Análisis de complejidad en algoritmos

1. Notación asintótica y complejidad

El análisis de algoritmos en ocasiones se enfoca en establecer un panorama general de la tasa de crecimiento del tiempo de ejecución del algoritmo en estudio como una función del tamaño de la entrada n, es decir, a veces es suficiente con decir por ejemplo que el tiempo de ejecución de un algoritmo crece proporcionalmente a n. Para ello, se complementa el análisis del tiempo de ejecución mediante una notación matemática (llamada notación asintótica) aplicada a funciones de modo que, entre otras cosas, se quitan factores constantes; de hecho, se caracteriza el tiempo de ejecución con funciones que corresponden al factor o término principal que determina la tasa de crecimiento en términos de n, eliminando en general detalles innecesarios, como los términos de menor magnitud de la función. Esto refleja el hecho de que cada paso en alto nivel de la descripción del algoritmo corresponde a un número pequeño y fijo de operaciones primitivas, así, se puede estimar el número de operaciones ejecutadas hasta un factor constante, en vez de hacer la medición específica de las operaciones que lleva a cabo la máquina en donde se implemente el algoritmo, así como el hardware o software que tenga.

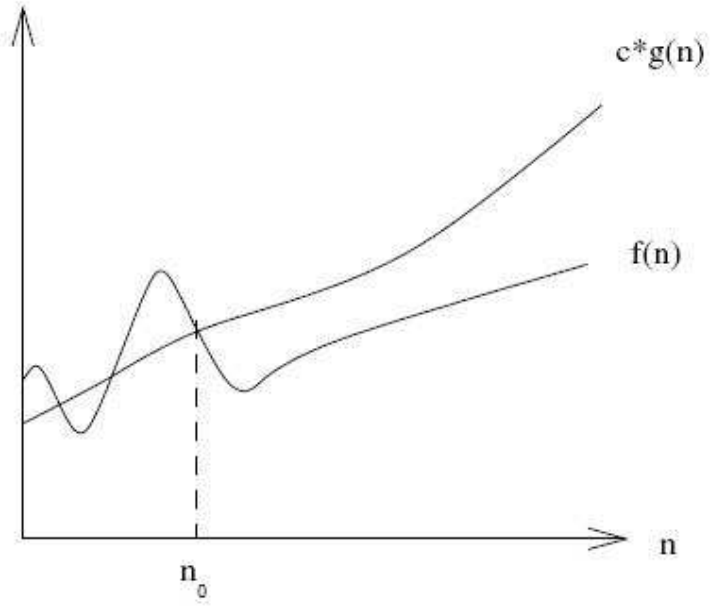
La notación asintótica que usamos en este curso se denomina O-grande y nos dice lo siguiente: si tenemos una función $f(n)$ que indica el tiempo de ejecución de un algoritmo, decimos que el tiempo es del *orden* de una función $g(n)$, lo cual denotamos por $O(g(n))$, si para determinado tamaño de la entrada, el algoritmo tiene un tiempo de ejecución acotado superiormente por la función $g(n)$, multiplicada por una constante no negativa, que depende de la implementación particular del algoritmo. En otras palabras $g(n)$ en algún momento tiene un crecimiento mayor que la función $f(n)$.

Ejemplo: La función $f(n) = 5n^4 + 3n^2 + 4$ es $O(n^4)$. Nos quedamos solamente con el término que influye de manera más significativa en el crecimiento de la función. Es más fácil decir que el timepo de ejecución es del orden de una función polinomial de grado 4 que describen con toda precisión los detalles de $f(n)$.

Formalmente, $f(n)$ es $O(g(n))$ si existe una constante c real y positiva, y un entero no negativo n_0 tal que para todo valor $n \geq n_0$ se cumple que $0 \leq f(n) \leq c \cdot g(n)$.

En este curso no nos enfocaremos a detalle en esta definición, sin embargo, será de utilidad para la asignatura Análisis de Algoritmos. La siguiente figura ilustra de forma geométrica que la función $f(n)$ es del orden de $g(n)$, o más corto, $f(n)$ es $O(g(n))$, a partir de cierto tamaño de entrada n_0, se puede garantizar que la función $f(n)$ queda acotada superiormente por otra función $g(n)$, multiplicada por un factor constante c, que consideramos una constante de implementación, derivada de la imprecisión de

nuestro conteo de operaciones elementales.



La notación asintótica permite determinar una cota superior asintótica al tiempo de ejecución de un algoritmo, lo cual corresponde a la **complejidad del algoritmo en el peor caso**. Para este curso, diremos simplemente que es la **complejidad del algoritmo**.

2. Funciones de complejidad

Son aquellas que representan tiempos de ejecución de algoritmos. Las complejidades (órdenes de magnitud) más comunes que estaremos viendo durante el curso son las siguientes:

- **Complejidad constante**. $O(1)$ son las funciones que tienen un número constante de operaciones, independientemente del tamaño de la entrada.
- **Complejidad logarítmica**. $O(\log n)$. Con esta complejidad aparecen algoritmos que de alguna forma descartan una parte constante de la entrada en cada fase (muy comúnmente la mitad).
- **Complejidad lineal**. $O(n)$. Con esta complejidad aparecen algoritmos que requieren examinar todos los elementos de la entrada de tamaño n.
- **Complejidad superlineal**. $O(n \log n)$. Con esta complejidad, aparecen algoritmos que involucran repetir un cómputo de complejidad logarítmica n veces, donde n es el tamaño de la entrada.

- **Complejidad cuadrática**. $O(n^2)$. Con esta complejidad, por ejemplo, aparecen algoritmos que involucran repetir un cómputo de complejidad lineal n veces, donde n es el tamaño de la entrada.

- **Complejidad exponencial**. $O(b^n)$, con $b > 1$. El caso más común es $b = 2$. Algunos algoritmos de esta complejidad son, por ejemplo, aquellos que requieren examinar todos los subconjuntos que se pueden obtener de la entrada de n elementos.

Vamos a considerar dos reglas para obtener la complejidad en algoritmos, con base en su tiempo de ejecución, que ayudan a simplificar los cálculos de tiempos de ejecución:

Sean A_1, A_2 algoritmos con tiempo de ejecución $f_1(n)$ y $f_2(n)$, respectivamente.

- **Regla de la suma**. Para evaluar la complejidad del algoritmo resultante de ejecutar A_1 seguido de A_2, sumamos los tiempos de ejecución $f_1(n) + f_2(n)$ y la complejidad es $O(f_1(n) + f_2(n))$. Esta regla se aplica para fragmentos independientes de código, uno se ejecuta después del otro.

- **Regla del producto**. Para evaluar la complejidad del algoritmo resultante de ejecutar A_2 dentro de A_1, se multiplican los tiempos de ejecución $f_1(n) \cdot f_2(n)$ y la complejidad es $O(f_1(n) \cdot f_2(n))$. Esta regla se aplica para estructuras repetitivas (ciclos).

3. Ejemplos de análisis de complejidad

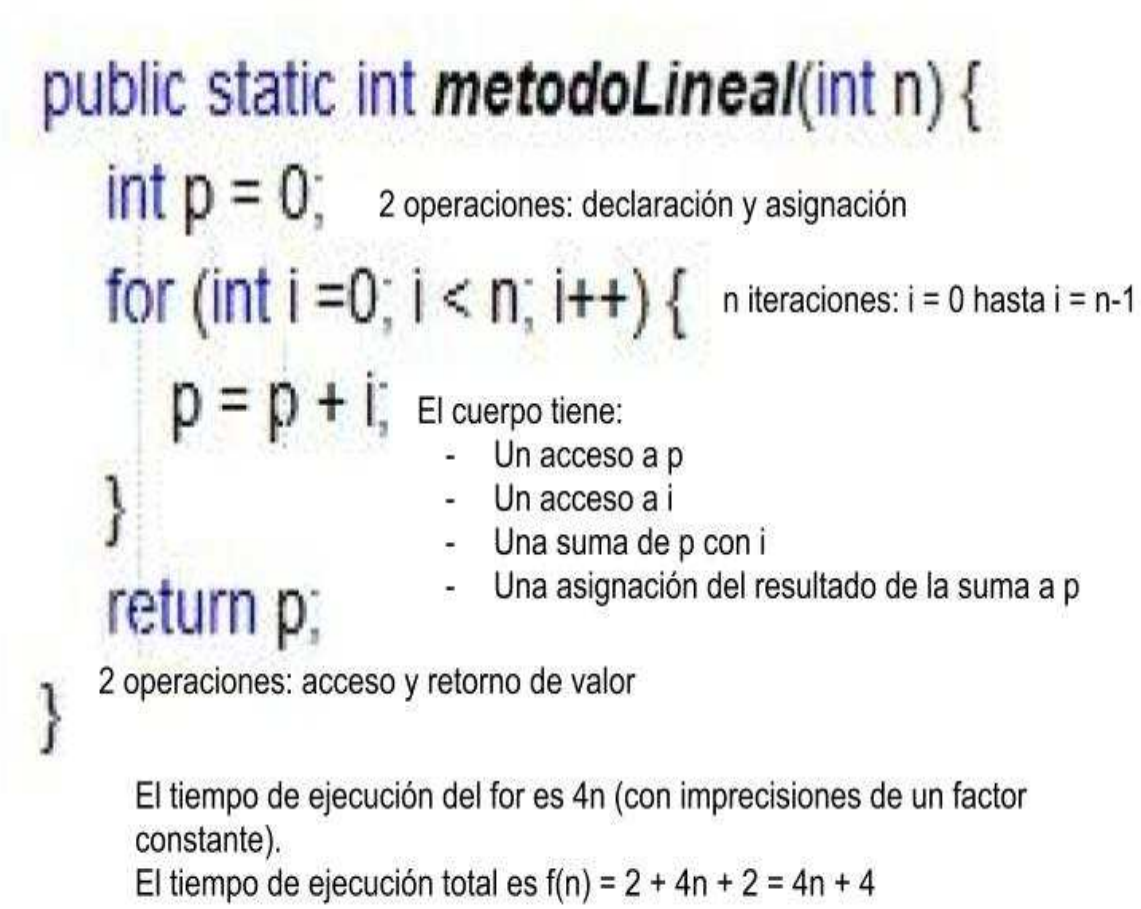
3.1. Estructuras repetitivas

El tiempo de ejecución de un ciclo se calcula empleando la regla del producto de los dos factores siguientes:

- número de iteraciones del ciclo

- tiempo de ejecución de las instrucciones que forman el cuerpo del ciclo

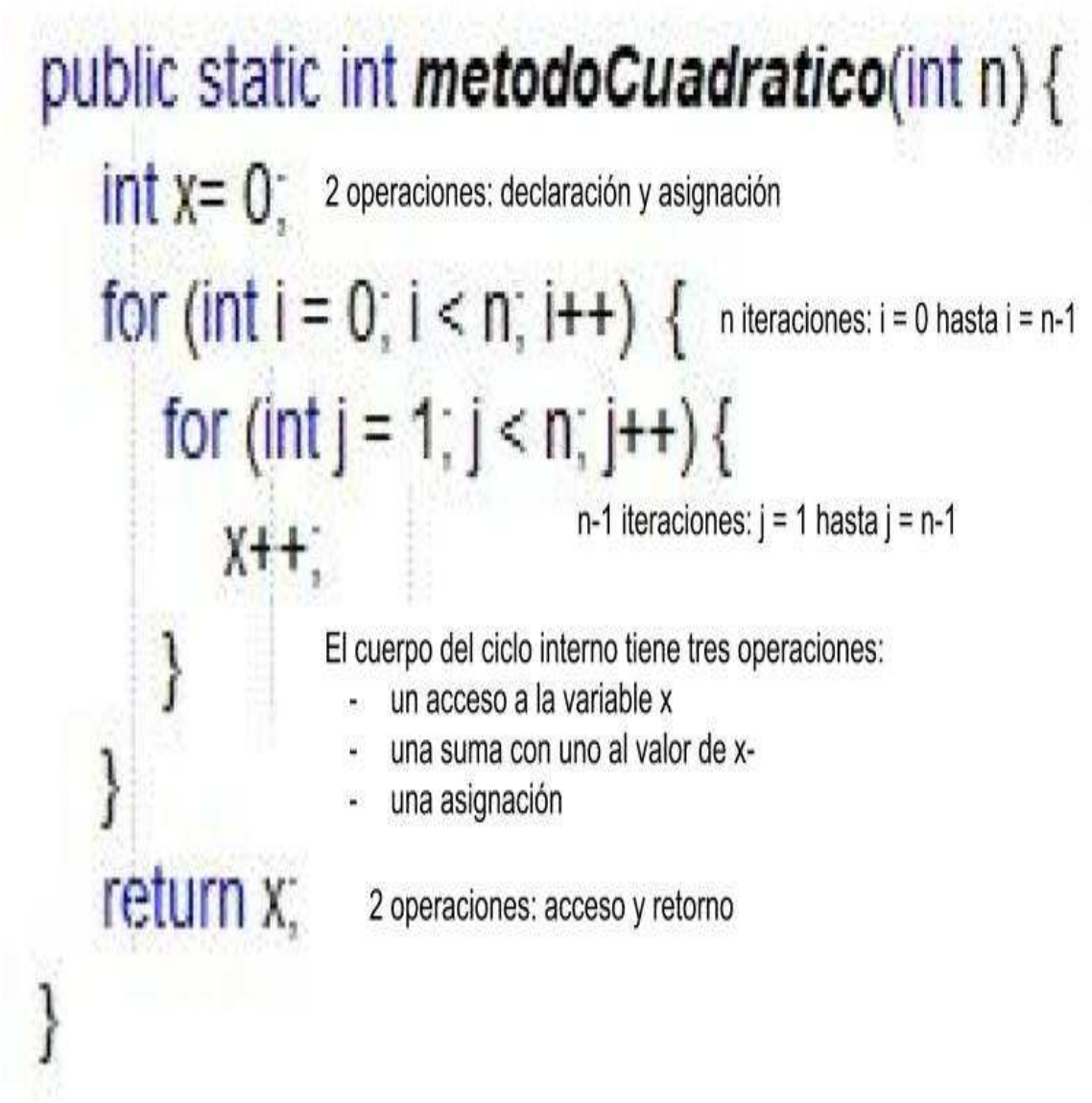
Veamos dos ejemplos sencillos que involucran ciclos.



El tiempo de ejecución de metodoLineal es una función lineal sobre n, es decir $f(n)$ es $O(n)$. No es importante entonces saber cuántas operaciones de comparación e incremento vienen dentro del ciclo for, la regla del producto nos permite omitir esos pequeños detalles de implementación del ciclo.

Cuando se tienen ciclos anidados, el análisis se comienza desde el ciclo interno y el tiempo de ejecución obtenido corresponde al tiempo de ejecución del cuerpo del ciclo inmediato exterior (el que le sigue de adentro hacia afuera). El análisis continúa de este modo hasta terminar con los ciclos anidados.

El siguiente ejemplo tiene dos ciclos anidados.



La parte interesante es la de los ciclos: el ciclo interno tiene $n-1$ iteraciones y el cuerpo tiene 3 operaciones, por tanto, el tiempo de ejecución del ciclo interno es $3(n-1)$.

El ciclo externo tiene n iteraciones y su cuerpo es el ciclo interno, cuyo tiempo es $3(n-1)$, por lo que el tiempo del total es $3n(n-1) = 3n^2 - 3n$.

El tiempo de ejecución total de metodoCuadratico es $2 + 3n^2 - 3n + 2 = 3n^2 - 3n + 4$, lo cual es $O(n^2)$, es decir, la complejidad es cuadrática.