**Eficiencia de los Algoritmos**

1. **Introducción**

La eficiencia de los algoritmos es un aspecto crucial en el diseño y análisis de algoritmos. Se refiere a la medida en que un algoritmo utiliza recursos, como tiempo y espacio, para resolver un problema dado. En el estudio de la eficiencia de los algoritmos, nos preocupamos por comprender cómo varía el tiempo de ejecución o el uso de recursos a medida que cambian las entradas del problema.

Lo más importante de un algoritmo es que sea fácil de entender, y por tanto, de mantener en una situación normal. Pero si presenta problemas de eficiencia, ya tendremos que valorar otras opciones que, aunque puedan restar legibilidad, puede que mejoren en velocidad y en uso de otros recursos. Tenemos que prestar especial atención a los bucles, por ejemplo, si hay operaciones dentro de un bucle que son idénticas en todas las operaciones y que se podrían hacer una sola vez fuera del bucle. Anidar bucles también es una práctica de riesgo, pues nos acerca a complejidades exponenciales. Evita calcular la longitud del elemento que recorres en cada iteración; si no va a variar, calcúlalo antes de entrar en el bucle, guárdalo en una variable y úsala para controlar el recorrido.

Todo algoritmo debe cumplir con las siguientes características:

1. **Definición de Eficiencia de Algoritmos**

La eficiencia de un algoritmo se puede definir en términos de su tiempo de ejecución y su uso de memoria. En términos generales, se busca minimizar el tiempo de ejecución y el espacio en memoria utilizados por un algoritmo.

* 1. **Explicacion Teorica:**

La eficiencia de los algoritmos se puede medir utilizando la notación de O grande (Big O). Big O proporciona una forma de expresar la eficiencia en términos asintóticos, es decir, cómo se comporta el tiempo de ejecución del algoritmo a medida que el tamaño de la entrada se acerca al infinito.

Por ejemplo, si tenemos un algoritmo cuyo tiempo de ejecución es O(n^2), esto significa que el tiempo de ejecución del algoritmo crece cuadráticamente con el tamaño de la entrada (n). Cuanto menor sea el grado del polinomio en la expresión O(n^x), más eficiente será el algoritmo.

* 1. **Explicación Matemática:** La notación Big O se define formalmente como sigue:

O(g(n)) = {f(n): existen constantes c y n0 tales que 0 ≤ f(n) ≤ cg(n) para todo n ≥ n0}

Esto significa que una función f(n) pertenece a O(g(n)) si y solo si existe una constante c positiva y un valor n0 tal que f(n) está acotada por encima por cg(n) para todo n mayor o igual a n0. Memoria que utiliza un programa para su ejecución, indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la memoria estática de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la memoria dinámica, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

1. **Clasificación**

Los algoritmos pueden clasificarse en diferentes categorías según su eficiencia. Algunas de las clasificaciones comunes incluyen:

Algoritmos de tiempo constante (O(1))

Algoritmos de tiempo lineal (O(n))

Algoritmos de tiempo logarítmico (O(log n))

Algoritmos de tiempo polinómico (O(n^k)), donde k es una constante

Algoritmos de tiempo exponencial (O(2^n))

1. **Ejemplos de Eficiencia**

Tenemos el siguiente ejemplo donde vamos a analizar la complejidad

**Complejidad Lineal**

**void numeros\_pares\_impares(int numero)** {

int cont\_imp = 0;

for (int i = 1; i <= numero; ++i) {

if (i % 2 == 0) {

std::cout << i << " Es Par" << std::endl;

} else {

std::cout << i << " Es Impar" << std::endl;

cont\_imp++;

}

}

std::cout << "Cantidad de Números Impares: " << cont\_imp << std::endl;

}

**int main()** {

int num;

std::cout << "Ingrese un número: ";

std::cin >> num;

numeros\_pares\_impares(num);

return 0;

}

Para calcular la complejidad algorítmica de este código, primero analicemos los diferentes bloques de código:

1. El bucle **for** itera desde **1** hasta **numero** inclusive, donde **numero** es el parámetro de entrada.
2. Dentro del bucle, se realiza una verificación condicional **if (i % 2 == 0)** para determinar si **i** es par o impar.
3. En función de la paridad de **i**, se imprime un mensaje correspondiente.
4. Además, se incrementa el contador **cont\_imp** cuando **i** es impar.
5. Finalmente, se imprime la cantidad de números impares encontrados.

Vamos a analizar la complejidad de cada uno de estos bloques:

* El bucle **for** ejecuta **numero** veces, por lo que su complejidad es O(numero).
* La verificación **if (i % 2 == 0)** se realiza en cada iteración del bucle, lo cual es una operación de tiempo constante O(1).
* La impresión de un mensaje es una operación de tiempo constante O(1).
* El incremento de **cont\_imp** es una operación de tiempo constante O(1).

Por lo tanto, la complejidad total del algoritmo es dominada por el bucle **for**, que es lineal en función de **numero**, por lo que la complejidad del algoritmo es O(numero).

Esto significa que la complejidad del algoritmo **es lineal**, lo que indica que el tiempo de ejecución del algoritmo crecerá linealmente con el valor de entrada **numero**.

**Complejidad Cuadrática**

#include <iostream>

**void imprimir\_pares(int n)** {

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

std::cout << "(" << i << ", " << j << ") ";

}

std::cout << std::endl;

}

}

**int main()** {

int numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

imprimir\_pares(numero);

return 0;

}

Este código tiene un bucle anidado dentro de otro bucle, lo que resulta en una complejidad cuadrática. Ahora, hagamos el análisis de la complejidad:

* El bucle externo ejecuta n veces, donde n es el valor ingresado por el usuario.
* Dentro de cada iteración del bucle externo, el bucle interno también se ejecuta n veces.
* Por lo tanto, el número total de iteraciones del bucle interno es n \* n, lo que resulta en una complejidad cuadrática.

Entonces, la complejidad de este algoritmo es O(n^2), donde n es el número ingresado por el usuario. Esto significa que el tiempo de ejecución del algoritmo aumentará cuadráticamente con el valor de entrada número.

**6. Conclusiones**

**7. Referencias**

*LUDA UAM-Azc.* (2024). Aniei.org.mx. <http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_01.html>

‌