# INF-221
# Algoritmos y Complejidad
*Documentación*

# Índice general

# 1 Consejos y consideraciones

## 1.1. Forma eficiente de leer un problema

- Leer la entrada y salida del problema.
- Leer el enunciado (eliminar información innecesaria en función del paso anterior).
- Obtener **observaciones** y **propiedades** del problema.
- Pensar en un algoritmo (si es menos de $10^8$ operaciones, es válido).

## 1.2. Estimación de complejidades aceptadas en función de tamaño de entrada

| n | Peor complejidad aceptada |
|---|---|
| $n <= 10$ | $O(n!), O(n^7), O(n^6)$ |
| $n <= 20$ | $O(2^n \cdot n), O(n^5)$ |
| $n <= 80$ | $O(n^4)$ |
| $n <= 400$ | $O(n^3)$ |
| $n <= 7500$ | $O(n^2)$ |
| $n <= 7 \cdot 10^4$ | $O(n\sqrt{n})$ |
| $n <= 5 \cdot 10^5$ | $O(n \log n)$ |
| $n <= 5 \cdot 10^6$ | $O(n)$ |
| $n > 10^8$ | $O(\log n), O(1)$ |

## 1.3. Técnicas para encontrar propiedades de un problema

1. Resolver desde lo específico a lo general.
2. Confiar en tus capacidades y supuestos.
3. Si no se te ocurre nada en 60 minutos, sigue con otra cosa.

## 1.4. Consejos para C++

La librería más utilizada y que engloba todo lo necesario para problemas de programación competitiva es:

```
#include <bits/stdc++.h>
```

También se recomienda definir aliases para tipos de datos recurrentes con el propósito de agilizar la escritura. Por ejemplo, algunas definiciones pueden ser

```
#define ll long long
#define ld long double
#define vvi std::vector<std::vector<int>>
/*...*/
```

A modo de optimizar, hay configuraciones que se pueden utilizar al inicio de cada programa:

```
std::ios_base::sync_with_stdio(false);
std::cin.tie(NULL);
std::cout.tie(NULL);
```

Para pruebas rápidas de samples, es posible usar el mismo ejecutable junto a un archivo de texto con las entradas esperadas.

```
bashiee@benjitooo $ g++ program.cpp
bashiee@benjitooo $ ./a.out < sample.txt
```

## 1.5. Consejos para programación dinámica

### Cuándo NO usar DP

- **NO hay subestructura óptima**: No hay subproblemas con soluciones óptimas.
- **No hay superposición de problemas**: No hay subproblemas que se resuelvan más de una vez.
- **Hay un greedy que funciona.**
- **Espacio/Tiempo excesivo para las dimensiones del problema.**
- **Problemas NP-Completos sin restricciones.**

### Metodología sistemática

1. **¿Qué quiero calcular? (costos, maximizar, minimizar...).**
2. **¿De qué variables depende?**
3. **Calcular complejidad (Espacio y tiempo).**
4. **¿Es muy costoso? (optimizar, volver al paso 2).**

### Patrones comúnes

- **DP en 1D:** Subproblemas dependen de estados anteriores inmediatos (fibonacci, coin, stairs...).
- **DP en 2D:** Subproblemas dependen de dos parámetros (knapsack, LCS...)

### Errores comúnes

- **Olvidar restricciones (estados inválidos).**
- **Índices incorrectos (límites del problema).**
- **Estados insuficientes (No se considera toda la información).**

### Consejos clave

- **Buscar opciones recursivas naturales:** ¿Qué pasaría si existiera una función mágica? ¿Cuál sería la última decisión por tomar?
- **Identificar patrones de repetición:** ¿Los mismos subproblemas aparecen más de una vez?
- **Definir un estado por completo:** ¿Qué información se necesita para definir un subproblema por completo?
- **Considerar casos bases:** ¿Cuándo termina la recursión?
- **Empezar con casos pequeños:** Resolver manualmente casos pequeños. Buscar patrones.
- **Pensar en los últimos elementos**: ¿Cómo se llegó a este estado?

# 2 Complejidad

## 2.1. Desempeño de algoritmos

Los algoritmos no tienen un rendimiento fijo para cualquier tipo de entrada. Generalmente, hay situaciones donde se comportan de mejor o peor manera dependiendo de la noción de la cual trabajan. En el análisis y diseño de algoritmos se tienen en cuenta los siguientes casos para evaluar el rendimiento de un algoritmo.

### Mejor caso

Caso donde el algoritmo funciona de la mejor manera. Por ejemplo, el mejor caso de algunos algoritmos de ordenamiento se considera el que la entrada ya esté ordenada.

### Caso promedio

Caso donde el algoritmo posee un tiempo de ejecución esperado dentro una distribución de entradas. Para comprobarlo, se necesita una hipótesis probabilística.

### Peor caso

Caso donde el algoritmo trabaja de peor forma. Es la situación que se utiliza generalmente para las notaciones asíntoticas. Un ejemplo es una entrada inversamente ordenada para los algoritmos de ordenamiento.

### Caso amortizado

Costo promedio de **una operación** en una secuencia larga de operaciones que no depende de la distribución de la entrada. Un ejemplo es el método *push_back* para la clase *vector* de *C++*.

## 2.2. Invariante de ciclo

Una técnica que se utiliza a menudo para demostrar la correctitud de un algoritmo es usar un **invariante de ciclo.** Un *invariante* es una afirmación o predicado sobre un conjunto de variables relacionadas que es **siempre verdadero**. La idea es que este invariante sea **constante** en cada iteración y que explique la relación entre las variables de entrada y salida del ciclo.

### Etapas del invariante de ciclo

**Inicialización:** El invariante debe ser verdadero **antes de empezar** el ciclo (caso base).

**Mantención:** El invariante es verdadero **antes de cada iteración del ciclo** y se mantiene verdadero hasta **antes de la siguiente iteración.** A menudo se utiliza **inducción** para probar este punto con una *hipótesis inductiva* para luego realizar un *paso inductivo*.

**Finalización:** El invariante debe ser verdadero **después de terminar** el ciclo.

**Conclusión:** Si el invariante es verdadero en cada etapa, el algoritmo es correcto.

## 2.3. Análisis asintótico

Una manera de medir la complejidad de un algoritmo es a través del *análisis asintótico*. Dentro de las dimensiones estudiadas, se consideran dos grandes aspectos:

- Complejidad temporal ($T(n)$): Predicción de operaciones fundamentales en base al tamaño de la entrada.
- Complejidad espacial ($E(n)$): Predicción del costo en memoria en base al tamaño de la entrada.

En base a estos criterios, se definen las siguientes notaciones:

### Notación Big O

La notación *Big O* se relaciona con $T(n)$.

Se utiliza $O(f(n))$ como representación del límite en el peor caso de tiempo de ejecución de un algoritmo en base a un tamaño de entrada $n$.

Formalmente, se define como:

$$T(n) = O(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) \leq c \cdot f(n), \forall n \geq n_0$$

### Notación little o

La notación *little o* se relaciona con $T(n)$.

A diferencia de *Big O*, la notación *little o* permite afirmar que $T(n)$ **siempre** es menor que $c \cdot f(n)$ por muy pequeño que sea $c$ a partir de un $n_0$ y puede hacerse más pequeña de manera arbitraria.

Formalmente, se define como:

$$T(n) = o(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) < c \cdot f(n), \forall n \geq n_0$$

Por otro lado, también se puede intuir a través de esta relación:

$$\lim_{n \to \infty} \left( \frac{T(n)}{f(n)} \right) = 0$$

### Notación Big Omega

La notación *Big Omega* se relaciona con $T(n)$.

Se utiliza $\Omega(f(n))$ como representación de la cota inferior del tiempo de ejecución de un algoritmo para un tamaño de entrada $n$.

Formalmente, se define como:

$$T(n) = \Omega(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) \geq c \cdot f(n), \forall n \geq n_0$$

### Notación little omega

La notación *litte omega* se relaciona con $T(n)$.

Similar a la notación *little o*, la notación *little omega* permite afirmar que $T(n)$ es **estrictamente** mayor que $c \cdot f(n)$ a partir de un $n_0$ sea cual sea el valor de $c$.

Formalmente, se define como:

$$T(n) = \omega(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) > c \cdot f(n), \forall n \geq n_0$$

Paralelamente, se puede intuir a partir de la siguiente relación:

$$\lim_{n \to \infty} \left( \frac{T(n)}{f(n)} \right) = \infty$$

### Notación Big Theta

La notación *Big Theta* se relaciona con $T(n)$.

Se utiliza $\Theta(f(n))$ para un $T(n)$ donde se cumple que $T(n) = \emptyset(f(n))$ y $T(n) = \Omega(f(n))$ para simbolizar la cota superior e inferior en tiempo de ejecución de un algoritmo frente a un tamaño de entrada $n$.

Formalmente, se define como:

$$T(n) = \Theta(f(n)) \quad ssi \quad \exists c_1, c_2, n_0 > 0 \quad t.q \quad c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \forall n \geq n_0$$

## 2.4. Teorema maestro

Para determinar la complejidad de un algoritmo recursivo se utiliza a menudo el **teorema maestro**.

Formalmente, se define la siguiente relación para casos grandes:

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

Parámetros1:
$a$ es el número de llamadas recursivas.
$b$ el factor de disminución del tamaño de entrada (sub-problemas).
$d$ el exponente asociado al trabajo no recursivo.

De esta relación y si se cumple que $a \geq 1, b > 1 y d \geq 0$, se cumple que:

$$T(n) = \begin{cases} O(n^d \log n) & \text{si } a = b^d \\ O(n^d) & \text{si } a < b^d \\ O(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

# 3 Estructuras de datos

# 4 Librería <*algorithm*> C++

apuntes

## 4.1. sección

apuntes

# 5 Algoritmos de Sorting

# 6 Grafos

# 7 Fuerza bruta

# 8 Algoritmos Greedy

# 9 Programación dinámica