COMPLEJIDAD ALGORÍTMICA

Mide la eficiencia de un algoritmo en relación al **tiempo** o al **espacio** que demanda su ejecución, sin tener en cuenta factores como velocidad del procesador o del disco, compilador usado, etc., ni el lenguaje utilizado. La complejidad permite analizar **cómo se comporta** un algoritmo al incrementar la **cantidad de datos de entrada (N)**.

"Big-O": notación usada para expresar la complejidad de un algoritmo

- **Complejidad constante:** no depende de N. Ejemplo:

```
int cantidad=0;
```

O(1)

- **Complejidad lineal:** el tiempo de ejecución es directamente proporcional a N. Si N se duplica (N*2), también se duplica el tiempo de ejecución. Ejemplo:

```
for (int i=0; i<N; i++)
    cantidad=0;
```

O(N)

- **Complejidad cuadrática:** proporcional a N^2. Si N se incrementa, el tiempo de ejecución se incrementa en N*N. Ejemplo:

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        cantidad=0;
```

$O(N^2)$

- **Complejidad logarítmica:** proporcional al logaritmo de N, o la cantidad de veces que N puede dividirse por 2. Ejemplo:

```
for (int i=N; i>0; i/=2)
    cantidad=0;
```

O(log N)

- **Complejidad exponencial:** el tiempo de ejecución crece en gran medida ante un incremento pequeño de N. Ejemplo:

```
int Fibonacci (int N) {
    if (N==1 || N==2)
        return 1;
    return Fibonacci(N-1)+Fibonacci(N-2);
}
```

$O(2^N)$

PROGRAMACIÓN DESDE CERO

CALCULAR LA COMPLEJIDAD RESPECTO AL TIEMPO

Se contabilizan los pasos elementales del algoritmo, independientemente del lenguaje. La notación *"Big-O"* elimina las constantes y estima el tiempo de ejecución en relación a N, a medida que N se acerca al infinito (peor caso).

"Peor caso": la mayor cantidad de pasos que podría realizar el algoritmo.

Ejemplo:

```
subprograma suma(lista)
    total = 0
    for i desde 0 hasta longitud(lista) - 1
        total += lista[i]
    return total
```

O(N)

• Cada iteración del bucle requiere una suma (para incrementar la variable) y una comparación (para verificar la condición de corte). Si el bucle se ejecuta N veces, se hacen N sumas y N comparaciones ⇨ Esto son *2N*.
• El bucle tiene una operación += que se ejecuta N veces ⇨ Las 2N anteriores + 1N hacen *3N*.
• También hay una inicialización de variable y una instrucción return. Son 2 instrucciones que no dependen de N. Entonces ⇨ **se requieren *3N+2* pasos para completar este algoritmo.**

"Big-O":

❖ La constante 2 se descarta, ya que no influye para grandes valores de N. Quedan 3N.
❖ Como se intenta obtener una estimación, se eliminan los productos (en este caso, el 3). Un producto sólo modifica el ritmo de crecimiento, pero no la forma en que crece el algoritmo *(la función 3N es una función lineal, al igual que la función N)*.
❖ **La complejidad de este algoritmo es O(N).**

PROGRAMACIÓN DESDE CERO