***UT9-PD4 - Prácticos Domiciliarios***

Ejercicio 1 - ShellSort: investigación de incrementos

Realicé una investigación en fuentes académicas y en publicaciones modernas para analizar distintas secuencias de incrementos (gaps) utilizadas en el algoritmo ShellSort, ya que estos influyen directamente en su rendimiento.

**Sedgewick (1990s)**
Propuso secuencias basadas en combinaciones matemáticas de potencias de 2 y 3, logrando una complejidad teórica cercana a O(n^(4/3)). Aunque son muy buenas en teoría, en la práctica no siempre son las más rápidas. Las fórmulas que propone son:
hk = 9*4^k* - 92^k + 1
hk = 4^k - 3*2^k + 1
Genera valores como: 1, 5, 19, 41, 109, entre otros.
Según Sedgewick, estas secuencias reducen el número de comparaciones y asignaciones, son fáciles de calcular y se comportan bien en arreglos aleatorios.
Referencia: R. Sedgewick, Analysis of Shellsort and Related Algorithms, PhD Thesis, 1975.

**Ciura (2001)**
Propuso una secuencia óptima obtenida empíricamente: 1, 4, 10, 23, 57, 132, 301, 701.
Se comprobó que es una de las más eficaces para arreglos pequeños y medianos.
Para tamaños mayores, se puede extender usando hk = floor(2.25 * hk-1).
Referencia: V. Ciura, Best Increments for the Average Case of Shellsort, 2001.

**Tokuda (1992)**
Planteó una fórmula con una razón cercana a 2.25, que muestra buen rendimiento en arreglos grandes y compite incluso con Ciura en ciertas configuraciones.
Referencia: Tokuda, ShellSort gap sequence, 1992.

Resultados experimentales en Java
Implementé ShellSort con estas secuencias y obtuve estos tiempos promedio:
Ciura: 0.526 ms
Tokuda: 0.628 ms
Sedgewick: 0.470 ms
Sedgewick resultó el más rápido en mis pruebas.

***Ejercicio 2 - QuickSort: funciones de pivote e implementación***

Investigación sobre estrategias de selección de pivote que impactan directamente en el rendimiento y en el riesgo del peor caso.

Pivote aleatorio
Elegir un elemento al azar como pivote reduce la probabilidad del peor caso O(n^2).
Referencia: Hoare, Quicksort, 1962.

Mediana de tres
Toma la mediana entre el primer, medio y último elemento del arreglo, ayudando a mejorar el rendimiento en datos parcialmente ordenados.
Referencia: Sedgewick, Algorithms in C, 1998.

Mediana de cinco o más
Usada en investigaciones como MQuickSort, busca mejorar el balance de las particiones, aunque aumenta el costo computacional.
Referencia: Weiss et al., 2012.

Median-of-medians
Garantiza O(n log n) en el peor caso, pero su alto costo constante lo hace poco práctico en QuickSort clásico.
Referencia: Blum et al., 1973.

Implementaciones en lenguajes modernos
Java usa Dual-Pivot Quicksort propuesto por Yaroslavskiy, que mejora la eficiencia práctica.
C++ (STL) emplea Introsort, que inicia con QuickSort y cambia a HeapSort si la recursión se profundiza demasiado.
Python utiliza Timsort, que combina MergeSort e InsertionSort, ideal para datos parcialmente ordenados.

Resultados con mis pruebas
Probé QuickSort con diferentes estrategias de pivote y estos fueron los tiempos:
Pivote fijo: 18.40 ms
Pivote aleatorio: 13.11 ms
Mediana de tres: 9.17 ms
La mediana de tres fue la más veloz.

***Ejercicio 3 - Algoritmo para conjuntos disjuntos***

Diseñé un algoritmo para determinar si dos conjuntos A y B (de tamaños m y n) son disjuntos.

Primero se ordena el conjunto más pequeño en O(m log m). Luego, para cada elemento del otro conjunto, se hace una búsqueda binaria en el primero en O(log m).
El tiempo total es O(m log m + n log m), lo que resulta muy eficiente sobre todo cuando m es mucho menor que n.