# Tema 5: Análisis de algoritmos

Parte 1

Introducción
Algoritmo
Eficiencia
Principio de invarianza
Instrucciones elementales
Utilización de otros recursos
Cálculo del número de operaciones que requiere un algoritmo:
    Función de operaciones elementales.
    Caso mejor, medio y peor.
    Expresión simplificada de la función de número de operaciones elementales
¿Por qué hay que buscar la eficiencia?

Notación asintótica "orden" ( $O$ )
    Regla del mínimo
    Regla del máximo (o de la suma)
    Regla del producto
Complejidad de un algoritmo
Cálculo del orden de complejidad de un algoritmo
    Secuencia de instrucciones
    Condicional simple (si-entonces-si_no):
    Condicional múltiple (casos)
    Bucle mientras
    Bucle repite hasta
    Bucle para
Caso mejor, medio y peor

Parte 2

Recurrencias

## Algoritmo:

El concepto más importante sobre el que trabajaremos en este tema es el de algoritmo. Un algoritmo es una secuencia de pasos para resolver un problema. Algunas características ampliamente aceptadas son:

1. Independiente: Del lenguaje y de la máquina.
2. Definido: Pasos claros y concretos.
3. Finito: Termina.
4. Preciso: Cada paso se puede llevar a cabo de forma correcta.
5. Entrada: Debe poseer datos de entrada.
6. Salida: Debe arrojar información de salida.

Lo que nos preocupa en el área del análisis de algoritmos es estudiar la bondad **de los algoritmos, no de las implementaciones concretas**, de modo que se emplearán técnicas para caracterizar el comportamiento de un algoritmo de forma independiente de sus detalles de implementación (en la medida de lo posible).

## Eficiencia:

La eficiencia es una medida de la buena utilización de los recursos disponibles. Dentro de esta definición, y en relación a la informática, podemos diferenciar en los algoritmos:

**Eficiencia en tiempo de ejecución**: un algoritmo es más eficiente mientras menor sea el tiempo que requiere para calcular la salida para una entrada.

**Eficiencia en uso de (otros, ya que el anterior se puede considerar como uso del procesador) recursos del sistema (memoria, archivos, reserva de periféricos, etc)**: un algoritmo es más eficiente mientras menos recursos necesite para calcular la salida correspondiente a una entrada. En este apartado suele tener mayor importancia el uso de la memoria.

## Principio de invarianza:

Dado un algoritmo y dos implementaciones suyas $I1$ e $I2$, que tardan $T1(n)$ y $T2(n)$ segundos respectivamente, existe una constante real $c > 0$ y un número natural $n0$ tales que para todo $n >= n0$ se verifica que $T1(n) >= cT2(n)$.

*"El tiempo de ejecución de dos implementaciones distintas de un algoritmo no va a diferir más que en una constante multiplicativa."*

**Ejemplo:** Comprobar si un elemento está en un array:

```
Implementación 1

indice = -1;
for (i=0;(i<tam)&&(A[i]!=e); i++  );
if (i==tam) indice = i;

Implementación 2

indice = -1;
salir = false;
i=0;
while (!salir){
      if ( A[i]==e ){
           indice = i;
           salir = true;
      }
      i++;
      if ( i==tam ) salir = true;
}
```

Según el principio de invarianza, aunque las dos implementaciones realicen un número de operaciones diferente, el tiempo total da las dos será diferente sólo en una constante de proporcionalidad. Esta constante se puede calcular toman tiempos para arrays de tamaños grandes y calculando la proporción de tiempos.

Recuerde que el algoritmo subyacente debe ser el mismo

**Instrucciones elementales:**

Para independizar el tiempo de la implementación concreta, se introduce el concepto de operación elemental:

**Una operación elemental será una operación que consume un tiempo unidad.**

En principio, puede haber consenso en que:

Los **accesos a memoria** (aparición de un nombre de variable) son operaciones elementales.

Las **asignaciones de valores a variables** ( *a=5* ) son operaciones elementales, ya que realizan un acceso a memoria.

Consideraremos que la aparición de valores constantes no consume tiempo de cómputo: *a=5* es una sola operación elemental y no dos.

```
indice = -1;      1 Operación: asignación
salir = false;    1 Operación: asignación
i=0;              1 Operación: asignación
A[i] = 10;        2 Operaciones: asignación + acceso (en la i)
j = k;            2 Operaciones: asignación + acceso (en la k)
k = A[i];         3 Operaciones: asignación + 2 accesos (A,i)
```

Las **operaciones aritméticas** básicas *(+,-,*,/,%,...)* son operaciones elementales.

Las **operaciones relacionales** *(<, >, /=, ==,...)* son operaciones elementales.

```
indice == -1;       2 Operación: operación relacional + acceso
salir != false;     2 Operación: operación relacional + acceso
i = j + 1;          3 Operaciones: asignación + acceso(j) + suma
A[i] = A[i] + k;    6 Operaciones: asignación + 4 accesos(2i,A,k) + op
A[i] += k;          6 Operaciones: equivale a la sentencia anterior
j = k;              2 Operaciones: asignación + acceso (en la k)
i++;                3 Operaciones: asignación + acceso (i) + op
```

Un **salto** es una operación elemental: veremos ejemplos en las estructuras.

Un **retorno** (return) se considerará como un salto: es una operación elemental.

Una **llamada** se considerará como un salto: es una operación elemental.

**Estructuras condicionales:**

Para las estructuras **si-entonces-si_no** (*if-then-else*), hay que considerar el coste de la comparación (que puede ser 1 o varias según haya o no accesos a memoria y demás) y un salto en el caso de que la condición no se verifique. Si la condición se verifica y hay caso *si_no*, entonces el caso verdadero también añade un salto incondicional al final de la estructura: las operaciones necesarias para la comparación en el caso **si** (se realiza la comparación) (+ 1 del salto si hay parte *si_no*), y eso + 1 (del salto) para el caso si_no (comparación y salto).

```
1) if ( x == 10 ){       2 ops: acceso + comparación (común)
2)    i = 7;             1 op: acceso (caso verdadero)
3) }                     1 op: salto al final (caso verdadero)
4) else{                 1 op: salto para el caso falso (caso falso)
5)    i++;               3 ops: asignación + acceso + op (caso falso)
6) }
```

Operaciones del caso verdadero = $T_{comp}+T_{cuerposi}+T_{salto} = 2+1+1 = 4$
Operaciones del caso falso = $T_{comp} + T_{salto}+T_{cuerpono} = 2+1+3 = 6$

```
1) if ( x == 10 ){       2 ops: acceso + comparación (común)
2)    i = 7;             1 op: acceso (caso verdadero)
3) }                     1 op: salto al final (caso falso)
```

Operaciones del caso verdadero = $T_{comp}+T_{cuerposi} = 2+1 = 3$
Operaciones del caso falso = $T_{comp} + T_{salto} = 2+1 = 3$

Las **estructuras de casos** (*case*) se puede calcular haciendo la comparación equivalente utilizando *si-entonces-si_no* encadenadas:

```
Ejercicio propuesto

1) switch (x){
2)    case 10: x++;
3)           break;
4)    case 20: i = x;
5)           break;
6)    default: x--;
7) }
```

**Estructuras repetitivas:**

**Mientras** (*while*): consideraremos que una evaluación de la condición y un salto fuera del bucle siempre se ejecutan. Además, para cada ciclo, se evalúa el cuerpo, se hace un salto al principio y se evalúa la condición. Sumaremos la aportación de cada ciclo del bucle. Para ello, utilizaremos una sumatoria con los índices adecuados.

```
1) i=0;                    1 op (inicialización)
2) while (i<n){            Comparación: 3 op: 2 accesos + op
3)          A[i] = 0;      2 op: asignación + acceso
4)          i++;           3 op: asignación + acceso + op
5) }                       1 op: salto al comienzo
6)                         1 op: salto condicional del final
```

Tiempo del cuerpo del bucle: $T_{cuerpo} = 2+3 = 5$

Tiempo del 1 ciclo: $T_{ciclo} = T_{cuerpo} + T_{salto} + T_{cond} = 5 + 1 + 3 = 9$

Tiempo total: $T_{ini} + T_{cond} + T_{salto} + \sum_{i=0}^{n-1} T_{ciclo} = 1+3+1+\sum_{i=0}^{n-1} 9 = 5+\sum_{i=0}^{n-1} 9$

**Repite hasta** (*repeat-until o do-while*): ahora consideraremos que siempre se realiza una evaluación del cuerpo del bucle y una evaluación de la condición. Por cada ciclo se ejecuta un salto, una evaluación del cuerpo del bucle y una evaluación de la condición.

```
1) i=0;                  1 op (inicialización)
2) do{
3)          A[i] = 0;    2 op: asignación + acceso
4)          i++;         3 op: asignación + acceso + op
5) }while (i<n)          cond: 3 op: salto al comienzo: 1 op
```

Tiempo del cuerpo del bucle: $T_{cuerpo} = 2+3 = 5$
Tiempo del 1 ciclo: $T_{ciclo} = T_{salto} + T_{cuerpo} + T_{cond} = 1 + 5 + 3 = 9$
Tiempo total: $T_{ini} + T_{cuerpo} + T_{cond} + \sum_{i=1}^{n-1} T_{ciclo} = 1+5+3+\sum_{i=1}^{n-1} 9 = 9+\sum_{i=1}^{n-1} 9$
Observe que lo que ha sucedido es que hemos incrementado el número de instrucciones fijo (independiente del número de ejecuciones del cuerpo del bucle) en una evaluación del cuerpo del bucle y hemos cambiado en recorrido del índice de modo que hace una repetición menos. Esto no es exactamente equivalente al caso anterior ya que añade siempre las instrucciones necesarias para una evaluación del cuerpo del bucle.

Bucle **para** (*for*): para el caso de C: `for( <inicialización>; <condición>; <incremento> ):`

El bucle para de C++ es equivalente a:

```
<inicialización>
while ( <condición> ){
      … // Cuerpo del bucle
      <incremento>
}
```

Con lo que el tiempo se calcula como en el caso del bucle mientras.

**Utilización de otros recursos:**

Las operaciones elementales que hemos definido anteriormente representan una unidad de utilización de tiempo. Si queremos medir la utilización de otros recursos, debemos definir operaciones elementales de utilización del recurso en cuestión.

```
Ejemplo: llenar una lista de elementos

1) N <- teclado;
2) Lista = new Lista;               +1 ¿?
3) Int i;                           +1 ¿?
4) For (i=0;i<n;i++)
5)    Añadir( Lista, teclado );     +1 ¿?
6) For (i=0;i<n;i++)
7)    Mostrar( Lista, i );
8) Eliminar Lista;                  - tamaño de Lista

   Observe la variabilidad de las operaciones según los tamaños de
los tipos que se estén utilizando. Es necesario además llevar la
cuenta de la memoria asignada a los diferentes elementos de nuestro
programa. Las áreas como compiladores se ocupan del estudio de la
utilización de la memoria
   Hay que considerar también que lo que nos interesa de un bloque
es obtener el máximo de memoria que puede llegar a utilizar, con lo
que habría que hacer un estudio según los casos y quedarse con las
subramas más costosas.
```

## Cálculo del número de operaciones que requiere un algoritmo:

### Función de operaciones elementales.

Es posible caracterizar un algoritmo con una función en base a determinados parámetros que, para un valor de sus parámetros que describa una entrada concreta para el algoritmo, nos devuelva el número de operaciones básicas que el algoritmo debe realizar.

### Casos mejor, medio y peor.

**Ejemplo: Ordenación por inserción**

La idea es considerar el array como una lista ordenada en la que vamos a insertar todos los elementos que actualmente contiene.

Insertaremos desde la primera posición hasta la última y consideraremos que la lista ordenada es el subarray que queda a la izquierda del índice por el que vamos insertando.

La implementación de la inserción consiste en ir desplazando el elemento "hacia dentro" de la lista, empezando por el final, hasta que ocupe su posición correcta, esto es: el elemento anterior es menor o igual y el siguiente es mayor. Lo que es lo mismo: desplazaremos el elemento hacia la izquierda mientras que el elemento anterior sea mayor o hayamos llegado al principio de la lista.

Nótese que el primer elemento ya está insertado.

```
1)  for ( i=1; i<n; i++ )      1,3,3
2)            for ( j=i;(j>0)&&(A[j-1]> A[j]);j-- ){    2,9,3
3)                    tmp = A[j];            3
4)                    A[j] = A[j-1];         5
5)                    A[j-1] = tmp;          4
6)            }
```

**En general:**

$$T_{cuerpoB2} = 12$$

$$\boldsymbol{T_{cicloB2}} = T_{cuerpoB2} + T_{incremento2} + T_{cond2} + T_{salto} = 12+3+9+1 = \mathbf{25}$$

$$\boldsymbol{T_{B2}} = T_{ini2} + T_{cond2} + T_{salto} + \sum_{j=i}^{?} T_{cicloB2} = 2+9+1+\sum_{j=i}^{?} 25 = \mathbf{12} + \sum_{j=i}^{?} \mathbf{25}$$

$$\boldsymbol{T_{cuerpoB1}} = \boldsymbol{T_{B2}}$$

$$\boldsymbol{T_{cicloB1}} = T_{cuerpoB1} + T_{incremento1} + T_{salto} + T_{cond1} = T_{B2} + 3+1+3 = 7+12+\sum_{j=i}^{?} 25 = \mathbf{19} + \sum_{j=i}^{?} \mathbf{25}$$

$$\boldsymbol{T_{B1}} = T_{ini1} + T_{cond1} + T_{salto} + \sum_{i=1}^{n-1} T_{cicloB1} = 1+3+1+\sum_{i=1}^{n-1} 19 + \sum_{j=i}^{?} 25 =$$

$$5+19(n-1)+\sum_{i=1}^{n-1}\sum_{j=i}^{?} 25 = \mathbf{19n - 14} + \sum_{i=1}^{n-1}\sum_{j=i}^{?} \mathbf{25}$$

El primer for se ejecuta siempre n-1 veces (para n=1..n-1)
El número de repeticiones del segundo bucle de penderá de los valores almacenados en el vector que se ordena

**Ejemplo: Ordenación por inserción (2)**

**Peor caso:** vector ordenado de mayor a menor.
El cuerpo del bucle más interno se ejecuta para j=i..1 (i veces):

$19n - 14 + \sum_{i=1}^{n-1} \sum_{j=i}^{1} 25 = 19n - 14 + \sum_{i=1}^{n-1} 25i = 19n - 14 + 25 \sum_{i=1}^{n-1} i = 19n - 14 + 25( ((1+(n-1))*(n-1))/2 ) = 19n - 14 + 25/2 ( n^2 - n ) = \mathbf{12.5n^2 + 6.5n - 14}$

**Mejor caso:** vector ordenado correctamente.
El cuerpo del bucle más interno se ejecuta 0 veces ya que la condición del bucle no se cumple nunca.

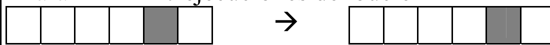
$19n - 14 + \sum_{i=1}^{n-1} 0 = \mathbf{19n - 14}$

**Ejemplo: Ordenación por inserción (3)**

**Caso medio:** debemos obtener el coste promedio de la ejecución del bucle interno, ya que el externo sabemos que se ejecuta con i=1..n-1 en todo caso. La idea es considerar todos los casos posibles de ejecución del segundo bucle y hacer la media del coste, es decir, sumar el coste de todos los casos y dividir por el número de casos.

Observamos que para un valor concreto del índice *i*, las posibilidades del bucle interno son desde repetirse 0 veces (el elemento insertado está ya correctamente insertado, con lo que no hay que moverlo hacia la izquierda) hasta repetirse *i* veces, en el caso de que el elemento deba ser desplazado hasta la primera posición del vector:

Mejor caso:
Para i = 4 → 0 ejecuciones del bucle 2



Peor caso:
Para i = 4 → 4 ejecuciones del bucle 2

Número de casos: 0,1,..,i = i+1

Promedio: suma de todos los casos dividida por el número de casos:

$$\boldsymbol{T_{cuerpoB2} = 12}$$
$$\boldsymbol{T_{cicloB2}} = T_{cuerpoB2} + T_{incremento2} + T_{cond2} + T_{salto} = 12+3+9+1 = \mathbf{25}$$

$$\boldsymbol{T_{B2}} = T_{ini2} + T_{cond2} + T_{salto} + \left( \frac{0 * \text{TcicloB2} + 1 * \text{TcicloB2} + 2 * \text{TcicloB2} + ... + \text{i} * \text{TcicloB2}}{i+1} \right) =$$

$$2+9+1+\sum_{j=0}^{i} \frac{jT_{cicloB2}}{i+1} = 12+\frac{T_{cicloB2}}{i+1}\sum_{j=0}^{i} j = 12+\frac{25}{i+1}\frac{(0+i)(i+1)}{2} = \mathbf{12+12.5\ i}$$

$$\boldsymbol{T_{cuerpoB1}(i) = T_{B2}(i)}$$
$$\boldsymbol{T_{cicloB1}(i)} = T_{cuerpoB1}(i)+T_{incremento1}+T_{salto}+T_{cond1} = T_{B2}(i)+3+1+3 = 7+12+12.5\ \text{i} = \mathbf{19+12.5\ i}$$

$$\boldsymbol{T_{B1}} = T_{ini1} + T_{cond1} + T_{salto} + \sum_{i=1}^{n-1} T_{cicloB1}(i) = 1+3+1+\sum_{i=1}^{n-1} 19+12.5\ i =$$

$$5+19(n-1)+\sum_{i=1}^{n-1} 12.5\ i = 19n - 14 + 12.5\sum_{i=1}^{n-1} i = 19n -4+12.5\frac{(1+(n-1))(n-1)}{2}$$
$$= 19n - 14 + 6.25\ (n^2 - n) = \mathbf{6.25n^2+12.75n - 14}$$

Para este cálculo se ha supuesto que todas las posibles ordenaciones del vector tienen la misma probabilidad de suceder ( 1/(i+1) ). En este caso podíamos haber supuesto directamente que el ciclo del bucle interno se iba a ejecutar la mitad de las veces y que el tiempo de ejecutarlo el cuerpo es constante, bastaría con utilizar como tiempo de ejecución la media entre el tiempo de ejecución del mejor caso y del peor caso = (0+25)/2

En caso de que los casos no tengan todos el mismo tiempo, habría que hacerlo del modo descrito anteriormente.

De igual modo, en caso de que los casos no tengan la misma probabilidad, habría que hacer la media ponderada: la suma de los costes de los casos multiplicados por la probabilidad que tienen de ocurrir:

Coste de los casos: $c_1, c_2, ..., c_n$
Probabilidad de cada caso: $p_1, p_2, ..., p_n$

Coste promedio: $p_1c_1 + p_2c_2 + p_3c_3 = \sum_{i=1}^{n} p_ic_i$

El flujo tomado dependerá de los datos de entrada. Normalmente se considerarán tres casos particulares en relación a la naturaleza de los datos: mejor caso, peor caso y caso medio.

El mejor caso será aquél en que los datos producen el flujo del algoritmo más favorable, básicamente **se tomarán las ramas condicionales menos costosas y los casos de los bucles más ventajosos**.

El peor caso consiste en suponer lo peor, es decir, **tomar las ramas condicionales más costosas.**

**El caso medio consiste en suponer una naturaleza de los datos que produce un flujo promedio entre el mejor y el peor caso**. Se debe tener en cuenta que **el caso medio no se corresponde con la media** entre el mejor y peor caso.

**Ejemplo**

```
5) for ( i=0; i<n; i++ )       1,3,3
6)    if ( A[i] % 4 == 0 )           4
7)          for ( j=0; j<n; j++ )    1,3,3
8)                B[j]++;      5
```

**En general:**

$p$ = probabilidad de que la condición sea verdadera
$1\text{-}p$ = probabilidad de que la condición sea falsa

$$T_{cicloB2} = T_{cuerpoB2} + T_{incrementoB2} + T_{salto} + T_{condB2} = \mathbf{12}$$

$$T_{B2} = T_{ini2} + T_{condB2} + T_{salto} + \sum_{j=0}^{n-1} T_{cicloB2} = 5 + \sum_{j=0}^{n-1} 12 = \mathbf{12n+5}$$

$$T_{ifSI} = T_{condif} + T_{B2}$$

$$T_{ifNO} = T_{condif} + T_{salto}$$

$$T_{if} = pT_{ifSI} + (1\text{-}p)T_{ifNO} = T_{condif} + pT_{B2} + (1\text{-}p)T_{salto} =$$

$$= 4 + p(12n+5) + (1\text{-}p) = 5 + 12pn + 4p = \mathbf{5 + (12n + 4)\,p}$$

$$T_{cicloB1} = T_{if} + T_{incrementoB1} + T_{salto} + T_{condB1} = 4 + 5 + (12n + 4)\,p$$

$$= \mathbf{9 + (12n + 4)\,p}$$

$$T_{B1} = T_{ini1} + T_{condB1} + T_{salto} + \sum_{j=0}^{n-1} T_{cicloB1} = 5 + \sum_{j=0}^{n-1} (9 + (12n + 4)\,p) =$$

$$5 + (9 + (12n + 4)\,p)n = \mathbf{5 + 9n + p(12n^2 + 4n)}$$

**Peor caso:** Condición siempre verdadera: $p = 1$

$5 + 9n + 12n^2 + 4n = \mathbf{12n^2 + 11n + 5}$

**Mejor caso:** Condición siempre falsa: $p=0$ :$\mathbf{5+9n}$

**Caso medio:** Si consideramos que el vector puede tener cualquier valor entero, la probabilidad de que la condición sea verdadera (ser divisible por 4) es una de cada 4, es decir: ¾ = 0.75

$5+9n+ ¾(12n^2 + 4n) = \mathbf{9n^2 + 12n + 5}$

Es posible que un algoritmo realice las mismas operaciones en todos los casos (por la ausencia de ramas condicionales o por su equivalencia en operaciones).

**Ejemplo**

```
1) for ( i=0; i<n; i++ )       1,3,3 (1 salto)
2)    A[i]++                   5 = 3 acesos,1 asig, 1 op
```

**Peor caso = Mejor caso = Caso medio**

$$\boldsymbol{T_{ciclo}} = T_{cuerpo} + T_{incremento} + T_{salto} + T_{cond} = 5 + 3 + 1 + 3 = \mathbf{12}$$

$$\boldsymbol{T} = T_{ini} + T_{cond} + T_{salto} + \sum_{i=0}^{n-1} T_{ciclo} = 1 + 3 + 1 + \sum_{i=0}^{n-1} 12 = \mathbf{12n + 5}$$

## Expresión simplificada de la función de número de operaciones elementales:

**Sumatorias**

1) La sumatoria de una suma es igual a la suma de las sumatorias:

$$\sum a + b = \sum a + \sum b$$

2) Cuando el cuerpo de la sumatoria es independiente de los índices, el valor es el número de valores diferentes que toma el índice multiplicado por el valor del cuerpo:

$$\sum_{i=0}^{n-1} a = a.n$$

$$\sum_{i=0}^{n-1} a + f(i) = \sum_{i=0}^{n-1} a + \sum_{i=0}^{n-1} f(i) = a.n + \sum_{i=0}^{n-1} f(i)$$

3) Cuando el cuerpo de la sumatoria se puede expresar como una constante independiente de los índices multiplicada por una expresión, el valor es el valor de la constante multiplicada por la sumatoria de la expresión:

$$\sum_{i=0}^{n-1} af(i) = a \sum_{i=0}^{n-1} f(i)$$

4) Suma de los valores de una progresión aritmética: Ej:(1+2+3+4+5),(4+6+8+10) ,(2+5+8+11+14)
$\Delta$ = incremento
$a_0$ = primer elemento
$a_{n-1}$ = último elemento
n = número de elementos
progresión aritmética: $a_n = a_0 + \Delta_n$

$$\sum_{i=0}^{n-1} a_i = ((a_i + a_{n-1})n)/2$$

*"El primer elemento más el último, multiplicado por el número de elementos y dividido por 2"*

5) Suma de los valores de una progresión geométrica: Ej:(1+2+4+8+16), (2+6+18+54)
$\Pi$ = razón
$a_0$ = primer elemento
n = número de elementos
progresión geométrica: $a_n = a_0 \, \Pi^n$
$(r-1)(1 + r + ... + r^{n-1}) = r^n - 1$

$$\sum_{i=0}^{n-1} a_i = \sum_{i=0}^{n-1} a_0 \Pi^i = a_0 \sum_{i=0}^{n-1} \Pi^i = a_0(\Pi^n-1)/(\Pi-1)$$

*"El primer elemento multiplicado por la razón elevada al número de elementos menos 1 y todo dividido por la razón menos 1"*

## ¿Por qué hay que buscar la eficiencia?

**Ejemplo: sucesión de Fibonacci**

La sucesión de Fibonacci es una conocida serie de números con determinadas propiedades curiosas y presente en múltiples fenómenos de la naturaleza.

Su definición es:

```
f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2)  para n natural y >1
```

Es decir, el primer número es 0, el segundo es 1 y los siguientes se calculan como la suma de los dos anteriores: 0 1 1 2 3 5 8 13 21 …

**Implementación 1: recursiva**

```
fibrec(n)               h(n) (tiempo para n)
1) if ( n==0 )          2,1(salto para el falso)
2)     return 0;        1
3) if ( n==1 )          2,1(salto para el falso)
4)     return 1;        1
5) return fibrec( n-1 ) + fibrec( n-2 )       8 + h(n-1)+h(n-2)
```

$$h(n)\begin{cases} 3 & n=0 \\ 6 & n=1 \\ 14+h(n-1)+h(n-2) & n>1 \end{cases}$$

$$h(n)=c_1+c_2\left(\frac{1+\sqrt{5}}{2}\right)^n+c_3\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$O(h(n))=O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)=O(1.62^n)$$

**Implementación 2: iterativa**

```
fibiter(n)
1) ant = 1;          1
2) fib = 0;          1
3) for ( i=0; i<n; i++ ){ 1,3,3
4)     fib = fib + ant;          4
5)     Ant = fib - ant;          4
6) }                             1 (salto)
7) return fib;                   2 (salo + acceso)
```

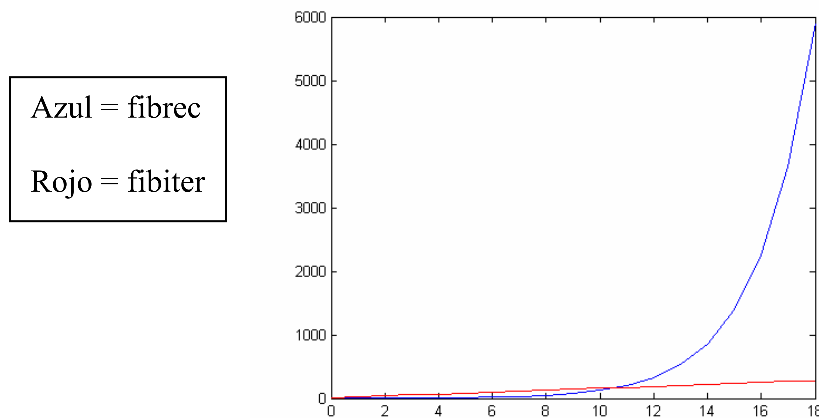
h(n) = $1+1+(1+3+1+\sum_{i=0}^{n-1} 4+4+3+1+3)+2=$ **15n + 9**

**Tiempos:** supondremos que 1 operación es un microsegundo ($10^{-6}$seg)

| | N= 3 | N = 10 | N = 20 | N=30 | N=40 | N=50 | N=60 |
|---|---|---|---|---|---|---|---|
| Fibrec | 4.25 | 124 | 15498 | 1929525 | $240e^6$=4min | $29.9e^9$=8.3h | $3.7e^{12}$=1.4 meses |
| Fibiter | 54 | 159 | 309 | 459 | 609 | 759 | 909 = 1 seg |

**Notación asintótica "orden" ( $O$ ):**

Llegados a este punto, somos capaces de caracterizar los algoritmos con una función de sus operaciones elementales. Con esta información ya nos es posible comparar la bondad de dos algoritmos. Deberíamos hacer un estudio de las funciones y ver cómo evoluciona cada uno de los algoritmos.




Sin embargo, es posible generalizar un poco más y realizar un estudio y clasificación de los algoritmos en base a sus comportamientos en "el límite", esto es, para entradas suficientemente grandes (o valores de los parámetros que se correspondan con las mayores cantidades de cálculo).

Introduciremos la notación asintótica orden ( $O$ ):

Dada

$f: N \rightarrow R+$

$$O(f(n)) = \left\{ t : N \rightarrow R^{+} \middle| (\exists c \in R^{+}, n_0 \in N)(\forall n \geq n_0 \in N)\left[ t(n) \leq cf(n) \right] \right\}$$

*"El orden de* f(n) *es el conjunto de todas las funciones* t(n) *para las que existe una constante real* ***positiva y distinta de 0*** c *y un número natural* $n_0$ *tal que si multiplicamos* f(n) *por* c*,* t(n) *será mayor o igual que* cf(n) *para todos los valores de* n *mayores o iguales que* $n_0$*"*

Lo que es lo mismo: *O(f(n))* es el conjunto de cotas inferiores de alguna función proporcional a *f(n)* para *n* suficientemente grandes.

Esto quiere decir que *O(f(n))* es el conjunto de todas las funciones que crecen, como máximo, tan rápido como *f(n)* (a lo sumo salvo por una constante multiplicativa) para valores de n suficientemente grandes.

En general, cuando nos den una función *g(n)*, asociaremos esa función con *O(f(n))* tal que *f(n)* sea la función más restrictiva (mínima) que cumple que $g(n) \in O(f(n))$:

**Ejemplo:**

$g(n) = 2n^3 + 3n^2 + 5$
$g(n) \in O(2^n)$
$g(n) \in O(n^3)$ → preferiremos esta porque nos da una cota superior más restrictiva. Estamos buscando los comportamientos peores, pero dentro de lo peor, lo mejor posible…

Para comprobar, lo que tenemos que hacer es buscar una constante real *c* y un $n_0$ que verifiquen la condición impuesta en la definición del orden. ( $g(n) < cf(n)$ para $n>=n0$ )

**Propiedades importantes:**

**Constantes multiplicativas:**

Por la definición de *O*, *O(cf()n)* = *O(f(n))* por lo que eliminaremos las constantes multiplicativas.

**Regla del mínimo:**

$f \in O(g)$ y $f \in O(h) \rightarrow f \in O(\min(g,h))$

**Regla del máximo (o de la suma):**

$f1 \in O(g), f2 \in O(h) \rightarrow f1 + f2 \in O(\max(g, h))$

*“Si tenemos una función que es suma de dos funciones, la cota superior de la función es del mismo orden que la función que crece más rápido”. Esto se debe a que O se ocupa de los comportamientos en el límite. Para valores de n altos, el crecimiento que aporta la función menor al crecimiento de la función suma es despreciable con respecto al crecimiento que aporta la función mayor.*

**Regla del producto:**

$f1 \in O(g), f2 \in O(h) \rightarrow f1*f2 = O(g*h)$

*“Si tenemos una función que es multiplicación de dos, el orden es la multiplicación de los órdenes de las funciones multiplicadas”*

Utilizando las propiedades anteriores, es fácil calcular el orden *O* de un algoritmo sin necesidad de hallar su función característica.

## Complejidad de un algoritmo

Llamaremos **complejidad** de un algoritmo a **su orden *O* referido al peor caso**.

## Cálculo del orden de complejidad de un algoritmo

### Secuencia de instrucciones:

Si tenemos una secuencia de instrucciones $I1, I2, .., Im$ que toman tiempos $T1(n)$, $T2(n)$, .. $Tm(n)$, el tiempo de ejecución total será: $T1(n) + T2(n) + \ldots + Tm(n)$.

Si $T1 \in O(f1(n))$, $T2 \in O(f2(n))$,.. $Tm \in O(fm(n))$, entonces, $f1(n)+f2(n)+..+fm(n)$ es cota superior del tiempo de ejecución $T1(n) + T2(n) + \ldots + Tm(n)$, con lo que nos sirve $O(f1(n)+f2(n)+..+fm(n))$ como orden de ejecución de $I1, I2, .., Im$ y, por la regla de la suma:

$$O(\ f1(n) + f2(n) + \ldots + fm(n)\ ) = O(\ max(f1(n), f2(n), \ldots, fm(n))\ )$$

```
Ejemplo
Secuencia
1) a = 5;                2 op: O(2) = O(1)
2) B[2]++;               3 op: asig + acceso + oper: O(3) = O(1)
```

O( Secuencia ) = O( max( 1,1 ) ) = O(1)

### Condicional simple (*si-entonces-si_no*):

Considerando que con la notación *O()* obtenemos una cota superior del tiempo de ejecución, bastará con que utilicemos para su cálculo una función que acote superiormente el tiempo de cálculo del bloque que estamos analizando. Con esto, sabemos que un bloque *si-entonces-si_no* está acotado superiormente por (tarda como máximo) el tiempo requerido para evaluar la parte más costosa del bloque:

Si $Tsi(n) \in O(\ fsi(n)\ )$ es el tiempo requerido por la parte si de la estructura *si-entonces-si_no* y $Tno(n) \in O(\ fno(n)\ )$ es el tiempo requerido por la parte no, diremos que el orden de la estructura será:

$$O(\ fest(n)\ ) = O(\ max(\ fsi(n), fno(n)\ )\ )$$

**NOTA:** ¡Observe que estamos considerando el peor caso!

**Ejemplo**

```
Condicional
1) if ( A[5] == 0 )       2 op: acceso + comp (común)= O(2) = O(1)
2)     B[2]++;            3 op: (caso verdadero) O(3) = O(1)
3)                        1 op: (caso falso) O(1)
```

O(verdadero) = O( max(1,1) ) = O(1)
O(falso) = O( max(1,1) ) = O(1)

O( Condicional ) = O( max( verdadero,falso ) ) = O( max(1,1) ) = O(1)

**Condicional múltiple (casos)**: al igual que en el caso anterior, el orden será el de el bloque más costoso.

**Bucle mientras (*while*):** Conociendo el número de veces que se repetirá el cuerpo del bucle ( normalmente en función de un parámetro) y sabiendo que el tiempo del salto incondicional y del salto final es *O(1)* y el de la comparación suele ser *O(1)* (habría que ver el caso concreto), el tiempo total sería (utilizando las propiedades de *O*):

*Tcomp(n) + rep(m)*( Tcuerpo(n) + Tsalto_inc + Tcomp(n) ) + Tsalto_cond*

está acotado por

*fcomp(n) + rep(m)*( fcuerpo(n) + fsalto_inc + fcomp(n) ) + fsalto_cond =*

*( Tcomp, Tsalto_inc, Tsalto_cond e O(1) )*

*1 + rep(m)*( fcuerpo(n) + 1 + 1 ) + 1 =*

*2 + rep(m)*fcuerpo(n) + 2rep(m)* = equivale asintóticamente a:

*1 + rep(m)*fcuerpo(n) + rep(m)*

*O( 1 + rep(m)*fcuerpo(n) + rep(m) )* = regla de la suma =

*O( max( 1, rep(m)*fcuerpo(n), rep(m)) ) =*

*O( rep(m)*fcuerpo(n) )*

Es decir, el orden del número de veces que se ejecuta el cuerpo del bucle multiplicado por el orden de la ejecución del cuerpo.

**Ejemplo**

```
Mientras
1) while ( i<n ){         3 op: 2acceso + comp = O(3) = O(1)
2)     B[i]++;            5 op: O(5) = O(1)
3)     i++;               2 op: O(3) = O(1)
4) }                      1 op: salto O(1)
```

O(cuerpo) = O( max(1,1,1,1) ) = O(1)
rep(n) = n

O( Mientras ) = O( rep(n) * cuerpo ) = O( n*cuerpo ) = O( n*1 ) = O(n)

**Bucle repite (repeat):** con un desarrollo similar al anterior, se llega a la misma conclusión: el orden del número de veces que se ejecuta el cuerpo del bucle multiplicado por el orden de la ejecución del cuerpo.

**Bucle para (for):** con un desarrollo similar al anterior, se llega a la misma conclusión: el orden del número de veces que se ejecuta el cuerpo del bucle multiplicado por el orden de la ejecución del cuerpo.

**Caso mejor, medio y peor.**

El hecho de que la notación asintótica de orden *O()* exprese una cota asintótica inferior al tiempo de ejecución no quiere decir que esté forzosamente relacionada con el peor caso de ejecución del algoritmo. Podemos calcular el orden del algoritmo para cualquiera de los tres casos vistos con anterioridad: casos mejor, medio y peor.

```
Ejemplo: Ordenación por inserción
      La idea es considerar el array como una lista ordenada en la
que vamos a insertar todos los elementos que actualmente contiene.
      Insertaremos desde la primera posición hasta la última y
consideraremos que la lista ordenada es el subarray que queda a la
izquierda del índice por el que vamos insertando.
      La implementación de la inserción consiste en ir desplazando
el elemento "hacia dentro" de la lista, empezando por el final,
hasta que ocupe su posición correcta, esto es: el elemento anterior
es menor o igual y el siguiente es mayor. Lo que es lo mismo:
desplazaremos el elemento hacia la izquierda mientras que el
elemento anterior sea mayor o hayamos llegado al principio de la
lista.
      Nótese que el primer elemento ya está insertado.

1) for ( i=1; i<n; i++ )      O(n) (O(n) repeticiones)
2)    for ( j=i;(j>0)&&(A[j-1]> A[j]);j-- ){   O(?)
3)          tmp = A[j];       O(1)
4)          A[j] = A[j-1];    O(1)
5)          A[j-1] = tmp;     O(1)
6)    }
```

**En general:** el primer bucle siempre se recorre en todo su rango

B1 = O( n B2 )

**Peor caso:** vector ordenado de mayor a menor

B2 = O(n)
B1 = O(n) * O(n) = O($n^2$) : orden cuadrático

**Mejor caso:** vector ordenado correctamente

B2 = O(1) (una comparación)
B1 = O(n) * O(1) = O(n) : orden lineal

**Caso medio:** podemos considerar que el bucle interno se ejecuta la mitad de los pasos que le corresponden en cada repetición. El bucle principal *siempre* se ejecuta en todo su rango.

B2 = O(n/2) = O(n)
B1 = O(n) O(n) = O($n^2$) : orden cuadrático