OPTIMIZACIÓN DE CÓDIGO

CODE UD6 – Tema 1

IES Plurilingüe Antón Losada Diéguez

Adrián Fernández González

Tabla de contenido

1. Introducción....3
2. Equilibrio o solución de compromiso (tradeoff)....3
3. Cuándo y qué optimizar....4
3.1. Consumo de tiempo....4
3.2. Bugs añadidos....4
3.3. Pérdida de legibilidad....4
3.4. Impacto....4
3.5. Cuellos de botella....4
4. Compilador optimizador....4
5. Rendimiento....5
5.1. Tiempos....5
5.2. Capacidad del sistema....5
6. Optimización de código....6
6.1. Evaluaciones innecesarias....6
6.2. Invocaciones innecesarias....6
6.3. Variables locales....7
6.4. Variables innecesarias....7
6.5. Expansión/desenroscar bucles....8
6.6. Creación de objetos....8
6.7. Liberación de recursos....9
6.8. Bucles infinitos....9
6.9. Return y break....10
6.9.1. Usos justificados del break....10
6.10. Anidamiento de condiciones....11
6.11. Reordenación del código....12
6.12. Tipos adecuados....12
6.13. Comprobaciones innecesarias....13
6.14. Bucles y recursividad....13
6.15. Algoritmos probados....13
6.16. División de matrices....13
6.17. División de mapas o arrays asociativos....14
7. Eficiencia algorítmica....14
7.1. Cálculo de O....14

7.1.1. Valor O(1). Constante 14
7.1.2. Valor O(log n). Logarítmico 14
7.1.3. Valor O(n). Lineal 14
7.1.4. Valor O(n log n). Quasiliniar 14
7.1.5. Valor O(n^2). Cuadrático 15
7.1.6. Valor O(n^x). Exponencial 15
7.2. Cálculo de tiempo 15

Optimización de código

1. Introducción

La optimización de código es el proceso por el cual se modifica el software para que sea más óptimo para un ámbito, circunstancias y sistema en concreto.

El proceso de optimización, **no modifica el funcionamiento** del software, solo hace que realice sus operaciones de **forma más eficiente**.

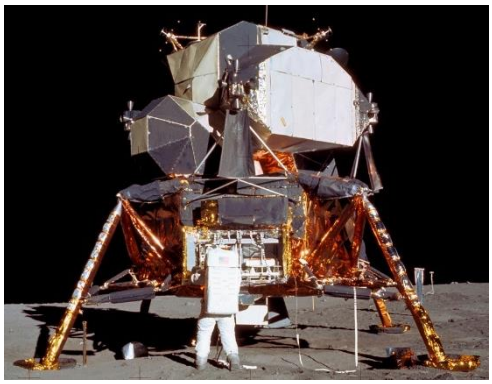
Esta mejora puede ser de cualquier recurso, tiempo, energía, consumo de red, memoria, disco, etc. Incluso el tiempo de programación puede ser optimizado si es necesario acortar los tiempos de desarrollo y mantenimiento.

2. Equilibrio o solución de compromiso (tradeoff)

El proceso de optimización del consumo de un recurso en concreto puede acarrear el empeoramiento del consumo de otro. Por ejemplo, para ahorrar energía puede ser necesario realizar un mayor número de operaciones y consumir más memoria, pero dichas operaciones ser más eficientes energéticamente.

Por otro lado, la seguridad es algo vital en un sistema, por lo que puede ser necesario priorizarla frente a otros procesos de optimización.

Por todo esto, es habitual tener que hacer sacrificios de unos recursos a favor de otros o, en el mejor de los casos, encontrar una solución de compromiso (tradeoff) que llegue a un acuerdo entre ambas partes, un equilibrio.



Por ejemplo, durante el proceso de fabricación del Módulo Lunar Apollo, por motivos de ligereza, era necesario que el módulo tuviese tres patas, pero era inseguro. Por motivos de seguridad tenía que tener cinco, pero era demasiado pesado. Al final, llegaron a un acuerdo intermedio, ponerle cuatro patas.

En informática es habitual que se sacrifique memoria para reducir el consumo de otros recursos. Un ejemplo de esto es la recursividad.

3. Cuándo y qué optimizar

Hay que tener en cuenta que el proceso de optimización no es perfecto y también acarrea una serie de factores a considerar para saber cuándo realizar dicho proceso o si realmente se debería realizar y en qué parte del sistema.

3.1. Consumo de tiempo

Como es lógico, el proceso de optimización conlleva un tiempo, por lo que hay que tenerlo en cuenta a la hora de realizarlo. Si una entrega está cerca, no es conveniente optimizar, es mejor retrasar el proceso para cuándo se tenga más tiempo, de lo contrario, el sistema puede retrasarse y/o la optimización no ser lo suficientemente óptima, lo que requerirá más tiempo en el futuro para volver a optimizarlo.

3.2. Bugs añadidos

Aunque no se cambie el funcionamiento del sistema, todo cambio en el mismo puede hacer que aparezcan errores inesperados.

Es importante realizar pruebas y análisis tras cada proceso de optimización para detectar y solventar todos los posibles bugs añadidos durante dicho proceso.

3.3. Pérdida de legibilidad

Un problema que puede acarrear la optimización es que el código se vuelva más difícil de leer en algunos casos, por lo que, puede ser necesario el documentar de una forma más intensiva para que el código se entienda con mayor facilidad.

3.4. Impacto

Otro factor a tener en cuenta es el impacto que tendrá dicho proceso de optimización.

Si la optimización de una parte del sistema no acarrea un impacto real en el consumo de recursos, es mejor no optimizarla, puesto que acarreará un tiempo que puede ser empleado en el desarrollo u optimización de otra parte más importante.

3.5. Cuellos de botella

Uno de los puntos importantes a detectar y optimizar son los cuellos de botella, los puntos críticos del sistema en los que se realizan la mayoría de las operaciones y por tanto el consumo de recursos. Al optimizar estos cuellos de botella se mejora el sistema de forma drástica.

Basándose en el principio de Pareto o regla del 80/20, en informática se cumple que el 90% de los recursos de un sistema son consumidos por el 10% del código. Por tanto, la optimización de esta pequeña parte del sistema puede producir un enorme impacto en el consumo de los recursos.

4. Compilador optimizador

No todos los procesos de optimización se realizan de forma manual por el programador, los propios compiladores, intérpretes y traductores realizan optimizaciones basándose en el funcionamiento del lenguaje o la arquitectura de la máquina sobre la que va a ser utilizado el sistema. Esto se aplica a los lenguajes compilados, interpretados y Just-In-Time.

Aunque se realicen de forma automática, pueden ser afinados mediante precompiladores y otro tipo de herramientas que realizan operaciones previas al compilado/interpretación.

En ningún caso, el uso de estas herramientas o la optimización por parte de los compiladores substituye al proceso de optimización del programador, en cuyo caso, lo complementan y mejoran en ciertos aspectos que puede no conocer o pasar por alto.

5. Rendimiento

El rendimiento es la medida de la velocidad con la que se realiza una tarea y el resultado que da. Aunque se suele pensar que el procesador es lo que más afecta en el rendimiento, cada parte del sistema, sus dependencias, otro software presente en la máquina, la red, el hardware y hasta la arquitectura del equipo afectan al rendimiento de la misma.

Estos valores varían dependiendo de la carga del sistema, las acciones solicitadas, la extensión de las mismas en el tiempo, etc. Por lo que es importante establecer los rangos aceptables para el sistema, sus partes y las acciones a realizar, así se podrá optimizar más a fondo y perfeccionar aquellas partes o acciones que no cumplan los parámetros aceptables.

Por todo esto, es necesario tener presente el rendimiento del sistema desde el inicio, escogiendo las tecnologías adecuadas y gastando más tiempo optimizando aquellas partes críticas para que el sistema cumpla con las exigencias.

El rendimiento se puede ver desde dos puntos de vista fundamentales, externo, el usuario o interno, los recursos del sistema y los principales factores a medir son los tiempos y la capacidad del sistema.

5.1. Tiempos

Existen dos fundamentales:

- **Tiempo de respuesta:** El que transcurre desde que el usuario solicita algo al sistema y este le devuelve lo solicitado.
- **Tiempo de reacción del sistema:** Tiempo que pasa desde que recibe la orden y se empieza a ejecutar dicha orden.

5.2. Capacidad del sistema

Existen tres tipos fundamentales:

- **Capacidad máxima:** La cantidad máxima de carga que puede tener el sistema antes de salir de los parámetros aceptables.
- **Carga de trabajo:** La cantidad de carga que el sistema acepta sin que el tiempo de respuesta cambie.
- **Capacidad de ejecución:** Cantidad de trabajo realizado en una unidad de tiempo determinada.

6. Optimización de código

Existen una serie de buenas prácticas, recomendaciones y cosas a evitar a la hora de programar que pueden mejorar y optimizar el código.

Si estas prácticas se interiorizan y se realizan cada vez que se programa, el código resultante será de mejor calidad, más eficiente, consumirá menos recursos y será más rápido.

Aunque hay muchos tipos y en cada lenguaje e incluso sistema pueden ser aplicables unas y otras no, a continuación, se exponen las más habituales.

6.1. Evaluaciones innecesarias

Si en un bloque de código, y en especial en los bucles, un valor calculado se utiliza múltiples veces, es más óptimo guardarlo en una variable y no recalcularlo múltiples veces.

```
for (int i = 0; i < 10; i++)
{
   a = (b + c) / i ;
}
```

En este ejemplo, el cálculo de b + c se realiza 10 veces, una por cada iteración. Dicho valor no varía de una iteración a otra, por lo que es mejor, más óptimo, guardarlo previamente en una variable.

```
int bc = b + c;
for (int i = 0; i < 10; i++)
{
    a = bc / i;
}
```

6.2. Invocaciones innecesarias

Es habitual el uso de llamadas a métodos y funciones varias veces en un bloque de código, sobre todo en los bucles, para obtener un valor determinado. La llamada a un método o función es muy costosa, por lo que es recomendable guardar el valor previo a su uso múltiples veces.

```
for(int i = 0; i < this.size(); i++)
{
    System.out.println("Iteración número: " + i);
}
```

En este ejemplo, se invoca repetidas veces al método *size*, una cada vez que se realiza una iteración, pese a que el valor no cambia. Por ello, es más óptimo guardar ese valor en una variable.

```
int size = this.size();
for(int i = 0; i < size; i++)
{
    System.out.println("Iteración número: " + i);
}
```

Otro caso es la llamada a atributos o la llamada a métodos. Si se tiene acceso a un atributo propio de la clase, ha de usarse directamente, no llamar al *getter* a no ser que el valor tenga que ser calculado.

```
for(int i = 0; i < this.length; i++)
{
    System.out.println("Iteración número: " + i);
}
```

Teniendo en cuenta el ejemplo anterior, es mucho más eficiente acceder al atributo *length* que llamar al método *size*, siempre que ambos sean locales.

Esto también es aplicable a los accesos a elementos dentro del bucle, si se va a utilizar múltiples veces el valor de un *array* en una posición determinada, por ejemplo, es recomendable guardarlo en una variable.

```
for(int i = 0; i < this.length; i++)
{
    String val = arr[i];
    //Uso del valor varias veces
}
```

6.3. Variables locales

Siempre es más eficiente la declaración y uso de variables locales, por lo que, siempre que sea posible, es recomendable hacerlo en el ámbito más pequeño si no se van a usar fuera de dicho ámbito.

```
String val = "";
for(int i = 0; i < this.length; i++)
{
    val = arr[i];
}
```

En este caso, lo recomendable es declarar la variable *val* dentro del *for*, ya que no se va a usar en ningún otro lugar.

```
for(int i = 0; i < this.length; i++)
{
    String val = arr[i];
}
```

Aunque a nivel de eficiencia este aspecto en concreto no afecta en gran medida gracias a los compiladores optimizadores de la mayoría de lenguajes, en ciertos ámbitos si ve afectado el rendimiento, por lo que es recomendable seguir siempre esta práctica.

6.4. Variables innecesarias

Como todo, las variables ocupan un espacio en memoria, por lo que, si no se van a utilizar o pueden ser obviadas, no se crean.

```java
public String mssg()
{
    String init = "Hola ";
    String fin = "Mundo";

    String res = init + fin;
    return res;
}
```

En este ejemplo, las variables son innecesarias, ya que nunca cambian de valor y se usan solo para concatenarlas, pudiendo ser declaradas como una si se fuese a utilizar posteriormente o, incluso, hacer el *return* directamente como la cadena.

La mayoría de los IDE avisan cuando una variable o atributo es declarado y nunca utilizado y otros más potentes o ciertas herramientas, avisan cuando una variable nunca cambia su valor y puede ser obviada.

6.5. Expansión/desenroscar bucles

Cuando un bucle es excesivamente grande y la operación relativamente corta, se puede hacer lo que se denomina expansión de bucles, en la que se realizan varias veces las operaciones para reducir el número de iteraciones.

```java
for(int i = 0; i < 1000; i++)
{
    funcion(arr[i]);
}
```

```java
for(int i = 0; i < 100;)
{
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
    funcion(arr[i++]);
}
```

En este ejemplo, se pasa de un bucle de 1000 iteraciones realizando una función a uno de 100 iteraciones que realiza 10 funciones. Nótese el *i++* y la falta del incremento en el *for*.

6.6. Creación de objetos

Aunque anteriormente se menciona que es más eficiente la creación de las variables lo más locales posibles, la creación de los objetos es extremadamente costosa, por lo que es recomendable reutilizar un objeto antes de crearlo una y otra vez.

Esto solo es aplicable en el caso de que no se necesite crearlo con el constructor, si no que se varían sus valores.

```
for(int i = 0; i < 10; i++)
{
    Cubo c = new Cubo(10, i * 2, i + 5);
    System.out.println("El área del cubo es: " c.getArea());
}
```

En este ejemplo, el cubo solo se utiliza para calcular un área, por lo que da igual utilizar el mismo objeto o no.

```
Cubo c = new Cubo(10, 0, 5);
for(int i = 0; i < 10; i++)
{
    c.setHeight(i * 2);
    c.setDeep(i + 5);
    System.out.println("El área del cubo es: " c.getArea());
}
```

En este otro ejemplo, el objeto se crea fuera del bucle y se establecen los valores dentro, ahorrando muchos recursos.

6.7. Liberación de recursos

Algo de vital importancia en todo sistema es la gestión de conexiones y flujos de datos, controlar cuando se dejan de utilizar y cerrarlos correctamente.

Si una conexión queda abierta, seguirá consumiendo los recursos del sistema y, en caso de conectarse a otro sistema, como una base de datos, los recursos de ambos podrían verse afectados.

Incluso el acceso a un fichero puede acarrear problemas graves, puesto que el archivo puede quedar bloqueado por el sistema e impedir que sea utilizado por otros mientras el flujo permanezca abierto.

Por todo esto, toda conexión y flujo que se utilice, ha de ser cerrado adecuadamente cuando ya no sea necesario.

6.8. Bucles infinitos

Aunque también forma parte de las malas prácticas de programar, es obvio que la aparición de bucles infinitos, intencionados o no, acarrea un consumo de recursos innecesarios.

```
while(true)
{
    ...
    if(x = false)
    {
        break;
    }
}
```

El uso de bloques de código como el del ejemplo acarrea problemas de rendimiento, puesto que, para romper el bucle es necesario el uso de *break*, que termina de forma abrupta e inesperada la ejecución del código.

Todo código creado de esta forma puede ser recreado de otra más eficiente, pues significa que no se ha encontrado la opción adecuada.

Otro problema de este mal uso es que el optimizador del compilador tampoco sabe cuándo ni cómo va a terminar, por lo que no puede optimizar ese bloque de código.

6.9. Return y break

El código está pensado para ser ejecutado de forma secuencial, sin cortes abruptos ni inesperados, por lo el uso de break está desaconsejado, reservado para casos muy concretos y los *switch*.

El uso de un *return* tampoco está justificado si el método o función no tiene previsto devolver nada, ya que causa una salida igual de abrupta.

```
public void prueba()
{
    ...
    if(cosa)
    {
        return;
    }
}
```

En este ejemplo se puede ver como el *return* no devuelve nada, simplemente se utiliza para romper el método.

Esto también es aplicable al final de los métodos y procedimientos que no devuelven nada, todo método termina al llegar a la última línea ejecutable, no hace falta poner un *return*.

```
public void prueba()
{
    ...
    return;
}
```

```
public void prueba()
{
    if(cosa)
    {
        ...
        return;
    }else{
        ...
        return;
    }
}
```

Por tanto, el *return* ha de quedar para uso exclusivo de métodos y funciones que devuelvan valores.

6.9.1. Usos justificados del break

En ciertos casos, el uso del *break* para romper un bucle está justificado, cuando el no uso conllevaría un gasto innecesario y extremo de recursos.

```
for(String dato : String[] datos)
{
    if(dato == "x")
    {
        //realizo operación
        break;
    }
}
```

En este ejemplo, se recorren una serie de datos de gran tamaño, en este caso un array, y se realiza una operación solo con un dato en concreto. Si la ejecución siguiese, el bucle recorrería todos los datos pese a que no se necesita realizar más operaciones, por lo que se gasta una cantidad de tiempo inútil.

Este uso justificado puede ser mejorado mediante el uso de una función o método auxiliar.

Si un bloque de código fuera a necesitar el uso del break, dicho bloque se debería convertir en un método o función auxiliar que, al alcanzar su objetivo, devolviese un valor mediante el uso de *return*. Esto rompe la ejecución de una forma prevista y consume menos recursos.

```
private boolean opDato(datos)
    for(String dato : String[] datos)
    {
        if(dato == "x")
        {
            //realizo operación
            return true;
        }
    }
    return false;
}
```

Como se puede ver en el ejemplo, el bloque de código anterior se ha transformado en un método sencillo que devuelve *true* al realizar la operación y *false* en caso contrario, realizando la operación de una forma más eficiente y sin el uso del *break*.

6.10. Anidamiento de condiciones

Además del problema de legibilidad, el uso inadecuado de los *if-else* y el correcto anidamiento de condiciones puede acarrear un consumo de recursos innecesarios.

```
if(x == 1)
{
    ...
}
if(x > 5)
{
    ...
}
if(x < -1)
{
    ...
}
```

Como se puede ver en el ejemplo, las tres opciones son excluyentes, por lo que nunca se darán a la vez. Al no estar anidadas, se realizarán las comparaciones de forma secuencial, evaluando casos innecesarios, incluso cuando una ya se ha cumplido, consumiendo recursos.

```
if(x == 1)
{
    ...
}else{
    if(x > 5)
    {
        ...
    }else{
        if(x < -1)
        {
            ...
        }
    }
}
```

En este otro ejemplo, se crea la misma estructura, pero de forma anidada, haciendo que solo se evalúen los otros casos si uno de ellos no se ha cumplido ya.

6.11. Reordenación del código

Algo tan simple como reordenar el código puede optimizarlo si se conoce cómo funciona el sistema sobre el que va a actuar. Por ejemplo, la obtención de datos de una consulta a una base de datos es más eficiente si se realiza de forma secuencial, de la primera fila a la última y de la primera columna a la última.

Por tanto, si se obtienen los valores en ese orden antes de realizar las operaciones, se optimizará el proceso.

6.12. Tipos adecuados

Elegir el tipo adecuado para una variable es algo de vital importancia, pues el uso de uno u otro determina el espacio que consume en memoria y sus funcionalidades serán más o menos eficientes para un caso concreto.

```
String str = "Hola";
char[] chars = new char[] {'H', 'o', 'l', 'a'};
```

En este ejemplo pueden verse dos formas de guardar la cadena “Hola”, la primera con una variable de tipo *String*, la segunda con un *array* de caracteres. Habitualmente, la opción más óptima sería la primera, ya que se necesitaría obtener cada valor del *array* para mostrar el mensaje, con el coste que eso supone.

El uso de tipos no básicos como listas, mapas, vectores, etc. es adecuado cuando se puede sacar partido a sus funcionalidades de una forma lógica y justificada. Cuando no se saca partido de ellos, su mayor consumo de recursos es innecesario frente al uso de un *array*, por ejemplo.

Otro factor a tener en cuenta son las limitaciones o el uso que se le va a dar al tipo en concreto. Por ejemplo, ciertas implementaciones de listas son más eficientes para un tipo de

operaciones en concreto, si van a ser utilizadas para dichas operaciones, es mejor optar por ese tipo frente a otro.

6.13. Comprobaciones innecesarias

Si una condición nunca se va a dar, no debe ser evaluada, ya que un simple *if* consume recursos.

```
Casa c = null;
if(c == null)
{
    c = new Casa();
}
```

En este ejemplo puede verse como la variable c siempre va a ser nula, la condición siempre se cumplirá, por lo que la comprobación es innecesaria y debe omitirse.

```
Casa c = null;
c = new Casa();
```

6.14. Bucles y recursividad

Un bucle puede ser transformado en una función recursiva y viceversa. Dependiendo del lenguaje, una opción u otra puede ser más eficiente.

Aunque la recursividad ocupa más memoria que un bucle, suele ser mucho más rápida.

Hay que tener en cuenta que la recursividad solo es eficiente si en su interior no se crean objetos y el menor número de variables posibles. Además, la mayoría de los lenguajes optimizan la recursividad solo cuando esta se realiza en la última línea de código ejecutado de la función o en un *return* (*tail recursion*).

A términos generales, la recursividad, se suele utilizar para operaciones de búsqueda, ordenación, recorrer conjuntos de datos y ciertas operaciones matemáticas y los bucles para lo demás.

6.15. Algoritmos probados

Existen diversos algoritmos extensamente probados y de demostrada eficiencia resolviendo el problema para el fueron diseñados.

Siempre que exista un algoritmo probado para una parte del sistema, es aconsejable su uso, puesto que será extremadamente eficiente.

Los algoritmos más conocidos son los de ordenación y búsqueda, diseñados para ordenar conjuntos de datos y para buscar un dato concreto en ellos.

6.16. División de matrices

Las matrices y los arrays multidimensionales ocupan gran cantidad de memoria, ya que se reserva la estructura completa. Además, se guarda de forma consecutiva, por lo que no se puede optimizar el espacio libre tan fácilmente.

Por ello, y siempre que sea posible, es recomendable dividir la estructura en varios arrays, que ocupan menos memoria, optimizando el proceso.

6.17. División de mapas o arrays asociativos

Los mapas o arrays asociativos en otros lenguajes, también pueden ser divididos en dos arrays parejos, de tal forma que uno represente los valores y otros las claves.

Hay que tener en cuenta que se tendrán que sincronizar, de tal forma que ambos han de tener el mismo tamaño y el índice de cada pareja ser el mismo para ambos arrays.

7. Eficiencia algorítmica

La eficiencia algorítmica es el cálculo de la eficiencia de un bloque de código o algoritmo en base a, habitualmente, el tiempo que tardaría.

Hay que tener en cuenta que la eficiencia de un algoritmo puede depender del número de elementos a tratar en un conjunto, el tipo de datos de entrada, etc. Por lo que este valor representa una media o tendencia teórica, pero cada caso concreto puede ser distinto y alejarse de este valor.

En ocasiones, se realiza un cálculo para el peor caso posible, el caso medio y el mejor caso posible o tomando como referencia un rango de valores posibles.

7.1. Cálculo de O

Lo primero que se suele hacer es el cálculo teórico del coste computacional de cada operación. Para ello, se establece un valor en forma de función O (cota superior asintótica), que puede ser representada como una gráfica de tiempo para una visualización rápida del coste de tiempo.

Una vez realizado el cálculo de cada uno, se guardan para pasarlo a tiempo real y se realiza una suma de todos los valores para tener una idea teórica de la función que representa.

Las instrucciones condicionales se calculan teniendo en cuenta el máximo de todos los posibles flujos de operación. Por ejemplo, en un *if-else*, se calcula el coste del *if*, el coste del *else* y se le establece el mayor coste de los dos.

7.1.1. Valor O(1). Constante

Este valor es el más bajo, pues es constante y se aplica a las operaciones básicas, desde asignaciones o creaciones a operaciones matemáticas o comparaciones.

7.1.2. Valor O(log n). Logarítmico

El valor logarítmico lo tienen las operaciones recursivas.

7.1.3. Valor O(n). Lineal

Este valor es el de los bucles, que dependerán del número de iteraciones que realicen.

7.1.4. Valor O(n log n). Quasiliniar

Este valor es el de las operaciones recursivas dentro de un bucle. El uso de recursividad y bucles de forma conjunta es escaso y solo unos pocos algoritmos la usan.

7.1.5. Valor $O(n^2)$. Cuadrático

El valor cuadrático es el de dos bucles anidados, uno dentro del otro, ya que tardarán el número de las iteraciones del superior por el del inferior.

7.1.6. Valor $O(n^x)$. Exponencial

Este es la generalidad del anterior para cualquier anidamiento de bucles, ya que tardan n veces por cada bucle que haya.

7.2. Cálculo de tiempo

Una vez se tiene el cálculo del coste computacional del bloque de código, se miden los tiempos de cada operación, se multiplica por el coste de los mismos y se suman los valores. Este cálculo se puede realizar por una única operación o mediante rangos.

Por ejemplo, si cada iteración del bucle se realiza en 0.5s, el tiempo que tardará será 0.5s * n, siendo n el número de iteraciones. Si se realizan entre 5 y 10 operaciones cada vez, el bucle tardará entre 2.5s y 5s.

Hay que tener en cuenta que esto es aproximado, ya que, a mayor cantidad de operaciones y complejidad de las mismas, los tiempos por operación pueden resultar ser mucho mayores por operación, no es lo mismo realizar una operación con cuatro valores que con mil. Además, los tiempos no son estables, en un momento dado el procesador puede estar sobrecargado o realizar otras operaciones y el tiempo aumentar.