

Introducción al análisis de algoritmos

Programación de estructuras de datos y algoritmos fundamentales

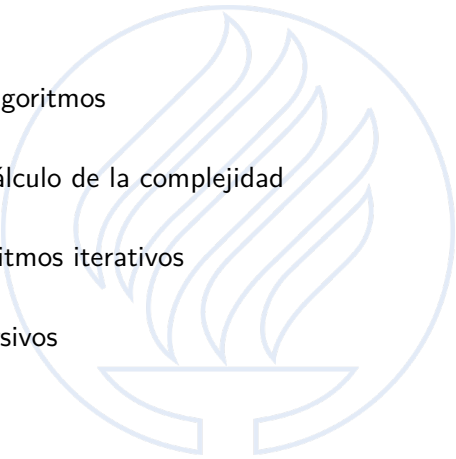
Francisco J. Navarro B. BEng, MSc, PhD
fj.navarro.barron@tec.mx

Tecnológico de Monterrey

08-2022



Contenido

- 
- 1 Análisis de los algoritmos
 - 2 Reglas para el cálculo de la complejidad
 - 3 Análisis de algoritmos iterativos
 - 4 Algoritmos recursivos

1 Análisis de los algoritmos

¿Cómo analizamos los algoritmos?

¿Big Ω , Big Θ ?, Big O ?

Jerarquía de los algoritmos

Complejidad vs. tiempo

2 Reglas para el cálculo de la complejidad

3 Análisis de algoritmos iterativos

4 Algoritmos recursivos



¿Cómo analizamos los algoritmos?

Cuando tenemos varios algoritmos para resolver un mismo problema, necesitamos una forma de determinar la mejor opción.



¿Cómo analizamos los algoritmos?

- Evaluamos según complejidad computacional → mejor rendimiento.
 - Menor tiempo de ejecución.
 - Menor utilización de memoria.
 - Menor utilización de energía.

¿Cómo analizamos los algoritmos?

- Pero, ¿qué es la **complejidad** de un algoritmo?
 - Es la medida de los recursos que necesita un algoritmo para su ejecución.
 - Complejidad temporal: El tiempo que necesita un algoritmo para terminar su ejecución.
 - Complejidad espacial: La cantidad de memoria que requiere un algoritmo durante su ejecución.

- El tiempo de ejecución de un algoritmo depende de:
 - **Factores externos:** La computadora donde se va a realizar la ejecución, el compilador (o interprete) usado, la experiencia del programador, los datos de entrada.
 - Factores internos: El número de instrucciones asociadas al algoritmo.
- Entonces, ¿cómo podemos estudiar el tiempo de ejecución del algoritmo?

Análisis empírico (a *posteriori*):

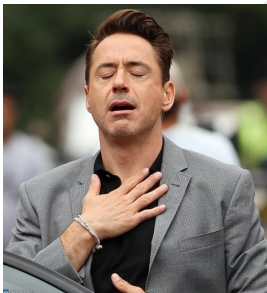
- Generando ejecuciones del algoritmo para distintos valores de entrada y cronometrando el tiempo de ejecución.
- Factores internos: Los resultados dependen de factores externos e internos.

¿Cómo analizamos los algoritmos?



Entonces, ¿tenemos que implementar cada uno de los algoritmos y medir su tiempo de ejecución para entonces decidir entre ellos?

¿Cómo analizamos los algoritmos?



- NO! en CS realizamos **análisis asintótico de complejidad**.
- Pseudocódigo (no implementación).
- Del pseudocódigo estimamos los “pasos” que toma en función del tamaño de la entrada.

¿Cómo analizamos los algoritmos?

- Análisis analítico (a *priori*):
 - Obtener una función que represente el tiempo de ejecución del algoritmo para cualquier valor de entrada.
 - Depende solo de los factores internos.
- Esto hace que nuestra estimación sea solo dependiente del algoritmo y no de otros factores como:
 - Frecuencia de operación del procesador
 - Cantidad de memoria de la computadora
 - Implementación del programador

¿Qué es un “paso”?

Un “paso” es una operación cuyo tiempo de ejecución no depende de los valores.

- Operación aritmética, lógica o de comparación.
- Asignación de valor
- Acceso a un elemento de un *array*

Ejemplo 1

Ejemplo. 1

Input: A, un arreglo de tamaño n

Output: ?

```
if (n > 100){  
    a = A[0] + A[25] + A[50] + A[75] + A[100];  
}  
else{  
    a = 42;  
}  
return a;
```

¿Tengo que analizar todas las rutas de ejecución y reportar todos los números de casos para cada una?

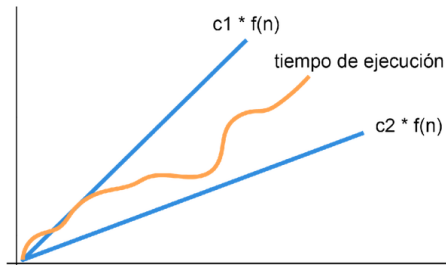
¿Big Ω , Big Θ ?, Big O ?

Cuando analizamos un algoritmos debemos tener en cuenta tres situaciones:

- El mejor de los casos: Cota inferior - $\Omega(n)$
- El caso promedio: Cota promedio - $\Theta(n)$
- El peor de los casos: Cota superior - $O(n)$

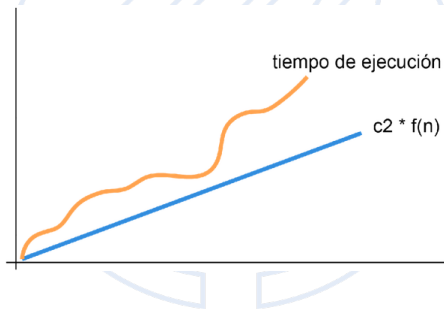
Big Θ

La notación *Big Θ* sirve para acotar de manera asintótica el crecimiento de un tiempo de ejecución dentro de un rango.



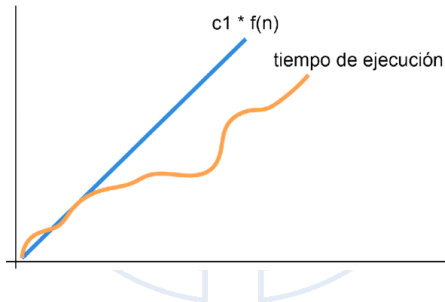
Big Ω

La notación *Big Ω* se usa cuando se desea decir que un algoritmo toma por lo menos cierta cantidad de tiempo, sin querer ofrecer la cota superior.



Big O

La notación *Big O* sirve para acotar de manera asintótica el crecimiento de un tiempo de ejecución por “arriba”.



Jerarquía de los algoritmos

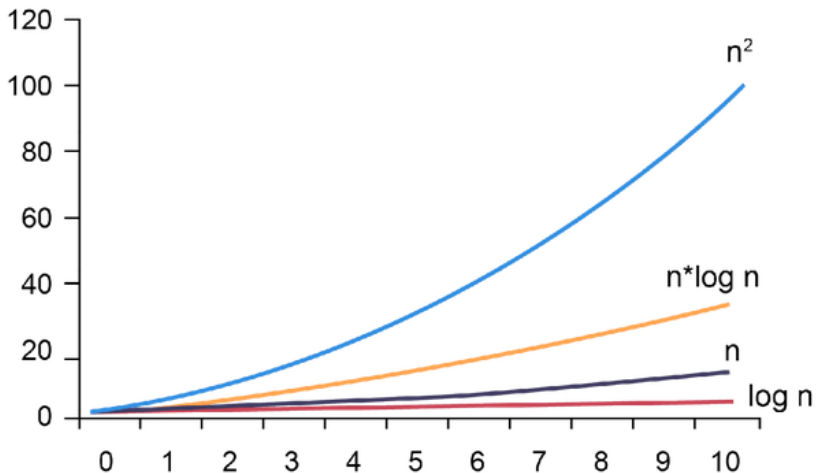
Las principales notaciones *Big O* que se tienen en los algoritmos son:

Notación O	Nombre
$O(1)$	Constante
$O(\log \log(n))$	log log
$O(\log(n))$	Logarítmica
$O(n)$	Lineal
$O(n \log(n))$	n log n
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(n^m)$	Polinomial
$O(m^n) m \geq 2$	Exponencial
$O(n!)$	Factorial

Complejidad vs. tiempo

N	10	100	1,000	10,000	100,000
$O(1)$	$1 \mu s$	$1 \mu s$	$1 \mu s$	$1 \mu s$	$1 \mu s$
$O(\log n)$	$3 \mu s$	$7 \mu s$	$10 \mu s$	$13 \mu s$	$17 \mu s$
\sqrt{n}	$3 \mu s$	$10 \mu s$	$31 \mu s$	$100 \mu s$	$316 \mu s$
n	$10 \mu s$	$100 \mu s$	$1,000 \mu s$	$10,000 \mu s$	$100,000 \mu s$
$n \log n$	$33 \mu s$	$664 \mu s$	$10,000 \mu s$	$133,000 \mu s$	1.6 seg
n^2	$100 \mu s$	$10,000 \mu s$	1 seg	1.7 min	16.7 min
n^3	1 ms	1 seg	16.7 min	11.6 día	31.7 año
2^n	1.024 ms	$4*10^{16} \text{ año}$	$3.39*10^{287} \text{ año}$
$n2^n$	10.24 ms	$4*10^{18} \text{ año}$
$n!$	4 seg	$2.95*10^{144} \text{ año}$

Crecimiento de las notaciones Big- O más básicas



1 Análisis de los algoritmos

2 Reglas para el cálculo de la complejidad

Sentencias simples

Condicionales

Ciclos

Procedimientos

3 Análisis de algoritmos iterativos

4 Algoritmos recursivos



Sentencias simples

La sentencias simples son aquellas que ejecutan operaciones básicas, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño está relacionado con el tamaño del problema. La inmensa mayoría de las sentencias simples requieren un tiempo constante de ejecución y su complejidad es $O(1)$.

Ejemplos:

$x \leftarrow 1$

$y \leftarrow z + x + w$

`print(x)`

`read(x)`

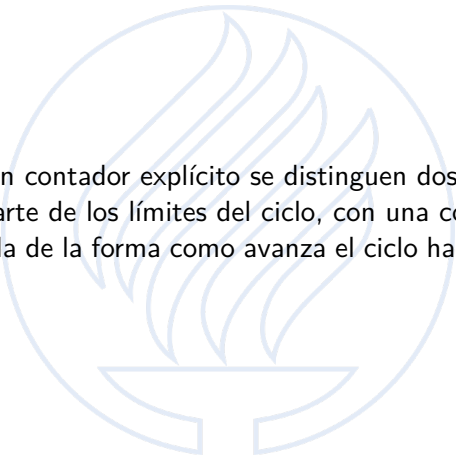
Condicionales

Los condicionales suelen ser $O(1)$, a menos que involucren un llamado a un procedimiento, y siempre se debe tomar la peor complejidad posible de las alternativas del condicional, bien en la rama afirmativa o bien en la rama positiva. En decisiones múltiples (*switch*) se tomará la peor de todas las ramas.

Ejemplos:

```
if  $a > b$  then
  for  $i = 1, \dots, n$  do
     $sum \leftarrow sum + 1$ 
  end for
else
   $sum \leftarrow 0$ 
end if
```

Ciclos (while, for, repeat-until)



En los ciclos con un contador explícito se distinguen dos casos: que el tamaño n forme parte de los límites del ciclo, con una complejidad basada en n , o que dependa de la forma como avanza el ciclo hacia su terminación.

Ciclos (while, for, repeat-until)

Si el ciclo se realiza un número constante de veces, independientemente de n , entonces la repetición solo introduce una constante multiplicativa que puede absorberse, lo cual da como resultado $O(1)$.

Ejemplo:

```
for  $i = 1, \dots, k$  do
    sentencias simples  $O(1)$ 
end for
```

Si el tamaño n aparece como límite de las iteraciones, entonces la complejidad será: $n \times O(1) \rightarrow O(n)$.

Ciclos (while, for, repeat-until)

Si los ciclos son anidados...

Ejemplo:

```

for  $i = 1, \dots, n$  do
  for  $j = 1, \dots, n$  do
    sentencias simples  $O(1)$ 
  end for
end for
  
```

En este caso, la complejidad sería: $n \times n \times O(1) \rightarrow O(n^2)$.

Ciclos (while, for, repeat-until)

Para ciclos anidados pero con variables independientes:

```
for  $i = 1, \dots, n$  do
  for  $j = 1, \dots, i$  do
    sentencias simples  $O(1)$ 
  end for
end for
```

En este caso, la complejidad sería:

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

Ciclos (while, for, repeat-until)

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

```
 $c \leftarrow 1$   
while  $c < n$  do  
     $c \leftarrow c \times 2$   
end while
```

El valor inicial de la variable c es 1, y llega a 2^n al cabo de n iteraciones
 $\rightarrow \log_2 n$.

Ciclos (while, for, repeat-until)

Y la combinación de los anteriores en el siguiente ejemplo:

```
for  $i = 1, \dots, n$  do
   $c \leftarrow n$ 
  while  $c > 0$  do
     $c \leftarrow c/2$ 
  end while
end for
```

Se tiene un ciclo interno de orden $O(\log_2 n)$ que se ejecuta n veces en el ciclo externo; por lo que, el ejemplo es de orden $O(n \log_2 n)$.

Llamada a procedimientos

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí.

Ejemplo:

$a \leftarrow 10$

$b \leftarrow 20$

$c \leftarrow \text{FACTORIAL}(a)$

$z \leftarrow a + b + c$

Llamada a procedimientos

Si se tiene un ciclo con un llamado a una función.

Ejemplo:

```
for  $i = 1, \dots, n$  do
     $x \leftarrow \text{FACTORIAL}(i)$ 
end for
```

Si hay un ciclo que se realiza n veces, lo que generaría una complejidad $O(n)$; pero como en su interior hay un llamado a la función *FACTORIAL*, la complejidad del ciclo es multiplicado por la complejidad de la función; en este caso sería $O(n) \times O(n) \rightarrow O(n^2)$

Llamada a procedimientos

Si hay dos o más llamadas a funciones.

Ejemplo:

`QUICKSORT(array, n)`

`DISPLAY(array, n)`

- La complejidad del *QUICKSORT* es $O(n \log_2 n)$
- Dado que *DISPLAY* simplemente muestra el contenido del arreglo en la pantalla este tiene una complejidad de $O(n)$
- \therefore la complejidad total será mayor de los dos llamadas a las funciones: $O(n \log_2 n)$.

Actividad

Ejemplo 1: Dada una lista de n enteros (desordenados), devolver *true* si todos son divisibles por uno de ellos.

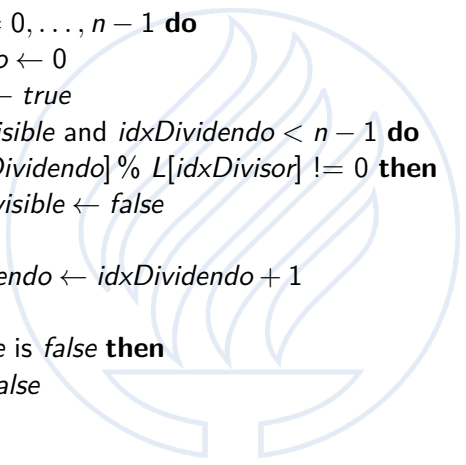


Algoritmo 1

Alg. 1: Encuentra el *min*, checa divisibilidad

```
m ← L[0]
for i = 1, ..., n − 1 do
    if L[i] < m then
        m ← L[i]
    end if
end for
for i = 0, ..., n − 1 do
    if L[i] % m != 0 then
        return false
    end if
end for
return true
```

Algoritmo 2: Brute Force



```
for  $idxDivisor = 0, \dots, n - 1$  do  
     $idxDividendo \leftarrow 0$   
     $esDivisible \leftarrow true$   
    while  $esDivisible$  and  $idxDividendo < n - 1$  do  
        if  $L[idxDividendo] \% L[idxDivisor] \neq 0$  then  
             $esDivisible \leftarrow false$   
        end if  
         $idxDividendo \leftarrow idxDividendo + 1$   
    end while  
    if  $esDivisible$  is false then  
        return false  
    end if  
end for  
return true
```

1 Análisis de los algoritmos

2 Reglas para el cálculo de la complejidad

3 Análisis de algoritmos iterativos

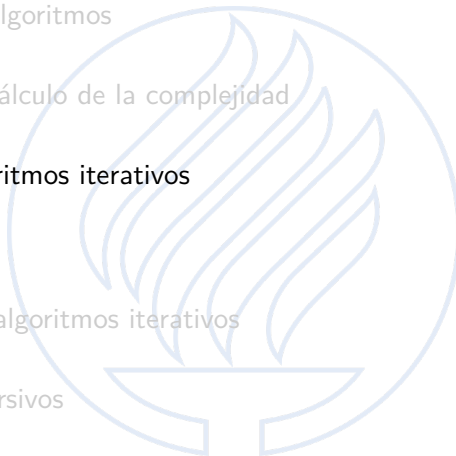
Secuencia

Condional

Ciclos

Ejemplos de algoritmos iterativos

4 Algoritmos recursivos



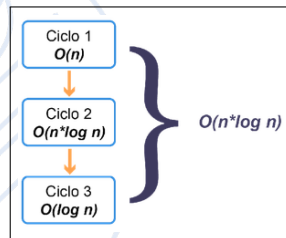
Análisis de algoritmos iterativos

Bajo la notación de Big- O , los algoritmos iterativos normalmente son polinomiales para los cuales se analiza el algoritmo desde lo más **interno** hasta lo mas **externo**; siguiendo 3 reglas fundamentales:

- Secuencia
- Condicional
- Ciclos

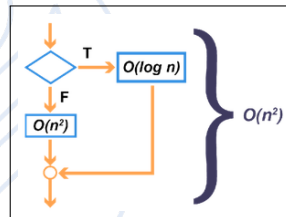
Secuencia

Cuando se tienen ciclos en forma secuencial se selecciona la que tenga un orden mayor, es decir, la que impacta con mayor fuerza a la secuencia de instrucciones.



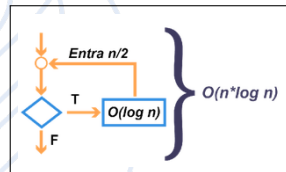
Condicional

Cuando se tienen condicionales (**if**) con 2 partes (parte verdadera y falsa), se selecciona la que tenga un orden mayor. Si solo existe la parte verdadera, se toma directamente como el orden de la condicional.



Ciclo

El orden de los ciclos será la cantidad de veces que se realiza el ciclo por el orden de las operaciones internas que se encuentran dentro de él.

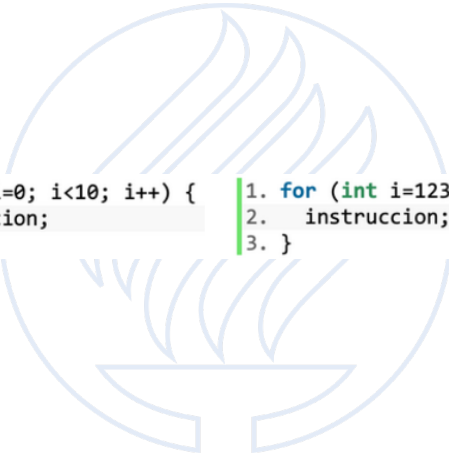


Ciclos

Cuando los ciclos tienen su rango de repetición en términos de **n**, entonces hay que revisar la variable de control y su comportamiento, de modo que el orden será:

- **Lineal**: cuando a la variable de control se le suma o resta una constante.
- **Logarítmico**: cuando a la variable de control se le divide o multiplica una constante.

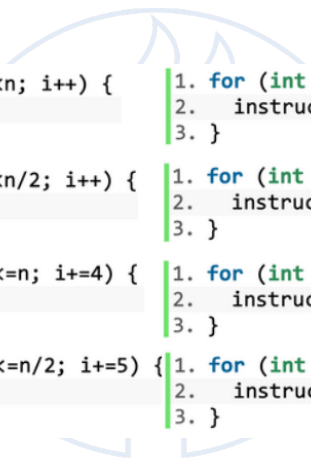
Ciclos $O(1)$



```
1. for (int i=0; i<10; i++) {  
2.   instruccion;  
3. }
```

```
1. for (int i=1234; i>0; i--) {  
2.   instruccion;  
3. }
```

Ciclos $O(n)$



```
1. for (int i=0; i<n; i++) {
2.   instruccion;
3. }
```

```
1. for (int i=n; i>0; i--) {
2.   instruccion;
3. }
```

```
1. for (int i=0; i<n/2; i++) {
2.   instruccion;
3. }
```

```
1. for (int i=n/5; i>0; i--) {
2.   instruccion;
3. }
```

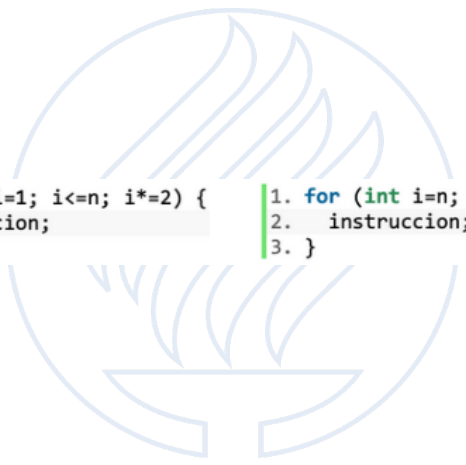
```
1. for (int i=1; i<=n; i+=4) {
2.   instruccion;
3. }
```

```
1. for (int i=n; i>0; i-=5) {
2.   instruccion;
3. }
```

```
1. for (int i=1; i<=n/2; i+=5) {
2.   instruccion;
3. }
```

```
1. for (int i=n/7; i>0; i-=3) {
2.   instruccion;
3. }
```

Ciclos $O(\log n)$



```
1. for (int i=1; i<=n; i*=2) {  
2.   instruccion;  
3. }
```

```
1. for (int i=n; i>0; i/=3) {  
2.   instruccion;  
3. }
```

Determina el Orden

¿Cuál es el orden de los siguientes bloques de código?


```
1.  if (n%2){
2.    for (int i=1; i<=n; i*=2){
3.      instruccion;
4.    }
5.  }
6.  else{
7.    for (int i=4; i<100; i++){
8.      instruccion;
9.    }
10. }
```

```
1.  if (n%2){
2.    for (int i=1; i<=n; i*=2){
3.      instruccion;
4.    }
5.  }
6.  else{
7.    for (int i=4; i<n; i++){
8.      instruccion;
9.    }
10. }
```

```
1.  for (int i=1; i<=n; i*=2){
2.    for (int j=4; j<n; j++){
3.      instruccion;
4.    }
5. }
```

```
1.  for (int i=1; i<=n; i+=2){
2.    for (int j=1; j<n; j++){
3.      instruccion;
4.    }
5. }
```

Determina el Orden



```

1.  if (n%=2){
2.    for (int i=1; i<=n; i*=2){
3.        instruccion;
4.    }
5.  }
6.  else{
7.    for (int i=4; i<100; i++){
8.        instruccion;
9.    }
10. }
```

$O(\log n)$

```

1.  if (n%=2){
2.    for (int i=1; i<=n; i*=2){
3.        instruccion;
4.    }
5.  }
6.  else{
7.    for (int i=4; i<n; i++){
8.        instruccion;
9.    }
10. }
```

$O(n)$

Copyrighted material

```

1.  for (int i=1; i<=n; i*=2){
2.    for (int j=4; j<n; j++){
3.        instruccion;
4.    }
5. }
```

$O(n * \log n)$

```

1.  for (int i=1; i<=n; i+=2){
2.    for (int j=1; j<n; j++){
3.        instruccion;
4.    }
5. }
```

$O(n^2)$

Copyrighted material

maxVal

Alg. 1: Devuelve el mayor elemento de un arreglo

```
int maxVal(int *A, int n){  
    int val = A[0];  
    for(i = 1; i < n; i++){  
        if(A[i] > val)  
            val = A[i];  
    }  
    return val;  
}
```

average

Alg. 2: Devuelve el promedio de los elementos de un arreglo

```
double average(int *A, int n){  
    int acum = 0;  
    for(i = 0; i < n; i++){  
        acum = acum + A[i];  
    }  
    return (acum / (double)n);  
}
```


pow2

Alg. 3: Calcula exponenciación mediante cuadrados

```
double pow2(double x, int n){
    double result = 0;
    while(n > 0){
        if(n % 2 == 1){
            result = result * x;
        }
        n = n / 2;
        x = x * x;
    }
    return result;
}
```

multMat

Alg. 4: Multiplicación de matrices cuadradas

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

Fibonacci

Alg. 5: Calcula el número n de la serie fibonacci

```
int fibonacci(int n){
    int previous, current, aux;
    previous = 1;
    current = 1;
    while (n > 2){
        aux = previous + current;
        previous = current;
        current = aux;
        n = n - 1;
    }
    return current;
}
```

1 Análisis de los algoritmos

2 Reglas para el cálculo de la complejidad

3 Análisis de algoritmos iterativos

4 Algoritmos recursivos

Orden $O(n)$

Orden $O(\log_b n)$

Orden $O(c^n)$

Orden $O(n^{\log_b c})$

Ejemplos de algoritmos recursivos

Fórmulas de recurrencia



Análisis de algoritmos recursivos

Bajo la notación de Big-O, los algoritmos recursivos requieren análisis de:

- La **cantidad de llamadas** recursivas en ejecución que se realizan.
- El comportamiento del **parámetro de control** de la función recursiva.



Orden $O(n)$

Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.

```
1. int sumaPares(int n){  
2.     if (n <= 0)  
3.         return 0;  
4.     else if (n%2 == 0)  
5.         return n+sumaPares(n-2);  
6.     else  
7.         return sumaPares(n-1);  
8. }
```

```
1. int fact(int n){  
2.     if (n <= 0)  
3.         return 1;  
4.     else  
5.         return n*fact(n-1)  
6. }
```

Orden $O(\log_b n)$

Cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor b constante.

```
1. int contPot2(int n){
2.     if (n <= 0)
3.         return 0;
4.     else
5.         return 1+contPot2(n/2);
6. }
```

$O(\log_2 n)$

```
1. int contPot3(int n){
2.     if (n <= 0)
3.         return 0;
4.     else
5.         return 1+contPot3(n/3);
6. }
```

$O(\log_3 n)$

Orden $O(c^n)$

Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.

```
1. int algo(int n){  
2.     if (n <= 0)  $O(4^n)$   
3.         return 400;  
4.     else  
5.         return algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);  
6. }
```

```
1. int algo(int n){  
2.     if (n <= 0)  $O(3^n)$   
3.         return 123;  
4.     else  
5.         return algo(n-4)+algo(n-4)+algo(n-4);  
6. }
```


Orden $O(n^{\log_b c})$

Cuando se tienen c llamadas recursivas en ejecución y su parámetro de control se divide o multiplica por un valor b constante.

```
1. int algo(int n){  
2.     if (n == 0)  $O(n^{\log_2 4}) = O(n^2)$   
3.         return 400;  
4.     else  
5.         return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);  
6. }
```

```
1. int algo(int n){  $O(n^{\log_4 3}) = O(n^{0.7924})$   
2.     if (n <= 0)  
3.         return 123;  
4.     else  
5.         return 1+algo(n/4)+algo(n/4)+algo(n/4);  
6. }
```

pow

Alg. 6: Calcula la potencia de x a la n

```
double pow(double x, int n){  
    if(n == 0){  
        return 1;  
    } else {  
        return x * pow(x, n - 1);  
    }  
}
```

Fibonacci

Alg. 7: Calcula el número n de la serie fibonacci

```
int fibonacci(int n){  
    if(n <= 0){  
        return 1;  
    } else {  
        return fibonacci(n - 1) + fibonacci(n - 2);  
    }  
}
```

pow2

Alg. 8: Calcula la exponenciación elevando al cuadrado recursivamente

```
double pow2(double x, int n){
    if(n < 0){
        return pow2(1 / x, -n);
    } else if(n == 0){
        return 1;
    } else if(n == 1){
        return x;
    } else if(n % 2 == 0){
        return pow2(x * x, n / 2);
    } else {
        return x * pow2(x * x, (n - 1) / 2);
    }
}
```

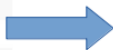
Fórmulas de recurrencia

Para comprobar la eficiencia de algoritmos recursivos, es necesario plantear el tiempo de ejecución en una fórmula de recurrencia en términos de $T(n)$:

- Igualándola por un lado a la cantidad de llamadas recursivas en términos de la modificación de su parámetro de control.
- Incluyendo también el tiempo requerido del caso base, en función de la cantidad de operaciones básicas que llevaría su proceso.

Ejemplo de recurrencia - Factorial

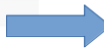
```
1. int factorial(int n){  
2.     if (n == 0)  
3.         return 1;  
4.     return 1*factorial(n-1);  
5. }
```



$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n-1), & n > 0 \end{cases}$$

Ejemplo de recurrencia - Factorial

```
1. int a(int n){  
2.   if (n == 0)  
3.     return 123;  
4.   return a(n-1)+a(n-1)+a(n-1);  
5. }
```



$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 3T(n-1), & n > 0 \end{cases}$$

