	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: MCIC. René Adrián Dávila Pérez

Asignatura: Estructura de Datos y Algoritmos I

Grupo: 19

No. de práctica(s): 12

Integrante(s): 319194359

No. de lista o brigada:

Semestre: 2023-2

Fecha de entrega: 2 de junio del 2023

Observaciones:

CALIFICACIÓN: _____

Introducción

Los algoritmos de ordenamiento son una parte fundamental de la ciencia de la computación y tienen una amplia variedad de aplicaciones en el mundo real. Existen muchos algoritmos diferentes para ordenar datos utilizando estructuras y cada uno tiene diferente complejidad o rendimiento. En este reporte se presenta una comparación gráfica entre la complejidad temporal de tres algoritmos de ordenamiento —Insertion Sort, Quick Sort y Merge Sort— utilizando el modelo RAM, con la finalidad de conocer cuál es más eficiente.

Para llevar a cabo esta comparación se utiliza un código en Python que genera para cada iteración una lista aleatoriamente desordenada pero de un cierto tamaño, que con el paso de los ciclos se va haciendo más grande. Y se almacena en una variable global cuántas veces repitió un pedazo de código dentro de la función del algoritmo de ordenamiento, es decir que contamos la parte de tiempo variable (bucles y recursividad). Los resultados se presentan en forma de gráficos para facilitar su análisis, donde el eje “x” representa el tamaño de la lista y el eje “y” la cantidad de pasos/operaciones realizadas.

La parte de tiempo constante de la complejidad de tiempo de un algoritmo al final no la contamos para obtener T(n), porque es una cantidad constante de operaciones que sólo se ejecutarán una vez, esto es: llamadas a funciones(calls), funciones aritméticas (+, -, *, //) e if’s, sin embargo en los if-else se consideran O(1) si las operaciones son similares (por ejemplo en ambas se suma). Entonces, en la complejidad temporal no importa tener una gran cantidad de operaciones que se ejecutan una vez, se aclara esto porque existen diferentes versiones de sorting algorithms.

El análisis asintótico se usa en el análisis de complejidad espacial o temporal, en esta práctica se analiza la complejidad de tiempo, ya que con la notación asintótica se puede conocer el tiempo de ejecución o running time, del algoritmo en función del tamaño de entrada. Sin embargo la notación asintótica no es exacta, sino que es una descripción de cómo se escala el algoritmo con tamaños de entrada más grandes.

Los sorting algorithm son deterministas porque encuentran soluciones exactas para entradas n pequeñas. Esto quiere decir que con una misma entrada, se obtendrá una misma salida, solo hay una solución.

Desarrollo

El código comienza importando las librerías necesarias para crear gráficos y generar números aleatorios. Luego, se definen las funciones para cada uno de los algoritmos de ordenamiento. Cada función tiene una variable global llamada `times` que lleva un registro del número de veces que el algoritmo tarda en ordenar la lista.

Después de definir las funciones para cada uno de los algoritmos de ordenamiento, el código define el tamaño de la lista a ordenar y crea tres listas vacías para almacenar los tiempos de cada algoritmo. A continuación, se utiliza un bucle `for` para generar listas aleatorias de diferentes tamaños utilizando el método `sample` de la librería `random`. Este método toma como argumentos un rango de números y un tamaño de muestra y devuelve una lista de números aleatorios del rango especificado y del tamaño especificado.

Una vez generadas las listas aleatorias, se ordenan utilizando cada uno de los algoritmos y los tiempos se agregan a las listas correspondientes para luego ser graficados.

Generación de listas aleatorias

El método `sample` se utiliza para generar listas aleatorias de números. Este método toma como argumentos un rango de números y un tamaño de muestra y devuelve una lista de números aleatorios del rango especificado y del tamaño especificado.

En el código, se utiliza la línea `random.sample(range(0 , 1000), num)` para generar una lista aleatoria de `num` números entre 0 y 1000. Esta lista se utiliza luego para comparar el rendimiento de los diferentes algoritmos de ordenamiento.

Una vez generadas las listas aleatorias, se ordenan utilizando cada uno de los algoritmos y los tiempos se agregan a las listas correspondientes para luego ser graficados.

Insertion Sort.

La función `insertionSort_graph()` implementa el algoritmo de ordenamiento Insertion Sort. Esta función toma como argumento una lista de números y la ordena.

El algoritmo comienza con un bucle for que recorre la lista desde el segundo elemento hasta el final. En cada iteración del bucle, se toma el elemento actual y se compara con los elementos anteriores en la lista. Si el elemento actual es menor que alguno de los elementos anteriores, se intercambian sus posiciones. Este proceso se repite hasta que el elemento actual esté en su posición correcta en la lista.

La variable global times se incrementa cada vez que se ejecuta un loop contando la primera vez, ya sea el bucle for o el bucle while. Esto se hace por que para obtener la complejidad temporal se requiere contar los pasos necesarios para

obtener una salida, no se cuenta la llamada (call) a una función por primera vez (como lo es hacerlo desde el código principal en esta práctica), y tampoco se cuentan las instrucciones/sentencias ya que dentro de ambos ciclos la cantidad es constante O(1).

Quick Sort.

La función `quickSort_graph()` implementa el algoritmo Quick Sort. Esta función toma como argumentos una lista de números, un índice bajo y un índice alto. La función ordena los elementos de la lista entre los índices bajo y alto utilizando el algoritmo Quick Sort.

Aplica el concepto de recurrencia al ser un algoritmo que divide el problema en subproblemas (divide y vencerás), y al final se combinan los resultados de estos últimos.

El algoritmo comienza verificando si el índice bajo es menor que el índice alto. Si es así, se llama a la función `partition()` para dividir la lista en dos sublistas: una con los elementos menores que el pivote y otra con los elementos mayores que el pivote. Luego, se llama recursivamente a la función `quickSort_graph()` para ordenar las dos sublistas.

La función `partition()` toma como argumentos una lista de números, un índice bajo y un índice alto. La función elige el último elemento de la lista como pivote y reordena los elementos de la lista de tal manera que todos los elementos menores que el pivote queden a su izquierda y todos los elementos mayores queden a su derecha. La función devuelve el índice del pivote después de reordenar los elementos.

La variable global `times` se incrementa en dos ocasiones: en las llamadas recursivas de esta misma función quickSort_graph() y cada vez que se ejecuta el bucle for en la función partition() para llevar un registro del número de veces que el algoritmo tarda en ordenar la lista, esto último es debido a que dentro de este bucle puede o no realizar alguna operación, dependiendo si la condición de if se cumple, es por ello que solo se incrementa times si se cumple y por tanto al estar en bucle repite más de una vez una serie de instrucciones, ya que esta es la finalidad de times, contar `pasos` que se repiten.

Merge Sort.

La función `mergeSort_graph()` implementa el algoritmo Merge Sort y es un ejemplo de vivide y vecerás. Esta función toma como argumento una lista de números y la ordena utilizando el algoritmo Merge Sort.

El algoritmo comienza verificando si la longitud de la lista es mayor que 1. Si es así, se divide la lista en dos mitades utilizando la sintaxis `lista[:mid]` para

obtener la primera mitad y `lista[mid:]` para obtener la segunda mitad. Luego, se llama recursivamente a la función `mergeSort_graph()` para ordenar cada mitad por separado. Finalmente, se utiliza un bucle `while` para fusionar las dos mitades ordenadas en una sola lista ordenada.

En esta función, a pesar de que haya un if-else dentro de uno de los tres loops, las expresiones dentro del ciclo son constantes O(1), ya que tanto if como else tienen dos sentencias, una de asignación y la otra aritmética, así que sin importar a cual entre se ejecutan la misma cantidad de instrucciones. La variable global times incrementa en dos ocasiones, en cada bucle y en cada llamada recursiva.

Código principal.

En la gráfica “x” representa el tamaño de la entrada/input, cada algoritmo recibe un tamaño de entrada cada vez más grande, es por ello que todos utilizan la misma variable eje_x porque los valores que toman para “x” son los mismos.

El cada uno de los valores de “y” representa cuantos pasos realizó el algoritmo para cada entrada. A diferencia de “x”, los valores de “y” aumentan de diferente manera para cada algoritmo, es por ello que cada uno tiene una variable global del eje “y”.

Consta de un bucle for, que se repite 100 veces, donde num toma valores en cada iteración desde 1 hasta 100, tocando ambos. Esto quiere decir que a lo largo del programa se ordenan 100 listas con cada vez más elementos, hasta llegar a longitud 100.

Cada algoritmo de ordenamiento ordena la misma lista desordenada aleatoriamente, es decir cada uno tiene el mismo caso promedio en cada ocasión. La lista creada por ciclo se copia en las listas respectivas de cada sorting algorithm, la primera usando .sample() y las otras dos listas mediante .copy(). Esto es así para evitar que las funciones de los algoritmos de ordenamiento modificarán la lista desordenada, ya que en python, cuando una variable tipo lista es asignada (=) a otra variable tipo lista, si se modifica cualquiera de las dos, se modifica la otra. Asimismo, una variable como argumento a una función, es como enviar la “variable original” en C por medio de su dirección para que se modifique su valor en lugar de modificar el valor de una copia. Esto es debido a que en este lenguaje las variables son referencias que apuntan a la memoria.

En cada iteración se imprimen la lista desordenada actual, seguido de los mismos elementos de está pero ordenada por cada uno de los sorting algorithm en listas distintas.

Interfaz gráfica.

Para crear gráficos y visualizar los resultados de la comparación entre los diferentes algoritmos de ordenamiento, el código utiliza la librería `matplotlib`. Esta librería proporciona una interfaz similar a MATLAB para crear visualizaciones estáticas, animadas e interactivas en 2D.

En el código, se importa el módulo `pyplot` de la librería `matplotlib` utilizando la línea `import matplotlib.pyplot as plt`. Este módulo proporciona funciones para crear gráficos y personalizar su estilo.

Después de generar las listas aleatorias y ordenarlas utilizando cada uno de los algoritmos, el código utiliza las funciones `plot` y `legend` del módulo `pyplot` para crear un gráfico que muestra el rendimiento de cada algoritmo en función del tamaño de la lista. La función `plot` toma como argumentos los datos a graficar y opcionalmente puede recibir argumentos adicionales para personalizar el estilo del gráfico. La función `legend` se utiliza para agregar una leyenda al gráfico.

Finalmente, el código utiliza la función `show` del módulo `pyplot` para mostrar un gráfico cuadriculado. El color magenta representa insertion sort, el color amarillo merge sort y el color cyan quick sort.

Ejemplos de Ejecución

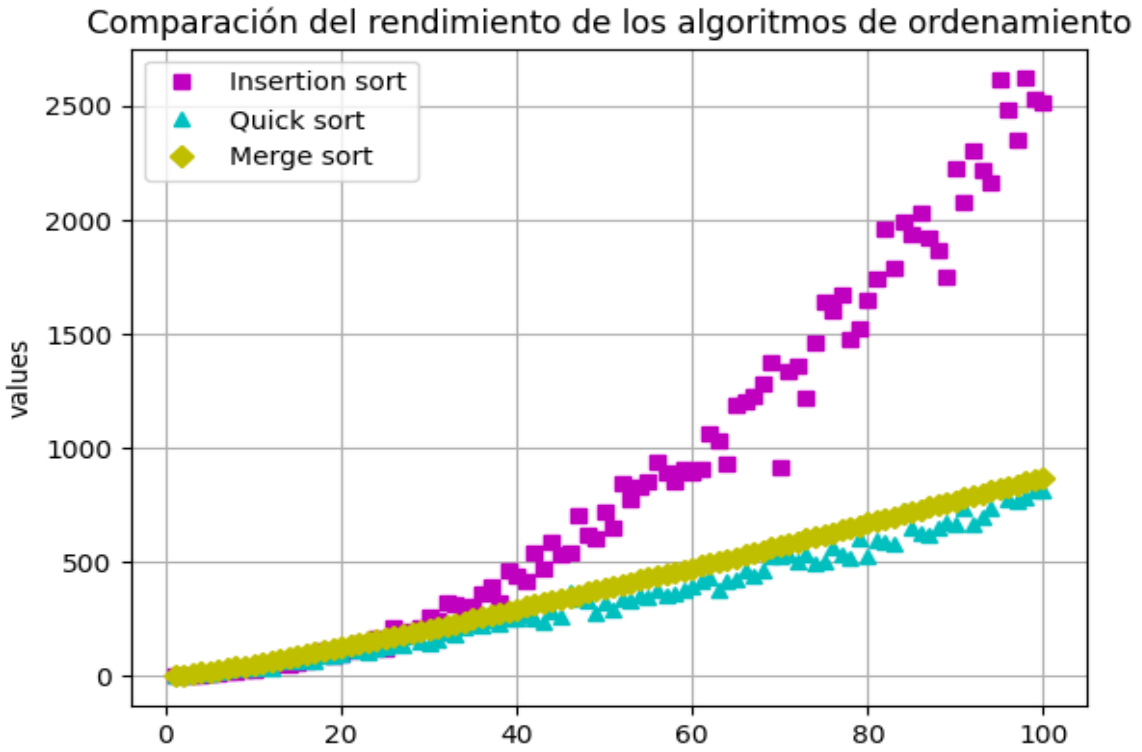
No se muestra toda la salida, solo se muestra la última parte, la lista más grande que es de 100 elementos, desordenada y ordenada por cada uno de los tres algoritmos de ordenamiento.

Lista ordenada usando Insertion Sort: [19, 31, 39, 45, 46, 55, 60, 61, 62, 80, 81, 96, 122, 133, 136, 148, 156, 170, 207, 226, 233, 247, 278, 279, 303, 304, 354, 355, 362, 375, 378, 380, 385, 404, 420, 421, 423, 438, 447, 467, 469, 481, 488, 496, 505, 514, 526, 528, 556, 560, 561, 571, 586, 593, 603, 608, 609, 615, 618, 620, 641, 648, 672, 673, 676, 698, 707, 709, 711, 714, 719, 743, 755, 765, 783, 785, 787, 799, 800, 821, 828, 846, 858, 869, 883, 889, 892, 893, 914, 935, 940, 948, 960, 963, 971, 975, 981, 985, 988, 996]

Lista ordenada usando Quick Sort: [19, 31, 39, 45, 46, 55, 60, 61, 62, 80, 81, 96, 122, 133, 136, 148, 156, 170, 207, 226, 233, 247, 278, 279, 303, 304, 354, 355, 362, 375, 378, 380, 385, 404, 420, 421, 423, 438, 447, 467, 469, 481, 488, 496, 505, 514, 526, 528, 556, 560, 561, 571, 586, 593, 603, 608, 609, 615, 618, 620, 641, 648, 672, 673, 676, 698, 707, 709, 711, 714, 719, 743, 755, 765, 783, 785, 787, 799, 800, 821, 828, 846, 858, 869, 883, 889, 892, 893, 914, 935, 940, 948, 960, 963, 971, 975, 981, 985, 988, 996]

Lista ordenada usando Merge Sort: [19, 31, 39, 45, 46, 55, 60, 61, 62, 80, 81, 96, 122, 133, 136, 148, 156, 170, 207, 226, 233, 247, 278, 279, 303, 304, 354, 355, 362, 375, 378, 380, 385, 404, 420, 421, 423, 438, 447, 467, 469, 481, 488, 496, 505, 514, 526, 528, 556, 560, 561, 571, 586, 593, 603, 608, 609, 615, 618, 620, 641, 648, 672, 673, 676, 698, 707, 709,

711, 714, 719, 743, 755, 765, 783, 785, 787, 799, 800, 821, 828, 846, 858, 869, 883, 889, 892, 893, 914, 935, 940, 948, 960, 963, 971, 975, 981, 985, 988, 996]



Conclusión

Este código permite comparar el rendimiento de tres algoritmos de ordenamiento: Insertion Sort, Quick Sort y Merge Sort. Cada uno tiene sus ventajas y desventajas dependiendo del tamaño y contenido de las listas a ordenar.

Insertion Sort es fácil de entender e implementar pero no es muy eficiente, es decir ocupa mucho el recurso tiempo para listas grandes en casos promedio debido a que su complejidad temporal cuadrática es$T(n) = \Theta(n^2)$.

Quick Sort es uno de los algoritmos más rápidos pero su rendimiento puede disminuir en casos específicos como cuando la lista ya está ordenada o casi ordenada. Cuando es el caso promedio $T(n) = \Theta(n\log n)$.

Merge Sort tiene una complejidad temporal logarítmica tanto en casos promedio como peores $T(n) = \Theta(n\log n)$, pero requiere espacio adicional para almacenar las dos mitades mientras se fusionan.

De esta manera aprendimos que en $\Theta(n\log n)$, merge y quick sort son $\log(n)$ porque son recursivos y son n porque esta operación se repite la misma cantidad veces como elementos hay en la lista.

Por lo que no existe un único algoritmo “mejor” para todas las situaciones sino que depende del caso específico cuál será más adecuado

El más eficiente es el que utiliza menos recursos de tiempo cuando una lista está desordenada sería quicksort, por lo tanto es el más rápido. Pero no podemos hacer conclusiones completamente sobre rendimiento, porque no se analizó el espacio en memoria (complejidad de espacio).

Además aprendimos que los sorting algorithms son deterministas y la clase complejidad computacional del problema que resuelve es P, porque cada lista desordenada tiene una única solución, y porque el running time/tiempo de ejecución se calcula con las variables implicadas usando un polinomio.

Repositorio

https://github.com/isaacmsd9/EDA-I-2023-2/blob/main/Semana-12/P12/p12.ipynb

Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

[2] S. Li, “Chapter 4: Divide and Conquer,” in Algorithms, University of Science and Technology of China. [Online]. Available: http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap04.htm

[3] “How to analyse complexity of recurrence relation?” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/

[4] “P versus NP problem,” Encyclopædia Britannica. [Online]. Available: https://www.britannica.com/science/P-versus-NP-problem#ref1119253