Índice

Excepciones

Una excepción es un error que puede ocurrir durante la ejecucion del programa debido a una mala entrada por parte del usuario, un argumento inválido para un cálculo matemático, una apertura de un archivo inexistente, etc.

Para remediar esto, idealmente hay que escribir los algoritmos y las validaciones necesarias para evitar a toda costa que un error de excepción pueda hacer que el programa se interrumpa de manera inesperada.

Sin embargo, en C++ hay un mecanismo para el manejo de excepciones que consta de **try**, **throw** y **catch**.

Es importante mencionar que las excepciones están declaradas en clases (representan objetos) y heredan entre sí. Por ejemplo `bad_array_new_lenght` hereda de `bad_alloc` que a su vez hereda de `exception`.

La estructura general de un bloque `try ... catch` es:

```
 try {
   // Bloque de código a comprobar
 }
 catch (tipo) {     // Paso algo en el try que termino la ejecución del bloque,
catch                       // agarra y analiza lo que paso
   // Bloque de código que analiza lo que tiró el try
 }
```

1. Dentro de un bloque **try** se pretende evaluar una o más expresiones y si dentro de dicho bloque se produce un **algo que no se espera** se lanza por medio de **throw** una excepción, la misma que deberá ser capturada por un **catch** específico.
2. Puesto que desde un bloque **try** pueden ser lanzados diferentes tipos de errores de excepción es que puede haber más de un **catch** para capturar a cada uno de los mismos.
3. Si desde un **try** se lanza una excepción y no existe el mecanismo **catch** para tratar dicha excepción el programa se **interumpirá** abruptamente despues de haber pasado por todos los **catchs** que se hayan definido y de no haber encontrado el adecuado.
4. Los tipos de excepciones lazados pueden ser de un tipo primitivo tal como: **int**, **float**, **char**, etc. aunque normalmente las exepciones son lanzadas por alguna clase escrita por el usuario o por una clase de las que vienen incluidas con el compilador.

Atrapando excepciones (catch)

Excepción con parámetro formal

```
 try {
    // Bloque de código a comprobar
 }
 catch (const exception& e) {
    // Bloque de código, por ejemplo
 }
```

Excepción con parámetro sin nombre

```
 try {
   // Bloque de código a comprobar
 }
 catch (const exception&) {
   // Bloque de código
 }
```

Excepción generica

Como los errores pueden ocurrir frente a un monton de situaciones, hay una forma de manejar excepciones genéricas o desconocidas

```
// Opcion 1
try {
   // Bloque de código a comprobar
}
catch (...) {
    cout << "Error, se produjo un error inesperado\n";
}
```

```
// Opcion 2
try {
    // Bloque de código
}
catch(exception &e) {
    cout << "Error, se produjo un fallo en la instruccion "\" << e.what() <<
"\"\n";
}
```

Lanzando excepciones (throw)

Para lanzar una excepción es necesario crear un objeto del tipo de excepción que se quiera lanzar.

Cuando se lanza una excepción se corta la ejecución del método y se busca si se produjo en el contexto de un `try ... catch` que capture ese tipo. En el caso de que no sea capturada la excepción, se termina la ejecución con un mensaje de error.

Veamos el siguiente ejemplo:

```
int comparar(int a, int b);

int main() {
    try {
        comparar(-2, 5);
    }
    catch(const invalid_argument &e) {
        cout << "Se produjo un error en los argumentos: " << e.what();
    }
    return 0;
}

int comparar(int a, int b) {
    if (a < 0 || b < 0) {
        throw invalid_argument("Valores negativos recibidos");
    }
    // Resto del código
}
```

En este caso, la salida será:

```
"Se produjo un error en los argumentos: Valores negativos recibidos"
```

Lo que podríamos hacer en vez de simplemente hacer un cout, es intentar procesar esos datos con otra función por ejemplo:

```
int compararNaturales(int a, int b);
int compararEnteros(int a, int b);

int main() {
    cout<< "\nComienza main\n";
    int a, b;
    cout << "a: "; cin >> a;
    cout << "b: "; cin >> b;
```

```
    try {
        cout << "Intentando comparar "<< a << " y " << b << " con
compararNaturales()...\n";
        compararNaturales(a, b);
    }
    catch(const invalid_argument& e) {
        cout <<"Se atrapó la excepción en el main.\n"
               "El error es: " << e.what() << "\n";
        cout << "Reintentando con compararEnteros()...\n";
        compararEnteros(a, b);
    }
    cout << "\nFinaliza main\n";
    return 0;
}

int compararNaturales(int a, int b) {
    try {
        if (a < 0 || b < 0) {
            throw invalid_argument("Valores negativos recibidos");
        }
        // ...
        cout << "Comparación de enteros positivos realizada\n";
    }
    catch (const invalid_argument& e) {
        cout << "Se atrapó la excepción en compararNaturales(). Lanzando...\n";
        throw;
    }
}

int compararEnteros(int a, int b) {
    // ...
    cout << "Comparación realizada con éxito\n";
}
```

La salida si el usuario ingresa *a = -1* y *b = 3*:

```
Comienza main

Intentando comparar -1 y 3 con compararNaturales()...
Se atrapó la excepción en compararNaturales(). Lanzando...
Se atrapó la excepción en el main.
El error es: Valores negativos recibidos
Reintentando con compararEnteros()
Comparación de enteros realizada

Finaliza main
```

O también reintentar el ingreso:

```
#include <iostream>
using namespace std;

int comparar(int a, int b);

int main() {
    int a, b;
```

```
    try {
        cout << "a: "; cin >> a;
        cout << "b: "; cin >> b;
        comparar(a, b);
    }
    catch(const invalid_argument& e) {
        cout << e.what() << "\nReintentando...\n";
        cout << "a: "; cin >> a;
        cout << "b: "; cin >> b;
        try {
            comparar(a, b);
            cout << "a: " << a; cout << "\n";
            cout << "b: "<< b; cout << "\n";
        }
        catch (const invalid_argument& e) {
            cout << e.what() << "\nSaliendo del programa\n";
        }
    }
    return 0;
}

int comparar(int a, int b) {
    if (a < 0 || b < 0) {
        throw invalid_argument("Valores negativos recibidos");
    }
    // ...
    cout << "Comparación realizada con éxito\n";
}
```

En este caso la salida podría ser:

```
////////////////////////////// CASO 1 ////////////////////////////
// Si inicialmente el usuario ingresa a = 4 y b = -1
Se produjo un error en los argumentos. Reintentando...
// Ahora el usuario ingresa a = 1 y b = 2
Operación realizada con éxito

////////////////////////////// CASO 2 ////////////////////////////
// Si inicialmente el usuario ingresa a = 6 y b = -3
Se produjo un error en los argumentos. Reintentando...
// Ahora el usuario ingresa a = -1 y b = 2
Se produjo un error en los argumentos. Saliendo del programa
```

Creación de excepciones

Podemos crear una clase y sobreescribir los métodos virtuales para tener nuestra propia excepción . Combinandolo con lo anterior podríamos tener:

```
#include <iostream>
using namespace std;

int comparar(int a, int b);

class miExcepcion: public exception {
    public:
        virtual const char* what() const throw() {
```

```
            return "Error miExcepcion";
        }
} miEx;

int main() {
    try {
        throw miEx;
    }
    catch(exception& e) {
        cout << e.what() << "\n";

        try {
            comparar(-2, 5);
        }

        catch(const invalid_argument &e) {

            cout << "Se produjo un error en los argumentos. Reintentando...\n";

            try {
                comparar(-2, 4);
            }

            catch(invalid_argument &e) {
                cout << "Se produjo un error en los argumentos. Saliendo del
programa.";
            }
        }
    }

    return 0;
}

int comparar(int a, int b) {
    if (a < 0 || b < 0) {
        throw invalid_argument("Valores negativos recibidos");
    }
}
```

Y la salida será

```
"Error miExcepcion"
"Se produjo un error en los argumentos. Reintentando..."
"Se produjo un error en los argumentos. Saliendo del programa."
```

Complejidad temporal

Un algoritmo es más eficiente que otro si utiliza menos recursos, la eficiencia se determina sobre la base del consumo de tiempo de ejecución y espacio de memoria utilizados por el algoritmo

La complejidad temporal es el tiempo necesario para ejecutar un algoritmo. Hay dos enfoques, el empírico y el teórico.

- Empírico: se mide el tiempo real de ejecución para determinados valores de entrada y en determinado procesador.
- Teórico: se determina una función que indique el tiempo de ejecución para determinados valores de entrada.

En general no se utiliza el enfoque empírico, porque depende de la computadora y del lenguaje, y no sirve para hacer comparaciones con distintos algoritmos o para hacer predicciones.

Hay tres conceptos que es importante distinguir antes de analizar la complejidad temporal:

1. Problema: necesidad inicial de la cual se busca alcanzar la solución mediante un algoritmo.
2. Ejemplar: individuo de una especie o género.
3. Algoritmo: serie de pasos ordenados y finitos cuyo objetivo es hallar la solución a un problema.

Ejemplo:

1. El problema consiste en ordenar un vector de n elementos.
2. Los ejemplares son los distintos vectores para cada uno de los diferentes valores de n.
3. Los algoritmos pueden ser los métodos de selección o burbujeo, etcétera.

Tamaño del problema

Para analizar la complejidad temporal dejamos de tener en cuenta de cusetiones como la velocidad de procesamiento, el numero de microprocesadores, el lenguaje utilizado, las características del compilador, etc.

El coste temporal, depende entre otras cosas del tamaño del problena. Este tamaño depende de la naturaleza del problema y corresponde a aquel o aquellos elementos que produzcan, al crecer, un aumento de tiempo de ejecución.

Ejemplos:

- En el cálculo del factorial, el tamaño del problema será el valor sobre el cual se quiera realizar el cálculo. Cuanto mayor sea el número, mayor será el tiempo consumido por la ejecución del algoritmo.
- En el caso de una búsqueda binaria, el tamaño del problema será el número de elementos que tenga el vector donde se realizaŕa la búsqueda.
- Al sumar dos matrices, el tamaño quedará determinado por el producto entre la cantidad de filas y columnas de las matrices.

Principio de invarianza

A partir de ahora, el tiempo que tarda en ejecutarse un algoritmo lo indicaremos con la letra *T* sin unidades especificas, pueden ser segundos, minutos, horas, etc.

Dado un algoritmo y dos implementaciones del mismo (I1 e I2) que tardarán un tiempo T1(n) y T2(n) respectivamente, entonces existe una constante real c > 0 y un n 0 ε N tales que ∀ n ≥ n 0 se verifica que:

$$T1(n) \leq c \cdot T2(n)$$

Este principio indica que dos ejecuciones distintas del mismo algoritmo solo difieren, para nuestro cálculo en cuanto a eficiencia, en un factor constante teniendo en cuenta valores de entrada suficientemente grandes.

Ejemplo: Se elige el algoritmo de ordenamiento mediante el método de selección de un vector de tamaño n. Se implementa en C++ (lenguaje compilado) y en Python (lenguaje interpretado) y en dos máquinas distintas (m1 y m2, esta última es más vieja y con menor capacidad de procesamiento). Luego tendríamos cuatro implementaciones diferentes:

- En C++ en m1
- En C++ en m2
- En Python en m1
- En Python en m2

Tomamos los tiempos en segundos, mostramos los resultados en la tabla:

-	n	Tm1	Tm2
C++	10 mil	0.467	0.87
C++	1000 mil	40.34	75.15
Python	10 mil	16.24	31.2
Python	100 mil	1211	2325

Analizando sólo la máquina uno, podemos observar que entre C++ y Python hay una proporción de 34 para *n = 10.000* y de 30 para *n = 100.000*, por lo tanto podemos suponer que el valor de *c* sería aproximadamente 30 para una misma máquina pero diferente lenguaje.

Analizando ambas máquinas pero sin comparar los lenguajes, podemos ver que los tiempos casi se duplican, por lo que la proporción entre ambas máquinas para el mismo lenguaje es de 2.

Entre ambas combinaciones, la proporción sería aproximadamente 60.

Running time

Se denomina running time al número de pasos requeridos por un algoritmo apra resolver un problema específico, y se analiza en función del tamaño de problema.

Diremos que T es función de n, es decir T = T(n). El valor de esta función es proporcional al tiempo de ejecución de un programa con una entrada de tamaño n

Casos a considerar

Para un mismo algoritmo, y un mismo tamaño de problema, hay tres situaciones referidas al consumo temporal:

- Mejor caso: corresponde a la secuencia de sentencias del algoritmo en la cual se ejecute una secuencia de instrucciones que corresponda al tiempo mínimo.
- Peor caso: es el que corresponde a la secuencia de sentencias en la cual se ejecute una secuencia de instrucciones que corresponda al tiempo máximo.
- Caso promedio: corresponde al promedio entre todas las trazas posibles ponderadas según su probabilidad. Este último caso es, en general el que presenta mayores dificultades para el análisis.

Conteo

En el cálculo de la complejidad temporal, hay operaciones a las cuales se les asigna una duración de una unidad de tiempo. Estas operaciones las llamamos operaciones elementales (OE).

Nota: si bien un tipo de operación elemental no tiene por qué tener la misma duración que otrapodemos acotar ese tiempo por una constante.

Tomaremos como OE a:

- Asignación de variables
- Operaciones aritméticas básicas: suma, resta, producto, división.
- Comparaciones lógicas.
- Llamadas a funciones y retorno de ellas.
- El acceso a una estructura indexada, como un vector o matriz.

Ejemplo con estructura secuencial

```
int a, b = 5;       // 1 OE
a = b + 1;          // 2 OE
a ++;               // 2 OE
                    // ____________________
                    // Total : T(n) = 5 OE
```

Ejemplo con estructura condicional

```
int a, b = 5;       // 1 OE
cin >> a;           // 1 OE

if (a > 5) {        // 1 OE
    a += 3;         // 2 OE
    b = a + 1;      // 2 OE
}
else
    b = 1;          // 1 OE
```

En este caso, a partir de la liena 3 la cantidad de OE dependerá del valor que ingresó el usuario, y acá es donde hablamos del mejor y el peor caso.

- Mejor caso: T(n) = 1 + 1 + 1 + 1 = 4
- Peor caso: T(n) = 1 + 1 + 1 + 2 + 2 = 7

Ejemplo con estructura repetitiva

```
int n, i = 0, resultado = 1;        // 2 OE
cin >> n;                           // 1 OE

while (i < n) {                     // 1 OE
    resultado *= 2;                 // 2 OE
    i++;                            // 2 OE
}                                   // -> Realiza 5 OE n cantidad de veces, y
cuando sale                                 // del ciclo hace 1 OE más (la
comparación lógica)
cout << resultado;                  // 1 OE
                                    // ____________________________
                                    // Total : T(n) = 5 + 5 * n OE
```

Ejemplo combinado

```
int busqueda (int[] vec, int n, int dato) {  // 2 OE
```

```
    int pos = 0;                            // 1 OE.
    bool esta = false;                      // 1 OE

    while ((pos < n) && (!esta)) {          // 3 OE

        if (vec[pos] == dato)               // 2 OE -> Mejor caso, estaba en la
primera                                             //    posicion
            esta = true;                    // 1 OE.
                                            // Despues de esto vuelve al while,
a ver si                                            // se sigue cumpliendo (+3
OE) y como no                                               // cumple
(analizando el mejor caso) sale


        else
            pos++;                          // 2 OE -> Peor caso: no estaba ->
hace                                                // todo lo anterior 7 veces
por cada n
    }

    if (esta)                               // 1 OE
        return pos;                         // 1 OE -> Mejor caso: suma 2
                                            //      -> Peor caso: suma 1
    return -1;                              // 1 OE
}
// El mejor caso es: 15 OE
// El peor caso es: 7 + 7 * n OE
```

Medidas asintóticas

En general, no interesa calcular el número exacto de OE de un algoritmo, sino de acotarlo apropiadamente. Por ejemplo si tenemos dos algoritmos donde las OE son

- T1 (n) = 3n^2 +5n
- T2 (n) = 4n^2 + 7

Podemos despreciar el 5n de T1 y el 7 de T2 ya que para valores de n muy grandes son despreciables. Además se dice que ambos son del mismo orden (2) porque tendrán la misma forma a pesar de tener coeficientes principales distintos.

De esta manera, se puede decir que un algoritmo tiene un orden lineal, cuadrático, logarítmico, etc. que expresará el comportamiento dominante cuando el tamaño de la entrada sea grande, sin necesidad de calcular el coste exacto.

Podemos estar interesados en encontrar una función que, multiplicada por una constante, acote ese tiempo superiormente o acote ese tiempo inferiormente. También podemos buscar una función que multiplicada por dos constantes distintas en lugar de una permita acote el tiempo tanto superior como inferiormente. Las medidas asintóticas definen las características de funciones que verifican lo que nos interesa.

Cota Superior *O*

El concepto de *O* grande o Big O dice que un algoritmo es de orden *O(f(n))* si existe una constante *c* tal que:

$T(n) \leq c * f(n)$ para n grandes

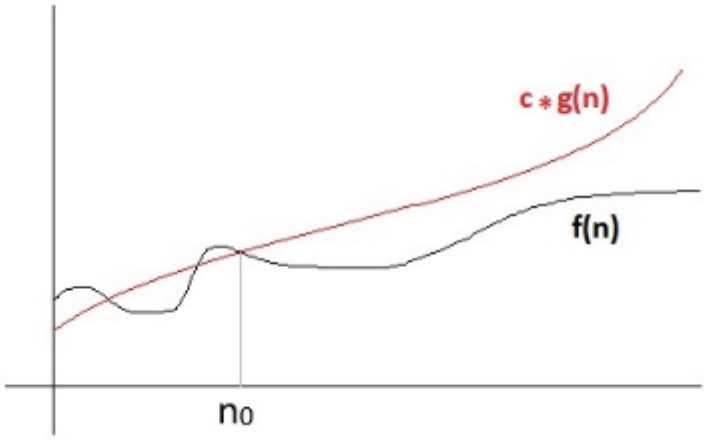
Por lo tanto, los tiempos reales nunca podrán superar a la función propuesta multiplicada por cierta constante.

En general se calcula la cota superior con el peor caso.

Definición: f pertenece a O(g) sí y solo sí existen constantes positivas c y n0 , tales que para todo n ≥ n0 se cumple que

$$0 \leq f(n) \leq c * g(n)$$

Ejemplo: si $T(n) = 5n^2 + 3n$, el algoritmo es $O(n^2)$ pero también es $O(n^3)$ y $O(2^n)$, etc. Si bien $5n^2$ no es menor que n^2, hay que recordar que no estamos indicando esa constante *c*. Si en este caso $c = 8$ la desiguladad se cumpliría para todo $n \varepsilon N$. Siempre vamos a buscar quedarnos con el órden más chico, en este caso sería el orden cuadrado.



Volviendo al ejemplo, podemos generalizar un poco más y decir que

$O(1) \subset O(\log(n)) \subset O(2) \subset \ldots \subset O(n) \subset O(n * \log(n)) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(2^n) \subset O(n!)$

Propiedades

Cuando un algoritmo es largo y tiene varios bloques puede ser dificultoso calcular el orden. Pero utilizando las propiedades que listamos a continuación podremos dividir la tarea y utilizar cálculos ya realizados.

1. Identidad: f es O(f)
2. Inclusión: Si O(f) es O(g) ⇒ $O(f) \subseteq O(g)$
3. Doble inclusión: $O(f) = O(g) \Longleftrightarrow$ f es O(g) y g es O(f)
4. Transitiva: Si f es O(g) y g es O(h) ⇒ f es O(h)
5. Cota mínima: Si f es O(g) y f es O(h) ⇒ f es O(min {g, h})
6. Suma: Si f es O(h) y g es O(i) ⇒ f + g es O(max {h, i})
7. Producto: Si f es O(h) y g es O(i) ⇒ f * g es O(h * i)(n)

Cómo calcular

La cota superior *O* simplifica mucho las cosas para calcular rápidamente el orden de un algoritmo, pero hay que tener en cuenta tres cosas:

1. Las O.E. son O(1)
2. En las secuencias se suman las complejidades individuales aplicando la regla de la suma. Por ejemplo, una secuencia finita de bloques de O(1) es O(1).
3. En los bucles en los que el número de iteraciones es fijo, el O es el del bloque ejecutado

A nosotros lo que nos interesa es ver la *forma* del crecimiento, si es O(1) es constante, si es O(n) es lineal, si es $O(n^2)$ es parabólico, etc.

Ejemplos

```
int i , a = 2 , b = 6 , c = 1;      // O(1)
for ( i = 0; i < 1000; i ++) {
    a = a * b + c - 5;              // O(1)
}
// Al principio se ejecutan sentencias que son O(1), luego en el ciclo se
ejecuta 1000 veces una sentencia que también es O(1), por lo lo tanto:
// T(n) = O(1) * O(1) = O(1)
```

Si contaramos las OE del ejemplo anterior, serían aproximadamente unas 5.000, pero como nos interesa ver el crecimiento, lo relevante es que ese número enorme es siempre igual, y el crecimiento va a ser una constante (O(1)).

```
// En los bucles que cuyo número de iteraciones es el tamaño n del problema, el
orden es // el O del bloque multiplicado por n

int i , a = 2 , b = 6 , c = 1 , n ;  // O(1)
cout << " Ingrese el valor de n ";   // O(1)
cin > > n ;                          // O(1)
for ( i = 0; i < n ; i ++) {
    a = a * b + c - 5;               // O(1)
}
// El bloque interior que es O(1) se ejecuta n veces, por lo tanto:
// T(n) = O(n) * O(1) = O(n)
```

```
// En los bucles andidados, se multiplican los O correspondientes a cada
anidamiento

int i , j , n , a = 3 , b = 2;      // O(1)
cout << " Ingrese el valor de n ";  // O(1)
cin >> n ; // O (1)
for (i = 0; i < n ; i ++) {         // O(n)
    for (j = 2; j > n ; j ++) {     // O(n)
        a = a+b;                    // O(1)
        b ++;                       // O(1)
    }
}
// El bloque interior que es O(1) depende de dos bucles anidados, por lo tanto:
// T(n) = O(n) * O(n) = (O(n²))
```

```
// En las bifurcaciones se suma el O correspondiente al análisis de la condición
más el de
// la peor rama

int i , n , a = 1 , b = 20;         // O(1)
cout << " Ingrese el valor de n ";  // O(1)
cin >> n ;                          // O(1)

if ( n > b )                        // O(1)
    for ( i = 0; i < n ; i ++) {
        a = a * b + 5
    }                               // Esta rama es O(n)
else
    b = 1;                          // Esta otra rama es O(1)
```

```
// Por lo tanto:
// T(n) = O(n)
```

Cota Inferior Ω

Es la medida asintótica de los mejores casos, no es una medida crítica porque tiene una visión optimista y no sirve para prevenir inconvenientes.

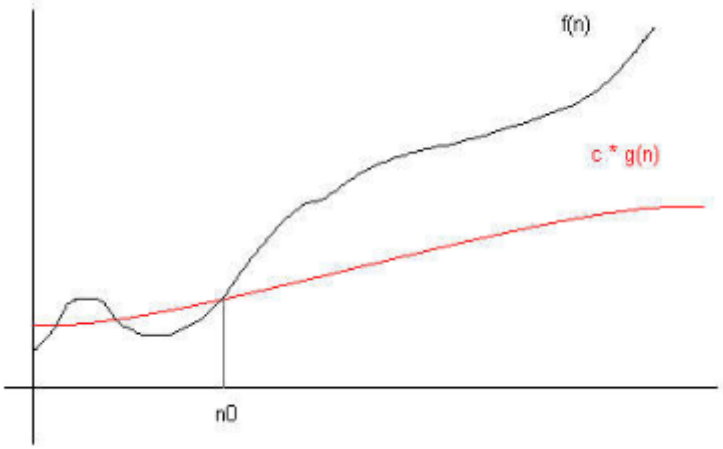
En general se utiliza la cota inferior con el mejor caso.

Definición: f(n) es Ω(g(n)) si y solo si existen constantes positivas *c* y *n0* , tales que se verifica:

$$0 \leq c * g(n) \leq f(n)$$

para todo x > n0 y para todo n ≥ k (los valores de *c* y *n0* no dependen de *n*, sino de f)

Se debe observar que f (n) es Ω(g(n)) sii g(n) es O(f (n)). En textos en inglés, suele decirse que la función f es 'lowly bounded' por g.



Orden exacto Θ

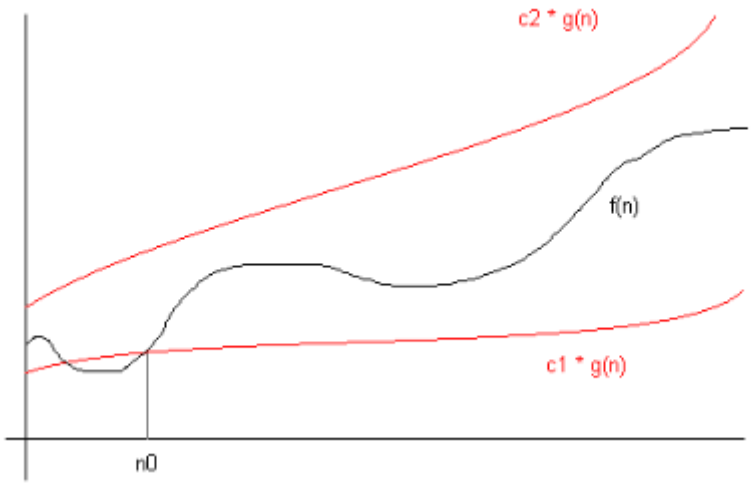
Cuando se acota la complejidad de un algoritmo tanto superior como inferiormente por la misma función se usa la notación Theta (Θ). Se dice que *f(n)* está acotado por "arriba" y por "abajo" para *n* suficientemente grande. Θ(g) es el conjunto de funciones que crecen exactamente al mismo ritmo que *g*.

En general se calcula el orden exacto para el caso promedio.

Definición: *f* es o pertenece a Θ(g(n)) significa que existen constantes reales positivas *c1*, *c2* y *k*, tales que se verifica

$$0 \leq c1 * g(n) \leq f(n) \leq c2 * g(n) \qquad \text{para todo } n \geq n0$$

- Si *f* es Θ(g), *g* es el orden exacto de *f*. Se verifica que $\Theta(f) = O(f) \cap W(f)$.
- Si *f(n)* es Θ(g(n)) se dice que *f* es de orden (de magnitud) *g*, o que *f* es de (o tiene) coste (o complejidad) *g*.



Complejidad en algoritmos recursivos

En el caso de algoritmos recursivos, el cálculo de O da origen a expresionesde recurrencia que tienen diversas formas de resolución. Básicamente, estos modos de resolución son:

1. Usando el método de expansión
2. Usando los métodos de resolución de ecuaciones de recurrencia que nos provee el Análisis Numérico
3. Usando algúin teorema cuando se den las condiciones para su aplicación

Ejemplo

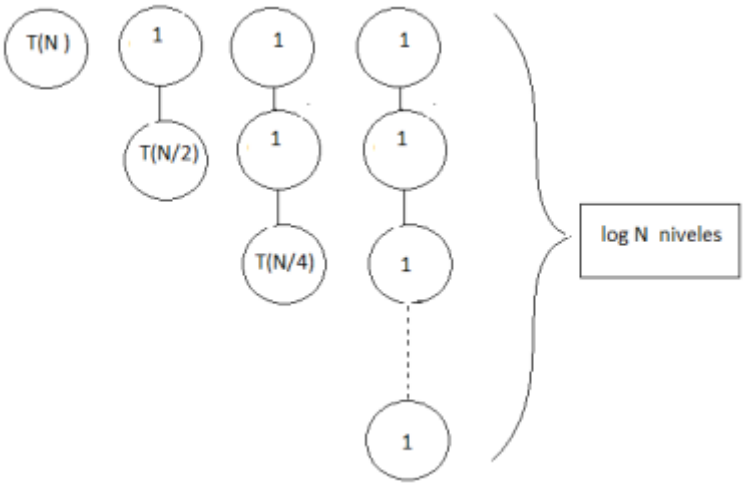
```
// Búsqueda Binaria Recursiva
int búsquedaBinaria (int vec[ ], int primero, int ultimo, int dato) {
    if(primero > ultimo)                                    // O(1)
        return -1;                                          // O(1)
    else {
        int medio = (primero + ultimo) / 2;                 // O(1)
        if (vec[medio] == dato)                             // O(1)
            return medio;                                   // O(1)
        else if (dato > vec[medio])                         // O(1)
            return BB(vec, medio + 1, ultimo);              // T(n/2)
        else
            return BB(vec, pri, medio - 1);                 // T(n/2)
    }
}
```

Se puede observar que las dos peores ramas tienen un coste de:

-- ***T(n)*** = *O(1) + O(1) + O(1) + O(1) + T(n/2) = O(1) + T(n/2) =* ***1 + T(n/2)*** --

El tiempo de ejecución del algoritmo será la sumatoria de los tiempos de los nodos. Tenemos nodos de coste 1 y hay una cantidad de ellos que se puede contabilizar como log2 N, por lo tanto

T (n) ε O(logn)



Ejemplo

```
int factorial (int n) {
    if (n < 2)
        return 1;
    return n * factorial (n -1);
}
```

*-- **T(n)** = 1 + T(n - 1) = 1 + 1 + T(n - 2) = 1 + 1 + 1 + T(n - 3) ... = n + T(0) = **n + 1** --*