**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.

1. **Definición de Eficiencia de Algoritmos**

Cuando se enfrenta un problema, es común encontrar múltiples algoritmos adecuados para su resolución, y la elección del mejor es fundamental. ¿Cómo decidir entre diferentes opciones algorítmicas para abordar un mismo problema?

En situaciones donde se debe resolver un número limitado de casos de un problema simple, la elección del algoritmo puede no ser crítica. En estos casos, la preferencia puede inclinarse hacia la rapidez de implementación o la disponibilidad de algoritmos preexistentes, sin necesidad de considerar sus propiedades teóricas. Sin embargo, cuando se enfrentan numerosos casos o problemas de mayor complejidad, la selección del algoritmo requiere un enfoque más meticuloso y fundamentado (Walter B, 2020).

Existen dos enfoques principales para la selección de algoritmos:

1. Enfoque empírico (o a posteriori): Implica la implementación de diferentes técnicas y su evaluación mediante pruebas en la computadora, utilizando una variedad de casos de prueba.
2. Enfoque teórico (o a priori): Consiste en determinar, mediante análisis matemático, los recursos necesarios para cada algoritmo en función del tamaño de los casos considerados. Los recursos de interés principal suelen ser el tiempo de ejecución (el más crítico) y el espacio de almacenamiento, principalmente la memoria principal de la computadora.

Se compararán los algoritmos principalmente desde el enfoque teórico, basándose en sus tiempos de ejecución. La eficiencia de un algoritmo se evaluará principalmente en función de su velocidad de ejecución, considerando como más eficiente aquellos que ejecuten las tareas en menos tiempo.

1. **Clasificación**

Dado que no hay una unidad de medida estándar para expresar la eficiencia de un algoritmo, se suele evaluar en términos del tiempo requerido por el algoritmo para ejecutarse. En este sentido, se puede afirmar que un algoritmo para resolver un problema necesita un tiempo aproximado de t(n), donde t es una función dada, si existe una constante positiva c y una implementación del algoritmo que pueda resolver todos los casos de tamaño n en un tiempo no mayor a ct(n) unidades de tiempo. Es importante destacar que la unidad de tiempo es arbitraria y puede ser definida según el contexto, ya sea años, minutos, días, horas, segundos, milisegundos, etc.

Esto mismo se puede definir formalmente:

T(n) = O(f(n)) si existe una constante c y un valor n0 tales que T(n)<= c f(n) cuando n>n0

Algunos órdenes de complejidad son tan comunes que tienen nombres propios, como se detalla a continuación. En la figura 1 se muestran las funciones de crecimiento para valores de tamaño del uno al diez. Aunque este rango de valores es limitado, es adecuado para visualizar las diferencias entre los distintos órdenes de complejidad y cómo cambia el tiempo de ejecución en función del tamaño de la entrada:

1. en el orden de O(c), o de tiempo constante;

2. en el orden de O(log n), o de tiempo logarítmico;

3. en el orden de O(n), o de tiempo lineal;

4. en el orden de O(n log n), o de tiempo casi lineal;

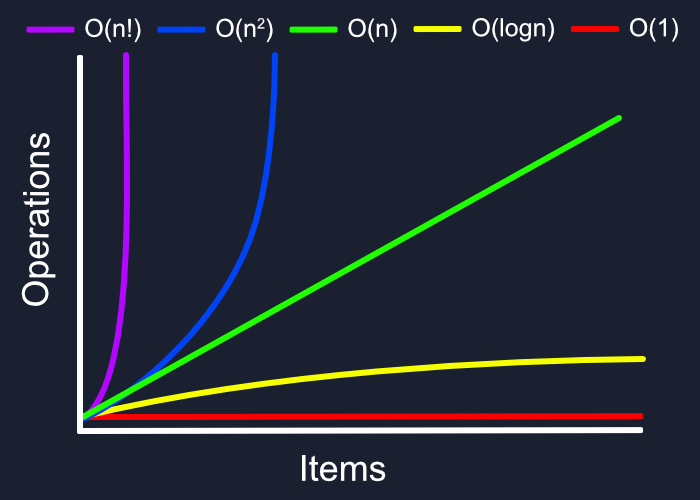
5. en el orden de O(n2), o de tiempo cuadrático;

6. en el orden de O(n3), o de tiempo cúbico;

7. en el orden de O(nk), o de tiempo polinómico;

8. en el orden de O(cn), o de tiempo exponencial;

9. en el orden de O(n!), o de tiempo factorial.



**Imagen 1** Eficiencia basado en la complejidad de las operaciones (undefinedworld.com)

1. **Enfoque de Análisis de Algoritmos**

Cuando se analiza un algoritmo se lo puede realizar desde tres enfoques útiles:

1. Mejor caso, poco útil dada la poca o escasa ocurrencia de dicho caso.

2. Caso medio, útil cuando los casos a resolver son muy volátiles e igualmente probables que ocurran (es decir equiprobables). Requiere de mucha información a priori respecto de los casos a resolver. Esto en general es difícil de conocer.

3. Peor caso, es adecuado para problemas críticos donde se debe conocer el tiempo requerido para el peor caso. No requiere información respecto de los casos a resolver.

Es necesario abordar dos conceptos fundamentales:

* **Operación elemental:** Se refiere a una instrucción básica en la computación, como una suma o una asignación, que se puede realizar en un tiempo constante, independientemente de la implementación específica. Por simplicidad, se considera que estas operaciones tienen un costo unitario.
* **Notación asintótica:** Esta notación se centra en el comportamiento de las funciones para valores muy grandes de sus parámetros. Permite comparar algoritmos incluso para tamaños moderados o grandes de datos. En esta notación, "n" representa el tamaño del conjunto de datos y "t(n)" indica la cantidad de recursos necesarios para implementar un algoritmo en función de ese tamaño.

**7. Referencias**

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

‌

**8. Anexo**

**Ejemplo 1**

#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;

**Ejemplo 2**

if num > 10:

  print('es mayor que 10')

print('su cuadrado es 100')

aux = num % 2

else:

  print('es menor que 10')

print('su cubo es 1000')

aux = num *// 2*

  print(aux)

producto = num \* 2

**Ejemplo 3**

if num > 10:

print('es mayor que 10')

print('su cuadrado es 100')

aux = num % 2

if(num%20):

print('es divisible por 2')

else:

print('es divisible por 2')

    num = num + 1

    aux = num \* num \* num

else:

print('es menor que 10')

print('su cubo es 1000')

    aux= num *// 2*

print(aux)

    producto = num \* 2

if(producto % 3 = 0):

print('es divisible por 3')

**9.** **Guía de ejercicios prácticos**

A continuación, se plantean una serie de fragmentos de códigos. Habrá que realizar un análisis de la complejidad de cada una de las diferentes funciones.

**Ejercicio 1**

numero= int(input('ingrese un número'))

while (numero≠0) and (len(lista) < 10000):

    lista.append(numero)

    numero = int(input('ingrese número'))

for i in range(0, len(lista)):

    print(lista[i])

**Ejercicio 2**

while (p ≤ u) and (pos = -1):

   med = (pu) *// 2*

   if(lista[med] x):

      pos = med

else:

    if(x > lista[med]):

         p = med + 1

else:

    p = med 1