Análisis de Algoritmos

Para ciertos problemas, es posible encontrar más de un algoritmo que lo resuelva, lo cual nos enfrenta a decidir por alguno de ellos.

La mayoría de las veces, dicha elección está orientada hacia la disminución del costo que implica la solución del problema.

Bajo este enfoque es posible dividir los criterios en dos clases:

- **Criterios orientados a minimizar el costo de desarrollo**: claridad, sencillez y facilidad de implementación, depuración y mantenimiento.

- **Criterios orientados a disminuir el costo de ejecución**: tiempo de procesador y cantidad de memoria utilizada.

- El programa que implementa el algoritmo...
  - ¿Cuántas veces será ejecutado?
    -

- El programa que implementa el algoritmo...
  - ¿Cuántas veces será ejecutado?
    - ¿Unas cuantas? → escribir programas sencillos.

- El programa que implementa el algoritmo...

  - ¿Cuántas veces será ejecutado?
    - ¿Unas cuantas? → escribir programas sencillos.
    - ¿Uso frecuente? → es necesario invertir tiempo en escribir programas eficientes.

- El programa que implementa el algoritmo...
  - ¿Cuántas veces será ejecutado?
    - ¿Unas cuantas? → escribir programas sencillos.
    - ¿Uso frecuente? → es necesario invertir tiempo en escribir programas eficientes.

Nunca eliminar ninguno de los dos criteros completamente!!

- Los recursos que consume un algoritmo pueden estimarse con herramientas teóricas → estas herramientas conforman una base confiable para la elección de un algoritmo.

- Los recursos que consume un algoritmo pueden estimarse con herramientas teóricas → estas herramientas conforman una base confiable para la elección de un algoritmo.

- **Análisis de Algoritmos: determinar la cantidad de recursos que consume un algoritmo.**

- Problema: partida de ajedrez:

  es posible implementar un algoritmo que triunfe siempre: el algoritmo elige el siguiente movimiento examinando todas las posibles secuencias de movimientos desde el tablero actual hasta uno que sea claro el resultado, y elige siempre uno que le asegure ganar.

- Problema “intratable”!! → esto se va modificando con el aumento de capacidad computacional de las computadoras.

Algoritmos de tiempo polinómico y problemas intratables

- Los científicos de la computación realizan la distinción entre algoritmos de **tiempo polinómico** y algoritmos de **tiempo exponencial** cuando se trata de caracterizar a los algoritmos como "suficientemente eficiente" y "muy ineficiente" respectivamente.

- Un algoritmo de tiempo polinomial se define como aquel con función de complejidad temporal en O(p(n)) para alguna función polinómica p, donde n denota el tamaño de la entrada. Cualquier algoritmo cuya función de complejidad temporal no pueda ser acotada de esta manera, se denomina algoritmo de tiempo exponencial.

- La mayoría de los algoritmos de tiempo exponencial son simples variaciones de una búsqueda exhaustiva, mientras que los algoritmos de tiempo polinomial, usualmente se obtienen mediante un análisis más profundo de la estructura del problema. En la teoría de la complejidad computacional, existe el consenso de que un problema no está "bien resuelto" hasta que se conozca un algoritmo de tiempo polinomial que lo resuelva. Por tanto, nos referiremos a un problema como intratable, si es tan difícil que no existe algoritmo de tiempo polinomial capaz de resolverlo.

- Se asume que un problema tiene solución algorítmica si además de existir el algoritmo, su tiempo de ejecución es relativamente corto:
  - Problemas intratables: solución que tarda años en computar → se considera que la solución no existe.

- Problema: ordenar un conjunto de valores: 2 valores, 20 valores, 2000000 valores...

- En general, la cantidad de recursos que consume un problema se incrementa conforme crece el tamaño del problema.

- Complejidad de algoritmos:
  - La función de complejidad *f(n)*, donde *n* es el tamaño del problema, da la medida de la cantidad de recursos que el algoritmo necesitará al implementarse y ejecutarse en una computadora.
  - La función de complejidad es monotonamente creciente: $f(n) > f(m)$ sii $n > m$, con respecto al tamaño del problema.

- Elección del tamaño del problema:

PROBLEMA	TAMAÑO DEL PROBLEMA
Búsqueda de un elemento en un conjunto	Número de elementos en el conjunto
Multiplicar dos matrices	Dimensión de las matrices
Recorrer un árbol	Número de nodos en el árbol
Resolver un sistema de ecuaciones lineales	Número de ecuaciones y/o incógnitas
Ordenar un conjunto de valores	Número de elementos en el conjunto

- La **memoria** y el **tiempo de procesador** son los recursos sobre los cuales se concentra todo el interés en el análisis de algoritmos.

- La **memoria** y el **tiempo de procesador** son los recursos sobre los cuales se concentra todo el interés en el análisis de algoritmos.

- Dos clases de función de complejidad:

- La **memoria** y el **tiempo de procesador** son los recursos sobre los cuales se concentra todo el interés en el análisis de algoritmos.

- Dos clases de función de complejidad:
  - *Función de complejidad espacial*: mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño *n*.

- La **memoria** y el **tiempo de procesador** son los recursos sobre los cuales se concentra todo el interés en el análisis de algoritmos.

- Dos clases de función de complejidad:
  - *Función de complejidad espacial*: mide la cantidad de memoria que necesitará un algoritmo para resolver un problema de tamaño *n*.
  - *Función de complejidad temporal:* indica la cantidad de tiempo que requiere un algoritmo para resolver un problema de tamaño *n*. Medida de la cantidad de CPU que requiere la ejecución del algoritmo.

- Complejidad espacial: se cuentan las celdas de memoria que utilice el algoritmo. Dos tipos de celdas de memoria:
  - Celdas estáticas: se utilizan todo el tiempo que dura la ejecución de un programa (por ej. variables globales).
  - Celdas dinámicas: se emplean sólo durante un momento de la ejecución, pueden ser asignadas y devueltas conforme se ejecuta el programa (memoria dinámica).

- Complejidad temporal: no se expresa en unidades de tiempo sino en términos de la cantidad de operaciones que realiza.
  - Cada operación requiere un cierto valor constante de tiempo en ejecutarse, con lo que si se cuentan la cantidad de operaciones realizadas por el algoritmo se obtiene una estimación del tiempo que le tomará resolver el problema.

¿Por qué se cuentan la cantidad de operaciones y no el tiempo?

- Reglas generales para el análisis de programas:
  - El tiempo de ejecución de cada **proposición de asignación** puede tomarse como constante: 1 (excepto lenguajes donde se puedan asignar matrices o llamado a funciones)

- Reglas generales para el análisis de programas:
  - El tiempo de ejecución de cada **proposición de asignación** puede tomarse como constante: 1 (excepto lenguajes donde se puedan asignar matrices o llamado a funciones)
  - El tiempo de ejecución de una **secuencia de proposiciones** se determina por la regla de la suma: el máximo de los tiempo de las secuencias.

- Reglas generales para el análisis de programas:
  - El tiempo de ejecución de cada **proposición de asignación** puede tomarse como constante: 1 (excepto lenguajes donde se puedan asignar matrices o llamado a funciones)
  - El tiempo de ejecución de una **secuencia de proposiciones** se determina por la regla de la suma: el máximo de los tiempo de las secuencias.
  - El tiempo de ejecución de una **proposición condicional *if*** es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo de evaluar la condición (generalmente constante).

- Reglas generales para el análisis de programas:
  - El tiempo de ejecución de cada **proposición de asignación** puede tomarse como constante: 1 (excepto lenguajes donde se puedan asignar matrices o llamado a funciones)
  - El tiempo de ejecución de una **secuencia de proposiciones** se determina por la regla de la suma: el máximo de los tiempo de las secuencias.
  - El tiempo de ejecución de una **proposición condicional *if*** es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo de evaluar la condición (generalmente constante).
  - El tiempo para una **construcción *if-then-else*** es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos: condición verdadera o falsa.

- Reglas generales para el análisis de programas:
  - El tiempo de ejecución de cada proposición de asignación puede tomarse como constante: 1 (excepto lenguajes donde se puedan asignar matrices o llamado a funciones)
  - El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma: el máximo de los tiempo de las secuencias.
  - El tiempo de ejecución de una proposición condicional *if* es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo de evaluar la condición (generalmente constante).
  - El tiempo para una construcción *if-then-else* es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos: condición verdadera o falsa.
  - El tiempo para **ejecutar un ciclo** es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación. A menudo suele ser el producto del número de iteraciones del ciclo y el mayor tiempo posible para la ejecución del cuerpo.

- Llamadas a procedimientos:
  - Se obtiene el tiempo de ejecución de los distintos procedimientos, uno a la vez, partiendo de los que no llaman a otros procedimientos.
  - Luego se obtiene el tiempo de los procedimientos que llaman a estos procedimientos, y así se continúa el proceso evaluando el tiempo de ejecución de cada procedimiento después de haber evaluado los tiempos correspondientes a los procedimientos que llama.

- Se elige una operación básica que observe un comportamiento similar al del número total de operaciones realizadas, y que será proporcional al tiempo total de ejecución.
- Se escoge una operación que esté relacionada con el problema a resolver, ignorando las asignaciones iniciales y finales, y las asignaciones, de por ejemplo, manejo de índices.
- Del ejemplo 1, se puede considerar la comparación entre el valor y los valores del vector como operación básica.

Algoritmo 1: obtiene el producto de los dos valores más grandes contenidos en un arreglo *A* de *n* enteros.

```
Producto2Mayores (A, n)

Comenzar
     Si A[1] > A[2]
        Mayor1 = A[1]
        Mayor2 = A[2]
     Sino
        Mayor1 = A[2]
        Mayor2 = A[1]
     Fin si
     i = 3
     Mientras (i <= n)
        Si A[i] > Mayor1
          Mayor2 = Mayor1
          Mayor1 = A[i]
        Sino
          Si A[i] > Mayor2
            Mayor2 = A[i]
        Fin si
        i = i + 1
     Fin Mientras
     Retornar (Mayor1 * Mayor2)
fin
```

¿Cuáles operaciones pueden ser elegidas como operaciones básicas?

Deben cumplir:

- Que tengan relación con el problema a resolver.
- Que se ejecuten una cantidad de veces proporcional al tiempo de ejecución total.

- Algunos ejemplos de elección de operaciones básicas:

PROBLEMA	OPERACIÓN BÁSICA
Búsqueda de un elemento en un conjunto	Comparación entre el valor y los elementos del conjunto
Multiplicar dos matrices	Producto de los elementos de las matrices
Recorrer un árbol	Visitar un nodo
Resolver un sistema de ecuaciones lineales	Suma
Ordenar un conjunto de valores	Comparación entre valores

- Análisis de comportamiento:
  - Mejor caso.
  - Caso promedio.
  - Peor caso.

```
BusquedaLineal (A, n, dato).

Comenzar
     i = 1
     Mientras (i <= n) and (A[i] != dato)
         i = i +1

     Devolver i
fin
```

A = [2, 7, 6, 77, 3, 45, 9, 99]

- Comportamiento Asintótico de funciones:

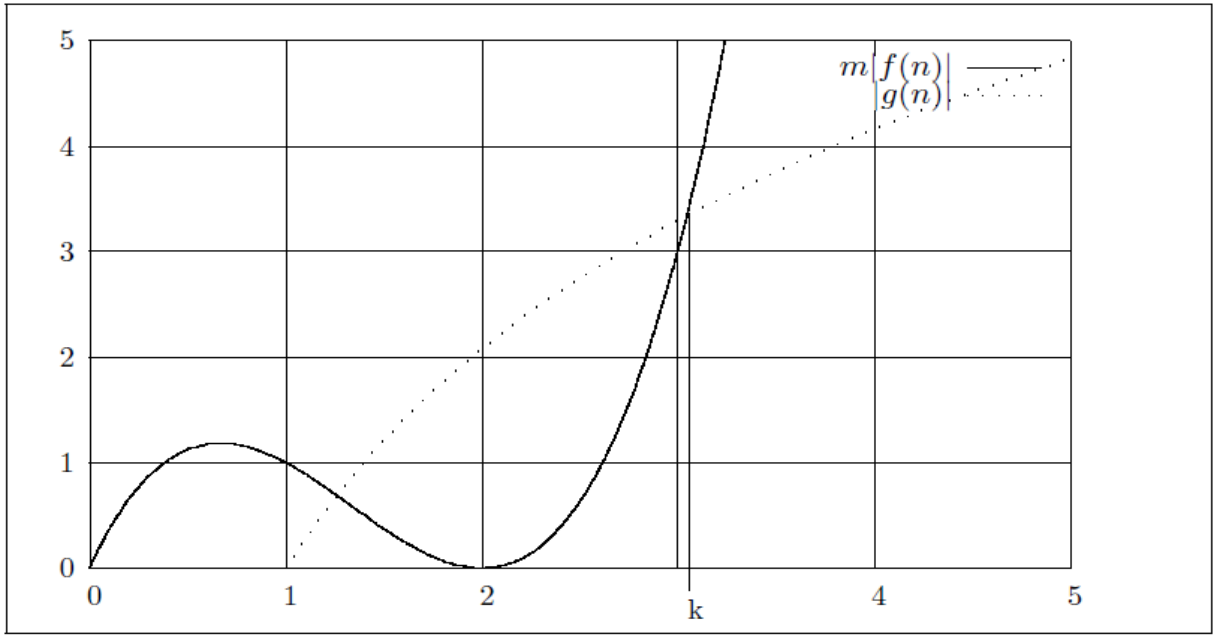
Cuando el tamaño del problema es grande, la cantidad de recursos que el algoritmo necesite puede crecer tanto que lo haga impráctico.

Por esta razón, para elegir entre dos algoritmos es necesario saber cómo se comportan con problemas grandes.

En atención a esta necesidad se estudiará la velocidad de crecimiento de la cantidad de recursos que un algoritmo requiere conforme el tamaño del problema se incrementa, es decir, se estudiará el comportamiento asintótico de las funciones complejidad.

Definición 1 *Sean f y g funciones de $\mathbb{N}$ a $\mathbb{R}$. Se dice que f domina asintóticamente a g o que g es dominada asintóticamente por f, si $\exists k \geq 0$ y $m \geq 0$ tales que*

$$|g(n)| \leq m|f(n)|, \forall n \geq k$$



En otros términos, podemos decir que si una función domina a otra, su velocidad de crecimiento es mayor o igual. Si esas funciones están representando el consumo de recursos de la computadora, obviamente la función dominada crecerá a lo más a la misma velocidad de la otra.

Ejemplo 6 *Sean* $f(n) = n$ *y* $g(n) = n^3$

i) Demostrar que g *domina asintóticamente a* f*. Esto es, demostrar que*

$$\exists m \geq 0, k \geq 0 \text{ tales que } |f(n)| \leq m|g(n)| \quad \forall n \geq k.$$

Substituyendo $f(n)$ *y* $g(n)$ *da*

$$|n| \leq m|n^3|, \forall n \geq k,$$

si se toma $m = 1$ *y* $k = 0$*, las desigualdades anteriores se cumplen, por lo tanto,* m *y* k *existen, y en consecuencia* g *domina asintóticamente a* f*.*

- Notación de orden:

Cuando describimos cómo es que el número de operaciones $f(n)$ depende del tamaño n; lo que nos interesa es encontrar el patrón de crecimiento para la función complejidad y así caracterizar al algoritmo; una vez hecha esta caracterización podremos agrupar las funciones de acuerdo al número de operaciones que realizan. Para tal propósito se da la siguiente definición.

Definición: El conjunto de todas las funciones que son asintóticamente dominadas por una función g es denotado $O(g)$ y se lee orden de g, o bien O-grande de g.

Es posible clasificar los algoritmos según el orden de su función de complejidad. Gran parte de los algoritmos tienen complejidad que cae en uno de los siguientes casos:

$O(1)$	Complejidad constante
$O(\log n)$	Complejidad logarítmica
$O(n)$	Complejidad lineal
$O(n \log n)$	Complejidad "n log n"
$O(n^2)$	Complejidad cuadrática
$O(c^n), c > 1$	Complejidad exponencial
$O(n!)$	Complejidad factorial

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(c^n) \subset O(n!)$$

Interpretando este teorema de acuerdo a nuestros intereses podemos decir que un algoritmo con complejidad factorial consume más recursos que uno con complejidad exponencial y uno con complejidad exponencial consume más que uno con complejidad cuadrática, etc.

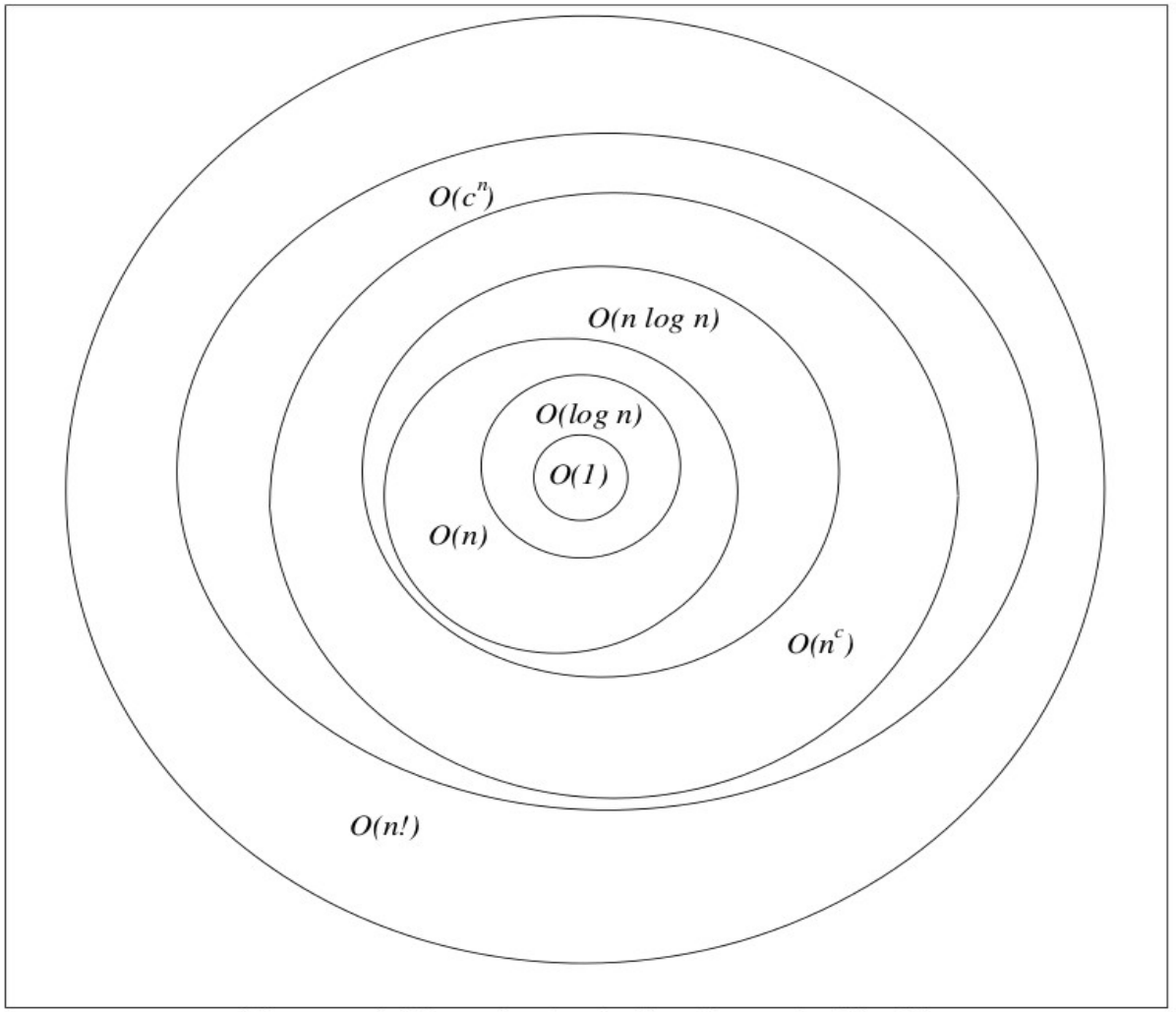


Figura 4: El conjunto de funciones de $\mathbb{N}$ a $\mathbb{R}$.

Comparar los algoritmos en base al comportamiento asintótico de sus funciones complejidad es una herramienta de elección muy poderosa que, sin embargo debe ser utilizada con cuidado pues, por ejemplo, si tenemos dos algoritmos con funciones complejidad

$$f(n) = cn, \text{ es } O(n) \text{ y}$$
$$g(n) = an^2, \text{ es } O(n^2) \text{ con } a \text{ y } c \text{ constantes}$$

en base a lo dicho, elegiríamos el algoritmo cuya función complejidad es $f(n)$, sin embargo, si $c = 25$ y $a = 1$, $g(n) < f(n)$ para $n < 25$, por lo tanto también es importante tomar en cuenta el tamaño del problema especifico.

- Con frecuencia encontraremos que al analizar un algoritmo, una sección de código es de un orden y otra parte es de otro orden, entonces ¿cuál es el orden del algoritmo?

  → regla de la suma: si P1 y P2 son dos fragmentos de programa con Orden O(f(n)) y O(g(n)) respectivamente, entonces, el tiempo de ejecución de P1 seguido de P2 es O(max(f(n), y g(n)).

  Ejemplo: se tienen tres pasos cuyos tiempos son, respectivamente $O(n^2)$, $O(n^3)$ y O(nlogn), entonces, el tiempo de ejecución de los primeros dos pasos es $O(max(n^2, n^3))$ que es $O(n^3)$. El tiempo de los tres es $O(max(n^3, nlogn))$, que es $O(n^3)$.

  → regla del producto: si P1 y P2 son dos fragmentos de programa y son O(f(n)) y O(g(n)) respectivamente, entonces el tiempo de P1xP2 es O(f(n)*g(n)). Según esta regla O(cf(n)) para c constante es lo mismo que O(f(n)).

  Ejemplo: $O(n^2/2)$ es lo mismo que $O(n^2)$.

- En general, el tiempo de ejecución de una secuencia finita de pasos, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución.

- Si *g(n)* <= *f(n)* para toda *n* mayor que una constante n_0, entonces O(f(n), g(n)) es lo mismo que O(f(n)). Ejemplo: $O(n^2 + n)$ es lo mismo que $O(n^2)$.

- Reglas generales para el análisis de programas:
  - Ciclos (iteraciones)
  - En los bucles con contador explícito, podemos distinguir dos casos, que el tamaño N forme parte de los límites o que no. Si el bucle se realiza un número fijo de veces, independiente de N, entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

    for (int i= 0; i < K; i++) { algo_de_O(1) } => K*O(1) = O(1)
  - Si el tamaño N aparece como límite de iteraciones ...

    for (int i= 0; i < N; i++) { algo_de_O(1) } => N * O(1) = O(n)
  - Otro ejemplo:

```
for (int i= 0; i < N; i++) {
    for (int j= 0; j < N; j++) {
      algo_de_O(1)
    }
}
```

    tendremos N * N * O(1) = $O(n^2)$

- Reglas generales para el análisis de programas:
  - Ciclos (iteraciones)

```
for (int i= 0; i < N; i++) {
      for (int j= 0; j < i; j++) {
        algo_de_O(1)
      }
   }
```

el bucle exterior se realiza N veces, mientras que el interior se realiza 1, 2, 3, ... N veces respectivamente. En total, 1 + 2 + 3 + ... + N = N*(1+N)/2 -> $O(n^2)$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

- Reglas generales para el análisis de programas:
  - Ciclos (iteraciones)

    A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

    - Ej: c = 1;

      while (c < N) {

      algo_de_O(1)

      c = 2*c; } El valor incial de "c" es 1, siendo "2k" al cabo de "k" iteraciones. El número de iteraciones es tal que 2k >= N => k= es (log2 (N)) [el entero inmediato superior] y, por tanto, la complejidad del bucle es O(log n).

    - Ej. c = N;

      while (c > 1) {

      algo_de_O(1)

      c = c / 2; } Un razonamiento análogo nos lleva a log2(N) iteraciones y, por tanto, a un orden O(log n) de complejidad.

    - Ej. for (int i= 0; i < N; i++) {

      c= i;

      while (c > 0) {

      algo_de_O(1)

      c= c/2;

      } } tenemos un bucle interno de orden O(log n) que se ejecuta N veces, luego el conjunto es de orden O(n log n)

- Ejemplo 1: Considérese el Algoritmo 1, que realiza una búsqueda lineal sobre un arreglo *A* con *n* elementos, y devuelve la posición en la que se encuentra el elemento Valor; si Valor no se encuentra devuelve *n*+1.

```
BusquedaLineal (A, n, dato)

Comenzar
     i = 1
     Mientras (i <= n) and (A[i] != 
dato)
          i = i +1

     Devolver i
fin
```

Si el ciclo se ejecuta k veces, se ejecutan:

k sumas (una por iteración)
k + 2 asignaciones
k +1 operaciones lógicas
k +1 comparaciones con el indice
k +1 comparaciones con el elemento del vector A.
k +1 accesos a los elementos de A

6k+ 6 operaciones en total.

No hace falta contar todas las operaciones en total!!!

- Ordenación por método de la Burbuja

```
     procedure burbuja ( var A: array [1..n] of integer );
         { burbuja clasifica el arreglo A de menor a mayor }
     var
         i, j, temp: integer;
         begin
(1)          for i := 1 to n-1 do
(2)              for j := n downto i + 1 do
(3)                  if A[j-1] > A[j] then begin
                         { intercambia A[j-1] y A[j] }
(4)                      temp := A[j-1];
(5)                      A[j-1] := A[j];
(6)                      A[j] := temp
                     end
         end; { burbuja }
```

Fig. 1.13. Clasificación burbuja.

Producto de matrices:

```
proc Matriz-Producto(n, A, B) ≡                                    
  comienza
    para i = 1 a n haz                                             I1
        para j = 1 a n haz                                         I2
            C[i, j] ← 0;                                           I3
            para k = 1 a n haz                                     I4
                C[i, j] ← C[i, j] + A[i, k] * B[k, j];             I5
            zah
        zah
    zah
  termina.
```

a) I_5 tiene orden constante, entonces es O(1)

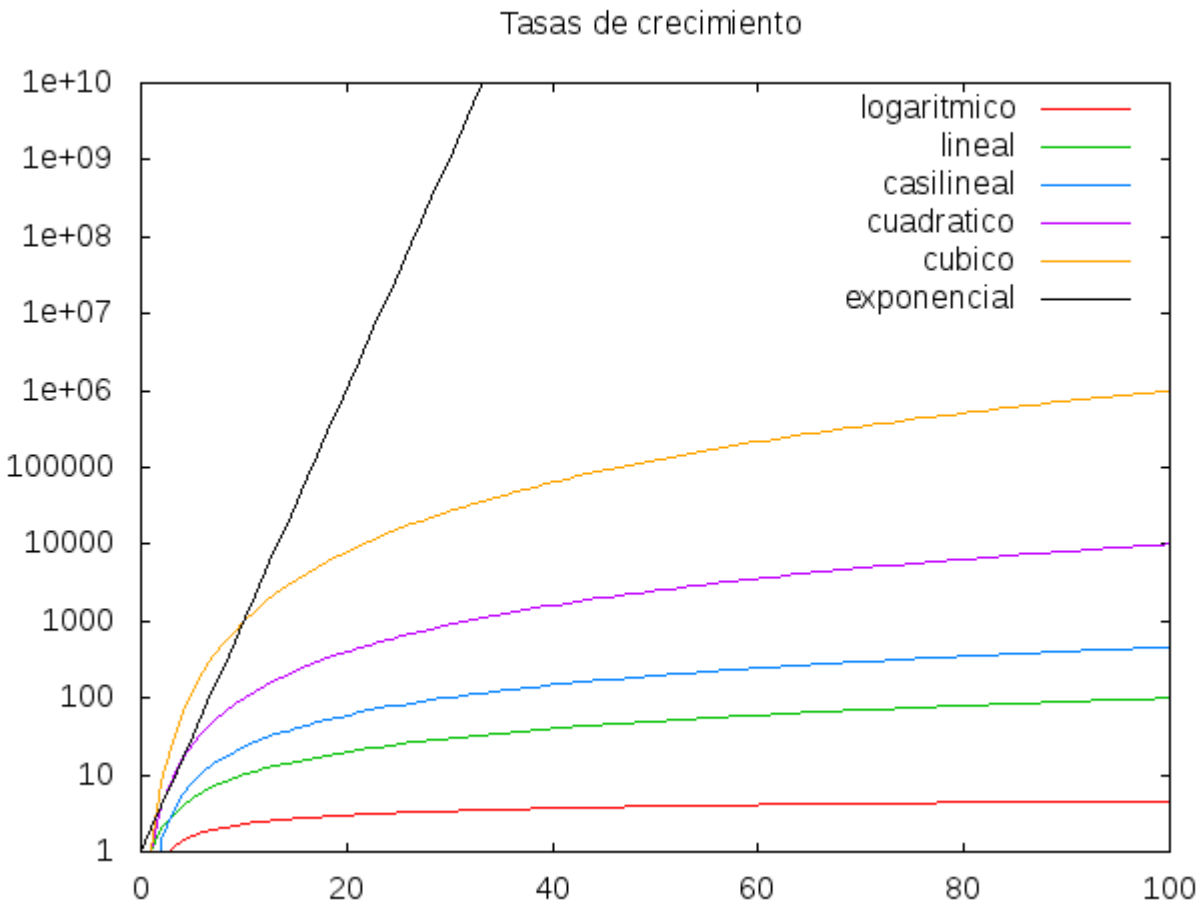
b) Como I_5 está dentro del ciclo I_4 entonces el orden es $O(n \times 1) = O(n)$.

c) El ciclo I_2 contiene al conjunto de instrucciones I_3 e I_4, si I_3 es $O(1)$ se tiene que la secuencia tiene orden $O(n)$, pero ésta se realiza n veces, así el orden para este bloque es $O(n \times n) = O(n^2)$.

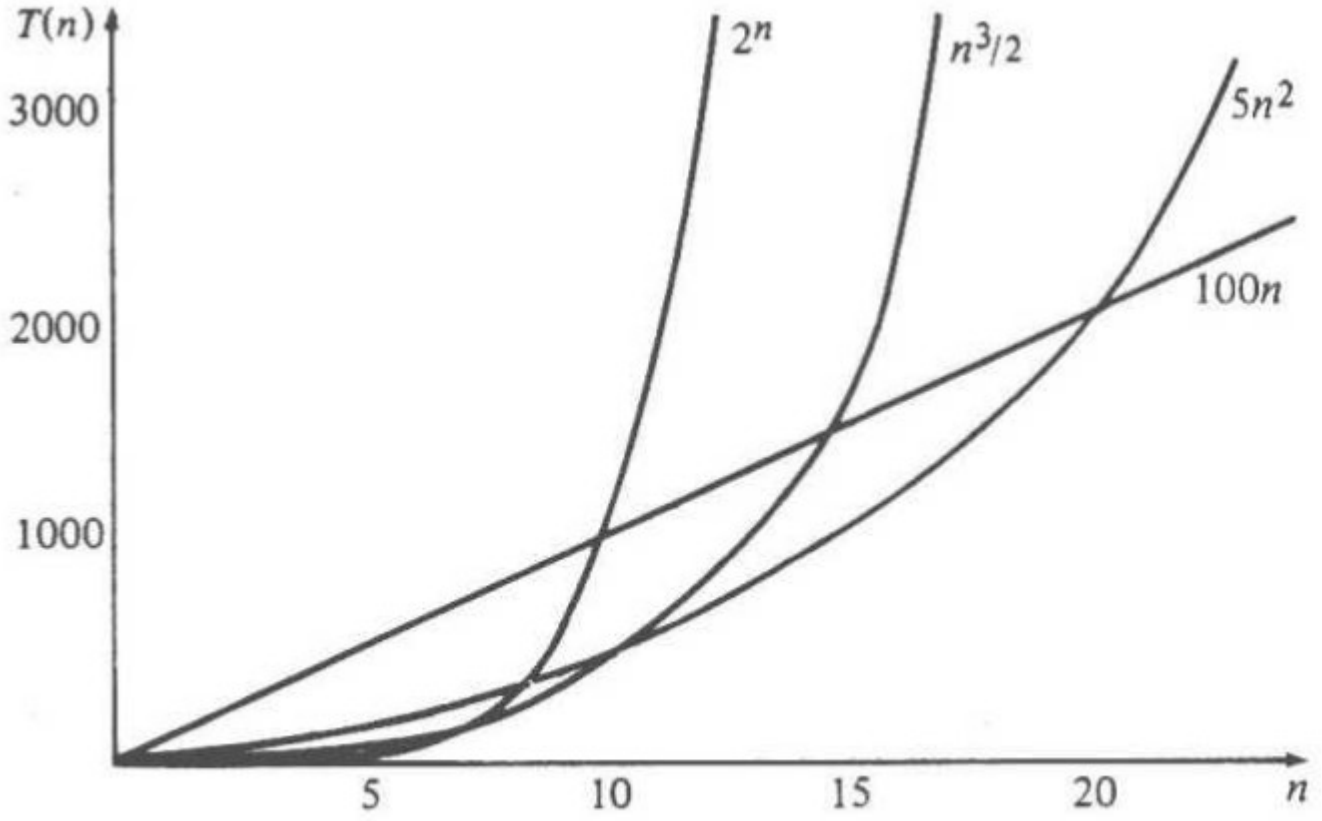
d) Por último, el ciclo I_1 contiene a la instrucción I_2 que se realiza n veces de ésta forma el orden para I_1, y en consecuencia para el Algoritmo 6, es:

$$O(n \times n^2) = O(n^3).^1$$

Tasas de Crecimiento



Tasas de Crecimiento



- Efecto de duplicar el dato de entrada

T(n)	n = 100	n = 200
log(n)	1 h.	1.15 h.
n	1 h.	2 h.
nlog(n)	1 h.	2.30 h.
n^2	1 h.	4 h.
n^3	1 h.	8 h.
2^n	1 h.	$1.27*10^{30}$ h. |

- Efecto de duplicar el tiempo disponible

T(n)	t = 1h	t = 2h
log(n)	n = 100	n = 10000
n	n = 100	n = 200
nlog(n)	n = 100	n = 178
n^2	n = 100	n = 141
n^3	n = 100	n = 126
2^n	n = 100	n = 101

- Algoritmos recursivos: existen distintos métodos. Uno es sustituir las recurrencias por su igualdad hasta llegar a un caso conocido.

```
int Fact (int n) {

     if (n <= 1)
          return(1);
     else
          return( n * Fact(n-1));
}
```

$$T(n) = \begin{cases} c_1, & si\ n \le 1 \\ T(n-1) + c_2, & si\ n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(n\text{-}1) + c_2 \\ &= (T(n\text{-}2) + c_2) + c_2 && = T(n\text{-}2) + 2*c_2 = \\ &= (T(n\text{-}3) + c_2) + 2*c_2 && = T(n\text{-}3) + 3*c_2 = \\ &\ldots \\ &= T(n\text{-}k) + k*c_2 \end{aligned}$$

Cuando $k = n-1$, tenemos que $T(n) = T(1) + c_2*(n\text{-}1)$, y es $O(n)$.

```
int Recursiva1 (int n) {

 if (n <= 1)
   return(1);
 else
   return(Recursiva1(n-1)+ Recursiva1(n-1));
}
```

$$T(n) = \begin{cases} 1, & si\ n \le 1 \\ 2 \cdot T(n-1)+1, & si\ n > 1 \end{cases}$$

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

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

. . . .

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

Para k = n-1, $T(n) = 2^{n-1}*T(1) + 2^{n-1}-1$, y por tanto T(n) es $O(2^n)$.

```
int Recursiva3 (int n) {

 if (n <= 1)
   return (1);
 else
   return (2 * Recursiva3(n /2));
}
```

$$T(n) = \begin{cases} 1, & si\ n \le 1 \\ T(n/2) + 1, & si\ n > 1 \end{cases}$$

$T(n) = T(n/2) + 1 =$

$= T(n/2^2) + 2 =$

....

$= T(n/2^k) + k$

Como la ecuación $n/2^k = 1$ se resuelve para $k = \log_2 n$, tenemos que

$T(n) = T(1) + \log_2 n = 1 + \log_2 n$, y por tanto $T(n)$ es $O(\log n)$.

```
int Recursiva4 (int n, int x) {
  int i;
  if (n <= 1)
      return (1);
  else {
         for (i=1;i<=n;i++) {
             x = x + 1;
         }
         return( Recursiva4(n-1, x));
   }
}
```

$$T(n) = \begin{cases} 1, & si\ n \le 1 \\ T(n-1)+n, & si\ n > 1 \end{cases}$$

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

$= ((T(n-3)+(n-2))+(n-1)+n = \ldots . = T(n-k) + \sum_{i=0}^{k-1}(n-i)$

Si $k = n-1$, $T(n) = T(1) + \sum_{i=0}^{n-2}(n-i) = 1 + (\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i) =$

$= 1 + n\,(n-1) - (n-2)(n-1)/2$. Por tanto, $T(n)$ es $O(n^2)$.

Problemas P, NP y NP-completos (extraído de: http://www.lab.dit.upm.es/~lprg/material/apuntes/o/)

Hasta aquí hemos venido hablando de algoritmos. Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el orden de complejidad de un problema es el del mejor algoritmo que se conozca para resolverlo. Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad.

Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafían la utilización de los ordenadores para resolverlos. En lo que sigue esbozaremos las clases de problemas.

Clase P

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables. Seria muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Clase NP

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Clase NP-completos

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!

Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

Entonces...

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica como se comporta el algoritmo para datos muy grandes y salvo algún coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

(extraído de: http://www.lab.dit.upm.es/~lprg/material/apuntes/o/)