Universidad Distrital Francisco José De Caldas

Algoritmos de ordenamiento

Ingeniería en Sistemas

Ciencias de la Computación I

Integrantes:

JHONATHAN DAVID DE LA TORRE GARCIA 20222020033

EDGAR ALEJANDRO MORA CHALA 20222020005

CRISTIAN SANTIAGO LOPEZ CADENA 20222020027

Bogotá Colombia

Agosto 2024

Introducción:

En el presente documento se presenta un análisis de la complejidad de tres algoritmos de ordenamiento ampliamente utilizados en la informática: MergeSort, HeapSort y QuickSort. Estos algoritmos son fundamentales en el estudio de la estructura de datos y la eficiencia algorítmica, debido a su rendimiento óptimo en diversas situaciones y su relevancia en aplicaciones prácticas.

Merge Sort

El algoritmo `MERGESORT` funciona en dos fases principales:

1. **División recursiva**: El array se divide repetidamente en subarrays hasta que cada subarray tiene un solo elemento.
2. **Mezcla (merge)**: Luego, se combinan los subarrays de manera ordenada.

Complejidad del Algoritmo:

1. **División recursiva**:
   - En cada nivel de recursión, el array se divide a la mitad. Esto genera un árbol de recursión cuya altura es logarítmica, es decir, $\log_2 n$
   - Cada nivel del árbol de recursión procesa n elementos en total (considerando todas las divisiones en ese nivel).
2. **Mezcla (merge)**:
   - El proceso de mezcla compara y combina elementos de subarrays. En cada nivel de recursión, la operación de mezcla toma tiempo proporcional a n porque todos los elementos deben ser procesados.
   - Esto se repite a lo largo de los $\log_2 n$ niveles.

Complejidades:

- **O(n log n)**: `MERGESORT` tiene una complejidad temporal en el peor caso $O(n \log n)$.
  Esto se debe a que el proceso de dividir el array toma $\log_2 n$ niveles, y en cada nivel se realizan n operaciones para la mezcla. Así que, en total, el peor caso es $O(n \log n)$.
- **Ω(n log n)**: `MERGESORT` también tiene una complejidad temporal en el mejor caso $\Omega(n \log n)$. Incluso si el array ya está ordenado, el algoritmo realiza las mismas divisiones y combinaciones. No hay un atajo que lo haga más rápido, por lo que el mejor caso también es $n \log n$.
- **Θ(n log n)**: `MERGESORT` es un algoritmo con complejidad $\Theta(n \log n)$, lo que significa que tanto el mejor como el peor caso tienen la misma complejidad temporal. Esto indica que la función de tiempo está limitada de manera asintótica tanto superior como inferiormente por $n \log n$.

DIFICULTADES:

- Merge Sort divide recursivamente el array en subarrays, y durante el proceso de fusión requiere espacio adicional proporcional al tamaño del array original. Por lo tanto, la complejidad espacial es $O(n)$, lo que implica que el espacio usado crece de manera lineal con el tamaño del arreglo.

Heap Sort

El algoritmo heap sort se basa en la estructura de datos Max-heap, es decir un árbol binario en el cual cada nodo es mayor o igual que sus hijos, la primera fase de este algoritmo consiste en construir el árbol a partir del array dado y organizarlo a fin de que el mayor elemento quede en la raíz, una vez se obtiene este elemento es almacenado al final del array y seguido de esto se vuelve a construir otro árbol en base al array pero excluyendo este último elemento y así sucesivamente hasta que todos los elementos estén ordenados.

COMPLEJIDAD:

- **O(n log n):** En el pero de los casos la función **search_Greatest** se llama sobre un nodo y potencialmente realiza comparaciones y swaps a lo largo de un camino en el árbol.
  Para un nodo en un árbol de altura h, la complejidad en el peor caso es O(h), ya que puede haber hasta h niveles en los que se realiza una comparación o swap.
  En un **heap** de n elementos, la altura máxima h es O (log n).
  Para las diversas iteraciones del ciclo for en la función **heap_sort**, se ejecuta $\frac{n}{2}$ veces (empezando desde $\frac{n}{2} - 1 \quad hasta\ 0$, y en cada iteración se llama a **search_Greatest;** Como **search_Greatest** tiene una complejidad O (log n), la complejidad total para construir el **Max-heap** es,$f(n) = O\ (n \log n)$ y se puede modelar una aproximación a la siguiente sumatoria:

$$\sum_{i=0}^{logn} \frac{n}{2^i} * O(i)$$

  En consecuencia, la función que mejor describiría la complejidad del algoritmo es:

$$f(n) = O(nlogn)$$

Quicksort

La primera fase de este algoritmo consiste en seleccionar un elemento del array al cual llamaremos pivote, luego se procederá a a reorganizar los elementos del array de tal forma que a un lado que den todos los elementos mayores que el pivote y al otro lado los menores obteniendo así dos sub arrays, finalmente este procedimiento se repetirá para cada sub-array hasta que se llegue a un sub-array de 0 o 1 elementos, lo que indicara que los elementos ya están organizados.

En la implementación del algoritmo de **Quicksort proporcionado en el archivo QuickSort.py** se realiza un análisis de los ciclos de la siguiente manera:

Para el bucle **for** , este bucle se ejecuta desde low hasta high - 1, lo que implica que se ejecuta (O(n)) veces, donde (n) es el tamaño del subarray.

Para las operaciones **dentro del bucle**, Cada iteración del bucle realiza comparaciones y, potencialmente, intercambios, ambos de los cuales son operaciones de tiempo constante (O (1)).

Por lo tanto, la complejidad de la función **partition** es (O(n)).

La función ***quickSor***t se llama recursivamente dos veces para cada partición. En el mejor caso, cada partición divide el array en dos mitades iguales, lo que lleva a una profundidad de recursión de (n log n).

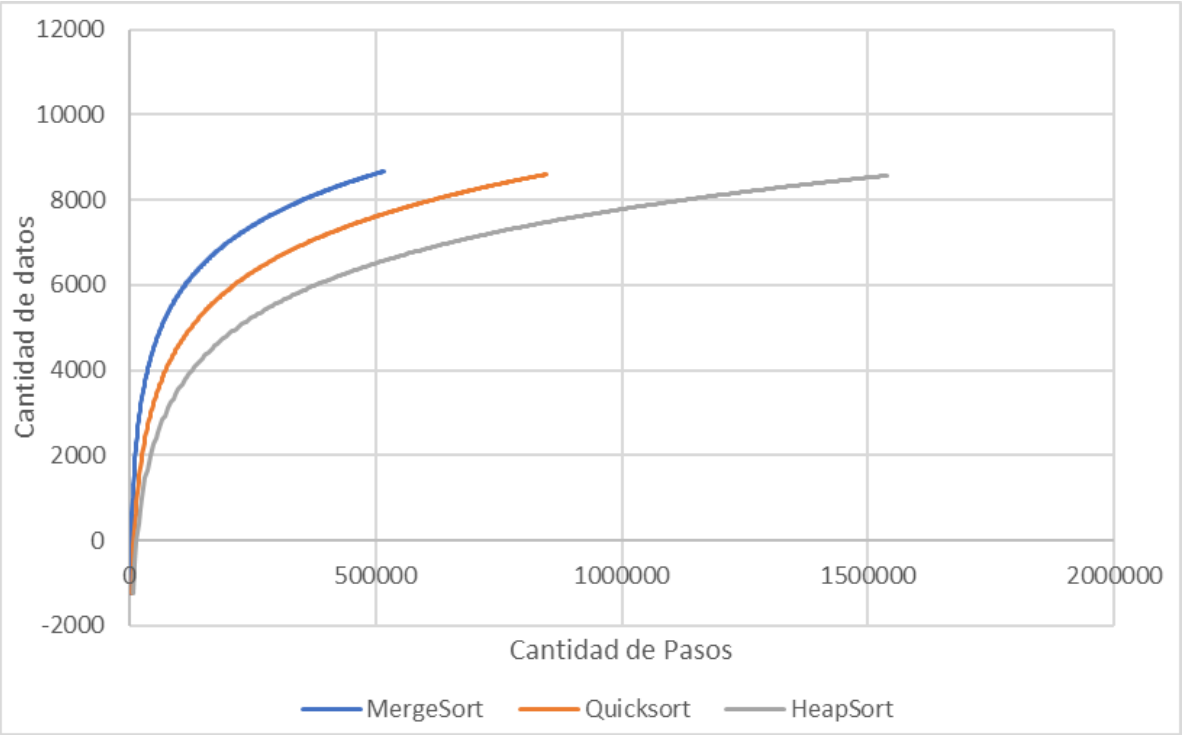
Por lo tanto, el análisis de complejidad para este algoritmo en el mejor de los casos es

$$f(n) = O(nlogn)$$

Y para el peor de los casos:

$$f(n) = O(n^2)$$

debido al número de iteraciones es



Conclusiones

Heap Sort, Merge Sort, y Quick Sort son tres algoritmos de ordenamiento que comparten una complejidad de O(n log n)o O(n^2) en la mayoría de los casos, pero difieren en cómo alcanzan esa eficiencia y en sus características prácticas:

- **Heap Sort** es consistente y eficiente en todos los casos, pero puede ser más lento en la práctica debido a las operaciones de intercambio frecuentes. No requiere memoria adicional, lo que lo hace adecuado para sistemas con limitaciones de espacio.
- **Merge Sort** siempre tiene un rendimiento predecible, con la misma complejidad en todos los casos. Sin embargo, necesita espacio adicional, lo que puede ser un inconveniente en ciertos entornos. Su estabilidad lo hace ideal cuando es importante mantener el orden relativo de los elementos.
- **Quick Sort** es generalmente el más rápido, pero su rendimiento puede degradarse si no se elige bien el pivote, llevándolo a O(n^2) en el peor caso. Aun así, con una buena estrategia de pivote, es extremadamente eficiente y es la elección preferida en la mayoría de las situaciones.