Logotipo

Descripción generada automáticamente

Lozano Contreras Luis Enrique Lozano

ADA2 - MetOrdenamiento2

Estructura de datos

3SB

**Ejercicio 1: Ordenar una lista grande de números enteros con ShellSort**

**Descripción del ejercicio:**

Tenemos una lista grande de números enteros que necesitamos ordenar de manera eficiente.

**Explicación:**

ShellSort es una extensión del algoritmo de inserción, que reduce el número de comparaciones e intercambios haciendo que los elementos se muevan más rápidamente hacia su posición final. Es útil para listas grandes debido a su capacidad de disminuir significativamente el tiempo de ordenamiento en comparación con la inserción directa.

**Posible mejora:**

Para listas muy grandes, Quicksort o Heapsort podrían ser más eficientes debido a sus mejores comportamientos promedio y peor caso.

**Conclusiones:**

ShellSort es eficiente para listas grandes y no completamente desordenadas. Sin embargo, puede no ser el más óptimo en todos los casos en comparación con otros algoritmos como Quicksort.

**Ejercicio 2: Ordenar una lista de números enteros desordenados con Quicksort**

**Descripción del ejercicio:**

Tenemos una lista de números enteros muy desordenados que necesitamos ordenar rápidamente.

**Explicación:**

Quicksort es un algoritmo de ordenamiento muy eficiente en promedio con un tiempo de ejecución de O(n log n). Utiliza un enfoque de divide y vencerás y es particularmente efectivo para listas grandes y desordenadas.

**Posible mejora:**

Heapsort puede ser una alternativa en casos donde se requiere garantizar un rendimiento O(n log n) en el peor caso.

**Conclusiones:**

Quicksort es muy rápido y generalmente eficiente, pero su rendimiento puede deteriorarse en el peor de los casos (O(n^2)) sin una buena estrategia de elección de pivote.

**Ejercicio 3: Ordenar una lista de números enteros con un rango pequeño usando Radix Sort**

**Descripción del ejercicio:**

Tenemos una lista de números enteros no negativos con un rango pequeño que necesitamos ordenar eficientemente.

**Explicación:**

Radix Sort es eficiente para listas de números enteros no negativos con un rango pequeño. Ordena los números por cada dígito, usando un algoritmo de ordenamiento estable como Counting Sort. Es lineal respecto al número de dígitos y el tamaño de la lista, haciendo que su tiempo de ejecución sea O(nk) donde k es el número de dígitos del número más grande.

**Posible mejora:**

Para listas con un rango muy grande de valores, Quicksort o Heapsort pueden ser más eficientes.

**Conclusiones:**

Radix Sort es extremadamente eficiente para listas de números enteros con un rango pequeño. Sin embargo, su uso se limita a casos específicos debido a su dependencia del rango de los números.

**Ejercicio 4: Ordenar una lista de elementos con prioridad usando Heapsort**

**Descripción del ejercicio:**

Tenemos una lista de elementos con diferentes prioridades y necesitamos ordenar esta lista basada en las prioridades.

**Explicación:**

Heapsort utiliza una estructura de datos de montón para ordenar elementos. Es eficiente y tiene un tiempo de ejecución O(n log n) tanto en el peor como en el mejor caso, lo que lo hace ideal para listas de elementos con prioridades.

**Posible mejora:**

Quicksort podría ser más rápido en la práctica debido a menores constantes en su tiempo de ejecución promedio, pero Heapsort es preferible cuando se necesita asegurar un rendimiento consistente.

**Conclusiones:**

Heapsort es una excelente opción para listas donde se necesita una garantía de rendimiento O(n log n). Su uso de una estructura de montón asegura que el algoritmo sea eficiente incluso en el peor de los casos.

**Conclusión general**

Cada método de ordenamiento tiene sus fortalezas y debilidades. ShellSort es bueno para listas medianamente desordenadas y de tamaño moderado, Quicksort es muy rápido en promedio para listas muy desordenadas, Radix Sort es ideal para listas de números enteros con un rango pequeño, y Heapsort es consistente y eficiente para listas de elementos con prioridad. La elección del método de ordenamiento depende de las características específicas de la lista a ordenar y de los requisitos de rendimiento del problema.

**Enlace GitHub:**