# GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: COMPLEJIDAD ALGORÍTMICA, ALGORITMOS LINEALES

# 1. Introducción

La eficiencia de un algoritmo es muy importante en la programación competitiva. Usualmente es facil diseñar un algoritmo que solucione el problema lentamente, pero el verdadero desafío esta en diseñar un algoritmo que resuelva el problema de la forma más rápida posible. Si el algoritmo solución es lento es posible que solo podamos obtener una parte de los puntos o ninguno. En la siguiente guía veremos algunas reglas y consejos de como calcular la complejidad de nuestros algoritmos lineales.

# 2. Conocimientos previos

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida teórica (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
2. Y otro que ofrece una medida real (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

En el caso nuestro vamos a utilizar la primera variante.

## 2.1. Tamaño de la entrada

El tamaño de la entrada es el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar.

# 3. Desarrollo

La complejidad de un algoritmo es un estimado de tiempo de acuerdo a la entrada. La idea es representar la eficiencia del algoritmo como una función parametrizada donde los parámetros son el tamaño de las entradas del problema.

El tiempo de complejidad de un algoritmo se denota como O(. . . ) donde los tres puntos representan alguna función que usualmente utiliza variables para denominar cada una de los tamaños de las entradas. Por ejemplo $N$ donde dicha variable pueda representar la cantidad de elementos de un arreglo.

A la hora de medir el tiempo, siempre lo haremos en función del número de operaciones elementales que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador,

los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenados que se encuentren ya los datos a ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta. Así suelen estudiarse tres casos para un mismo algoritmo:*caso peor*, *caso mejor* y *caso medio*.

El caso mejor corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones. Análogamente, el caso peor corresponde a la traza del algoritmo que realiza más instrucciones. Respecto al caso medio, corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

### 3.1. Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

1. Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el Principio de Invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.
2. El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.
3. El tiempo de ejecución de la sentencia CASE C OF$v1 : S1|v2 : S2|\ldots|vn : Sn$ END; es $T = T(C) + \max\{T(S_1), T(S_2), \ldots, T(S_n)\}$. Obsérvese que $T(C)$ incluye el tiempo de comparación con $v_1, v_2, \ldots, v_n$
4. El tiempo de ejecución de la sentencia IF C THEN S1 ELSE S2 END; es $T = T(C) + \max\{T(S1), T(S2)\}$.
5. El tiempo de ejecución de un bucle de sentencias WHILE C DO S END; es $T = T(C) + (n^o \quad iteraciones) \times (T(S) + T(C))$. Obsérvese que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.
6. Para calcular el tiempo de ejecución del resto de sentencias iterativas (FOR, REPEAT, LOOP) basta expresarlas como un bucle WHILE.
7. El tiempo de ejecución de una llamada a un procedimiento o función $F(P_1, P_2, \ldots, P_n)$ es 1 (por la llamada), más el tiempo de evaluación de los parámetros $P_1$ , $P_2$ ,. . . , $P_n$ , más el

tiempo que tarde en ejecutarse $F$, esto es, $T = 1 + T(P_1) + T(P_2) + \cdots + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.

8. El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente no en esta guía.

9. También es necesario tener en cuenta, cuando el compilador las incorpore, las optimizaciones del código y la forma de evaluación de las expresiones, que pueden ocasionar *cortocircuitos* o realizarse de forma *perezosa* (lazy). En el presente trabajo supondremos que no se realizan optimizaciones, que existe el cortocircuito y que no existe evaluación perezosa.

### 3.2. Clases o tipos de complejidades

A continuación una lista de las complejidades mas comunes que nos podemos encontrar en el diseño de algoritmos:

- O(1): Complejidad **constante** que significa que el algoritmo no de depende del tamaño de la entrada. Por lo general esta complejidad se ve en algoritmos con fórmula que directamente calcula la respuesta.
- O($\log N$): Complejidad **logarítmica** que implica que el algoritmo en el próximo o iteración descarta la mitad del tamaño de la entrada del paso o iteracción actual. Ejemplo de esto es la búsqueda binaria.
- O($\sqrt{N}$): Una complejidad **raiz cuadrádica** es mas lenta que una O($\log N$) pero más rápida que O($N$). Una especial propiedad de la raíz cuadrada es que $\sqrt{N} = N/\sqrt{N}$, entonces la raíz de $\sqrt{N}$ miente, en algunos casos es la mitad del tamaño de la entrada.
- O($N$): Una complejidad **linear** en un algoritmo implica que se tuvo que recorrer todos los elementos de la entrada para dar la respuesta.
- O($N \log N$): Esta complejidad indica que el algoritmo ordeno la entrada porque es la complejidad de los algoritmos de ordenamiento eficientes. Otra posibilidad es que el algoritmo utiliza alguna estructura de datos que por cada operación tiene una complejidad de O($\log N$).
- O($N^2$): Complejidad **cuadrática** en un algoritmo a menudo indica que este presenta dos ciclo anidado como pudiera ser el caso de generar todos los pares de las entradas.
- O($N^3$): Complejidad **cúbica** en un algoritmo a menudo indica que este presenta tres ciclo anidado como es el caso del algoritmo Floyd-Warshall
- O($2^N$): Esta complejidad indica a menudo que el algoritmo itera sobre todos los subconjuntos de los elementos de la entrada
- O($N!$): Esta complejidad indica a menudo que el algoritmo itera sobre todas las permutaciones de los elementos de la entrada

Un algoritmo tiene complejidad polinomial si se expresa como O($n^k$) donde $k$ es una constante. Todas las complejidades vistas anteriormente son polinomiales excepto O($2^N$) y O($N!$). Lo que sucede es que en la práctica el valor de $k$ es usualemente pequeño lo que hace que esas complejidades polinomiales para un algoritmo sean eficientes.

## 4. Aplicaciones

El saber como determinar la complejidad a nuestros algoritmos a partir del tamaño de la entrada y siempre pensando en el peor de los casos nos vas a permitir evitar envios de soluciones algorítmicas que a priori ya podemos determinar que no van a cumplir con las cota de tiempo del problema a resolver. Nos va permitir decidir que algoritmo implementar en caso que contemos con varios algoritmos para solucionar el problema porque seleccionaremos aquellos cuya complejidad no sobrepase el tiempo limite definido para el problema.

## 5. Complejidad

Para calcular la complejidad de un algoritmo es posible chequear que el algoritmo es eficiente para el problema antes implementarlo. Podemos realizar una estimación basado en el factor que las computadoras del 2020 pueden realizar 100 millones ($10^8$) de operaciones en un segundo.

Por ejemplo asumamos que el tiempo limite del problema es un segundo y el tamaño de la entrada es $N = 10^5$. Si diseñamos un algoritmo cuya complejidad es O($N^2$), el algoritmo para una entrada de $10^5$ realizará $10^{10}$ operaciones. Esto podría hacer que el algoritmo se demore al menos un segundo y fracción. Entonces el algoritmo es lento para resolver el problema.

Basado en lo anterior vamos a mostrar una tabla donde a partir del tamaño de la entrada podemos definir la maxima complejidad que podría tener nuestro algoritmo para que pueda ejecutarse en menos de un segundo.

| **Tamaño de la entrada** | **Complejidad** |
|---|---|
| $N \leq [10 \dots 11]$ | O($N!$), O($N^6$) |
| $N \leq [17 \dots 19]$ | O($2^N \times N^2$) |
| $N \leq [18 \dots 22]$ | O($2^N \times N$) |
| $N \leq [24 \dots 26]$ | O($2^N$) |
| $N \leq 100$ | O($N^4$) |
| $N \leq 450$ | O($N^3$) |
| $N \leq 1{,}5 \times 10^3$ | O($N^{2,5}$) |
| $N \leq 2{,}5 \times 10^3$ | O($N^2 \log N$) |
| $N \leq 10 \times 10^3$ | O($N^2$) |
| $N \leq 200 \times 10^3$ | O($N^{1,5}$) |
| $N \leq 4{,}5 \times 10^6$ | O($N \log N$) |
| $N \leq 10 \times 10^6$ | O($N \log \log N$) |
| $N \leq 100 \times 10^6$ | O(1), O($\log N$), O($N$) |

Por ejemplo si el tamaño de la entrada es $N = 10^6$, es probable que el algoritmo solución al problema no supere una complejidad de O($N$) o O($N \log N$). Esta información a veces pasada por alto hace que sea más fácil el diseño de un algoritmo solución o que seleccionemos entre algunos posibles candidatos de algoritmo solución aquel que sea más idonéo.