Introducción a la Computación
Algoritmos

Estos apuntes no son parte de la bibliografa de la materia. Solo son guas que utilizo para las clases teoricas. Las ponemos a disposicion de todos a pedido de algunos alumnos, pero son versiones borradores que pueden contener algunos errores.

1 Definición

Un algoritmo es un secuencia de pasos que termina en un tiempo finito. Deben estar formulados en términos de pasos sencillos que sean:

- precisos: indicar el orden de ejecución de cada paso
- bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado
- finitos: el algoritmo tiene que tener un número determinado de pasos

Los métodos enseñados en la primaria para multiplicar o dividir números son ejemplos de algoritmos, también el algoritmo de Euclides para calcular el máximo común divisor entre dos números.

Un algoritmo bien diseñado siempre brinda una respuesta, puede ser que la respuesta sea que no hay respuesta. También debe estar garantizado que el algoritmo termina. La descripción debe ser clara y precisa, no dejar lugar a la utilización de la intuición o creatividad. Una receta de cocina es un algoritmo si no dice, por ejemplo, "salar a gusto".

Ejemplo de un algoritmo para cebar mate:

1. **Si** la pava no está llena **entonces** llenar la pava con agua
2. **Si** el mate no está vacío **entonces** vaciarlo
3. Colocar la yerba dentro del mate
4. Colocar la bombilla dentro del mate
5. Encender la hornalla y colocar la pava sobre ella
6. **Mientras** el agua no alcance los 72°C, esperar
7. Apagar la hornalla
8. Introducir al agua de la pava dentro del mate

El algoritmo anterior tiene 8 pasos. En algunos de ellos, paso 1 y 2, se deben tomar decisiones. El paso 6, involucra una repetición, esperar hasta que el agua alcance los 72°C.

Ejemplo de un algoritmo para ver una película:

1. Buscar el dvd de la película
2. **Si** el televisor está apagado **entonces** encenderlo
3. **Si** el dvd está apagado **entonces** encenderlo
4. Sacar el dvd de su caja
5. Introducirlo en el reproductor de dvd
6. Agarrar el control remoto del televisor y el del dvd
7. Acomodarse en el sillón
8. Presionar "play"

Ejemplo de un algoritmo para multiplicar dos número naturales:

Escribir multiplicando y multiplicador en dos columnas
Repetir
    Dividir el número de la columna izquierda por 2 y colocar el resultado en esta columna
    Multiplicar el número de la columna derecha por 2 y agregar el resultado en esta columna
hasta que el número en la columna izquierda es 1
Para cada uno de los números de la columna izquierda **hacer**
    Si este número es impar **entonces**
        sumar el valores de la columna izquierda correspondiente
Dar como resultado esta suma

Ejemplo de aplicación del algoritmo para multiplicar dos naturales: Queremos multiplicar 123 y 980.

123	980	980
61	1960	1960
30	3920	
15	7840	7840
7	15680	15680
3	31360	31360
1	62720	62720
		120540

Si queremos describir un algoritmo en lenguaje natural podemos generar confusión, como con el algoritmo anterior. Una forma de escribir un algoritmo para intentar evitar imprecisiones es mediante **pseudocódigos**, semejantes a los usados para escribir los programas informáticos. Ahora vamos a definir el lenguaje del pseudocódigo que utilizaremos.

2 Estructura de un programa

Nuestros algoritmos respetarán la siguiente estructura:

- Encabezado: Nombre del algoritmo/programa, tipo de valor que retorna y parámetros.
- Declaraciones: Declaración de constantes y variables del programa.
- Cuerpo del programa: Conjunto de instrucciones.

tipo de retorno identificador_programa(*parámetros*)
    declaración de constantes
    declaración de variables
    cuerpo del programa

En el transcurso de la clase veremos cada una de estas partes.

3 Variables

En general, los algoritmos trabajan con datos, entonces debemos guardarlos en algún lugar. Este lugar se llama *variable*. Una variable es un *caja*, posición de memoria, utilizada para contener un valor. Para poder identificar y actuar sobre los datos guardados, se le asigna a cada variable (y a todos los objetos de un programa) un nombre o identificador. Es deseable que los identificadores den pistas sobre el significado o uso del objeto que están representando.

Cada variable también tiene asociado un tipo de datos. El tipo determina los valores que pueden almacenarse en dicha variable. Si queremos almacenar el DNI de un persona, la variable será de tipo entero, para una cantidad de dinero puede ser un número con decimales, para el título de un libro, una secuencia de letras. Antes de utilizar una variable tiene que ser declarada:

tipo identificador

Por ejemplo

entero DNI
real sueldo_neto
cadena titulo2

En distintos instantes durante la ejecución del algoritmo, el valor del dato representado por un variable puede ser distinto. El valor de una variable en un momento determinado es el valor almacenado en ese momento en la posición de memoria asociada a la variable. En el algoritmo anterior tendríamos una variable, identificada como *hornalla*, que puede almacenar dos valores, *prendida* y *apagada*. En los pasos 5 y 7, la variable hornalla cambia de valor, pasando primero de apagada a encendida y luego de encendida a apagada.

4 Constantes

Una constante es un valor al que se hace referencia mediante un identificador. Conviene utilizarlas siempre que haya un valor especial que se use en el algoritmo. El valor asignado a una constante no puede ser modificado durante la ejecución del programa. La declaración de una constante tiene la sintaxis:

const *identificador* ← *expresión*

Por ejemplo:

const entero min_dia ← 3600*24
const real porcentaje ← 10,5

5 Literales

Un literal es un valor de cualquier tipo que se utiliza como tal. Por ejemplo:

- 4 es un literal entero.
- 'Acd9_' es un literal cadena
- 'v' es un literal caracter

6 Estados

Las instrucciones transforman los valores almacenados en las variables. Los valores que tienen las variables en un determinado instante durante la ejecución del algoritmo se llama *estado*. En todo estado, cada variable tiene asociado un único valor (del tipo correspondiente) o está indefinida.

Entonces, podemos ver a un algoritmo como una transformación de estados. El estado final debería resolver le problema. La instrucción de asignación transforma los estados, mientras que el resto de las instrucciones son de control.

En el algoritmo para cebar mate, la instrucción 5 cambió el estado de la variable *hornalla*. Antes de ejecutarla su estado era apagado, luego de su ejecución es prendido.

7 Tipos de datos básicos

Como mencionamos anteriormente, los datos utilizados en un algoritmo pueden ser de diferentes tipos. Cada tipo de datos tiene operaciones asociadas. Todo variable tiene un tipo asociado, que limita las operaciones en la que esta variable puede tomar parte. Los tipos básicos son:

- entera: es una cadena de dígitos que no incluye comas o puntos decimales. Por ejemplo, 0, 137, -2516, +17745 son enteros válidos, mientras que los siguientes son inválidos: $5,280$, 16.0, $7-$.
- real: deben representarse como números decimales ordinarios o en notación científica. Por ejemplo, en notación decimal, 1.234, -0.1536, $+56473.0$ son valores reales válidos, mientras que los siguientes no lo son: $1,752.63$, 82, .01, 24.. La notación científica o exponencial consiste en una constante entera o real en forma decimal seguida por la letra E (o e) seguida por una constante entera, que se interpreta como exponente de la base 10. Por ejemplo, 337.456 también puede escribirse como 3.37456E2, 0.337456E3, 337.456E0, 33745.6E-2.
- carácter: incluye los dígitos del 0 al 9, las letras mayúsculas de la A a la Z, las letras minúsculas de la a a la z, los símbolos de puntuación y símbolos especiales como +, =.
- cadena: es una secuencia de caracteres, una constante de tipo cadena consiste en una cadena encerrada entre comillas simples. Por ejemplo, 'Juan J. Garcia' y 'PDQ123-A' son constantes de tipo cadena válidas.
- booleano: sólo hay dos constantes booleanas: *verdadero* y *falso*.

8 Expresiones y operaciones

Las variables y constantes pueden formar parte de operaciones y funciones adecuadas a su tipo. Una *expresión* es un conjunto de datos o funciones unidos por operadores.

8.1 Operadores Aritméticos

Con variables de tipo entero o real se pueden utilizar operados aritméticos para construir expresiones. Los operadores disponibles son:

- suma: $+$
- resta: $-$
- multiplicación: $*$
- división entera: **div**
- división real: /
- módulo: **mod**

En el caso de $+$, $-$ y $*$, si todos lo operandos son enteros, el resultado será entero. Si alguno de ellos es real, el resultado tendrá tipo real.

La división entera, **div**, trabaja sólo sobre operandos enteros y descarta la parte decimal del resultado. La expresión 10 **div** 4 da como resultado 2 y 20 **div** 3 da 6.

En la división real, /, todos los operandos deben ser de tipo real y da como resultado un valor real. Por ejemplo 10.0 **div** 4 retorna 2.5. En todos los casos, la división por cero da como resultado *indefinido*.

El operador **mod** sólo puede utilizarse con números enteros. Este operador calcula el resto de la divisón entera. La expresión 7 **mod** 2 da como resultado 1.

8.2 Operadores Lógicos

Los operadores lógicos se aplican sobre datos y expresiones de tipo booleano y retornan un booleano, verdadero o falso. Los operadores usuales son los conocidos:

- **y**: retorna verdadero si y sólo si los dos operandos son verdaderos.
- **o**: retorna verdadero si al menos uno de los dos operandos es verdadero.
- **no**: actúa sobre un sólo operando, negando su valor.

8.3 Operadores Relacionales

Se utilizan para describir relaciones entre dos valores. Son los operadores habituales de comparación entre números, el resultado es de tipo booleano:

- mayor que: $>$

- menor que: $<$
- mayor o igual que: $\geq$
- menor o igual que: $\leq$
- igual a: $=$
- distinto a: $\neq$

Por ejemplo:

- $10 > 8$ resulta **verdadero**
- $10 \leq 8$ resulta **falso**
- $x < 8$ **y** $x \geq 3$ resulta **verdadero** si $x \in [3, 8)$ y **falso** en caso contrario

También pueden utilizarse para comparar dos elementos de tipo caracter. Si la variable *letra* contiene una *a*, la expresión *letra*='a' será **verdadera**, en caso contrario será **falsa**. El orden establecido para los elementos de tipo caracter es el siguiente: los caracteres numéricos son menores que las letras mayúsculas que a su vez son menores que las letras minúsculas. Dentro de cada uno de estos grupos el orden es: '0' < '1'< ... < '9'; 'A' < 'B'< ... < 'Z' y 'a' < 'b'< ... < 'z'.

8.4 Funciones sobre cadenas

Nuestro pseudocódigo tendrá definidos dos funciones sobre los datos de tipo cadena.

- **long**: retorna en un entero la longitud de la cadena.
- **carac**: se aplica sobre un dato de tipo cadena y un dato entero. Retorna el caracter que se encuentra en la posición de la cadena que indica el entero. El primer caracter de una cadena se encuentra en la posición 0, y el último en la longitud de la cadena menos 1. Si se intenta aplicar sobre un entero fuera del rango anterior, la respuesta estará indefinida.

Ejemplos:

- long('una cadena') retorna 10
- 'una cadena'$_4$ retorna 'c'
- 'una cadena'$_1$1 tiene respuesta indefinida

8.5 Prioridad en la evaluación de expresiones

Cuando en una expresión aparecen dos o más operandos, tenemos que definir en que orden se realizan las operaciones, es decir, cúal es el *orden de prioridad* de las operaciones. Vamos a seguir las siguientes reglas:

1. Todas las subexpresiones entre paréntesis se evalúan primero. Las subexpresiones con paréntesis anidados se evalúan de adentro hacia afuera.

2. Dentro de los operadores aritméticos, primero se evalúan $*$, **div** y **mod** y luego $+$ y $-$.

3. Dentro de los operadores lógicos, el orden es primero **no**, después **y** y por último **o**.

4. En una expresión o subexpresión con operadores de distinto tipo se sigue el siguiente orden:

   (a) **no**
   (b) *, **div**, **mod**
   (c) +, -
   (d) $<$, $\leq$, $\geq$, $>$
   (e) $=$, $\neq$
   (f) **y**
   (g) **o**

5. Los operadores con igual nivel de prioridad se evaluan de izquierda a derecha.

Ejemplos:

- $4 + 2 * 5 = 4 + 10 = 14$
- $23 * 2$ **div** $5 = 46$ **div** $5 = 9$
- $3 + 5 * (10 - (2 + 4)) = 3 + 5 * (10 - 6) = 3 + 5 * 4 = 3 + 20 = 23$
- $2.1 * (1.5 + 3 * 4.1) = 2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98$
- $(2 + 3 * 3) \neq 6 + (6 - 1)$ retorna **falso**
- $6 \geq 3 + 3$ retorna **verdadero**
- **no**('A'='Z') retorna **verdadero**
- En la expresión **no** p **o** q **y** r, primero se evalúa **no** p, después q **y** r y por último el **o**.
- En la expresión **no** (p **o** q **y** r), primero se evalúa q **y** r, luego el **o** y por último la operación **no**.

9 Asignación

La instrucción de asignación se usa para asignar o modificar el valor de una variable. Las variables están indefinidas hasta que sus valores se espefican explícitamente mediante una instrucción de asignación. La sentencia de asignación tiene la forma:

variable $\leftarrow$ *expresión*

Cuando se ejecuta una sentencia de asignación, primero se evalua la expresión situada a la derecha del $\leftarrow$ y luego se coloca el valor resultado en la variable indicada a la izquierda. Por ejemplo,

$mes \leftarrow 6$

En la sentencia anterior, la variable es *mes* y la expresión es el número 6. La computadora ejecuta esta asignación poniendo el literal 6 en la variable *mes*, reemplazando cualquier valor que tuviera anteriormente. La instrucción de asignación es destructiva, el valor almacenado anteriormente en la variable se destruye y se sustituye por el nuevo valor.

La aparición de una variable en una instrucción de asignación tiene distinto sentido si aparece a la derecha o izquierda del símbolo de asignación. Si aparece a la izquierda, representa una celda de memoria, si aparece a la derecha, representa el valor contenido en una celda de memoria. Si ejecutamos

$mes \leftarrow 6$
$mes\_anterior \leftarrow mes - 1$

La primera instrucción hace que se almacene el valor 6 en la variable *mes*. La segunda indica que al valor guardado en la variable *mes* se le reste 1 y luego el valor resultante sea almacenado en la variable *mes_anterior*, desapareciendo el valor guardado anteriormente en ella. El valor guardado en la variable *mes* no resulta afectado por la segunda asignación.

Ejemplos de asignaciones válidas son:

$a \leftarrow a + 1$
$a \leftarrow 10$
$a \leftarrow 2 * a + 5 * b$

Mientras que las siguientes no lo son:

$a + 3 \leftarrow b$
$10 \leftarrow b$

Una variable en el lado derecho de una instrucción de asignación debe tener un valor antes de que se ejecute esta. Si ejecutamos

entero a, b
$a \leftarrow b + 1$

como la variable b no tiene asignado valor, es impredecible que valor tomará a.

Los tipos del identificador y de la expresión deben ser el mismo o compatibles. Por ejemplo, los valores enteros pueden almacenarse en variables de tipo real. La asignación de un valor real a una variable de tipo entero provocará la perdida de la parte decimal.

10 Estructuras de control

En lo desarrollado hasta ahora, cada instrucción se ejecuta exactamente una vez en el orden en que aparecía. Frecuentemente se presentan situaciones en las que se debe proporcionar instrucciones alternativas que pueden o no ejecutarse dependiendo de los datos de entrada. Para determinan el orden en que las instrucciones de un algoritmo se ejecutarán se utilizan las estructuras de control. Las estructuras básicas son:

- Secuencia
- Decisión
- Repetición

10.1 Secuencia

Cada paso del algoritmo es ejecutado en el orden especificado. En el ejemplo anterior, si no se respeta el orden de los pasos, por ejemplo, se coloca la pava en la hornalla antes de colocarle agua, el algoritmo probablemente falle. Las instrucciones contenidas en un algoritmo se ejecutan secuencialemte. Para definir un bloque o secuencia de instrucciones tabularemos las instrucciones.

La ejecución de una secuencia se realiza ejecutando las instrucciones una detrás de otra según el orden de su escritura, de izquierda a derecha y de arriba a abajo.

Si ejecutamos la secuencia:

$x \leftarrow 4$
$x \leftarrow y+1$

Primero se realiza la primera asignación, que almacena 4 en la variable x. Posteriormente se realiza la segunda asignación, que almacena en la variable x el valor contenido en la variable y sumándole 1. Este ejemplo carece de sentido, porque la segunda instrucción destruye lo realizado por la primera instrucción.

Veamos un ejemplo algo más interesante. Tenemos dos variable, x e y, y queremos intercambiar sus valores. Dado que se quiere almacenar en cada variable el valor contenido en la otra, una primera propuesta puede ser:

$x \leftarrow y$
$y \leftarrow x$

Si la asignación se realizara simultaneamente, la solución sería correcta, pero su ejecución es secuencial, por lo que se pierde el valor inicial de x.

Si, por ejemplo, inicialmente $x = 10$, $y = 4$, al ejecutar la primera asignación x pasará a contener el valor 4 e y no modificará su valor. La segunda asignación no modificará el valor de x y a y se le asignará el valor de x, o sea seguirá valiendo 4. El error consiste en que la primera asignación hace desaparecer el valor almacenado en x.

La solución consiste en utilizar una variable auxiliar para poder guardar temporalmente el valor de x antes de perder su valor:

```
aux ← x
x ← y
y ← aux
```

10.2 Decisión o condicional

Una instrucción condicional permite decidir si se ejecuta o no ciertas instrucciones. La evaluación de una decisión está basada en alguna condición que sólo puede resultar ser verdadera o falsa, no puede haber "más o menos". Por ejemplo:

```
si hoy es viernes entonces
        desconectar el despertador
```

Las construcciones utilizadas para expresar decisiones que deben ser tomadas tienen la forma (sentencia *SI*):

```
si condición entonces
        cuerpo del SI
```

Una condición es una expresión lógica, sólo puede tomar valor **verdadero** o **falso**, por ejemplo es **verdadero** que hoy es viernes o es **falso** que hoy es viernes, no puede ser ambos ni ninguno. Si la condición es verdadera entonces se ejecuta el *cuerpo del SI*. Luego el control pasará a la instrucción siguiente a la sentecia *SI*.

También pueden tener la forma:

```
si condición entonces
        cuerpo del SI
sino
        cuerpo del SINO
```

Esto significa que si la condición es verdadera se ejecuta el *cuerpo del SI*, en caso contrario, se ejecuta *el cuerpo del SINO*.

Podemos hallar el máximo entre dos números x e y mediante las siguientes instrucciones:

```
si x > y entonces
        max ← x
sino
        max ← y
```

Más ejemplos. Queremos saber si un número es positivo o no.

```
cadena PosNeg(entero x)
        cadena cadena_salida
        si x > 0 entonces
                cadena_salida ← 'valor positivo'
        sino
                cadena_salida ← 'valor positivo o cero'
        retornar cadena_salida
```

10.3 Sentencias SI anidadas

La instrucción siguiente a la palabra **entonces** o **sino** puede ser cualquiera, incluso otra sentencia SI. Cuando una o ambas bifurcaciones también contiene una sentencia SI, se dice que dichas sentencias están anidadas. Una sentencia SI anidad se puede utilizar para construir decisiones con diferentes alternativas. Se pueden presentar diferentes alternativas, por ejemplo:

```
si condición 1 entonces
        cuerpo del SI 1
sino
        si condición 2 entonces
                cuerpo del SI 2
        sino
                cuerpo del SINO 2
```

Para hallar el máximo entre tres números x, y y z podemos ejecutar:

```
si x > y y x > z entonces
        max ← x
sino
        si x > y entonces
                max ← z
        sino
                max ← y
```

Ejemplo: Una compañía de alquiler de autos desea un programa para emitir las facturas de sus clientes, teniendo presentes los siguientes puntos:

- La empresa cobra una tarifa fija de $200 cuando el auto es utilizado menos de 300km.
- Cuando la distancia recorrida es mayor de 300km:
  - Si la distancia es menor a 1000 km, la tarifa es de $200 más $0,5 por cada km extra.
  - Si la distancia es mayor a 1000 km, la tarifa es de $200 más $0,5 por cada km entre los 300 y 1000 más $0,3 por cada km que supere los 1000km.

Entrada:

- Costo fijo hasta 300km.
- Costo por km entre los 300 y 1000km.
- Costo por km superior a 1000km.
- Kilómetros recorridos

Salida:

- Monto de la tarifa

```
real CalculoTarifa(entero km, real costo_fijo, real costo_300, real costo_1000)
        real tarifa
        si km ≤ 300 entonces
                tarifa ← costo_fijo
        sino
                si km ≤ 1000 entonces
                        tarifa ← 200+costo_300*(km − 300)
                sino
                        tarifa ← 200+costo_300*1000+costo_1000*(km − 1000)
        retornar tarifa
```

10.4 Repetición

Una sentencia repetitiva, ciclo, engloba una secuencia de intrucciones que se escribe una sola vez, pero permite que se ejecute varias veces. Las intrucciones englobadas se llaman cuerpo del ciclo y cada ejecución de ellas se llama iteración. Las repeticiones pueden tomar tres formas:

- Ciclo **mientras**: generalmente el número de iteraciones no se conoce por anticipado y el cuerpo del ciclo se ejecuta hasta que se satisfaga una determinada condición.
- Ciclo **repetir**: es una variante del ciclo **mientras**.
- Ciclo **para**: se ejecuta un número determinado de veces, conocido cuando comienza la ejecución del ciclo, pero no necesariamente en tiempo de compilación.

La sintaxis de la construcción **mientras** es la siguiente:

```
mientras condición hacer
        cuerpo del ciclo
```

Cuando se utiliza la estructura *mientras* el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando la sentencia comienza a ejecutarse, el primer paso es la evaluación de la condición lógica. Si se obtiene

falso como resultado, ninguna acción se realiza y el programa prosigue en la siguiente sentencia después del ciclo. Si la condición lógica resulta verdadera, entonces se ejecutan las sentencias contenidas dentro del cuerpo del ciclo y luego se evalúa nuevamente la condición. Este proceso se repite mientras que la condición lógica sea verdadera. Después de cada iteración, la condición lógica es evaluada y si es verdadera, el ciclo se repite nuevamente, si cambia de verdadera a falsa, la sentencia mientras finaliza y la ejecución prosigue con la siguiente sentencia. La expresión de control de un ciclo puede ser compuesta: $suma < 100$ **y** $letra =' a'$.

Mientras la condición sea verdadera el ciclo se ejecutará. Esto significa que el ciclo se ejecutará indefinidamente a menos que "algo" en el interior del ciclo modifique la condición, haciendo que su valor pase a falso. Si la expresión lógica siempre conserva el valor verdadero, el algoritmo no terminará.

Podemos escribir el siguiente algoritmo para calcular el menor número impar cuyo cuadrado sea mayor a 500:

```
entero impar
impar ← 1
mientras impar * impar > 500 hacer
        impar ← impar + 2
```

¿Qué sucede si en el ejemplo anterior modificamos la condición de esta forma?:

```
entero impar
impar ← 1
mientras impar * impar ≠ 500 hacer
        impar ← impar + 2
```

El ciclo nunca termina!!!!

Un programa terminará si y sólo si todos sus ciclos terminan. Al diseñar un algoritmo debemos tener especial cuidado en poder asegurar esto.

Más ejemplos. Queremos saber el número de 'a' de una cadena dada.

```
entero Cantidad_de_a(cadena texto)
        entero cantidad
        entero indice
        indice ← 0
        mientras indice < long(texto) entonces
                si carac(texto,indice)= 'a' entonces
                        cantidad ← cantidad + 1
                indice ← indice + 1
        retornar cantidad
```

Queremos escribir un algoritmo para saber si un número entero dado es primo.

```
booleano EsPrimo?(entero numero)
        booleano primo
        enteros divisor
        primo ← verdadero
        divisor ← 2
        mientras divisor < numero entonces
                si numero mod divisor = 0 entonces
                        primo ← falso
                divisor ← divisor + 1
        retornar primo
```

El ciclo **repetir** es una variante de la sentencia **mientras**. La diferencia es que la condición lógica se evalúa después de ejecutar el cuerpo del ciclo, o sea, al finalizar cada iteración. Al comenzar la ejecución de una sentencia **mientras**, si la condición es falsa, el cuerpo del ciclo jamás se ejecuta. Cuando se utiliza la sentencia **repetir** el cuerpo del ciclo se repite hasta que la condición se hace verdadera. Tiene la forma:

```
repetir
        cuerpo del ciclo
hasta condición
```

Por ejemplo:

```
suma ← 0
cont ← 0
repetir
        suma ← suma + cont
        cont ← cont + 1
hasta suma ≥ 5
```

El cuerpo del ciclo es $suma \leftarrow suma + cont$ y $cont \leftarrow cont + 1$ y la condición $suma \geq 5$. El ciclo **repetir** realiza un proceso, cuerpo del ciclo, **antes** de evaluar la condición. Después de cada iteración del cuerpo del ciclo se evalúa la condición y si resulta verdadera el ciclo se termina y se sale de él, ejecutándose la siguiente sentencia. Si la condición es falsa el ciclo se repite. En el ejemplo anterior, al finalizar el ciclo en la variable *suma* estará almacenado el valor 6.

Dado un natural n, el siguiente algoritmo calcula la suma de los primeros n enteros no negativos.

```
entero Suma(entero n)
        entero contador
        entero suma
        contador ← 0
        suma ← 0
        repetir
                suma ← suma + contador
                contador ← contador + 1
        hasta contador = n
        retornar suma
```

La elección de utilizar una sentencia **mientras** o una **repetir** dependerá del diseño del algoritmo. Si bien con la sentencia **mientras** se pueden escribir todos los algoritmos, en muchas ocasiones la sentencia **repetir** facilita la escritura del algoritmo.

En numerosas ocasiones podemos querer que un ciclo se ejecute un número dado de veces, y este número de iteraciones se conoce antes de la ejecución del ciclo. Generalmente, para este tipo de aplicaciones se utiliza la sentencia **para**. Tiene el siguiente formato:

```
para variable en límite inferior, límite superior cada incremento hacer
        cuerpo del ciclo
```

Esta instrucción usa una variable, llamada variable de índice, que toma distintos valores en las sucesivas ejecuciones del cuerpo. La variable de índice debe ser entera. Los valores que toma la variable de índice se indican mediante un subrango (límite inferior y límite superior) y un incremento. El subrango se expresa mediante un valor inicial y un valor final. El incremento es dado por una expresión entera.

Al comenzar la ejecución del ciclo, la variable índice toma el valor inicial del subrango. Tras cada iteración, el valor de la variable índice se actualiza, sumándole a su valor el incremento. Si el valor de la variable índice es menor o igual que el valor final del subrango se realiza la siguiente iteración del ciclo.

¿Qué calculo realizan los siguientes dos algoritmos?

```
entero numero
entero suma
suma ← 0
para numero en 1,100 cada 1 hacer
        suma ← suma + numero
```

```
entero numero
entero suma
suma ← 0
para numero en 50,100 cada 2 hacer
        suma ← suma + numero
```

Dado un número natural n, vamos a desarrollar un algoritmo que calcule el n-ésimo número de Fibonacci.

```
entero Fibonacci(entero n)
        entero i
        entero j
        entero contador
        anterior ← 1
        siguiente ← 0
        para contador en 1,n cada 1 hacer
                siguiente ← anterior + siguiente
                anterior ← siguiente − anterior
        retornar siguiente
```

Los siguientes tres algoritmos calculan el factorial de un natural utilizando las diferentes estructuras repetitivas.

```
entero Factorial(entero n)
        entero contador
        entero resultado
        contador ← 1
        resultado ← 1
        mientras contador ≠ n hacer
                resultado ← resultado * contador
                contador ← contador + 1
        retornar resultado
```

```
entero Factorial(entero n)
        entero contador
        entero resultado
        contador ← 1
        resultado ← 1
        repetir
                resultado ← resultado * contador
                contador ← contador + 1
        hasta contador > n
        retornar resultado
```

```
entero Factorial(entero n)
        entero contador
        entero resultado
        resultado ← 1
        para contador en 1,n cada 1 hacer
                resultado ← resultado * contador
        retornar resultado
```

Aunque con la sentencia **mientras** se pueden desarrollar casi todos los algoritmos repetitivos, cada clase de ciclo resulta útil en situaciones diferente, ayudando a la claridad del algoritmo y haciendolo más eficiente.

El cuerpo del ciclo puede contener cualquier tipo de sentencias, inclusive sentencias selectivas o repetitivas. Cuando se anidan estructuras de control, la esctructura interior debe estar contenida completamente dentro de la estructura externa. Si las estructuras se cruzan no serán válidas.

Desarrollemos en pseudocódigo el algoritmo que vimos anteriormente para calcular el producto entre dos número naturales.

```
entero Producto(entero n, entero m)
        entero resultado
        resultado ← 0
        repetir
                si impar(m) entonces
                        resultado ← resultado + n
                m ← m div 2
                n ← n * 2
        hasta m = 1
        retornar resultado
```

La condición "impar(m)" hace referencia a un subproblema, para el cual asumimos que existe un algoritmo llamado "impar" que retorna verdadero cuando se lo aplica a un número impar y falso en caso contrario. Más adelante vamos a desarrollar este tema.