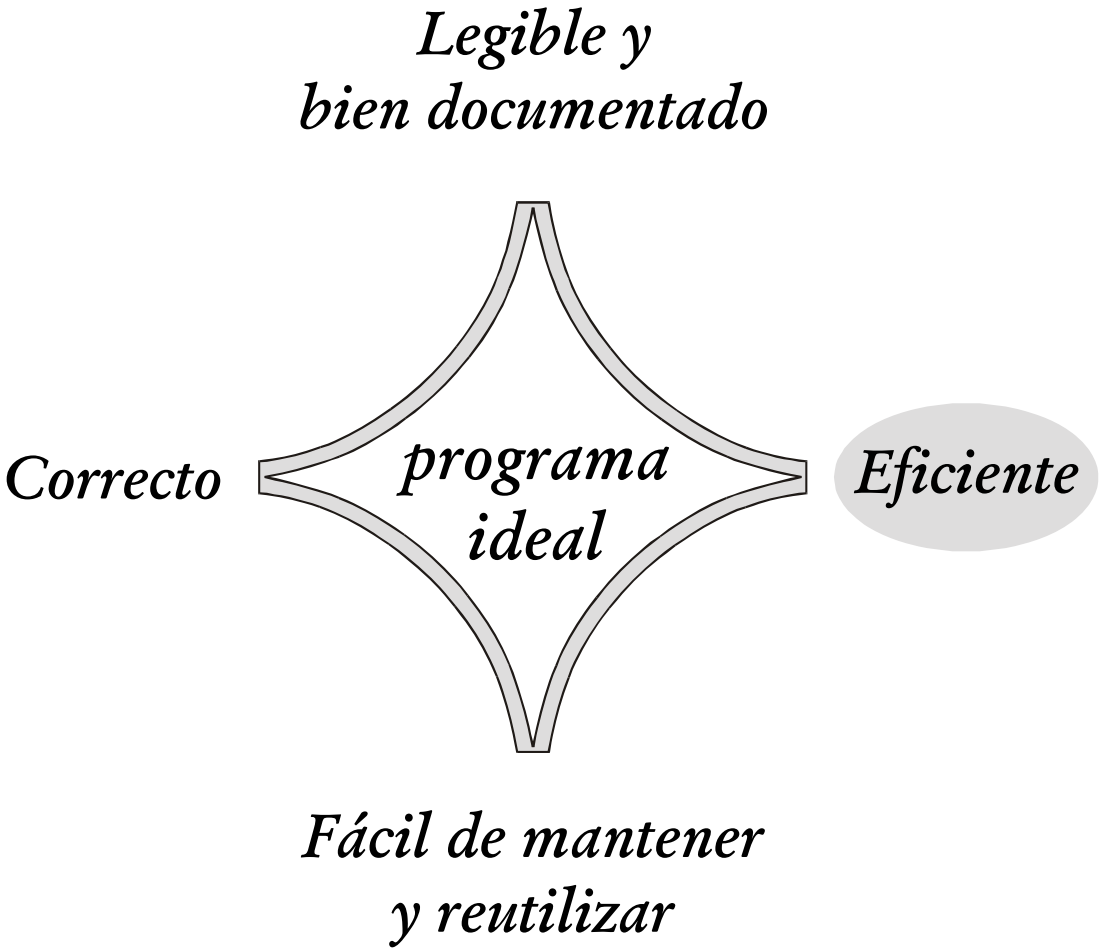
TEMA 1
ANÁLISIS DE LA EFICIENCIA

1. Complejidad de los algoritmos
2. Medidas asintóticas de la complejidad
3. Ordenes de complejidad
4. Métodos de análisis de la complejidad temporal de los algoritmos iterativos

Bibliografía: *Fundamentals of Data Structures in C++*
E. Horowitz, S. Sahni, D. Mehta
Computer Science Press, 1995

1.1 Complejidad de los algoritmos



- Un algoritmo será tanto más eficiente cuantos menos recursos consuma: tiempo y espacio de memoria necesario para ejecutarlo.

- La eficiencia de los algoritmos se cuantifica con las *medidas de complejidad*:
  - *Complejidad temporal*: tiempo de cómputo de un programa
  - *Complejidad espacial*: memoria que utiliza un programa en su ejecución

- En general, es más importante conseguir buenas complejidades temporales

¡¡¡ Seamos generosos con las variables !!!

- La complejidad de un programa depende de
  - la máquina y el compilador.
  - el tamaño de los datos de entrada,
  - el valor de los datos de entrada, y

- En los estudios teóricos de la complejidad no nos ocupamos de la máquina ni del compilador.

- El "tamaño de los datos" depende del tipo de datos y del algoritmo:
  - para un vector, su longitud,
  - para un número, su valor o su número de dígitos, …

- Para un tamaño de datos dado, hablamos del valor de los datos como pertenecientes al caso mejor, al caso peor, o, si tenemos en cuenta todos los valores posibles y sus probabilidades, al caso promedio. Por ejemplo,

```
const int N = 10;
int i, j, x;
int v[N];

for ( i = 1; i < N; i++ )
{
  x = v[i];
  j = i-1;
  while ( (j >= 0) && (v[j] > x) )
  {
      v[j+1] = v[j];
      j = j-1;
  }
   v[j+1] = x;
}
// ¿qué hace este algoritmo?
```

con tamaño 4: uno = [1,2,5,4], dos = [1,2,3,4] y tres = [4,3,2,1]

  - Caso mejor: *dos* (no se entra nunca en el while)
  - Caso peor: *tres* (se ejecuta *(i-1)* veces el bucle por cada *i)*

Medida del tiempo de ejecución de un algoritmo

- Vamos a medir el tiempo de ejecución del algoritmo de ordenación anterior, aplicado sobre datos de distinto tamaño
  - Convertimos el algoritmo de ordenación en un procedimiento

```
void ordena ( int v[], int num ) {
  int i, j, x;

  for ( i = 1; i < num; i++ ) {
    x = v[i];
    j = i-1;
    while ( (j >= 0 ) && (v[j] > x) ) {
      v[j+1] = v[j];
      j = j-1;
    }
    v[j+1] = x;
  }
}
```

  - Para medir el tiempo utilizamos la función *clock( )* que devuelve un valor de tipo *clock_t* con el número de ciclos de reloj en ese instante.

```
const int N = 50;
clock_t t1, t2;
int v[N];
double tiempo;

for ( i = 0; i < N; i++ )
  v[i] = rand();
t1 = clock();
ordena(v, N);
t2 = clock();
tiempo = double(t2-t1)/CLOCKS_PER_SEC;
```

el problema de este método es que la precisión del reloj del sistema suele estar entorno a los milisegundos, y el tiempo a medir es inferior o de ese orden.

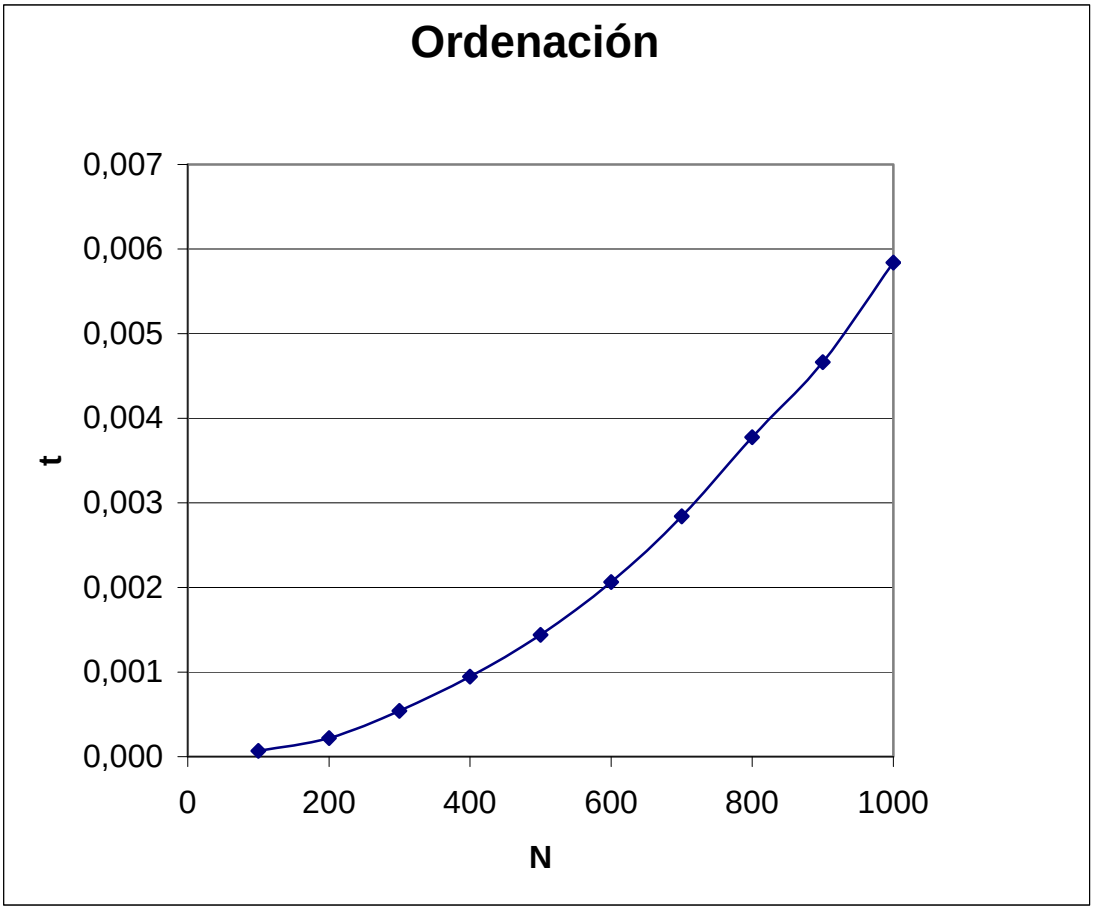
La solución es repetir la ejecución del procedimiento que pretendemos medir un cierto número de veces hasta que el tiempo medido sea significativo. Como pretendemos tomar medidas para distintos tamaños de datos, repetiremos también el proceso para cada valor de N, y utilizaremos un número de repeticiones distinto dependiendo de dicho valor.

- El programa de toma de tiempos queda entonces

```
int main( int argc, char* argv[] )
{
  const int N = 1000;
  const int NumPuntos = 10;
  int i, j, k;
  int tamanyos[NumPuntos] =     {  100,  200,  300,  400,  500,  600,
                                    700,  800, 900, 1000 };
  int repeticiones[NumPuntos] = { 5000, 4000, 4000, 3000, 3000, 3000,
                                  2000, 1000, 500,  100 };
  double tiempos[NumPuntos];
  int u[N], v[N];
  clock_t t1, t2;

  for ( i = 0; i < N; i++ )
    u[i] = rand();

  for ( k = 0; k < NumPuntos; k++ ) {
    t1 = clock();
    for( i = 0; i < repeticiones[k]; i++) {
      for ( j = 0; j < tamanyos[k]; j++ )
        v[j] = u[j];
      ordena(v, tamanyos[k]);
    }
    t2 = clock();
    tiempos[k] = (double(t2-t1)/CLOCKS_PER_SEC) / repeticiones[k];
  }

  for ( k = 0; k < NumPuntos; k++ )
    cout << "N = " << tamanyos[k] << "; t = " << tiempos[k] << endl;

  char c;
  cin >> c;
  return 0;
}
```

- Los resultados de la ejecución anterior (en un Pentium III a 650MHz, compilado en C++ Builder 5)

N	t
100	0,000067
200	0,000220
300	0,000544
400	0,000947
500	0,001439
600	0,002063
700	0,002842
800	0,003775
900	0,004662
1000	0,005840



Definición de complejidad temporal

- El tiempo que tarda un algoritmo *A* en procesar una entrada concreta $\vec{x}$ lo notaremos como

  $t_A(\vec{x})$

- Definimos la complejidad de un algoritmo *A* en el caso peor, que notaremos

  $T_A(n)$

  como la función

  $T_A(n) =_{def} \max\{ t_A(\vec{x}) \mid \vec{x} \text{ de tamaño } n \}$

- Definimos la complejidad de un algoritmo A en el caso promedio, que notaremos

  $TM_A(n)$

  como la función

  $$TM_A(n) =_{def} \sum_{\vec{x} \text{ de tamaño n}} p(\vec{x}) \cdot t_A(\vec{x})$$

  siendo $p(\vec{x})$ la función de probabilidad –$p(\vec{x}) \in [0,1]$– de que la entrada sea el dato $\vec{x}$.

- La complejidad en el caso promedio supone el conocimiento de la distribución de probabilidades de los datos.

- La complejidad en el caso peor proporciona una medida pesimista, pero fiable.

- Dado que realizamos un estudio teórico, ignorando los detalles de la máquina y el compilador, y teniendo en cuenta que las diferencias en eficiencia se hacen significativas para tamaños grandes de los datos, no pretendemos obtener la forma exacta de la función de complejidad sino que nos limitaremos a estudiar su *comportamiento asintótico*.

1.2 Medidas asintóticas de la complejidad

- Una *medida asintótica* es un conjunto de funciones que muestran un comportamiento similar cuando los argumentos toman valores muy grandes.
  Las medidas asintóticas se definen en términos de una *función de referencia f*.

- Tanto las funciones de complejidad, *T*, como las funciones de referencia de las medidas asintóticas, *f*, presentan el perfil

  $T, f : \aleph \rightarrow \Re^+$

  donde $\aleph$ es el dominio del tamaño de los datos y $\Re^+$ el valor del coste del algoritmo.

- Definimos la *medida asintótica de cota superior f*, que notaremos

  $O(f)$

  como el conjunto de funciones

  $\{ T \mid$ existen $c \in \Re^+$, $n_0 \in \aleph$ tales que,
  para todo $n \geq n_0$, $T(n) \leq c \cdot f(n) \}$

  Si $T \in O(f)$ decimos que "$T(n)$ es del orden de $f(n)$" y que "f es asintóticamente una cota superior del crecimiento de T".

- Definimos la *medida asintótica de cota inferior f*, que notaremos

  $\Omega(f)$

  como el conjunto de funciones

  $\{ T \mid$ existen $c \in \Re^+$, $n_0 \in \aleph$ tales que,
  para todo $n \geq n_0$, $T(n) \geq c \cdot f(n) \}$

  Si $T \in \Omega(f)$ podemos decir que "f es asintóticamente una cota inferior del crecimiento de T".

- Definimos la *medida asintótica exacta f*, que notaremos

  $\Theta(f)$

  como el conjunto de funciones

  $\{ T \mid$ existen $c_1, c_2 \in \Re^+$, $n_0 \in \aleph$, tales que,
  para todo $n \geq n_0$, $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \}$

  Si $T \in \Theta(f)$ decimos que "$T(n)$ es del orden exacto de $f(n)$".

- Ejemplos
  - $f \in O(f)$
  - $(n+1)^2 \in O(n^2)$, con $c=4$, $n_0=1$
  - $3n^3+2n^2 \in O(n^3)$, con $c=5$, $n_0=0$
  - $P(n) \in O(n^k)$, para todo polinomio P de grado k
  - $3^n \notin O(2^n)$. Si lo fuese existirían $c \in \Re^+$, $n_0 \in \aleph$ tales que

    $$\forall\, n : n \geq n_0 : 3^n \leq c \cdot 2^n$$
    $$\Rightarrow \quad \forall\, n \geq n_0 : (3/2)^n \leq c$$
    $$\Leftrightarrow \quad \text{falso}$$

    pues $\lim_{n\to\infty}(3/2)^n = \infty$
  - $f \in \Omega(f)$
  - $n \in \Omega(2n)$ con $c = ½$, $n_0=0$
  - $(n+1)^2 \in \Omega(n^2)$ con $c = 1$ y $n_0=1$
  - $P(n) \in \Omega(n^k)$ para todo polinomio P de grado k
  - $f \in \Theta(f)$
  - $2n \in \Theta(n)$ con $c_1=1$, $c_2=2$ y $n_0=0$
  - $(n+1)^2 \in \Theta(n^2)$ con $c_1=1$, $c_2 = 4$, $n_0=1$
  - $P(n) \in \Theta(n^k)$ para todo polinomio P de grado k

  Estas afirmaciones se pueden demostrar por inducción sobre n, una vez fijadas las constantes c –c_1 , c_2– y n_0.

- Se puede demostrar que

  $$\Theta(f) = O(f) \cap \Omega(f)$$

  o lo que es igual

  $$T \in \Theta(f) \Leftrightarrow T \in O(f) \wedge T \in \Omega(f)$$

1.3 Ordenes de complejidad

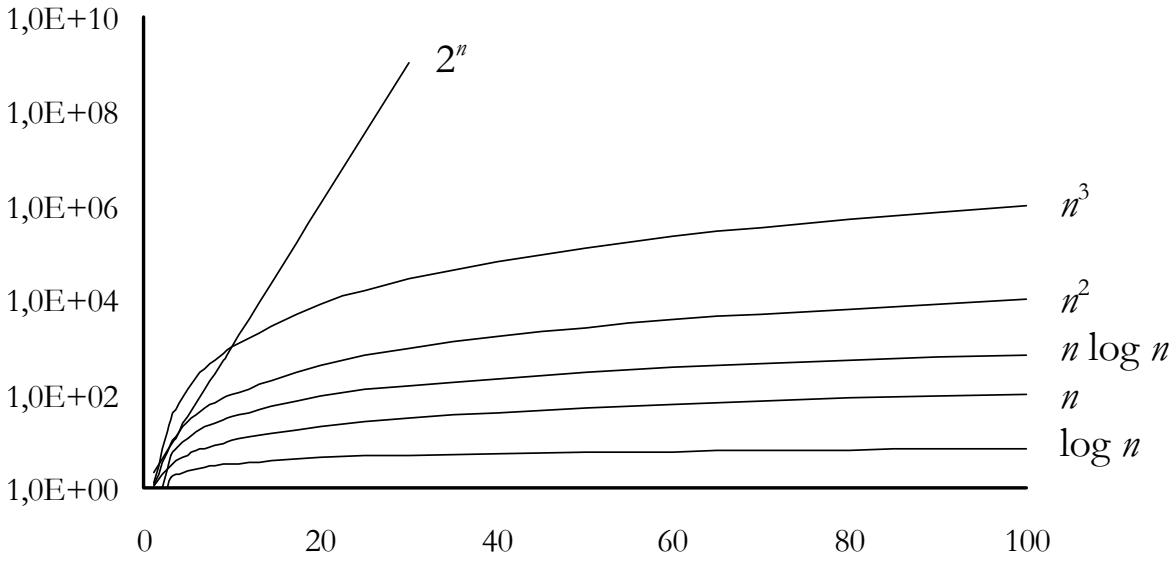
- Nos referiremos a las medidas asintóticas aplicadas a funciones concretas como *órdenes*.

  Algunos órdenes tienen nombres particulares:

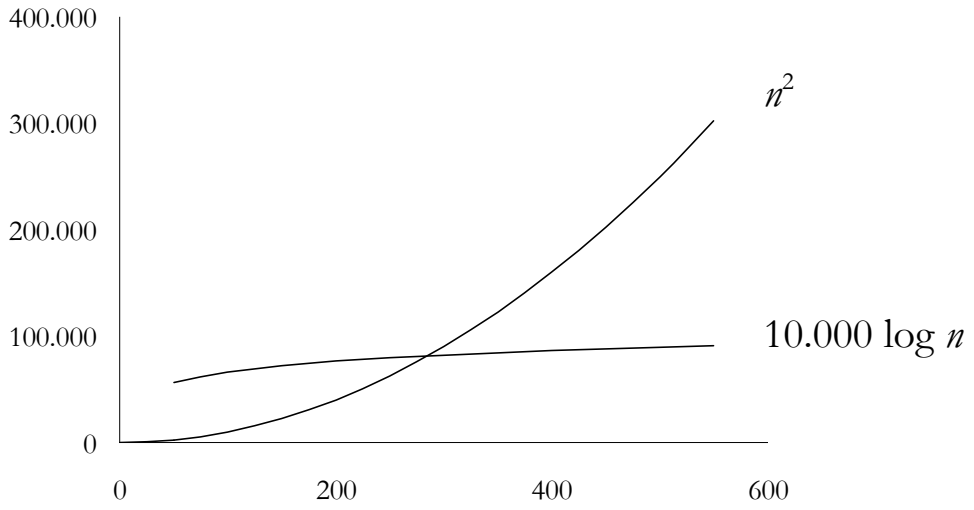
$O(1)$	→ orden constante	$O(n^2)$	→ orden cuadrático
$O(\log n)$	→ orden logarítmico	$O(n^3)$	→ orden cúbico
$O(n)$	→ orden lineal	$O(n^k)$	→ orden polinómico
$O(n \log n)$	→ orden cuasi-lineal	$O(2^n)$	→ orden exponencial

- Es posible demostrar que se cumplen las siguientes relaciones de inclusión entre los órdenes de complejidad, que permiten definir una *jerarquía de órdenes de complejidad* (a saberse):

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset$$
$$O(n^2) \subset O(n^3) \subset ... \subset O(n^k) \subset ...$$
$$O(2^n) \subset O(n!)$$



- Debemos recordar que los órdenes de complejidad son medidas asintóticas. Para datos pequeños, y si las constantes multiplicativas son distintas, nos podemos encontrar situaciones como ésta



Para determinar con exactitud las constantes multiplicativas, es necesario realizar medidas empíricas.

¿Cómo determinarías la función exacta a partir de medidas empíricas?

¿La eficiencia importa?

- ¿Para qué emplear tiempo en diseñar algoritmos eficientes si los ordenadores van cada vez más rápido?
  - En una computadora capaz de procesar un dato cada 10^{-4} seg., ¿cuánto se tarda en ejecutar un algoritmo A_1 de coste exponencial 2^n?

n	Tiempo
10	≈ 1 décima de seg.
20	≈ 2 min.
30	> 1 día
40	> 3 años

  - ¿y con una máquina que va 100 veces más rápido?, es decir, que procesa un dato en 10^{-6} seg

    Se tardarían *sólo* 35 años en procesar 50 datos

  - ¿Y un algoritmo A_2 de coste cúbico n^3 en la máquina antigua (10^{-4} seg. por dato)?

n	tiempo
10	≈ 1 décima de seg.
1000	≈ 1 día
7000	≈ 1 año

  - Por último, para un algoritmo de complejidad lineal n la máquina antigua es capaz de procesar 10.000 datos es un segundo.

- Dados dos algoritmos *A* y *B* que resuelven el mismo problema pero con complejidades diferentes:

  $T_A \in O(\log n)$ $\quad$ $T_B \in O(2^n)$

  – *A* puede consumir más memoria que *B*

  – Para *A,* aumentar "mucho" el tamaño de los datos aumenta "poco" el tiempo necesario; y aumentar "poco" el tiempo disponible –la velocidad de la máquina– aumenta "mucho" el máximo tamaño tratable:

  $$\log(2n) = 1 + \log n$$

  al doblar el tamaño de los datos, sólo se suma 1 al tiempo

  $$2 \cdot \log n = \log n^2$$

  al doblar la velocidad, los datos aumentan al cuadrado

  – Para *B,* aumentar "poco" el tamaño de los datos aumenta "mucho" el tiempo necesario; y aumentar "mucho" el tiempo disponible aumenta "poco" el máximo tamaño tratable

  $$2^{2n} = (2^n)^2$$

  duplicar el tamaño de los datos, eleva al cuadrado el tiempo necesario

  $$2 \cdot 2^n = 2^{n+1}$$

  duplicar el tiempo, sólo aumenta en 1 el máximo tratable

- Conclusión: Sólo un algoritmo eficiente, con un orden de complejidad bajo, puede tratar grandes volúmenes de datos.

  Se suele considerar: que un algoritmo es

  – muy eficiente si su complejidad es de orden $\log n$

  – eficiente si su complejidad es de orden n^k

  – ineficiente si su complejidad es de orden 2^n.

  Decimos que un problema es *tratable* si existe un algoritmo que lo resuelve con complejidad de orden menor que 2^n, y que es *intratable* en caso contrario.

- Aunque, en ocasiones, es más importante el tiempo del programador que el de la CPU

  – La eficiencia no es tan importante si, por ejemplo, el programa va a ejecutarse pocas veces o con datos de pequeño tamaño.

  – En cualquier caso, es conveniente partir de un diseño claro, aunque sea ineficiente, y optimizar posteriormente.

1.4 Métodos de análisis de la complejidad temporal de los algoritmos iterativos

- Si $T_1(n) \in O(f_1(n))$ y $T_2(n) \in O(f_2(n))$, se cumplen las siguientes propiedades

  – Regla de la suma

  $$T_1(n) + T_2(n) \in O(\max(f_1(n), f_2(n)))$$

  – Regla del producto

  $$T_1(n) * T_2(n) \in O(f_1(n) \cdot f_2(n))$$

  Para demostrarlo, usamos la definición de medida asintótica de cota superior:

  existen $c_1, c_2 \in \Re^+$; $n_1, n_2 \in \aleph$ tales que

  $\forall\ n \geq n_1\ T_1(n) \leq c_1 \cdot f_1(n)$ $\qquad$ $\forall\ n \geq n_2\ T_2(n) \leq c_2 \cdot f_2(n)$

  tomando $n_0 = \max(n_1, n_2)$

  – suma

  $$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 \cdot f_1(n) + c_2\, f_2(n) \\ &\leq (c_1+c_2) \cdot \max(f_1(n), f_2(n)) \end{aligned}$$

  por lo que para $n \geq n_0$ se cumple la propiedad

  – producto

  $$\begin{aligned} T_1(n) \cdot T_2(n) &\leq c_1 \cdot f_1(n) \cdot c_2\, f_2(n) \\ &\leq (c_1 \cdot c_2) \cdot (f_1(n) \cdot f_2(n)) \end{aligned}$$

  por lo que para $n \geq n_0$ se cumple la propiedad

- Veamos cuál es la complejidad para cada una de las instrucciones básicas de un lenguaje imperativo como C++.
  Esto es una simplificación que puede no funcionar en todos los casos.

- Si tenemos un algoritmo A de la forma

  – x = e

    entonces $T_A(n) \in O(1)$, excepto si la evaluación de e es compleja, en cuyo caso $T_A(n)$ será del orden de evaluar dicha expresión.

  – A_1 ; A_2 y $T_{Ai}(n) \in O(f_i(n))$
    entonces
    $$T_A(n) = T_{A1}(n)+T_{A2}(n)$$
    y por la regla de la suma $T_A(n) \in O(\max(f_1(n), f_2(n)))$

  – **if** (B_1) A_1 **else if** ... **else if** (B_r) **then** A_r y
    $$T_{Ai}(n) \in O(f_i(n)) \qquad (1 \le i \le n)$$
    entonces $T_A(n) \in O(\max(f_1(n), \ldots, f_r(n)))$
    Siempre que podamos suponer que la evaluación de las condiciones B_i consume tiempo constante. Si esta suposición no es razonable y sabemos que
    $$T_{Bi}(n) \in O(g_i(n)) \qquad (1 \le i \le n)$$
    entonces $T_A(n) \in O(\max(g_1(n), \ldots, g_r(n), f_1(n), \ldots, f_r(n)))$

– **while** (B) A_1 y

$T_{A1}(n) \in O(f(n))$ $T_B(n) \in O(g(n))$

tomamos el máximo de ambas

$h(n) =_{def} \max(g(n), f(n))$

de forma que el tiempo de ejecución de una vuelta del bucle será $O(h(n))$

Si podemos estimar que el número de vueltas en el caso peor es $O(t(n))$, entonces

$T_A(n) \in O(h(n) \cdot t(n))$

Generalmente es posible realizar una estimación más fina de $T_A(n)$, estimando por separado el tiempo de cada vuelta y calculando un sumatorio.

Si el tiempo de ejecución de la vuelta número i es $O(h(n, i))$ entonces:

$T_A(n) \in O(\Sigma\, i : 1 \leq i \leq t(n) : h(n, i))$

- Ejemplo

```
const int N = 10;
int v[N][N];
bool b;
int i, j;

  b = true;
  for ( i = 0; i < N-1; i++ )
    for ( j = i+1; j < N; j++ )
      b = b && (v[i][j] == v[j][i])
```

Como tamaño de los datos tomamos N.

Suponemos que todas las asignaciones, evaluación de condiciones y acceso a vectores tienen complejidad constante $O(1)$

- cuerpo del bucle interno: 2
- coste del bucle interno: $((N-i-1) \cdot (2+1)) + 1 = 3 \cdot (N-i) - 2$
- cuerpo del bucle externo: $(3 \cdot (N-i) - 2) + 1 = 3 \cdot (N-i) - 1$
- coste del bucle externo

$$[\sum_{i=0}^{N-2} (3 \cdot (N-i) - 1 + 1)] + 1$$

$$= 3 \cdot \sum_{i=0}^{N-2} (N-i) + 1$$

$$= 3 \cdot \sum_{i=0}^{N-2} N - 3 \cdot \sum_{i=0}^{N-2} i + 1 \qquad \sum_{i=\min}^{\max} i = \frac{\min+\max}{2} \cdot (\max-\min+1)$$

$$= 3 \cdot (N-1) \cdot N - 3 \cdot \frac{(N-2) \cdot (N-1)}{2} + 1$$

$$= 3\,N^2 - 3\,N - 3/2\,N^2 + 9/2\,N - 3 + 1$$

$$= 3/2\,N^2 + 3/2\,N - 2$$

- y el coste del algoritmo, considerando la asignación inicial

$$T_A(N) = 3/2\,N^2 + 3/2\,N - 2 + 1 = 3/2\,N^2 + 3/2\,N - 1$$

con lo que $T_A(N) \in O(N^2)$

- En lugar de las reglas generales, en muchos casos es suficiente estimar el orden de complejidad a partir de la *acción característica*. Las acciones características de un algoritmo son aquellas que se ejecutan con mayor frecuencia.

  El orden de magnitud de $T_A(n)$ se podrá estimar calculando el número de ejecuciones de las acciones características.

  – Ejemplo

```
const int N = 10;

bool esSim( int a[N][N] ) {
  int i, j;
  bool b;

  b = true;
  i = 0;
  while ( (i < N-1) && b ) {
    j = i+1;
    while ( (j < N) && b ) {
      b = b && ( a[i][j] == a[j][i] );
      j++;
    }
    i++;
  }
  return b;
}
```

  Consideramos como acciones características las asignaciones que componen el cuerpo del bucle más interno. El bucle interno se ejecuta $N–i–1$ veces para cada valor de i

$$
\begin{aligned}
T(N) &\approx \sum_{i=0}^{N-2} (N - i - 1) \\
&= \sum_{i=0}^{N-2} N - \sum_{i=0}^{N-2} i - \sum_{i=0}^{N-2} 1 \\
&= (N-1) \cdot N - \frac{(N-2)\cdot(N-1)}{2} - (N-1) \\
&= N^2 - N - 1/2\ N^2 + 3/2\ N - 1 - N + 1 \\
&= 1/2\ N^2 - 1/2\ N
\end{aligned}
$$

  Por lo tanto $T(N) \in O(N^2)$

  ¿Cuál de las dos versiones del algoritmo es más eficiente?

- Ejemplo

```
int x, y, p;

// Pre.: x == X, y == Y, y >= 0
p = 0;
while ( y != 0 ) {
  if ( (y % 2) == 0 ) {
    x = x + x;
    y = y / 2;
  }
  else {
    p = p + x;
    y = y - 1;
  }
}
// Post.: p = X*Y
```

Como tamaño de los datos tomamos $n = Y$

La acción característica es el cuerpo del bucle, que tiene complejidad constante $O(1)$. Por tanto, la complejidad del bucle será el número de vueltas

$T_A(n) \in O(t(n))$

Si en cada vuelta el tamaño de los datos se divide por 2 hasta llegar a 1, ¿cuántas vueltas da el bucle?

El tamaño de los datos en las sucesivas iteraciones será:

$n, n/2, n/4, \ldots, 1$

o lo que es igual

$n/2^0, n/2^1, n/2^2, \ldots, n/2^k$

siendo k el número de veces que se ha ejecutado el bucle, de forma que

$1 = n/2^k$

$\Leftrightarrow \quad 2^k = n$

$\Leftrightarrow \quad k = \log n$

Con lo que, si $t(n) \in O(\log n)$ entonces $T_A(n) \in O(\log n)$

¿ Este razonamiento es válido para cualquier valor de n ?

- Podemos demostrar este resultado.
  Estudiando algunos valores concretos de n

$n = 0:\quad y = 0$
$n = 1:\quad y = 1 \quad y = 0$
$n = 2:\quad y = 2 \quad y = 1 \quad y = 0$
$n = 3:\quad y = 3 \quad y = 2 \quad y = 1 \quad y = 0$
$n = 4:\quad y = 4 \quad y = 2 \quad y = 1 \quad y = 0$
$n = 5:\quad y = 5 \quad y = 4 \quad y = 2 \quad y = 1 \quad y = 0$
$n = 6:\quad y = 6 \quad y = 3 \quad y = 2 \quad y = 1 \quad y = 0$
$n = 7:\quad y = 7 \quad y = 6 \quad y = 3 \quad y = 2 \quad y = 1 \quad y = 0$
$n = 8:\quad y = 8 \quad y = 4 \quad y = 2 \quad y = 1 \quad y = 0$

  podemos formular la siguiente conjetura

$$t(n) \leq 2 \log n \qquad \text{para } n \geq 2$$

  que demostramos por inducción sobre n

  – casos base[1]

$n = 2$

$$t(n) = 2 = 2 \log 2$$

$n = 3$

$$t(n) = 3 \leq 2 \log 3$$

que se puede demostrar de la siguiente forma

$$\begin{aligned} & 3 \leq 2 \log 3 \\ \Leftrightarrow \quad & 2^3 \leq 2^{2 \log 3} \\ \Leftrightarrow \quad & 2^3 \leq (2^{\log 3})^2 \\ \Leftrightarrow \quad & 2^3 \leq 3^2 \\ \Leftrightarrow \quad & 8 \leq 9 \\ \Leftrightarrow \quad & \text{cierto} \end{aligned}$$

[1] No podemos usar el 1 como caso base porque log 1 = 0; y necesitamos incluir el caso base del 3 porque no entra en los casos inductivos ($n = 2n' + 1$, $n' \geq 2$).

– pasos inductivos

  – $n > 3$, par $\quad n = 2n', n' \geq 2$

    HI: $t(n') \leq 2 \log n'$

    Para llegar al valor n' desde el valor n hay que dar una sola pasada por el bucle, por lo tanto:

$$\begin{aligned} t(n) = 1 + t(n') &\leq_{HI} 1 + 2 \log n' \\ &< 2 \cdot (1 + \log n') \\ &= 2 \cdot \log(2 \cdot n') \\ &= 2 \cdot \log n \end{aligned}$$

  – $n > 3$, impar $\quad n = 2n' + 1, n' \geq 2$

    HI: $t(n') \leq 2 \log n'$

    Para llegar al valor n' desde el valor n hay que dar dos pasadas por el bucle, una para llegar a $2n'$ y otra para llegar a n', por lo tanto:

$$\begin{aligned} t(n) = 2 + t(n') &\leq_{HI} 2 + 2 \log n' \\ &= 2 \cdot (1 + \log n') \\ &= 2 \cdot \log(2 n') \\ &< 2 \cdot \log(2n' + 1) \\ &= 2 \cdot \log n \end{aligned}$$