Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas
Instituto Politécnico Nacional

Unidad 1 Análisis de Complejidad Computacional

Práctica 01: Análisis de Casos

Análisis de Complejidad Computacional

Alumno:
Kevin Marin Ramirez Reyna

Septiembre 2023

1. Objetivo de la Práctica

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.

1.1. Objetivo de la Prácticas

Comprender la Variabilidad de la Ejecución: Los algoritmos pueden comportarse de manera diferente según las características de los datos de entrada. Al analizar el mejor, peor y caso promedio,los estudiantes aprenderán a apreciar cómo un algoritmo puede variar en su rendimiento.

Aplicar Conceptos de Complejidad Computacional: La práctica ayudará a los estudiantes a aplicar conceptos fundamentales de la teoría de la complejidad computacional, como la notación O grande (Big O), para describir y comparar el rendimiento de los algoritmos.

Desarrollar Habilidades de Análisis Crítico: A través del análisis de diferentes casos, los estudiantes mejorarán sus habilidades de análisis crítico, aprendiendo a identificar situaciones en las que un algoritmo puede ser más eficiente o ineficiente.

Preparación para Desarrollo de Algoritmos Eficientes: Al comprender los casos de mejor, peor y caso promedio, los estudiantes estarán mejor preparados para diseñar y seleccionar algoritmos eficientes en situaciones reales, lo que es fundamental en la resolución de problemas computacionales.

Promover la Resolución de Problemas: A través de la práctica, los estudiantes aprenderán a abordar y resolver problemas computacionales de manera más efectiva, seleccionando o adaptando algoritmos en función de las restricciones de tiempo y recursos.

2. Desarrollo de la Práctica

2.1. Implementación de los algoritmos

Escribe en python los siguientes métodos de ordenamiento:

Burbuja

Metodo de Burbuja
import time **Libreria para el tiempo**
inicio = time.time()

Se declara el arreglo
arreglo = []

Llenamos el arreglo
for i in range(0,5):
arreglo.insert(i,int(input(f'Introduzca el dato No. i+1 : ')))

Imprimimos el arreglo
print(Asi quedo el arreglo: ")
for i in range(0,5):
print(arreglo[i])

Metodo de la burbuja
for i in range(1,5):
for j in range(0,5-i):

```
if(arreglo[j] ¿arreglo[j+1]):
aux = arreglo[j]
arreglo[j] = arreglo[j+1]
arreglo[j+1] = aux

print(Arreglo Ordenado")
for i in range(0,5):
print(arreglo[i])

Imprimimos el tiempo que tomo ejecutar el programa
fin = time.time()
tiempo ejecutado = fin - inicio
print(f'Vaya si que fue un poco tardado me tarde: tiempo ejecutado')
```

Burbuja Optimizada Metodo de Burbuja Mejorado

```
import time Libreria para el tiempo
inicio = time.time()

Se declara el arreglo
arreglo = []

Llenamos el arreglo
for i in range(0,5):
arreglo.insert(i,int(input(f'Introduzca el dato No. i+1 : ')))

Imprimimos el arreglo
print(Asi quedo el arreglo: ")
for i in range(0,5):
print(arreglo[i])

Metodo de la burbuja mejorado
for i in range(1,5):
Utilizamos la bandera como un auxiliar extra para evitar recorrer el arreglo si ya esta ordenado
bandera = 0 y se inicia en 0
for j in range(0,5-i):
if(arreglo[j]¿arreglo[j+1]):
aux = arreglo[j]
arreglo[j] = arreglo[j+1]
arreglo[j+1] = aux
bandera = 1Si hubo algun cambio la bandera se iguala a 0
if(bandera == 0):
breakSi no hubo cambios se acaba el for

print(Arreglo Ordenado")
for i in range(0,5):
print(arreglo[i])

Imprimimos el tiempo que tomo ejecutar el programa
fin = time.time()
tiempo ejecutado = fin - inicio
print(f'Vaya si que fue un poco tardado me tarde: tiempo ejecutado')
```

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.

2.2. Análisis de Casos

Desarollar un reporte con lo siguiente:

Calcular el mejor, peor y caso promedio para ambos algoritmos de ordenamiento, dando 3 ejemplos de datos de entrada para cada caso.

Análisis de mejores casos en notación Big O. En este caso se requiere de un mínimo de pasos necesarios en los cuales cuando son ejecutados para inicializar la tarea, es decir que el mejor caso se preocupa por la eficacia y sobre todo por eficiencia, pues este solo busca dar los mejores y mas prontos casos para realizar la tarea.
Con Big O Notation expresamos el tiempo de ejecución en términos de: qué tan rápido crece en relación con la entrada, a medida que la entrada

Análisis de peor casos en notación Big O. En este caso se requiere de una cantidad máxima de pasos necesarios en los cuales cuando son ejecutados para realizar la tarea, se puede decir que el peor caso se preocupa por la eficacia y pero no por eficiencia, y no importa cuantos pasos ocupe.
Por ejemplo, realizar un análisis preciso del peor caso como:
T (n) = 12754n2 + 4353n + 834 lg2 n + 13546 Claramente sería un trabajo muy difícil, pero nos proporciona poca información adicional a la observación de que ;el tiempo crece cuadráticamente con n. Ahora podemos ver que expresar tiempos algorítmicos en el mejor, medio y peor caso posible puede ser extremadamente tedioso y complicado.

Análisis de los casos promedios en notación Big O. En esta parte se puede decir que es un poco ambiguo debido a que no hay una medida la cual marque que sea promedio, de que tanto es el promedio de arranque, debido a que todo depende de el tipo de complejidad que tenga el algoritmo, y dependiendo de este, sabremos si puede ser un algoritmo fácil o si tiene mayor complejidad.

Encontrar y justificar a cuál clase de las siguientes pertence cada caso de cada algoritmo:

O(1) Tiempo constante, significa que el tiempo de ejecucion no depende del tamaño de entrada. La complejidad constante nos indica que, sin importar el tamaño de entrada o salida, el tiempo de ejecución y recursos utilizados por nuestro algoritmo siempre será el mismo. Podemos verlas como funciones "estáticas" debido a que siempre se comportarán de la misma manera, no importa las veces que sea ejecutada ni donde.

O(log n) Tiempo logaritmico, comun en algoritmos de busqueda binaria. La complejidad logarítmica es dada cuando el tiempo de ejecución o uso de recursos es directamente proporcional al resultado logarítmico del tamaño del valor de entrada. Es decir, si tenemos un dato de entrada cuyo tamaño es 10 y nos toma 1 segundo en la implementación del algoritmo, significa que por un valor de entrada cuyo tamaño es 100, nos debe tomar 2 segundos en realizar el algoritmo, un valor de 1000 nos debe tomar 3 segundos y así consecuentemente.

O(n) Tiempo linal crece linealmente con el tamaño de entrada. Decimos que un algoritmo tiene complejidad linear, cuando su tiempo de ejecución y/o uso de recursos es directamente proporcional (es decir que se incrementa linealmente) al tamaño del valor de entrada necesario para la ejecución del algoritmo.

O(n log n) Tiempo log linal comun en algoritmos de ordenacion eficiente como el algoritmo de Quicksort y Megasort

$O(n^2)$ Tiempo cuadratico comun algoritmos de fuerza bruta, principalmente se usan en algoritmos para descubrir contraseñas. Encontramos complejidad cuadrática en los algoritmos, cuando su rendimiento es directamente proporcional al cuadrado del tamaño del valor de entrada. Es decir, si tenemos como dato de entrada un arreglo con un tamaño de 4 elementos que queremos comparar para ver si hay elementos repetidos, tendremos que hacer 16 comparaciones en total para completar nuestro algoritmo. Esta complejidad es común encontrarla en algoritmos de ordenamiento de datos como el método de la burbuja, el de inserción y el método de selección, entre algunos otros.

$O(2^n)$ Tiempo exponencial generalmente ineficiente en problemas grandes, es decir entre mayor sea el problema menis eficiente sera. Cuando un algoritmo tiene complejidad exponencial, su rendimiento se incrementa al doble cada vez que se agregue un nuevo dato al valor de entrada, por ende, incrementando su tamaño de manera exponencial.

2.3. Comparacion de Resultados

Comparar el tiempo de ejecución de cada algoritmo de ordenamiento para el mejor, peor y caso promedio. ¿Qué conclusiones puedes sacar de estos resultados?

Que dependiendo segun el tipo de dificultad, que este tenga, es el grado de complejidad que abarcara y no solo eso, si no que tambien abarcara lo que son los tiempos en ejecutarse el o los programas y asi mismo, tambien el tamaño que tengan estos, debido a su dificultad. Y es por eso que es mejor hacer u ocupar una manera distinta de hacerlas, como lo es en el caso de metodo de burbuja y el metodo de burbuja mejorado que en uno tiene que recorrer todo el arreglo para irlos ordenando, mientras que el otro ocupa un auxiliar para que no se recorra todo y solo se recorra de ser necesario y asi evitar gastar memoria y espacio, y tener un programa mas eficiente

3. Conclusión

Por ultimo podemos llegar a la conclusión, de que segun sea el tipo de programa que vayamos a crear para que este sea ejecutado, automaticamente habran puntos claves que nos haran saber que posible dificultad Big O, puede llegar a tener, y no solo eso, si no que tambien mediante la construcción de como se vaya creando, tal vez modificandolo, creando que sea mas eficiente y tenga un menor grado de dificultad y complejidad, para asi, este poder ser incluso mas practico y gaste menos memoria de teclado. Asi mismo podemos llegar a la conclusion, de que existen una variedad de dificultades, que segun su cantidad de problemas, y no solo de los problemas o dificultades que se presenten, pues segun tambien su entrada de datos, como se manejen, entre otros factores al momento de crear el codigo, es como podemos calsificarlos, y saber que tanto pueden tardar en ejecutarse, y posiblemente que tanto gasto de memoria ocupen.

4. Referencias

¿Qué es el Big O Notation en programación? (platzi.com)