Análisis de Algoritmos

Programación de Estructuras de Datos y Algoritmos Fundamentales (TC1031)

M.C. Xavier Sánchez Díaz
mail@tec.mx

Tecnológico de Monterrey

Outline

¿Qué es un *algoritmo*?

Eficiencia de un algoritmo
Introducción

Q: Si tuviéramos dos algoritmos que resuelven el mismo problema. . . ¿cómo sabemos cuál de ellos nos conviene utilizar?

A: Aquél que sea el más *eficiente* (casi siempre)

Ya sea hablando de **tiempo de ejecución** o bien en **uso de espacio en memoria**.

¿Qué es más barato? ¿Tiempo de procesamiento o almacenamiento?

Eficiencia de un algoritmo

Introducción

Q: Si tuviéramos dos algoritmos que resuelven el mismo problema. . . ¿cómo sabemos cuál de ellos nos conviene utilizar?

A: Aquél que sea el más *eficiente* (casi siempre)

Ya sea hablando de **tiempo de ejecución** o bien en **uso de espacio en memoria**.

¿Qué es más barato? ¿Tiempo de procesamiento o almacenamiento?

Q: Si tuviéramos dos algoritmos que resuelven el mismo problema. . . ¿cómo sabemos cuál de ellos nos conviene utilizar?

A: Aquél que sea el más *eficiente* (casi siempre)

Ya sea hablando de **tiempo de ejecución** o bien en **uso de espacio en memoria**.

¿Qué es más barato? ¿Tiempo de procesamiento o almacenamiento?

Tiempo de Ejecución
Introducción

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

Tiempo de Ejecución

Introducción

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

Tiempo de Ejecución

Introducción

El tiempo de ejecución de un algoritmo depende de muchos factores:

1. **Entrada del programa**. No es lo mismo ordenar una lista con 10 elementos que una de 1 millón de elementos.
2. **Calidad del código compilado**. Existen algunas instrucciones más rápidas que otras, y algunos compiladores más veloces que otros.
3. **Implicaciones de hardware**. Algunos procesadores son más rápidos que otros.
4. **Complejidad del problema**. Es más difícil multiplicar dos matrices que ordenar una lista.

La complejidad **temporal** de un algoritmo hace referencia al tiempo requerido por un algoritmo para ejecutarse, expresado con base en una **función** que depende del tamaño del problema.

Podemos usar la misma idea para expresar también su **complejidad espacial**, es decir el **espacio en memoria** que necesita dicho algoritmo…

La complejidad **temporal** de un algoritmo hace referencia al tiempo requerido por un algoritmo para ejecutarse, expresado con base en una **función** que depende del tamaño del problema.

Podemos usar la misma idea para expresar también su **complejidad espacial**, es decir el **espacio en memoria** que necesita dicho algoritmo...

Ejemplo
Complejidad

z = 0;	$\longrightarrow$	1 asignación
for x = 1; x $\leq n$; x++ **do**	$\longrightarrow$	1 asignación + $(n+1)$ comparaciones
for y = 1; y $\leq n$; y++ **do**	$\longrightarrow$	$n(n+2) = n^2 + 2n$
z = z + a[x, y];	$\longrightarrow$	$n \times n = n^2$
end for	$\longrightarrow$	$2n^2$ (incremento + 1 `goto` implícito)
end for	$\longrightarrow$	n (`goto` implícito cuando y sea `false`)
	$\longrightarrow$	$2n$ (incremento + `goto` implícito)
	$\longrightarrow$	1 (`goto` implícito cuando x sea `false`)

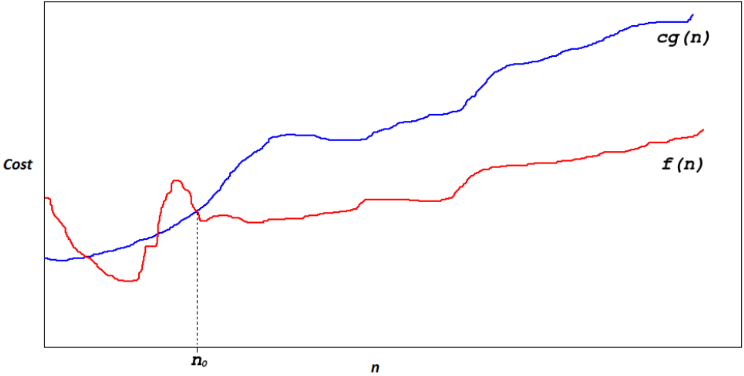
$$\text{Total} = 4n^2 + 6n + 4$$

Notación asintótica: $\mathcal{O}(g)$

Orden Asintótico

Definition 1

Sea $g\colon \mathbb{N} \to \mathbb{R}^+$. $\mathcal{O}(g)$ es el conjunto de funciones $f\colon \mathbb{N} \to \mathbb{R}^+$ tal que para cualquier constante $c \in R^+$ y alguna $n_0 \in \mathbb{N}$, $f(n) \leq cg(n) \forall n \geq n_0$.



Ejemplos

Notación asintótica

- $g = n + 5$ pertenece al grupo de funciones $\mathcal{O}(n)$ pues $n + 5 \leq 2n$ para cada $n \geq 5$.
- $g = (n + 1)^2$ es del grupo $\mathcal{O}(n^2)$ porque $(n + 1)^2 \leq 4n^2$ para cada $n \geq 1$.
- $g = n + 1)^2$ no es del grupo $\mathcal{O}(n)$ porque para cualquier $c > 1$ no se cumplirá nunca que $(n + 1)^2 \leq cn$.

Otra manera de verlo
Orden asintótico

Podemos pensar en la notación de $\mathcal{O}(g)$ como un "*a lo mucho*":

$g = n + 5$ crece a lo mucho como n

pues la notación $\mathcal{O}$ impone un **límite superior** (*upper bound*).

Otra manera de verlo

Orden asintótico

Podemos pensar en la notación de $\mathcal{O}(g)$ como un "*a lo mucho*":

$g = n + 5$ crece a lo mucho como n

pues la notación $\mathcal{O}$ impone un **límite superior** (*upper bound*).

Otra manera de verlo

Orden asintótico

Podemos pensar en la notación de $\mathcal{O}(g)$ como un "*a lo mucho*":

$g = n + 5$ crece a lo mucho como n

pues la notación $\mathcal{O}$ impone un **límite superior** (*upper bound*).

Notación asintótica: $\Omega(g)$ y $\Theta(g)$

Orden asintótico

- $\Omega(g)$ se usa como un **límite inferior** (*lower bound*), es decir que una función $f \in \Omega(g)$ es una función que crece al menos tan rápido como g.

- $\Theta(g)$ se usa para definir un grupo de funciones del **mismo orden**, es decir que una función $f \in \Theta(g)$ es una función que crece igual de rápido como g.

Órdenes más comunes

Orden Asintótico

- $\mathcal{O}(1)$ que es constante
- $\mathcal{O}(\log n)$ que es logarítmico
- $\mathcal{O}(n)$ que es *lineal*
- $\mathcal{O}(n \log n)$
- $\mathcal{O}(n^2)$ que es *cuadrático*
- $\mathcal{O}(n^2 logn)$
- $\mathcal{O}(n^m)$ que es *polinomial*
- $\mathcal{O}(m^n)$ que es *exponencial*
- $\mathcal{O}(n!)$ que es *factorial*

Reglas de dedo para análisis de algoritmos

Análisis Práctico

El uso del **orden asintótico** nos ayuda a simplificar el análisis de complejidad de los algoritmos.

Para hacerlo más *sencillo*, lo que suele hacerse es usar la notación de $\mathcal{O}$ debido a que es la que agrega un **límite superior**, quitando coeficientes y quedándonos con alguno de los órdenes comunes:

En el peor de los casos, este algoritmo será, a lo mucho, igual de tardado que este otro

Reglas de dedo para análisis de algoritmos

Análisis Práctico

El uso del **orden asintótico** nos ayuda a simplificar el análisis de complejidad de los algoritmos.

Para hacerlo más *sencillo*, lo que suele hacerse es usar la notación de $\mathcal{O}$ debido a que es la que agrega un **límite superior**, quitando coeficientes y quedándonos con alguno de los órdenes comunes:

En el peor de los casos, este algoritmo será, a lo mucho, igual de tardado que este otro

Reglas de dedo para análisis de algoritmos
Análisis Práctico

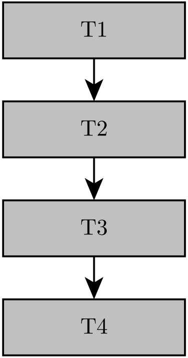
El uso del **orden asintótico** nos ayuda a simplificar el análisis de complejidad de los algoritmos.

Para hacerlo más *sencillo*, lo que suele hacerse es usar la notación de $\mathcal{O}$ debido a que es la que agrega un **límite superior**, quitando coeficientes y quedándonos con alguno de los órdenes comunes:

En el peor de los casos, este algoritmo será, a lo mucho, igual de tardado que este otro

Instrucciones secuenciales

Análisis Práctico



- **Complejidad exacta**: $\Theta(\sum T_i)$
- **Complejidad asintótica**: $\mathcal{O}(\max(T_i))$

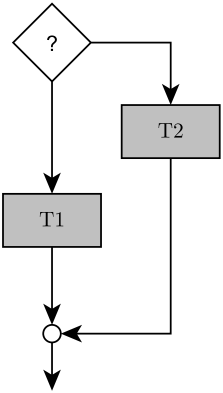
Por ejemplo, una secuencia de 4 instrucciones donde:

- $T_1 = \mathcal{O}(n)$
- $T_2 = \mathcal{O}(\log n)$
- $T_3 = \mathcal{O}(n^2)$
- $T_4 = \Theta(2n^2 + 3)$

tendrá entonces una complejidad de $T = \mathcal{O}(n^2)$

Decisiones

Análisis Práctico

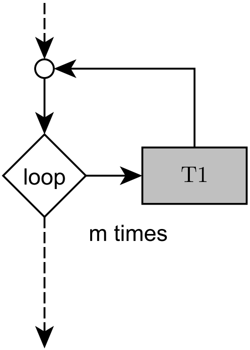


- **Complejidad promedio**: avg(T_1, T_2)
- **Complejidad asintótica**: $\mathcal{O}(\max(T_1, T_2))$

Por ejemplo, una decisión con 2 ramas de instrucciones donde:

- $T_1 = \mathcal{O}(\log n)$
- $T_2 = \mathcal{O}(n \log n)$

tendrá entonces una complejidad de $T = \mathcal{O}(n \log n)$



- **Complejidad exacta**: $\Theta(mT_1)$
- **Complejidad asintótica**: depende

Por ejemplo, asumiendo que $T_1 = \Theta(n)$:

- Si $m << n$ entonces $T = \mathcal{O}(n)$
- Si $m \approx n$ entonces $T = \mathcal{O}(n^2)$

Es muy importante considerar que la notación $\mathcal{O}$ elimina coeficientes, por lo que a veces no podemos ignorar el tamaño de m si $m \not\ll n$.

En el estudio de algoritmos, divide and conquer hace referencia a una técnica de diseño que se enfoca en **partir** un problema en **problemas más pequeños**:

- La idea es juntar las soluciones de cada *subproblema* y obtener así la solución total del problema original
- Esta técnica es muy eficiente cuando se utiliza **recursión**
- La implementación de algoritmos diseñados con esta técnica es ideal para resolverse de manera *paralela*

En el estudio de algoritmos, divide and conquer hace referencia a una técnica de diseño que se enfoca en **partir** un problema en **problemas más pequeños**:

- La idea es juntar las soluciones de cada *subproblema* y obtener así la solución total del problema original
- Esta técnica es muy eficiente cuando se utiliza **recursión**
- La implementación de algoritmos diseñados con esta técnica es ideal para resolverse de manera *paralela*

En el estudio de algoritmos, divide and conquer hace referencia a una técnica de diseño que se enfoca en **partir** un problema en **problemas más pequeños**:

- La idea es juntar las soluciones de cada *subproblema* y obtener así la solución total del problema original
- Esta técnica es muy eficiente cuando se utiliza **recursión**
- La implementación de algoritmos diseñados con esta técnica es ideal para resolverse de manera *paralela*

En el estudio de algoritmos, divide and conquer hace referencia a una técnica de diseño que se enfoca en **partir** un problema en **problemas más pequeños**:

- La idea es juntar las soluciones de cada *subproblema* y obtener así la solución total del problema original
- Esta técnica es muy eficiente cuando se utiliza **recursión**
- La implementación de algoritmos diseñados con esta técnica es ideal para resolverse de manera *paralela*

¿Cuál es la primera idea que viene a tu mente si queremos obtener tanto el máximo como el mínimo de un conjunto?

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k = 1) \\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k = 2) \\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)} \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && \text{(sustituye por } 2^i\text{)} \\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k = 3) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k = 1) \\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k = 2) \\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)} \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && \text{(sustituye por } 2^i\text{)} \\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k = 3) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k = 1) \\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k = 2) \\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)} \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && \text{(sustituye por } 2^i\text{)} \\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k = 3) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k = 1) \\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k = 2) \\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)} \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && \text{(sustituye por } 2^i\text{)} \\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k = 3) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k = 1)\\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k = 2)\\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)}\\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && (\text{sustituye por } 2^i)\\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k = 3)\\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k=1)\\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k=2)\\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)}\\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && (\text{sustituye por } 2^i)\\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k=3)\\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

Con una entrada de $n = 2^k$, y considerando que cada *vuelta* **divides en 2 partes de la mitad del tamaño**, y que en cada vuelta haces **una** comparación entre **dos elementos**, entonces:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 2 && (k=1) \\
&= 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + 2\right) + 2 && (k=2) \\
&= 4T\left(\frac{n}{4}\right) + 4 + 2 && \text{(desarrollando)} \\
&= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2^1 && (\text{sustituye por } 2^i) \\
&= 2\left(2^2T\left(\frac{\frac{n}{2^2}}{2}\right) + 2^2 + 2\right) + 2 && (k=3) \\
&= 2^3T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 && \text{(desarrolla \& sustituye)}
\end{aligned}
$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 \qquad \text{(cont.)}$$

$$\vdots$$

$$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2^1 \qquad \text{(tras } k-1 \text{ pasos)}$$

Y considerando que $n = 2^k$, sabemos que $2^{k-1} = \frac{n}{2}$ así que podemos sustituirlo en la ecuación. Del mismo modo, podemos sustituir la parte en azul por la sumatoria

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

donde $x = 2$ y $n = k - 1$. Es importante notar que la sumatoria va desde 0 hasta n. En nuestro caso, empezamos desde $k = 1$ en adelante, por lo que hay que restarle el primer término: $2^0 = 1$.

Reemplazando entonces la sumatoria, n por 2^k, tenemos

$$= 2^{k-1}T\left(\frac{2^k}{2^{k-1}}\right) + \frac{2^k - 1}{2 - 1} - 1 \qquad \text{(sust.)}$$

$$= \frac{n}{2}T(2) + \frac{n-1}{1} - 1 \qquad \text{(sust.)}$$

$$= \frac{n}{2}(1) + n - 2 \qquad \text{(sust.)}$$

$$= \frac{n}{2} + n - 2$$

$$\therefore \quad T(n) = \mathcal{O}(n)$$

Es un show.

Existe una manera más sencilla de darse cuenta desde el principio dado el tipo de recurrencia.

Supongamos que $n = c^k$ y $c > 1$. Entonces

$$T(n) = \begin{cases} b & \text{si } n = 1 \\ aT\left(\frac{n}{c}\right) + bn^x & \text{si } n > 1 \end{cases}$$

- a es el número de **subproblemas**
- $\frac{n}{c}$ es el tamaño de los **subproblemas**
- bn^x es el costo de la operación realizada en cada vuelta (como partir o unir, comparar, sumar, multiplicar...)

Theorem 2

Sabiendo que $n = c^k$ *y* $c > 1$, *y*

$$T(n) = \begin{cases} b & \textit{si } n = 1 \\ aT\left(\frac{n}{c}\right) + bn^x & \textit{si } n > 1 \end{cases}$$

entonces

$$T(n) = \begin{cases} \Theta(n^{\log_c a}) & \textit{si } a > c^x \\ \Theta(n^x \log_c n) & \textit{si } a = c^x \\ \Theta(n^x) & \textit{si } a < c^x \end{cases}$$

Propiedades de Logaritmos
Equivalencias útiles

1. $\log_b 1 = 0$
2. $\log_b b^a = a$
3. $\log_b xy = \log_b x + \log_b y$
4. $\log_b \dfrac{x}{y} = \log_b x - \log_b y$
5. $\log_b x^a = a \log_b x$
6. $x^{\log_b y} = y^{\log_b x}$
7. $\log_c x = \dfrac{\log_b x}{\log_b c} = (\log_c x)(\log_b c)$

Equivalencias en sumatorias y series

Equivalencias

1. $$\sum_{i=1}^{n} 1 = n$$
2. $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
3. $$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$
4. $$\sum_{i=1}^{n} \frac{1}{i} = \ln n$$
5. $$\sum_{i=1}^{n} ix^i = \frac{(n-1)x^{n+1} - nx^n + x}{(x-1)^2}$$
6. $$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$