**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;

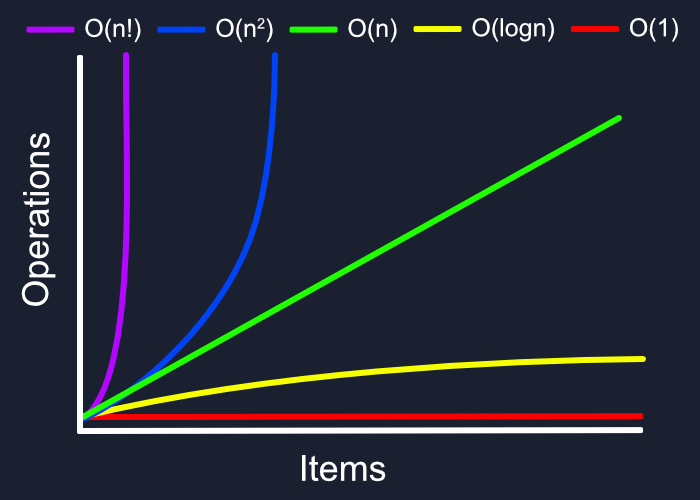
5. en el orden de O(n2), o de tiempo cuadrático;

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.

**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

**Tiempo de ejecución**

* + 1. **Concepto**

El tiempo de ejecución se refiere a la cantidad de tiempo que tarda un algoritmo en completar su tarea para una entrada de datos dada. Es una métrica fundamental para evaluar la eficiencia de un algoritmo y determinar su escalabilidad a medida que aumenta el tamaño de los datos de entrada.

* + 1. **Notación**

Existen diferentes notaciones para expresar el tiempo de ejecución de un algoritmo, siendo la notación asintótica la más común. La notación asintótica se enfoca en el comportamiento del algoritmo cuando el tamaño de la entrada crece indefinidamente, ignorando factores constantes y términos menos significativos. Las notaciones más utilizadas son:

* **Notación O grande (O):** Representa el peor caso o el límite superior del tiempo de ejecución del algoritmo.
* **Notación Omega (Ω):** Representa el mejor caso o el límite inferior del tiempo de ejecución del algoritmo.
* **Notación Theta (Θ):** Representa el caso promedio o el orden exacto del tiempo de ejecución del algoritmo.
  + 1. **Ejemplo**

Supongamos que tenemos un algoritmo de búsqueda lineal en un arreglo de n elementos. En el mejor caso, el elemento buscado se encuentra en la primera posición, por lo que el tiempo de ejecución es Ω(1). En el peor caso, el elemento buscado se encuentra en la última posición o no está presente, por lo que el tiempo de ejecución es O(n). En el caso promedio, el tiempo de ejecución es Θ(n).

* 1. **Aporte matemático**
     1. **Complejidad temporal**

La complejidad temporal se refiere al tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Proporciona una medida cuantitativa de la eficiencia del algoritmo

El análisis de la complejidad temporal se realiza considerando tres casos principales: el mejor caso, el caso promedio y el peor caso. Esto permite comprender el comportamiento del algoritmo en diferentes escenarios.

Se utilizan herramientas matemáticas, como el álgebra y el cálculo, para determinar la complejidad temporal de un algoritmo. Esto implica analizar el número de operaciones realizadas en función del tamaño de la entrada y expresar la complejidad mediante notaciones asintóticas.

* + 1. **Algoritmos**

Los algoritmos pueden ser representados de manera formal utilizando notaciones matemáticas, como pseudocódigo, diagramas de flujo o descripciones matemáticas precisas.

Se utilizan técnicas de demostración matemática, como la inducción, para probar la corrección de un algoritmo, asegurando que produzca resultados correctos para todas las entradas válidas. Además, se demuestra la terminación del algoritmo, garantizando que finalice en un tiempo finito.

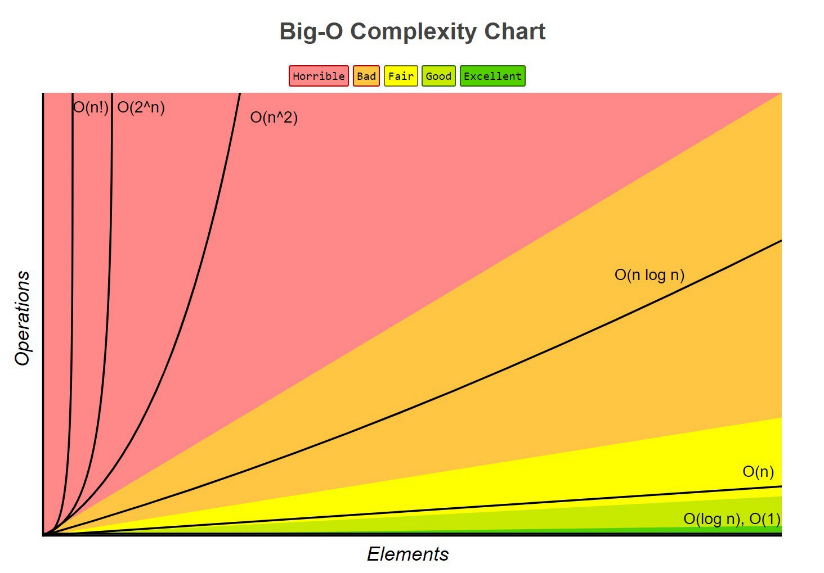
Muchos algoritmos recursivos pueden ser analizados mediante relaciones de recurrencia, que describen la relación entre el costo de una instancia del problema y el costo de sus subproblemas. Se utilizan métodos matemáticos, como el método de sustitución, el método de iteración y el método del árbol de recursión, para resolver estas relaciones de recurrencia.

* + 1. **Notación Big O**

La notación Big O se utiliza para describir el comportamiento asintótico de la complejidad temporal de un algoritmo a medida que el tamaño de la entrada crece. Proporciona una cota superior en el peor caso**.**

Existen reglas y propiedades matemáticas que rigen el cálculo y la manipulación de la notación Big O, como la aditividad, la multiplicación y la transitividad.

La notación Big O permite comparar la eficiencia de diferentes algoritmos que resuelven el mismo problema, al observar cómo se comporta su complejidad temporal a medida que aumenta el tamaño de la entrada.

**Figura 1**

*Figura 1: Tabla de notación Big O*

**Notación Asintótica**

Para entender el concepto de la notación asintótica debemos comprender dos conceptos básicos. Primero, el tiempo que se tarda un algoritmo dependiendo del tamaño de su entrada. Para esto tomamos en cuenta que el número máximo de intentos en las búsquedas lineales y las búsquedas binarias aumentan a medida que la longitud del arreglo aumenta. Siguiendo esta idea podemos concluir que el tiempo de ejecución del algoritmo es una función del tamaño de su entrada.

Segundo debemos saber en qué tan rápido crece una función por su tamaño de entrada, esto se llama tasa de crecimiento del tiempo de ejecución. Para ser mas eficaces con este concepto, tenemos que simplificar la función y extraer la parte mas importante. Tomemos de ejemplo un algoritmo que corre con una entrada de tamaño , se tarda instrucciones de maquina es ser completado. El termino , sin importar el valor de n, va a crecer a un ritmo mas acelerado que los términos .

Gráfico, Gráfico de líneas

Descripción generada automáticamente  
*Ilustración 2: Grafico tiempo de ejecución*

Se puede concluir que el tiempo de ejecución crece como , descartando el coeficiente y los términos restantes, siempre que el tiempo de ejecución sea para siempre habrá un valor de para que sea mayor al resto de términos.

Al descartar estos términos no significativos y con coeficientes constantes, nos enfocamos en la tasa de crecimiento de un algoritmo, en este momento utilizamos la notación asintótica. Tiene tres formas, la notación grande (Big), notación O grande (Big O) y notación Ω grande (Big Ω).

**Notación θ grande (Big θ):**

Tenemos como ejemplo el código:

**void** **busqueda\_lineal**(std::string funcion, **char** valor) {

**for** (**int** intentos = **0**; intentos < funcion.length(); guess++) {

**if** (funcion[intentos] === valor) {

**return** intentos;//valor encontrado y su posición

}

}

**return** -**1**;// no se encontro el valor

};

En este código el tamaño del string sería el cual es el número máximo de veces que se puede ejecutar el bucle for.

Cada que el bucle for itera, tiene que hacer varios cálculos:

* Compara “intentos” con el tamaño del string.
* Compara el valor de la posición del string con el valor.
* Regresar el valor de intentos (si es que encuentra el valor).
* Incrementar el valor de intentos.

Cada calculo toma un tiempo constante y si el bucle itera veces, entonces el tiempo para todas las iteraciones sería donde es la suma de los tiempos para el cálculo en una iteración del bucle. no tiene un valor fijo dado que depende de muchos factores. es el tiempo que el bucle for y también es una constante. Por tanto, el tiempo de búsqueda lineal, en el caso de que no se encuentre el valor, es de .

Para valores menores de , no es importante el tiempo de ejecución. Pero cuando es lo suficientemente grande utilizamos Theta de , o (), y se concluye que el tiempo de ejecución estará en el intervalo inferior de y .

Gráfico, Gráfico de líneas

Descripción generada automáticamente *Ilustración 3: Tasa de crecimiento Big Theta.*

En el calculo de Theta, no importan las unidades de tiempo que usemos, al utilizar la notación grande, decimos que tenemos una *cota* *asintóticamente ajustada* sobre el tiempo de ejecución. Asintóticamente porque solo importan los valores grandes de , Cota ajustada porque ajustamos el tiempo de ejecución dentro del rango de una constante mayor y menor.

Las funciones en notación asintótica se clasifican de mayor a menor dependiendo que tan rápido crecen. Es así como la primera crece a un ritmo lento y la ultima a un ritmo bastante acelerado:

1. (1)
2. ()
3. ()
4. ()
5. ()
6. ()
7. ()
8. ())
9. )

**Notación O grande (Big O):**

Utilizamos Big O para acotar asintóticamente el crecimiento de factores dentro de constantes por arriba y por abajo. Aunque también se pueden acotar solo por arriba.

Por ejemplo, el peor tiempo de ejecución de una búsqueda binaria es , aunque no siempre se ejecuta en ese tiempo. Si se encuentra el valor en la primera iteración entonces es valido decir que se ejecuto en un tiempo . En conclusión, el tiempo de ejecución nunca es peor que pero si puede ser mejor.

Usamos Big O significa que el tiempo de ejecución crece a lo más por una constate dada, pero este tiempo también puede crecer más lentamente.

*Un tiempo de ejecución para una lo suficientemente grande, el tiempo de ejecución es para una constante de esta forma:*Gráfico, Gráfico de líneas

Descripción generada automáticamente *Ilustración 4: Tasa de crecimiento Big O.*

Utilizamos Big O para cotas superiores asintóticas, ya que la cota es para el tiempo de ejecución por arriba de entradas con entradas suficientemente grandes.

Entonces, si comparamos la definición de Big Theta y Big O vemos que tienen similitudes, pero Big O se concentra únicamente en la cota de arriba. Se puede decir que el tiempo de ejecución en una búsqueda binaria va a ser siempre para cualquier valor de n.  
Una búsqueda binaria puede ejecutarse siempre en un tiempo pero no siempre se ejecuta en un tiempo .

Big O dará un valor que puede ser mayor al tiempo de ejecución, pero esta en el rango, podemos utilizar una analogía como ejemplo. Si tenemos 10 dólares en el bolsillo y afirmamos a un amigo que tenemos menos de 100 dólares, la afirmación va a ser correcta pero no muy precisa. Así mismo funciona la notación Big O.

**Notación Omega grande (Big Ω)**

En la notación Omega grande decimos que el algoritmo toma por lo menos una cierta cantidad de tiempo, para una suficientemente grande, el tiempo de ejecución es por lo menos para una constante . El tiempo de ejecución se grafica de la forma:

Gráfico, Gráfico de líneas

Descripción generada automáticamente *Ilustración 4: Tasa de crecimiento Big Ω.*

A diferencia de Big O, Big Ω utiliza los límites asintóticos inferiores, ya que acota los crecimientos por debajo de un lo suficientemente grande.

Así como implica , también implica , por ende, el peor caso de búsqueda binaria para Big Ω es

Con la misma analogía del dinero en el bolsillo utilizado en Big O, podemos decir que, “tengo por lo menos 1 dólar en el bolsillo”. Sigue siendo correcto, pero no muy preciso, de igual forma es el caso de la búsqueda binaria de Big Ω al afirmar que el mejor de los casos es, es correcto, pero no preciso pues sabemos que hay al menos un tiempo constante.

1. **Aritmética de notación O.**

Decimos que un algoritmo tiene un orden sí existen un y un siendo tal que para todo , La relación denota una dominancia de funciones, en donde la función está acotada superiormente por un múltiplo de la función ( es dominada por ) Así, la expresión refleja que el orden de crecimiento asintótico de la función es inferior o igual al de la función . (Diaz, N., 2014).

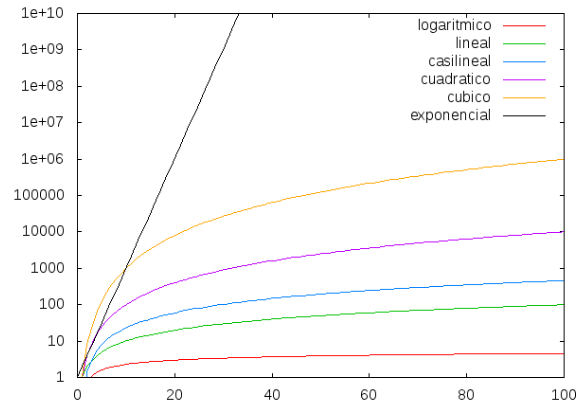
A continuación, se enumeran las propiedades básicas de la notación 0:



1. **Reglas de Cálculo de Complejidad:**

* El tiempo de ejecución de cada sentencia simple, por lo común puede tomarse como . Algunas operaciones matemáticas no deben ser tratadas como operaciones elementales. Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos. Sin embargo, en la práctica se consideran elementales siempre que los datos que se usen tengan un tamaño razonable.
* El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Es el máximo tiempo de ejecución de una proposición de la sentencia.
* Para las sentencias de bifurcación (IF, CASE) el orden resultante será el de la bifurcación con mayor orden.
* Para los bucles es el orden del cuerpo del bucle sumado tantas veces como se ejecute el bucle. Este tiempo debería incluir: el tiempo propio del cuerpo y el asociado a la evaluación de la condición y su actualización, en su caso. Si se trata de una condición sencilla (sin llamadas a función) el tiempo es , que no es relevante para al cálculo asintótico. Cuando se trata de un bucle donde todas las iteraciones son iguales, entonces el tiempo total será el producto del número de iteraciones por el tiempo que requiere cada una.
* Para los bucles anidados el orden corresponde a la multiplicación del orden del bucle interno por el orden del bucle externo.
* El orden de una llamada a un subprograma no recursivo es el orden del subprograma. (Diaz, N., 2014).
* Algunos de órdenes de complejidad que se presentan con frecuencia son:
* Lineal:
* Cuadrático:
* Polinómico: con que pertenece a los naturales.
* Logarítmico:
* Exponencial:

En la figura 1 se presenta una comparación entre los principales órdenes de complejidad. Es de destacar el excesivo costo que representa una complejidad de orden , frente a la estabilidad de la complejidad logarítmica.



**Imagen 1.** *Aritmética de notación O.*

1. Ejemplo.

En el ejemplo se utiliza un bucle for para recorrer la matriz y encontrar el máximo elemento. La notación O para este algoritmo es O(n), donde n es el tamaño de la matriz. Esto significa que el tiempo de ejecución crece linealmente con respecto al número de elementos en la matriz.

main.cpp

#include <iostream>  
#include <vector>  
  
// Función para encontrar el máximo elemento en una matriz  
**int** encontrarMaximo(**const** std::vector<**int**>& arr) {  
 **int** maxElemento = arr.at(0);  
 **for** (**int** i = 1; i < arr.size(); ++i) {  
 **if** (arr.at(i) > maxElemento) {  
 maxElemento = arr.at(i);  
 }  
 }  
 **return** maxElemento;  
}  
  
**int** main() {  
 std::vector<**int**> numeros = { 10, 5, 8, 15, 3, 20, 12 };  
 **int** maximoElemento = encontrarMaximo(numeros);  
 std::cout << "El máximo elemento es: " << maximoElemento << std::endl;  
 **return** 0;  
}

**Complejidad**

1. **Introducción**

Un algoritmo es un método para resolver un problema, debe presentarse como una secuencia ordenada de instrucciones que siempre se ejecutan en un tiempo finito y con una cantidad de esfuerzo también finito. En un algoritmo siempre debe haber un punto de inicio y uno de terminación, estos deben ser únicos y deben ser fácilmente identificables (*LUDA UAM-Azc.*, 2024).

Todo algoritmo debe cumplir con las siguientes características:

1. Debe ser ***Preciso***; es decir, debe especificar sin ambigüedad el orden en quese deben ejecutar las instrucciones.
2. Debe ser ***Definido***; esto es, que cada vez que se ejecute bajo las mismas condiciones, la secuencia de ejecución deberá ser la misma y entregar los mismos resultados.
3. Debe ser ***Finito***; es decir, que siempre se realizarán un número finito de instrucciones en un tiempo y esfuerzo finitos.
4. **Definición de Complejidad Algorítmica**

Existen muchas alternativas de solución para un problema, debemos seleccionar el algoritmo más eficiente con el mejor conjunto de pasos, que su tiempo de ejecución sea el menor y cuyas líneas de código sean las menos posibles.

A simple vista parece algo muy simple, pero a medida que un programa crece, se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la eficiencia teórica del programa, al estudio de estos casos se denomina ***Complejidad Algorítmica***.

La ***complejidad algorítmica*** representa la cantidad de recursos (temporales y espaciales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo, pero son medidas relativas al tamaño del problema.  
Un algoritmo será más eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.  
**1. Complejidad Temporal:** Tiempo de cómputo necesario para ejecutar algún programa.

**2. Complejidad Espacial:** 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.

Nosotros estudiaremos las complejidades temporales, con este fin, para cada problema determinaremos una medida N, que llamaremos tamaño de la entrada o número de datos a procesar por el programa, intentaremos hallar respuestas en función de dicha N.  
Esto dependerá de la naturaleza del problema, por ejemplo, si hablamos de un arreglo se puede ver a N como el rango del arreglo, para una matriz, el número de elementos que la componen; para un grafo, podría ser el número de nodos o arcos, no se puede establecer una regla para N, pues cada problema tiene su propia complejidad.

La familia ***O(f(n))*** representa un ***Orden de Complejidad*** de donde ***f(n)*** es la función más sencilla de la familia.

Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la muestra de entrada ***n***, ordenadas de mayor a menor eficiencia son:

Tabla

Descripción generada automáticamente

***Imagen 1*** *Tipos de Complejidad algorítmica*

A continuación, se mostrará su definición de cada tipo de complejidad

|  |  |
| --- | --- |
| Complejidad | Definición |
| O(1) | Complejidad constante. Las instrucciones se ejecutan una vez. |
| O(log n) | Complejidad logarítmica. Aparece en algoritmos como búsqueda binaria. |
| O(n) | Complejidad lineal. Se encuentra en bucles simples con instrucciones constantes. |
| O(n log n) | Complejidad cuasi-lineal. Presente en algoritmos de tipo divide y vencerás. |
| O(n^2) | Complejidad cuadrática. Se da en bucles doblemente anidados. |
| O(n^3) | Complejidad cúbica. Se da en bucles con triple anidación. |
| O(n^a) | Complejidad polinómica (a > 3). El tiempo de ejecución aumenta con n^a. |
| O(2^n) | Complejidad exponencial. Tiempos de ejecución muy altos, poco prácticos. |

Gráfico, Gráfico de superficie

Descripción generada automáticamente

***Imagen 2*** *Grafica de comportamiento de ordenes de crecimiento de algoritmos*

1. **Reglas elementales para el análisis de algoritmos**

Cuando se dispone de distintos algoritmos para resolver un problema, es necesario definir cuál de ellos es el más eficiente. Una herramienta esencial para este propósito es el análisis de algoritmos.  Este análisis suele efectuarse desde dentro del algoritmo hacia afuera, determinando el tiempo requerido por las instrucciones secuenciales y cómo se combinan estos tiempos con las estructuras de control que enlazan las distintas partes del algoritmo. A continuación, se presentan las reglas básicas para el análisis de algoritmos:

**Operaciones elementales:** el coste computacional de una operación elemental es 1 (asignación, en trata/salida, operación aritmética que no desborde el tamaño de la variable utilizada). La mayoría de las instrucciones de un algoritmo son operaciones elementales que están en el orden de *O*(1).

**Secuencia:** en un fragmento de código, su coste computacional será la suma del coste individual de cada una de sus operaciones o fragmentos de código, aplicando la regla anterior. Es decir, si se cuenta con un fragmento de código compuesto de cinco operaciones elementales, el coste de dicho fragmento será *O*(5).

**Condicional *if-else* (decisión):** si *g* es el coste de la evaluación de un condicional *if*, su valor es de  *O*(1) por cada condición que se deba evaluar (considerando siempre el peor caso, en el que se deba  realizar todas las comparaciones dependiendo del operador lógico usado). A esto se le suma el coste  de la rama de mayor valor, sea esta la verdadera o falsa (considerando las anidadas si corresponde),  aplicando las reglas anteriores.

Es decir el coste de un fragmento condicional es del orden de *O*(*g* + *máximo*(rama verdadera,  rama falsa)).

**Ciclo *for*:** El coste de dicho fragmento se considera como una constante. Y el mismo se considera despreciable o de *O* (n) respecto al número de iteraciones del ciclo.

Es decir el coste total estará en el orden de *O*(n \* *O*(1)) lo que significa que es de *O*(n). Si dentro del *F1* tenemos la presencia de otro ciclo del mismo tipo, el coste total estará en el orden de *O*(n \* n \* *O*(1))  lo que significa que es de *O*(n2).

**Ciclo *while*:** en este tipo de ciclos se aplica la regla anterior, solo se debe tener en cuenta que a veces no se cuenta con una variable de control numérica, sino que depende del tamaño de las estructuras con las que se esté trabajando y su dimensión, o del tipo de actividad que se realice dentro de dicho ciclo.

**Recursividad:** el cálculo del coste total de un algoritmo recursivo no es tan sencillo como los casos que vimos previamente. Para realizar este cálculo existen dos técnicas comúnmente utilizadas:  ecuaciones recurrentes y teoremas maestros. Para realizar la primera de ellas se busca una ecuación que elimine la recursividad para poder obtener el orden de dicha función, muchas veces esto no es una tarea sencilla. En cambio, el segundo utiliza funciones condicionales y condiciones de regularidad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este libro por lo que solo se explicarán algunos ejemplos utilizando la técnica de ecuaciones recurrentes para determinar el orden de funciones recurrentes.

**3 Ejemplos de complejidad**

Tenemos el siguiente ejemplo donde vamos a analizar la complejidad

**Ejemplo 1:  
Complejidad Lineal**

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

**int** cont\_imp = 0; // O(1)

**for** (**int** i = 1; i <= numero; ++i) { // O(n)

**if** (i % 2 == 0) { // O(1)

std::cout << i << " Es Par" << std::endl; // O(1)

} **else** { // O(1)

std::cout << i << " Es Impar" << std::endl;// O(1)

cont\_imp++; // O(1)

}

}

std::cout << "Cantidad de Números Impares: " << cont\_imp << std::endl; // O(1)

}

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).

Tenemos:

O(1) + O(n) \* (O(1) + O(1) + O(1) + O(1)) + O(1) = O(n)

Por lo tanto, la complejidad total del programa es O(n), donde n es el número ingresado como parámetro. Esto significa que el tiempo de ejecución del algoritmo crecerá linealmente con el valor de entrada numero.

**Ejemplo 2:**

**Complejidad Cuadrática**

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

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

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

std::cout << "(" << i << ", " << j << ") "; // O(1)

}

std::cout << std::endl; // O(1)

}

}

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.

Tenemos:

O(n) \* (O(n) \* O(1) + O(1)) = O(n^2)

Por lo tanto, la complejidad total del programa es O(n^2), donde n es el valor de entrada. Esto significa que el tiempo de ejecución del algoritmo crecerá cuadráticamente con el valor de entrada n.

**Ejemplo 3:**

**Complejidad algorítmica**

**int** potencia\_logaritmica(**int** base, **int** exponente) { // O(1)

**if** (exponente == 0) // O(1)

**return** 1; // O(1)

**int** mitad\_potencia = potencia\_logaritmica(base, exponente / 2); // O(log n)

**int** resultado = mitad\_potencia \* mitad\_potencia; // O(1)

**if** (exponente % 2 == 1) // O(1)

resultado \*= base; // O(1)

**return** resultado; // O(1)

}

El análisis de complejidad de tu función **potencia\_logaritmica** es diferente al ejemplo del bucle anidado, pero vamos a desglosarlo:

1. La verificación **if (exponente == 0)** y la operación de retorno **return 1;** son operaciones de tiempo constante O(1). Esto se debe a que no importa el valor de **exponente**, estas operaciones siempre toman una cantidad constante de tiempo.
2. La llamada recursiva **int mitad\_potencia = potencia\_logaritmica(base, exponente / 2);** tiene una complejidad de O(log n). En cada llamada recursiva, el exponente se reduce a la mitad, lo que resulta en aproximadamente logarítmicamente muchas llamadas recursivas.
3. La multiplicación **int resultado = mitad\_potencia \* mitad\_potencia;** es una operación de tiempo constante O(1).
4. La verificación **if (exponente % 2 == 1)** y la multiplicación adicional **resultado \*= base;** también son operaciones de tiempo constante O(1).
5. La operación de retorno **return resultado;** es una operación de tiempo constante O(1).

Entonces, podemos resumir la complejidad de esta función como la suma de todas estas operaciones:

O(1) + O(1) + O(log n) + O(1) + O(1) + O(1) = O(log n)

Por lo tanto, la complejidad total de tu función **potencia\_logaritmica** es O(log n), donde n es el exponente. Esto significa que el tiempo de ejecución del algoritmo aumentará logarítmicamente con el valor del exponente.

**Ejemplo 4**

**Complejidad factorial**

**unsigned** **long** **long** factorial(**unsigned** **int** n) {

**if** (n == 0 || n == 1) // O(1)

**return** 1; // O(1)

**else**

**return** n \* factorial(n - 1); // O(n) \* O(n-1)

}

Ahora, hagamos el análisis de la complejidad:

1. La llamada recursiva factorial(n - 1) se realiza n veces, ya que el algoritmo calcula el factorial de n multiplicando n por el factorial de n-1.
2. Cada llamada recursiva implica una multiplicación, que es una operación de tiempo constante O(1).
3. Por lo tanto, la complejidad total del algoritmo es O(n!), donde n! es el factorial de n.

Por lo tanto, la complejidad factorial significa que el tiempo de ejecución del algoritmo crecerá factorialmente con el valor de entrada n. Esto hace que los algoritmos con complejidad factorial sean muy ineficientes para valores de entrada grandes.

**Ejemplo 5**

**Complejidad cubica**

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

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

**for** (**int** j = 0; j < n; ++j) { // O(n) \* O(n)

**for** (**int** k = 0; k < n; ++k) { // O(n) \* O(n) \* O(n)

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

}

std::cout << std::endl; // O(1)

}

std::cout << std::endl; // O(1)

}

}

1. El bucle externo se ejecuta n veces.
2. Dentro de cada iteración del bucle externo, hay un bucle interno que se ejecuta n veces.
3. Dentro de cada iteración del bucle interno, hay otro bucle interno que también se ejecuta n veces.
4. Por lo tanto, el número total de iteraciones del bucle más interno es n \* n \* n = n^3.

La complejidad total del algoritmo es O(n^3), donde n es el número ingresado por el usuario. Esto significa que el tiempo de ejecución del algoritmo aumentará cúbicamente con el valor de entrada n, lo que lo hace menos eficiente para valores grandes de n.

**Ejemplo practico de como calcular el tiempo de ejecución de diferentes tipos de complejidad y la respectiva grafica**

Para ello se utiliza Code::Blocks donde se tendrá que cambiar el compilador a 32bits para que el código funcione correctamente (esto se debe a que se utiliza la librería “#include <graphics.h>”, a continuación se mostrar como configurar su Code:: Blocks:

Primero abrimos Stting y Compiler para modificar:

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

**Imagen 3** Configuración de compilador.

A continuación tendrá que descargarse y cargar el compilador de 32bits a Code :: Blocks y colocar en linker settings lo siguiente: -lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**Imagen 4** Configuración de compilador a 32bits

Y ejecutaremos se presentará un fragmento del código donde se podrá observar la diferencia de tiempo en de dos funciones con el mismo propósito, pero con diferente dipo de complejidad (Lineal y Cuadrática)

**función Cuadrática O(n^2)**

vector<**int**> sumandos\_version1(**const** vector<**int**> &numeros, **int** suma) {

**for** (**auto** a : numeros) {

**int** diferencia = suma - a;

**for** (**auto** b : numeros) {

**if** ((b != a) && (b == diferencia))

**return** {a, b};

}

}

**return** {0, 0};

}

**Función Lineal O(n)**

vector<**int**> sumandos\_version2(**const** vector<**int**> &numeros, **int** suma) {

set<**int**> conjunto;

**for** (**auto** a : numeros) {

**int** diferencia = suma - a;

**if** (conjunto.find(diferencia) != conjunto.end())

**return** {a, diferencia};

**else**

conjunto.insert(a);

}

**return** {0, 0};

}

**Ejecución**

**Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

**Imagen 5** Compilación función O(n^2) **Imagen 6** Compilación función O(n)

Donde podemos observar que el tiempo de ejecución de la función de complejidad O(n^2) es mayor a la función de complejidad O(n).

**7.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**

#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 3**

#include <iostream>

**int** potencia\_logaritmica(**int** base, **int** exponente) {

**if** (exponente == 0)

**return** 1;

**int** mitad\_potencia = potencia\_logaritmica(base, exponente / 2);

**int** resultado = mitad\_potencia \* mitad\_potencia;

**if** (exponente % 2 == 1)

resultado \*= base;

**return** resultado;

}

**int** main() {

**int** base, exponente;

std::cout << "Ingrese la base: ";

std::cin >> base;

std::cout << "Ingrese el exponente: ";

std::cin >> exponente;

**int** resultado = potencia\_logaritmica(base, exponente);

std::cout << "El resultado de " << base << " elevado a " << exponente << " es: " << resultado << std::endl;

**return** 0;

}

**Ejemplo 4**

#include <iostream>

**unsigned** **long** **long** factorial(**unsigned** **int** n) {

**if** (n == 0 || n == 1)

**return** 1;

**else**

**return** n \* factorial(n - 1);

}

**int** main() {

**unsigned** **int** numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

**unsigned** **long** **long** resultado = factorial(numero);

std::cout << "El factorial de " << numero << " es: " << resultado << std::endl;

**return** 0;

}

**Ejemplo 5**

#include <iostream>

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

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

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

**for** (**int** k = 0; k < n; ++k) {

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

}

std::cout << std::endl;

}

std::cout << std::endl;

}

}

**int** main() {

**int** numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

imprimir\_cubo(numero);

**return** 0;

}

**Ejemplo 6**

**Ejemplo practico**#include <iostream>

#include <vector>

#include <set>

#include <bits/stdc++.h>

#include <sys/time.h>

#include <windows.h>

#include <locale.h>

#include <conio.h>

#include <string>

#include <limits>

#include <graphics.h>

#include <math.h>

**using** **namespace** std;

vector<**int**> sumandos\_version1(**const** vector<**int**> &numeros, **int** suma) {

**for** (**auto** a : numeros) {

**int** diferencia = suma - a;

**for** (**auto** b : numeros) {

**if** ((b != a) && (b == diferencia))

**return** {a, b};

}

}

**return** {0, 0};

}

vector<**int**> sumandos\_version2(**const** vector<**int**> &numeros, **int** suma) {

set<**int**> conjunto;

**for** (**auto** a : numeros) {

**int** diferencia = suma - a;

**if** (conjunto.find(diferencia) != conjunto.end())

**return** {a, diferencia};

**else**

conjunto.insert(a);

}

**return** {0, 0};

}

**void** graficarCuadraticaO\_n2() {

initwindow(900, 900, "Plano Cartesiano", 200, 100);

// Dibujar ejes x e y

setcolor(WHITE);

line(400, 400, 750, 400); // Eje x

line(400, 400, 400, 50); // Eje y

// Etiquetas de los ejes

settextstyle(DEFAULT\_FONT, HORIZ\_DIR, 0);

outtextxy(770, 400, "x");

outtextxy(400, 30, "y");

// Dibujar la línea que representa la complejidad O(n^2) en el primer cuadrante

settextstyle(DEFAULT\_FONT, HORIZ\_DIR, 0);

**int** x2, y2;

**double** c = 8;

**for** (x2 = 0; x2 <= 100; x2++) {

y2 = 400 - c \* pow(2, x2 / 20.0);

**if** (y2 >= 50) { // Asegurarse de que y esté en el primer cuadrante

putpixel(x2 + 400, y2, BLUE); // Desplazar x2 para que esté en el primer cuadrante

}

}

// Etiqueta de la complejidad

setcolor(BLUE);

outtextxy(750, 45, "O(n^2)");

}

**void** graficarLinealO\_n() {

initwindow(800, 800, "Plano Cartesiano", 200, 100);

// Dibujar ejes x e y

setcolor(WHITE);

line(400, 400, 750, 400); // Eje x

line(400, 400, 400, 50); // Eje y

// Etiquetas de los ejes

settextstyle(DEFAULT\_FONT, HORIZ\_DIR, 0);

outtextxy(770, 400, "x");

outtextxy(400, 30, "y");

// Dibujar la línea que representa la complejidad O(n)

setcolor(BLUE);

line(400, 400, 750, 45);

// Etiqueta de la complejidad

outtextxy(750, 45, "O(n)");

}

**int** main() {

**struct** timeval inicio, fin;

**int** totalOpciones = 3;

**int** opcion = 1;

**while** (**true**) {

system("cls"); // Limpiar la pantalla antes de mostrar las opciones

cout << " \n\t COMPLEJIDAD ALGORITMICA? " << endl;

**for** (**int** i = 1; i <= totalOpciones; i++) {

**if** (i == opcion) {

cout << " ";

**if** (i == 1) cout << ">> ";

} **else** {

cout << " ";

}

**switch** (i) {

**case** 1:

cout << "1: Lineal O(n)" << endl;

**break**;

**case** 2:

cout << "2: Cuadratica O(n^2)" << endl;

**break**;

**case** 3:

cout << "3: Salir" << endl;

**break**;

}

}

**int** tecla = \_getch(); // Utilizamos getch() en lugar de \_getch()

**if** (tecla == 224) { // Tecla de flecha

tecla = \_getch(); // Obtenemos la segunda parte del código de la tecla de flecha

**switch** (tecla) {

**case** 72: // Flecha arriba

opcion = (opcion > 1) ? opcion - 1 : totalOpciones;

**break**;

**case** 80: // Flecha abajo

opcion = (opcion < totalOpciones) ? opcion + 1 : 1; // Mover la opción hacia abajo, volver al inicio si está en la parte inferior

**break**;

}

} **else** **if** (tecla == 13) { // Tecla Enter

**if** (opcion == 3) {

system("cls");

cout << "Saliendo del programa..." << endl;

**break**;

} **else** **if** (opcion == 1) {

system("cls");

gettimeofday(&inicio, NULL);

ios\_base::sync\_with\_stdio(**false**);

vector<**int**> resultado = sumandos\_version1({6, 1, 4, 2, 9, 7, 3, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}, 3);

cout << "Los sumandos son: ";

**for** (**int** num : resultado) {

cout << num << " ";

}

cout << endl;

gettimeofday(&fin, NULL);

**double** tiempo\_ejecucion = (fin.tv\_sec - inicio.tv\_sec) \* 1e6;

tiempo\_ejecucion = (tiempo\_ejecucion + (fin.tv\_usec - inicio.tv\_usec)) \* 1e-6;

cout << "Tiempo de ejecucion de la version 1, complejidad O(n): " << fixed << tiempo\_ejecucion << endl;

graficarLinealO\_n(); // Supongo que esta función la tienes definida en otro lugar

system("pause");

} **else** **if** (opcion == 2) {

system("cls");

gettimeofday(&inicio, NULL);

vector<**int**> resultado = sumandos\_version2({6, 1, 4, 2, 9, 7, 3, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}, 3);

cout << "Los sumandos son: ";

**for** (**int** num : resultado) {

cout << num << " ";

}

cout << endl;

gettimeofday(&fin, NULL);

**double** tiempo\_ejecucion = (fin.tv\_sec - inicio.tv\_sec) \* 1e6;

tiempo\_ejecucion = (tiempo\_ejecucion + (fin.tv\_usec - inicio.tv\_usec)) \* 1e-6;

cout << "Tiempo de ejecucion de la version 2, complejidad O(n^2): " << fixed << tiempo\_ejecucion << endl;

graficarCuadraticaO\_n2();

system("pause");

}

}

}

cout << "Presione cualquier tecla para continuar...";

\_getch();

**return** 0;

}

**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:**

**int** suma\_arreglo(**const** std::vector<**int**>& arr) {

**int** suma = 0;

**for** (**int** num : arr) {

suma += num;

}

**return** suma;

}

**Ejercicio 2:**

**bool** buscar\_elemento(**const** std::vector<**int**>& lista, **int** elemento) {

**for** (**int** num : lista) {

**if** (num == elemento)

**return** **true**;

}

**return** **false**;

}

**Ejercicio 3:**

**int** fibonacci\_recursivo(**int** n) {

**if** (n <= 1)

**return** n;

**return** fibonacci\_recursivo(n - 1) + fibonacci\_recursivo(n - 2);

}

**Ejercicio 4:**

**void** ordenamiento\_seleccion(std::vector<**int**>& arr) {

**int** n = arr.size();

**for** (**int** i = 0; i < n - 1; ++i) {

**int** min\_idx = i;

**for** (**int** j = i + 1; j < n; ++j) {

**if** (arr[j] < arr[min\_idx])

min\_idx = j;

}

std::swap(arr[i], arr[min\_idx]);

}

}

**Ejercicio 5:**

**unsigned** **long** **long** factorial\_iterativo(**unsigned** **int** n) {

**unsigned** **long** **long** resultado = 1;

**for** (**unsigned** **int** i = 2; i <= n; ++i) {

resultado \*= i;

}

**return** resultado;

}

**Análisis de Recurrencia**

1. **Introducción**

En el vasto dominio de la ciencia de la computación, el análisis de recursos destaca como una herramienta vital para la eficiencia y la toma de decisiones en el desarrollo de software. Enfocado en algoritmos y estructuras de datos, este análisis desentraña la complejidad computacional, evaluando recursos desde tiempo de ejecución hasta espacio de memoria. Este proceso no es meramente teórico; es una brújula pragmática que guía decisiones cruciales en el diseño de algoritmos, superando desafíos prácticos como simplificaciones teóricas y variabilidades de hardware. Exploraremos estos fundamentos, resaltando su relevancia y desafíos en el contexto real, revelando cómo convergen las decisiones informadas y el diseño eficiente en la intersección de tiempo, espacio y algoritmos.

1. **Definición**

El análisis de recursos, en el contexto de algoritmos y estructuras de datos, se refiere a la evaluación de los recursos computacionales utilizados por un algoritmo. Estos recursos pueden incluir tiempo de ejecución, espacio de memoria, ancho de banda de red u otros recursos relevantes para el problema en cuestión. El objetivo principal del análisis de recursos es entender y cuantificar cómo un algoritmo utiliza estos recursos en función del tamaño de la entrada.

El análisis de recursos es esencial para tomar decisiones informadas sobre la eficiencia de los algoritmos y estructuras de datos, y para diseñar soluciones que optimicen el uso de recursos en aplicaciones del mundo real. Hay dos aspectos principales del análisis de recursos:

1. **Análisis de Tiempo:** Se centra en medir o estimar el tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Esto implica la evaluación del rendimiento temporal en términos de operaciones básicas o complejidad algorítmica. Las relaciones recurrentes y las notaciones como la notación Big O son comúnmente utilizadas en el análisis de tiempo para describir la eficiencia temporal de los algoritmos.
2. **Análisis de Espacio:** Examina la cantidad de memoria o espacio de almacenamiento requerido por un algoritmo en función del tamaño de la entrada. Este análisis es fundamental para determinar la escalabilidad y eficiencia en términos de la utilización de recursos de memoria. La complejidad espacial también se expresa comúnmente utilizando notaciones como la notación Big O.
3. **Proceso**

Para realizar el análisis de recurrencia en estructuras de datos, se sigue generalmente este proceso:

1. **Definición de la recurrencia:** Se establece una ecuación que describe la relación entre el tiempo de ejecución de una operación en la estructura de datos y el tamaño del problema.
2. **Resolución de la recurrencia:** Se intenta resolver la ecuación de recurrencia para obtener una expresión cerrada que represente el tiempo de ejecución en términos del tamaño del problema.
3. **Análisis del resultado:** Se analiza el resultado obtenido para comprender el comportamiento temporal del algoritmo o estructura de datos en función del tamaño del problema.

Este análisis es fundamental para comprender la eficiencia y complejidad temporal de algoritmos y estructuras de datos, ya que proporciona información sobre cómo el rendimiento del algoritmo evoluciona a medida que el tamaño del problema crece.

Es importante destacar que el análisis de recurrencia no solo se aplica a algoritmos y estructuras de datos recursivas, sino que también puede utilizarse en contextos no recursivos donde haya una relación recurrente entre el tamaño del problema y el tiempo de ejecución.

Imagina que estás jugando con bloques de construcción y quieres entender cuánto tiempo te llevará construir una torre de bloques.

**Ejemplo de análisis**

Un ejemplo común en el análisis de recurrencia es la relación para el tiempo de ejecución de algoritmos de ordenación, como el Merge Sort o el QuickSort. Estas relaciones recurrentes suelen tomar la forma de:

*T*(*n*)=*a*⋅*T*(*bn*​)+*f*(*n*)

Donde:

* *T*(*n*) es el tiempo de ejecución del algoritmo para un problema de tamaño n.
* *a* es el número de sub problemas en los que se divide el problema principal.
* *b* es el factor por el cual se reduce el tamaño del problema en cada iteración.
* *f*(*n*) es la función que representa el tiempo necesario para dividir el problema, combinar las soluciones de los sub problemas y cualquier trabajo adicional realizado en cada nivel de recursión.

El análisis de recurrencia es un paso importante para comprender la eficiencia de los algoritmos y estructuras de datos, y puede ayudar a tomar decisiones informadas sobre qué enfoques son más eficientes para problemas particulares.

1. **Ventajas**
2. Diseño Eficiente: Ayuda en el diseño eficiente de algoritmos, permitiendo a los desarrolladores comprender y optimizar el rendimiento en términos de recursos computacionales.
3. Comparación de Algoritmos: Facilita la comparación objetiva de diferentes algoritmos para un problema dado, permitiendo identificar cuál es más eficiente en términos de tiempo y espacio.
4. Escalabilidad: Permite evaluar la escalabilidad de un algoritmo, es decir, cómo se comporta a medida que el tamaño de la entrada aumenta, proporcionando una visión clara de su rendimiento a gran escala.
5. Identificación de Problemas: Ayuda a identificar y corregir posibles problemas de rendimiento antes de implementar un algoritmo en un entorno de producción.
6. Comunicación Efectiva: Facilita la comunicación entre los desarrolladores al proporcionar una métrica objetiva para discutir y comparar el rendimiento de los algoritmos.
7. **Desventaja**
8. Simplificaciones Teóricas: A menudo, el análisis de recursos se basa en simplificaciones teóricas y asume un modelo de máquina abstracta, lo que puede no reflejar con precisión las complejidades del hardware y del entorno de ejecución real.
9. Variabilidad del Hardware: El rendimiento real puede variar según el hardware subyacente, el compilador utilizado y otros factores específicos de la implementación.
10. Omisión de Factores Constantes: La notación Big O y otras formas de análisis asintótico tienden a ignorar factores constantes, lo que significa que dos algoritmos con la misma complejidad asintótica pueden tener diferencias significativas en el rendimiento real.
11. No Considera Constantes Ocultas: A veces, la notación Big O no tiene en cuenta constantes ocultas que podrían afectar el rendimiento en situaciones de pequeña escala.
12. Complejidad Espacial vs. Temporal: El análisis de recursos a menudo se centra más en la complejidad temporal y puede pasar por alto la importancia de la complejidad espacial en algunos contextos.
13. No Toma en Cuenta Factores Externos: No considera factores externos como la variabilidad en la entrada de datos o el impacto de operaciones de entrada/salida, que pueden afectar el rendimiento real.
14. **Aplicaciones**
15. Diseño de Algoritmos Eficientes: Ayuda en el diseño de algoritmos que utilizan eficientemente los recursos disponibles, minimizando el tiempo y el espacio requeridos para resolver un problema.
16. Desarrollo de Software: Es esencial en el desarrollo de software para garantizar que las aplicaciones sean eficientes y respondan de manera rápida, especialmente en entornos donde los recursos son limitados.
17. Optimización de Bases de Datos: En el diseño y la optimización de bases de datos, el análisis de recursos es crucial para garantizar consultas rápidas y eficientes.
18. Computación en la Nube: En entornos de computación en la nube, donde los recursos son compartidos y pagados según el uso, el análisis de recursos es fundamental para controlar costos y mejorar la eficiencia.
19. Sistemas Empotrados: En el desarrollo de sistemas empotrados con recursos limitados, el análisis de recursos es crucial para garantizar que el software funcione eficientemente en hardware con restricciones.
20. **Análisis**

El análisis de recursos en algoritmos y estructuras de datos es un tema fundamental en la ciencia de la computación que busca entender y cuantificar cómo los algoritmos utilizan los recursos computacionales, como el tiempo y el espacio, en función del tamaño de la entrada. Este análisis es esencial para diseñar algoritmos eficientes y tomar decisiones informadas en el desarrollo de software. A continuación, se presentan algunos aspectos clave del análisis del tema:

**Importancia del Análisis de Recursos:**

Eficiencia del Software: Permite diseñar software eficiente que resuelva problemas de manera rápida y con un uso óptimo de los recursos disponibles.

Toma de Decisiones: Facilita la toma de decisiones informadas al elegir entre diferentes algoritmos o enfoques para resolver un problema particular.

Optimización: Ofrece una base teórica para la optimización de algoritmos y estructuras de datos, mejorando su rendimiento en situaciones del mundo real.

**Aspectos Clave del Análisis:**

Análisis de Tiempo: Evalúa el tiempo de ejecución del algoritmo en función del tamaño de la entrada. Utiliza notaciones como Big O para expresar la complejidad temporal.

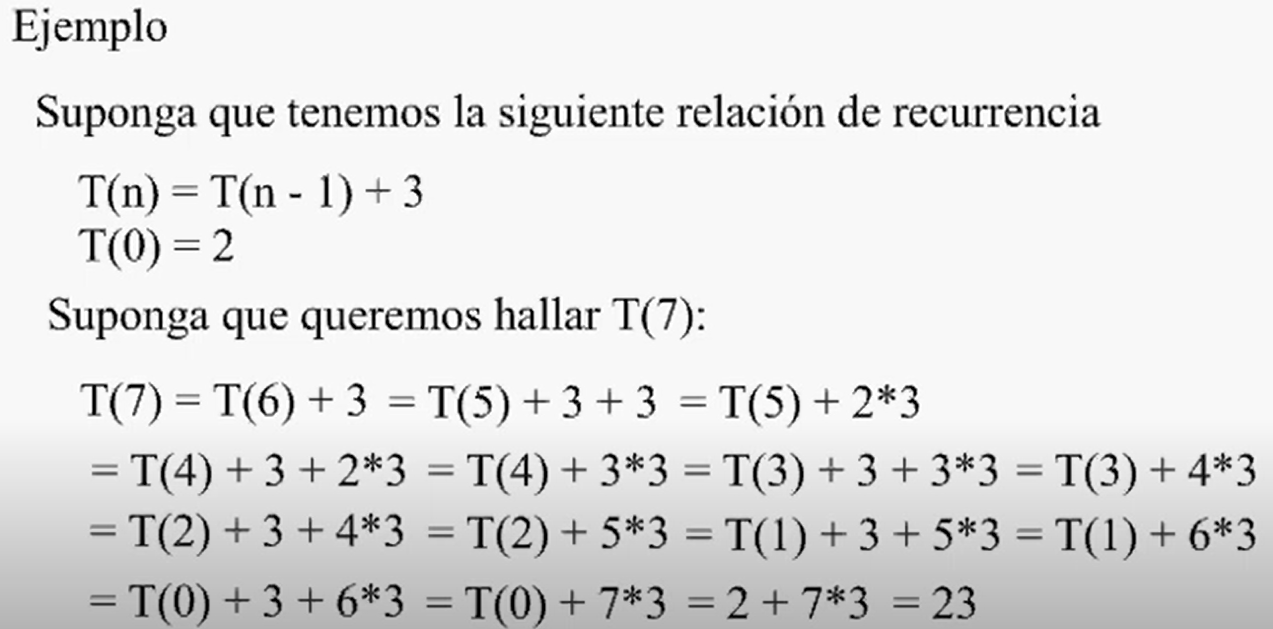
Análisis de Espacio: Examina la cantidad de memoria o espacio de almacenamiento necesario en función del tamaño de la entrada. La complejidad espacial también se expresa con notación Big O.

Comparación de Algoritmos: Permite comparar diferentes algoritmos y seleccionar el más eficiente para un problema específico.

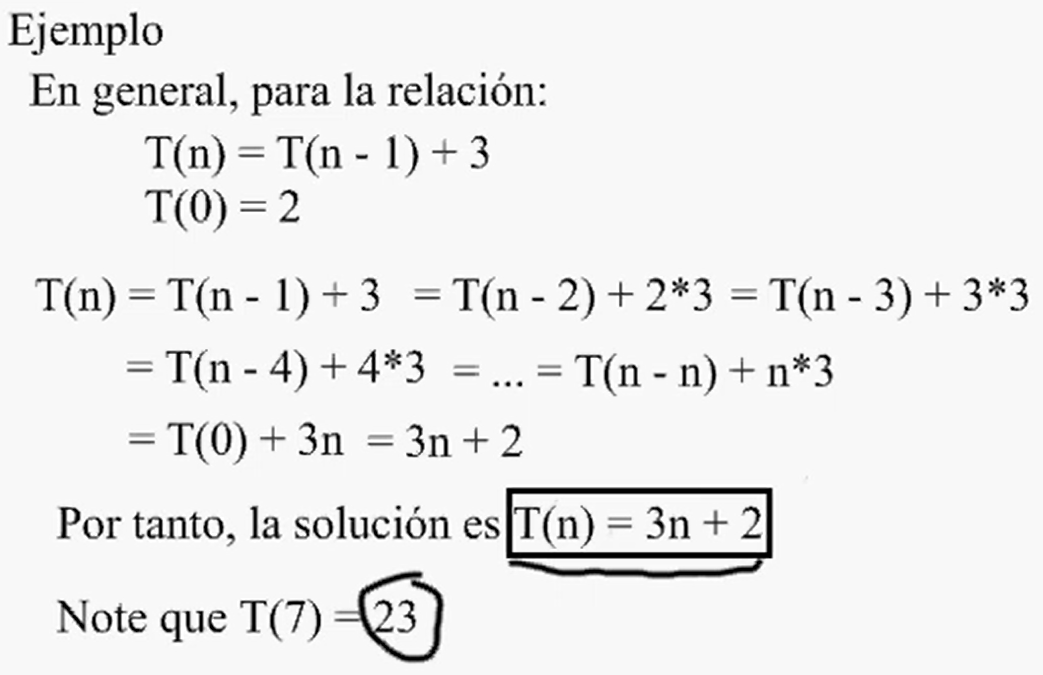
Consideración de Casos Especiales: Analiza el rendimiento en mejores casos, peores casos y casos promedio para comprender el comportamiento del algoritmo en diversas situaciones.

Validación Empírica: Complementa el análisis teórico con pruebas prácticas para validar las predicciones y garantizar que la eficiencia se mantenga en situaciones reales.

1. **Ejemplo – Resolución de una Recurrencia por el Método de Expansión**

****

***Figura 1:*** *Primera Parte de la resolución de una recurrencia por expansión.  
Realizado por Cordova, J. Extraído de: https://www.youtube.com/watch?v=CWiUBI2C9Q0*

****

***Figura 2:*** *Segunda Parte de la resolución de una recurrencia por expansión  
Realizado por Cordova, J. Extraído de: https://www.youtube.com/watch?v=CWiUBI2C9Q0*

1. **Conclusiones**

* Importancia del Análisis de Recursos: El análisis de recursos es fundamental para entender cómo los algoritmos utilizan el tiempo y el espacio, permitiendo tomar decisiones informadas en el diseño de software y la elección de algoritmos.
* Optimización de Algoritmos: A través del análisis de recursos, se puede lograr la optimización de algoritmos, diseñándolos de manera que utilicen eficientemente los recursos disponibles y mejoren el rendimiento general del software.
* Comparación de Algoritmos: Permite comparar diferentes enfoques para resolver un problema, facilitando la elección del algoritmo más adecuado en función de las necesidades específicas y las restricciones de recursos.
* Escalabilidad: El análisis de recursos proporciona información sobre cómo el rendimiento de un algoritmo escala a medida que aumenta el tamaño de la entrada, lo que es crucial para aplicaciones que deben manejar conjuntos de datos cada vez más grandes.
* Validación Empírica: Complementar el análisis teórico con pruebas prácticas es esencial para garantizar que las predicciones sean consistentes en situaciones del mundo real y para confirmar la eficiencia del software.

1. **Recomendaciones:**

* Considerar Factores Prácticos: Aunque el análisis teórico es valioso, es importante considerar factores prácticos como la implementación específica, la arquitectura del hardware y el entorno de ejecución, ya que estos pueden influir en el rendimiento real.
* Actualización Constante: Dado que la tecnología y las plataformas evolucionan, es recomendable actualizar regularmente el análisis de recursos para asegurarse de que las decisiones de diseño sigan siendo válidas y eficientes en contextos cambiantes.
* Uso de Herramientas de Perfilado: Para validar las predicciones teóricas, se recomienda utilizar herramientas de perfilado para realizar un seguimiento del rendimiento real del software en diferentes situaciones y entornos.
* Experimentación en Diversas Escalas: Realizar experimentos con conjuntos de datos de diferentes tamaños y enfoques de entrada para comprender completamente el comportamiento del algoritmo en diversas escalas.
* Documentación Clara: Documentar el análisis de recursos junto con las decisiones de diseño tomadas proporcionará una guía valiosa para el mantenimiento futuro y para otros desarrolladores que trabajen en el proyecto.

**Bibliografia**

* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
* Skiena, S. S. (2008). The Algorithm Design Manual.
* Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.
* Sedgewick, R., & Wayne, K. (2016). *Computer science: An interdisciplinary approach*. Addison-Wesley Professional.
* Javier Cordova. (2015, November 29). *solucion de relaciones de recurrencia* [Video]. YouTube. <https://www.youtube.com/watch?v=CWiUBI2C9Q0>
* *LUDA UAM-Azc.* (2024). Aniei.org.mx. <http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_01.html>