CAPITULO 1

# Algoritmos y estructuras de datos

**Objetivos**

Con el estudio de este capítulo, usted podrá:

- Revisar los conceptos básicos de tipos de datos.
- Introducirse en las ideas fundamentales de estructuras de datos.
- Revisar el concepto de algoritmo y programa.
- Conocer y entender la utilización de la herramienta de programación conocida por “pseudocódigo”.
- Entender los conceptos de análisis, verificación y eficiencia de un algoritmo.
- Conocer las propiedades matemáticas de la notación O.
- Conocer la complejidad de las sentencias básicas de todo programa Java.

**Contenido**



**Conceptos clave**

- ♦ Algoritmo.
- ♦ Complejidad.
- ♦ Eficiencia.
- ♦ Estructura de datos.
- ♦ Notación asintótica.
- ♦ Pseudocódigo.
- ♦ Rendimiento de un programa.
- ♦ Tipo de dato.

# INTRODUCCIÓN

La representación de la información es fundamental en ciencias de la computación y en informática. El propósito principal de la mayoría de los programas de computadoras es almacenar y recuperar información, además de realizar cálculos. De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y de los algoritmos que las manipulan constituye el núcleo central de la *informática* y de la *computación*.

Se revisan en este capítulo los conceptos básicos de *dato, abstracción, algoritmos* y *programas*, así como los criterios relativos a *análisis y eficiencia de algoritmos.*

## 1.1. TIPOS DE DATOS

Los lenguajes de programación tradicionales, como Pascal y C, proporcionan *tipos de datos* para clasificar diversas clases de datos. La ventajas de utilizar tipos en el desarrollo de software [TREMBLAY 2003] son:

- Apoyo y ayuda en la prevención y en la detección de errores.
- Apoyo y ayuda a los desarrolladores de software, y a la comprensión y organización de ideas acerca de sus objetos.
- Ayuda en la identificación y descripción de propiedades únicas de ciertos tipos.

---

### Ejemplo 1.1

*Detección y prevención de errores mediante lenguajes tipificados (tipeados), como Pascal y ALGOL, en determinadas situaciones.*

La expresión aritmética

```
6 + 5 + "Ana la niña limeña"
```

contiene un uso inapropiado del literal cadena. Es decir, se ha utilizado un objeto de un cierto tipo (datos enteros) en un contexto inapropiado (datos tipo cadena de caracteres) y se producirá un error con toda seguridad en el momento de la compilación, en consecuencia, en la ejecución del programa del cual forme parte.

---

Los tipos son un enlace importante entre el mundo exterior y los elementos datos que manipulan los programas. Su uso permite a los desarrolladores limitar su atención a tipos específicos de datos, que tienen propiedades únicas. Por ejemplo, el tamaño es una propiedad determinante en los arrays y en las cadenas; sin embargo, no es una propiedad esencial en los valores lógicos, verdadero `(true)` y falso `(false)`.

**Definición 1:** Un *tipo de dato* es un conjunto de valores y operaciones asociadas a esos valores.

**Definición 2:** Un *tipo de dato* consta de dos partes: un conjunto de datos y las operaciones que se pueden realizar sobre esos datos.

En los lenguajes de programación hay disponible un gran número de tipos de datos. Entre ellos se pueden destacar los tipos primitivos de datos, los tipos compuestos y los tipos agregados.

## 1.1.1. Tipos primitivos de datos

Los tipos de datos más simples son los *tipos de datos primitivos,* también denominados *datos atómicos* porque no se construyen a partir de otros tipos y son entidades únicas no descomponibles en otros.

Un **tipo de dato atómico** es un conjunto de datos atómicos con propiedades idénticas. Estas propiedades diferencian un tipo de dato atómico de otro. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

***Tipo de dato atómico***

1. Un conjunto de valores.
2. Un conjunto de operaciones sobre esos valores.

---

**Ejemplo 1.2**

*Diferentes tipos de datos atómicos*

**Enteros**

| | | |
|---|---|---|
| | valores | `-∞ , …, -3, -2, -1, 0, 1, 2, 3, …, +∞` |
| | operaciones | `*, +, -, /, %, ++, --, …` |

**Coma flotante**

| | | |
|---|---|---|
| | valores | `- , …, 0.0, …,` |
| | operaciones | `*, +, -, %, /, …` |

**Carácter**

| | | |
|---|---|---|
| | valores | `\0, …, 'A', 'B', …, 'a', 'b', …` |
| | operaciones | `<, >, …` |

**Lógico**

| | | |
|---|---|---|
| | valores | `verdadero, falso` |
| | operaciones | `and, or, not, …` |

---

Los **tipos numéricos** son, probablemente, los tipos primitivos más fáciles de entender, debido a que las personas están familiarizadas con los números. Sin embargo, los números pueden también ser difíciles de comprender por los diferentes métodos en que son representados en las computadoras. Por ejemplo, los números decimales y binarios tienen representaciones diferentes en las máquinas. Debido a que el número de bits que representa los números es finito, sólo los subconjuntos de enteros y reales se pueden representar. A consecuencia de ello, se pueden presentar situaciones de desbordamiento (*underflow* y *overflow*) positivo y negativo al realizar operaciones aritméticas. Normalmente, los rangos numéricos y la precisión varía de una máquina a otra. Para eliminar estas inconsistencias, algunos lenguajes modernos, como Java y C#, tienen especificaciones precisas para el número de bits, el rango y la precisión numérica de cada operación para cada tipo numérico.

```
0          23         786        456          999
7.56       4.34       0.897      1.23456      99.999
```

El tipo de dato **boolean (lógico)** suele considerarse como el más simple debido a que sólo tiene dos valores posibles: `verdadero (true)` y `falso (false)`. El formato sintáctico de estas constantes lógicas puede variar de un lenguaje de programación a otro. Algunos lenguajes ofrecen un conjunto rico de operaciones lógicas. Por ejemplo, algunos lenguajes tienen construcciones que permiten a los programadores especificar operaciones condicionales o en cortocircuito. Por ejemplo, en Java, en los operadores lógicos `&&` y `||` sólo se evalúa el segundo operando si su valor se necesita para determinar el valor de la expresión global.

El **tipo carácter** consta del conjunto de caracteres disponible para un lenguaje específico en una computadora específica. En algunos lenguajes de programación, un carácter es un símbolo indivisible, mientras que una cadena es una secuencia de cero o más caracteres. Las cadenas se pueden manipular de muchas formas, pero los caracteres implican pocas manipulaciones. Los tipos carácter, normalmente, son dependientes de la máquina.

```
'Q'      'a'      '8'      '9'      'k'
```

Los códigos de carácter más utilizados son EBCDIC (utilizado por las primeras máquinas de IBM) y ASCII (el código universal más extendido). La aparición del lenguaje Java trajo consigo la representación de caracteres en Unicode, un conjunto internacional de caracteres que unifica a muchos conjuntos de caracteres, incluyendo inglés, español, italiano, griego, alemán, latín, hebreo o indio. En Java, el tipo `char` se representa como un valor de 16 bits. Por consiguiente, el rango de `char` está entre `0` y `65.536`, lo que permite representar prácticamente la totalidad de los alfabetos y formatos numéricos y de puntuación universales.

## 1.1.2. Tipos de datos compuestos y agregados

Los **datos compuestos** son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado. Un ejemplo sencillo es el número de su teléfono celular `51199110101`. Realmente, este número consta de varios campos, el código del país (`51`, Perú), el código del área (`1`, Lima) y el número propiamente dicho, que corresponde a un celular porque empieza con `9`.

En algunas ocasiones los datos compuestos se conocen también como datos o tipos agregados. Los **tipos agregados** son tipos de datos cuyos valores constan de colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos. Existen tres tipos agregados básicos: *arrays* (arreglos), *secuencias* y *registros*.

Un ***array*** o **arreglo**[1] es, normalmente, una colección de datos de tamaño o longitud fija, cada uno de cuyos datos es accesible en tiempo de ejecución mediante la evaluación de las expresiones que representan a los subíndices o índices correspondientes. Todos los elementos de un array deben ser del mismo tipo.

```
array de enteros:   [4, 6, 8, 35, 46, 0810]
```

Una **secuencia** o **cadena** es, en esencia, un array cuyo tamaño puede variar en tiempo de ejecución. Por consiguiente, las secuencias son similares a arrays dinámicos o flexibles.

```
Cadena  = "Aceite picual de Carchelejo"
```

[1] El término inglés *array* se traduce en casi toda Latinoamérica por arreglo, mientras que en España se ha optado por utilizar el término en inglés o bien su traducción por "lista" , "tabla" o "matriz".

Un **registro** puede contener elementos datos agregados y primitivos. Cada elemento agregado, eventualmente, se descompone en *campos* formados por elementos primitivos. Un registro se puede considerar como un tipo o colección de datos de tamaño fijo. Al contrario que en los arrays, en los que todos sus elementos deben ser del mismo tipo de datos, los campos de los registros pueden ser de diferentes tipos de datos. A los campos de los registros se accede mediante identificadores.

El registro es el tipo de dato más próximo a la idea de objeto. En realidad, el concepto de objeto en un desarrollo orientado a objetos es una generalización del tipo registro.

```
Registro   {
              Dato1
              Dato2
              Dato3
               ...
           }
```

## 1.2. LA NECESIDAD DE LAS ESTRUCTURAS DE DATOS

A pesar de la gran potencia de las computadoras actuales, la eficiencia de los programas sigue siendo una de las características más importantes a considerar. Los problemas complejos que procesan las computadoras cada vez más obligan, sobre todo, a pensar en su eficiencia dado el elevado tamaño que suelen alcanzar. Hoy, más que nunca, los profesionales deben formarse en técnicas de construcción de programas eficientes.

En sentido general, una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Bajo esta óptica, cualquier representación de datos, incluso un número entero o un número de coma flotante almacenado en la computadora, es una sencilla estructura de datos. En un sentido más específico, una estructura de datos es una organización o estructuración de una colección de elementos dato. Así, una lista ordenada de enteros almacenados en un array es un ejemplo de dicha estructuración.

Una **estructura de datos** es una agregación de tipos de datos compuestos y atómicos en un conjunto con relaciones bien definidas. Una estructura significa un conjunto de reglas que contienen los datos juntos.

Las estructuras de datos pueden estar *anidadas:* se puede tener una estructura de datos que conste de otras.

***Estructura de datos***

1. Una combinación de elementos en la que cada uno es o bien un tipo de dato u otra estructura de datos.
2. Un conjuntos de asociaciones o relaciones (estructura) que implica a los elementos combinados.

La Tabla 1.1 recoge las definiciones de dos estructuras de datos clásicas: *arrays* y registros.

**Tabla 1.1** Ejemplos de estructura de datos

| Array | Registro |
|---|---|
| Secuencias homogéneas de datos o tipos de datos conocidos como elementos. | Combinación de datos heterogéneos en una estructura única con una clave identificada. |
| Asociación de posición entre los elementos. | Ninguna asociación entre los elementos. |

La mayoría de los lenguajes de programación soporta diferentes estructuras de datos. Además, esos mismos lenguajes suelen permitir a los programadores crear sus propias nuevas estructuras de datos con el objetivo fundamental de resolver del modo más eficiente posible una aplicación.

Será necesario que la elección de una estructura de datos adecuada requiera también la posibilidad de poder realizar operaciones sobre dichas estructuras. La elección de la estructura de datos adecuada redundará en una mayor eficiencia del programa y, sobre todo, en una mejor resolución del problema en cuestión. Una elección inadecuada de la estructura de datos puede conducir a programas lentos, largos y poco eficientes.

Una solución se denomina **eficiente** si resuelve el problema dentro de las *restricciones de recursos requeridas*. Restricciones de recursos pueden ser el espacio total disponible para almacenar los datos (considerando la memoria principal independiente de las restricciones de espacio de discos, fijos, CD, DVD, *flash*…) o el tiempo permitido para ejecutar cada subtarea. Se suele decir que una solución es eficiente cuando requiere menos recursos que las alternativas conocidas.

El **costo** de una solución es la cantidad de recursos que la solución consume. Normalmente, el coste se mide en término de recursos clave, especialmente el tiempo.

### 1.2.1. Etapas en la selección de una estructura de datos

Los pasos a seguir para seleccionar una estructura de datos que resuelva un problema son [SHAFFER 97]:

1. Analizar el problema para determinar las restricciones de recursos que debe cumplir cada posible solución.
2. Determinar las operaciones básicas que se deben soportar y cuantificar las restricciones de recursos para cada una. Ejemplos de operaciones básicas son la inserción de un dato en la estructura de datos, suprimir un dato de la estructura o encontrar un dato determinado en dicha estructura.
3. Seleccionar la estructura de datos que cumple mejor los requisitos o requerimientos.

Este método de tres etapas para la selección de una estructura de datos es una aproximación centrada en los datos. Primero se diseñan los datos y las operaciones que se realizan sobre ellos, a continuación viene la representación de esos datos y, por último, viene la implementación de esa representación.

Las restricciones de recursos sobre ciertas operaciones clave, como búsqueda, inserción de registros de datos y eliminación de registros de datos, normalmente conducen el proceso de selección de las estructuras de datos.

Algunas consideraciones importantes para la elección de la estructura de datos adecuada son:

- ¿Todos los datos se insertan en la estructura de datos al principio o se entremezclan con otras operaciones?
- ¿Se pueden eliminar los datos?
- ¿Los datos se procesan en un orden bien definido o se permite el acceso aleatorio?

***RESUMEN***

Un tipo es una colección de valores. Por ejemplo, el tipo boolean consta de los valores *true* y *false*. Los enteros también forman un tipo.

Un tipo de dato es un tipo con una colección de operaciones que manipulan el tipo. Por ejemplo, una variable entera es un miembro del tipo de dato entero. La suma es un ejemplo de una operación sobre tipos de datos enteros.

Un elemento dato es una pieza de información o un registro cuyos valores se especifican a partir de un tipo. Un elemento dato se considera que es un miembro de un tipo de dato. El entero es un elemento de datos simple ya que no contiene subpartes.

Un registro de una cuenta corriente de un banco puede contener varios campos o piezas de información como nombre, número de la cuenta, saldo y dirección. Dicho registro es un dato agregado.

## 1.3. ALGORITMOS Y PROGRAMAS

Un **algoritmo** es un método, un proceso, un conjunto de instrucciones utilizadas para resolver un problema específico. Un problema puede ser resuelto mediante muchos algoritmos. Un algoritmo dado correcto, resuelve un problema definido y determinado (por ejemplo, calcula una función determinada). En este libro se explican muchos algoritmos y, para algunos problemas, se proponen diferentes algoritmos, como es el caso del problema típico de ordenación de listas.

La ventaja de conocer varias soluciones a un problema es que las diferentes soluciones pueden ser más eficientes para variaciones específicas del problema o para diferentes entradas del mismo problema. Por ejemplo, un algoritmo de ordenación puede ser el mejor para ordenar conjuntos pequeños de números, otro puede ser el mejor para ordenar conjuntos grandes de números y un tercero puede ser el mejor para ordenar cadenas de caracteres de longitud variable.

Desde un punto de vista más formal y riguroso, un algoritmo es "un conjunto ordenado de pasos o instrucciones ejecutables y no ambiguas". Obsérvese en la definición que las etapas o pasos que sigue el algoritmo deben tener una estructura bien establecida en términos del orden en que se ejecutan las etapas. Esto no significa que las etapas se deban ejecutar en secuencia: una primera etapa, después una segunda, etc. Algunos algoritmos, conocidos como *algoritmos paralelos,* por ejemplo, contienen más de una secuencia de etapas, cada una diseñada para ser ejecutada por procesadores diferentes en una máquina multiprocesador. En tales casos, los algoritmos globales no poseen un único hilo conductor de etapas que conforman el escenario de primera etapa, segunda etapa, etc. En su lugar, la estructura del algoritmo es el de múltiples hilos conductores que se bifurcan y se reconectan a medida que los diferentes procesadores ejecutan las diferentes partes de la tarea global.

Durante el diseño de un algoritmo, los detalles de un lenguaje de programación específico se pueden obviar frente a la simplicidad de una solución. Generalmente, el diseño se escribe en español (o en inglés, o en otro idioma hablado). También se utiliza un tipo de lenguaje mixto entre el español y un lenguaje de programación universal que se conoce como ***pseudocódigo*** (o *seudocódigo*).

### 1.3.1. Propiedades de los algoritmos

Un algoritmo debe cumplir diferentes propiedades[2]:

1. *Especificación precisa de la entrada.* La forma más común del algoritmo es una transformación que toma un conjunto de valores de entrada y ejecuta algunas manipulaciones para producir un conjunto de valores de salida. Un algoritmo debe dejar claros el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.
2. *Especificación precisa de cada instrucción.* Cada etapa de un algoritmo debe ser definida con precisión. Esto significa que no puede haber *ambigüedad* sobre las acciones que se deban ejecutar en cada momento.
3. *Exactitud, corrección.* Un algoritmo debe ser *exacto, correcto.* Se debe poder demostrar que el algoritmo resuelve el problema. Con frecuencia, esto se plasma en el formato de un argumento, lógico o matemático, al efecto de que ***si*** las condiciones de entrada se cumplen y se ejecutan los pasos del algoritmo, ***entonces*** se producirá la salida deseada. En otras palabras, se debe calcular la función deseada y convertir cada entrada a la salida correcta. Un algoritmo se espera que resuelva un problema.
4. *Etapas bien definidas y concretas.* Un algoritmo se compone de una serie de *etapas concretas,* lo que significa que la acción descrita por esa etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo. Por consiguiente, el algoritmo nos proporciona una "receta" para resolver el problema en etapas y tiempos concretos.
5. *Número finito de pasos.* Un algoritmo se debe componer de un número *finito* de pasos. Si la descripción del algoritmo consta de un número infinito de etapas, nunca se podrá implementar como un programa de computador. La mayoría de los lenguajes que describen algoritmos (español, inglés o pseudocógio) proporciona un método para ejecutar acciones repetidas, conocidas como iteraciones, que controlan las salidas de bucles o secuencias repetitivas.
6. *Un algoritmo debe terminar.* En otras palabras, no puede entrar en un bucle infinito.
7. *Descripción del resultado o efecto.* Por último, debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa con la producción de un valor como resultado que tenga ciertas propiedades. Con menor frecuencia, los algoritmos se ejecutan para un *efecto lateral,* como imprimir un valor en un dispositivo de salida. En cualquier caso, la salida esperada debe estar especificada completamente.

---

**Ejemplo 1.3**

*¿Es un algoritmo la instrucción siguiente?*

```
Escribir una lista de todos los enteros positivos
```

Es imposible ejecutar la instrucción anterior dado que hay infinitos enteros positivos. Por consiguiente, cualquier conjunto de instrucciones que implique esta instrucción no es un algoritmo.

---

[2] En [BUDD 1998], [JOYANES 2003], [BROOKSHEAR 2003] y [TREMBLAY 2003], además de en las referencias incluidas al final del libro en la bibliografía, puede encontrar información ampliada sobre teoría y práctica de algoritmos.

### 1.3.2. Programas

Normalmente, se considera que un programa de computadora es una representación concreta de un algoritmo en un lenguaje de programación. Naturalmente, hay muchos programas que son ejemplos del mismo algoritmo, dado que cualquier lenguaje de programación moderno se puede utilizar para implementar cualquier algoritmo (aunque algunos lenguajes de programación facilitarán su tarea al programador más que otros). Por definición un algoritmo debe proporcionar suficiente detalle para que se pueda convertir en un programa cuando se necesite.

El requisito de que un algoritmo "debe terminar" significa que no todos los programas de computadora son algoritmos. Su sistema operativo es un programa en tal sentido; sin embargo, si piensa en las diferentes tareas de un sistema operativo (cada una con sus entradas y salidas asociadas) como problemas individuales, cada una es resuelta por un algoritmo específico implementado por una parte del programa sistema operativo, cada una de las cuales termina cuando se produce su correspondiente salida.

***Para recordar***

1. Un problema es una función o asociación de entradas con salidas.
2. Un algoritmo es una receta para resolver un problema cuyas etapas son concretas y no ambiguas.
3. El algoritmo debe ser correcto y finito, y debe terminar para todas las entradas.
4. Un programa es una "ejecución" *(instanciación)* de un algoritmo en un lenguaje de programación de computadora.

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características principales:

1. Que sea fácil de entender, codificar y depurar.
2. Que consiga la mayor eficiencia para los recursos de la computadora.

Idealmente, el programa resultante debería ser el más eficiente. ¿Cómo medir la eficiencia de un algoritmo o programa? El método correspondiente se denomina *análisis de algoritmos* y permite medir la dificultad inherente a un problema. En este capítulo se desarrollará este concepto y la forma de medir la medida de la eficiencia.

## 1.4. EFICIENCIA Y EXACTITUD

De las características que antes se han analizado y deben cumplir los algoritmos, destacan dos por su importancia en el desarrollo de algoritmos y en la construcción de programas: eficiencia y exactitud, que se examinarán y utilizarán amplia y profusamente en los siguientes capítulos.

Existen numerosos enfoques a la hora de resolver un problema. ¿Cómo elegir el más adecuado entre ellos? Entre las líneas de acción fundamentales en el diseño de computadoras se suelen plantear dos objetivos (a veces conflictivos y contradictorios entre sí) [SHAFFER 1997]:

1. Diseñar un algoritmo que sea fácil de entender, codificar y depurar.
2. Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadora.

Idealmente, el programa resultante debe cumplir ambos objetivos. En estos casos, se suele decir que dicho programa es "elegante". Entre los objetivos centrales de este libro está la medida de la eficiencia de algoritmo, así como su diseño correcto o exacto.

### 1.4.1. Eficiencia de un algoritmo

Raramente existe un único algoritmo para resolver un problema determinado. Cuando se comparan dos algoritmos diferentes que resuelven el mismo problema, normalmente se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. En este sentido, lo importante es que el programador sea capaz de reconocer y elegir el algoritmo más eficiente.

Entonces, ¿qué es eficiencia? La ***eficiencia*** de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo más corto posible o utilizando la cantidad más pequeña posible de recursos físicos y que sea compatible con su exactitud o corrección. Un buen programador buscará el algoritmo más eficiente dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

¿Cómo medir la eficiencia de un algoritmo o programa informático? Uno de los métodos más sobresalientes es el ***análisis de algoritmos,*** que permite medir la dificultad inherente de un problema. Los restantes capítulos utilizan con frecuencia las técnica de análisis de algoritmos siempre que estos se diseñan. Esta característica le permitirá comparar algoritmos para la resolución de problemas en términos de eficiencia.

Aunque las máquinas actuales son capaces de ejecutar millones de instrucciones por segundo, la eficiencia permanece como un reto o preocupación a resolver. Con frecuencia, la elección entre algoritmos eficientes e ineficientes pueden mostrar la diferencia entre una solución práctica a un problema y una no práctica. En los primeros tiempos de la informática moderna (décadas de los 60 a los 80), las computadoras eran muy lentas y tenían una capacidad de memoria pequeña. Los programas tenían que ser diseñados cuidadosamente para hacer uso de los recursos escasos, como almacenamiento y tiempo de ejecución. Los programadores gastaban horas intentando recortar radicalmente segundos a los tiempos de ejecución de sus programas o intentando comprimir los programas en un pequeño espacio en memoria utilizando todo tipo de tecnologías de comprensión y reducción de tamaño. La eficiencia de un programa se medía en aquella época como un factor dependiente del binomio *espacio-tiempo*.

Hoy, la situación ha cambiado radicalmente. Los costes del hardware han caído drásticamente, mientras que los costes humanos han aumentado considerablemente. El tiempo de ejecución y el espacio de memoria ya no son factores críticos como lo fueron anteriormente. Hoy día, el esfuerzo considerable que se requería para conseguir la eficiencia máxima no es tan acusado, excepto en algunas aplicaciones como, por ejemplo, sistemas en tiempo real con factores críticos de ejecución. Pese a todo, la eficiencia sigue siendo un factor decisivo en el diseño de algoritmos y construcción posterior de programas.

Existen diferentes métodos con los que se trata de medir la eficiencia de los algoritmos; entre ellos, los que se basan en el número de operaciones que debe efectuar un algoritmo para realizar una tarea; otros métodos se centran en tratar de medir el tiempo que se emplea en llevar a cabo una determinada tarea, ya que lo importante para el usuario final es que ésta se efectúe de forma correcta y en el menor tiempo posible. Sin embargo, estos métodos presentan varias dificultades, ya que cuando se trata de generalizar la medida hecha, ésta depende de factores como la máquina en la que se efectuó, el ambiente del procesamiento y el tamaño de la muestra, entre otros factores.

*Brassard* y *Bratley* acuñaron, en 1988, el término **algoritmia** *(algorithmics)*[3], que definía como "el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes". Este estudio fue ampliado en 1997[4] con la consideración de que la determinación de la eficiencia de un algoritmo se podía expresar en el tiempo requerido para realizar la tarea en función del tamaño de la muestra e independiente del ambiente en que se efectúe.

El estudio de la eficiencia de los algoritmos se centra, fundamentalmente, en el análisis de la ejecución de bucles, ya que en el caso de funciones lineales —no contienen bucles—, la eficiencia es función del número de instrucciones que contiene. En este caso, su eficiencia depende de la velocidad de las computadoras y, generalmente, no es un factor decisivo en la eficiencia global de un programa.

Al crear programas que se ejecutan muchas veces a lo largo de su vida y/o tienen grandes cantidades de datos de entrada, las consideraciones de eficiencia, no se pueden descartar. Además, en la actualidad existen un gran número de aplicaciones informáticas que requieren características especiales de *hardware* y *software* en las cuales los criterios de eficiencia deben ser siempre tenidas en cuenta.

Por otra parte, las consideraciones espacio-tiempo se han ampliado con los nuevos avances en tecnologías de hardware de computadoras. Las consideraciones de espacio implican hoy diversos tipos de memoria: principal, *caché, flash,* archivos, discos duros USB y otros formatos especializados. Asimismo, con el uso creciente de redes de computadoras de alta velocidad, existen muchas consideraciones de eficiencia a tener en cuenta en entornos de informática o computación distribuida.

> ***Nota***
>
> La eficiencia como factor espacio-tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad de mantenimiento del programa.

## 1.4.2. Formato general de la eficiencia

En general, el formato se puede expresar mediante una función:

*f (n) = eficiencia*

Es decir, la eficiencia del algoritmo se examina como una función del número de elementos que tienen que ser procesados.

### Bucles lineales

En los bucles se repiten las sentencias del *cuerpo del bucle* un número determinado de veces, que determina la eficiencia del mismo. Normalmente, en los algoritmos los bucles son el término dominante en cuanto a la eficiencia del mismo.

---

[3] Giles Brassard y Paul Bratley. *Algorithmics. Theory and Practice*. Englewood Cliffs, N. N.: Prentice-Hall, 1988.

[4] *Fundamental of algorithmics* (Prentice-Hall, 1997). Este libro fue traducido al español, publicado también en Prentice-Hall (España) por un equipo de profesores de la Universidad Pontificia de Salamanca, dirigidos por el co-autor de este libro, el profesor Luis Joyanes.

### Ejemplo 1.4

*¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?*

```
1  i = 1
2  iterar (i <= n)
      1 código de la aplicación
      2 i = i + 1
3  fin _ iterar
```

Si `n` es un entero, por ejemplo de valor `100`, la respuesta es `100` veces. El número de iteraciones es directamente proporcional al factor del bucle, `n`. Como la eficiencia es directamente proporcional al número de iteraciones, la función que expresa la eficiencia es:

```
f(n) =  n
```

### Ejemplo 1.5

*¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?*

```
1  i = 1
2  iterar (i <= n)
      1 código de la aplicación
      2 i = i + 2
3  fin _ iterar
```

La respuesta no siempre es tan evidente como en el ejercicio anterior. Ahora el contador `i` avanza de `2` en `2`, por lo que la respuesta es `n/2`. En este caso, el factor de eficiencia es:

```
f(n) =  n / 2
```

### Bucles algorítmicos

Consideremos un bucle en el que su variable de control se multiplique o divida dentro de dicho bucle. ¿Cuántas veces se repetirá el cuerpo del bucle en los siguientes segmentos de programa?

```
1  i = 1
2  mientras (i < 1000)
      { código de la aplicación }
      i = i * 2
3  fin _ mientras
```

```
1  i = 1000
2  mientras (i >= 1)
     { código aplicación }
     i = i/2
3  fin _ mientras
```

La Tabla 1.2 contiene las diferentes iteraciones y los valores de la variable `i`.

**Tabla 1.2** Análisis de los bucles de multiplicación y división

| Bucle de multiplicar | | Bucle de dividir | |
|---|---|---|---|
| Iteración | Valor de i | Iteración | Valor de i |
| 1 | 1 | 1 | 1000 |
| 2 | 2 | 2 | 500 |

*(Continúa)*

| Bucle de multiplicar | | Bucle de dividir | |
|---|---|---|---|
| Iteración | Valor de i | Iteración | Valor de i |
| 3 | 4 | 3 | 250 |
| 4 | 8 | 4 | 125 |
| 5 | 16 | 5 | 62 |
| 6 | 32 | 6 | 31 |
| 7 | 64 | 7 | 15 |
| 8 | 128 | 8 | 7 |
| 9 | 256 | 9 | 3 |
| 10 | 512 | 10 | 1 |
| *salida* | 1024 | *salida* | 0 |

En ambos bucles se ejecutan 10 iteraciones. La razón es que, en cada iteración, el valor de `i` se dobla en el bucle de multiplicar y se divide por la mitad en el bucle de división. Por consiguiente, el número de iteraciones es una función del multiplicador o divisor, en este caso 2.

*Bucle de multiplicar* $2^{\text{iteraciones}} < 1000$
*Bucle de división* $1000 / 2^{\text{iteraciones}} >= 1$

Generalizando el análisis, se puede decir que las iteraciones de los bucles especificados se determinan por la siguiente fórmula:

$f(n) = [\log_2 n]$

### Bucles anidados

En el caso de bucles anidados (bucles que contienen otros bucles), se determinan cuántas iteraciones contiene cada bucle. El total es entonces el producto del número de iteraciones del bucle interno y el número de iteraciones del bucle externo.

```
iteraciones : iteraciones del bucle externo x iteraciones bucle interno
```

Existen tres tipos de bucles anidados: *lineal logarítmico, cuadrático dependiente* y *cuadrático* que con ejemplos y análisis similares a las anteriores nos conducen a ecuaciones de eficiencia contempladas en la Tabla 1.3.

**Tabla 1.3** Fórmulas de eficiencia

| Lineal logarítmica | Dependiente cuadrática | Cuadrática |
|---|---|---|
| $f(n) = [n\log_2 n]$ | $f(n) = \frac{n\,(n+1)}{2}$ | $f(n) = n^2$ |

## 1.4.3. Análisis de rendimiento

La medida del rendimiento de un programa se consigue mediante la complejidad del espacio y del tiempo de un programa.

La *complejidad del espacio* de un programa es la cantidad de memoria que se necesita para ejecutarlo hasta la compleción (*terminación).* El avance tecnológico proporciona hoy en día memoria abundante; por esa razón, el análisis de algoritmos se centra fundamentalmente en el tiempo de ejecución.

La *complejidad del tiempo* de un programa es la cantidad de tiempo de computadora que se necesita para ejecutarlo. Se utiliza una función, *T(n),* para representar el número de unidades de tiempo tomadas por un programa o algoritmo para cualquier entrada de tamaño *n*. Si la función *T(n)* de un programa es *T(n) = c * n,* entonces el tiempo de ejecución es linealmente proporcional al tamaño de la entrada sobre la que se ejecuta. Tal programa se dice que es de *tiempo lineal* o, simplemente *lineal.*

---

### Ejemplo 1.6

*Tiempo de ejecución lineal de una función que calcula una serie de n términos.*

```
double serie(double x, int n)
{
   double s;
   int i;
   s = 0.0;                        // tiempo t1
   for (i = 1; i <= n; i++)        // tiempo t2
   {
      s += i*x;                    // tiempo t3
   }
   return s;                       // tiempo t4
}
```

La función *T(n)* del método es:

```
T(n) = t1 + n*t2 + n*t3 + t4
```

El tiempo crece a medida que lo hace *n,* por lo que es preferible expresar el tiempo de ejecución de tal forma que indique el comportamiento que va a tener la función con respecto al valor de *n*.

---

Considerando todas las reflexiones anteriores, si *T(n)* es el tiempo de ejecución de un programa con entrada de tamaño *n*, será posible valorar *T(n)* como el número de sentencias, en nuestro caso en Java, ejecutadas por el programa, y la evaluación se podrá efectuar desde diferentes puntos de vista:

**Peor caso.** Indica el tiempo peor que se puede tener. Este análisis es perfectamente adecuado para algoritmos cuyo tiempo de respuesta es crítico; por ejemplo, para el caso del programa de control de una central nuclear. Es el que se emplea en este libro.

**Mejor caso.** Indica el tiempo mejor que podemos tener.

**Caso medio.** Se puede computar *T(n)* como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño *n.* El tiempo de ejecución medio es a veces una medida más realista de lo que el rendimiento será en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el peor caso.

## 1.5. NOTACIÓN O-GRANDE

La alta velocidad de las computadoras actuales (frecuencias del procesador de 3 GHz ya son usuales en computadoras comerciales) hace que la medida exacta de la eficiencia de un algoritmo no sea una preocupación vital en el diseño, pero sí el orden general de magnitud de la misma. Si el análisis de dos algoritmos muestra que uno ejecuta 25 iteraciones mientras otro ejecuta 40, la práctica muestra que ambos son muy rápidos; entonces, ¿cómo se valoran las diferencias? Por otra parte, si un algoritmo realiza 25 iteraciones y otro 2.500 iteraciones, entonces debemos estar preocupados por la rapidez de uno o la lentitud de otro.

Se ha comentado anteriormente que el número de sentencias ejecutadas en la función para `n` datos es función del número de elementos y se expresa por la fórmula *f(n)*. Aunque la ecuación para obtener una función puede ser compleja, el factor dominante que se debe considerar para determinar el orden de magnitud del resultado es el denominado factor de eficiencia. Por consiguiente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que determina la magnitud. Este factor se define como **"O grande",** que representa *"está en el orden de"* y se expresa como *O(n),* es decir, *"en el orden de n"*.

La notación *O* indica la cota superior del tiempo de ejecución de un algoritmo o programa. Así, en lugar de decir que un algoritmo emplea un tiempo de *4n-1* en procesar un array de longitud *n*, se dirá que emplea un tiempo *O(n)* que se lee *"O grande de n"*, o bien *"O de n"* y que informalmente significa *"algunos tiempos constantes n"*.

Con la notación O se expresa una aproximación de la relación entre el tamaño de un problema y la cantidad de proceso necesario para hacerlo. Por ejemplo, si

`f(n) = n2 -2n +3` entonces `f(n)` es `O(n2)`.

### 1.5.1. Descripción de tiempos de ejecución con la notación O

Sea *T(n)* el tiempo de ejecución de un programa, medido como una función de la entrada de tamaño *n*. Se dice que *"T(n) es O(g(n))"* si *g(n)* acota superiormente a *T(n)*. De modo más riguroso, *T(n)* es *O(g(n))* si existe un entero $n_0$ y una constante $c > 0$ tal que para todos los enteros $n \geq n_o$ se tiene que $T(n) \leq cg(n)$.

---

#### Ejemplo 1.7

*Dada la función* $f(n) = n^3 + 3n + 1$, *encontrar su "`O grande`" (complejidad asintótica).*

Para valores de $n \geq 1$ se puede demostrar que:

$$f(n) = n^3 + 3n + 1 \leq n^3 + 3n^3 + 1n^3 = 5n^3$$

Escogiendo la constante $c = 5$ y $n_0 = 1$ se satisface la desigualdad $f(n) \leq 5n^3$. Entonces se puede asegurar que:

$$f(n) = O(n^3)$$

Ahora bien, también se puede asegurar que $f(n) = O(n^4)$ y que $f(n) = O(n^5)$, y así sucesivamente con potencias mayores de n. Sin embargo, lo que realmente interesa es la cota superior más ajustada que informa de la tasa de crecimiento de la función con respecto a n.

Una función f(x) puede estar acotada superiormente por un número indefinido de funciones a partir de ciertos valores $x_0$,

$$f(x) \leq g(x) \leq h(x) \leq k(x) \ldots$$

La complejidad asintótica de la función f(x) se considera que es la cota superior mas ajustada:

$$f(x) = O(g(x))$$

***Para recordar***

La expresión la eficiencia de un algoritmo se simplifica con la función *O* grande.

Si un algoritmo es cuadrático, se dice entonces que su eficiencia es $O(n^2)$. Esta función expresa cómo crece el tiempo de proceso del algoritmo, de tal forma que una eficiencia $O(n^2)$ muestra que crece con el cuadrado del número de entradas.

## 1.5.2. Determinar la notación O

La notación *O* grande se puede obtener de f(n) utilizando los siguientes pasos:

1. En cada término, establecer el coeficiente del término en 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor:

$$\log_2 n \quad n \quad n\log_2 n \quad n^2 \quad n^3 \quad \ldots \quad n^k \quad 2^n \quad n!$$

### Ejemplo 1.8

*Calcular la función O grande de eficiencia de las siguientes funciones:*

a. $F(n) = 1/2n(n+1) = 1/2n^2 + 1/2n$

b. $F(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

*Caso a.*

1. Se eliminan todos los coeficientes y se obtiene

   $n^2 + n$

2. Se eliminan los factores más pequeños

   $n^2$

3. La notación *O* correspondiente es

   $O(f(n)) = O(n^2)$

*Caso b.*

1. Se eliminan todos los coeficientes y se obtiene

   $f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$

3. Se eliminan los factores más pequeños y el término de exponente mayor es el primero

   $n^k$

4. La notación *O* correspondiente es

   $O(f(n)) = O(n^k)$

---

## 1.5.3. Propiedades de la notación O-grande

De la definición conceptual de la notación *O* se deducen las siguientes propiedades de la notación *O*.

1. Siendo *c* una constante, *c*O(f(n)) = O(f(n))*.

   Por ejemplo, si $f(n) = 3n^4$, entonces $f(n) = 3*O(n^4) = O(n^4)$

2. *O(f(n)) + O(g(n)) = O(f(n)+g(n))*.

   Por ejemplo, si $f(n) = 2e^n$ y $g(n) = 2n^3$:

   $O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(2e^n + 2n^3) = O(e^n)$

3. *Maximo(O(f(n)),O(g(n))) = O(Maximo(f(n),g(n))*.

   Por ejemplo,

   $Maximo(O(\log(n)),O(n)) = O(Maximo(\log(n),n)) = O(n)$

4. *O(f(n)) * O(g(n)) = O(f(n)*g(n))*.

   Por ejemplo, si $f(n) = 2n^3$ y $g(n) = n$:

   $O(f(n))* O(g(x)) = O(f(x)*g(x)) = O(2n^3 *n) = O(n^4)$

5. $O(\log_a(n)) = O(\log_b(n))$ para $a, b > 1$.

   Las funciones logarítmicas son de orden logarítmico, independientemente de la base del logaritmo.

6. *O(log(n!)) = O(n*log(n))*.

7. Para $k > 1$ $O(\sum_{i=1}^{n} i^k) = O(n^{k+1})$.

8. $O(\sum_{i=2}^{n} \log(i) = O(n \log(n))$ .

## 1.6. COMPLEJIDAD DE LAS SENTENCIAS BÁSICAS DE JAVA

Al analizar la complejidad de un método no recursivo, se han de aplicar las propiedades de la notación *O* y las siguientes consideraciones relativas al orden que tienen las sentencias, fundamentalmente a las estructuras de control.

- Las sentencias de asignación, son de orden constante *O(1).*
- La complejidad de una sentencia de selección es el máximo de las complejidades del bloque `then` y del bloque `else`.
- La complejidad de una sentencia de selección múltiple `(switch)` es el máximo de las complejidades de cada uno de los bloques `case`.
- Para calcular la complejidad de un bucle, condicional o automático, se ha de estimar el número máximo de iteraciones para el *peor caso;* entonces, la complejidad del bucle es el producto del número de iteraciones por la complejidad de las sentencias que forman el cuerpo del bucle.
- La complejidad de un bloque se calcula como la suma de las complejidades de cada sentencia del bloque.
- La complejidad de la llamada a un método es de orden 1, complejidad constante. Es necesario considerar la complejidad del método invocado.

---

### Ejemplo 1.9

*Determinar la complejidad del método:*

```
double mayor(double x, double y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

El método consta de una sentencia de selección, cada alternativa tiene complejidad constante, *O(1);* entonces, la complejidad del método `mayor()`es *O(1).*

### Ejemplo 1.10

*Determinar la complejidad del siguiente método:*

```
void escribeVector(double[] x, int n)
{
   int j;
   for (j = 0; j < n; j++)
   {
      System.out.println(x[j]);
   }
}
```

El método consta de un bucle que se ejecuta *n* veces, *O(n).* El cuerpo del bucle es la llamada a `println()`, complejidad constante *O(1).* Como conclusión, la complejidad del método es `O(n)*O(1) = O(n)`.

### Ejemplo 1.11

*Determinar la complejidad del método:*

```
double suma(double[]d, int n)
{
   int k ; double s;
   k = s = 0;
   while (k < n)
   {
      s += d[k];
      if (k == 0)
         k = 2;
      else
         k = 2 * k;
   }
   return s;
}
```

`suma()` está formado por una sentencia de asignación múltiple, *O(1),* de un bucle condicional y la sentencia que devuelve control de complejidad constante, *O(1).* Por consiguiente, la complejidad del método está determinada por el bucle. Es necesario determinar el número máximo de iteraciones que va a realizar el bucle y la complejidad de su cuerpo. El número de iteraciones es igual al número de veces que el algoritmo multiplica por dos a la variable `k`. Si *t* es el número de veces que `k` se puede multiplicar hasta alcanzar el valor de `n`, que hace que termine el bucle, entonces $k = 2^t$.

$0, 2, 2^2, 2^3, \ldots, 2^t \geq n.$

Tomando logaritmos: $t \geq \log_2 n$; por consiguiente, el máximo de iteraciones es $t = \log_2 n$.

El cuerpo del bucle consta de una sentencia simple y un sentencia de selección de complejidad *O(1)*, por lo que tiene complejidad constante, *O(1).* Con todas estas consideraciones, la complejidad del bucle y del método es:

$O(\log_2 n) * O(1) = O(\log_2 n)$; *complejidad logarítmica* `O(log n)`

### Ejemplo 1.12

*Determinar la complejidad del método:*

```
void traspuesta(float[][] d, int n)
{
   int i, j;
   for (i = n - 2; i > 0; i--)
   {
      for (j = i + 1; j < n; j++)
      {
         float t;
         t = d[i][j];
         d[i][j] = d[j][i];
         d[j][i] = t;
      }
   }
}
```

El método consta de dos bucles `for` anidados. El bucle interno está formado por tres sentencias de complejidad constante, *O(1).* El bucle externo siempre realiza `n-1` veces el bucle interno. A su vez, el bucle interno hace *k* veces su bloque de sentencias, *k* varía de 1 a *n-1,* de modo que el número total de iteraciones es:

$$C = \sum_{k=1}^{n-1} k$$

El desarrollo del sumatorio produce la expresión:

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

Aplicando las propiedades de la notación *O,* se deduce que la complejidad del método es $O(n^2)$. El término que domina en el tiempo de ejecución es $n^2$, se dice que la *complejidad es cuadrática*.

## RESUMEN

Una de las herramientas típicas más utilizadas para definir algoritmos es el pseudocódigo. El *pseudocódigo* es una representación en español (o en inglés, brasilero, etc.) del código requerido para un algoritmo.

Los datos atómicos son datos simples que no se pueden descomponer. Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas. Este tipo de datos se definen por un conjunto de valores y un conjunto de operaciones que actúa sobre esos valores.

Una estructura de datos es un agregado de datos atómicos y datos compuestos en un conjunto con relaciones bien definidas.

La eficiencia de un algoritmo se define, generalmente, en función del número de elementos a procesar y el tipo de bucle que se va a utilizar. Las eficiencias de los diferentes bucles son:

| | |
|---|---|
| *Bucle lineal:* | `f(n) = n` |
| *Bucle logarítmico:* | `f(n) = log n` |
| *Bucle logarítmico lineal:* | `f(n) = n * log n` |
| *Bucle cuadrático dependiente:* | `f(n) = n(n+1)/2` |
| *Bucle cuadrático dependiente:* | $f(n) = n^2$ |
| *Bucle cúbico:* | $f(n) = n^3$ |

## EJERCICIOS

**1.1.** El siguiente algoritmo pretende calcular el cociente entero de dos enteros positivos (un dividendo y un divisor) contando el número de veces que el divisor se puede restar del dividendo antes de que se vuelva de menor valor que el divisor. Por ejemplo, 14/3 proporcionará el resultado 4 ya que 3 se puede restar de 14 cuatro veces. ¿Es correcto? Justifique su respuesta.

```
Cuenta ← 0;
Resto  ← Dividendo;
repetir
   Resto  ← Resto - Divisor
   Cuenta ← Cuenta + 1
hasta _ que (Resto < Divisor)
Cociente  ← Cuenta
```

**1.2.** El siguiente algoritmo está diseñado para calcular el producto de dos enteros negativos `x` e `y` por acumulación de la suma de copias de `y` (es decir, 4 por 5 se calcula acumulando la suma de cuatro cinco veces). ¿Es correcto? Justifique su respuesta.

```
producto ← y;
cuenta ← 1;
mientras (cuenta < x) hacer
   producto ← producto + y;
   cuenta ← cuenta + 1
fin _ mientras
```

**1.3.** Determinar la *O-grande* de los algoritmos escritos en los ejercicios 1.2 y 1.3.

**1.4.** Diseñar un algoritmo que calcule el número de veces que una cadena de caracteres aparece como una subcadena de otra cadena. Por ejemplo, `abc` aparece dos veces en la cadena `abcdabc`, y la cadena `aba` aparece dos veces en la cadena `ababa`.

**1.5.** Diseñar un algoritmo para determinar si un número *n* es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad.)

**1.6.** Determinar la *O-grande* de los algoritmos que resuelven los ejercicios 1.4 y 1.5.

**1.7.** Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura `(S = ½ Base x Altura)`.

**1.8.** Escribir un algoritmo que calcule y muestre la longitud de la circunferencia y el área de un círculo de radio dado.

**1.9.** Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como "radar".

**1.10.** Calcular la eficiencia de los siguientes algoritmos:

a.
```
i = 1
mientras (i <= n)
   j = 1
   mientras (j <= n)
      j = j * 2
   fin _ mientras
   i = i + 1
fin _ mientras
```

b.
```
i = 1
mientras (i <= n)
   j = 1
   mientras (j <= i)
      j = j + 1
   fin_mientras
   i = i + 1
fin_mientras
```

c.
```
i = 1
mientras (i <= 10)
   j = 1
   mientras (j <= 10)
      j = j + 1
   fin_mientras
   i = i + 2
fin_mientras
```