PROGRAMACIÓN

Escuela Universitaria de Informática - Curso 2001-2002

Tema 4. LA ITERACIÓN

Isabel Galiano, Francisco Marqués y Natividad Prieto

Índice General

1 La iteración como estrategia: sus fases

El factor común de los programas escritos hasta el momento es que sólo hacen lo que hacen **una** única vez; el programa `adivinanza` sólo permite *adivinar* **un** número; el programa `ValidarFecha` sólo permite *validar* **una** fecha; el programa `intercambio` sólo permite *intercambiar* **una** única vez el valor de dos variables.

Ahora bien, solucionado un problema **una** vez, código correspondiente incluido, sería deseable poder reutilizar dicha solución muchas más, tantas como fuera necesario; ¿por qué no poder jugar a *adivinar* un número hasta que nos cansemos o *ValidarFecha* tantas fechas como sea necesario? Evidentemente, la respuesta no es repetir la ejecución de `adivinar` hasta que nos cansemos o `ValidarFecha` tantas veces como sea necesario.

Para poder *repetir* de forma controlada una instrucción o un bloque de instrucciones de un programa, esto es para poder *iterar*, cualquier lenguaje de programación proporciona al menos una instrucción *iterativa* o *bucle*. Por su naturaleza, las partes principales de un bucle -o de la iteración que representa- son dos :

- *Cuerpo*: instrucción o bloque de instrucciones que se repiten;
- *Criterio de Repetición*: condición que controla la ejecución de una nueva repetición.

El tipo de bucle que ejemplifica con mayor claridad la función iterativa es el llamado bucle "`while`". En Java, su forma general es:

```
while (CriteriodeRepetición) Cuerpo
```

Nótese que `CriteriodeRepetición` y `Cuerpo` son las mismas construcciones que están presentes en la instrucción `if` que se estudió en el tema precedente.

La ejecución de la instrucción `while` procede como sigue:

1. se evalúa la condición `CriteriodeRepetición`;
2. si `CriteriodeRepetición` == `false`, el bucle termina;
3. si `CriteriodeRepetición == true`, el `Cuerpo` del bucle se ejecuta o, dicho de otra forma, *se entra en el bucle*; tras la ejecución del `Cuerpo` se vuelve al paso 1.

Al igual que las condicionales, la instrucción `while` permite alterar el flujo de un programa.

No sólo los ejemplos con los que iniciábamos este apartado hacen explícita la necesidad de *iterar*. En realidad, cualquier programa de cierta entidad contiene como mínimo un bucle, más o menos sofisticado; está demostrado que toda la computación se puede realizar sólo con iteración y con un conjunto muy pequeño de operaciones elementales. Por ejemplo, las operaciones aritméticas "*" y "/" se pueden implementar, siempre contando con la iteración, como sumas repetidas y restas repetidas, respectivamente. Aún más, las operaciones aritméticas "+" y "-" son a su vez programables con incrementos y disminuciones en 1 repetidos.

Ahora bien, toda la potencia y expresividad que proporciona la *iteración* obligan a un uso cuidadoso; piénsese, por ejemplo, que un `bucle` *incorrecto* puede serlo no sólo porque no calcule el resultado esperado sino también porque no termine (`bucle` *infinito*).

Por tanto, dada la frecuencia con la que se usan los *bucles* y el cuidado que requiere su construcción, conviene introducir una estrategia que, independientemente del lenguaje de programación que se utilice, se pueda emplear con éxito para construir cualquier tipo de bucle. Esta estrategia *Iterativa* consta de distintas fases, que se presentan a continuación y se ejemplifican para la resolución del siguiente problema:

- diséñese un programa que, sin utilizar la operación "`*`", calcule el `producto` de dos números enteros no negativos, `a` y `b`.

1.1 Descubriendo la estructura iterativa del problema.

Tras la lectura del enunciado, somos capaces de crear un esqueleto del programa Java `multiplicar` cuyos datos son `a` y `b`, de tipo entero, y cuyo resultado es `producto`, también de tipo entero. Nótese que sólo serán datos válidos aquellos que, siendo enteros, son positivos o cero. A partir de aquí, y por eso hemos hablado de un esqueleto de programa, la tarea básica que nos resta es determinar cuál es la(s) instrucción(es) a realizar para calcular `producto` a partir de `a` y `b`, sin utilizar el operador "`*`".

```
....
class multiplicar {
    public static void main(String args[])  {
        ...
        //declaración e inicialización de variables dato y resultado
        int a = Teclado.readint(), b = Teclado.readint(), producto;
        if ( a>=0 && b>=0) {

            // multiplicar a y b dejando el resultado en producto,
            // sin usar el operador "*"

        } else System.out.println("Lo siento, datos no válidos");
    }
}
```

En primer lugar, y a partir del enunciado del problema, debemos plantearnos si en `multiplicar` hay que utilizar un bucle; esto es, ¿se puede expresar `a * b` como el resultado de la repetición, controlada, de una cierta operación? En efecto, sabemos que el producto de dos números es una suma repetida:

`a * b == producto == 0 + b + ... + b` "`producto` es 0 sumado a `b`, `a` veces".

El uso de "0" en la definición de la multiplicación nos permite contemplar el caso particular `a` == 0; 0 se suma a `b` *ninguna* vez, con lo que 0 * `b` == 0.

Esta definición informal es lo suficientemente clara como para poner de manifiesto:

- que el *Cuerpo* de la iteración es una suma repetida;

- que el *Criterio de Repetición* de la iteración es que hay que seguir sumando mientras el número de repeticiones realizadas sea distinto de `a`; así, cuando el bucle termine, `producto` == `a` * `b`
- que *inicialmente*, antes de la primera repetición, `producto` vale 0.

A partir de esta defínción, podemos completar nuestro programa `multiplicar` como sigue:

```
....
class multiplicar {
    ....
    if ( a>=0 && b>=0) {
        // multiplicar a y b dejando el resultado en producto,
        // sin usar el operador "*"

        // inicialización de producto
        producto = 0;

        // forma general del while
        while (CriteriodeRepetición) Cuerpo;

        System.out.println("El producto es "+producto);
        } else System.out.println("Lo siento, datos no válidos");
    ...
```

Este esqueleto de programa incorpora ya la inicialización de `producto`. Sin embargo, su bucle es, *sólo*, la forma general del `while`. Los restantes pasos de la estrategia *Iterativa* refinarán esta primer solución hasta transformarla en un código Java totalmente operativo.

1.2 Eligiendo las variables de la iteración.

Para expresar el *Criterio de Repetición* informal podemos utilizar un contador del número de repeticiones que se van haciendo. Así, necesitamos definir una variable **cont**ador, `cont`, de tipo entero, en el esqueleto de `multiplicar`; para recalcar que `cont` está asociada al bucle, es aconsejable que su declaración se realice en el mismo bloque donde se encuentra el `while`.

Además de llevar la cuenta del número de repeticiones, también es necesario llevar la cuenta del resultado de la suma más reciente o, lo que es lo mismo, *acumular* el resultado de las sumas realizadas hasta una repetición dada. *Para ello no hace falta definir una nueva variable* puesto que esta cuenta la puede llevar la variable `producto`. Veámoslo:

- tras la *primera* repetición habremos sumado *una* vez `b` a 0, con lo que `cont`==1 y `producto` == 1 * `b` == (0 + `b`);

- tras la *segunda* repetición habremos sumado *dos* veces `b` a 0, con lo que `cont`==2 y `producto` == 2 * `b` == (0 + `b`) + (0 + `b`);
- y así sucesivamente hasta que tras la *última* repetición, en la que `cont`==`a`, `producto` == `a` * `b` == (0 + `b`) + ... + (0 + `b`) "`a` veces".

A partir de su valor inicial 0, repetición tras repetición, se acumula sobre la variable `producto` el resultado de las sumas realizadas, y sólo contendrá el resultado del programa, `a` * `b`, cuando la iteración termine.

Esta descripción de `producto`, como variable *acumulador* de las sumas realizadas hasta una repetición dada, hace explícita la estrecha relación que existe entre `cont` y `producto`:

`producto` == `cont` * `b` "`producto` es `0` sumado a `b`, `cont` veces"

Si esta relación deja de mantenerse en cualquier momento de la ejecución de la iteración diseñada, ésta será incorrecta. Por tanto, los valores que se den y el uso que se haga de ambas variables deben mantener inalterada esta relación, repetición a repetición. Dicho esto, en lo que resta nos dedicaremos a dar valores y usar dichas variables para mantener inalterada esta relación.

1.3 Valores iniciales.

Aunque sólo queda por inicializar la variable `cont`, en este apartado revisitaremos la decisión de inicializar `producto` a cero.

Con los razonamientos realizados hasta el momento y en base a las variables definidas, la única inicialización posible de `cont` es:

```
cont = 0;
```

A saber, si nuestro *Criterio de Repetición* es que hay que realizar `a` repeticiones de *Cuerpo* entonces, antes de empezar a repetir, en la repetición 0, `cont` == 0. De hecho, ésta es la única inicialización posible de `cont` si se ha inicializado `producto` a cero: la relación `producto` == `cont` * `b` se hace 0, lo que obliga a que también `cont` == 0.

1.4 Transformando el *Criterio de Repetición* de la iteración en una condición Java.

Hasta el momento, hemos utilizado *Criterio de Repetición* como la forma natural de representar el concepto de repetición controlada - i.e. se produce una repetición si y sólo si el *Criterio de Repetición* es cierto. Ahora bien, para determinar la condición Java que traduce un *Criterio de Repetición* es mejor pensar justo al revés; esto es, ¿cuándo termina el bucle?. Si sabemos qué condición se cumple cuando termina el bucle, entonces la condición que traduce a Java nuestro *Criterio de Repetición* será justo su negación (la contraria a la de terminación). A esta condición se la suele denominar *condición del bucle*.

Nuestra definición informal expresa claramente que son `a` las veces que hay que repetir la suma; si `cont` es la variable que cuenta el número de sumas, y de repeticiones,

realizadas hasta el momento, cuando `cont` == `a` ya no habrá que seguir repitiendo y la iteración terminará. En ese momento `producto` contendrá el valor `a` * `b` . De hecho, éste es el único valor posible que puede tener `producto` cuando termine la iteración porque entonces, tras la última repetición, la relación `producto` == `cont` * `b` == `a` * `b`, ya que `cont` == `a`.

Por tanto, si la iteración termina cuando `cont` == `a`, mientras `cont != a` se seguirá iterando.

Si actualizamos nuestro esqueleto de `multiplicar` tendremos:

```
....
if ( a>=0 && b>=0) {
    // multiplicar a y b dejando el resultado en producto,
    // sin usar el operador *

    // declaración de la variable contador del bucle
    int cont;

    // incialización de variables
    cont = 0;
    producto = 0;
    // En la repetición 0, producto == cont * b

    while (cont != a) Cuerpo;
    // Tras cada repetición, también la última, producto == cont * b

    System.out.println("El producto es "+producto);
} else System.out.println("Lo siento, tus datos no son válidos");
...
```

1.5 Transformando el *Cuerpo* de la iteración en un bloque Java.

Determinar las instrucciones que conforman el *Cuerpo* de nuestra iteración tiene mucho que ver con las decisiones que hemos tomado hasta el momento. La primera que se tendrá en cuenta es la *condición del bucle* elegida o, equivalentemente, la condición de terminación. Es fácil ver que, aún habiendo escogido la condición correcta del bucle, éste no terminará si en su *Cuerpo* no existe instrucción alguna que lo haga "*avanzar hacia la terminación*".

En nuestro ejemplo el bucle termina cuando `cont`==`a`; como inicialmente `cont`==0, "*avanzar hacia la terminación*" en su *Cuerpo* significará incrementar la variable `cont` cada vez que se realice una repetición:

```
cont++;
```

En segundo lugar, debemos traducir a Java el concepto de "*suma repetida*" con el que calculamos, por acumulación tras cada repetición, el valor final de `producto`; ya

hemos dicho que `producto` guarda el valor acumulado hasta el momento, por lo que la instrucción Java que estamos buscando es:

```
producto = producto + b;
```

Por tanto, el programa `multiplicar` quedará como sigue:

```
...
class multiplicar {
    ....
    if ( a>=0 && b>=0) {
         // multiplicar a y b dejando el resultado en producto,
        // sin usar el operador *
        // declaración de la variable contador del bucle
        int cont;

        // incialización de variables
        cont = 0;
        producto = 0;
        // En la repetición 0, producto == cont * b

        while (cont != a) {
            producto = producto + b;
            cont++;
        {
        // Tras cada repetición, también la última, producto == cont * b

        System.out.println("El producto es "+producto);
        } else System.out.println("Lo siento, tus datos no son válidos");
    ...
```

1.6 Conclusiones.

Emplear la *estrategia Iterativa* definida en esta sección nos permite construir cualquier tipo de bucle con cierta *garantía de éxito*. La ventaja de esta estrategia es que permite al programador empezar con una solución intuitiva e informal y pulirla hacia una solución completa en Java. Por otra parte, el enfoque seguido es capaz de suplir posibles deficiencias en nuestra intuición. Por ejemplo, a modo de prueba, se plantean las siguientes cuestiones sobre el ejemplo resuelto:

1. indíquese cómo variarían nuestra estrategia y programa si la inicialización de `cont` hubiera sido a uno en vez de a cero;

2. indíquese qué ocurriría si la condición del bucle, `cont == a`, se substituyera por `cont < a`.

Ahora bien, un sólo ejemplo de aplicación no permite poner de manifiesto todos los detalles y potencia que la estrategia Iterativa tiene, pudiendo llegar a sesgar la aplicación que hagamos de ella en adelante. Por otra parte Java proporciona tres tipos de bucles para la iteración, de los que sólo hemos utilizado uno, el `while`. Por tanto, sería conveniente disponer de un esquema de aplicación de nuestra estrategia para cada uno de ellos y que dicho esquema se instanciara para varios ejemplos significativos. Así, y dentro de lo posible, conseguiríamos no sólo una visión más general de la propia estrategia sino también *ajustar* su traducción a cada instrucción iterativa de Java. En la siguiente sección revisitaremos la ya introducida instrucción `while` en Java.

2 La instrucción de repetición `while`

2.1 Esquema *Iterativo* para el bucle `while`.

La forma general, la sintaxis, de la sentencia `while` Java es:

```
while (CriteriodeRepetición) Cuerpo
```

donde

- `CriteriodeRepetición` es una expresión de valor booleano. También se le denomina *condición del bucle*;
- `Cuerpo` es la instrucción o bloque que debe repetirse.

Esta sintaxis no pone de manifiesto que, en general, una iteración tiene una fase de inicialización o *repetición cero*; tampoco dice nada acerca de cómo *garantizar la terminación* o *avanzar hacia ella*. Es la estrategia *Iterativa* la que lo hace, dictando el siguiente *esquema Iterativo*:

```
ValoresIniciales;
while (!CriteriodeTerminación){
    ActualizarResultado;
    Avanzar_hacia_la_Terminación;}
// el bucle finaliza cuando CriteriodeTerminación==true
ValoresFinales;
```

Este esquema no es más que un bucle `while` de Java al que se han incorporado las fases de la estrategia *Iterativa*.

En lo que resta, se analizará el esquema presentado aplicándolo a algunos ejemplos significativos. Comenzaremos por el ya conocido, y resuelto, problema de multiplicar dos números sin utilizar la operación "*", con lo que tendríamos:

```
int cont = 0;
producto = 0;
while (cont != a){
```

```
            producto = producto + b;
            cont = cont + 1;}
        // el bucle finaliza cuando cont==a
```

En esta instanciación no aparece instrucción alguna que se corresponda con el item `ValoresFinales` del esquema. El motivo es que en este ejemplo, al terminar el bucle, `producto` no sólo contiene el resultado del bucle sino también el resultado del programa. Por tanto, en este caso `ValoresFinales` es la instrucción vacía, que no hace falta poner.

Como regla general, siempre que el resultado de la iteración **no** coincida con el resultado del programa, `ValoresFinales` es la instrucción o bloque que permite calcular a partir del resultado del bucle el resultado del programa. Si sabemos perfectamente cómo termina el bucle, y nuestra estrategia nos lo permite, también sabremos cuál es el resultado del bucle y cómo transformarlo en el resultado del programa.

Un problema sencillo en el que sí aparece `ValoresFinales` es el siguiente: calcular la suma de los `n` primeros números naturales, dónde `n` es un número natural, y obtener su media aritmética. Se deja su resolución propuesta como ejercicio. En el próximo tema, cuando se planteen los denominados *problemas de Búsqueda* aparecerán muchos más.

2.2 Ejemplos de utilización de la instrucción while

En este apartado se presenta la resolución iterativa de diversos problemas numéricos. Todos ellos han sido resueltos aplicando la estrategia y esquema *Iterativos*; y también, en casi todos ellos, aparecen aspectos nuevos de la estrategia iterativa. La resolución que se da es código Java *con profusión de comentarios*, que son los que marcan las fases de la estrategia iterativa seguida para obtener el código.

1. Diséñese un programa que, sin utilizar las operaciones "/" y "%", obtenga el `cociente` y `resto` de la división de dos números enteros no negativos, `dividendo` y `divisor`.

```
int dividendo, divisor, cociente, resto;
if ( dividendo>=0 && divisor>0) {
    // la división se puede expresar como una resta repetida;
    // dividendo - cociente * divisor == resto
    // donde cociente == número de repeticiones de la resta
    // y     resto    == minuendo de la última resta.

    int minuendo = dividendo;
    cociente = 0;
    resto = dividendo;
    // casos en los que dividendo < divisor y, en particular,
    // dividendo == 0. Alternativamente, en la repetición 0,
    //dividendo == divisor * cociente + resto y 0 <=minuendo<divisor

    while (minuendo >= divisor){
```

```
            cociente++;
            minuendo = minuendo - divisor;}
        //dividendo == divisor * cociente + minuendo y 0<=minuendo<divisor

        resto = minuendo;
    } else ....
```

2. Diséñese un programa que calcule el máximo común divisor, `mcd`, de dos números enteros positivos `a` y `b`.

```
int a, b, mcd;
if ( a>0 && b>0) {
    // el mcd de dos número a y b es el mayor
    // de los divisores comunes de a y b. Por tanto,
    // si a == b, mcd == a == b
    // si a > b, mcd de a y b == mcd de (a-b) y b
    // si b > a, mcd de a y b == mcd de   a   y (b-a)

    // >Por qué no inicializamos mcd?

    while (a != b)
        if (a>b) a=a-b; else b=b-a;
    //Tras cada repetición, también la última,
    // si a == b, mcd == a == b
    // si a > b, mcd de a y b == mcd de (a-b) y b
    // si b > a, mcd de a y b == mcd de   a   y (b-a)

    mcd = a;
} else ....
```

Una traza de este segmento de código para los valores `a` == 65 y `b` == 39 sería:

a	b	|a-b|
65	39	26
26	39	13
26	13	13
13	13	0

3. Multiplicación *a la rusa* de dos enteros positivos. Para utilizar el método de multiplicación *a la rusa* sólo son necesarias las operaciones de multiplicar y dividir por 2. El método es el siguiente: el multiplicando se divide por dos y el multiplicador se dobla, así hasta que el multiplicador se iguala a uno. Seguidamente se suman los valores del multiplicador siempre que correspondan a un multiplicando impar.

   A continuación se ejemplifica el método para multiplicar 981 por 1234.

multiplicando	multiplicador	producto
981	1234	1234
490	2468	
245	4936	4936
122	9872	
61	19744	19744
30	39488	
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808
		+ 1210554

El algoritmo sería:

```
int multiplicando, multiplicador, producto;
// inicialmente multiplicando == A y multiplicador == B

if ( multiplicando>0 && multiplicador>0) {
    // para todo natural n se cumple que n  = 2 * (n / 2) + (n % 2)

    producto = 0;
    // En la repetición 0,
    // A * B == multiplicando * multiplicador + producto

    while (multiplicando != 1) {
        if (multiplicando % 2)!=0 producto+=multiplicador;
        multiplicando=multiplicando/2;
        multiplicador=multiplicador*2;
    {
    //Tras cada repetición, en particular la última:
    //A * B == multiplicando * multiplicador + producto

    //al finalizar el bucle multiplicando == 1, que es impar
    //Además: A * B == 1 * multiplicador + producto, de donde:
    producto+=multiplicador;
} else ....
```

En la tabla siguiente se describe el funcionamiento del bloque para los valores `multiplicando` == 87 y `multiplicador` == 25.

multiplicando	multiplicador	producto
87	25	0
43	50	25
21	100	75
10	200	175
5	400	175
2	800	575
1	1600	**575+1600=2175**

3 La instrucción de repetición `for`

En este apartado se estudia un caso particular del esquema *Iterativo* para la instrucción `while`, el denominado *bucle de conteo*. Su uso es tan frecuente que la mayoría de los lenguajes de programación proporcionan, como alternativa a la instrucción `while`, una instrucción de repetición específica para este tipo de bucles: la instrucción `for`.

3.1 Bucles de conteo: la alternativa `for`.

En algunas ocasiones, el número de repeticiones a realizar para conseguir el objetivo de la iteración *es conocido a-priori* o, lo que es lo mismo, el *Criterio de Repetición* de la iteración *viene dado*. Por ejemplo, para multiplicar dos enteros no negativos, `a` y `b`, sin usar "*", el número de veces que se ha de repetir la operación (0 + `b`) es `a`), un *dato* del problema, y por tanto, conocido *a-priori*; para calcular la media aritmética de los `n` primeros naturales, `n` es el número de repeticiones y el *dato* del problema; para elevar una `base` a un `exponente`, `exponente` es el número de repeticiones y un *dato* del problema; etc.

Al tipo de bucle asociado a este caso particular de iteración se le denomina *bucle de conteo*, ya que alcanza su objetivo tras *contar* un número dado de repeticiones. La incorporación de este conocimiento "extra" en el esquema *Iterativo* `while` conduce a una versión simplificada de éste, el esquema *Iterativo de conteo*. A saber:

- sea VALOR_FINAL el número total de repeticiones a realizar, *conocido a-priori*;
- sea `cont` la variable **cont**ador del número de repeticiones **ya** realizadas;

entonces, el bucle termina cuando `cont` == VALOR_FINAL. Por lo tanto, en el esquema:

- `CriteriodeRepetición` es la condición (`cont` != VALOR_FINAL);
- `ValoresIniciales` consta, al menos, de dos instrucciones: `cont = VALOR_INICIAL;` y, en función de `VALOR_INICIAL`, la inicialización de la variable resultado;
- `Avanzar_hacia_la_Terminación` es incrementar el valor de `cont`.

De donde el esquema *Iterativo de conteo* para la instrucción `while` es:

```
// El resultado de la iteración se obtiene tras
// repetir VALOR_FINAL veces una cierta operación

    int cont = VALOR_INICIAL;
    InicializaResultado;

    while ( cont != VALOR_FINAL)
    {
        ActualizaResultado;
        Incrementa_cont;
    }

    ValoresFinales;
```

Conviene señalar ahora que el esquema *Iterativo de conteo* presentado sigue una estrategia **ascendente**; i.e., si `cont` representa el número de repeticiones **ya** realizadas, VALOR_FINAL se alcanza incrementando el valor de `cont` en cada repetición. Para obtener el esquema **descendente** alternativo, lo que se deja propuesto como ejercicio, basta con definir `cont` como la variable que representa el número de repeticiones **que restan** por realizar.

En el siguiente ejemplo se aplica el esquema *Iterativo de conteo* al problema de elevar una `base` a un `exponente`, ambos enteros positivos.

```
//  Para obtener en potencia base ** exponente
//  ha de repetirse "exponente" veces el producto (1 * base)

    int cont = 1;
    potencia = base;

    while (cont != exponente)
    {
        potencia = potencia * base;
        cont++;
    }
```

La inicialización realizada permite tratar el caso particular $\texttt{base}^{\mathbf{1}} ==$ base. Pero entonces, como el producto (1* `base`) se ha realizado *una* vez, `cont` se inicializa a 1. Así, el bucle se repetirá siempre `exponente` - 1 veces. Una solución alternativa se obtiene tan sólo modificando como sigue la inicialización del bucle:

```
    int cont = 0;
    // potencia == base ** 0 == 1
    potencia = 1;
```

En este caso, el producto (1* `base`) se realiza *cero* veces, por lo que el bucle se repetirá siempre `exponente` veces.

Aunque hasta el momento se ha utilizado la instrucción `while`, en la práctica un *bucle de conteo* se expresa con la instrucción de repetición `for`. Independientemente de cuál se utilize más, `for` y `while` son, tan sólo, dos alternativas posibles para expresar una misma estrategia. Para recalcar este hecho, introduciremos la instrucción `for` de Java por comparación de su esquema *Iterativo de conteo* con el ya estudiado para `while`.

```
// Esquema Iterativo de conteo con "while"

    int cont = VALOR_INICIAL;
    InicializaResultado

    while (cont != VALOR_FINAL)
    {
        ActualizaResultado;
        Incrementa_cont;
    }

    ValoresFinales;

// Esquema Iterativo de conteo con "for"

    int cont;
    InicializaResultado

    for (cont = VALOR_INICIAL; cont != VALOR_FINAL; Incrementa_cont)
        ActualizaResultado;

    ValoresFinales;
```

La comparación de estos dos esquemas pone de manifiesto las características de la instrucción `for`. A saber,

- la sintaxis del `for` obliga a escribir dentro del mismo paréntesis, separados por punto y coma, la inicialización de la variable `cont`, la condición del bucle y el incremento de `cont`. Por tanto, el `Cuerpo` del `for` se reduce a `ActualizaResultado`.
- para que los esquemas `while` y `for` sean **equivalentes**, la ejecución de una instrucción `for` debe proceder como sigue:
  1. se inicializa `cont`;
  2. se evalúa la condición del bucle, `cont != VALOR_FINAL`; si el resultado es cierto, se *entra en el bucle* y, sino, el bucle termina;
  3. si se ha *entrado en el bucle*, en cada repetición, `Incrementa_cont` se ejecuta *después de* `ActualizaResultado` **y** *antes de* volver a evaluar la condición del bucle (paso 2); aunque sintácticamente aparezcan en el orden contrario al de ejecución.

Los siguientes ejemplos sirven para incidir en el modo de funcionamiento de un bucle *de conteo* `for`; para su desarrollo se ha utilizado el correspondiente esquema *Iterativo de conteo*.

- La siguiente instrucción de repetición escribe cinco filas de estrellas por pantalla:

```
for (int cont = 0; cont < 5; cont++)
     System.out.println("******");
```

Veamos cómo funciona este bucle: `cont` empieza valiendo 0. Se comprueba que el valor de `cont` es menor que 5, se imprime una fila de estrellas y se incrementa `cont` en 1. Y así hasta la última repetición, a la que se entra con `cont` == 4, y de la que se sale con `cont` valiendo 5. Tras esta repetición, `cont < 5` se evalúa a falso con lo que termina el bucle y el control pasa a la siguiente instrucción del programa. Así pues se imprimirán cinco filas de estrellas para valores de `cont` desde 0 hasta 4.

En la siguiente tabla se describe el funcionamiento de este bucle:

cont	Condición	Salida
0	cont<5?	******
1	cont<5?	******
2	cont<5?	******
3	cont<5?	******
4	cont<5?	******
5	cont<5?	

- La siguiente instrucción `for` **anidada**[1]:

```
for (int i=0; i<5; i++)
{
   for (int j=0; j<i+1; j++) System.out.print(''*'');
   System.out.println();
}
```

tiene como efecto mostrar por pantalla:

```
*
**
***
****
*****
```

[1]Una instrucción de repetición anidada es aquella cuyo `Cuerpo` es, a su vez, una instrucción de repetición.

Veamos cómo funciona este bucle anidado:

- la variable contador del bucle más externo `i`, toma valores desde 0 hasta 5; para cada uno de los valores de `i` desde 0 hasta 4 se realiza el bucle for interno y se escribe un cambio de línea;
- en cada iteración del bucle externo, la variable contador del bucle interno `j`, toma valores entre 0 e `i` + 1; para cada valor de `j` entre 0 e `i` escribe un asterisco;
- por lo tanto, globalmente, en la iteración en la que `i` == 0, se escribe un asterisco (para `j` == 0) y se cambia de linea; cuando `i` == 1 se escriben dos asteriscos (para `j`== 0 y `j` == 1) y se cambia de linea; cuando `i` == 2 se escriben tres asteriscos (para `j` == 0, `j` == 1 y `j` == 2) y se cambia de linea; cuando `i` == 3 se escriben cuatro asteriscos (para `j` == 0, `j` == 1, `j` == 2 y `j` == 3) y se cambia de linea; para finalizar, cuando `i` == 4 se escriben cinco asteriscos (para `j` == 0, `j` == 1, `j` == 2, `j` == 3 y `j` == 4) y se cambia de linea.

En la siguiente tabla se describe el comportamiento de estos bucles anidados:

i	j	Condición	Salida	i	j	Condición	Salida
0	0	j<1?	*	3	0	j<4?	*
	1	j<1?	nl		1	j<4?	*
1		i<5?			2	j<4?	*
1	0	j<2?	*		3	j<4?	*
	1	j<2?	*		4	j<4?	nl
	2	j<2?	nl	4		i<5?	
2		i<5?		4	0	j<5?	*
2	0	j<3?	*		1	j<5?	*
	1	j<3?	*		2	j<5?	*
	2	j<3?	*		3	j<5?	*
	3	j<3?	nl		4	j<5?	*
3		i<5?			5	j<5?	nl
				5		i<5?	

3.2 Ejemplos de utilización de la instrucción `for`.

1. Cálculo de la suma de los `n` primeros números naturales, dónde `n` es un número natural.

```
{
   int suma=0;

   for (int i=1; i<=n; i++) suma+=i;

   System.out.print("La suma de los "+n+" primeros naturales es: ");
   System.out.println(suma);
}
```

En la tabla siguiente se describe el funcionamiento del bloque para `n` == 5:

i	1	2	3	4	5	6
suma	0	1	3	6	10	**15**

2. Presentación de una tabla con los cuadrados de los 100 primeros números naturales.

```
{
    System.out.println("Número -- Cuadrado");
    System.out.println("------    --------");

    for (int i=1; i<=100; i++) System.out.println(i+"--"+i*i);

    System.out.println("-----------------");
}
```

3. Cálculo del factorial de un número `n`.

```
 {
    int fact=1;

    for (int i=2; i<=n; i++) fact*=i;

    System.out.println("Factorial de " + n +"=" + fact);
 }
```

En la tabla siguiente se describe el funcionamiento del bloque para `n` == 5

fact	1	2	6	24	**120**
i	2	3	4	5	6

3.3 Sintaxis de la instrucción `for`.

Aunque la instrucción `for` de Java se utiliza típicamente como *bucle de conteo*, en su forma general es una alternativa a la instrucción `while` para **cualquier** tipo de iteración.

Reproduciendo la comparación de esquemas presentada en la sección anterior, se puede deducir fácilmente el siguiente esquema *Iterativo* para la instrucción `for`:

```
for (ValoresIniciales; !CriteriodeTerminación; Avanzar_hacia_Terminación)
   ActualizarResultado;

ValoresFinales;
```

A continuación se presenta la solución `for` para dos problemas ya resueltos con `while`; con ello se pretende recalcar que los esquemas *Iterativos* `for` y `while` son equivalentes, dos alternativas posibles para expresar la misma estrategia *Iterativa*.

- Esquema Iterativo `for` para la multiplicación, `producto`, de dos números enteros no negativos, `a` y `b`, sin utilizar la operación "`*`":

```
{
    int cont;
    for (cont = 0, producto = 0; cont != a; cont++)
        producto = producto + b;
}
```

  Como muestra este ejemplo, el bucle `for` permite varias inicializaciones, no sólo la del contador, siempre que *se separen con comas*.

- Esquema Iterativo `for` para la Multiplicación *a la rusa*, `producto`, de dos enteros positivos, `a` y `b`.

```
{   int cont;
    for (cont = a, producto = 0; cont != 1; cont /= 2, b *= 2)
        if (cont % 2) != 0 producto += b;
    producto+= b;
}
```

  Como muestra este ejemplo, `Avanzar_hacia_la_Terminación` no tiene por qué ser una simple actualización del contador; también puede ser una secuencia de instrucciones *separadas por comas*.

4 La instrucción `do ...while`

La estructura `do ...while` tiene la siguiente sintaxis:

```
ValoresIniciales;

do
{
    ActualizarResultado;
    Avanzar_hacia_la_Terminación;

} while (!CriteriodeTerminación)

// el bucle finaliza cuando CriteriodeTerminación==true
ValoresFinales;
```

La única diferencia con la estructura `while` es que la condición se evalúa después de la ejecución del bucle; es por esto que el `Cuerpo` se realiza *por lo menos una vez*. A diferencia del `while` las llaves son obligatorias.

5 Introducción al coste: conteo de instrucciones

Habitualmente se dispone de varios programas que resuelven un mismo problema; por ejemplo, `multiplicar` y `MultiplicaralaRusa` son dos programas que sirven para multiplicar dos números sin utilizar la operación "*". Para decidir cuál es *el mejor*, un criterio objetivo de comparación es el de ***eficiencia***: el programa más eficiente, el mejor, será aquel *que menos recursos requiera para su ejecución.* Visto que los recursos básicos de un ordenador son la memoria (RAM) y el tiempo de CPU, la *eficiencia* de un programa se expresa en términos de:

- su **coste** *espacial* o medida del espacio que ocupa en memoria a lo largo de su ejecución; sería la suma de los tamaños de todas las variables que implícita o explícitamente se utilizan.
- su **coste** *temporal* o una medida del tiempo empleado por éste para ejecutarse y dar un resultado a partir de los datos de entrada.

A partir de estas definiciones, los costes de un programa concreto dependen de dos tipos de factores. A saber,

1. Factores *propios* del programa utilizado, como son su estrategia de resolución o los tipos de datos que emplea.
2. Factores *que dependen del entorno de programación* donde se vaya a ejecutar el programa, como son el tipo de computador, el lenguaje de programación utilizado, el compilador que se utiliza, la carga del sistema, etc.

Por tanto, se pueden seguir dos aproximaciones para establecer el coste de un programa dado:

1. *Análisis teórico o "a priori"*: cálculo del coste en función de los factores *propios* del programa, y por lo tanto, *independiente del entorno de programación.*
2. *Análisis experimental o "a posteriori"*: medida del tiempo, en segundos, y memoria, en bytes, empleados en la ejecución del programa:
   - en un *entorno de programación* determinado y
   - para un conjunto de datos de entrada adecuado.

Es importante recalcar que ambos tipos de análisis son complementarios, y no excluyentes; realizar un análisis *teórico* de su coste evita en ocasiones implementaciones tan laboriosas como inútiles. Más aún, la eficiencia es un criterio a incorporar en la estrategia de diseño de un programa; independientemente del entorno donde se ejecute, un programa debe ser eficiente.

En lo que resta nos centraremos exclusivamente en el estudio del **coste** *temporal a-priori* de los programas, dejando el estudio de la complejidad *espacial* para asignaturas sucesivas y el análisis *a-posteriori* para las prácticas en el laboratorio.

5.1 El coste temporal de los programas.

El coste temporal *a-priori* de un programa es una medida del tiempo que éste emplea para ejecutarse, *independiente del entorno de programación* donde lo haga. Esta definición plantea una pregunta obvia: ¿cómo medir el tiempo que el programa tarda en ejecutarse si éste ¡**no** se está ejecutando en máquina alguna!? La respuesta es que el coste temporal *a-priori* de un programa no es un valor, sino una función matemática[2] que expresa el tiempo que tarda en ejecutarse el programa en función de los parámetros propios de éste. Ahora bien, queda por resolver en qué unidades se expresa esta función y cuáles son sus parámetros.

El desarrollo del siguiente ejemplo nos servirá para resolver todas estas cuestiones e introducir una metodología efectiva de cálculo del coste temporal *a-priori*.

Ejemplo: para obtener el `cuadrado` de un número `num` se dispone de tres algoritmos diferentes, que figuran a continuación. ¿Cuál de ellos requerirá un tiempo menor de ejecución?

- ```
  // programa A1:
  cuadrado=num*num;
  ```

- ```
  // programa A2:
  cuadrado=0;
  for (int i=0; i<num; i++) cuadrado=cuadrado+num;
  ```

- ```
  // programa A3:
  cuadrado=0;
  for (int i=0; i<num; i++)
     for (int j=0; j<num; j++) cuadrado++;
  ```

La repuesta que nos dicta la intuición es que el programa más rápido es el A1, ya que es el que menos operaciones realiza para calcular el cuadrado de `num`. Refinando este razonamiento, si se define el coste temporal de un programa como la suma de los costes de las operaciones *elementales* que implica, tenemos:

- *Coste de A1:* $T_{A1} = t_a + t_{op}$

  dónde t_a es el coste de la operación de asignación y t_{op} es el coste de una operación aritmética, en concreto "*".

- *Coste de A2:* $T_{A2} = t_a + t_a + (num + 1) * t_e + num * t_a + 2 * num * t_{op}$

  dónde t_e es el coste de evaluar una condición y t_{op} es el coste de una operación aritmética "+" ó "++".

- *Coste de A3:* $T_{A3} = t_a + t_a + (num + 1) * t_e + num * t_a + num * (num + 1) * t_e + num^2 * t_a + 2 * num^2 * t_o p + num * t_o p$

[2] A esta función se le denomina **complejidad**. La sutil diferencia entre los términos **complejidad** y **coste** no es habitual en la práctica, donde uno se utiliza por otro sin más.
  ```

Comparar los costes de los tres programas utilizando las funciones T_{A1}, T_{A2} y T_{A3} resulta bastante difícil, ya que los tiempos de las operaciones *elementales* pueden variar significativamente en función del entorno de programación utilizado. Por otra parte, el cálculo de costes así planteado requiere un considerable esfuerzo de conteo.

5.2 La complejidad temporal definida por conteo de pasos.

Una primera simplificación de las funciones de coste consiste en independizarlas de los tiempos de ejecución de las operaciones *elementales*. Para ello, realizaremos las siguientes consideraciones:

1. Las operaciones elementales se ejecutan en tiempo constante, independientemente del tipo de operandos sobre los que se aplican. Por tanto, si $t_a = k_1$, $t_{op} = k_2$ y $t_e = k_3$, los costes de los programas A1, A2 y A3 serían:
   - $T_{A1} = k_1 + k_2$
   - $T_{A2} = num * (k_1 + 2 * k_2 + k_3) + 2 * k_1 + k_3$
   - $T_{A3} = num^2 * (k_1 + 2 * k_2 + k_3) + num(k_1 + k_2 + k_3) + 2k_1 + k_3$
2. *Cualquier* combinación de operaciones elementales es también una operación elemental, nueva, llamada ***paso de programa***, y por tanto se ejecutará en tiempo constante; de donde, un paso de programa se puede considerar también como unidad de tiempo. Si aplicamos esta definición a los costes anteriores tendríamos:
   - *Coste del programa A1:* $T_{A1} = 1$ paso
   - *Coste del programa A2:* $T_{A2} = num + 2$ pasos
   - *Coste del programa A3:* $T_{A3} = num^2 + num + 2$ pasos

Considerar un paso de programa como una unidad de tiempo permite definir la función de coste de un programa como el número de pasos de programa que realiza y, por tanto, hacerla independiente del coste de las operaciones elementales.

La decisión sobre *qué es un paso* en un determinado programa no es única; así, distintas decisiones conducen a funciones de coste diferentes, *pero sólo en constantes multiplicativas o aditivas*. Ahora bien, como veremos más adelante, estas diferencias no son significativas a la hora de comparar programas utilizando un criterio de eficiencia.

5.3 El coste como una función de la talla del problema.

Las funciones de coste T_{A1}, T_{A2} y T_{A3} expresan *en función de* `num`, *el dato del problema*, cuántos pasos de programa hay que realizar para obtener su `cuadrado`. Así, si se utiliza el programa A1, el número de pasos a realizar es siempre **1**, independientemente del valor que tenga `num`; i.e., tanto si `num` vale **1** como si vale **1000** ó **100000**, A1 realizará en un tiempo constante el cálculo de `cuadrado`. Sin embargo, si se emplean A2 ó A3, el número de pasos depende de `num`, de forma lineal para A2, $T_{A2} = num + 2$ pasos, y cuadrática para A3, $T_{A3} = num^2 + num + 2$ pasos; i.e., para valores de `num` **1**, **1000** y

100000, A1 tardará, respectivamente, (**1** + 2), (**1000** + 2) y (**100000** + 2) pasos de programa en calcular `cuadrado`, tres tiempos distintos pero que dependen de la misma forma de `num`: linealmente. En el caso de A3 ocurre lo mismo, pero con el cuadrado de `num`: *aproximadamente*, para `num` == **1** tardará $\mathbf{1}^2$ pasos, para `num` == **1000** tardará $\mathbf{1000}^2$ y para `num` == **100000** tardará ¡$\mathbf{100000}^2$!

En general, como era de esperar, el coste temporal (espacial) de un programa es una función que depende de la cantidad, o magnitud, de los datos que tiene que procesar; éste es uno de los parámetros *propios* del programa a los que se aludía al definir el análisis "a priori" del coste. Ahora bien, más que del programa, la cantidad, o magnitud, de datos a procesar es una característica del problema que se quiere resolver; será la estrategia que se elija para resolverlo la que determine el tipo de dependencia - constante, lineal, cuadrática, etc. Precisamente:

- Se define *talla o tamaño de un problema* como el dato(s) del problema que representa(n) una medida de la dificultad de su resolución. Por ejemplo, `num` es la talla del problema en los programas A1, A2 y A3; la talla de multiplicar dos números sin usar "*" es el valor que representa la variable `a` en el programa `Multiplicar`, y la variable `multiplicando` en el programa `MultiplicaralaRusa`.

- Se define el coste temporal (espacial) de un programa como la función no decreciente de la cantidad de tiempo (espacio) necesario (a) que éste necesita para ejecutarse en función de la talla del problema.

A partir de ahora, notaremos el coste temporal de un algoritmo `A` como $T_A(talla)$; por ejemplo, el coste del algoritmo A2 se expresará como $T_{A2}(num)$.

La siguiente cuestión, que se deja por resolver, ayuda a clarificar el concepto de *talla del problema*:

- considérese el problema, propuesto en los ejercicios del tema 3, de adivinar un número, `premio`, de un intervalo dado, [`inicio`,`final`], a base de preguntas. ¿Cuál es la talla de este problema?

5.4 Comparación de los costes de los algoritmos

A partir de la definición de coste, comparar costes es comparar funciones (no decrecientes) de las que nos interesa sólo su *tasa de crecimiento*; en otras palabras, para comparar costes lo que sirve, más que la función de coste, es el *tipo* de dependencia que ésta muestra con la talla del problema. Por ejemplo, si $T_A(talla)$ es un polinomio, entonces el monomio de mayor grado del mismo es el que da el aspecto de la curva de crecimiento; así, para comparar los costes de los algoritmos A1, A2 y A3, *aproximamos* su función de coste a, respectivamente, una función *constante*, una función *lineal*, num, y una función *cuadrática*, num^2; evidentemente, la función que crece más rápidamente, a partir de un cierto valor de la talla, es la cuadrática y, por lo tanto, el programa A3 es el más lento[3]

[3]En la sección anterior se ha ilustrado este crecimiento para valores concretos de `num`.

En la siguiente tabla se muestra en orden creciente de *tasa de crecimiento* distintas funciones que describen comúnmente el tiempo de ejecución de los programas.

Función	Nombre
c	constante
$logn$	logarítmica
log^2n	logarítmica al cuadrado
n	lineal
$nlogn$	nlogn
n^2	cuadrática
n^3	cúbica
2^n	exponencial

6 Ejercicios

1. Escríbase un programa Java que obtenga en `potencia` el resultado de elevar un número entero no negativo, `base`, a otro también no negativo, `exponente`.

2. Escríbase un programa Java que, utilizando tan sólo incrementos y disminuciones en uno, obtenga en `resta` la sustracción de dos números enteros no negativos, `minuendo` y `sustraendo`.

3. Describe qué hace el siguiente segmento de código en Java. Supongamos que `n` es una variable de tipo `int` que ya tiene asignado un determinado valor.

```
{    double resultado;
     int i;
     if n<0 i=-n; else i=n;
     resultado=0.0;
     while (i>=1)
     {   resultado+=(1/i);
         i--;
     }
}
```

4. Escribir un programa en Java que calcule el cuadrado del número que se introduce como dato utilizando únicamente sumas y restas.

5. Escribir un programa en Java para calcular el término `n`-ésimo de la serie:

$$\begin{aligned} a_n &= 3 * a_{n-1} + 2 \\ a_0 &= 1 \end{aligned}$$

6. Escribir un programa en Java para calcular el término `n`-ésimo de la serie de Fibonacci:

$$\begin{aligned} a_n &= a_{n-1} + a_{n-2} \\ a_1 &= 1 \\ a_0 &= 0 \end{aligned}$$

7. Escribir un programa en Java que muestre por pantalla el siguiente dibujo, escribiendo los caracteres blanco y asterisco.

```
      **
    ****
  ******
********
**********
```

8. Escribir un programa en Java que se comporte como un reloj digital, mostrando en pantalla el siguiente mensaje:

```
dia   hora   minutos   segundos
```

El programa pedirá al usuario que introduzca el día de la semana y la hora, por ejemplo:

```
Sábado  11      59         58
```

Si los datos son correctos, el programa mostrará por pantalla el día y la hora correcta (hora, minutos y segundos) para los 100 segundos siguientes. Para el ejemplo:

```
Sábado 11 59 59
Domingo 12 0 0
Domingo 12 0 1
...
Domingo 12 1 38
```