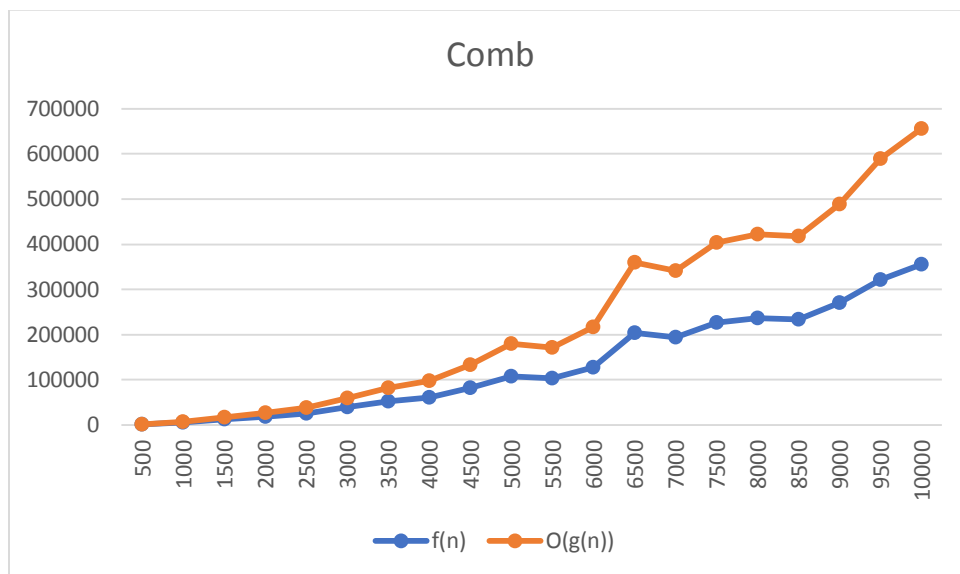


Fig. 28 Comparación de pasos obtenidos para el método de Comb y una función $O(n^2)$

Tabla 7 Valores graficados método Merge

Merge		
n	f(n)	O(g(n))
500	63.2355558	189.158544
1000	142.8	511.070647
1500	226.704445	886.943608
2000	319.002223	1326.63816
2500	406.994446	1764.09001
3000	502.930002	2256.70068
3500	598.795555	2762.21985
4000	703.018884	3324.3723
4500	796.630001	3838.86677
5000	896.044444	4393.945
5500	996.691111	4964.0214
6000	1104.01908	5580.01572
6500	1565.94991	8309.57412
7000	1317.07453	6824.50153
7500	1421.09557	7441.41619
8000	1538.46357	8144.06812
8500	1642.26305	8770.89135
9000	1741.13223	9372.34944
9500	1845.39512	10011.0047
10000	1955.37834	10689.3026

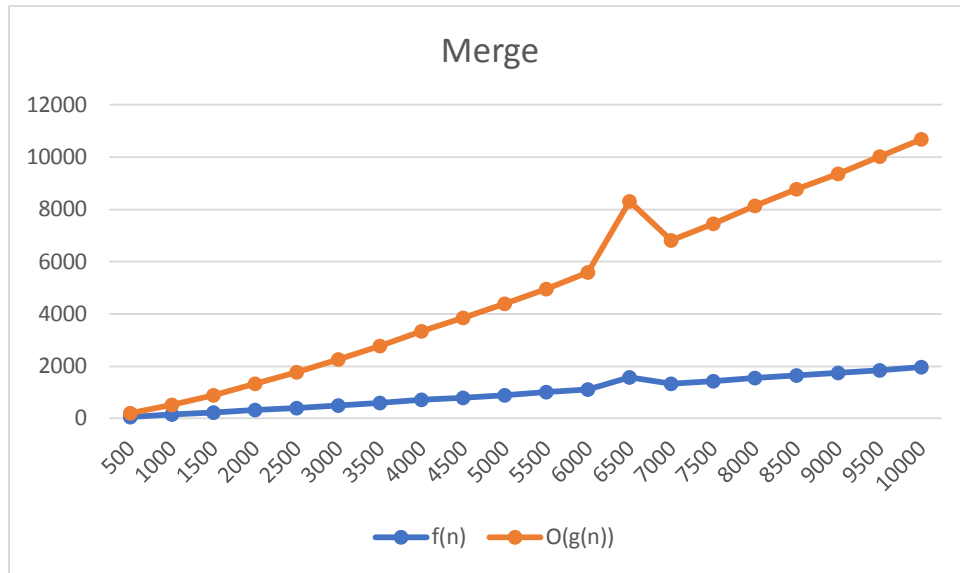


Fig. 29 Comparación de pasos obtenidos para el método de Merge y una función $O(n \log n)$

Tabla 8 Valores graficados método Quicksort

Quicksort		
n	f(n)	O(g(n))
500	2335.3	2613.05876
1000	5013.46667	6162.34909
1500	7550.46667	9726.7755
2000	10004.1667	13293.8502
2500	12758.4	17401.3856
3000	15450.9	21500.5504
3500	19047.9	27081.0553
4000	20437.8667	29264.8934
4500	22992.3333	33313.282
5000	25480.6667	37296.3459
5500	27286.7667	40209.5442
6000	31410.2333	46923.5812
6500	34397.6333	51837.308
7000	37162.4	56418.3059
7500	40877.8	62620.8242

8000	42385.4	65151.7842
8500	44189.9333	68191.3869
9000	46834.8333	72665.6223
9500	50827.7	79460.6071
10000	52554.5333	82413.5365

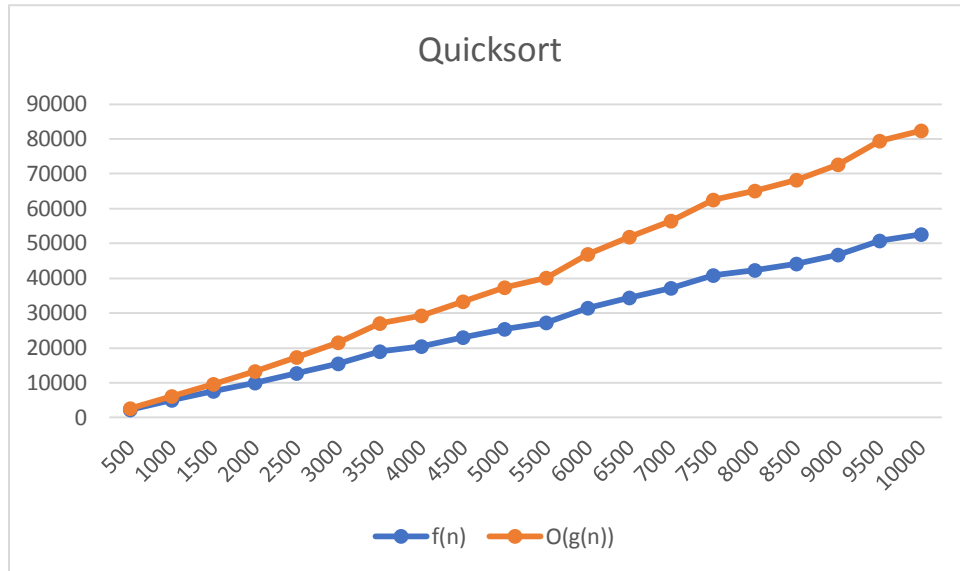


Fig. 30 Comparación de pasos obtenidos para el método de Quicksort y una función $O(n \log n)$

Comparación asintótica de resultados obtenidos utilizando Java

Tabla 9 Valores graficados método de la burbuja

Burbuja		
n	f(n)	O(g(n))
500	61688.8333	187500
1000	247072.267	750000
1500	556651.433	1687500
2000	989731.4	3000000
2500	1546600.4	4687500
3000	2227128.23	6750000
3500	3031419.17	9187500
4000	3957988.8	12000000
4500	5009800.93	15187500
5000	6183667.57	18750000
5500	7486112.67	22687500
6000	8913395.63	27000000
6500	10456461.8	31687500
7000	12124521.2	36750000
7500	13921196.4	42187500
8000	15839388.7	48000000
8500	17875557.8	54187500
9000	20032716.2	60750000
9500	22314249.6	67687500
10000	24750128	75000000

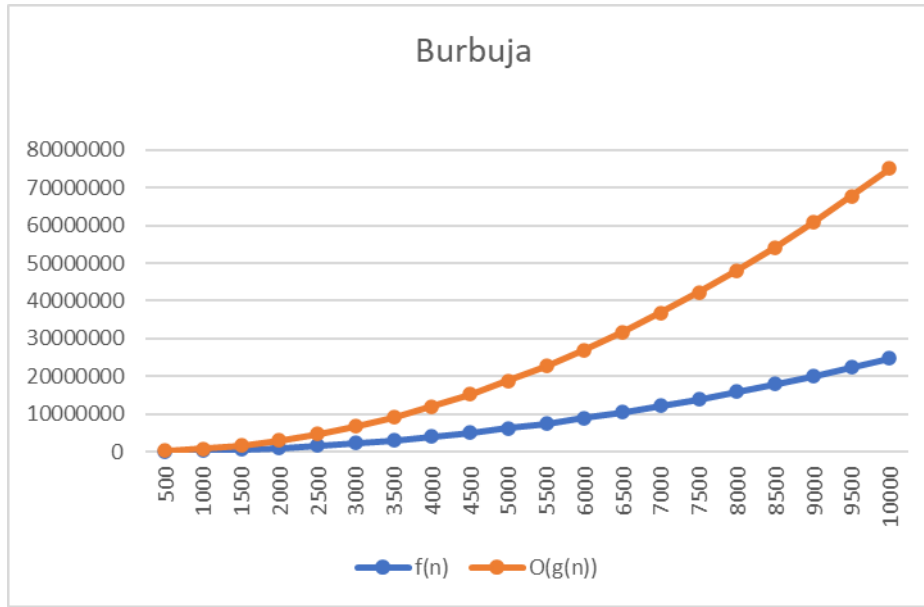


Fig. 31 Comparación de pasos obtenidos para el método de la Burbuja y una función $O(n^2)$

Tabla 10 Valores graficados método Stooge

Stooge		
n	f(n)	O(g(n))
500	41880.0333	10276.84263
1000	170092.267	67220.83121
1500	355959.4	201662.4936
2000	555546.067	439691.4804
2500	760252.2	804872.0108
3000	1158409.3	1319074.441
3500	1536361.67	2002914.395
4000	1820936.77	2876022.127
4500	2427460.07	3957223.324
5000	3201809.4	5264668.103
5500	3069047.1	6815927.233
6000	3826481.73	8628066.38
6500	4721633.53	10717704.95
7000	5806278.03	13101063.76
7500	6793334.73	15794004.31

8000	7027812.67	18812061.73
8500	7022994.47	22170472.66
9000	8077907.63	25884199.14
9500	9568990.53	29967949.4
10000	10670506.9	34436195.9

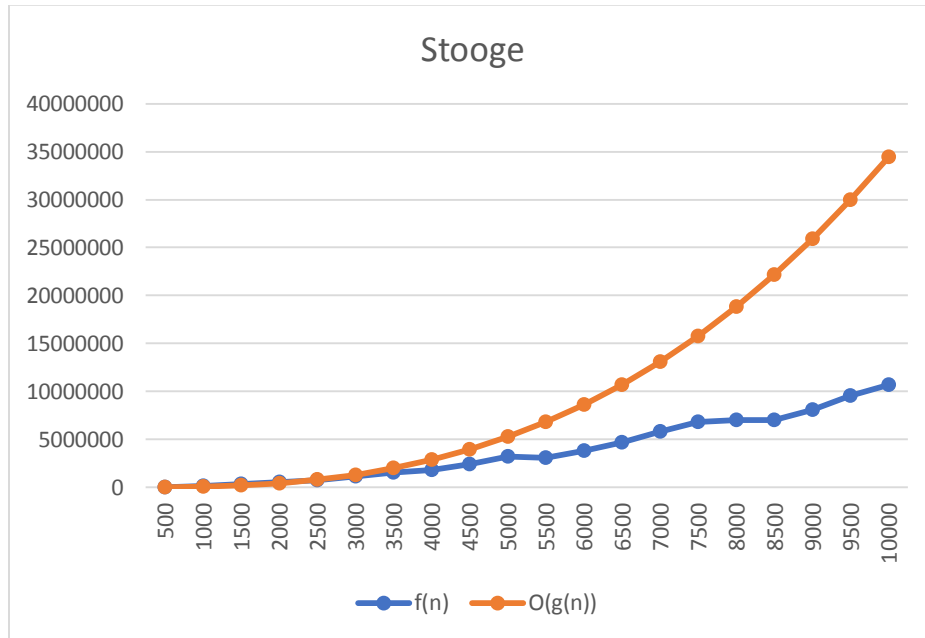


Fig. 32 Comparación de pasos obtenidos para el método de Stooge y una función $O(n^{\log_{1.5} 3})$

Tabla 11 Valores graficados método Shell

Shell		
n	f(n)	O(g(n))
500	2586.9	2932.770747
1000	5848.9	7319.283986
1500	9102.4	11971.50494
2000	13023.0667	17800.94539
2500	19582.4	27919.15635
3000	20114.3667	28755.37527
3500	24898.4333	36361.09277
4000	28626.3667	42381.49888

4500	35008.4333	52846.68255
5000	43885.5667	67677.94633
5500	39640.3	60549.29079
6000	43915	67727.58458
6500	53115	83373.7217
7000	54327.1667	85453.2985
7500	61515.2	97862.41134
8000	61932.2333	98586.22397
8500	71196.2333	114764.863
9000	76372.0333	123881.218
9500	83442.8	136416.4612
10000	95224.1	157491.5116

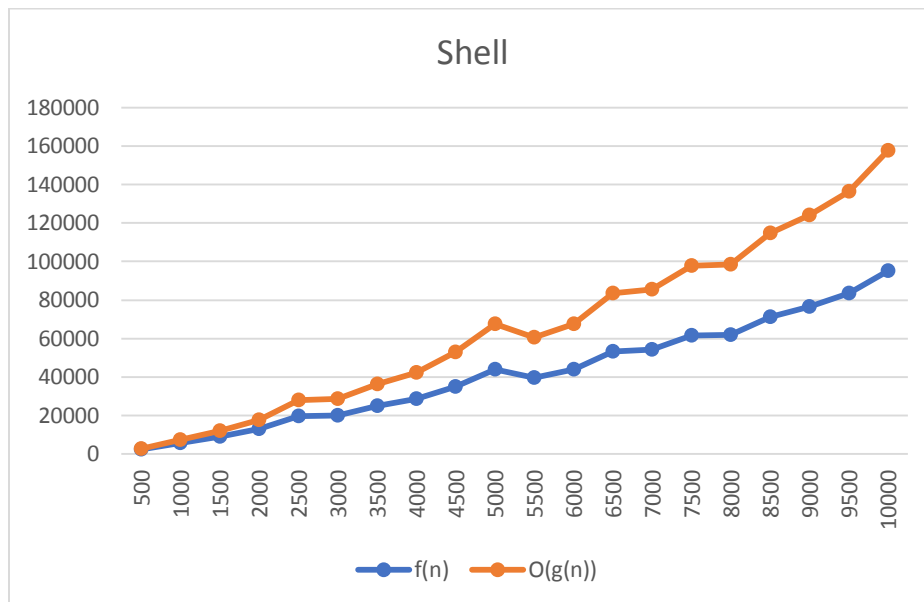


Fig. 33 Comparación de pasos obtenidos para el método de Shell y una función $O(n \log n)$

Tabla 12 Valores graficados método Comb

Comb		
n	f(n)	O(g(n))
500	1418.73333	2228.20321
1000	2994.66667	5187.44233

1500	4463.13333	8116.54796
2000	6066.23333	11434.7758
2500	7598.56667	14693.5515
3000	9116.63333	17988.4196
3500	10644.6	21360.254
4000	12248.6	24951.0031
4500	13685.2333	28205.9482
5000	15232	31746.8796
5500	16769.6333	35300.6571
6000	18336.8	38954.1137
6500	19849.3	42507.6745
7000	21330.1	46010.9554
7500	22870.3667	49678.5253
8000	24629.3667	53894.3198
8500	26101.8667	57444.4611
9000	27413.2667	60621.3689
9500	29277.7667	65161.4081
10000	30534.3333	68235.7375

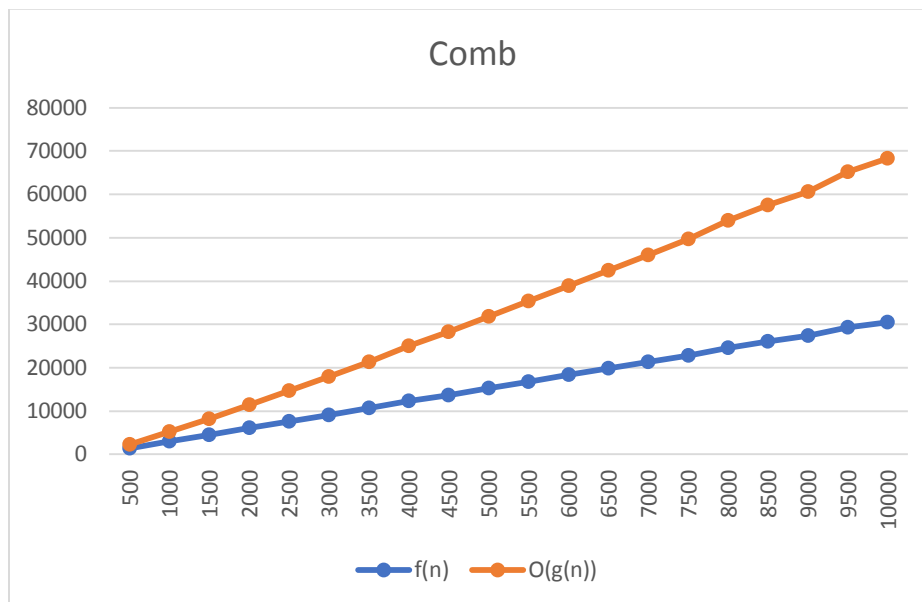


Fig. 34 Comparación de pasos obtenidos para el método de Comb y una función $O(n^2)$

Tabla 13 Valores graficados método Merge

Merge		
n	f(n)	O(g(n))
500	1895.8	2064.25909
1000	4285.63333	5170.74206
1500	6799.96667	8657.24969
2000	9561.03333	12642.5074
2500	12205.9333	16569.9146
3000	15085.2667	20939.6363
3500	17958.4667	25379.5813
4000	21106.4667	30320.2783
4500	23901.2	34763.8065
5000	26886.6667	39562.6634
5500	29894.3	44445.5981
6000	33136.0667	49757.4979
6500	36188.8667	54801.7356
7000	39377.9333	60110.8081
7500	42655.4333	65605.9414
8000	46171.7333	71541.8185
8500	49308.2333	76869.2722
9000	52282.1	81947.1176
9500	55396.7667	87291.5335
10000	58548.9	92725.9682

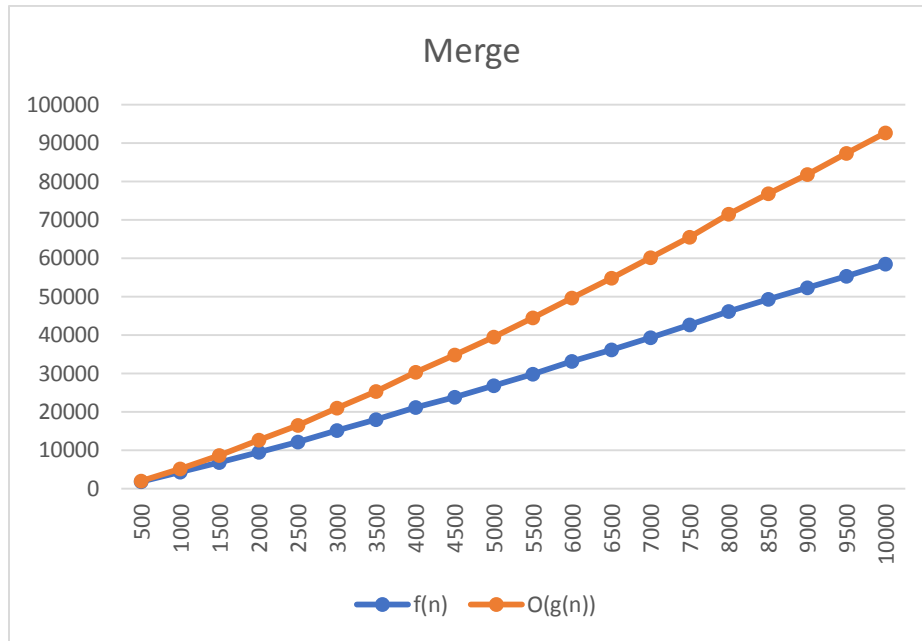


Fig. 35 Comparación de pasos obtenidos para el método de Merge y una función $O(n \log n)$

Tabla 14 Valores graficados método Quicksort

Quicksort		
n	f(n)	O(g(n))
500	2334.63333	2612.21663
1000	5002.7	6147.563505
1500	7530.8	9698.606611
2000	10135.1333	13486.90042
2500	12723.7667	17349.15893
3000	15211.9667	21133.86217
3500	17875.9333	25251.06247
4000	20495.6667	29356.00751
4500	23256.1333	33733.77403
5000	25689.8	37632.75177
5500	28138.9333	41590.13142
6000	30876.9333	46050.60514
6500	33555.0333	50447.4477
7000	35791.4667	54142.92574

7500	38749.8667	59062.17371
8000	41175.5667	63120.08856
8500	43452.8333	66948.48699
9000	46189.6	71572.0805
9500	47826.5333	74348.85027
10000	51993.2667	81452.84487

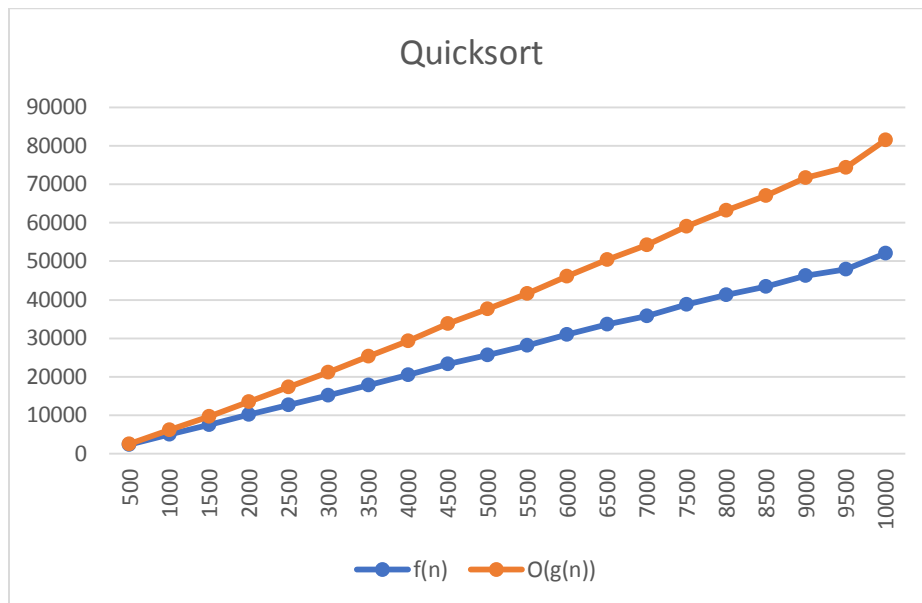


Fig. 36 Comparación de pasos obtenidos para el método de Quicksort y una función $O(n \log n)$

Conclusiones

En este trabajo se presenta el estudio sobre complejidades de diferentes algoritmos de ordenamiento de números. Burbuja, Stoooge, Quicksort, Mergesort, Shell y Comb, en donde las complejidades obtenidas de nuestro análisis son $\Theta(n^2)$ y $\Theta(n^{\log_{1.5} 3})$, para los métodos de la Burbuja y Stoooge, respectivamente, y $\Theta(n \log n)$ para Quicksort, Mergesort, Sheel y Comb.

Se realizaron varias corridas con diferentes tamaños de arreglos y se registraron las cantidades de intercambios y de comparaciones que se hace en cada algoritmo para posteriormente graficarlo y compararlo con las funciones de complejidad que se obtuvieron en la etapa de análisis.

Se puede observar en las gráficas hechas que los resultados obtenidos al correr los algoritmos son muy cercanos a las funciones de complejidad que se obtuvieron en el análisis de estos. Esto es, que las funciones, con sus respectivos ajustes de constantes y/o parámetros, pueden ajustarse a los valores obtenidos de las corridas.

Como se mencionó anteriormente, los algoritmos se implementaron y corrieron en diferentes lenguajes de programación y hardware. De acuerdo a los resultados obtenidos, se puede ver que ni el lenguaje de programación ni el hardware impactan en el desempeño de los algoritmos.

De esta forma, se concluye que el análisis de complejidad realizado a cada algoritmo es correcto ya que, considerando los resultados obtenidos de las corridas, se ajustan a los modelos matemáticos de complejidad obtenidos. Pero también se puede concluir que ni el lenguaje de programación ni el hardware repercuten en el desempeño de los algoritmos, al menos no de forma significativa.

Referencias

1. Arroyo, Carlos. (2009). *Programación en Java*. Edición: Claudio Peña, Miguel Lederkremer. USERS Ebooks- LPCU287. <https://www.studocu.com/es-mx/document/instituto-tecnologico-de-veracruz/redes-de-computadoras/programacion-en-java-volumen-1-carlos-arroyo-diaz-z-lib/35385265>
2. Ayala, Joel. Aguilar, Irene. García, Farid y Gómez, Hipólito. (2019). *Introducción al análisis de algoritmos*. (Primera Edición). Centro de Estudios e Investigaciones para el Desarrollo Docente. México.
3. Baudino, Geremias. (2023). *Métodos de Ordenamiento*. Recuperado el 8 de agosto de 2023, de README. <https://github.com/gbaudino/MetodosDeOrdenamiento>
4. Cairó, Osvaldo. Guardati, Silvia. (2006). *Estructuras de Datos*. (Tercera Edición). Mc. Graw-Hill. <https://eduardmandov.files.wordpress.com/2017/05/datastructures-estructuras-de-datos-osvaldo-cairo.pdf>
5. Cordero, Francisco. (3 de junio de 2014). *Análisis de Algoritmo*. Recuperado el 19 de septiembre de 2023, de ShellSort. <https://franciscocordero.wordpress.com/unidad-ii-metodos-de-ordenamiento/algoritmos-de-ordenamiento/metodos-de-ordenamiento/shellsort/>
6. Cormen, Thomas. Leiserson, Charles. Rivest, Ronald y Stein, Clifford. (2009). *Introduction to algorithms*. (Third Edition) The MIT Press. ISBN 978-0-262-03384-8. London, England.
7. Fager, José. Pantoja, Libardo. Villacrés, Marisol. Páez, Luz. Ochoa, Daniel y Cuadros, Ernesto. (marzo 2014). *Estructuras de Datos*. (Primera Edición). Iniciativa Latinoamericana de Libros de Texto Abiertos. <https://www.studocu.com/bo/document/universidad-mayor-de-san-andres/programacion-i/estructuras-de-datos-cc-by-sa-3/67800950>
8. Gurin, Sebastián. (30 de noviembre de 2014). *Algoritmos de ordenación*. Recuperado el 17 de mayo de 2023, de Servicio RedIRIS. http://es.tldp.org/Tutoriales/doc-programacion-algoritmos-ordenacion/alg_orden.pdf

9. Quincho, Lehi. (2023). *Comb Sort Algoritmo de Ordenación*. Recuperado el 9 de agosto de 2023, de UDocz. <https://www.udocz.com/apuntes/35609/comb-sort-algoritmo-de-ordenacion>
10. Robledano, Ángel. (22 de julio de 2019). *Qué es C++: Características y aplicaciones*. Recuperado el 20 de septiembre de 2023, de Open Webinars. <https://openwebinars.net/blog/que-es-cpp/>
11. Skiena, Steven. (2008). *The algorithm design manual*. (Second Edition). Springer. DOI: 10.1007/978-1-84800-070-4. London.
12. Williams, J. (1964). Heapsort: Algorithm 232Comm. ACM 7, 1964.
13. Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, N.

Apéndice

Códigos fuentes en C/C++

Código fuente de Merge, Shell, Burbuja, Stoooge, Comb utilizado en C++

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define N 10000 // Cantidad de numeros

void start(int[]);
void copy(int[],int[]);
void writeln(int[]);
void Merge(int[],int,int,int);
void Mergesort(int[],int,int);
void Shell(int[]);
void Bubble(int[]);
void Stoooge(int[],int,int);
void Combsort(int[], int);

int a[N],a1[N],a2[N],a3[N],a4[N],a5[N];
int ct,T[30][5],x;

void start(int b[N]){
    int k;

    for(k=0; k<N; k++)
        b[k]=rand() % 100;
//    b[k]=N-k;
}

void copy(int b[],int d[]){
```

```

int k;

for(k=0; k<N; k++)
    b[k]=d[k];
}

void writeln(int b[N]){
    int k;

    for(k=0; k<N; k++)
        printf("%d ",b[k]);

    printf("\n");
}

void Mergesort(int b[],int l,int h){
    int m;

    if(l<h){
        m=(l+h)/2;
        Mergesort(b,l,m);
        Mergesort(b,m+1,h);
        Merge(b,l,m,h);
    }
}

void Merge(int c[],int l,int m,int h){
    int b[N],i,j,k,p;

    p=l; i=l; j=m+1;
    while(p<=m && j<=h){
        if(c[p]<=c[j]){
            b[i]=c[p];
            p++;

```

```

    }
    else{
        b[i]=c[j];
        j++;
        ct++;
    }
    i++;
}
if(p>m)
    for(k=j; k<=h; k++){
        b[i]=c[k];
        i++;
    }
else
    for(k=p; k<=m; k++){
        b[i]=c[k];
        i++;
    }
for(k=1; k<=h; k++)
    c[k]=b[k];
}

void Shell(int b[N]){
    int i,j,k,in,t;

    in=(N-1)/2;
    while(in>=0){
        for(i=in; i<N; i++){
            j=i-in;
            while(j>=0){
                k=j+in;
                if(b[j]<=b[k])
                    j=-1;
                else{

```

```

        ct++;
        t=b[j];
        b[j]=b[k];
        b[k]=t;
    }
    j=j-in;
}
}
if(in==0)
    in=-1;
else
    in=in/2;
}
}

void Bubble(int b[N]){
    int i,j,t;

    for(i=1; i<N; i++){
        for(j=0; j<N-i; j++){
            if(b[j]>b[j+1]){
                ct++;
                t=b[j];
                b[j]=b[j+1];
                b[j+1]=t;
            }
        }
    }
}

void Stooage(int b[N],int i,int j){
    int k,t;

    if(b[i]>b[j]){

```

```

    ct++;
    t=b[i];
    b[i]=b[j];
    b[j]=t;
}
if(i+1>=j)
    return;
k=floor((j-i+1)/3);
Stooge(b,i,j-k);
Stooge(b,i+k,j);
Stooge(b,i,j-k);
}

int getNextGap(int gap)
{
    gap = (gap*10)/13;

    if (gap < 1)
        return 1;
    return gap;
}

void Combsort(int a[N], int n)
{
    int gap = n;
    bool swap = true;

    while (gap != 1 || swap == true)
    {
        gap = getNextGap(gap);
        swap = false;

```

```

        for (int i=0; i<n-gap; i++)
        {
            if (a[i] > a[i+gap])
            {
                //swap(a[i], a[i+gap]);
                int temp = a[i];
                a[i] = a[i+gap];
                a[i+gap] = temp;
                swap = true;
                ct++;
            }
        }
    }
}

```

```

int main(){
    int i,j;

    srand(time(NULL));
    for(x=0; x<30; x++){
        start(a);

        //writeln(a);
        copy(a1,a);
        copy(a2,a);
        copy(a3,a);
        copy(a4,a);
        copy(a5,a);

        ct=0;
        Mergesort(a4,0,N-1);
    }
}

```



```

T[x][0]=ct;
printf("+");
ct=0;
Shell(a3);
T[x][1]=ct;
printf("*",x+1);
ct=0;
Bubble(a2);
T[x][2]=ct;
printf("#");
ct=0;
Stooge(a1,0,N-1);
T[x][3]=ct;
printf("@");
Combsort(a5,N);
T[x][4]=ct;
printf("?");
}
printf("\n");
for(i=0; i<30; i++){
    printf("%d ",i+1);
    for(j=0; j<4; j++){
        printf("%d ",T[i][j]);
    }
    printf("\n");
}

float s1=0,s2=0,s3=0,s4=0,s5=0;
for(i=0; i<1; i++){
    s1+=T[i][0]; // Mergesort
    s2+=T[i][1]; // Shell
    s3+=T[i][2]; // Bubble
    s4+=T[i][3]; // Stooge
    s5+=T[i][4]; // Combsort
}

```

```
}
printf("\n");
printf("Merge: %f\n",s1/30);
printf("Shell: %f\n",s2/30);
printf("Burbuja: %f\n",s3/30);
printf("Stooge: %f\n",s4/30);
printf("Comb: %f\n",s5/30);

return 0;
}
```

Código Quicksort utilizado en C++

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define N 10000 // Cantidad de numeros

int particion(int[],int,int);
void quicksort(int[],int,int);
void start(int[]);
void writeln(int[]);

int a[N],ct;
```

```

int particion(int b[N],int lo,int hi){
    int i,j,pv,aux;

    pv=b[hi];
    i=lo;
    for(j=lo; j<=hi; j++){
        if(b[j]<pv){
            aux=b[i];
            b[i]=b[j];
            b[j]=aux;
            i++;
            ct++;
        }
    }
    aux=b[i];
    b[i]=b[hi];
    b[hi]=aux;
    ct++;

    return i;
}

void quicksort(int b[N],int lo,int hi){
    int p;

    if(lo<hi){
        p=particion(b,lo,hi);
        quicksort(b,lo,p-1);
        quicksort(b,p+1,hi);
    }
}

void start(int b[N]){
    int k;

```

```

    srand(time(NULL));
    for(k=0; k<N; k++)
        b[k]=rand() % 100;
}

void writeln(int b[N]){
    int k;

    for(k=0; k<N; k++){
        printf("%d ",b[k]);
    }
    printf("\n");
}

int main(){
    ct=0;

    start(a);
    writeln(a);
    quicksort(a,0,N-1);
    writeln(a);
    printf("ct=%d\n",ct);

    return 0;
}

```

Tabla de promedios de intercambio Merge, Shell, Burbuja, Stooge, Quicksort, Comb desde 500 hasta 10000 elementos.

Ordenamiento	Merge	Shell	Burbuja	Stooge	Quicksort	Comb
500	63	86	2054	1400	2335	1506
1000	143	198	8254	5688	5013	5913
1500	227	309	18541	12085	7550	12208
2000	319	435	32820	18452	10004	18918
2500	407	658	51454	25291	12758	25877
3000	503	673	74137	38572	15451	39276
3500	599	834	101209	51285	19048	52110
4000	703	949	131655	60663	20438	61592
4500	797	1138	166749	80831	22992	81895
5000	896	1448	206154	106730	25481	107911
5500	997	1311	249878	102454	27287	103038
6000	1104	1441	297277	138668	31410	127777
6500	1566	1735	349327	204417	34398	204242
7000	1317	1794	404981	194067	37162	194539
7500	1421	2059	457524	226693	40878	227201
8000	1538	2039	529409	235843	42385	236061
8500	1642	2342	573560	233090	44190	233952
9000	1741	2625	669348	269960	46835	270879
9500	1845	2792	746362	320066	50828	321650
10000	1955	3192	821041	354868	52555	355602

Tabla 15 Tabla de los promedios de intercambios en 30 corridas con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Merge, Shell, Burbuja, Stooge, Quicksort y Comb.

Tabla de promedios de comparaciones Merge, Shell, Burbuja, Stooage, Quicksort, Comb desde 500 hasta 10000 elementos.

Ordenamiento	Merge	Shell	Burbuja	Stooage	Quicksort	Comb
500	128	206	4158	717445	191	295
1000	128	204	4159	717445	181	313
1500	464	770	37475	6457008	849	1085
2000	647	1078	66633	19371026	1370	1579
2500	835	1485	104125	58113076	1863	1974
3000	1028	1717	149950	58113076	2512	2668
3500	1224	2013	204108	58113076	3427	3229
4000	1427	2408	266600	31173642	4227	3424
4500	1631	2826	337425	31173642	4894	4151
5000	1839	3298	416583	31173642	5810	4446
5500	2049	3555	504075	31173642	7080	4890
6000	2258	3679	599900	31173642	7877	5735
6500	2468	4121	704058	31173642	9213	6862
7000	2685	4466	816550	31173642	10591	6457
7500	2898	4765	937375	31173642	12069	7418
8000	3117	5178	1066533	31173642	13504	8179
8500	3334	6016	1204025	31173642	14942	8407
9000	3562	6251	1349850	31173642	16521	8901
9500	3784	6546	1504008	31173642	18206	9079
10000	4007	7281	1666500	31173642	19809	10224

Tabla 16 de los promedios de comparaciones en 30 corridas con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Merge, Shell, Burbuja, Stooage, Quicksort y Comb.

Códigos fuentes en Java

```
public class Ordenamiento {

    private int[] A;
    private int ct;

    public Ordenamiento(int n) {
        ct = 0;
        A = new int[n];
        for (int k = 0; k < n; k++) {
            A[k] = (int) (100 * Math.random());
        }
    }

    public void writeln(int[] b) {
        int k;

        for (k = 0; k < b.length; k++) {
            System.out.print(b[k] + " ");
        }

        System.out.println();
    }

    public int[] getA() {
        return A;
    }

    public int getC() {
        return ct;
    }
}
```

```

public void setC() {
    ct = 0;
}

public int[] bubble(int[] b) {
    int aux;

    for (int i = 1; i < b.length; i++) {
        for (int j = 0; j < b.length - i; j++) {
            if (b[j] > b[j + 1]) {
                aux = b[j];
                b[j] = b[j + 1];
                b[j + 1] = aux;
                ct++;
            }
        }
    }
    return b;
}

public void stooge(int[] b, int i, int j) {
    int k, aux;

    if (b[i] > b[j]) {
        aux = b[i];
        b[i] = b[j];
        b[j] = aux;
        ct++;
    }
    if (i + 1 >= j) {
        return;
    }
    k = (int) ((j - i + 1) / 3);

```



```

    stooge(b, i, j - k);
    stooge(b, i + k, j);
    stooge(b, i, j - k);
}

public void mergesort(int[] b, int l, int h) {
    int m;

    if (l < h) {
        m = (l + h) / 2;
        mergesort(b, l, m);
        mergesort(b, m + 1, h);
        merge(b, l, m, h);
    }
}

protected void merge(int[] c, int l, int m, int h) {
    int p, i, j;
    int[] b = new int[c.length];

    p = l;
    i = l;
    j = m + 1;
    while (p <= m && j <= h) {
        if (c[p] <= c[j]) {
            b[i] = c[p];
            p++;
        } else {
            b[i] = c[j];
            j++;
            ct++;
        }
        i++;
    }
}

```

```

    }
    if (p > m) {
        for (int k = j; k <= h; k++) {
            b[i] = c[k];
            i++;
        }
    } else {
        for (int k = p; k <= m; k++) {
            b[i] = c[k];
            i++;
        }
    }
    for (int k = 1; k <= h; k++) {
        c[k] = b[k];
    }
}

public int[] shell(int[] b) {
    int in, aux, j, k;

    in = (b.length - 1) / 2;
    while (in >= 0) {
        for (int i = in; i < b.length; i++) {
            j = i - in;
            while (j >= 0) {
                k = j + in;
                if (b[j] <= b[k]) {
                    j = -1;
                } else {
                    aux = b[j];
                    b[j] = b[k];
                    b[k] = aux;
                    ct++;
                }
            }
        }
        in = in / 2;
    }
}

```

```

        }
        j = j - in;
    }
}
if (in == 0) {
    in = -1;
} else {
    in = in / 2;
}
}
return b;
}

protected int partition(int[] b, int lo, int hi) {
    int i, pv, aux;

    pv = b[hi];
    i = lo;
    for (int j = lo; j <= hi; j++) {
        if (b[j] < pv) {
            aux = b[i];
            b[i] = b[j];
            b[j] = aux;
            i++;
            ct++;
        }
    }
    aux = b[i];
    b[i] = b[hi];
    b[hi] = aux;
    ct++;

    return i;
}

```

```

}

public void quicksort(int[] b, int lo, int hi) {
    int p;

    if (lo < hi) {
        p = partition(b, lo, hi);
        quicksort(b, lo, p - 1);
        quicksort(b, p + 1, hi);
    }
}

public int[] cocktail(int[] b) {
    int aux;
    boolean sp;

    do {
        sp = false;
        for (int i = 0; i <= b.length - 2; i++) {
            if (b[i] > b[i + 1]) {
                aux = b[i];
                b[i] = b[i + 1];
                b[i + 1] = aux;
                sp = true;
                ct++;
            }
        }
        if (!sp) {
            break;
        }
        sp = false;
        for (int i = b.length - 2; i >= 0; i--) {
            if (b[i] > b[i + 1]) {

```

```

        aux = b[i];
        b[i] = b[i + 1];
        b[i + 1] = aux;
        sp = true;
        ct++;
    }
}
} while (sp);

return b;
}

public int[] comb(int[] b) {
    int i, aux, gap = b.length;
    double shrink = 1.3;
    boolean sorted = false;

    while (sorted == false) {
        gap = (int) (gap / shrink);
        if (gap <= 1) {
            gap = 1;
            sorted = true;
        }
        i = 0;
        while (i + gap < b.length) {
            if (b[i] > b[i + gap]) {
                aux = b[i];
                b[i] = b[i + gap];
                b[i + gap] = aux;
                sorted = false;
                ct++;
            }
            i++;
        }
    }
}

```

```

        }
    }
    return b;
}

public int[] gnome(int[] b) {
    int aux, pos = 0;

    while (pos < b.length) {
        if (pos == 0 || b[pos] >= b[pos - 1]) {
            pos++;
        } else {
            aux = b[pos];
            b[pos] = b[pos - 1];
            b[pos - 1] = aux;
            pos--;
            ct++;
        }
    }
    return b;
}

/**
 * @param args the command line arguments
 */
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // TODO code application logic here

    int N =
        500; // Cantidad de numeros
}

```

```

long t1, t2;
int[] v = new int[N], w = new int[N];
double[] rt = new double[8];
double[] r = new double[8];
int[][] T = new int[30][8];
long[][] Tt = new long[30][8];
String[] nm = {"Bu"

        + "rbuja", "Stooge", "Merge",
        "Shell", "Quicksort", "Cocktail",
        "Comb", "Gnome"};

for (int i = 0; i < 30; i++) {
    Ordenamiento t = new Ordenamiento(N);
    v = t.getA();

    // Burbuja
    for (int k = 0; k < N; k++) {
        w[k] = v[k];
    }
    t1 = System.currentTimeMillis();
    w = t.bubble(w);
    t2 = System.currentTimeMillis();
    Tt[i][0] = t2 - t1;
    T[i][0] = t.getC();
    System.out.print("!");

    // Stooge
    t.setC();
    for (int k = 0; k < N; k++) {
        w[k] = v[k];
    }
    t1 = System.currentTimeMillis();

```

```

t.stooge(w, 0, N - 1);
t2 = System.currentTimeMillis();
Tt[i][1] = t2 - t1;
T[i][1] = t.getC();
System.out.print("@");

// Merge
t.setC();
for (int k = 0; k < N; k++) {
    w[k] = v[k];
}
t1 = System.currentTimeMillis();
t.mergesort(w, 0, N - 1);
t2 = System.currentTimeMillis();
Tt[i][2] = t2 - t1;
T[i][2] = t.getC();
System.out.print("#");

// Shell
t.setC();
for (int k = 0; k < N; k++) {
    w[k] = v[k];
}
t1 = System.currentTimeMillis();
w = t.shell(w);
t2 = System.currentTimeMillis();
Tt[i][3] = t2 - t1;
T[i][3] = t.getC();
System.out.print("$");

// Quicksort
t.setC();
for (int k = 0; k < N; k++) {

```



```

        w[k] = v[k];
    }
    t1 = System.currentTimeMillis();
    t.quickSort(w, 0, N - 1);
    t2 = System.currentTimeMillis();
    Tt[i][4] = t2 - t1;
    T[i][4] = t.getC();
    System.out.print("%");

    /**
     * // Cocktail
     * t.setC();
     * for (int k = 0; k < N; k++) {
     *     w[k] = v[k];
     * }
     * t1 = System.currentTimeMillis();
     * w = t.cocktail(w);
     * t2 = System.currentTimeMillis();
     * Tt[i][5] = t2 - t1;
     * T[i][5] = t.getC();
     * System.out.print("&");
    */

    // Comb
    t.setC();
    for (int k = 0; k < N; k++) {
        w[k] = v[k];
    }
    t1 = System.currentTimeMillis();
    w = t.comb(w);
    t2 = System.currentTimeMillis();
    Tt[i][6] = t2 - t1;
    T[i][6] = t.getC();
    System.out.print("*");

```

```

/**
    // Gnome
    t.setC();
    for (int k = 0; k < N; k++) {
        w[k] = v[k];
    }
    t1 = System.currentTimeMillis();
    w = t.gnome(w);
    t2 = System.currentTimeMillis();
    Tt[i][7] = t2 - t1;
    T[i][7] = t.getC();
    System.out.print("+");
    */
}
System.out.println();
for (int j = 0; j < 8; j++) {
    for (int i = 0; i < 30; i++) {
        r[j] += T[i][j];
        rt[j] += (Tt[i][j] * 1.0);
    }
    r[j] /= 30;
    rt[j] /= 30;
    System.out.println(nm[j] + ":" + (int) r[j] + " tiempo:" +
rt[j] * 1000);
}
System.out.println();
}
}

```

Tabla de los promedios de intercambios en 30 ejecuciones con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Burbuja, Shell, Merge, Quicksort, Stooze y Comb.

Promedios						
Ordenamiento	Burbuja	Shell	Merge	Quicksort	Stooge	Comb
500	61688.8333	2586.9	1895.8	2334.63333	41880.0333	1418.73333
1000	247072.267	5848.9	4285.63333	5002.7	170092.267	2994.66667
1500	556651.433	9102.4	6799.96667	7530.8	355959.4	4463.13333
2000	989731.4	13023.0667	9561.03333	10135.1333	555546.067	6066.23333
2500	1546600.4	19582.4	12205.9333	12723.7667	760252.2	7598.56667
3000	2227128.23	20114.3667	15085.2667	15211.9667	1158409.3	9116.63333
3500	3031419.17	24898.4333	17958.4667	17875.9333	1536361.67	10644.6
4000	3957988.8	28626.3667	21106.4667	20495.6667	1820936.77	12248.6
4500	5009800.93	35008.4333	23901.2	23256.1333	2427460.07	13685.2333
5000	6183667.57	43885.5667	26886.6667	25689.8	3201809.4	15232
5500	7486112.67	39640.3	29894.3	28138.9333	3069047.1	16769.6333
6000	8913395.63	43915	33136.0667	30876.9333	3826481.73	18336.8
6500	10456461.8	53115	36188.8667	33555.0333	4721633.53	19849.3
7000	12124521.2	54327.1667	39377.9333	35791.4667	5806278.03	21330.1
7500	13921196.4	61515.2	42655.4333	38749.8667	6793334.73	22870.3667
8000	15839388.7	61932.2333	46171.7333	41175.5667	7027812.67	24629.3667
8500	17875557.8	71196.2333	49308.2333	43452.8333	7022994.47	26101.8667
9000	20032716.2	76372.0333	52282.1	46189.6	8077907.63	27413.2667
9500	22314249.6	83442.8	55396.7667	47826.5333	9568990.53	29277.7667
10000	24750128	95224.1	58548.9	51993.2667	10670506.9	30534.3333

Tabla 17 de los promedios de intercambios en 30 ejecuciones con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Burbuja, Shell, Merge, Quicksort, Stooge y Comb.

Tabla de promedios de comparaciones desde 500 hasta 10000 elementos.

Comparaciones						
n	Burbuja	Shell	Merge	Quicksort	Stooge	Comb
500	124750	6337	3854	5951	21523360	9137
1000	499500	14297	8699	14696	64570081	21310
1500	1124250	23435	13949	26140	193710244	33152
2000	1999000	31908	19393	40506	581130733	48113
2500	3123750	45803	25084	56470	143392200	60558
3000	4498500	51383	30878	75920	143392200	76356
3500	6123250	62061	36791	97224	143392200	93394
4000	7998000	70168	42762	121711	935209305	106323
4500	10122750	86398	48907	146835	935209305	123054
5000	12497500	101759	55151	175840	935209305	135716
5500	15122250	102631	61414	206537	935209305	148740
6000	17997000	112829	67745	242026	935209305	168853
6500	21121750	127386	74129	279074	935209305	183783
7000	24496500	133950	80565	317039	935209305	198623
7500	28121250	146967	87003	359689	935209305	217800
8000	31996000	154955	93472	403125	935209305	247252
8500	36120750	177764	100111	447858	935209305	248542
9000	40495500	187793	106831	498487	935209305	261357
9500	45120250	201246	113520	549135	935209305	276831
10000	49995000	220152	120227	602907	935209305	305736

Tabla 18 de los promedios de comparaciones en 30 ejecuciones con arreglos de tamaño desde 500 a hasta 10000 elementos, utilizando los métodos de ordenamiento Burbuja, Shell, Merge, Quicksort, Stooge y Comb.