IMPORTANCIA DE LA OPTIMIZACION DE ALGORITMOS EN TIEMPO Y ESPACIO.

José Alejandro Gómez Castro
e-mail: jgomezc@ucenfotec.ac.cr
Andrés Salas Olsen
e-mail: asalaso@ucenfotec.ac.cr

1 INTRODUCCIÓN

El siguiente reporte tiene como objetivo explicar, ¿Por qué es importante optimizar los algoritmos?, ¿Cuáles son las ventajas de la optimización de los algoritmos? , estas preguntas son clave para todo desarrollador debido que la respuestas a estas preguntas marca la diferencia entre programadores.

Estas preguntas surgen a la necesidad de crear aplicaciones con respuestas rápidas, y a pesar que las componentes de las computadoras son más novedosas y con mayor eficiencia o velocidad de procesar instrucciones, se busca con mayor razón apps con respuestas rápidas ya que los usuarios cada vez son más ambiciosos como por ejemplos: Transacciones web, realidad virtual, simulaciones gráficas, gráficas de alta resolución, entre otras más.

2 Análisis de Algoritmos

2.1 Conceptos

Un concepto clave en este tema es el término de “Eficiencia” el cual es la medida de la cantidad de recursos de la computadora que consume un programa en ejecución; y surge otro concepto importante el cual es “Recursos” y su definición es el tiempo que se demora una aplicación en ejecución y el espacio o cantidad de memoria que este necesita para ejecutarse correctamente. Entre menos tiempo y espacio mayor será su eficiencia.

2.2 Análisis de Algoritmos

El propósito del análisis de algoritmos es establecer una medida de calidad de los algoritmos, que permita compararlos sin necesidad de implementarlos. Además de usarlo para evaluar algoritmos, también se usa para evaluar el diseño de las estructuras de datos de un programa, dad una representación.

En un código a la hora que declaramos una variable y la inicializamos (Ej: int x = 0;) estamos utilizando la memoria y cuando utilizamos una estructura de control como el “if-else”, “For”, “Do-While”, estamos hablando de la estructura del programa.

2.3 Tiempo de Ejecución y el Espacio

El tiempo de ejecución de los programas van a depender de:

- La velocidad de la computadora en donde se ejecuta la aplicación.
- Las características del compilador o la plataforma en la que se desarrolló el programa
- La complejidad o simplicidad de la estructura La complejidad o simplicidad de la estructura del algoritmo o instrucciones del programa
- El número o tamaño de los datos de entrada

Y El espacio que usan los programas depende del número, el tamaño y la complejidad de datos de entrada, variables temporales o auxiliares, y datos de salida que el algoritmo usa en su ejecución.

3 Cálculo de la complejidad en tiempo

Su objetivo es asociar cada algoritmo con una función matemática que mida su eficiencia. El tiempo de ejecución puede variar dependiendo de los datos de entrada como por ejemplo la búsqueda de un dato de un arreglo puede ser afectado por el tamaño del arreglo.

Por lo anterior, para el análisis se pueden considerar los tiempos utilizados por el algoritmo en el mejor caso, el peor caso y en el caso promedio.

En el mejor caso es un algoritmo es analizado, pero usualmente no es de mucho interés porque no refleja el comportamiento típico. El caso promedio también puede analizarse y a menudo refleja el comportamiento típico. El peor caso representa una garantía de la eficiencia de eficiencia de cualquier entrada posible al algoritmo. Por esta razón es el que más se tiene en cuenta.

3.1 Reglas para calcular la complejidad de un algoritmo. (Big O)

Se trata de encontrar una función f(n) fácil de calcular y conocida, que mejor acote el crecimiento del tiempo del algoritmo en función de los datos de entrada.

Al decir que un algoritmo es O(f(n)), significa que al aumentar la cantidad de datos de entrada, o sea "n" el tiempo del algoritmo crece como crece "f" en relación a "n".

La técnica para el cálculo de la complejidad en notación O es: Primero observar la estructura del algoritmo, Segundo analizar parte por parte y prestar atención a los ciclos anidados y finalmente aplicar las reglas para determinar la complejidad total del algoritmo.

3.1.1 Instrucciones Simples

La técnica para el cálculo de la complejidad en notación O es: Primero observar la estructura del algoritmo, Segundo analizar parte por parte y prestar atención a los ciclos anidados y finalmente aplicar las reglas para determinar la complejidad total del algoritmo.

Ejemplo: int x,y,z; O(1)
a = 0; O(1)

3.1.2 Eliminación de Constantes

Sea A1 un algoritmo, y suponga que la complejidad en tiempo de A1 es de O(K f(n)), entonces es de O(f(n)). Donde K es una constante.

Ejemplo: Si la complejidad es de O(2n), entonces se simplifica a O(n).

3.1.3 Secuencia de instrucciones o algoritmos

Sea A un algoritmo que tiene dos partes secuenciales e independientes A1 y A2. Y suponga que la complejidad en tiempo de A1 es O(f1(n)) y la complejidad en tiempo de A2 es de O(f2(n)), entonces el tiempo de ejecutarse ambas partes es O(á i (f1( ) f2( ))) O(máximo(f1(n), f2(n))).

3.1.4 Instrucciones repetitivas

Para saber la complejidad en tiempo de A, se halla el máximo entre $O(n^3)$, $O(n^2)$ y $O(5n^3)$. Luego la complejidad en tiempo de A es $O(5n^3)$. Y por la regla 3 la complejidad de A queda en $O(n^3)$.

4 Ejemplos (Recursividad, Patrones,..)

1. **Cálculo de la potencia de un numero: recursividad.**

```
int potencia(int, int);

using namespace std;

int main(int argc, char** argv) {

    cout << potencia(2,8);

    return 0;
}

int potencia(int b, int e){
    if(e > 1){
        return b*potencia(b, e - 1);
    }else{
        if(e == 0){
            return 1;
        }else{
            return b;
        }
    }
}
```

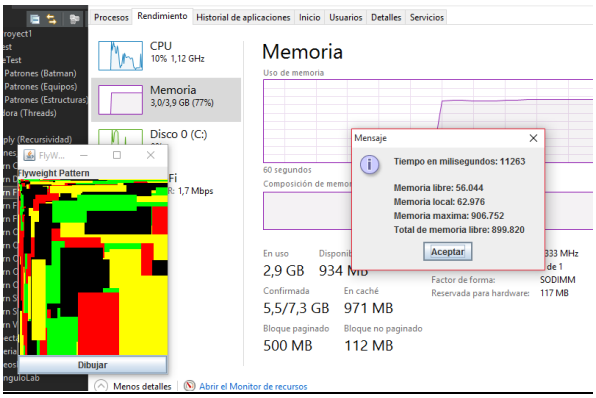
2. **Torres de Hanoi : recursividad.**

```
#include <iostream>

using namespace std;

void hanoi(int, char, char, char);

int main(int argc, char** argv) {

    int disks = 5;

    hanoi(disks, 'A', 'C', 'B');

    cin.get();

    return 0;
}

// hanoi(disks, origin, destiny, auxiliar)
void hanoi(int d, char ori, char dest, char aux){

    if( d == 1 ){
        cout << "Move disk " << d << " from " << ori << " to " << dest << endl;
    }else{
        hanoi( (d-1), ori, aux, dest );
        cout << "Move disk " << d << " from " << ori << " to " << dest << endl;
        hanoi( (d-1), aux, dest, ori );
    }

    cout << "\n-- execute algorithm --\n" << endl;

    return;
}
```

3. **Patrones de Diseño: FlyWeight (Peso Ligero).**

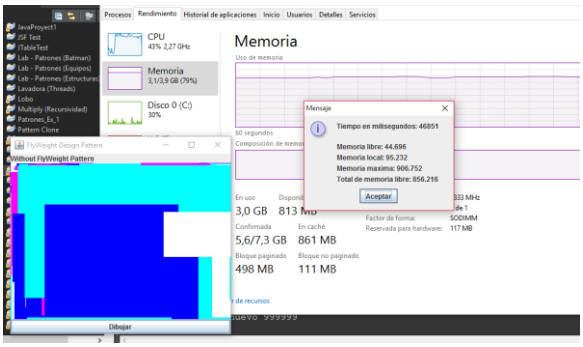
Aplicando el Patrón: Comparar Resultados.

```
Memoria libre: 56.044
Memoria local: 62.976
Memoria máxima: 906.752
Total de memoria: 899.820
```



Sin Patrón: Comparar Resultados.

Memoria libre: 44.696
Memoria local: 95.232
Memoria máxima: 906.752
Total de memoria: 856.216



5 Referencias

[1] Weiss, Mark Allen. Data Structures & Algorithm Analysis in C++. Addison Wesley.1999. Pág 44-50.

[2] Kingston, Jeffrey. Algorithms and Data Structures. Addison Wesley. 1997. Cáp 2

[3] Github, Jose Alejandro Gomez Castro. Astrofreakazoid. Costa Rica. 2016. Recursividad y Patrones de Diseño.

- https://github.com/AstroFreakazoid/CoolCodes_Cplusplus
- https://github.com/AstroFreakazoid/CoolCodes_JAVA