

Introducción a la Computación

Ciclos

while, for, do while, patrones y getchar

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Repetir: `mientras`, `para` y la flecha que regresa
2. `while` y el ciclo infinito
3. `for`: un `while` con las tres partes juntas
4. Salirse por uno. No controlar un ciclo con `double`
5. `do while`: validar una entrada
6. Patrones: contador, acumulador, centinela y buscar el mayor
7. Leer hasta el fin de la entrada con `getchar`

> **Lectura**
>
> Apuntes de la semana 5. Código en `code/week-05`.

SECCIÓN 01

Repetir

01



Las monedas de la semana 1

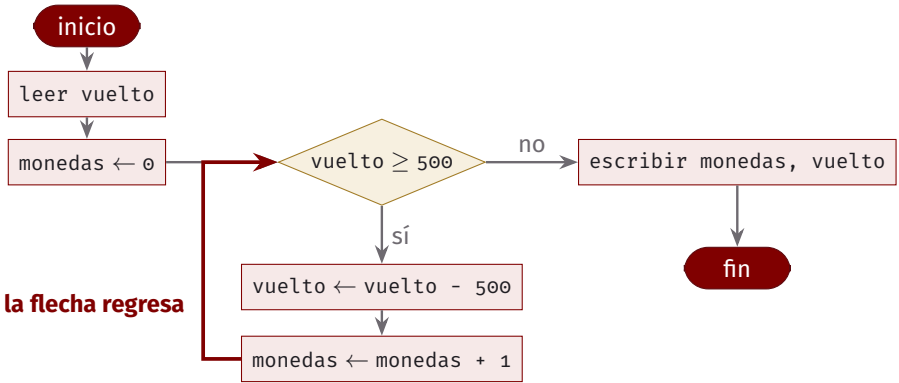
pseudocódigo

```
algoritmo monedas_de_cinco()
    leer vuelto
    monedas ← 0
    mientras vuelto ≥ 500 hacer
        vuelto ← vuelto - 500
        monedas ← monedas + 1
    fin mientras
    escribir monedas, vuelto
fin algoritmo
```

Vuelta	**¿≥ 500?**	vuelto	monedas
		1780	0
1	sí	1280	1
2	sí	780	2
3	sí	280	3
	no	280	3

Cada pasada por el bloque es una **vuelta** (o iteración).

En un diagrama: la flecha que regresa



En un `si`, los dos caminos se juntan más abajo. En un ciclo, uno de ellos vuelve atrás.

Tres preguntas para cualquier ciclo

Pregunta	**En `monedas_de_cinco`**
Inicio: ¿qué valen las variables al empezar?	`vuelto` es el dato, `monedas` es 0
Parada: ¿cuándo se deja de repetir?	cuando `vuelto` $<$ 500
Avance: ¿qué cambia en cada vuelta?	`vuelto` baja 500

Idea clave

Si no pueden responder la tercera, probablemente tienen un **ciclo infinito** (*(infinite loop)*).

`para`: repetir con un contador

pseudocódigo

```
algoritmo suma_hasta_n()
    leer n
    suma ← 0
    i ← 1
    mientras i ≤ n hacer
        suma ← suma + i
        i ← i + 1
    fin mientras
    escribir suma
fin algoritmo
```

pseudocódigo

```
algoritmo suma_hasta_n_con_para()
    leer n
    suma ← 0
    para i desde 1 hasta n hacer
        suma ← suma + i
    fin para
    escribir suma
fin algoritmo
```

`hasta` incluye el último valor: hay una vuelta con `i` igual a `n`.

SECCIÓN 02

while

02



`while` en C

coins.c

```c
while (change >= 500) {
    change = change - 500;
    coins = coins + 1;
}
```

terminal

```
$ echo 1780 | build/coins
3 monedas de S/ 5; quedan 280 céntimos
$ echo 300 | build/coins
0 monedas de S/ 5; quedan 300 céntimos
```

Regla

Se pregunta **antes** de cada vuelta. Por eso `while` puede dar cero vueltas.

La seguimos a mano, con `n = 4`

sum_to_n.c

```c
int sum = 0;
int i = 1;
while (i <= n) {
    sum = sum + i;
    i = i + 1;
}
```

Pregunta	sum	i
antes de empezar	0	1
¿1 <= 4? sí	1	2
¿2 <= 4? sí	3	3
¿3 <= 4? sí	6	4
¿4 <= 4? sí	10	5
¿5 <= 4? no	10	5

Para contar

4 vueltas y 5 preguntas: la condición se evalúa una vez más que las vueltas. Al salir, `i` vale 5, no 4.

Formas cortas de asignar

Forma corta	Equivale a
`i++`	`i = i + 1`
`i--`	`i = i - 1`
`sum += x`	`sum = sum + x`
`n -= x`	`n = n - x`
`w /= 2`	`w = w / 2`

En este curso

`i++` va solo, como instrucción aparte o como avance de un `for`. No lo metan dentro de otras expresiones.

El ciclo infinito

errors/no_update.c

```c
int n = 3;
while (n > 0) {
    printf("%d\n", n);
}
```

- Nada cambia `n`. `gcc` no avisa: es C válido.
- En la terminal se corta con **Ctrl+C**.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/
    no_update.c -o build/no_update
$ build/no_update | head -n 3
3
3
3
```

`head -n 3` deja pasar 3 líneas y termina. Con eso, el programa que le escribía también se detiene.

Un acumulador puede dar overflow

terminal

```
$ echo 70000 | build/sum_to_n
sum_to_n.c:15:13: runtime error: signed integer overflow: 2147450880 + 65536 cannot be represented in type 'int'
```

- 2147450880 + 65536 no cabe en un `int`.
- Es **comportamiento indefinido**: el estándar de C no dice qué pasa.
- El número que el programa escriba después no significa nada. Por eso no lo copiamos aquí.
- Solución: limitar el dato antes del ciclo, o elegir un problema cuyo resultado quepa.

SECCIÓN 03

for

03



`for` se comporta como un `while`

la forma

```
for (inicio; condición; avance) {
    cuerpo
}
```

equivale a

```
inicio;
while (condición) {
    cuerpo
    avance;
}
```

1. `inicio` se ejecuta **una sola vez**.
2. Se evalúa `condición`. Si es falsa, el ciclo termina.
3. Se ejecuta el `cuerpo`.
4. Se ejecuta `avance`, después del cuerpo, y se vuelve al paso 2.

Lo seguimos línea por línea

for_trace.c

```c
int i;
for (i = 0; i < 10; i += 3) {
    printf("i vale %d\n", i);
}
printf("Al salir, i vale %d\n", i);
```

terminal

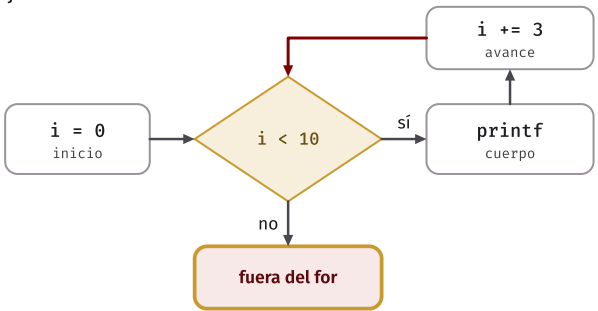
```
$ build/for_trace
i vale 0
i vale 3
i vale 6
i vale 9
Al salir, i vale 12
```

i	i < 10	**Qué pasa**
0	sí	escribe; pasa a 3
3	sí	escribe; pasa a 6
6	sí	escribe; pasa a 9
9	sí	escribe; pasa a 12
12	no	se sale

Cuatro vueltas. `i` queda en el primer valor que hace falsa la condición. `for_as_while.c`, con `while`, escribe lo mismo.

El orden de las partes del `for`

```
for (i = 0; i < 10; i += 3) {
    printf("i vale %d\n", i);
}
```



VARIABLE

i 12

SALIDA

```
i vale 0
i vale 3
i vale 6
i vale 9
Al salir, i vale 12
```

al salir, i vale 12: el primer valor que hizo falsa la condición

De para a for

Pseudocódigo	C
`para i desde 1 hasta n hacer`	`for (int i = 1; i <= n; i++)`
`para i desde 0 hasta n - 1 hacer`	`for (int i = 0; i < n; i++)`

- `int i` dentro del `for`: solo existe dentro de ese `for`, como en un bloque.
- `para` avanza de 1 en 1. Con otro paso, en pseudocódigo se usa `mientras`.

terminal

```
$ echo 7 | build/mult_table | head -n 3
7 X  1 = 7
7 X  2 = 14
7 X  3 = 21
```

Un binario sin repetir líneas

to_binary.c

```c
for (int weight = 128; weight >= 1; weight
     /= 2) {
    if (n >= weight) {
        printf("1");
        n -= weight;
    } else {
        printf("0");
    }
}
```

terminal

```
$ echo 202 | build/to_binary
11001010
```

`weight`	`n` antes	**Escribe**	`n` después
128	202	1	74
64	74	1	10
32	10	0	10
16	10	0	10
8	10	1	2
4	2	0	2
2	2	1	0
1	0	0	0

Después de 1 viene `1 / 2`, que en enteros es 0. Ahí termina.

Error común: un ; después del paréntesis

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/semicolon_for.c -o build/semicolon_for
errors/semicolon_for.c:7:5: warning: this ‘for’ clause does not guard... [-Wmisleading-indentation]
    7 |     for (i = 1; i <= 3; i++);
$ build/semicolon_for
4
```

- El `;` termina el `for`: su cuerpo queda vacío.
- El bloque de abajo corre una sola vez, con el valor de salida: `4`.
- Con `while (n > 0);` es peor: se queda preguntando para siempre.

SECCIÓN 04

Salirse por uno

04

Una vuelta de más o de menos

Ciclo	`i`	Vueltas
`for (int i = 0; i < n; i++)`	0 a n−1	n
`for (int i = 1; i <= n; i++)`	1 a n	n
`for (int i = 0; i <= n; i++)`	0 a n	n+1
`for (int i = 1; i < n; i++)`	1 a n−1	n−1

Vueltas con `n` ≥ 1. De `a` hasta `b`, incluyendo los dos, hay `b - a + 1` valores: del 3 al 7 hay 5.



Quería escribir del 1 al 5, con `i < 5`. `gcc` no avisa. Es salirse por uno (*(off-by-one)*).

Idea clave

Revisen a mano la **primera** vuelta, la **última** y el caso de **cero** vueltas.

No controlen un ciclo con `double`

errors/double_never_equal.c

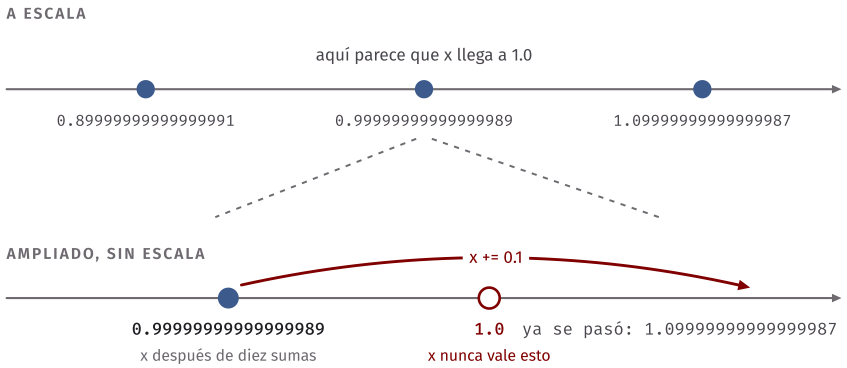
```c
for (double x = 0.0; x != 1.0; x += 0.1)
    {
    printf("%.17f\n", x);
}
```

- Tras diez sumas, `x` no es 1.0: la siguiente suma ya se pasó.
- `x != 1.0` nunca es falso. Aquí lo detiene `head`.
- Y con `x < 1.0`, hay 11 vueltas, no 10.

terminal

```
$ build/double_never_equal | head -n 12
0.00000000000000000
0.10000000000000001
0.20000000000000001
0.30000000000000004
0.40000000000000002
0.50000000000000000
0.59999999999999998
0.69999999999999996
0.79999999999999993
0.89999999999999991
0.99999999999999989
1.09999999999999987
```

Pasa junto a 1.0, sin tocarlo



Por eso x != 1.0 nunca se hace falsa, y con x < 1.0 hay una vuelta de más.

Cuenten con un entero

double_steps.c

```c
for (int i = 0; i <= 10; i++) {
    double x = i / 10.0;
    printf("%.1f\n", x);
}
```

- La cantidad de vueltas ya no depende de ningún redondeo: siempre son 11.
- El mismo cuidado con `!=` y un paso entero: en `for (i = 0; i != 10; i += 3)`, `i` salta el 10.
- Mejor `i < 10`, que termina aunque no caiga justo ahí.

SECCIÓN 05

do while

05



`do while`: al menos una vez

while_vs_do.c

```c
int n = 5;

while (n < 5) {
    printf("while: entré\n");
    n++;
}

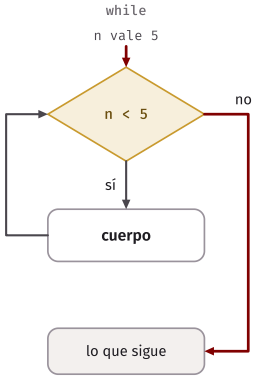
do {
    printf("do while: entré\n");
    n++;
} while (n < 5);
```

terminal

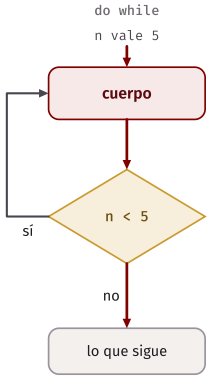
```
$ build/while_vs_do
do while: entré
n vale 6
```

- `while` pregunta **antes**.
- `do while` pregunta **después** del cuerpo.
- Lleva `;` al final.

La misma pregunta, en otro lugar



Con n en 5, el cuerpo no corre:
cero vueltas



El cuerpo corre aunque n < 5 sea falso:
una vuelta, y n queda en 6

Validar una entrada

pseudocódigo

```
algoritmo pedir_mes()
    leer mes
    mientras mes < 1 o mes > 12 hacer
        escribir "Mes fuera de rango,
     escriba otro"
        leer mes
    fin mientras
    escribir mes
fin algoritmo
```

En pseudocódigo usamos `mientras`: la lectura aparece dos veces. En C, `do while` la deja en una.

read_month.c

```c
do {
    if (scanf("%d", &month) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (month < 1 || month > 12) {
        printf("Mes fuera de rango,
      escriba otro\n");
    }
} while (month < 1 || month > 12);
```

`scanf` puede fallar en cada vuelta: se revisa **dentro** del ciclo.

Las tres formas de terminar

terminal

```
$ echo 13 0 7 | build/read_month
Mes fuera de rango, escriba otro
Mes fuera de rango, escriba otro
Mes elegido: 7
$ echo abc | build/read_month; echo $?
Entrada inválida
1
$ echo 13 | build/read_month; echo $?
Mes fuera de rango, escriba otro
Entrada inválida
1
```

- **Un mes válido:** sale del ciclo.
- **abc:** `scanf` devuelve 0.
- **Se acabó la entrada:** `scanf` devuelve `EOF`.

En los dos últimos casos, el programa avisa y termina con `return 1`.

Sin revisar `scanf`, no termina

errors/read_forever.c

```c
do {
    scanf("%d", &month);
    printf("Mes fuera de rango\n");
} while (month < 1 || month > 12);
```

terminal

```
$ echo abc | build/read_forever |
    head -n 3
Mes fuera de rango
Mes fuera de rango
Mes fuera de rango
```

`%d` no puede leer `abc`: no lo consume y devuelve 0. La vuelta siguiente encuentra lo mismo. Al acabarse la entrada pasa igual: sigue devolviendo `EOF`.

Regla

Un `scanf` en un ciclo se revisa **dentro** del ciclo, y una falla termina el ciclo.

¿Cuál ciclo elegir?

Situación	Conviene	Ejemplo
Se conoce la cantidad de vueltas antes de empezar	`for`	la tabla del 1 al 12
Se repite mientras algo se cumpla, y puede no dar ninguna vuelta	`while`	las monedas
El cuerpo tiene que correr antes de poder preguntar	`do while`	validar un dato

Ninguno es obligatorio

Cualquiera se puede escribir con otro. Se elige el que deja el programa más claro.

SECCIÓN 06

Patrones

06

Contador: hasta el fin de la entrada

count_passed.c

```c
int status = scanf("%d", &grade);
while (status == 1) {
    total++;
    if (grade >= 11) {
        passed++;
    }
    status = scanf("%d", &grade);
}

if (status != EOF) {
    printf("Entrada inválida\n");
    return 1;
}
```

- Empieza en 0 y suma 1 cuando pasa algo.
- **Se lee antes de preguntar**: una lectura antes del ciclo y otra al final de cada vuelta.
- El `if` de abajo distingue por qué salió.

terminal

```
$ build/count_passed < grades.txt
Aprobaron 3 de 5
$ echo 14 abc | build/count_passed
    ; echo $?
Entrada inválida
1
```

Error común: `scanf(...) != EOF`

errors/not_eof.c

```c
while (scanf("%d", &n) != EOF) {
    count++;
    printf("Llevo %d datos; el último es %d\n", count, n);
}
```

terminal

```
$ echo abc | build/not_eof | head -n 3
Llevo 1 datos; el último es 0
Llevo 2 datos; el último es 0
Llevo 3 datos; el último es 0
```

Con `abc`, `scanf` devuelve 0, que es distinto de `EOF`. Pregunten siempre `== 1`.

Acumulador

Qué es

Una variable que va juntando un resultado, como `sum` en `sum_to_n.c`.

- Empieza en el resultado de “ningún dato todavía”: **0** para una suma, **1** para un producto.
- Sin valor inicial, `int sum;` y luego `sum += x` es comportamiento indefinido.
- Con `sum = 0` dentro del ciclo, la suma se reinicia en cada vuelta.
- Puede dar overflow: limiten los datos.

Centinela: un valor que avisa el final

sentinel_sum.c

```c
int status = scanf("%d", &price);
while (status == 1 && price != 0) {
    total += price;
    items++;
    status = scanf("%d", &price);
}

if (status != 1) {
    printf("Entrada inválida\n");
    return 1;
}
```

Cortocircuito: si `status` no es 1, `price` no se mira. El 0 no se suma ni se cuenta.

Lee	¿Sigue?	total	items
1250	sí	1250	1
300	sí	1550	2
4050	sí	5600	3
0	no	5600	3

terminal

```
$ echo 1250 300 4050 0 | build/
    sentinel_sum
3 productos: 56.00 soles
$ echo 1250 300 | build/
    sentinel_sum; echo $?
Entrada inválida
1
```

Buscar el mayor

max_of_n.c

```c
int max = 0;  // Este 0 nunca se usa: la
      primera vuelta lo reemplaza.
for (int i = 0; i < n; i++) {
    int temperature;
    if (scanf("%d", &temperature) != 1) {
        printf("Entrada inválida\n");
        return 1;
    }
    if (i == 0 || temperature > max) {
        max = temperature;
    }
}
```

i	temp.	**¿Cambia?**	max
0	−3	sí, `i == 0`	−3
1	−7	no	−3
2	−1	sí	−1

terminal

```
$ echo 3 -3 -7 -1 | build/max_of_n
La mayor es -1
```

Error común: empezar el mayor en 0

errors/max_from_zero.c

```c
int max = 0;
...
if (temperature > max) {
    max = temperature;
}
```

terminal

```
$ echo 3 -3 -7 -1 | build/max_from_zero
La mayor es 0
```

Idea clave

0 no es un dato: el programa responde algo que nadie midió. El mayor empieza con **el primer dato**.

SECCIÓN 07

getchar

07



`getchar`: byte por byte, hasta `EOF`

count_bytes.c

```c
int count = 0;

int c = getchar();
while (c != EOF) {
    count++;
    c = getchar();
}
```

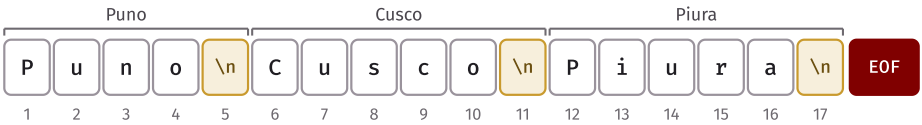
terminal

```
$ build/count_bytes < cities.txt
17 bytes
$ wc -c cities.txt
17 cities.txt
$ echo ñ | build/count_bytes
3 bytes
```

- Cada llamada devuelve el siguiente byte, también espacios y saltos de línea.
- `ñ` ocupa 2 bytes en UTF-8, más el salto de `echo`. Cuenta bytes, no letras.
- Desde el teclado, la entrada se termina con **Ctrl+D**.

Lo que ve `getchar`

```
build/count_bytes < cities.txt
```



count llega a 17: 14 letras y 3 saltos de línea

EOF no es un byte del archivo: getchar lo devuelve cuando ya no quedan.

```
echo ñ | build/count_bytes
```



count llega a 3: en UTF-8, la ñ ocupa 2 bytes,

y echo agrega un salto de línea. Cuenta bytes, no letras.

Procesar lo que se lee

to_upper.c

```c
int c = getchar();
while (c != EOF) {
    if (c >= 'a' && c <= 'z') {
        c = c - 'a' + 'A';
    }
    putchar(c);
    c = getchar();
}
```

terminal

```
$ echo hola mundo | build/to_upper
HOLA MUNDO
$ echo ñandú | build/to_upper
ñANDú
```

- `putchar(c)` escribe un byte.
- Funciona porque en ASCII las letras van seguidas.
- Los bytes de `ñ` y `ú` no están en el rango: solo sirve para ASCII.

La forma corta, y sus paréntesis

copy.c

```
1  while ((c = getchar()) != EOF) {
2      putchar(c);
3  }
```

Se lee de adentro hacia afuera: trae un byte, lo guarda en `c`, y compara ese valor con `EOF`.

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/getchar_no_parens.c -o build/getchar_no_parens
errors/getchar_no_parens.c:8:12: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
    8 |     while (c = getchar() != EOF) {
$ build/getchar_no_parens < cities.txt
0 saltos de línea
```

Sin paréntesis interiores, `c` recibe el resultado de la comparación: 1, nunca un salto de línea.

`c` debe ser `int`, nunca `char`

Por qué

`getchar` tiene que poder devolver los 256 valores de un byte **y además** `EOF`. Eso no cabe en un `char`.

El estándar no dice si `char` tiene signo. Con `char c`:

Donde	Qué pasa
`char` sin signo (Linux sobre ARM)	`c` nunca es negativo, nunca es igual a `EOF`: el ciclo no termina. `gcc` avisa con `-Wextra`.
`char` con signo (Linux sobre x86-64)	Parece funcionar y `gcc` no avisa, pero un byte 0xFF se confunde con `EOF` y la lectura para antes.

SECCIÓN 08

Laboratorio

08

Cuatro programas

1. **Suma, promedio y mayor** hasta un 0. Centinela con acumulador, contador y mayor. Si el primero es 0: `Sin datos`, no dividan entre 0.
2. **¿Es primo?** Dejen de buscar al encontrar un divisor. 0, 1 y los negativos no son primos.
3. **Contar líneas** con `getchar` y `<`. Se cuentan los `'\n'`, igual que `wc -l`.
4. **Validar una nota** con `do while`. Si `scanf` falla, avisen y terminen.

> **Cómo se prueba**
>
> `make test-lab` falla hasta que terminen. Un ciclo infinito también hace fallar la prueba.

¿Es primo? Sin overflow

- Si `n` es `a * b`, uno de los dos factores es a lo sumo $\sqrt{n}$. No hace falta llegar a `n`.
- La condición natural es `d * d <= n`. Con 2147483647, `d * d` da overflow cuando `d` llega a 46341.
- Sin multiplicar: `d <= n / d` dice lo mismo con enteros positivos.
- Pruébenlo con 0, 1, 2, -7, 49 y 2147483647. Los cuadrados perfectos son el borde.

Para la próxima clase

- Hoy, en el laboratorio: los cuatro programas.
- Leer los apuntes de la semana 6: ciclos anidados y números aleatorios.
- Antes de correr un ciclo: inicio, parada, avance, y cero vueltas.

Preguntas

¿Cuántas veces corre el cuerpo de `for (i = 1; i <= 10; i += 3)`? Reescríbanlo con `while`.
¿Qué hace `while (n > 0) { printf("%d", n); }`? ¿Cómo se corrige?
Escriban en pseudocódigo un algoritmo que cuente los dígitos de un número.