INFORME DE LAS PRÁCTICAS DE EXPERIMENTACIÓN Y APLICACIÓN DE LOS APRENDIZAJES

1. **Datos Informativos:**

Facultad:	***CIENCIAS ADMINISTRATIVAS, GESTIÓN EMPRESARIAL E INFORMÁTICA***
Carrera:	***SOFTWARE***
Asignatura:	***ESTRUCTURAS DISCRETAS***
Ciclo:	***PRIMER CICLO***
Docente:	***DARWIN PAUL CARRION BUENAÑO***
Título de la práctica:	***NOTACIÓN O GRANDE***
No. de práctica:	***02***
Escenario o ambiente de aprendizaje de la práctica	***LABORATORIO***
No. de horas:	***48***
Fecha:	***18 DE JULIO DE 2024***
Estudiantes:	***Ariel Alejandro Calderon, Jacson Antonio Narváez, Hermelinda Guadalupe Ochoa***
Calificación	

2. **Introducción:**

La notación Big O es una notación matemática que nos sirve para poner nota a la velocidad de procesamiento de un algoritmo atendiendo a cómo se comporta conforme aumenta el tamaño del trabajo a procesar, por lo que nos sirve para clasificar la eficacia de los mismos. Útil tanto para valorar las necesidades de procesamiento como de espacio necesario para llevar a cabo el algoritmo, y en definitiva valorar qué tan bueno es un algoritmo dado para resolver problemas muy grandes.

3. **Objetivo de la práctica:**

Describir la notación Big O, lo cual es una forma matemática básica de expresar cuánto tarda un algoritmo en ejecutarse atendiendo sólo a grandes rasgos su eficiencia y así poder compararlo con otros. En definitiva, evaluar su complejidad y poner nota a su eficiencia.

4. Descripción del desarrollo de la práctica:

- O (1) - Tiempo constante: es el mejor resultado, y quiere decir que el tiempo de ejecución no varía conforme aumenta el tamaño de los datos de entrada, y la respuesta siempre tarda lo mismo sin importar la magnitud de entrada.

- O(n) - Tiempo lineal: el crecimiento es lineal en tanto el tiempo de ejecución es cada vez mayor de modo proporcional a cómo se incrementa el tamaño de la entrada. Por lo que, si tenemos el doble de elementos de entrada, tardará el doble, aunque despreciamos realmente la pendiente de la misma y sólo nos quedamos con que aumenta de forma lineal.

- O (log n) - tiempo logarítmico: una forma de crecimiento que crece al inicio, pero tiende a estabilizarse conforme aumentan el tamaño de entrada, por lo que es una buena nota para un algoritmo ya que no tiende a resentirse.

- O(n2) - tiempo cuadrático: el crecimiento es de forma exponencial por lo que será un algoritmo a evitar ya que para valores pequeños de entrada el tiempo será asumible, pero conforme aumente el tamaño de los datos de entrada el tiempo tenderá a ser muy elevado y es probable que el procesador se quede inoperativo.

- O(n!) - tiempo factorial: el crecimiento es factorial, por lo que rápidamente tiende a valores imposibles de tratar, en lo que sería una recta vertical.

5. Metodología:

Centrada en la experimentación práctica. Esta aproximación nos permite analizar cómo se comportan los algoritmos en términos de tiempo de ejecución y uso de recursos conforme aumenta el tamaño de los datos de entrada. Mediante la implementación de ejercicios específicos y la medición de métricas clave como el tiempo de ejecución y el consumo de memoria, podremos comparar y contrastar distintos algoritmos bajo condiciones controladas.

6. Resultados obtenidos:

A continuación, enlistamos la invocación de los distintos ejemplos de funciones en lenguaje Javascript:

```
// Ejemplo de función con tiempo constante O(1)
function imprimirPrimerElemento(array) {
    console.log(array[0]);
}

// Ejemplo de función con tiempo lineal O(n)
function imprimirTodosLosElementos(array) {
    for (let i = 0; i < array.length; i++) {
        console.log(array[i]);
    }
}

// Ejemplo de función con tiempo logarítmico O(log n)
function busquedaBinaria(array, elemento) {
    let inicio = 0;
    let fin = array.length - 1;

    while (inicio <= fin) {
        let medio = Math.floor((inicio + fin) / 2);
        if (array[medio] === elemento) {
            return medio; // Elemento encontrado
        } else if (array[medio] < elemento) {
            inicio = medio + 1; // Buscar en la mitad derecha
        } else {
            fin = medio - 1; // Buscar en la mitad izquierda
        }
    }

    return -1; // Elemento no encontrado
}

// Ejemplo de función con tiempo cuadrático O(n^2)
function imprimirParesDelArray(array) {
    for (let i = 0; i < array.length; i++) {
        for (let j = 0; j < array.length; j++) {
            console.log(array[i], array[j]);
        }
```

```
    }
}

// Ejemplo de función con tiempo factorial O(n!)
function calcularPermutaciones(cadena) {
    let permutaciones = [];

    function permutar(prefijo, cadena) {
        if (cadena.length === 0) {
            permutaciones.push(prefijo);
        } else {
            for (let i = 0; i < cadena.length; i++) {
                let nuevoPrefijo = prefijo + cadena.charAt(i);
                let nuevaCadena = cadena.substring(0, i) + cadena.substring(i + 1);
                permutar(nuevoPrefijo, nuevaCadena);
            }
        }
    }

    permutar('', cadena);
    return permutaciones;
}
```

```
JULIANA CALDERON@Multi-PC MINGW64 ~/Desktop
$ nodemon main.js
[nodemon] 3.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node main.js`
Ejemplo de tiempo constante O(1):
Primer elemento del array: 1
-------------------------------------
Ejemplo de tiempo lineal O(n):
Elementos del array:
1
2
3
4
5
-------------------------------------
Ejemplo de tiempo cuadrático O(n^2):
Pares del array:
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
-------------------------------------
Ejemplo de tiempo logarítmico O(log n):
Elemento encontrado en la posición: 4
-------------------------------------
Ejemplo de tiempo factorial O(n!):
Todas las permutaciones: [ 'abc', 'acb', 'bac', 'bca', 'cab', 'cba' ]
-------------------------------------
[nodemon] clean exit - waiting for changes before restart
```

7. Conclusiones:

En conclusión, el estudio de la eficiencia de los algoritmos mediante la notación Big O y la experimentación práctica nos proporciona herramientas poderosas para entender y comparar el rendimiento de diferentes soluciones algorítmicas. A través de ejemplos implementados en JavaScript, hemos visto cómo las complejidades Big O como O(1), O(n), O(n^2), O(log n) y O(n!) se reflejan en el tiempo de ejecución y el uso de recursos computacionales.

8. Recomendaciones:

- Comprender la importancia del análisis de complejidad
- Aplicar la teoría en la práctica
- Experimentar con conjuntos de datos variados
- Utilizar herramientas y frameworks

9. Bibliografía:

[1] Explicación de la notación Big O con Ejemplos: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.freecodecamp.org/espanol/news/explicacion-de-la-notacion-big-o-con-ejemplo/

[2] Notación Big O. Artículo básico de análisis: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://medium.com/%40diego.coder/notaci%25C3%25B3n-big-o-615bd1e0a227

[3] Todo sobre Big O Notación y su impacto en algoritmos: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://elmundodelosdatos.com/todo-sobre-big-o-notation-y-su-impacto-en-algoritmos/

10. Anexos: