Notación O-Mayúscula

- Decimos que una función $T(n)$ es $O(f(n))$ si existen constantes n_0 y c tales que $T(n) \leq cf(n)$ para $n \geq n_0$:

$$T(n) \text{ es } O(f(n)) \Leftrightarrow \exists\, c \in \mathrm{R},\ \exists\, n_0 \in \mathrm{N},\ \text{tq } \forall n \geq n_0,\ T(n) \leq cf(n)$$

- Ejemplos:
  - $T(n) = (n + 1)^2$ es $O(n^2)$
  - $T(n) = 3n^3 + 2n^2$ es $O(n^3)$
  - $T(n) = 3^n$ no es $O(2^n)$

- *Flexibilidad en la notación*: Emplearemos $O(f(n))$ aun cuando en un número finito de valores de n $f(n)$ sea negativa o no esté definida. Ej: $n/\log_2 n$ no está definida para $n = 0$ ó $n = 1$.

- **Regla de la suma**
  Sean $T_1(n)$ y $T_2(n)$ los tiempos de dos trozos de código tales que

  $$T_1(n) \text{ es } O(f(n))$$
  $$T_2(n) \text{ es } O(g(n))$$

  entonces

  $$T_1(n) + T_2(n) \text{ es } O(\max(f(n), g(n)))$$

- **Regla del producto**
  Sean $T_1(n)$ y $T_2(n)$ los tiempos de dos trozos de código tales que

  $$T_1(n) \text{ es } O(f(n))$$
  $$T_2(n) \text{ es } O(g(n))$$

  y que ninguna es negativa para ningún n, entonces

  $$T_1(n)T_2(n) \text{ es } O(f(n)g(n))$$

Ejemplo de uso

```
[redacted] cin >>n;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[i][j] = 0;
for (k = 0; k < n; k++)
  A[k][k] = 1;
```

Regla de la suma

$$\left.\begin{array}{l} T_1(n) \text{ es } O(f(n)) \\ T_2(n) \text{ es } O(g(n)) \end{array}\right\} T_1(n)+T_2(n) \text{ es } O(\max(f(n), g(n)))$$

Regla del producto

$$\left.\begin{array}{l} T_1(n) \text{ es } O(f(n)) \\ T_2(n) \text{ es } O(g(n)) \end{array}\right\} T_1(n)\,T_2(n) \text{ es } O(f(n)\,g(n))$$

OPERACIÓN ELEMENTAL

Definición: Operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante.

Para nuestro análisis sólo contará el número de operaciones elementales ejecutadas y no el tiempo exacto necesario para cada una de ellas.

Consideraciones:

1. En la descripción de un algoritmo puede ocurrir que una línea de código corresponda a un número variable de operaciones elementales. Por ejemplo, si `A` es un vector con `n` elementos, y queremos calcular
   `x = max{A[k], 1 ≤ k ≤ n}`,
   el tiempo para hacerlo depende `n`, no es constante.

2. Algunas operaciones matemáticas no deben ser tratadas como tales operaciones elementales. Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos. Sin embargo, en la práctica se consideran elementales siempre que los datos que se usen tengan un tamaño razonable.

No obstante, desde el punto de vista teórico, consideraremos las operaciones suma, diferencia, multiplicación, división, módulo, operaciones booleanas, comparativas y asignaciones como elementales, y por tanto de costo unidad, a no ser que explícitamente se establezca otra cosa.

Reglas para el cálculo del tiempo de ejecución

1. Sentencias simples

Consideraremos que cualquier sentencia simple (lectura, escritura, asignación, ...) va a consumir un tiempo constante, O(1), salvo que la sentencia contenga una llamada a función.

```
for (i = 0; i < n-1; i++) {
  indice = i;
  for (j = i+1; j < n; j++)
    if (A[j] < A[indice])
      indice = j;
    temporal = A[indice];
    A[indice] = A[i];
    A[i] = temporal;
};
```

2. Bucles

El tiempo invertido en un bucle es la suma del tiempo invertido en cada iteración. Este tiempo debería incluir tanto el tiempo propio del cuerpo como el asociado a la evaluación de la condición y su actualización, en su caso. Si se trata de una condición sencilla (sin llamadas a función) el tiempo es O(1), que no es relevante de cara al cálculo asintótico. Por eso, usualmente, se obvia este valor.

Cuando se trata de un bucle simple, o sea, todas las iteraciones son iguales, entonces el tiempo total será el producto del número de iteraciones por el tiempo que requiere cada vuelta.

Ejemplo 1:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    A[i][j] = 0;
```

$$\sum_{j=0}^{n-1} 1 = n = n \times 1$$

$$\sum_{i=0}^{n-1} n = n^2 = n \times n$$

```
for (i=0; i<n; ++i)
    a[i]=0;
```

Nº operaciones elementales reales

$$\underbrace{1}_{\text{Asignación antes de la ejecución del bucle}} + \underbrace{1}_{\text{Evaluación de la condición antes de entrar al bucle}} + \underbrace{n}_{\text{número de iteraciones}} (\underbrace{2}_{\text{dos operaciones para el cuerpo del bucle: indexación + asignación}} + \underbrace{2}_{\text{operación de incremento y evaluación de la condición}}) = 4n+2$$

En la práctica

$$\underbrace{1}_{\text{Resume las operaciones elementales para cada iteración del bucle}} \cdot \underbrace{n}_{\text{número de iteraciones}} + \underbrace{1}_{\text{Resume las operaciones elementales antes de la ejecución del bucle}} = n+1$$

+ simple

n operaciones elementales

Ejemplo 2:

```
k = 0;
while (k < n && A[k] <> Z)
  k++;
```

3. Sentencias IF-ELSE

El tiempo de ejecución es el máximo tiempo de la parte `if` y de la parte `else`, de forma que si son respectivamente, $O(f(n))$ y $O(g(n))$ será $O(\text{máx}(f(n), g(n)))$. Si el tiempo de la condición es $O(1)$ y no está inmerso en bucles o llamadas a función, se desprecia.

Ejemplo:

```
if (A[0][0] == 0)
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        A[i][j] = 0;
  else
    for (k =0; k < n; k++)
      A[k][k] = 1;
```

4. Bloques de sentencias

Se aplica la regla de la suma, de forma que se calcula el tiempo de ejecución tomando el máximo de los tiempos de ejecución de cada una de las partes (sentencias individuales, bucles o condicionales) en que puede dividirse.

5. Llamadas a Funciones

Si una determinada función P tiene una eficiencia de $O(f(n))$ con n la medida del tamaño de los argumentos, cualquier función o procedimiento que llame a P tiene en la llamada una cota superior de eficiencia de $O(f(n))$.

Más precisamente:

- Las asignaciones con llamadas a función deben sumar las cotas del tiempo de ejecución de cada llamada.

- La misma consideración anterior es válida para las condiciones de bucles y condicionales. En el caso de los bucles habrá que sumar esta cota al tiempo total tantas veces como iteraciones haga el bucle. Tener en cuenta que los bucles `while` siempre hacen una evaluación más del número de iteraciones.
- Si la llamada a función está en la inicialización de un blucle `for`, sumar su coste al tiempo total del bucle.

Ejemplo:

```
int funcion1(int n)
{
  int i, x;

  for (i = 0, x = 0; i < n; i++)
    x += funcion2(i, n);

  return x;
}

int funcion2(int x, int n)
{
  int i;

  for (i = 0; i < n; i++)                     1
    x += i;

  return x;
}

int main()
{
  int control, n;

  [redacted] cin >>n;
  control = funcion1(n);
  [redacted] cout << control <<endl;

  return 1;
}
```

UN PRIMER EJEMPLO

```
[1] cin >>n;                          O(1)
[2] for (i=0; i<n; i++)
[3]     for (j=0; j<n; j++)
[4]         A[i][j]=0;   O(1)
[5] for (k=0; k<n; k++)
[6]     A[k][k]=1;   O(1)
```

[3]–[4]: $O(n \times 1) = O(n)$

[2]–[4]: $O(n \times n) = O(n^2)$

[5]–[6]: $O(n \times 1) = O(n)$

$O(\max(1, n^2, n)) = O(n^2)$

Asignación [1]

$O(1)$

Bucle for [2]–[4] $O(n^2)$

↓

Bucle for [3]–[4] $O(n)$

↓

Asignación [4] $O(1)$

Regla del producto

Bucle for [5]–[6] $O(n)$

↓

Asignación [6] $O(1)$

Regla del producto

$O(\max(1, n^2, n)) =$

$O(n^2)$

Regla de la suma

↓

$\underline{\underline{O(n^2)}}$ → Eficiencia cuadrática

```
#include <iostream[redacted]>
#include <math[redacted]>

int main(){
    double a, b, c;
    double r1, r2;
 cout <<"\n Introduce coeficiente de 2 grado:";  O(1)
 cin >>a;   O(1)
cout << "\n Introduce coeficiente de 1 grado:";  O(1)      O(1)
 cin>>b;  O(1)
cout <<"\n Introduce termino independiente:";  O(1)
 cin>>c;  O(1)
if (a!=0){ O(1)
                 O(1)   O(1)     O(1)           O(1)
      r1=(-b + sqrt(b*b - 4*a*c))/(2*a);   O(1)
      r2= (-b- sqrt(b*b-4*a*c))/(2*a);   O(1)                O(1)
      cout <<"Las raices son" << r1 << "y"
                                                 O(1)           O(1)
              << r2 << endl;
    }
else {
      r1=-c/b    O(1)
      cout <<"la única raiz es      O(1)
                   " <<r1 << endl;
   }
}
```

```
int BusSecuencial1 (const int v[], int n, int elemento) {

    int i, posicion;
    bool encontrado;

    i=0;                    O(1)
    encontrado = false;     O(1)                              O(1)

    while ((i<n) && (!encontrado))
                O(1)   O(1)   O(1)
        if v[i] == elemento   O(1)
            {
            posicion =i;   O(1)
            encontrado = true;   O(1)        O(1)      O(nx1)=O(n)
            }                                           (*)
        else
            i++;   O(1)

    if(!encontrado)   O(1)
        return-1;   O(1)                              O(1)
    else
        return posicion;   O(1)

}
                                                                  O(n)
```

(*) El número de iteraciones se determina teniendo en cuenta que el caso peor se da cuando se recorre el vector de $\boxed{n}$ elementos y no se encuentra el elemento buscado.

```
int busbinaria (const int v[], int [redacted] n, int elemento) {
    int izq, dch, centro;

    izq = 0;                    -> O(1) ┐
    dch = [redacted] n - 1;     -> O(1) │ O(1)
    centro = (izq + dch)/2;     -> O(1) ┘

    while ((izq <= dch) &&                 O(1)
           (v[centro] != elemento)) {      O(1)
        if (elemento < v[centro])  -> O(1) ┐
            dch = centro - 1;      -> O(1) │ O(1)
        else                               │
            izq = centro + 1;      -> O(1) ┘
        centro = (izq + dch)/2;    -> O(1)
    }                                      } O(log_2 n) (*)
    if (izq > dch)              -> O(1) ┐
        return -1;              -> O(1) │ O(1)
    else                                │
        return centro;          -> O(1) ┘
}                                       } O(log_2 n)
```

(*) El caso peor se da cuando se particiona el vector de (n) elementos por la mitad hasta que no sea posible hacerlo mas, y no se encuentra el elemento, en total $\log_2 n$ veces.

UNA RAPIDA REVISION MATEMATICA

- LOGARITMOS Y EXPONENTES
  - propiedades de los logaritmos:

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x^\alpha = \alpha \log_b x$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

  - propiedades de las exponenciales:

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b/a^c = a^{(b-c)}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c*\log_a b}$$

SEGUIMOS CON LA REVISION:

$\lfloor x \rfloor \equiv$ mayor entero $\leq x$

$\lceil x \rceil \equiv$ menor entero $\geq x$

- Sumatorias:
  - definición general:

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \dots + f(t)$$

con f una función, s el índice inicial y t el índice final

- Progresión geométrica $f(i) = a^i$
  - Dado un entero $n \geq 0$ y un número real a ($0 < a \neq 1$)

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \dots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

- Progresión aritmética $f(i) = i$

Dado un entero $n > 1$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \dots + n = \frac{1+n}{2} \cdot n$$

- Suma de cuadrados

$\forall n \geq 1 \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

```
void seleccion (matriz A, int n)
  {
    int i, j, min, t;
    for (i=0; i<n; i++)
      {
        min=i;                          → O(1)
        for (j=i+1; j<n+1; j++)                     → O(n-i)
          if (A[j] < A[min])   → O(1)
              min = j;         → O(1)                  O(n-i)
        t = A[min];                     → O(1)
        A[min] = A[i];                  → O(1)
        A[i] = t;                       → O(1)
      }
  }
```

$$\sum_{i=0}^{n-1}(n-i) = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n(n) - \frac{n}{2}(n-1)$$

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

$O(n^2)$

```
void intercambiar (int &a, int & b){
        int aux=a;  → O(1)
        a = b;      → O(1)     } O(1)
        b = aux     → O(1)
    }

void ordseleccion (int v[], int [redacted] n){
        int i, j, min;

    for (i=0; i< [redacted] n -1; i++){
          min = i;  → O(1)
          for (j=i+1; j< [redacted] n; j++)
              if (v[j] < v[min]) O(1)  } O(1)  } → O(n-i-1)
                  min=j; → O(1)
          intercambiar (v[i], v[min]); → O(1)
    }
}
```

$$\boxed{\begin{gathered} O((n-1) \times (n-i-1)) \\ \shortparallel \\ O(n^2) \end{gathered}} \; ?$$

$$\sum_{i=0}^{n-2} (n-i-1) = \underbrace{\sum_{i=0}^{n-2} n}_{n(n-1)} - \underbrace{\sum_{i=0}^{n-2} i}_{\frac{0+(n-2)}{2}\cdot(n-1)} - \underbrace{\sum_{i=0}^{n-2} 1}_{(n-1)} \rightarrow \underline{\underline{O(n^2)}}$$

```
void ejemplo2 (int n)
{
 int i, j, k;
 for (i=1 ; i<N ; i++)                 → O(n-1)
   for (j=i+1; j<N+1; j++)             → O(n-i)
     for (k=1; k<j+1; k++)             → O(j)
       /* Alguna sentencia O(1) */
}
```

→ j veces

$$\rightarrow \sum_{j=i+1}^{n} j = \frac{n+(i+1)}{2}(n-i) = \tfrac{1}{2}(n^2+n-i^2-i)$$

$$\rightarrow \frac{1}{2}\underbrace{\sum_{i=1}^{n-1}[n^2+n-i^2-i]}$$

$$\hookrightarrow \sum_{i=1}^{n-1} n^2 + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i^2 - \sum_{i=1}^{n-1} i$$

$$\sum_{i=1}^{n-1} n^2 = n^2(n-1)$$

$$\sum_{i=1}^{n-1} n = n(n-1)$$

$$\sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6}$$

$$\sum_{i=1}^{n-1} i = \frac{1+(n-1)}{2}\cdot n-1 = \frac{n(n-1)}{2}$$

$$n^2(n-1) + n(n-1) - \frac{(n-1)n(2n-1)}{6} - \frac{n(n-1)}{2}$$

$$O(n^3)$$

```
void ejemplo3 (int n)
 { int i,j,x,y;
  for (i=1; i<n+1; i++)
     if (i%2)=0
        {
        for (j=i; j<n+1; j++)
            x++;
        for (j=1; j<i+1; j++)
            y++;
        }
  }
```

$O(\frac{n}{2} \cdot (n+1))$

$O(1)$ — $O(n-i+1)$

$O(1)$ — $O(i)$

$\rightarrow O(n-i+1) + O(i) = O(n+1)$

se ejecuta $n/2$ veces, luego es:

$O(\frac{n}{2}(n+1))$ y por tanto $\underline{O(n^2)}$

```
void cualeslaeficiencia (int n)
   {
    int x, contador;
    contador = 0;  ──────────→ O(1)
    x = 2;  ─────────────────→ O(1)
    while (x <= n)
        {
         x = 2 * x;  → O(1)        O(?)        O(?)
         contador++;  → O(1)
        }
    cout << contador;  ──────→ O(1)
   }
```

```
void ejemplo2 (int n)
  {
  int i, j, k;
  for (i=0; i<n; i++)
     for (j=0; j<n; j++)
        {
          C[i][j]=0;                                  O(1)
          for (k=0; k<n; k++)                                O(n)     O(n)
            C[i,j] += A[i][k] * B[k][j];  O(1)
        }
     }
```

$O(n \times n)$

$=$

$O(n^2)$

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

$\underline{\underline{O(n^3)}}$

PRODUCTO DE MATRICES

$$\left.\begin{array}{l} M_1 \rightarrow 10 \times 20 \\ M_2 \rightarrow 20 \times 50 \\ M_3 \rightarrow 50 \times 1 \\ M_4 \rightarrow 1 \times 100 \end{array}\right\} \quad M_1 \times M_2 \times M_3 \times M_4$$

a)

$$\underbrace{\underbrace{\underbrace{\underset{(10\times20)}{M_1} \times \underset{(20\times50)}{M_2}}_{\substack{M_{12}\ [10.000] \\ (10\times50)}} \times \underset{(50\times1)}{M_3}}_{\substack{M_{123}\ [500] \\ (10\times1)}} \times \underset{(1\times100)}{M_4}}_{\substack{M_{1234}\ [1.000] \\ (10\times100)}} \quad \Bigg\} \ [11.500]$$

b)

$$\underbrace{\underbrace{\underset{(10\times20)}{M_1} \times \underbrace{\underset{(20\times50)}{M_2} \times \underset{(50\times1)}{M_3}}_{\substack{M_{23}\ [1.000] \\ (20\times1)}}}_{\substack{M_{123}\ [200] \\ (10\times1)}} \times \underset{(1\times100)}{M_4}}_{\substack{M_{1234}\ [1.000] \\ (10\times100)}} \quad \Bigg\} \ [2.200]$$