Prof. Robert Espinoza Domínguez

Complejidad Algorítmica

Introducción

- Antes de planear un programa se debe definir los objetivos que deseamos alcanzar a través del programa (requerimientos).
- El proceso de planeación consiste en diseñar un programa que cumpla con los requerimientos.
- Debemos priorizar la actividad de diseño y razonamiento frente a la depuración mediante ejecución.
- Existe la necesidad de especificar formalmente cada una de las etapas del ciclo de vida del desarrollo de un sistema.

Motivación

- Necesitamos exponer nuestra idea a otros programadores.
- Poder hacerle ver a otro colega que esto hace lo que digo.
- Poder dejar claro qué es lo que tiene que hacer el programador.
- Necesitamos especificar formalmente el problema.

Especificación formal

- Consiste en expresar, utilizando cierto mecanismo formal (por ejemplo, un lenguaje matemático-lógico) cuál es el efecto que un algoritmo va a tener sobre determinados datos de entrada.
- Por ejemplo, si nos piden construir un algoritmo que calcule la suma de los N primeros números naturales, una especificación formal podría ser:

$$\forall n \in \mathbb{N}.(Suma(n) = \sum_{i=1}^{n} i)$$

Verificación formal

- Consiste en un conjunto de técnicas de comprobación formales que permiten demostrar si un programa funciona correctamente.
  - **Programa que funciona correctamente:** cumple con las especificaciones dadas.
  - **Técnicas de comprobación:** La verificación consiste en un proceso de inferencia. Por tanto cada tipo de sentencia ejecutable posee una regla de inferencia
  - **Especificación formal:** Ternas de Hoare.

Ternas de Hoare

{P} S {Q}

- S es una parte del código.
- Cualquier aserción o predicado {P} se denomina precondición de S si {P} sólo implica el estado inicial.
- Cualquier aserción o predicado {Q} se denomina **postcondición** si {Q} sólo implica el estado final.
- Su interpretación es:

  “Si P es cierta antes de la ejecución de S y S termina, entonces Q será cierta.”

Precondición y postcondición

- **Precondición:** Es una condición que debe cumplirse antes de la ejecución de una porción de código (subprograma o método). A veces no existe precondición.
- **Postcondición:** Es una condición que debe satisfacerse después de la ejecución de un código o de una operación.
- Ejemplo1: Cálculo del factorial de un número N,

  Precondición: la entrada N debe cumplir la precondición de ser un entero mayor o igual que cero.

  $$\{N \in Z \Lambda n >= 0\}$$

  Postcondición: El resultado es un entero mayor o igual que 1.

  $$\{N \in Z \Lambda n >= 1\}$$

Ternas de Hoare

- Ejemplo 2:

  { y≠0 } x=1/y { x=1/y }

- Ejemplo 3:

  { } a=b { a=b }

  - No existe precondición: { } aserción vacía.
  - Esto significa que siempre se obtienen estados que satisfacen la postcondición independientemente de la precondición.

- Ejemplo 4:

$$P : \{x = X \wedge y = Y \wedge X \geq 0 \wedge Y \geq 0\}$$

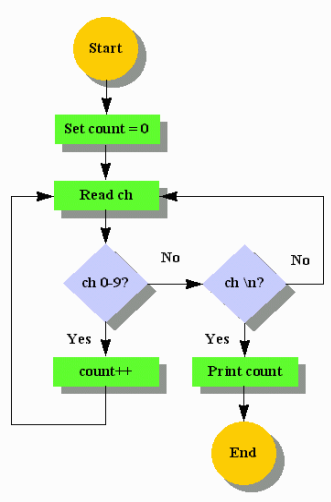
$$S : m = x + y$$

$$Q : \{m = X + Y\}$$

Complejidad algorítmica

Algoritmia

- Es la ciencia que permite evaluar los resultados de diferentes factores externos sobre los algoritmos disponibles, de tal modo que permita seleccionar el que se ajuste más a las condiciones particulares.
- Es la ciencia que nos permite indicar la forma de diseñar un nuevo algoritmo para una tarea concreta.
- Se define como el estudio de los algoritmos.



Algoritmo

- Secuencia finita de pasos ordenados y finitos, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo finito para solucionar un problema.
- Por pasos se entiende el conjunto de acciones u operaciones que se efectúan sobre ciertos objetos.
- Para la solución de un problema se toma en cuenta un algoritmo de un conjunto de algoritmos, dependiendo de las características particulares del problema (algoritmo seleccionado).

Características de un algoritmo

- Un algoritmo debe poseer las siguientes características:
  - **Precisión:** Un algoritmo debe expresarse sin ambigüedad.
  - **Determinismo:** Todo algoritmo debe responder del mismo modo antes las mismas condiciones.

  - **Finito:** La descripción de un algoritmo debe ser finita.

Cualidades de un algoritmo

- Un algoritmo además debe alcanzar los siguientes objetivos, que suelen contradecirse:
  - **Sencillo:** Que sea fácil de entender, codificar y depurar.
  - **Eficiente:** Uso eficiente de los recursos del computador en tiempo, espacio (de memoria) y procesador, para que se ejecute con la mayor rapidez posible
- Por lo general es difícil encontrar un algoritmo que reúna ambas por lo que se debe alcanzar un compromiso que satisfaga lo mejor posible los requisitos del problema.

Cualidades de un algoritmo

- ¿Cuál de los objetivos es más importante?
  - Si el código será usado pocas veces o con pocos datos, entonces es deseable un programa simple y fácil de programar (poco costo de programación)

    > Costo de programación ⇔ **<** Costo de procesamiento

  - Si el código será usado muchas veces o con muchos datos, entonces será mejor que sea eficiente y robusto. (costo de procesamiento > costo de programación)

    < Costo de programación ⇔ **>** Costo de procesamiento

Relación entre la Estructura de Datos y el Algoritmo

- La relación determina un análisis cuantitativo.
- El tipo de estructura de datos elegida es dependiente de:
  - Tipo de datos que administra el algoritmo.
  - Operaciones que se realizan sobre dicha estructura
  - El compromiso espacio de almacenamiento - tiempo de procesamiento que se desea obtener.
- Las estructuras de datos deben ser:
  - Lo suficientemente **complejas** para que representen la relación entre los datos y la realidad.
  - Lo suficientemente **sencillas** para que las operaciones que acceden a los datos, se realicen en forma eficiente

Evaluación de algoritmos

- Para la evaluación de algoritmos se toma en cuenta:
  - Eficacia (fin que se busca). Calidad de la solución obtenida.
  - Eficiencia (tiempo y espacio). Cantidad de recursos computacionales consumidos por el algoritmo. El tiempo y el espacio utilizados en dicha solución miden la mayor o menor eficiencia de un algoritmo.

Complejidad algorítmica

- La complejidad algorítmica representa la cantidad de recursos(temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo.
- No está referido a la dificultad para diseñar algoritmos.
- Los criterios que se van a emplear para evaluar la complejidad algorítmica no proporcionan medidas absolutas sino medidas relativas al tamaño del problema.
- Lo relevante es la medida de la complejidad temporal en términos de los datos de entrada.

Parámetros de la Eficiencia

- Un algoritmo será mas eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.
- Se tomará en cuenta el evaluar a los algoritmos por eficiencia:
  - Tasa de crecimiento en tiempo

    Tiempo de ejecución: Tiene que ver con el tiempo que tarda un programa para ejecutarse (tiempo de procesamiento).
  - Tasa de crecimiento en espacio

    Espacio de memoria: Estudia la cantidad de espacio que es necesario para las operaciones durante la ejecución del programa (espacio de almacenamiento y procesamiento).

Complejidad Temporal y Espacial

- La eficiencia de un algoritmo puede ser cuantificada con las siguientes medidas de complejidad:
  - ***Complejidad Temporal o Tiempo de ejecución:*** Tiempo de cómputo necesario para ejecutar algún programa.
  - ***Complejidad Espacial:*** Memoria que utiliza un programa para su ejecución, La eficiencia en memoria de un algoritmo indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la *memoria estática* de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la *memoria dinámica*, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

Relación Espacio - Tiempo

- El espacio de almacenamiento por lo general es múltiplo de los valores de entrada.
- Como la expansión de la memoria es constante el único factor a tomar en cuenta es el tiempo de procesamiento.

Mas espacio ⟺ Menor tiempo

Menos espacio ⟺ Mayor tiempo

Complejidad algorítmica – medida del tiempo

- La medida del tiempo tiene que ser independiente:
  - de la máquina
  - del lenguaje de programación
  - del compilador
  - de cualquier otro elemento hardware o software que influya en el análisis.
- Para conseguir esta independencia una posible medida abstracta puede consistir en determinar cuantos pasos se efectúan al ejecutarse el algoritmo.

Complejidad algorítmica – medida del tiempo

- El tiempo empleado por el algoritmo se mide en pasos.
- El coste depende del tamaño de los datos.
- A la hora de evaluar el coste se debe de tener en consideración tres posibles casos:
  - El coste esperado o promedio
  - El coste mejor
  - El coste peor
- Si el tamaño de los datos es grande lo que importa es el comportamiento asintótico de la eficiencia.

La complejidad temporal y los datos

- El tiempo requerido por una algoritmo está en función del tamaño de los datos. Por esta razón la complejidad temporal se expresa como **el tiempo de ejecución con una entrada de tamaño n**:

  $$T(n)$$

- Dependiendo del problema, el tamaño del dato representa cosas diferentes:
  - el número en sí
  - el número de dígitos o elementos que lo compone.
- Otra característica importante es que no todos los datos, dentro de un problema, poseen la misma importancia de cara a la complejidad algorítmica.

La complejidad temporal y los datos

- Ejemplo 1: Algoritmo que determina la paridad de un número restando 2 sucesivamente mientras el resultado sea mayor que 1 para finalmente comprobar el resultado.
  - El problema tendrá n / 2 restas (depende de n).
- Ejemplo 2: Algoritmo de suma lenta

  Mientras b>0 hacer

  a←a+1;

  b←b-1;

  Fin Mientras

  En este caso T = T(b).

Caso peor, mejor y promedio

- **Caso Peor ($T_{max}(n)$):** Representa la complejidad temporal en el peor de los casos es decir el máximo valor que puede alcanzar T(n) para cualquier tamaño de entrada y en base al número de operaciones.
- **Caso Mejor ($T_{min}(n)$):** Representa la complejidad en el mejor de los casos posibles, es decir, el valor mínimo que puede alcanzar T(n) para cualquier tamaño de entrada y en base al número de operaciones.
- **Caso Promedio o esperado ($T_{med}(n)$):** Expresa la complejidad temporal en el caso promedio, o sea el valor más esperado o valor promedio que puede alcanzar T(n) para cualquier tamaño de entrada y en base al número de operaciones. Para su cálculo se suponen que todas las entradas son equiprobables.

$$C = (CM + CP) / 2$$

Caso peor, mejor y promedio

- Ejemplo: Búsqueda de un elemento en un vector

```
Función Búsqueda(vector, n, valor)
  i←0
  Mientras (vector[i] < > valor y i<= n) hacer
        i ← i+1;
  Fin mientras
  Si vector[i] = valor entonces
    retornar i
  sino
    retornar -1
  Fin Si
Fin Función
```

Caso peor, mejor y promedio

- En este algoritmo se puede dar las siguientes situaciones:
  - **Caso mejor:** El elemento esté en la primera posición
  - **Caso peor:** Se tenga que recorrer todo el vector.
  - **Caso promedio o esperado:** Puesto que todas la posiciones son equiprobables el tiempo será n/2 pasos.

Notación Asintótica

Notación Asintótica

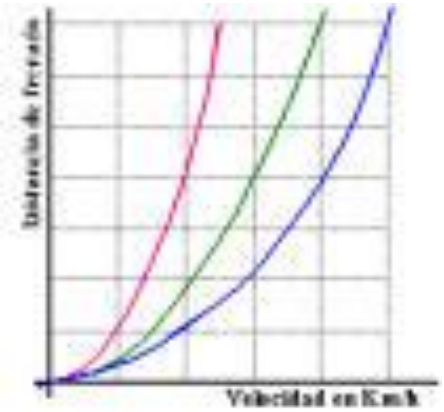
- Sabemos que podemos definir la eficiencia de un algoritmo como una función **t(n)**.
- A la hora de **analizar un algoritmo** nos interesa, principalmente, la **forma en que se comporta el algoritmo al aumentar el tamaño de los datos**; es decir, cómo aumenta su tiempo de ejecución.
- Esto se conoce como **eficiencia asintótica de un algoritmo** y nos permitirá comparar distintos algoritmos puesto que deberíamos elegir aquellos que se comportarán mejor al crecer los datos.
- La notación asintótica se describe por medio de una función cuyo dominio es el conjunto de números naturales, **N**.

Notación asintótica

- El análisis de la eficiencia algorítmica nos lleva a estudiar el comportamiento de los algoritmos frente a condiciones extremas.
- Matemáticamente hablando, cuando *N* tiende al infinito φ, es un comportamiento asintótico.
- Se utiliza esta notación para hacer referencia a la velocidad de crecimiento del tiempo de ejecución cuando crece la cantidad de datos.
- Es mejor elegir el algoritmo cuyo tiempo de ejecución tenga la menor velocidad de crecimiento.

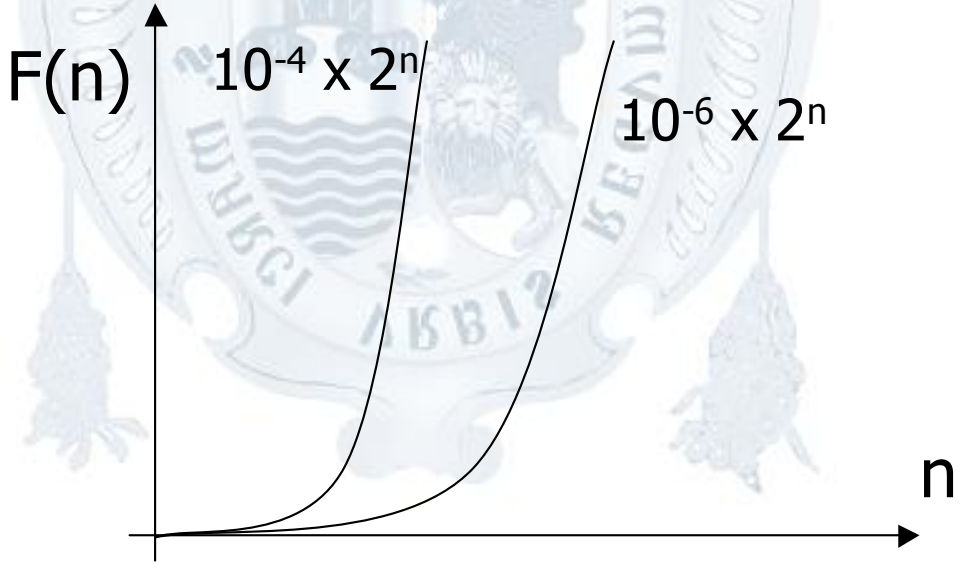
Notación asintótica

- Explorar el comportamiento de una función o de una relación entre funciones, cuando algún parámetro de la función tiende hacia un valor asintótico.
- Se denomina “asintótica” pues trata de funciones que tienden al límite.
- Las notaciones asintóticas nos permiten hallar la tasa de crecimiento del tiempo de ejecución
- Un algoritmo que sea superior asintóticamente es preferible; por que se considera el máximo límite que llega una función.



Notación asintótica

- Ejemplar de un algoritmo con tiempo de $\mathbf{10^{-4} \times 2^n}$ seg.
  - Para n =10 en 1/10 de segundo
  - Para n = 20 en aproximadamente 2 minutos
  - No es necesario esperar a la siguiente generación de computadores.
- Con un nuevo computador aparecerá ejemplar de $\mathbf{10^{-6} \times 2^n}$ segundos



Complejidad asintótica

- Diseño de un nuevo algoritmo con tiempo de $\mathbf{10^{-2} \times n^3}$ seg.
  - Para n = 10 necesita de 10 segundos
  - Para n = 20 necesita de 1 minuto y 20 segundos
  - Para n = 30 necesita de 4 minutos y medio.
- Con un nuevo computador aparecerá ejemplar de $\mathbf{10^{-4} \times n^3}$ segundos
  - Para n = 10 necesita de 0.1 segundos
  - Para n = 20 necesita de 0.8 segundos
  - Para n = 30 necesita de 2.7 segundos

Notación asintótica

- Funciones conocidas($\log_2 n$, n, $n\log_2 n$, n^2, n^3, 2^n).

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	1000	1000
100	7	100	700	10000	10^6	10^{30}
1000	10	1000	10000	10^6	10^9	10^{300}

Notación Asintótica O(n) – O grande de n

- Se describe la **notación O -** límite asintótico superior como:

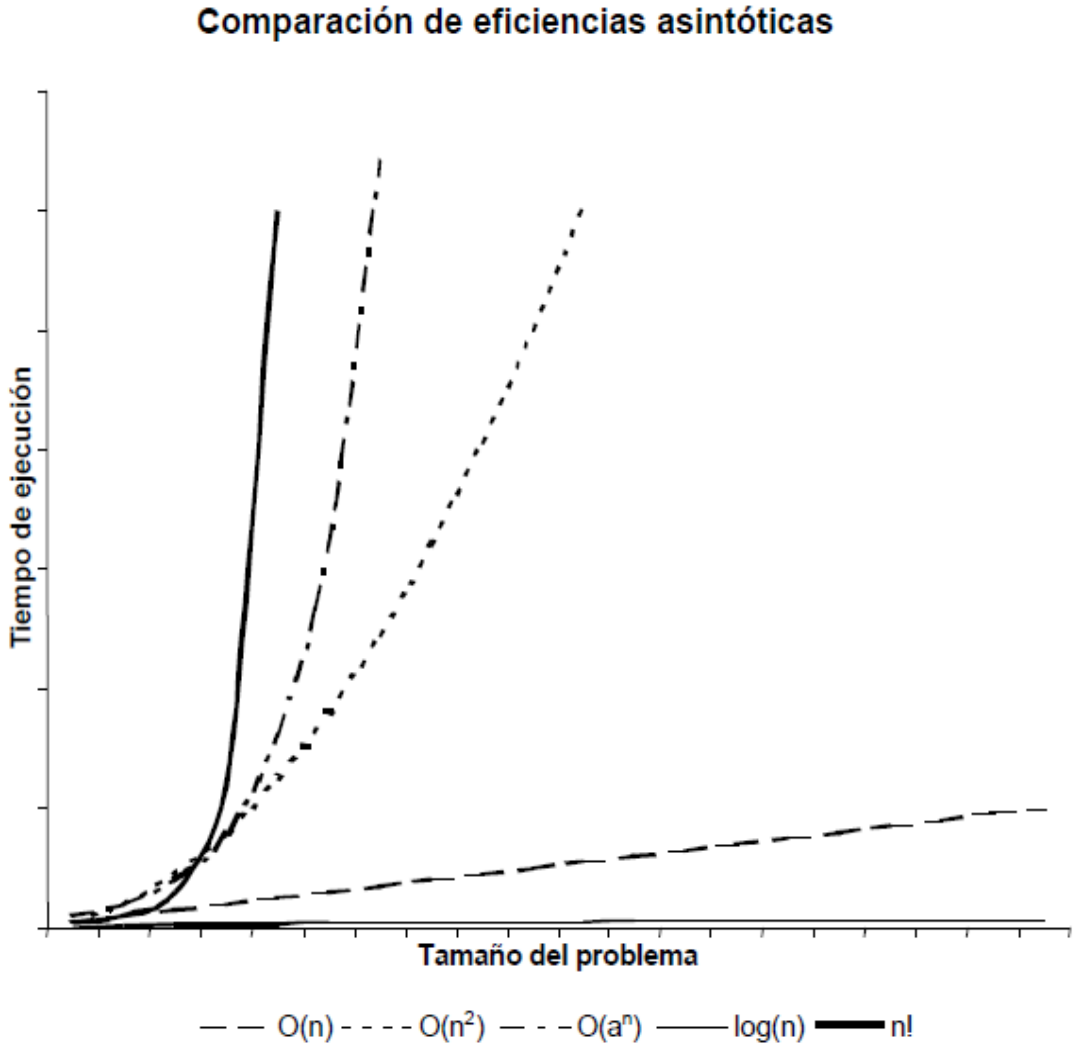
$$O(g(n)) = \{f(n) / \exists c > 0, n_0 > 0, 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$$

$$f(n) : Z^+ \rightarrow R^+$$

$$g(n) : Z^+ \rightarrow R^+$$

- La notación anterior básicamente nos dice que si tenemos un algoritmo cuyo tiempo de ejecución es **f(n)** podemos encontrar otra función de n, **g(n)**, y un tamaño de problema $\mathbf{n_0}$ de tal forma que g(n) acota superiormente al tiempo de ejecución para todos los problemas de tamaño superior a n_0.
- **Esta notación**, **facilitará en las comparaciones de eficiencia entre algoritmos diferentes.**

Notación Asintótica O(n) – O grande de n



- En la gráfica se puede ver distintos órdenes de complejidad “típicos”.
- O(log(n)) crece de manera imperceptible mientras que O(n!), $O(a^n)$ y $O(n^a)$ crecen muy rápidamente.
- La mayor parte de algoritmos son $O(n^a)$, los algoritmos más eficientes son O(log(n)); es muy difícil lograr algoritmos O(1) –tiempo constante.

Notación Asintótica O(n) – O grande de n

- Sean $t(n) = 4n^2 + 6n + 10$ y $f(n) = n^2$ y $n <= n^2$, $1 <= n^2$ siempre y cuando $1 <= n$.

  Por lo tanto para $n >= 1$

  $t(n) = 4n^2 + 6n + 10$

  $<= 4n^2 + 6n^2 + 10n^2$

  $= 20\ n^2$

  $= 20\ f(n)$

  $<= 20\ f(n)$

  Entonces $t(n) \in O(f(n))$ t (n) está en el orden de f (n)

  ya que existe $c = 20$ tal que $t(n) <= 20\ f(n)$ para $n >= 1$

Notación Asintótica O(n) – O grande de n

- $T(n) = 3\ n^3 + 2\ n^2$ es $O(n^3)$

  Sean $n_0 = 0$ y $c = 5$.

  Entonces para $n >= 0$, existe $c = 5$, tal que

  $3\ n^3 + 2\ n^2 <= 5\ n^3$

- $T(n) = (n + 1)^2$

  $T(n)$ es $O(n^2)$, cuando $n_0 = 1$ y $c = 4$, es decir para $n >= 1$

Notación Asintótica O(n) – O grande de n

- El orden de complejidad se define como una función que domina la ecuación que expresa en forma exacta el tiempo de ejecución del algoritmo.
- **g(x) domina a f(x)**, si dada una constante **C** cualquiera, $C.g(x) \geq f(x) \; \forall \; x$
- g(x) domina asintóticamente a f(x), si g(x) domina a f(x) para valores muy grandes de x.
- Por ejemplo, si $f(n)=n^2 + 5n + 100$ y $g(n)= n^2$ entonces g(n) domina a f(n).

Notación Asintótica O(n) – O grande de n

Algunas reglas:

- $O(C \cdot g) = O(g)$, C es una constante.
- $O(f \cdot g) = O(f) \cdot O(g)$, y viceversa.
- $O(f / g) = O(f) / O(g)$, y viceversa.
- O(f+g) = función dominante entre O(f) y O(g)
- $n \cdot \log_2 n$ domina a $\log_2 n$
- b^n domina a c^n si b>=c
- n^k domina a n^m si k>=m
- $\log_a n$ domina a $\log_b n$ si $1 \leq a \leq b$
- $n!$ domina a b^n
- b^n domina a n^a si a>=0
- n domina a $\log_a n$ si a>=1
- $\log_a n$ domina a 1 si a>=1

Notación Asintótica O(n) – O grande de n

- Por ejemplo, para el siguiente algoritmo:

```
para i de 1 a n
        para j de 1 a n
                escribir i + j
        fpara
fpara
```

- El orden de complejidad del algoritmo se calcularía de la siguiente manera

Notación Asintótica O(n) – O grande de n

Paso 1:

```
para i de 1 a n
    para j de 1 a n
        escribir i + j          O(2)
    fpara
fpara
```

La complejidad de esta sentencia es constante

Paso 2:

```
para i de 1 a n
    para j de 1 a n
        escribir i + j          O(2n)
    fpara
fpara
```

La complejidad del bucle es el producto del número de ejecuciones por la complejidad de la/s sentencia/s ejecutada/s.

Notación Asintótica O(n) – O grande de n

Paso 3:

```
para i de 1 a n
    para j de 1 a n
        escribir i + j
    fpara
fpara
```

$O(2n.n) = O(2n^2) <= O(n^2)$

La complejidad del bucle es el producto del número de ejecuciones por la complejidad de la/s sentencia/s ejecutada/s.

- Se llega a la conclusión de que su complejidad es **$O(n^2)$**. Esto quiere decir, por ejemplo, que si el **tamaño de los datos aumenta en un orden de magnitud** (se multiplica por 10), **el tiempo de ejecución aumentaría** (en realidad estaría acotado superiormente) **en dos órdenes de magnitud** (se multiplicaría por 100).

Eficiencia y notación asintótica - Resumen

- Para resolver un **problema** pueden existir **varios algoritmos**. Por tanto, es lógico elegir aquel que use menos recursos (tiempo o espacio).
- Al hablar de la **eficiencia** en tiempo de ejecución de un algoritmo nos referiremos a lo **"rápido"** que se ejecuta.
- La eficiencia de un algoritmo dependerá, en general, del **"tamaño"** de los datos de entrada.
- Un algoritmo que afirma resolver un problema debe resolver todos los ejemplares del mismo. Si hay un solo ejemplar que no sea resuelto por el algoritmo el algoritmo no es correcto.
- No todos los ejemplares de un problema son iguales. Por ello, la eficiencia del algoritmo puede variar en función del ejemplar o del grupo de ejemplares.

Eficiencia y notación asintótica - Resumen

- A la hora de analizar un algoritmo es necesario saber que pueden darse tres tipos de ejemplares o casos: **caso mejor, caso peor y caso medio**.
- Una operación elemental es aquella cuyo tiempo de ejecución tiene una cota superior constante que sólo depende de su implementación. Se considera que son de **coste unitario**.
- La eficiencia no tiene unidades de medida, se usa la notación O(n).
- La notación anterior básicamente nos dice que si tenemos un algoritmo cuyo tiempo de ejecución es **f(n)** podemos encontrar otra función de n, **g(n)**, y un tamaño de problema $\mathbf{n_0}$ de tal forma que g(n) acota superiormente al tiempo de ejecución para todos los problemas de tamaño superior a n_0.

Eficiencia y notación asintótica - Resumen

- Esta notación, facilitará en las comparaciones de eficiencia entre algoritmos diferentes.
- Existen reglas y métodos para calcular el orden de complejidad de nuestros algoritmos para poder compararlos.