Unidad Profesional Interdisciplinaria de Ingeniería Campus Zacatecas Instituto Politécnico Nacional

UNIDAD 1

REPORTE

Análisis de Complejidad Computacional

Nombre	Boleta
Enid Aimee Perez Robles	2023670020

Grupo:
3CM2

Docente:

M. en C. Erika Sánchez-Femat

Introducción

La mayoría de las veces nuestros problemas consisten en encontrar estructuras que nos faciliten la resolución de otros problemas. El objetivo principal de esta práctica es que los estudiantes adquieran una comprensión sólida y profunda de los conceptos clave relacionados con la complejidad computacional de los algoritmos. Los tres casos principales a analizar, mejor caso, peor caso y caso promedio, permiten una evaluación completa del rendimiento de un algoritmo bajo diferentes circunstancias

Desarrollo

Se realizará en python los siguientes métodos de ordenamiento:
—Burbuja
—Burbuja Optimizada
En la cual cada método de ordenamiento deberá desarrollarse en una función de programación diferente, con el objetivo de que el usuario, mediante un menú, pueda seleccionar cuál método de ordenamiento desea utilizar. Una vez que el usuario seleccione el método de ordenamiento, el programa pedirá al usuario ingresar el tamaño de la lista y los elementos de la misma, los cuales serán la entrada de los algoritmos.

Funcionamiento del Código:

El usuario inicia el programa y proporciona el tamaño de la lista que desea ordenar. Luego, el código genera una lista ordenada de 1 a 'n'. Despues,ejecuta el algoritmo de ordenamiento burbuja y el algoritmo de ordenamiento burbuja optimizada en copias de la lista ordenada para medir el tiempo de ejecución en el mejor caso.
Finalmente, muestra los tiempos de ejecución en la pantalla,el resultado del código muestra los tiempos de ejecución en segundos para ambos algoritmos en el mejor caso, junto con la lista ordenada. Se proporciona una copia de la lista ordenada para evitar afectar el orden original de la lista.

Análisis de casos

Calcular **el Mejor Caso**:
El mejor caso para el algoritmo de ordenamiento burbuja y su variante optimizada ocurre cuando la lista de entrada ya está ordenada. En este caso, el algoritmo no necesita hacer ningún intercambio durante su ejecución. El mejor caso tiene una complejidad de tiempo de O(n) para ambos algoritmos, donde 'n' es el número de elementos en la lista.

Ejemplo : **Lista ya ordenada de manera ascendente**

El mejor caso ocurre cuando la lista de entrada ya está ordenada de manera ascendente. En este caso, el tiempo de ejecución será bajo. Puedes usar listas ya ordenadas como entrada y observar los tiempos de ejecución.
arr = list(range(1, n+1)) Lista ya ordenada de 1 a n

Ejemplo Búsqueda Binaria En una búsqueda binaria en un conjunto ordenado de datos, el algoritmo divide el conjunto a la mitad en cada paso. Esto significa que en el mejor caso (cuando el elemento buscado está en el centro), el algoritmo puede encontrar el resultado en "O(log n)" operaciones, ya que reduce el espacio de búsqueda en cada iteración.

Calcular **el Peor Caso**:

El peor caso para el algoritmo de ordenamiento burbuja y su variante optimizada ocurre cuando la lista de entrada está ordenada de manera descendente o en orden inverso. En este escenario, ambos algoritmos tendrán que realizar el máximo número de comparaciones e intercambios posibles, lo que resulta en un tiempo de ejecución cuadrático de $O(n^2)$

Ejemplo : **Lista en orden descendente con un solo elemento**

El peor caso ocurre cuando la lista de entrada está ordenada de manera descendente. Puedes crear listas ordenadas en orden inverso como entrada para medir el tiempo en el peor caso.

arr = list(range(n, 0, -1)) Lista ordenada en orden inverso de n a 1

Ejemplo Árboles de Búsqueda Binaria

En la búsqueda, inserción y eliminación en árboles de búsqueda binaria equilibrados,debido a la estructura equilibrada del árbol.Esto significa que el árbol mantiene su altura logarítmica en función del número de nodos.

Calcular **el Caso promedio**:

El caso promedio considera todas las posibles entradas y calcula el rendimiento esperado en función de la probabilidad de que cada entrada ocurra. Es una medida más realista del rendimiento de un algoritmo en situaciones del mundo real, ya que no siempre se sabe si la entrada será la mejor o la peor.

Ejemplo : **Lista aleatoria**
Para el caso promedio, puedes generar listas aleatorias de diferentes tamaños y ejecutar el programa múltiples veces, calculando el tiempo promedio de ejecución.

import random

def generaterandomlist(n):
return [random.randint(1, 1000) for $_i nrange(n)$]

Ejecutar el programa varias veces con listas generadas aleatoriamente y calcular el tiempo promedio.

Ahora justificaremos a cuál clase de las siguientes pertence cada caso de cada algoritmo:

O(1):

pertenece al mejor caso.indica que la complejidad es constante, es decir, no depende del tamaño de la entrada.El algoritmo tiene un rendimiento constante y rápido

O(log n): Generalmente se asocia con el mejor caso en algoritmos que involucran búsqueda, división o reducción del tamaño del problema en cada paso.Es el mejor caso de rendimiento que se puede lograr.

O(n): se asocia con el caso promedio y el peor caso en muchos algoritmos, especialmente en aquellos donde el tiempo de ejecución es directamente proporcional al tamaño de la entrada 'n'.

Esto significa que donde el tiempo de ejecución crece linealmente con el tamaño de la entrada.

O(n log n): comúnmente al peor caso en algoritmos de tipo "divide y vencerás" y a menudo se relaciona con algoritmos de ordenamiento eficientes como el algoritmo de mergesort y el algoritmo de heapsort. También se aplica a algunos algoritmos de búsqueda y estructuras de datos avanzadas.

O(n^2) :

representa una complejidad cuadrática, lo que significa que el número de operaciones requeridas aumenta cuadráticamente con el tamaño de la entrada.

Estos algoritmos pueden ser ineficientes para conjuntos de datos grandes y generalmente se buscan alternativas más eficientes para problemas de mayor escala.

Comparación de resultados

Comparar el tiempo de ejecución de algoritmos de ordenamiento en diferentes casos (mejor, peor y caso promedio) es una forma importante de evaluar su eficiencia y rendimiento en situaciones del mundo real. Aquí hay algunas conclusiones generales que puedes obtener al comparar estos casos:

Algoritmo de Ordenamiento Burbuja:

Mejor Caso: En el mejor caso, cuando la lista ya está ordenada, el algoritmo de burbuja realiza un número mínimo de comparaciones y movimientos. El tiempo de ejecución es relativamente rápido para listas pequeñas, pero se vuelve ineficiente para listas más grandes debido a su complejidad cuadrática "$O(n^2)$".

Peor Caso: En el peor caso, cuando la lista está ordenada en orden inverso, el algoritmo de burbuja requiere el máximo número de comparaciones y movimientos en cada iteración. Esto resulta en un tiempo de ejecución lento y se evidencia la ineficiencia de este algoritmo para conjuntos de datos más grandes.

Caso Promedio: El caso promedio del algoritmo de burbuja suele estar cerca del peor caso, especialmente cuando los datos están distribuidos de manera aleatoria o desordenada. Esto se debe a que, en promedio, requerirá un número significativo de comparaciones y movimientos en listas de tamaño moderado a grande.

Algoritmo de Ordenamiento Burbuja Optimizada:

Mejor Caso: La versión optimizada del algoritmo de burbuja realiza menos comparaciones en el mejor caso, lo que resulta en un mejor rendimiento en comparación con el algoritmo de burbuja básico. Sin

embargo, sigue teniendo una complejidad cuadrática y puede ser lento para conjuntos de datos grandes.

Peor Caso: El peor caso sigue siendo el mismo que el del algoritmo de burbuja básico, con una complejidad cuadrática "$O(n^2)$". $El tiempo de ejecución en el peor caso será ineficiente para listas grandes.$

Caso Promedio: Al igual que en el caso del algoritmo de burbuja básico, el caso promedio para el algoritmo de burbuja optimizada tiende a estar cerca del peor caso, especialmente con datos aleatorios o desordenados.

Conclusiones

Este código proporciona una implementación de los algoritmos de ordenamiento burbuja y burbuja optimizada y mide su tiempo de ejecución en el mejor caso.

Es una herramienta útil para comprender cómo funcionan estos algoritmos y cómo su rendimiento puede variar según el estado inicial de la lista. También destaca la importancia de la optimización de algoritmos de ordenamiento para mejorar el rendimiento en casos reales de uso.

Ambos algoritmos de burbuja tienen un rendimiento deficiente en comparación con algoritmos de ordenamiento más eficientes como mergesort, quicksort o heapsort.

La eficiencia del algoritmo de burbuja está fuertemente influenciada por la disposición de los datos de entrada. Funciona mejor cuando los datos ya están parcialmente ordenados (mejor caso), pero sufre cuando los datos están completamente desordenados (peor caso).

En términos de tiempo de ejecución, el algoritmo de burbuja optimizada muestra una ligera mejora en comparación con el algoritmo de burbuja básico, pero ambos tienen una complejidad cuadrática.

Para conjuntos de datos de gran tamaño, se recomienda utilizar algoritmos de ordenamiento más eficientes, como mergesort, quicksort o heapsort, que tienen una complejidad "O(n log n)" en el peor caso y suelen ser más rápidos en la práctica.