Programación

Del primer programa a divide y vencerás

Facultad de Matemática y Computación, Universidad de La Habana

Índice

Prefacio 5

Recursos básicos y programación estructurada 7

Lo básico de Python 8

1. Qué es un programa 8
2. Expresiones y tipos 8
3. Instrucciones y variables 10
4. Escribir en la consola 13
5. Leer de la consola 14
6. La biblioteca `math` 15
7. Resumen 17

Ejercicios 17

Condicionales y ciclos 19

1. Decidir 19
2. Condiciones compuestas 24
3. Repetir 25
4. Adivina el número 28
5. Acumuladores y el ciclo contado 31
6. Le toca a la computadora 33
7. Resumen 17

Ejercicios 17

Funciones 37

1. Un programa sin nombre 37
2. Parámetros y argumentos 39
3. `return` 40
4. Componer funciones 43
5. Mejorar en un solo lugar 44
6. Alcance de las variables 47
7. Valores por defecto y argumentos con nombre 49
8. Los docstrings 50
9. Lo que falta 50
10. Resumen 51

Ejercicios 17

Listas 53

1. La lista 53
2. Recorrer una lista 55
3. Modificar una lista 57
4. Rebanadas 60
5. El proyecto: las notas del grupo 61
6. Tuplas 63

7. Listas de listas 66
8. La criba de Eratóstenes 67
9. Resumen 71
Ejercicios 17

Cadenas y archivos **74**
1. Una cadena es una secuencia 74
2. Métodos de cadena 76
3. Archivos 78
4. Cuando las cosas salen mal 80
5. El proyecto: contar las palabras 81
6. Resumen 83
Ejercicios 17

Diccionarios y conjuntos **85**
1. El problema, con lo que ya sabemos 85
2. El diccionario 86
3. Recorrer un diccionario 88
4. El patrón de conteo 89
5. Las diez más comunes 91
6. Conjuntos 92
7. Por qué buscar una clave no cuesta nada 95
8. El proyecto completo 97
9. Resumen 71
Ejercicios 17

Búsqueda, ordenación y costo **100**
1. Búsqueda lineal 100
2. Búsqueda binaria 101
3. Ordenación por selección 105
4. Ordenación por inserción 106
5. Ordenación por intercambio 109
6. Contar en lugar de cronometrar 111
7. El reloj 112
8. El proyecto: las palabras por frecuencia 114
9. Resumen 71
Ejercicios 17

Estrategias de solución y recursividad **119**

Recursión **120**
1. Una función que se llama a sí misma 120
2. La búsqueda binaria, otra vez 121
3. La pila de llamadas 122
4. Recursión de cola 123
5. El límite de la pila 125
6. Más de una llamada: las Torres de Hanói 126
7. Recursión sobre listas y cadenas 128

8. Fibonacci y el reloj 130
9. Resumen 71
Ejercicios 17

Divide y vencerás 134
1. Mezclar dos listas ordenadas 134
2. Ordenación por mezcla 135
3. Por qué es tan rápida 136
4. El esquema 138
5. Ordenación rápida 138
6. El pivote 140
7. El reloj 112
8. Resumen 142
Ejercicios 17

Divide y vencerás, cuando no es evidente 145
1. La mejor racha 145
2. Partir por la mitad 146
3. Devolver más de lo que se pide 148
4. La potencia rápida 150
5. Contar inversiones 153
6. La pregunta 155
7. Resumen 17
Ejercicios 17

Prefacio

Este libro reúne las conferencias del primer semestre de Programación en la Facultad de Matemática y Computación de la Universidad de La Habana, para las carreras de Ciencia de la Computación y Ciencia de Datos. Cada capítulo es una conferencia de una semana, y se lee en el mismo orden en que se imparte.

No supone ningún conocimiento previo de programación. Sí supone el preuniversitario: aritmética, álgebra elemental, la ecuación de segundo grado, funciones y el lenguaje de conjuntos. El lenguaje es Python.

Cada capítulo se construye alrededor de un problema concreto: una calculadora de ecuaciones de segundo grado, los primos hasta un tope, las palabras de un capítulo del Quijote, un millón de números que ordenar. El programa que lo resuelve se escribe por partes, se rompe a propósito para ver qué error da, y se arregla. Por eso conviene leer con Python abierto y escribir cada ejemplo en lugar de copiarlo.

Todo el código del libro se ejecuta cada vez que el libro se construye. Lo que aparece debajo de cada bloque es lo que imprimió ese bloque, y las figuras que ilustran un algoritmo se dibujan con el estado que ese algoritmo calculó. Los tiempos que se miden dependen de la máquina en que se construyó el libro: en la tuya serán otros, pero las razones entre ellos, que es lo que el texto discute, deberían parecerse. Los archivos de datos que usan los ejemplos están en la carpeta `conferences/2026/libro` del repositorio del curso.

El libro tiene dos partes, que son los dos primeros temas del programa de la asignatura. La primera, de recursos básicos y programación estructurada, va de las expresiones y los tipos hasta la búsqueda, la ordenación y el costo de un algoritmo. La segunda, de estrategias de solución, empieza con la recursión y llega hasta divide y vencerás. Backtrack, combinatoria y programación dinámica completan ese segundo tema y se agregarán como capítulos nuevos cuando se impartan.

El libro se lee en línea en https://matcom.github.io/programming/ y se descarga completo en PDF desde https://matcom.github.io/programming/programacion.pdf. Las dos versiones salen del mismo texto y se actualizan juntas.

Cada capítulo termina con un resumen y una lista de ejercicios. Los ejercicios son parte del curso: varios retoman en capítulos posteriores funciones que se dejaron para escribir allí.

PARTE I

Recursos básicos y programación estructurada

1

Lo básico de Python

En este capítulo escribimos nuestros primeros programas en Python. Al terminarlo vas a saber escribir un programa que le pide datos al usuario, hace cuentas con ellos y muestra el resultado. Ese programa va a ser una calculadora de las raíces de una ecuación de segundo grado, y la vamos a construir pieza a pieza durante la hora.

1. Qué es un programa

Un programa es un texto con instrucciones que la computadora ejecuta en orden, de arriba hacia abajo. En Python ese texto se guarda en un archivo con extensión `.py`. Creamos el archivo `hola.py` con una sola línea:

```
print("Hola, MatCom")
```

y lo ejecutamos desde la terminal:

```
$ python hola.py
Hola, MatCom
```

Python también tiene un modo interactivo, la consola o REPL. Si escribes `python` sin ningún archivo, aparece el símbolo `>>>` y cada línea que escribas se ejecuta en el momento. La consola sirve para probar cosas pequeñas. Los programas de verdad van en archivos.

```
$ python
>>> print("Hola")
Hola
>>> exit()
```

2. Expresiones y tipos

Una expresión es un fragmento de código que produce un valor. La consola evalúa cada expresión que escribes y muestra su valor, así que funciona como calculadora. Antes de mirar cada resultado, intenta adivinarlo.

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
>>> 7 / 2
3.5
```

```
>>> 7 // 2
3
>>> 7 % 2
1
>>> 2 ** 10
1024
```

Los operadores aritméticos de Python son estos:

OPERADOR	OPERACIÓN	EJEMPLO	RESULTADO
`+`	suma	`17 + 5`	`22`
`-`	resta	`17 - 5`	`12`
`*`	multiplicación	`17 * 5`	`85`
`/`	división	`17 / 5`	`3.4`
`//`	división entera	`17 // 5`	`3`
`%`	resto de la división	`17 % 5`	`2`
`**`	potencia	`17 ** 2`	`289`

La precedencia es la de las matemáticas. `**` va primero, después `*`, `/`, `//` y `%`, y al final `+` y `-`. Si tienes dudas, usa paréntesis.

Tipos

Todo valor tiene un tipo. Por ahora nos interesan tres:

- `int`, los números enteros: `7`, `-3`, `0`.
- `float`, los números reales con punto decimal: `3.5`, `-0.25`, `10.0`.
- `str`, el texto (en inglés *string*), siempre entre comillas: `"hola"`, `'7'`.

La función `type` dice el tipo de un valor:

```
>>> type(7)
<class 'int'>
>>> type(7 / 1)
<class 'float'>
>>> type("7")
<class 'str'>
```

Hay tres detalles que conviene ver con calma.

La división `/` siempre devuelve un `float`, aunque la división sea exacta. `10 / 2` vale `5.0` y no `5`. Si quieres un entero, usa `//`.

Los enteros de Python no tienen límite de tamaño. `2 ** 100` da `1267650600228229401496703205376` sin ningún problema.

Los `float` son aproximados, porque la computadora guarda los números en binario y `0.1` no tiene representación exacta en binario:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Por eso nunca se compara si dos `float` son exactamente iguales. Lo retomaremos más adelante en el curso.

Los operadores también funcionan con texto, con otro significado. `+` concatena y `*` repite:

```
>>> "Hola, " + "mundo"
'Hola, mundo'
>>> "ja" * 3
'jajaja'
```

3. Instrucciones y variables

Una instrucción es una orden completa: hace algo, pero no necesariamente produce un valor. `print("Hola")` es una instrucción. Un programa es una secuencia de instrucciones que se ejecutan una detrás de otra.

La diferencia entre la consola y un archivo importa aquí. En la consola, si escribes la expresión `2 + 3`, ves `5`. Si pones esa misma línea sola dentro de un archivo `.py`, Python la calcula y no muestra nada. En un programa, lo que quieras ver hay que imprimirlo con `print`.

Variables

Una variable es un nombre que guarda un valor. Se crea con una asignación:

```
a = 1
b = -3
c = 2
discriminante = b ** 2 - 4 * a * c
print(discriminante)
```

```
1
```

El signo = significa "calcula lo que está a la derecha y guárdalo con el nombre de la izquierda". No es la igualdad de las matemáticas.

Un programa se ejecuta en el tiempo

Las instrucciones se ejecutan una a una, y cada variable guarda su valor actual, el que recibió en la última asignación. Mira este programa y decide, antes de seguir leyendo, si imprime `10` o `20`:

```
a = 5
b = a * 2
a = 10
print(b)
```

Para responder, seguimos la memoria paso a paso. Cada fila de la tabla muestra el valor de cada variable después de ejecutar esa instrucción, y la casilla en negrita es la que cambió:

INSTRUCCIÓN EJECUTADA	a	b
`a = 5`	**5**	no existe
`b = a * 2`	5	**10**
`a = 10`	**10**	10
`print(b)`	10	10

En la segunda fila, Python lee el valor actual de `a`, que es 5, calcula `5 * 2` y guarda `10` en `b`. La expresión `a * 2` sirve para calcular ese valor y después se descarta. En `b` queda el número 10 y nada más: `b` no recuerda que salió de `a`. Por eso, cuando `a` cambia en la tercera fila, `b` sigue valiendo 10.

```
a = 5
b = a * 2
a = 10
print(b)
```

```
10
```

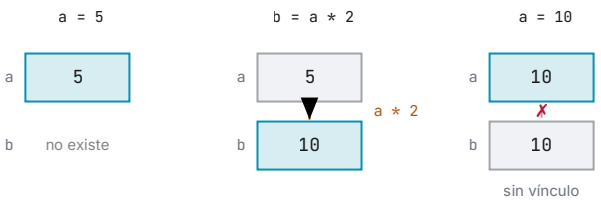


Figura 1. *La flecha existe solo en el instante de la asignación. Cuando* `a` *cambia, ya no queda nada que una las dos casillas.*

Si = fuera una igualdad matemática, `b = a * 2` tendría que seguir siendo cierta después de cambiar `a`, y el programa imprimiría 20. Es una asignación: ocurre una vez, en un momento del tiempo, con los valores que había en ese momento.

Con esa idea, esta línea tiene sentido en Python aunque no lo tenga en álgebra:

```
contador = 10
contador = contador + 1
print(contador)
```

```
11
```

INSTRUCCIÓN EJECUTADA	`contador`
`contador = 10`	**10**
`contador = contador + 1`	**11**
`print(contador)`	11

La segunda línea lee el valor actual de `contador` (10), le suma 1 y guarda el resultado (11) en la misma variable. El valor viejo se pierde.

Las reglas para nombrar variables son pocas. El nombre puede tener letras, dígitos y `_`, no puede empezar con un dígito y no puede ser una palabra reservada de Python como `if` o `for`. Python distingue mayúsculas de minúsculas, así que `area` y `Area` son dos variables distintas. La costumbre en Python es escribir los nombres en minúsculas y separar las palabras con `_`, como en `precio_total`.

Usar una variable que todavía no existe es un error:

```
print(radio)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'radio' is not defined
```

Lee el mensaje de error de abajo hacia arriba. La última línea dice qué pasó (`NameError`, el nombre `radio` no está definido) y las de arriba dicen dónde.

4. Escribir en la consola

`print` recibe uno o más valores, los escribe separados por un espacio y termina con un salto de línea:

```
x = 7
print("x vale", x, "y su cuadrado es", x ** 2)
```

```
x vale 7 y su cuadrado es 49
```

Dos parámetros opcionales cambian ese comportamiento. `sep` es el separador entre valores y `end` es lo que se escribe al final:

```
print(2026, 9, 16, sep="-")
print("sin salto de línea", end=" ")
print("y seguimos en la misma línea")
```

```
2026-9-16
sin salto de línea y seguimos en la misma línea
```

f-strings

La forma más cómoda de mezclar texto y valores es un f-string: un texto con una `f` delante de las comillas, donde todo lo que va entre llaves se evalúa y se inserta.

```
a = 1
b = -3
c = 2
print(f"La ecuación es {a}x² + {b}x + {c} = 0")
print(f"El discriminante vale {b ** 2 - 4 * a * c}")
```

```
La ecuación es 1x² + -3x + 2 = 0
El discriminante vale 1
```

Dentro de las llaves va cualquier expresión. Después de `:` se puede indicar el formato; `.2f` significa "con dos cifras decimales":

```
precio = 1234.5678
print(f"Total: {precio:.2f} pesos")
```

```
Total: 1234.57 pesos
```

5. Leer de la consola

`input` detiene el programa, espera a que el usuario escriba una línea y presione Enter, y devuelve lo que escribió. Si le pasas un texto, lo muestra como pregunta.

```
nombre = input("¿Cómo te llamas? ")
print(f"Hola, {nombre}")
```

```
$ python saludo.py
¿Cómo te llamas? Ana
Hola, Ana
```

Probemos ahora un programa que suma dos números. Antes de ejecutarlo, piensa qué va a imprimir si el usuario escribe `2` y `3`.

```
x = input("Primer número: ")
y = input("Segundo número: ")
print(f"La suma es {x + y}")
```

```
$ python suma.py
Primer número: 2
Segundo número: 3
La suma es 23
```

El programa imprime `23`, porque `input` siempre devuelve un `str`, aunque el usuario escriba dígitos. `x` vale `"2"`, `y` vale `"3"`, y `+` entre dos textos los concatena.

Conversiones

Para operar con los números hay que convertir el texto. Cada tipo tiene una función con su mismo nombre que convierte a ese tipo:

```
print(int("42") + 1)
print(float("2.5") * 2)
print(str(42) + "!")
print(int(3.9))
print(int(-3.9))
```

```
43
5.0
42!
3
-3
```

`int` aplicado a un `float` descarta la parte decimal: no redondea, corta hacia el cero. El programa de la suma queda así:

```
x = int(input("Primer número: "))
y = int(input("Segundo número: "))
print(f"La suma es {x + y}")
```

La conversión falla si el texto no representa un número del tipo pedido. `int("3.5")` también falla, porque `"3.5"` no es un entero escrito en texto:

```
int("3.5")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.5'
```

```
int("hola")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hola'
```

Si el usuario puede escribir decimales, convierte con `float`. Y sumar un texto con un número sin convertir da otro error, esta vez de tipo:

```
"5" + 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

6. La biblioteca `math`

Python trae una biblioteca estándar con muchos módulos listos para usar. Las funciones matemáticas están en el módulo `math`, que hay que importar antes de usarlo:

```
import math

print(math.sqrt(16))
print(math.pi)
print(math.floor(3.7), math.ceil(3.2))
```

```
4.0
3.141592653589793
3 4
```

NOMBRE	QUÉ HACE	EJEMPLO	RESULTADO
`math.sqrt(x)`	raíz cuadrada	`math.sqrt(2)`	`1.4142135623730951`
`math.pi`	la constante π	`math.pi`	`3.141592653589793`
`math.floor(x)`	mayor entero menor o igual que `x`	`math.floor(-3.7)`	`-4`
`math.ceil(x)`	menor entero mayor o igual que `x`	`math.ceil(3.2)`	`4`
`math.sin(x)`, `math.cos(x)`	seno y coseno, `x` en radianes	`math.cos(0)`	`1.0`

Dos funciones parecidas no necesitan `import`, porque son parte del lenguaje: `abs(-5)` da `5`, y `round(3.14159, 2)` da `3.14`.

Compara `math.floor(-3.7)`, que da `-4`, con `int(-3.7)`, que da `-3`. `floor` baja siempre y `int` corta hacia el cero.

La calculadora completa

Ya tenemos todas las piezas. Las raíces de $ax^2 + bx + c = 0$ son

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

y el programa es la fórmula escrita en Python:

```python
import math

a = float(input("a: "))
b = float(input("b: "))
c = float(input("c: "))

discriminante = b ** 2 - 4 * a * c
x1 = (-b + math.sqrt(discriminante)) / (2 * a)
x2 = (-b - math.sqrt(discriminante)) / (2 * a)

print(f"x1 = {x1}")
print(f"x2 = {x2}")
```

```
$ python raices.py
a: 1
b: -3
```

```
c: 2
x1 = 2.0
x2 = 1.0
```

Probemos ahora con $x^2 + 1 = 0$, es decir `a = 1`, `b = 0`, `c = 1`. El discriminante vale -4 y la raíz cuadrada de un número negativo no es un número real:

```
$ python raices.py
a: 1
b: 0
c: 1
Traceback (most recent call last):
  File "raices.py", line 8, in <module>
    x1 = (-b + math.sqrt(discriminante)) / (2 * a)
                ~~~~~~~~~^^^^^^^^^^^^^^^^
ValueError: expected a nonnegative input, got -4.0
```

En versiones de Python anteriores a la 3.14 el mensaje es `ValueError: math domain error`. Para que el programa responda "no tiene raíces reales" en vez de fallar, necesita decidir qué hacer según el valor del discriminante. Eso es lo que vemos en el capítulo siguiente: los condicionales. Si pruebas `a = 0`, vas a encontrar otro error que también se arregla con un condicional.

7. Resumen

- Un programa es una secuencia de instrucciones que se ejecutan en orden.
- Una expresión produce un valor, y todo valor tiene un tipo: `int`, `float`, `str`.
- `=` guarda un valor en una variable; no es la igualdad matemática.
- `print` escribe en la consola, y los f-strings mezclan texto y valores.
- `input` lee de la consola y siempre devuelve un `str`. Para hacer cuentas hay que convertir con `int` o `float`.
- `import math` da acceso a `sqrt`, `pi`, `floor`, `ceil` y el resto de las funciones matemáticas.

Ejercicios

1. Escribe un programa que lea los dos catetos de un triángulo rectángulo e imprima la hipotenusa con dos cifras decimales.
2. Escribe un programa que lea una temperatura en grados Celsius y la imprima en Fahrenheit. La fórmula es $F = \frac{9}{5}C + 32$.
3. Escribe un programa que lea un número de segundos e imprima cuántas horas, minutos y segundos son. Por ejemplo, `3665` debe imprimir `1 h 1 min 5 s`. Usa `//` y `%`.
4. Escribe un programa que lea el radio de un círculo e imprima su área y su perímetro.
5. Escribe un programa que lea un número entero de tres cifras e imprima la suma de sus cifras. Por ejemplo, `472` debe imprimir `13`.

6. Sin ejecutarlo, di qué imprime cada línea. Después compruébalo en la consola.

```
print(17 // 5, 17 % 5, -17 // 5)
print(int("7") * 2, "7" * 2)
print(round(2.5), round(3.5))
```

   Si alguno de los resultados te sorprende, busca en la documentación de Python cómo redondea `round`.

2

Condicionales y ciclos

El capítulo anterior terminó con una calculadora de raíces que funcionaba solo si le dabas los coeficientes correctos. Con `a = 1`, `b = 0`, `c = 1` se rompía, y con `a = 0` también. Aquí la arreglamos. Para eso hace falta que el programa **decida** qué hacer según lo que valen sus datos, y eso son los condicionales. Después veremos cómo hacer que un programa **repita** una parte de su trabajo, que son los ciclos. Con esas dos herramientas, al final del capítulo vamos a construir un juego de adivinar números.

1. Decidir

Hasta ahora todos nuestros programas se ejecutaban igual siempre: la misma secuencia de instrucciones, de arriba hacia abajo, sin importar los datos. Un condicional rompe esa línea recta y permite que un pedazo de código se ejecute o no.

Valores de verdad

Antes de decidir hay que preguntar, y en Python una pregunta es una expresión como cualquier otra: produce un valor.

```
print(3 < 0)
print(3 > 0)
print(type(3 < 0))
```

```
False
True
<class 'bool'>
```

Ese es el cuarto tipo del curso, después de `int`, `float` y `str`. Se llama `bool`, en honor a George Boole, y tiene exactamente dos valores: `True` y `False`. Se escriben con mayúscula inicial y sin comillas, porque no son texto.

Los operadores que producen valores de verdad son las comparaciones:

OPERADOR	PREGUNTA	EJEMPLO	RESULTADO
`==`	¿son iguales?	`3 == 3`	`True`
`!=`	¿son distintos?	`3 != 3`	`False`

OPERADOR	PREGUNTA	EJEMPLO	RESULTADO
`<`	¿menor?	`2 < 3`	`True`
`⩽`	¿menor o igual?	`3 ⩽ 3`	`True`
`>`	¿mayor?	`2 > 3`	`False`
`⩾`	¿mayor o igual?	`2 ⩾ 3`	`False`

Fíjate en `==`, con dos signos. Un solo `=` es la asignación que vimos en el capítulo pasada, que guarda un valor en una variable. Dos `==` preguntan si dos valores son iguales. Son cosas distintas y confundirlas es el error más común de los primeros días.

La instrucción `if`

`if` ejecuta un bloque de código solo cuando una condición es verdadera:

```
discriminante = -4

if discriminante < 0:
    print("La ecuación no tiene raíces reales")
```

```
La ecuación no tiene raíces reales
```

Hay dos cosas nuevas en la sintaxis. El **dos puntos** al final de la línea del `if` abre un bloque. La **indentación** de la línea siguiente, cuatro espacios por convención, dice qué instrucciones pertenecen a ese bloque.

Esto es propio de Python y conviene entenderlo bien desde ahora: en la mayoría de los lenguajes la indentación es cosmética y los bloques se delimitan con llaves. En Python la indentación *es* la sintaxis. Cambiarla cambia el significado del programa, y olvidarla es un error:

```
if 3 < 0:
print("hola")
```

```
  File "<stdin>", line 2
    print("hola")
    ^^^^^
IndentationError: expected an indented block after 'if' statement on line 1
```

El bloque puede tener varias líneas. Lo que está indentado se ejecuta solo si la condición es verdadera; lo que vuelve al margen se ejecuta siempre.

```
x = 7

if x > 5:
    print("x es grande")
    print("bastante grande")

print("fin del programa")
```

```
x es grande
bastante grande
fin del programa
```

Prueba a cambiar `x` a `3` y observa qué desaparece de la salida y qué no.

`else`

`else` da el camino alternativo, el que se toma cuando la condición es falsa. Su bloque también va indentado.

```
discriminante = -4

if discriminante < 0:
    print("No hay raíces reales")
else:
    print("Hay raíces reales")
```

```
No hay raíces reales
```

Siempre se ejecuta uno de los dos bloques, nunca los dos y nunca ninguno.

`elif` y los tres casos de la ecuación

La ecuación de segundo grado no tiene dos casos sino tres, según el signo del discriminante: dos raíces distintas, una raíz doble, o ninguna raíz real. Para encadenar varias condiciones está `elif`, que es la contracción de *else if*:

```
import math

a = 1
b = -2
c = 1
discriminante = b ** 2 - 4 * a * c

if discriminante > 0:
    x1 = (-b + math.sqrt(discriminante)) / (2 * a)
    x2 = (-b - math.sqrt(discriminante)) / (2 * a)
    print(f"Dos raíces: {x1} y {x2}")
elif discriminante == 0:
    print(f"Una raíz doble: {-b / (2 * a)}")
```

```
else:
    print("No tiene raíces reales")
```

```
Una raíz doble: 1.0
```

Python evalúa las condiciones en orden, de arriba hacia abajo. Ejecuta el bloque de la **primera** que resulte verdadera y se salta todas las demás, incluido el `else`. Solo se ejecuta una rama. Puede haber tantos `elif` como haga falta, y el `else` final es opcional.

Que el orden importe tiene una consecuencia práctica: si escribes primero una condición más general y después una más específica, la segunda nunca se ejecuta.

La trampa de = contra ==

Si escribes un solo = dentro de un `if`, Python no te deja ni empezar:

```
discriminante = 0

if discriminante = 0:
    print("raíz doble")
```

```
  File "<stdin>", line 3
    if discriminante = 0:
       ^^^^^^^^^^^^^^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

El mensaje es de los buenos: te dice exactamente qué quisiste escribir.

El otro error: cuando a vale cero

Queda el segundo problema del capítulo anterior. Si `a` vale 0 la ecuación no es de segundo grado, y la fórmula divide por `2 * a`, que es cero:

```
import math

a = 0
b = 2
c = -4
discriminante = b ** 2 - 4 * a * c
x1 = (-b + math.sqrt(discriminante)) / (2 * a)
```

```
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
ZeroDivisionError: division by zero
```

Hay que preguntar por `a` **antes** de aplicar la fórmula. Y si `a` es cero, la ecuación es lineal, `bx + c = 0`, con solución `x = -c/b`, salvo que `b` también sea cero y entonces no hay ecuación ninguna. Eso son condicionales dentro de condicionales, y cada nivel suma cuatro espacios de indentación:

```
a = 0
b = 2
c = -4

if a == 0:
    if b == 0:
        print("Eso no es una ecuación")
    else:
        print(f"Es lineal: x = {-c / b}")
else:
    print("Es de segundo grado")
```

```
Es lineal: x = 2.0
```

La calculadora completa

Juntando todo, la calculadora del capítulo anterior queda así, y ya no se rompe con ninguna entrada:

```
import math

a = float(input("a: "))
b = float(input("b: "))
c = float(input("c: "))

if a == 0:
    if b == 0:
        print("Eso no es una ecuación")
    else:
        print(f"Es lineal: x = {-c / b}")
else:
    discriminante = b ** 2 - 4 * a * c

    if discriminante > 0:
        x1 = (-b + math.sqrt(discriminante)) / (2 * a)
        x2 = (-b - math.sqrt(discriminante)) / (2 * a)
        print(f"Dos raíces: x1 = {x1}, x2 = {x2}")
    elif discriminante == 0:
        print(f"Una raíz doble: x = {-b / (2 * a)}")
    else:
        print("No tiene raíces reales")
```

```
$ python raices.py
a: 1
b: 0
c: 1
No tiene raíces reales

$ python raices.py
a: 0
b: 2
```

```
c: -4
Es lineal: x = 2.0
```

Mira la forma del programa, no su contenido. La indentación dibuja la estructura de la decisión, y un programa bien indentado se lee como un esquema.

2. Condiciones compuestas

Muchas decisiones necesitan más de una condición. Los operadores `and`, `or` y `not` combinan valores de verdad.

```python
edad = 20

print(edad >= 18 and edad < 65)
print(edad < 18 or edad >= 65)
print(not (edad >= 18))
```

```
True
False
False
```

`and` es verdadero solo si ambos lo son. `or` es verdadero si al menos uno lo es. `not` invierte.

p	q	p and q	p or q	not p
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

La precedencia va `not`, después `and`, después `or`, igual que en lógica. Si tienes dudas, usa paréntesis, que además se lee mejor.

Un detalle cómodo de Python: las comparaciones se pueden encadenar como en matemáticas, y significan lo que parece que significan.

```python
x = 7

print(0 <= x <= 10)
print(0 <= x and x <= 10)
```

```
True
True
```

Comparar números reales

Aquí pagamos una deuda del capítulo anterior. Dijimos que los `float` son aproximados y que nunca se comparan con `==`. Ahora que tenemos comparaciones, veamos por qué:

```
print(0.1 + 0.2)
print(0.1 + 0.2 == 0.3)
```

```
0.30000000000000004
False
```

El resultado es matemáticamente correcto y prácticamente inútil. La forma de comparar dos reales es preguntar si están *suficientemente cerca*, y para eso está `math.isclose`:

```
import math

print(math.isclose(0.1 + 0.2, 0.3))
print(math.isclose(1e-15, 0))
print(math.isclose(1e-15, 0, abs_tol=1e-9))
```

```
True
False
True
```

La segunda línea muestra que comparar contra cero es un caso aparte: `isclose` mide la diferencia *relativa* al tamaño de los números, y el cero no tiene tamaño. Para compararse contra cero hay que dar una tolerancia absoluta con `abs_tol`.

Esto afecta a nuestra calculadora. `discriminante == 0` es exactamente la comparación que acabamos de decir que no se hace, porque `a`, `b` y `c` vienen de `float(input(...))`. Funciona mientras el usuario escriba números enteros, que se representan exacto, y deja de funcionar con coeficientes que salgan de una medición. La versión rigurosa escribe `math.isclose(discriminante, 0, abs_tol=1e-9)`.

3. Repetir

La calculadora resuelve una ecuación y se muere. Si quieres resolver diez, la ejecutas diez veces. Para que un programa repita una parte de su trabajo está el ciclo.

El ciclo `while`

`while` repite un bloque **mientras** una condición sea verdadera. La sintaxis es la misma del `if`: dos puntos, bloque indentado.

```
n = 5

while n > 0:
    print(n)
    n = n - 1

print("¡Despegue!")
```

```
5
4
3
2
1
¡Despegue!
```

Todo ciclo `while` tiene tres partes, y conviene aprender a buscarlas:

1. **Inicializar**: `n = 5`, antes del ciclo, deja la variable en su valor de partida.
2. **Condición**: `n > 0`, se evalúa antes de cada vuelta.
3. **Avanzar**: `n = n - 1`, dentro del bloque, acerca la variable al final.

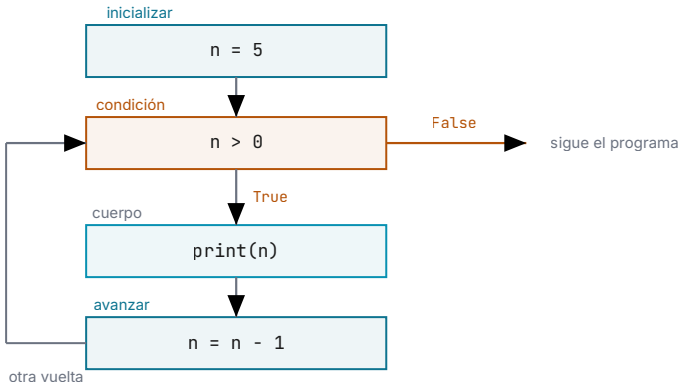


Figura 2. *Si falta la tercera parte, la condición nunca deja de ser cierta y el ciclo no termina.*

Sigamos la memoria como hicimos en el capítulo anterior. Cada fila es una evaluación de la condición:

VUELTA	n AL ENTRAR	n > 0	IMPRIME	n AL SALIR
1	5	True	5	4
2	4	True	4	3
3	3	True	3	2
4	2	True	2	1

VUELTA	n AL ENTRAR	n > 0	IMPRIME	n AL SALIR
5	1	True	1	0
6	0	False	—	—

En la sexta evaluación la condición es falsa, el bloque no se ejecuta y el programa continúa en la línea de después, que imprime `¡Despegue!`.

La línea `n = n - 1` es la misma idea de `contador = contador + 1` del capítulo anterior: lee el valor actual, calcula el nuevo, lo guarda en la misma variable. Como se escribe todo el tiempo, Python trae una abreviatura para cada operador aritmético:

```
n = 10
n -= 1      # igual que n = n - 1
n += 5      # igual que n = n + 5
n *= 2      # igual que n = n * 2
print(n)
```

```
28
```

El ciclo infinito

Si olvidas avanzar, la condición nunca deja de ser verdadera y el programa no termina:

```
n = 5

while n > 0:
    print(n)
```

```
$ python cuenta.py
5
5
5
5
5
^C
Traceback (most recent call last):
  File "cuenta.py", line 4, in <module>
    print(n)
KeyboardInterrupt
```

Eso es un ciclo infinito. Se interrumpe con **Ctrl+C**, que le manda una señal al programa y produce el `KeyboardInterrupt` de arriba. Lo vas a hacer muchas veces este curso, así que acostúmbrate: si el programa no responde y la terminal no vuelve al prompt, Ctrl+C.

4. Adivina el número

Ya tenemos las dos herramientas. Vamos a construir un juego con ellas: la computadora piensa un número entre 1 y 100 y tú tienes que adivinarlo.

Números al azar

El módulo `random` de la biblioteca estándar genera números al azar. Se importa igual que `math`:

```
import random

print(random.randint(1, 100))
print(random.randint(1, 100))
print(random.randint(1, 100))
```

```
82
16
59
```

`random.randint(a, b)` devuelve un entero al azar entre `a` y `b`, ambos incluidos. Si ejecutas el programa otra vez, salen otros números.

Paso 1: un solo intento

La primera versión es el condicional de la sección 1 con otra ropa:

```
import random

secreto = random.randint(1, 100)
intento = int(input("Adivina mi número: "))

if intento == secreto:
    print("¡Acertaste!")
else:
    print(f"No. Era {secreto}")
```

Funciona, pero como juego es pésimo: una oportunidad entre cien.

Paso 2: repetir hasta acertar

Metemos la pregunta en un ciclo que repite mientras el intento sea distinto del secreto:

```
import random

secreto = random.randint(1, 100)
intento = 0
```

```
while intento != secreto:
    intento = int(input("Adivina mi número: "))

print("¡Acertaste!")
```

La inicialización `intento = 0` merece un comentario. El valor 0 no es un intento del jugador: está ahí porque la condición se evalúa **antes** de la primera vuelta y hace falta que sea verdadera. Se escoge 0 justamente porque está fuera del rango 1 a 100, así que nunca puede coincidir con el secreto por accidente. Es un truco que vas a usar mucho.

Y fíjate en algo: la condición del ciclo *es* la comparación que antes estaba en el `if`. Condicionales y ciclos no son dos temas pegados con cinta; un ciclo es un condicional que se vuelve a preguntar.

Paso 3: guiar al jugador

Cien intentos a ciegas siguen siendo aburridos. Le decimos al jugador si se pasó o se quedó corto, con un condicional dentro del ciclo:

```
import random

secreto = random.randint(1, 100)
intento = 0

while intento != secreto:
    intento = int(input("Adivina mi número: "))

    if intento < secreto:
        print("Mi número es mayor")
    elif intento > secreto:
        print("Mi número es menor")

print("¡Acertaste!")
```

```
$ python adivina.py
Adivina mi número: 50
Mi número es mayor
Adivina mi número: 75
Mi número es menor
Adivina mi número: 62
Mi número es mayor
Adivina mi número: 68
¡Acertaste!
```

El `elif` no tiene `else`, y está bien: cuando ninguna de las dos condiciones se cumple es porque el jugador acertó, y de eso se encarga la condición del ciclo.

Paso 4: contar los intentos y ponerles un límite

Para contar cuántas veces intentó el jugador hace falta una variable que empiece en cero y suba una unidad por vuelta. Y ya que llevamos la cuenta, démosle siete intentos y ni uno más, lo que convierte la condición del ciclo en una condición compuesta:

```python
import random

secreto = random.randint(1, 100)
intento = 0
intentos = 0

while intento != secreto and intentos < 7:
    intento = int(input("Adivina mi número: "))
    intentos += 1

    if intento < secreto:
        print("Mi número es mayor")
    elif intento > secreto:
        print("Mi número es menor")

if intento == secreto:
    print(f"¡Acertaste en {intentos} intentos!")
else:
    print(f"Se acabaron los intentos. Era {secreto}")
```

Aparece algo nuevo al final. El ciclo ahora puede terminar por dos razones distintas, así que después de salir hay que **preguntar por cuál de las dos fue**. Ese `if` de después del ciclo no es un detalle: cada vez que la condición de un `while` tenga un `and`, vas a necesitarlo.

`break`

Hay otra manera de escribir lo mismo. `break` interrumpe el ciclo en el acto, desde dentro del bloque:

```python
import random

secreto = random.randint(1, 100)
intentos = 0

while True:
    intento = int(input("Adivina mi número: "))
    intentos += 1

    if intento == secreto:
        print(f"¡Acertaste en {intentos} intentos!")
        break

    if intentos == 7:
        print(f"Se acabaron los intentos. Era {secreto}")
        break

    if intento < secreto:
        print("Mi número es mayor")
    else:
        print("Mi número es menor")
```

`while True` es un ciclo cuya condición es verdadera siempre, es decir, un ciclo infinito a propósito. Lo que lo termina son los `break`. La versión con `break` hace visible cada razón de salida en el lugar donde ocurre, y evita el `if` de después. La versión anterior pone toda la lógica de terminación en la condición del `while`, donde se lee de un golpe. Ninguna de las dos es mejor siempre; escoge según cuál explique mejor tu programa.

5. Acumuladores y el ciclo contado

El patrón acumulador

Contar los intentos es un caso de algo que se repite en todo el curso: una variable que empieza en un valor neutro y se actualiza en cada vuelta hasta contener el resultado. Se llama **acumulador**.

Sumar los números de 1 a n:

```
n = 10
suma = 0
i = 1

while i <= n:
    suma += i
    i += 1

print(suma)
```

```
55
```

Multiplicarlos, que es el factorial:

```
n = 5
producto = 1
i = 1

while i <= n:
    producto *= i
    i += 1

print(producto)
```

```
120
```

La única diferencia está en el valor inicial y en la operación. El valor neutro de la suma es 0 y el del producto es 1, porque son los que no alteran el resultado. Empezar el producto en 0 daría 0 siempre, y es un error clásico.

`for` y `range`

Los dos ciclos anteriores hacen algo muy común: recorrer los enteros de un rango. Para eso Python tiene una forma más corta:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

`range(5)` produce los enteros desde 0 hasta 4. Empieza en cero y **termina antes** del número que le das, que es la convención de casi todo en programación y cuesta acostumbrarse. `range` admite hasta tres argumentos:

FORMA	QUÉ PRODUCE
`range(n)`	`0, 1, ..., n-1`
`range(a, b)`	`a, a+1, ..., b-1`
`range(a, b, p)`	`a, a+p, a+2p, ...` sin llegar a `b`

```
for i in range(2, 7):
    print(i, end=" ")
print()

for i in range(0, 10, 3):
    print(i, end=" ")
print()

for i in range(10, 0, -2):
    print(i, end=" ")
print()
```

```
2 3 4 5 6
0 3 6 9
10 8 6 4 2
```

La suma de 1 a `n` con `for` queda así:

```
n = 10
suma = 0

for i in range(1, n + 1):
    suma += i

print(suma)
```

```
55
```

Compara las dos versiones. El `for` pone las tres partes del ciclo en una sola línea: inicializa `i`, lo compara y lo avanza por ti. Por eso es imposible olvidar el avance y provocar un ciclo infinito con un `for`. Lo que ganas en seguridad lo pagas en flexibilidad, porque el avance es siempre el mismo.

La regla para escoger es corta: **si sabes cuántas vueltas son, `for`; si depende de algo que pasa dentro del ciclo, `while`.** El juego de adivinar es `while`, porque nadie sabe cuántos intentos va a necesitar el jugador. La suma de 1 a `n` es `for`.

`break` también funciona en un `for`. Y hay un pariente suyo, `continue`, que no sale del ciclo sino que salta a la vuelta siguiente:

```
for i in range(1, 11):
    if i % 2 == 0:
        continue
    print(i, end=" ")
```

```
1 3 5 7 9
```

6. Le toca a la computadora

Invirtamos el juego. Tú piensas un número entre 1 y 100, y la computadora pregunta.

La estrategia obvia es un ciclo que pregunte «¿es 1?», «¿es 2?», «¿es 3?» hasta acertar. Funciona, y en el peor caso pregunta cien veces. La computadora estaría jugando mucho peor que tú, porque tú no ibas probando en orden.

La estrategia buena es la que usa cualquiera: mantener un rango donde el número tiene que estar, y preguntar por su mitad. Cada respuesta descarta la mitad del rango.

```
bajo = 1
alto = 100
preguntas = 0

while bajo < alto:
    medio = (bajo + alto) // 2
    preguntas += 1
    respuesta = input(f"¿Tu número es mayor que {medio}? (s/n) ")

    if respuesta == "s":
        bajo = medio + 1
    else:
        alto = medio

print(f"Tu número es {bajo}, y me tomó {preguntas} preguntas")
```

```
$ python adivina_ella.py
¿Tu número es mayor que 50? (s/n) s
¿Tu número es mayor que 75? (s/n) n
¿Tu número es mayor que 62? (s/n) s
```

```
¿Tu número es mayor que 68? (s/n) n
¿Tu número es mayor que 65? (s/n) s
¿Tu número es mayor que 66? (s/n) s
¿Tu número es mayor que 67? (s/n) n
Tu número es 68, y me tomó 7 preguntas
```

El ciclo termina cuando `bajo` y `alto` se juntan, porque entonces solo queda un número posible. Y el rango se encoge a la mitad cada vez.

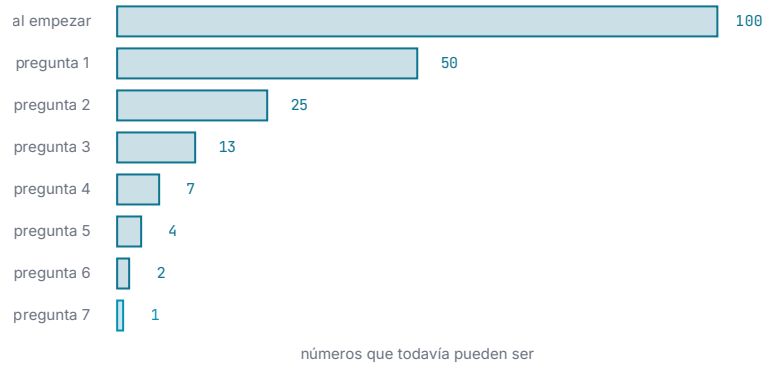


Figura 3. *Cada pregunta descarta la mitad de lo que quedaba. Siete preguntas bastan para cien números; con un millón serían veinte.*

Siete preguntas en el peor caso, contra cien de la versión anterior. Con un millón de números serían veinte.

Eso se llama **búsqueda binaria**, y es uno de los algoritmos más importantes que vas a aprender. Vuelve, con nombre y apellidos, en el tema de recursión.

7. Resumen

- Una comparación produce un valor de tipo `bool`, que vale `True` o `False`.
- `=` asigna, `==` compara. No son lo mismo.
- `if`, `elif` y `else` escogen un camino. Los dos puntos abren el bloque y la indentación dice qué instrucciones lo forman.
- `and`, `or` y `not` combinan condiciones.
- Dos `float` nunca se comparan con `==`, se comparan con `math.isclose`.
- `while` repite mientras una condición sea verdadera. Tiene tres partes: inicializar, condición y avanzar. Si falta el avance, el ciclo es infinito y se sale con Ctrl+C.
- Un acumulador es una variable que empieza en un valor neutro y se actualiza en cada vuelta.
- `for i in range(...)` es la forma corta cuando sabes cuántas vueltas son.
- `break` sale del ciclo y `continue` salta a la vuelta siguiente.

Ejercicios

1. El penúltimo dígito del carné de identidad indica el sexo: par es masculino, impar es femenino. Escribe un programa que lea un carné como número entero e imprima el sexo. No uses `str`: con `//` y `%` alcanza.
2. Escribe un programa que lea tres enteros e imprima qué tipo de triángulo forman: `ninguno` si no son los lados de un triángulo, `escaleno`, `isósceles` o `equilátero`. Recuerda que cada lado tiene que ser menor que la suma de los otros dos.
3. Escribe un programa que lea un entero e imprima cuántos dígitos tiene, sin usar `str`. Piensa qué le pasa a un número cuando lo divides entre 10.
4. La secuencia de Collatz empieza en un entero `n` y sigue la regla: si el número es par, se divide entre 2; si es impar, se multiplica por 3 y se le suma 1. Termina al llegar a 1. Escribe un programa que lea `n` e imprima la secuencia completa y su longitud. Para `n = 17` la secuencia es `17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1`. Nadie ha demostrado todavía que este programa siempre termine.
5. Un número es **perfecto** si es igual a la suma de sus divisores propios. Por ejemplo, `28 = 1 + 2 + 4 + 7 + 14`. Escribe un programa que lea un entero y diga si es perfecto. Después imprime todos los números perfectos menores que 10000, y observa cuánto tarda.
6. La sucesión de Fibonacci empieza con 0 y 1, y cada término es la suma de los dos anteriores. Escribe un programa que lea `n` e imprima el término `n`-ésimo. Usa un ciclo y dos acumuladores; no hacen falta más.
7. Sin ejecutarlos, di qué imprime cada uno de estos tres programas. Después compruébalo.

   (a)

```
for i in range(3):
    for j in range(3):
        if i == j:
            print(i, j)
```

   (b)

```
n = 0
while n < 5:
    n += 2
print(n)
```

   (c)

```
x = 5
if x > 0:
    print("positivo")
if x > 3:
    print("mayor que tres")
else:
    print("otro")
```

El tercero es el interesante: cámbiale el segundo `if` por un `elif` y explica por qué la salida cambia.

3

Funciones

Un número entero mayor que 1 es **primo** si no se puede dividir exactamente entre ningún otro salvo 1 y él mismo. Decidirlo para un número dado está al alcance de lo que ya sabes: un ciclo, un `if` y el operador `%`. Lo que no está al alcance todavía es usar ese programa dentro de otro, y por eso en este capítulo aprendemos a ponerle nombre a un pedazo de programa.

Al final del capítulo vamos a tener un programa que imprime todos los primos hasta un número dado y otro que los cuenta, los dos construidos sobre la misma pieza. Y vamos a dejar planteado, sin poder escribirlo todavía, un método mucho mejor.

1. Un programa sin nombre

Empecemos por el problema de un solo número. ¿Es 91 primo? Con lo del capítulo anterior se escribe así:

```
n = 91
primo = True

for d in range(2, n):
    if n % d == 0:
        primo = False

print(primo)
```

```
False
```

Es el patrón acumulador con un `bool` en lugar de un número: `primo` empieza en `True`, y cualquier divisor que aparezca lo tumba a `False`. El ciclo prueba todos los enteros desde 2 hasta `n - 1`, que son todos los candidatos posibles a divisor.

El programa está bien y sirve de poco. Para probar otro número hay que editar la primera línea y volver a correrlo. Y si quisieras los primos menores que cien, necesitarías meter este programa dentro de otro ciclo cien veces, y no hay manera de hacerlo: **el programa no tiene nombre, así que no se puede mencionar.**

Una función es exactamente eso, un nombre para un pedazo de programa. Se define con `def`, un nombre, unos paréntesis y dos puntos, y el cuerpo va indentado igual que el de un `if` o un `while`:

```
def es_primo(n):
    primo = True

    for d in range(2, n):
        if n % d == 0:
            primo = False

    return primo

print(es_primo(91))
print(es_primo(97))
```

```
False
True
```

Las mismas seis líneas de antes, y ahora `es_primo` es una palabra del vocabulario del programa. `91` es `7 * 13`, así que la primera respuesta es `False`; `97` sí es primo.

Hay dos momentos distintos aquí y conviene separarlos desde ahora. La **definición** es el bloque del `def`: le dice a Python qué significa ese nombre, y no ejecuta nada. La **llamada** es `es_primo(91)`, con los paréntesis: ahí es donde el cuerpo corre de verdad. Definir una función y no llamarla nunca es perfectamente legal:

```
def es_primo(n):
    print("me llamaron")

print("el programa terminó")
```

```
el programa terminó
```

Los paréntesis son obligatorios para llamar. Sin ellos no hay error, pero tampoco pasa nada útil, porque el nombre a secas se refiere a la función misma y no a su resultado:

```
def es_primo(n):
    return n == 2

print(es_primo)
```

```
<function es_primo at 0x777e72d2b5e0>
```

Eso que imprime es la función como valor. Aquí no lo vamos a usar, pero recuérdalo: significa que una función es un dato más, como un número o un texto, y sobre esa idea está construida media biblioteca estándar.

Antes de seguir, una advertencia sobre nuestra `es_primo`: dice que `1` es primo, y dice que `0` también. Los dos son mentira. Lo arreglamos en la sección 3.

2. Parámetros y argumentos

`n` es un **parámetro**: una variable que se escribe entre los paréntesis de la definición y que recibe un valor distinto en cada llamada. Puede haber varios, separados por comas, y se emparejan por posición con los valores de la llamada:

```
def es_divisible(n, d):
    return n % d == 0

print(es_divisible(91, 7))
print(es_divisible(91, 5))
```

```
True
False
```

Esa función chiquita nos va a servir después. De momento fíjate en dos palabras que se confunden todo el tiempo y que vas a oír en los exámenes. **Parámetro** es el nombre que aparece en la definición: `n`, `d`. **Argumento** es el valor concreto que se pasa en la llamada: `91`, `7`. El parámetro es el hueco y el argumento es lo que se le mete.

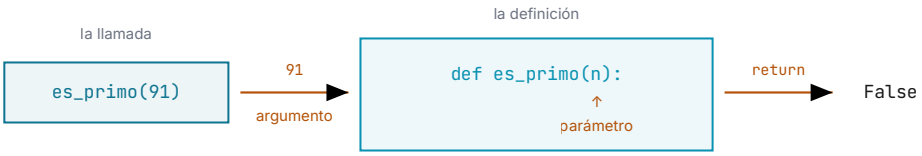


Figura 4. *La definición abre un hueco y cada llamada le mete un valor distinto.*

Como el emparejamiento es por posición, el orden importa y equivocarlo no da ningún error, solo una respuesta falsa:

```
def es_divisible(n, d):
    return n % d == 0

print(es_divisible(7, 91))
```

```
False
```

`7 % 91` es 7, que no es cero, así que la respuesta es `False`. Lo que preguntamos sin darnos cuenta fue si 7 es divisible entre 91.

Si el número de argumentos no coincide con el de parámetros, ahí sí Python se queja, y se queja antes de ejecutar nada del cuerpo:

```
def es_divisible(n, d):
    return n % d == 0

print(es_divisible(91))
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: es_divisible() missing 1 required positional argument: 'd'
```

3. `return`

`return` es la instrucción que entrega un valor a quien llamó. Es lo más útil que puede hacer una función, porque el que llama se queda con el resultado y hace con él lo que quiera:

```
def es_divisible(n, d):
    return n % d == 0

if es_divisible(100, 4) and es_divisible(100, 25):
    print("100 se divide entre 4 y entre 25")
```

```
100 se divide entre 4 y entre 25
```

Recupera la distinción del capítulo 1: **una función con `return` es una expresión**, y por eso se puede meter dentro de un `and`, de una suma o de otra llamada.

Imprimir no es devolver

La diferencia entre imprimir y devolver es la que más trabajo cuesta al principio, así que vamos a verla de frente. Estas dos funciones parecen hacer lo mismo:

```
def divisible_imprime(n, d):
    print(n % d == 0)

def divisible_devuelve(n, d):
    return n % d == 0

divisible_imprime(100, 4)
print(divisible_devuelve(100, 4))
```

```
True
True
```

La salida es idéntica. Pero intenta contar con ellas los divisores de 100:

```
def divisible_imprime(n, d):
    print(n % d == 0)
```

```python
def divisible_devuelve(n, d):
    return n % d == 0

divisores = 0

for d in range(2, 11):
    if divisible_devuelve(100, d):
        divisores += 1

print(f"la que devuelve encontró {divisores} divisores")

print(divisible_imprime(100, 4) + divisible_imprime(100, 5))
```

```
la que devuelve encontró 4 divisores
True
True
Traceback (most recent call last):
  File "<stdin>", line 15, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

La que imprime no sirve para calcular. Escribe en la pantalla y no deja nada. La regla práctica: **una función que calcula algo lo devuelve; imprimir es trabajo del programa principal.** Así puedes cambiar la presentación sin tocar el cálculo, y eso es justamente lo que vamos a hacer con `es_primo` en un rato.

Funciones que no devuelven nada

¿Qué valor tiene entonces `divisible_imprime(100, 4)`? El error de arriba lo dice: `None`, que es lo que Python devuelve cuando no le dices otra cosa. `None` es un valor de pleno derecho, con su propio tipo, y significa «nada»:

```python
def divisible_imprime(n, d):
    print(n % d == 0)

resultado = divisible_imprime(100, 4)
print(resultado)
print(type(resultado))
```

```
True
None
<class 'NoneType'>
```

Una función sin `return` no está mal escrita. `print` misma es una de ellas. Lo que está mal es esperar un valor de una función que no devuelve ninguno.

`return` termina la función

En cuanto se ejecuta un `return`, la función se acaba y lo que venga después no corre. Eso arregla los dos problemas que dejamos abiertos:

```
def es_primo(n):
    if n < 2:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True

print(es_primo(0), es_primo(1), es_primo(2), es_primo(91), es_primo(97))
```

```
False False True False True
```

Las tres líneas con `return` hacen tres trabajos distintos. La primera es un **filtro de entrada**: 0, 1 y los negativos no son primos, y la función lo dice y se va sin entrar al ciclo. La segunda está dentro del ciclo, y ahí está lo bueno: en el momento en que aparece un divisor ya no hay nada más que averiguar, así que la función contesta y sale. No hace falta `break`, porque `return` sale del ciclo y de la función a la vez.

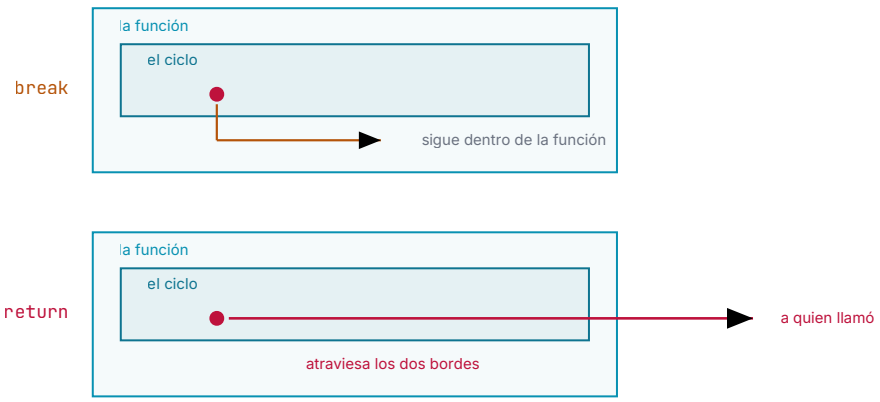


Figura 5. `break` *termina el ciclo y el programa sigue dentro de la función.* `return` *atraviesa los dos bordes de una vez.*

La tercera solo se alcanza si el ciclo terminó sin encontrar nada.

Compara esa versión con la de la sección 1. La de la sección 1 encuentra el divisor 7 de 91 y sigue probando hasta el 90, con la respuesta ya decidida. Cuánto cuesta esa terquedad se ve poniéndole el reloj. `time.perf_counter()` devuelve un número de segundos que solo sirve para restarlo de otro:

```
import time

def con_bandera(n):
```

```
    if n < 2:
        return False

    primo = True

    for d in range(2, n):
        if n % d == 0:
            primo = False

    return primo

def con_return(n):
    if n < 2:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True

inicio = time.perf_counter()
print(con_bandera(1000000), f"{time.perf_counter() - inicio:.6f} s")

inicio = time.perf_counter()
print(con_return(1000000), f"{time.perf_counter() - inicio:.6f} s")
```

```
False 0.056371 s
False 0.000006 s
```

Un millón es par. La primera versión hizo casi un millón de divisiones para decirlo. La segunda hizo una.

4. Componer funciones

Con `es_primo` en el vocabulario, el programa que en el capítulo anterior era imposible se escribe en cuatro líneas:

```
def es_primo(n):
    """Dice si n es primo, probándolo contra todos los enteros menores que él."""
    if n < 2:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True

def primos_hasta(n):
    """Imprime todos los primos entre 2 y n, separados por espacios."""
    for k in range(2, n + 1):
        if es_primo(k):
            print(k, end=" ")

    print()

primos_hasta(100)
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Lee `primos_hasta` y fíjate en lo que **no** dice. No dice nada de `%`, ni de divisores, ni de ciclos internos. Dice: recorre los números del 2 al `n`, y de cada uno pregunta si es primo. Eso es lo que compran las funciones. `primos_hasta` no sabe cómo se decide si un número es primo; sabe que hay alguien que lo decide.

El método se llama **descomponer**, y la pregunta que lo guía es siempre la misma: *¿puedo explicar lo que hace este pedazo en una sola frase?* Si la respuesta es sí, ese pedazo es una función y esa frase es su nombre. «Dice si un número es primo» y «imprime los primos hasta n» son dos frases, así que son dos funciones.

Una tercera frase, que vamos a usar todo el resto del capítulo: «cuenta cuántos primos hay hasta n».

```
def contar_primos(hasta):
    """Devuelve cuántos primos hay entre 2 y `hasta`."""
    total = 0

    for k in range(2, hasta + 1):
        if es_primo(k):
            total += 1

    return total

print(contar_primos(100), "primos hasta 100")
```

```
25 primos hasta 100
```

`contar_primos` es el mismo recorrido de `primos_hasta` con un acumulador en lugar de un `print`. Devuelve, no imprime, y por eso se puede cronometrar sin llenar la pantalla de números.

5. Mejorar en un solo lugar

`es_primo` es correcta y es tonta. Para saber si 999983 es primo prueba a dividirlo entre 999981 números, y casi todos son absurdos. Vamos a arreglarla tres veces, y lo que importa de las tres veces es **dónde** se hace el arreglo.

Primero, la medición de partida. `medir` es otra función de una frase: cuenta los primos hasta un tope y dice cuánto tardó.

```
import math
import time

def es_primo(n):
    if n < 2:
```

```
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True

def contar_primos(hasta):
    total = 0

    for k in range(2, hasta + 1):
        if es_primo(k):
            total += 1

    return total

def medir(hasta):
    """Cuenta los primos hasta `hasta` e imprime cuánto tardó."""
    inicio = time.perf_counter()
    total = contar_primos(hasta)
    print(f"{total} primos hasta {hasta} en {time.perf_counter() - inicio:.3f} s")

medir(30000)
```

```
3245 primos hasta 30000 en 2.512 s
```

Primera mejora. Ningún divisor de `n`, aparte de `n` mismo, puede ser mayor que `n // 2`. Probar más arriba es tiempo perdido:

```
def es_primo(n):
    if n < 2:
        return False

    for d in range(2, n // 2 + 1):
        if n % d == 0:
            return False

    return True

medir(30000)
```

```
3245 primos hasta 30000 en 1.242 s
```

Segunda mejora. Si `n = a * b` y los dos factores fueran mayores que $\sqrt{n}$, el producto pasaría de `n`. Así que en cualquier descomposición hay un factor que no llega a la raíz, y basta buscar ahí:

```
def es_primo(n):
    if n < 2:
        return False

    for d in range(2, int(math.sqrt(n)) + 1):
        if n % d == 0:
            return False
```

```
    return True

medir(30000)
```

```
3245 primos hasta 30000 en 0.027 s
```

Tercera mejora. Si `n` es par y no es 2, ya está resuelto. Y si no es par, ningún divisor suyo lo es, así que el ciclo puede ir de dos en dos:

```
def es_primo(n):
    if n < 2:
        return False

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    for d in range(3, int(math.sqrt(n)) + 1, 2):
        if n % d == 0:
            return False

    return True

medir(30000)
```

```
3245 primos hasta 30000 en 0.016 s
```

De segundos a centésimas de segundo, sin tocar el ciclo de `medir` ni el de `contar_primos` ni el de `primos_hasta`. Los tres siguen diciendo `es_primo(k)`, la misma palabra de siempre, y los tres corrieron más rápido porque alguien cambió una línea en otro lugar.

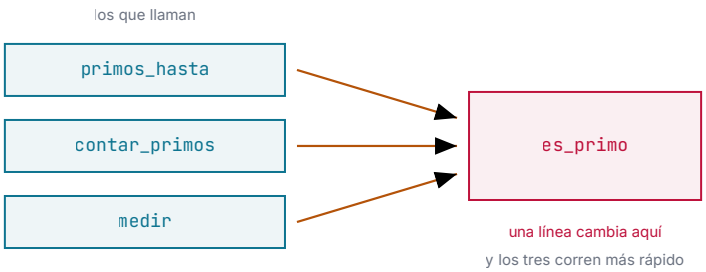


Figura 6. *Ninguno de los tres sabe cómo se decide si un número es primo, así que la mejora se hace una vez y les llega a los tres.*

Ese es el argumento entero de este capítulo, y es mucho más fuerte que «no repitas código». Una función es un contrato: dice qué recibe y qué devuelve, y no dice cómo. Todo lo que está dentro del contrato se puede cambiar sin avisarle a nadie. Un programa sin funciones no tiene contratos, y entonces cualquier mejora hay que aplicarla en todos los lugares donde se copió, y descubrir cuáles son es tu problema.

6. Alcance de las variables

Las variables que nacen dentro de una función viven solo ahí. Cuando la función termina, desaparecen, y eso incluye a la variable del ciclo:

```
import math

def primer_divisor(n):
    """Devuelve el menor divisor de n mayor que 1, o n mismo si es primo."""
    for d in range(2, int(math.sqrt(n)) + 1):
        if n % d == 0:
            return d

    return n

print(primer_divisor(91))
print(d)
```

```
7
Traceback (most recent call last):
  File "<stdin>", line 12, in <module>
NameError: name 'd' is not defined. Did you mean: 'id'?
```

Eso se llama **alcance local**, y no es una limitación sino la razón de ser de las funciones. `primer_divisor` usa una `d` sin preguntarle permiso a nadie, y ninguna otra parte del programa puede estorbarla. Cada función es una caja cerrada.

Al revés sí funciona: desde dentro se puede **leer** una variable de afuera, llamada global.

```
TOPE = 100

def esta_en_rango(n):
    return 2 <= n <= TOPE

print(esta_en_rango(50))
print(esta_en_rango(500))
```

```
True
False
```

Pero si intentas **asignarle** un valor a una variable global desde dentro, pasa algo que sorprende a todo el mundo la primera vez. Digamos que queremos saber cuántas divisiones hace `es_primo`, y llevamos la cuenta en una variable de afuera:

```
import math

divisiones = 0

def es_primo(n):
    if n < 2:
```

```
        return False

    for d in range(2, int(math.sqrt(n)) + 1):
        divisiones = divisiones + 1

        if n % d == 0:
            return False

    return True

print(es_primo(97))
```

```
Traceback (most recent call last):
  File "<stdin>", line 17, in <module>
  File "<stdin>", line 10, in es_primo
UnboundLocalError: cannot access local variable 'divisiones' where it is not associated
with a value
```

Python decide si una variable es local o global mirando el cuerpo entero de la función antes de ejecutarlo. Como ahí dentro hay una asignación a `divisiones`, la declara local para toda la función, incluida la línea que intenta leerla. Y esa variable local todavía no tiene valor cuando se la quiere leer.

Existe la palabra `global` para forzar el otro comportamiento. Casi nunca es la solución correcta, y en este curso no la vamos a usar. **Si una función necesita un valor de afuera, se lo pasas como parámetro; si produce un valor, lo devuelve con `return`.** Ese es el contrato de la sección anterior, y el contador de divisiones respeta el contrato así:

```
import math

def divisiones_para(n):
    """Devuelve cuántas divisiones hacen falta para decidir si n es primo."""
    cuenta = 0

    for d in range(2, int(math.sqrt(n)) + 1):
        cuenta += 1

        if n % d == 0:
            return cuenta

    return cuenta

print(divisiones_para(1000000), "divisiones para 1000000")
print(divisiones_para(999983), "divisiones para 999983")
```

```
1 divisiones para 1000000
998 divisiones para 999983
```

Un número par se descarta con una sola división. Un primo de seis cifras cuesta casi mil. Contar operaciones en lugar de medir segundos es una idea a la que le vamos a dedicar un capítulo entero, el 7.

7. Valores por defecto y argumentos con nombre

Un parámetro puede traer un valor de fábrica, que se usa cuando la llamada no lo menciona. `contar_primos` cuenta desde 2, y si alguna vez queremos otro punto de partida no hace falta una segunda función:

```
import math

def es_primo(n):
    if n < 2:
        return False

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    for d in range(3, int(math.sqrt(n)) + 1, 2):
        if n % d == 0:
            return False

    return True

def contar_primos(hasta, desde=2):
    """Devuelve cuántos primos hay entre `desde` y `hasta`."""
    total = 0

    for k in range(desde, hasta + 1):
        if es_primo(k):
            total += 1

    return total

print(contar_primos(100))
print(contar_primos(100, 50))
print(contar_primos(100, desde=90))
```

```
25
10
1
```

Los parámetros con valor por defecto van siempre al final de la lista, porque si no Python no sabría cómo emparejar los argumentos por posición. Y en la llamada se puede nombrar el parámetro explícitamente, como en la última línea, en cuyo caso el orden ya no importa.

Esto sirve para dos cosas. Una función con buenos valores por defecto es cómoda en el caso corriente y sigue siendo flexible en el raro: la mayoría de las llamadas van a ser `contar_primos(100)`. Y una llamada como `contar_primos(100, desde=90)` se lee sin ir a mirar la definición, cosa que `contar_primos(100, 90)` no.

8. Los docstrings

Ese texto entre comillas triples en la primera línea del cuerpo, que llevamos usando todo el capítulo, se llama **docstring**, y es la forma normal de decir qué hace una función. No es un comentario: Python lo guarda y lo puede mostrar.

```
def es_primo(n):
    """Dice si n es primo."""
    return n == 2

print(es_primo.__doc__)
```

```
Dice si n es primo.
```

La misma función `help` que puedes usar en la consola con `help(print)` lee esos docstrings. Escríbelos en una línea y en presente: «devuelve», «dice», «cuenta».

Y úsalos como prueba. Si no te sale un docstring de una sola frase, la función hace más de una cosa y hay dos funciones ahí dentro.

9. Lo que falta

`es_primo` quedó rápida, pero `primos_hasta` sigue siendo tonta de una manera que no se arregla mejorando `es_primo`. Mira lo que hace. Para decidir si 1009 es primo no usa absolutamente nada de lo que averiguó sobre 1008, ni sobre 1007, ni sobre ninguno de los mil anteriores. Empieza de cero mil veces.

Hace dos mil trescientos años, a Eratóstenes se le ocurrió lo contrario. En lugar de preguntar número por número, **se tacha.** Escribe los números del 2 al 25 en una fila. El 2 es primo, y de un tirón tacha todos sus múltiplos. El siguiente sin tachar es el 3, que por eso mismo es primo, y tacha todos los suyos. Después el 5. Lo que queda sin tachar son los primos, y no se hizo ni una sola división:



Figura 7. *Cada color es el primo que tachó esa casilla, y no hubo una sola división. El reparto es muy desparejo: de las 15 casillas tachadas, el 2 tacha 11, el 3 tacha 3 y el 5 tacha 1. El primer primo hace casi todo el trabajo, y de ahí sale lo barata que es la criba.*

Cada primo que aparece se gasta inmediatamente en descartar de golpe a muchos otros, que es justamente lo que el método de este capítulo no hace. Y todavía no lo podemos escribir.

Piensa qué haría falta. Para tachar hay que acordarse de los tachones: por cada número entre 2 y `n`, un sí o un no. Con variables eso son `n` variables, y el problema no es que sean muchas. **El problema es que `n` lo escoge quien usa el programa, y las variables hay que escribirlas antes, cuando el programa se escribe.** No se puede teclear una cantidad de nombres que todavía no se sabe cuál es.

Ahí se acaba lo que da de sí una variable por valor, y esa es exactamente la pregunta con la que abre el capítulo siguiente. La respuesta tiene nombre, y la criba la escribimos allí, al final, con la cuenta de operaciones puesta al lado del método de este capítulo.

10. Resumen

- `def` define una función; los paréntesis la llaman. Definir no ejecuta.
- Los **parámetros** están en la definición, los **argumentos** en la llamada, y se emparejan por posición.
- `return` devuelve un valor y termina la función. Una función con `return` es una expresión y se puede usar dentro de otra cuenta.
- Un `return` dentro de un ciclo sale del ciclo y de la función a la vez. No necesita `break`.
- Una función que calcula devuelve; imprimir es trabajo del programa principal.
- Una función sin `return` devuelve `None`.
- Las variables de dentro no se ven desde fuera. Desde dentro se puede leer una global, pero asignarle crea una local y da error. Los valores entran por parámetros y salen por `return`.
- Los parámetros pueden traer valor por defecto, siempre al final, y en la llamada se pueden nombrar.
- Descomponer es buscar los pedazos que se explican en una frase. Esa frase es el nombre de la función, y el mejor sitio para escribirla es el docstring.
- Una función es un contrato. Mejorarla por dentro mejora gratis a todos los que la llaman, y eso es lo que hicimos cuatro veces con `es_primo`.

Ejercicios

1. Escribe `mcd(a, b)`, el máximo común divisor, con el algoritmo de Euclides: el mcd de `a` y `b` es el mcd de `b` y el resto de dividir `a` entre `b`, hasta que el resto sea cero. Con ella escribe `mcm(a, b)` en una sola línea.
2. Escribe `factorial(n)` y úsala para escribir `combinaciones(n, k)`, que calcula de cuántas maneras se pueden escoger `k` elementos de un conjunto de `n`. La fórmula es $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$. Comprueba que `combinaciones(5, 2)` da 10.
3. Escribe `suma_de_divisores(n)`, que devuelve la suma de los divisores propios de `n` (los que son menores que `n`). Con ella escribe `es_perfecto(n)` en una línea, y con esa impri-

me todos los números perfectos menores que 10000. Compara este programa con el que escribiste en el capítulo anterior para lo mismo.

4. Escribe `longitud_collatz(n)`, que devuelve cuántos pasos tarda la secuencia de Collatz del capítulo anterior en llegar a 1. Úsala para encontrar el número menor que 1000 con la secuencia más larga.
5. Los **primos gemelos** son parejas de primos que difieren en 2, como 11 y 13, o 41 y 43. Escribe `contar_gemelos(n)` que cuente cuántas parejas hay hasta `n`. Guarda la respuesta, que en el capítulo siguiente vas a escribir otra versión y a compararlas.
6. Escribe `dibujar_triangulo(altura, caracter="*")` que imprima un triángulo. Con `dibujar_triangulo(4)` debe salir:

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

   Añádele un parámetro `invertido=False` que lo dibuje al revés cuando valga `True`.

7. Sin ejecutarlos, di qué imprime cada uno de estos tres programas. Después compruébalo.

   (a)

```
def f(x):
    x = x + 1
    return x

y = 5
f(y)
print(y)
```

   (b)

```
def g(n):
    if n > 1:
        return "puede ser primo"
    print("seguro que no")

print(g(5))
print(g(1))
```

   (c)

```
def h(a, b=2, c=3):
    return a * 100 + b * 10 + c

print(h(1))
print(h(1, 5))
print(h(1, c=5))
```

4
Listas

Hay que procesar las notas de un grupo. Con tres estudiantes se resuelve con lo que ya sabes:

```
nota1 = int(input("Nota: "))
nota2 = int(input("Nota: "))
nota3 = int(input("Nota: "))
```

Con treinta da vergüenza escribirlo, pero todavía se puede. El problema de verdad aparece cuando no sabes cuántos son. Un programa que sirva para cualquier grupo necesitaría tantas variables como estudiantes tenga el grupo, y las variables hay que escribirlas antes, cuando el programa se escribe, mucho antes de que nadie diga cuántos estudiantes hay. **No se puede teclear una cantidad de nombres que todavía no se sabe cuál es.**

Ese es el límite con el que cerró el capítulo anterior, y no se arregla con más variables ni con nombres más listos. Hace falta otra cosa: un solo nombre que guarde muchos valores, tantos como haga falta y decididos mientras el programa corre. Se llama **lista**, y es todo el contenido de este capítulo.

Mira además lo que se abre con eso. Hasta ahora un acumulador te daba el promedio de las notas, pero no la mediana, ni las notas ordenadas, ni cuántas están por encima del promedio, porque en cuanto leías la nota siguiente la anterior se perdía. Guardarlas todas es lo que permite mirarlas dos veces. Y al final del capítulo, con eso solo, cae la criba de Eratóstenes.

1. La lista

Una lista es una secuencia de valores. Se escribe entre corchetes, con los elementos separados por comas:

```
notas = [4, 5, 3, 5, 2, 4]
print(notas)
print(type(notas))
print(len(notas))
```

```
[4, 5, 3, 5, 2, 4]
<class 'list'>
6
```

`len` devuelve cuántos elementos tiene. Una lista puede estar vacía, y puede contener valores de cualquier tipo, incluso mezclados, aunque en la práctica casi siempre guardamos cosas del mismo tipo.

```
vacia = []
mezclada = [1, "dos", 3.0]
print(len(vacia), len(mezclada))
```

```
0 3
```

Los índices

A cada elemento se llega por su posición, que se llama **índice** y se escribe entre corchetes. **Los índices empiezan en cero.**

```
notas = [4, 5, 3, 5, 2, 4]
print(notas[0])
print(notas[1])
print(notas[5])
```

```
4
5
4
```

Que empiecen en cero tiene una consecuencia que hay que tener presente todo el tiempo: el último índice válido de una lista de `n` elementos es `n - 1`, no `n`. Pedir uno más allá del final es un error:

```
notas = [4, 5, 3, 5, 2, 4]
print(notas[6])
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

Ese `IndexError` lo vas a ver muchas veces este semestre, y casi siempre significa lo mismo: un ciclo que dio una vuelta de más.



Figura 8. *Un solo nombre para seis casillas. Arriba el índice, que empieza en cero; abajo el índice negativo, que cuenta desde el final.*

Python admite además índices negativos, que cuentan desde el final. `-1` es el último, `-2` el penúltimo. Es cómodo y evita escribir `notas[len(notas) - 1]`:

```
notas = [4, 5, 3, 5, 2, 4]
print(notas[-1])
print(notas[-2])
```

```
4
2
```

2. Recorrer una lista

Aquí es donde el `for` del capítulo 2 enseña por fin su verdadera cara. Hasta ahora lo usábamos siempre sobre `range`, para contar. Pero `for` no sirve para contar: sirve para **recorrer una secuencia**, y `range` era solo una secuencia de números.

```
notas = [4, 5, 3, 5, 2, 4]

for nota in notas:
    print(nota, end=" ")
```

```
4 5 3 5 2 4
```

La variable `nota` toma en cada vuelta el valor de un elemento, en orden. No hay índices, no hay contador, no hay manera de pasarse del final. Esta es la forma normal de recorrer una lista en Python y debe ser tu primer instinto.

Con lo que ya sabes de acumuladores, calcular el promedio es inmediato:

```
notas = [4, 5, 3, 5, 2, 4]
suma = 0

for nota in notas:
    suma += nota

print(suma / len(notas))
```

```
3.8333333333333335
```

Cuando hace falta el índice

A veces necesitas saber en qué posición vas, por ejemplo para imprimir un número de orden. Entonces se recorre `range(len(...))`:

```python
notas = [4, 5, 3, 5, 2, 4]

for i in range(len(notas)):
    print(f"Estudiante {i + 1}: {notas[i]}")
```

```
Estudiante 1: 4
Estudiante 2: 5
Estudiante 3: 3
Estudiante 4: 5
Estudiante 5: 2
Estudiante 6: 4
```

Y si necesitas las dos cosas, el índice y el elemento, Python tiene `enumerate`, que es más limpio que indexar a mano:

```python
notas = [4, 5, 3, 5, 2, 4]

for i, nota in enumerate(notas):
    print(f"Estudiante {i + 1}: {nota}")
```

```
Estudiante 1: 4
Estudiante 2: 5
Estudiante 3: 3
Estudiante 4: 5
Estudiante 5: 2
Estudiante 6: 4
```

Las dos variables separadas por coma tienen una explicación, que llega en la sección 6. Por ahora tómalo como la forma de escribirlo.

La regla: recorre con `for nota in notas` siempre que puedas, y usa el índice solo cuando de verdad lo necesites.

Buscar

El operador `in` dice si un valor está en la lista, y `count` cuántas veces:

```
notas = [4, 5, 3, 5, 2, 4]
print(5 in notas)
print(6 in notas)
print(notas.count(5))
print(notas.index(3))
```

```
True
False
2
2
```

`index` devuelve la posición de la primera aparición, y da error si el valor no está.

3. Modificar una lista

Todo lo que hemos visto hasta aquí era **inmutable**: no se puede cambiar un número ni una parte de un texto. Cuando escribías `x = x + 1` no modificabas el 5, creabas un 6 y le ponías el mismo nombre.

Las listas son distintas. Una lista **se puede modificar en el lugar**, y eso cambia cómo hay que pensar sobre ellas.

```
notas = [4, 5, 3, 5, 2, 4]
notas[2] = 5
print(notas)
```

```
[4, 5, 5, 5, 2, 4]
```

Las operaciones que más se usan:

OPERACIÓN	QUÉ HACE
`lista.append(x)`	agrega `x` al final
`lista.insert(i, x)`	inserta `x` en la posición `i`
`lista.pop()`	saca y devuelve el último
`lista.pop(i)`	saca y devuelve el de la posición `i`
`lista.remove(x)`	elimina la primera aparición de `x`
`lista.sort()`	ordena la lista en el lugar
`lista.reverse()`	la invierte en el lugar

```
notas = [4, 5, 3]
notas.append(2)
print(notas)

ultima = notas.pop()
print(ultima, notas)

notas.insert(0, 5)
print(notas)
```

```
[4, 5, 3, 2]
2 [4, 5, 3]
[5, 4, 5, 3]
```

`append` es la que más vas a usar, porque es como se construye una lista a partir de una vacía:

```
cuadrados = []

for i in range(1, 6):
    cuadrados.append(i * i)

print(cuadrados)
```

```
[1, 4, 9, 16, 25]
```

La sorpresa más cara del semestre

Presta atención a esto, porque es la fuente de más errores de este curso que cualquier otra cosa. Con los números, asignar copia el valor:

```
a = 5
b = a
a = 6
print(a, b)
```

```
6 5
```

`b` sigue valiendo 5, como trazamos en el capítulo 1. Ahora lo mismo con listas:

```
a = [1, 2, 3]
b = a
a.append(4)
print(a)
print(b)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

`b` cambió también, y nadie lo tocó. La razón es que `b = a` no copia la lista: copia el **nombre**. Después de esa línea hay una sola lista con dos nombres, y modificarla por cualquiera de los dos la modifica para ambos.

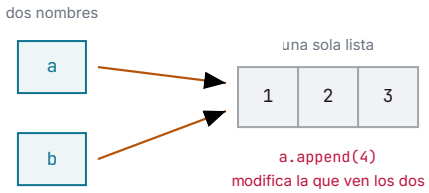


Figura 9. `b = a` *no duplica las casillas. Duplica la flecha.*

Esto no pasaba antes porque los números no se pueden modificar en el lugar; la única manera de cambiar `a` era asignarle otra cosa, y eso rompe el vínculo. Con una lista, `a.append(4)` no asigna nada: modifica el objeto que los dos nombres comparten.

Para copiar de verdad hay que pedirlo:

```
a = [1, 2, 3]
b = list(a)
a.append(4)
print(a)
print(b)
```

```
[1, 2, 3, 4]
[1, 2, 3]
```

Lo mismo pasa con los parámetros de una función. Si le pasas una lista a una función y la función la modifica, la lista de afuera queda modificada:

```
def agregar_cero(lista):
    lista.append(0)

notas = [4, 5, 3]
agregar_cero(notas)
print(notas)
```

```
[4, 5, 3, 0]
```

Compara con lo que viste en el capítulo anterior, donde una función no podía cambiar el valor de un número de afuera. No es una contradicción: en los dos casos la función recibe el valor del argumento, pero el valor de una lista es la lista misma, no una copia de ella. A ve-

ces esto es exactamente lo que quieres. Cuando no lo es, la función debe trabajar sobre una copia y devolverla.

4. Rebanadas

Se puede pedir un pedazo de una lista con dos índices separados por dos puntos. El primero se incluye y el segundo no, la misma convención de `range`:

```
notas = [4, 5, 3, 5, 2, 4]
print(notas[1:4])
print(notas[:3])
print(notas[3:])
print(notas[::-1])
```

```
[5, 3, 5]
[4, 5, 3]
[5, 2, 4]
[4, 2, 5, 3, 5, 4]
```

Si omites el primer índice, empieza desde el principio; si omites el segundo, llega hasta el final. Un tercer número es el paso, y `-1` recorre al revés, que es la forma más corta de invertir una lista.

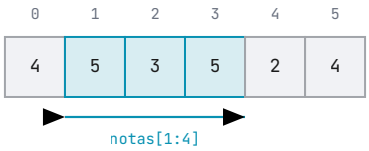


Figura 10. *El primer índice entra y el segundo no, la misma convención de* `range`.

Una rebanada siempre construye una **lista nueva**. Por eso `notas[:]` es otra manera de copiar, y por eso modificar la rebanada no toca el original:

```
notas = [4, 5, 3, 5, 2, 4]
copia = notas[:]
copia[0] = 100
print(notas)
print(copia)
```

```
[4, 5, 3, 5, 2, 4]
[100, 5, 3, 5, 2, 4]
```

5. El proyecto: las notas del grupo

Con esto ya podemos escribir el programa completo. Son tres piezas, cada una una función, como aprendimos en el capítulo anterior.

Leer las notas hasta que el usuario escriba una línea en blanco:

```
def leer_notas():
    """Lee notas de la consola hasta una línea vacía. Devuelve la lista."""
    notas = []

    while True:
        linea = input("Nota (Enter para terminar): ")

        if linea == "":
            return notas

        notas.append(int(linea))
```

Calcular el promedio y el máximo. El máximo lo escribimos a mano, que es el patrón acumulador del capítulo 2 aplicado a una lista:

```
def promedio(notas):
    """Devuelve la media aritmética de una lista no vacía."""
    return sum(notas) / len(notas)

def maximo(notas):
    """Devuelve el mayor elemento de una lista no vacía."""
    mayor = notas[0]

    for nota in notas:
        if nota > mayor:
            mayor = nota

    return mayor

print(promedio([4, 5, 3, 5, 2, 4]))
print(maximo([4, 5, 3, 5, 2, 4]))
```

```
3.8333333333333335
5
```

Fíjate en que `maximo` empieza con `notas[0]` y no con cero. Empezar en cero funcionaría para notas, que son positivas, y fallaría con temperaturas bajo cero. Empezar por el primer elemento es siempre correcto, y a cambio la función exige que la lista no esté vacía.

Ahora que lo escribiste, te digo que Python ya lo trae: `max`, `min` y `sum` hacen esto mismo. Úsalos de aquí en adelante, pero el ejercicio valía la pena, porque en el examen te van a pedir el segundo mayor, o el mayor que cumple una condición, y ahí no hay función que lo haga por ti.

```
notas = [4, 5, 3, 5, 2, 4]
print(sum(notas), max(notas), min(notas), len(notas))
```

```
23 5 2 6
```

Lo que antes no se podía

Con las notas guardadas, aparecen las preguntas que el capítulo 2 no podía responder. Cuántas están por encima del promedio, que exige recorrer la lista dos veces:

```
notas = [4, 5, 3, 5, 2, 4]
media = sum(notas) / len(notas)
sobre_la_media = 0

for nota in notas:
    if nota > media:
        sobre_la_media += 1

print(f"Promedio {media:.2f}, y {sobre_la_media} notas por encima")
```

```
Promedio 3.83, y 4 notas por encima
```

Y la mediana, que es el valor del medio una vez ordenadas:

```
def mediana(notas):
    """Devuelve la mediana de una lista no vacía."""
    ordenadas = sorted(notas)
    mitad = len(ordenadas) // 2

    if len(ordenadas) % 2 == 1:
        return ordenadas[mitad]

    return (ordenadas[mitad - 1] + ordenadas[mitad]) / 2

print(mediana([4, 5, 3, 5, 2, 4]))
print(mediana([4, 5, 3]))
```

```
4.0
4
```

`sorted` devuelve una lista nueva ordenada, y no toca la original; `lista.sort()` ordena en el lugar y no devuelve nada. Aquí los usamos como cajas negras. En el capítulo 7 vamos a escribir nosotros el algoritmo que hay dentro.

6. Tuplas

Python tiene otra secuencia que se parece mucho a la lista. Se llama **tupla** y se escribe con paréntesis en lugar de corchetes:

```
punto = (3, 4)
print(punto)
print(type(punto))
print(punto[0], punto[1], len(punto))
```

```
(3, 4)
<class 'tuple'>
3 4 2
```

Se indexa, se rebana y se recorre con `for` igual que una lista. La diferencia es una sola, y es la de la sección 3: **una tupla no se puede modificar.**

```
punto = (3, 4)
punto[0] = 10
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
```

No tiene `append`, ni `pop`, ni `sort`. Una vez construida se queda como está, y por eso la sorpresa de `b = a` no puede pasar con una tupla: los dos nombres comparten el mismo objeto, pero ninguno de los dos lo puede cambiar.

Para qué sirve una lista que no se puede cambiar

Una lista guarda muchos valores, cuántos no se sabe de antemano, y todos significan lo mismo: todos son notas. Una tupla guarda unos pocos valores, siempre la misma cantidad, y cada posición significa algo distinto. En `(3, 4)` el primero es la x y el segundo la y. En `("Ana", 5)` el primero es un nombre y el segundo una nota. Agregarle un tercer elemento a un punto del plano no tiene sentido, y la tupla no deja hacerlo.

Las dos cosas se combinan. Un grupo es una lista, porque no se sabe cuántos estudiantes tiene, y cada estudiante es una tupla, porque siempre es un nombre y una nota:

```
grupo = [("Ana", 5), ("Luis", 3), ("Marta", 4)]

for estudiante in grupo:
    print(estudiante[0], "sacó", estudiante[1])
```

```
Ana sacó 5
Luis sacó 3
Marta sacó 4
```

Desempaquetar

Los índices `[0]` y `[1]` funcionan, pero obligan a recordar qué había en cada posición. Python permite repartir una tupla entre varias variables de un golpe, una por posición:

```
punto = (3, 4)
x, y = punto
print(x)
print(y)
```

```
3
4
```

Eso se llama **desempaquetar**. A la izquierda tiene que haber tantas variables como elementos tenga la tupla:

```
x, y = (3, 4, 5)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2, got 3)
```

Y se puede desempaquetar en el mismo `for`, que es como se recorre de verdad una lista de tuplas:

```
grupo = [("Ana", 5), ("Luis", 3), ("Marta", 4)]

for nombre, nota in grupo:
    print(nombre, "sacó", nota)
```

```
Ana sacó 5
Luis sacó 3
Marta sacó 4
```

Con esto ya se puede leer lo que hacía `enumerate` en la sección 2. `enumerate` produce tuplas de dos elementos, la posición y el valor, y `for i, nota in enumerate(notas)` desempaqueta cada una:

```
notas = [4, 5, 3]
print(list(enumerate(notas)))
```

```
[(0, 4), (1, 5), (2, 3)]
```

La coma es la que hace la tupla

Los paréntesis son casi siempre opcionales. Lo que construye la tupla es la coma:

```
t = 3, 4
print(t, type(t))

u = (5)
v = (5,)
print(type(u), type(v))
```

```
(3, 4) <class 'tuple'>
<class 'int'> <class 'tuple'>
```

`(5)` es el número 5 entre paréntesis, como en `(5 + 1) * 2`. Una tupla de un solo elemento necesita la coma al final.

De ahí salen dos cosas que vas a usar todo el tiempo. La primera es que una función puede devolver varios valores, porque `return a, b` devuelve una tupla, y quien la llama la desempaqueta:

```
def minimo_y_maximo(notas):
    """Devuelve el menor y el mayor elemento de una lista no vacía."""
    return min(notas), max(notas)

menor, mayor = minimo_y_maximo([4, 5, 3, 5, 2, 4])
print(menor, mayor)
```

```
2 5
```

La segunda es el intercambio de dos variables. Escribir `a = b` y después `b = a` no sirve, porque la primera línea pierde el valor viejo de `a`. Con tuplas sale en una línea:

```
a = 1
b = 2
a, b = b, a
print(a, b)
```

```
2 1
```

La derecha se evalúa primero y construye la tupla `(2, 1)` con los valores viejos; después la izquierda la desempaqueta. Te va a servir en el ejercicio 3.

7. Listas de listas

Un elemento de una lista puede ser cualquier cosa, incluso otra lista. Así se representa una tabla: las notas de varios estudiantes en varias asignaturas, una lista por estudiante.

```
tabla = [
    [5, 4, 5],
    [3, 3, 4],
    [2, 5, 3],
]

print(len(tabla))
print(tabla[0])
print(tabla[0][2])
```

```
3
[5, 4, 5]
5
```

`tabla[0]` es la primera fila, que es una lista. `tabla[0][2]` es la tercera nota de esa fila. El primer índice escoge la fila y el segundo la columna.

Para recorrerla hacen falta dos ciclos, uno dentro de otro:

```
tabla = [
    [5, 4, 5],
    [3, 3, 4],
    [2, 5, 3],
]

for fila in tabla:
    for nota in fila:
        print(nota, end=" ")
    print()
```

```
5 4 5
3 3 4
2 5 3
```

El `print()` sin argumentos al final del ciclo exterior es lo que salta de línea al terminar cada fila. Quítalo y verás las nueve notas en una sola línea.

Combinando lo de este capítulo, el promedio de cada estudiante sale en cuatro líneas:

```
tabla = [
    [5, 4, 5],
    [3, 3, 4],
```

```
    [2, 5, 3],
]

for i, fila in enumerate(tabla):
    print(f"Estudiante {i + 1}: promedio {sum(fila) / len(fila):.2f}")
```

```
Estudiante 1: promedio 4.67
Estudiante 2: promedio 3.33
Estudiante 3: promedio 3.33
```

Construir una matriz, y una trampa

Para hacer una tabla de ceros uno intenta lo obvio, y lo obvio está mal:

```
tabla = [[0] * 3] * 2
print(tabla)

tabla[0][0] = 9
print(tabla)
```

```
[[0, 0, 0], [0, 0, 0]]
[[9, 0, 0], [9, 0, 0]]
```

Cambiamos una casilla y se cambiaron dos. Es exactamente el problema de la sección 3: `[fila] * 2` no hace dos filas, hace **una sola fila con dos nombres**. La forma correcta es construir cada fila por separado:

```
tabla = []

for i in range(2):
    tabla.append([0] * 3)

tabla[0][0] = 9
print(tabla)
```

```
[[9, 0, 0], [0, 0, 0]]
```

`[0] * 3` sí está bien, porque los ceros son inmutables y no importa que se compartan. Lo que no se puede repetir con `*` es una lista que después vas a modificar.

8. La criba de Eratóstenes

En el capítulo anterior quedó planteado un problema y no se pudo resolver. Recuérdalo: para imprimir los primos hasta `n`, `es_primo` los interroga de uno en uno, y para decidir si 1009 es primo no usa nada de lo que averiguó sobre los mil anteriores. Eratóstenes hace lo

contrario. En lugar de preguntar número por número, **tacha.** El 2 es primo, y de un tirón se tachan todos sus múltiplos; el siguiente sin tachar es el 3, que por eso mismo es primo, y se tachan los suyos; después el 5:



Figura 11. *Cada color es el primo que tachó esa casilla, y no hubo una sola división. El reparto es muy desparejo: de las 15 casillas tachadas, el 2 tacha 11, el 3 tacha 3 y el 5 tacha 1. El primer primo hace casi todo el trabajo, y de ahí sale lo barata que es la criba.*

Lo que faltaba era dónde anotar los tachones: un sí o un no por cada número entre 2 y `n`, con `n` decidido por quien usa el programa. Eso ya es una sola línea:

```
n = 25
compuesto = [False] * (n + 1)

print(len(compuesto))
print(compuesto[0], compuesto[25])
```

```
26
False False
```

Veintiséis casillas, todas en `False`, y el 26 salió de una variable. Con casillas de la 0 a la `n`, el número `k` vive en la casilla `k`; cuesta dos casillas que no se usan y ahorra estar restando. Y `[False] * (n + 1)` es seguro, a diferencia de la tabla de la sección anterior, porque `False` es inmutable: da igual que las veintiséis casillas compartan el mismo, porque nadie lo va a modificar en el lugar, solo reemplazarlo.

Con eso la criba cabe en doce líneas:

```
def criba(n):
    """Devuelve la lista de los primos hasta n, por el método de Eratóstenes."""
    compuesto = [False] * (n + 1)
    primos = []

    for candidato in range(2, n + 1):
        if compuesto[candidato]:
            continue

        primos.append(candidato)

        for multiplo in range(candidato * candidato, n + 1, candidato):
            compuesto[multiplo] = True

    return primos

print(criba(25))
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Léela contra la figura 11. `compuesto[k]` es el tachón del número `k`. El ciclo de afuera recorre los candidatos; si uno viene tachado, `continue` salta a la vuelta siguiente. Si no viene tachado es primo, se anota con `append`, y el ciclo de adentro tacha sus múltiplos.

Dos detalles que pagan la pena de leerlos despacio. El ciclo de adentro empieza en `candidato * candidato`, no en `candidato * 2`, porque cualquier múltiplo menor que el cuadrado ya tiene un factor más chico que lo tachó antes: cuando llegamos al 5, el 10 y el 15 y el 20 ya están tachados por el 2 y por el 3. Y el paso del `range` es `candidato`, que es lo que hace que el ciclo salte de múltiplo en múltiplo sin comprobar nada.

Los dos métodos, medidos

El mismo trabajo por los dos caminos: contar los primos que hay hasta un millón.

```
import math
import time

def es_primo(n):
    if n < 2:
        return False

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    for d in range(3, int(math.sqrt(n)) + 1, 2):
        if n % d == 0:
            return False

    return True

def contar_primos(hasta):
    total = 0

    for k in range(2, hasta + 1):
        if es_primo(k):
            total += 1

    return total

inicio = time.perf_counter()
print(contar_primos(1000000), "primos, uno por uno")
print(f"{time.perf_counter() - inicio:.2f} s")

inicio = time.perf_counter()
print(len(criba(1000000)), "primos, tachando")
print(f"{time.perf_counter() - inicio:.2f} s")
```

```
78498 primos, uno por uno
1.67 s
```

```
78498 primos, tachando
0.14 s
```

Segundos contra décimas de segundo, y las dos respuestas son la misma. Pero los segundos no son un buen instrumento: dependen de la máquina, de lo que esté haciendo el sistema operativo y de la versión de Python. Corre ese bloque dos veces y te van a salir números distintos.

Lo que no cambia es **cuántas operaciones hace cada método**. El de uno en uno divide, así que se cuentan sus divisiones. La criba tacha, así que se cuentan sus tachones. Esos dos números son exactos y siempre los mismos:

```python
import math

def divisiones_hasta(n):
    """Cuenta las divisiones que hace el método de uno en uno para llegar a n."""
    total = 0

    for k in range(2, n + 1):
        for d in range(2, int(math.sqrt(k)) + 1):
            total += 1

            if k % d == 0:
                break

    return total

def tachones_hasta(n):
    """Cuenta los tachones que hace la criba para llegar a n."""
    compuesto = [False] * (n + 1)
    total = 0

    for candidato in range(2, n + 1):
        if compuesto[candidato]:
            continue

        for multiplo in range(candidato * candidato, n + 1, candidato):
            compuesto[multiplo] = True
            total += 1

    return total

topes = [100000, 300000, 900000]
divisiones = []
tachones = []

for tope in topes:
    divisiones.append(divisiones_hasta(tope))
    tachones.append(tachones_hasta(tope))

    print(f"hasta {tope:>7}: {divisiones[-1]:>10} divisiones"
          f"   contra {tachones[-1]:>8} tachones")
```

```
hasta  100000:    2745694 divisiones   contra   193078 tachones
hasta  300000:   12582930 divisiones   contra   607825 tachones
hasta  900000:   58461003 divisiones   contra  1902617 tachones
```

Las dos cuentas se van guardando en sendas listas, que es justo lo de este capítulo: no sabemos de antemano cuántos topes vamos a probar.

Mira esa tabla por columnas y no por filas. Cada línea multiplica el tope por tres. Los tachones de la criba también se multiplican por tres, más o menos: hacer el triple de trabajo para resolver el triple de números es lo mejor que se puede esperar. Las divisiones del otro método se multiplican por casi cinco.

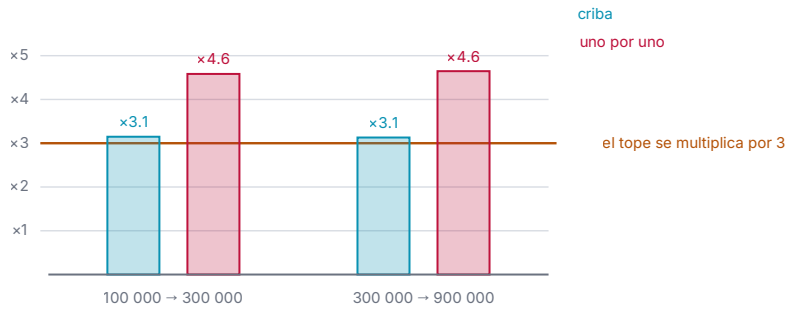


Figura 12. *Lo que se multiplica cada cuenta cuando el tope se multiplica por tres. La criba se mantiene en el triple; el otro método se va a casi el quíntuple.*

Ahí está la diferencia, y no es que un programa sea diez veces más lento que el otro. Es que **la distancia entre los dos se ensancha cada vez que crece `n`.** Un factor de diez se paga comprando una computadora mejor. Esto no.

Eso se llama **orden de crecimiento**, es el tema del capítulo 7, y esta tabla se va a volver a hacer allí con nombres y con fórmulas. Por ahora quédate con la razón intuitiva de por qué la criba gana. El método de uno en uno empieza de cero con cada número y tira a la basura todo lo que averiguó. La criba hace lo contrario: cada primo que encuentra lo gasta inmediatamente en descartar de golpe a muchos otros, así que cada respuesta que consigue le abarata las siguientes.

Dejar de recalcular y empezar a recordar es la idea de la programación dinámica, que queda para el segundo semestre, y para recordar hace falta dónde guardar, que es lo de este capítulo.

9. Resumen

- Una lista guarda muchos valores con un solo nombre. Se escribe con corchetes y su tamaño se pide con `len`.
- Los índices empiezan en cero y el último es `len - 1`. Los negativos cuentan desde el final. Pasarse da `IndexError`.
- `for nota in notas` recorre los elementos y es la forma normal. Usa el índice solo si lo necesitas, y entonces `enumerate`.
- Las listas son mutables: `append`, `pop`, `insert`, `remove`, `sort`.

- `b = a` no copia la lista, le pone otro nombre. Para copiar, `list(a)` o `a[:]`.
- Una función que recibe una lista puede modificarla, y el cambio se ve desde fuera.
- Una rebanada `a[i:j]` construye una lista nueva.
- `sum`, `max`, `min` y `sorted` existen, pero conviene saber escribirlos.
- Una tupla es una secuencia que no se puede modificar. Lo que la construye es la coma, y sirve para unos pocos valores donde cada posición significa algo: un punto, un nombre con su nota.
- `x, y = punto` desempaqueta una tupla. `for nombre, nota in grupo` y `for i, nota in enumerate(notas)` hacen lo mismo en cada vuelta, `return a, b` devuelve dos valores y `a, b = b, a` los intercambia.
- Una lista de listas es una tabla: `tabla[fila][columna]`, y se recorre con dos ciclos anidados.
- `[[0] * 3] * 2` no construye dos filas. Constrúyelas en un ciclo. `[False] * n` sí está bien, porque `False` es inmutable.
- Una lista es lo que permite guardar tantos resultados intermedios como haga falta sin saber cuántos son al escribir el programa. La criba de Eratóstenes vive entera de eso.

Ejercicios

1. Escribe `segundo_mayor(lista)` que devuelva el segundo elemento más grande, sin ordenar la lista y recorriéndola una sola vez. Piensa qué debe pasar con `[5, 5, 3]`.
2. Escribe `minimo_y_maximo(lista)` que devuelva una tupla con el menor y el mayor elemento, sin usar `min` ni `max` y recorriendo la lista una sola vez.
3. Escribe `invertir(lista)` que devuelva una lista nueva con los elementos al revés, sin usar `[::-1]` ni `reverse`. Después escribe `invertir_en_el_lugar(lista)` que no devuelva nada y modifique la que recibió, intercambiando el primero con el último, el segundo con el penúltimo, y así. ¿Hasta dónde tiene que llegar el ciclo?
4. Escribe `sin_repetidos(lista)` que devuelva una lista nueva con los elementos de la original, en el mismo orden, pero sin repeticiones.
5. Escribe `es_capicua(lista)` que diga si la lista se lee igual al derecho y al revés. Resuélvelo sin construir ninguna lista nueva.
6. Con una tabla de números como la de la sección 7, escribe `transpuesta(tabla)` que devuelva una tabla nueva donde las filas son las columnas de la original. Comprueba que la transpuesta de la transpuesta es la tabla de partida.
7. Vuelve a escribir `contar_gemelos(n)` del ejercicio 5 del capítulo anterior, ahora sobre la lista que devuelve `criba`. Cronometra las dos versiones con `n` de un millón y compara el resultado, que tiene que ser el mismo número.
8. Cuenta los primos menores que un millón con la criba, y después prueba con diez millones y con cien millones. En algún punto el programa deja de ser lento y pasa a ser impo-

sible. Di en qué punto y por qué; la respuesta no tiene que ver con el tiempo sino con lo que ocupa `compuesto`. Estima cuántos bytes hacen falta por casilla.

9. Sin ejecutarlos, di qué imprime cada uno de estos tres programas. Después compruébalo.

   (a)

```
a = [1, 2, 3]
b = a
c = a[:]
a[0] = 99
print(b[0], c[0])
```

   (b)

```
def f(lista):
    lista = [0, 0, 0]

def g(lista):
    lista[0] = 0

x = [1, 2, 3]
f(x)
print(x)
g(x)
print(x)
```

   (c)

```
numeros = [1, 2, 4, 3]

for n in numeros:
    if n % 2 == 0:
        numeros.remove(n)

print(numeros)
```

   El tercero quiere eliminar los pares y no lo consigue. Explica por qué, y arréglalo. Después corre el mismo programa con `[1, 2, 3, 4]`, donde sí da la respuesta correcta, y explica por qué esa entrada esconde el error. Es la clase de coincidencia que hace que un programa roto pase las pruebas.

5

Cadenas y archivos

Llevas cuatro capítulos usando cadenas de texto, pero solo para imprimirlas y para leerlas de la consola. Aquí las vamos a abrir por dentro y vas a ver que una cadena es una secuencia, igual que una lista, con las mismas reglas de índices y rebanadas que aprendiste en el capítulo anterior. Y como el texto de verdad no se escribe a mano sino que vive en archivos, también aprendemos a leerlos. El proyecto del capítulo es un programa que abre un archivo y cuenta sus palabras.

Trabajamos sobre el archivo `quijote.txt`, que puedes descargar de la carpeta `conferences/2026/libro` del repositorio del curso y contiene los tres primeros párrafos del Quijote.

1. Una cadena es una secuencia

Todo lo que aprendiste de listas sirve aquí. Una cadena tiene longitud, se indexa desde cero, admite índices negativos y se rebana:

```
palabra = "computación"

print(len(palabra))
print(palabra[0])
print(palabra[-1])
print(palabra[0:5])
print(palabra[5:])
print(palabra[::-1])
```

```
11
c
n
compu
tación
nóicatupmoc
```



Figura 13. *La misma fila de casillas del capítulo anterior, con un carácter en cada una. Lo único que cambia es que estas no se pueden modificar.*

Y se recorre con `for`, elemento por elemento, que en una cadena significa carácter por carácter:

```
for letra in "MatCom":
    print(letra, end="-")
```

```
M-a-t-C-o-m-
```

Contar con esto es inmediato. Cuántas vocales tiene una palabra:

```
palabra = "computación"
vocales = 0

for letra in palabra:
    if letra in "aeiouáéíóú":
        vocales += 1

print(vocales)
```

```
5
```

Ahí hay un detalle que conviene mirar. En una lista, `x in lista` pregunta si `x` es uno de los elementos. En una cadena, `in` pregunta si aparece como **pedazo**, no solo como carácter suelto:

```
print("a" in "hola")
print("ola" in "hola")
print("alo" in "hola")
```

```
True
True
False
```

Eso es cómodo para buscar palabras dentro de un texto y no tiene equivalente en las listas.

Las cadenas no se pueden modificar

Aquí está la diferencia grande con las listas. Una lista se modifica en el lugar; una cadena no:

```
palabra = "hola"
palabra[0] = "H"
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Las cadenas son **inmutables**, como los números. La consecuencia práctica es que toda operación sobre una cadena construye una cadena nueva y deja la original intacta. Para "cambiar" una cadena hay que reasignar el nombre:

```
palabra = "hola"
mayuscula = palabra.upper()

print(palabra)
print(mayuscula)

palabra = palabra.upper()
print(palabra)
```

```
hola
HOLA
HOLA
```

Y como no se pueden modificar, tampoco tienen el problema de los nombres compartidos que vimos en el capítulo anterior. Dos nombres para la misma cadena nunca se sorprenden el uno al otro, porque ninguno puede cambiarla.

2. Métodos de cadena

Una cadena trae muchas funciones propias, que se llaman con un punto detrás del valor. Estas son las que vas a usar todo el tiempo:

MÉTODO	QUÉ DEVUELVE
`s.lower()`, `s.upper()`	la misma cadena en minúsculas o mayúsculas
`s.strip()`	la cadena sin espacios ni saltos al principio y al final
`s.replace(a, b)`	una copia con cada `a` cambiado por `b`
`s.split()`	la **lista** de las palabras, separando por espacios
`s.split(c)`	la lista de los pedazos, separando por `c`
`separador.join(lista)`	pega los elementos de la lista con ese separador
`s.startswith(t)`, `s.endswith(t)`	si empieza o termina con `t`

MÉTODO	QUÉ DEVUELVE
`s.find(t)`	la posición de `t`, o `-1` si no está
`s.count(t)`	cuántas veces aparece `t`

Ninguno modifica la cadena: todos devuelven una nueva.

```
linea = "  En un lugar de la Mancha  "

print(repr(linea.strip()))
print(linea.strip().lower())
print(linea.count("a"))
print(linea.find("lugar"))
```

```
'En un lugar de la Mancha'
en un lugar de la mancha
4
8
```

`repr` imprime la cadena con sus comillas, y sirve para ver exactamente dónde están los espacios. Úsalo cuando algo no cuadre.

`split` y `join`

De todos, el importante para nosotros es `split`, porque convierte un texto en una lista de palabras y te devuelve al terreno del capítulo anterior:

```
linea = "En un lugar de la Mancha"
palabras = linea.split()

print(palabras)
print(len(palabras))
print(palabras[2])
```

```
['En', 'un', 'lugar', 'de', 'la', 'Mancha']
6
lugar
```



Figura 14. `split` *parte por los espacios y los descarta. Lo que devuelve es una lista, así que todo lo del capítulo anterior vuelve a servir.*

`join` hace el camino inverso: recibe una lista y la pega en una sola cadena. Se escribe al revés de como uno esperaría, con el separador delante:

```
palabras = ["En", "un", "lugar"]

print(" ".join(palabras))
print("-".join(palabras))
print("".join(palabras))
```

```
En un lugar
En-un-lugar
Enunlugar
```

La pareja `split` y `join` es la manera normal de transformar un texto en Python: partir en palabras, trabajar con la lista, volver a pegar.

3. Archivos

Un archivo se abre con `open`, que recibe la ruta y el modo. Los modos son `"r"` para leer, `"w"` para escribir desde cero y `"a"` para agregar al final.

La forma correcta de abrir un archivo es con `with`, que lo cierra solo al terminar el bloque, incluso si algo falla dentro:

```
with open("quijote.txt", encoding="utf-8") as archivo:
    contenido = archivo.read()

print(len(contenido))
print(contenido[:60])
```

```
1101
En un lugar de la Mancha, de cuyo nombre no quiero acordarme
```

`read` devuelve todo el archivo como una sola cadena. El `with` se lee así: abre el archivo, llámalo `archivo` mientras dure este bloque, y ciérralo al salir. Fuera del bloque el archivo ya está cerrado, pero `contenido` sigue disponible porque es una variable normal.

La codificación

Ese `encoding="utf-8"` no es decoración. Un archivo de texto es en realidad una secuencia de bytes, y la codificación es la tabla que dice qué letra representa cada byte. UTF-8 es el estándar de hoy y el que debes usar siempre al crear archivos.

El problema es que todavía circulan archivos viejos en otras codificaciones. Si abres un archivo latino con la codificación equivocada, las tildes y las eñes salen rotas o el programa se cae. Cuando te pase, ya sabes qué buscar. Escribe `encoding="utf-8"` siempre, explícitamente, aunque tu sistema lo tenga por defecto, porque el programa tiene que funcionar también en la computadora del profesor.

Leer línea por línea

Un archivo se puede recorrer con `for` como cualquier secuencia, y entonces cada elemento es una línea. Esto es mejor que `read` para archivos grandes, porque no carga todo en memoria de una vez:

```
with open("quijote.txt", encoding="utf-8") as archivo:
    for numero, linea in enumerate(archivo, start=1):
        if numero > 3:
            break
        print(numero, repr(linea[:28]), "...", repr(linea[-4:]))
```

```
1 'En un lugar de la Mancha, de' ... 'que\n'
2 'vivía un hidalgo de los de l' ... 'lgo\n'
3 'corredor. Una olla de algo m' ... 's y\n'
```

Fíjate en el final de cada línea: leer una línea incluye el `\n` del salto. Por eso casi siempre se le hace `strip()` antes de trabajar con ella.

Escribir

Con el modo `"w"` se escribe. Cuidado, porque **borra el archivo si ya existía**:

```
with open("salida.txt", "w", encoding="utf-8") as archivo:
    archivo.write("primera línea\n")
    archivo.write("segunda línea\n")

with open("salida.txt", encoding="utf-8") as archivo:
    print(archivo.read())
```

```
primera línea
segunda línea
```

`write` no agrega el salto de línea; hay que ponerlo a mano.

4. Cuando las cosas salen mal

Todo lo anterior supone que el archivo existe. Si no existe, el programa se muere:

```
with open("no-existe.txt", encoding="utf-8") as archivo:
    print(archivo.read())
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'no-existe.txt'
```

Hasta ahora, cada error que hemos visto ha terminado el programa. Un programa serio no puede darse ese lujo: si el usuario escribe mal el nombre del archivo, lo correcto es decírselo y seguir, no volcarle un traceback en la pantalla.

Para eso está `try`. El bloque `try` contiene lo que puede fallar, y el `except` dice qué hacer si falla:

```
try:
    with open("no-existe.txt", encoding="utf-8") as archivo:
        contenido = archivo.read()
    print(f"El archivo tiene {len(contenido)} caracteres")
except FileNotFoundError:
    print("No encontré ese archivo")

print("y el programa sigue")
```

```
No encontré ese archivo
y el programa sigue
```

Esa última línea es el punto entero de la sección. El error ocurrió, el programa lo atendió y continuó.

`FileNotFoundError` es el nombre del tipo de error, el mismo que aparecía en la última línea del traceback. Cada tipo de problema tiene el suyo, y ya te has encontrado varios este curso:

ERROR	CUÁNDO
`ValueError`	`int("hola")`, el texto no representa un número
`IndexError`	`lista[10]` en una lista de tres elementos
`NameError`	usar una variable que no existe

ERROR	CUÁNDO
`ZeroDivisionError`	dividir entre cero
`FileNotFoundError`	abrir un archivo que no está
`TypeError`	`"5" + 3`, operar tipos incompatibles

Una función que atiende su propio error queda así. Un `try` puede llevar varios `except`, uno por cada tipo de problema, y se ejecuta el primero que empareja:

```
def leer_entero(texto):
    """Convierte texto a entero. Devuelve None si no se puede."""
    try:
        return int(texto)
    except ValueError:
        print(f"'{texto}' no es un número entero")
        return None

print(leer_entero("42"))
print(leer_entero("3.5"))
print(leer_entero("hola"))
```

```
42
'3.5' no es un número entero
None
'hola' no es un número entero
None
```

Una advertencia. Se puede escribir `except:` a secas, sin nombrar el error, y atrapa cualquier cosa. **No lo hagas.** Atrapa también los errores que no esperabas, incluidos tus propios fallos de programación, y los esconde. Nombra siempre el error que sabes manejar y deja que los demás se vean.

5. El proyecto: contar las palabras

Ya están todas las piezas. El programa abre un archivo y reporta cuántas palabras tiene, cuál es la más larga y cuál es el promedio de letras.

El primer problema es que `split` a secas no sirve: deja los signos de puntuación pegados, así que `Mancha,` y `Mancha` serían palabras distintas. Hay que limpiar antes de partir.

```
SIGNOS = ".,;:¿?¡!()«»\"'—-"

def limpiar(texto):
    """Devuelve el texto en minúsculas y sin signos de puntuación."""
    texto = texto.lower()

    for signo in SIGNOS:
        texto = texto.replace(signo, " ")
```

```
    return texto

print(limpiar("En un lugar de la Mancha, ¡de cuyo nombre...!"))
```

```
en un lugar de la mancha   de cuyo nombre
```

`SIGNOS` va en mayúsculas por convención: es una constante, un valor que se fija una vez y no cambia. Python no lo impide, pero el nombre en mayúsculas avisa al que lea.

Con eso, el resto sale corto:

```
def palabras_de(ruta):
    """Devuelve la lista de palabras de un archivo de texto."""
    with open(ruta, encoding="utf-8") as archivo:
        return limpiar(archivo.read()).split()

def mas_larga(palabras):
    """Devuelve la palabra más larga de la lista."""
    mayor = palabras[0]

    for palabra in palabras:
        if len(palabra) > len(mayor):
            mayor = palabra

    return mayor

def promedio_de_letras(palabras):
    """Devuelve el promedio de letras por palabra."""
    letras = 0

    for palabra in palabras:
        letras += len(palabra)

    return letras / len(palabras)

palabras = palabras_de("quijote.txt")

print(f"{len(palabras)} palabras")
print(f"la más larga es '{mas_larga(palabras)}'")
print(f"promedio de {promedio_de_letras(palabras):.2f} letras")
```

```
196 palabras
la más larga es 'entresemana'
promedio de 4.46 letras
```

`mas_larga` es el mismo acumulador del capítulo anterior, con `len(palabra)` en lugar del valor. Ese patrón, guardar el mejor visto hasta ahora, lo vas a escribir cien veces.

Y el programa principal, con la ruta pedida al usuario y el error atendido:

```
ruta = input("Archivo: ")

try:
    palabras = palabras_de(ruta)
except FileNotFoundError:
```

```
    print(f"No encontré el archivo {ruta}")
else:
    print(f"{len(palabras)} palabras")
    print(f"la más larga es '{mas_larga(palabras)}'")
```

El `else` de un `try` se ejecuta solo si no hubo error, y sirve para dejar dentro del `try` únicamente la línea que puede fallar. Cuanto más pequeño el `try`, más claro qué error estás atendiendo.

Queda una pregunta natural que todavía no podemos responder: **cuántas veces aparece cada palabra**. Podrías intentarlo con dos listas en paralelo, una de palabras y otra de conteos, y funcionaría, pero sería lento y incómodo. La estructura que resuelve eso es el diccionario, y es el capítulo siguiente.

6. Resumen

- Una cadena es una secuencia: `len`, índices desde cero, índices negativos, rebanadas y `for`, igual que una lista.
- En una cadena, `in` busca un pedazo, no solo un carácter.
- Las cadenas son inmutables. Todo método devuelve una cadena nueva y deja la original intacta.
- `split` convierte texto en lista de palabras; `join` hace lo contrario.
- Los archivos se abren con `with open(ruta, encoding="utf-8")`, que los cierra solo.
- El modo `"w"` borra lo que había. Recorrer el archivo con `for` da una línea por vuelta, con su salto de línea incluido.
- `try` y `except` permiten que un programa atienda un error en vez de morirse. Nombra siempre el error concreto; `except:` a secas esconde tus propios fallos.

Ejercicios

1. Escribe `es_palindromo(frase)` que diga si una frase se lee igual al derecho y al revés, ignorando mayúsculas, espacios y signos. Debe dar `True` con `"Dábale arroz a la zorra el abad"`.
2. Escribe `contar_lineas(ruta)`, `contar_palabras(ruta)` y `contar_caracteres(ruta)`. Compara tus resultados con los del comando `wc` de la terminal sobre el mismo archivo, y explica cualquier diferencia.
3. Escribe `capitalizar(frase)` que ponga en mayúscula la primera letra de cada palabra, sin usar el método `title` ni `capitalize`. Usa `split`, un ciclo y `join`.
4. Escribe un programa que lea `quijote.txt` y escriba `quijote-numerado.txt` con el mismo contenido pero con cada línea precedida de su número y un espacio.

5. Escribe `palabras_largas(ruta, minimo=8)` que devuelva la lista de las palabras distintas del archivo con al menos `minimo` letras, sin repeticiones y en orden alfabético. Puedes usar `sorted`.
6. Escribe un programa que le pida números al usuario uno por línea hasta una línea vacía, y que sume solo los que sean válidos, avisando de cada entrada mala sin terminar. Debe sobrevivir a que el usuario escriba `hola`, `3.5` y `--`.
7. Sin ejecutarlos, di qué imprime cada uno de estos tres programas. Después compruébalo.

   (a)

```
s = "hola"
s.upper()
print(s)
```

   (b)

```
print(len("a,b,,c".split(",")))
print(len("a b  c".split()))
```

   (c)

```
try:
    print(int("7") + int("ocho"))
except ValueError:
    print("mal")
except TypeError:
    print("peor")
```

6

Diccionarios y conjuntos

El capítulo anterior terminó con una pregunta que no pudimos responder: cuántas veces aparece cada palabra de un texto. Aquí la respondemos con el **diccionario**, la colección que asocia cada clave con un valor, y con su pariente cercano, el **conjunto**, que guarda elementos sin repetir. El proyecto del capítulo es un programa que lee un archivo y reporta las diez palabras más frecuentes.

Trabajamos sobre dos archivos que están en la carpeta `conferences/2026/libro` del repositorio del curso: `quijote.txt`, los tres párrafos del capítulo anterior, y `quijote-cap1.txt`, el primer capítulo completo del Quijote, tomado de la edición del Proyecto Gutenberg. El primero es pequeño para poder seguirlo a mano; el segundo es lo bastante grande para que las diferencias de velocidad se vean.

1. El problema, con lo que ya sabemos

Empecemos por lo que sí sabemos hacer. Con dos listas en paralelo, una de palabras vistas y otra de conteos, la palabra de la posición `i` en la primera tiene su conteo en la posición `i` de la segunda. Las funciones `limpiar` y `palabras_de` son las del capítulo anterior:

```
SIGNOS = ".,;:¿?¡!()«»\"'—-"

def limpiar(texto):
    """Devuelve el texto en minúsculas y sin signos de puntuación."""
    texto = texto.lower()

    for signo in SIGNOS:
        texto = texto.replace(signo, " ")

    return texto

def palabras_de(ruta):
    """Devuelve la lista de palabras de un archivo de texto."""
    with open(ruta, encoding="utf-8") as archivo:
        return limpiar(archivo.read()).split()

def frecuencias_con_listas(palabras):
    """Devuelve dos listas paralelas: las palabras distintas y sus conteos."""
    vistas = []
    conteos = []

    for palabra in palabras:
        if palabra in vistas:
            i = vistas.index(palabra)
            conteos[i] += 1
        else:
            vistas.append(palabra)
            conteos.append(1)
```

```
    return vistas, conteos

vistas, conteos = frecuencias_con_listas(palabras_de("quijote.txt"))

print(len(vistas), "palabras distintas")
print(vistas[:6])
print(conteos[:6])
```

```
126 palabras distintas
['en', 'un', 'lugar', 'de', 'la', 'mancha']
[5, 3, 1, 20, 4, 1]
```

Funciona. El método `index` de una lista devuelve la posición de un elemento, y la tupla que devuelve la función se desempaqueta en dos nombres, como vimos con las listas.

Pero mira lo que hace cada vuelta del ciclo. `palabra in vistas` recorre la lista de palabras vistas de una en una hasta encontrarla, y después `index` la recorre otra vez para saber dónde estaba. Cada palabra del texto paga un recorrido de todas las palabras distintas encontradas hasta ese momento. Y el programa es incómodo de leer, porque la relación entre una palabra y su conteo no está escrita en ningún sitio: solo la sostiene la coincidencia de posiciones.

Lo que de verdad queremos es una tabla donde se pueda preguntar "¿cuánto vale `"de"`?" y obtener la respuesta directamente.

2. El diccionario

Un diccionario es una colección de parejas **clave: valor**. Se escribe entre llaves, con dos puntos entre la clave y su valor:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

print(conteo)
print(conteo["de"])
print(len(conteo))
```

```
{'de': 20, 'que': 9, 'hidalgo': 2}
20
3
```

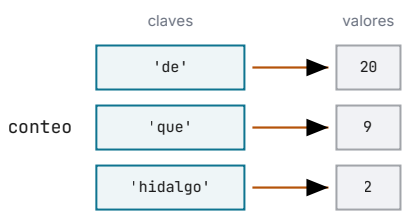


Figura 15. *Un diccionario no tiene posiciones. A cada valor se llega por su clave, y* `len` *cuenta las parejas.*

Se indexa con corchetes, igual que una lista, pero lo que va dentro de los corchetes no es una posición sino una **clave**. Pedir una clave que no existe es un error, el equivalente al `IndexError` de las listas:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}
print(conteo["rocín"])
```

```
Traceback (most recent call last):
  File "<stdin>", line 54, in <module>
KeyError: 'rocín'
```

Asignar a una clave hace una de dos cosas. Si la clave no estaba, **agrega** la pareja; si estaba, **reemplaza** su valor. Es la misma instrucción para los dos casos:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

conteo["rocín"] = 1
conteo["de"] = 21
conteo["hidalgo"] += 1

print(conteo)
```

```
{'de': 21, 'que': 9, 'hidalgo': 3, 'rocín': 1}
```

Esto es lo que en una lista no se puede hacer: `lista[10] = x` en una lista de tres elementos da `IndexError`, mientras que en un diccionario la clave nueva simplemente aparece.

Para preguntar si una clave está se usa `in`, que en un diccionario mira solo las claves, nunca los valores. Para quitar una pareja se usa `del`:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

print("hidalgo" in conteo)
print(20 in conteo)

del conteo["que"]
print(conteo)
```

```
True
False
{'de': 20, 'hidalgo': 2}
```

Y un diccionario vacío se escribe `{}`, que es por donde empieza casi todo diccionario que se llena dentro de un ciclo.

Las claves no pueden cambiar

Un diccionario se modifica, igual que una lista: se agregan parejas, se cambian valores y se quitan. Pero las **claves** tienen que ser de un tipo que no se pueda modificar: números, cadenas, tuplas. Una lista no sirve como clave:

```
posiciones = {}
posiciones[[3, 4]] = "reina"
```

```
Traceback (most recent call last):
  File "<stdin>", line 68, in <module>
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

La tupla, en cambio, sí:

```
tablero = {}
tablero[(3, 4)] = "reina"
tablero[(0, 0)] = "torre"

print(tablero[(3, 4)])
print((5, 5) in tablero)
```

```
reina
False
```

Ahí tienes un uso de la tupla que no vimos en el capítulo 4: una pareja de coordenadas como clave. El porqué de la regla, y de la palabra *unhashable* en el error, lo vemos en la sección 7.

3. Recorrer un diccionario

Un `for` sobre un diccionario recorre sus claves, en el orden en que se agregaron:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

for palabra in conteo:
    print(palabra, conteo[palabra])
```

```
de 20
que 9
hidalgo 2
```

Cuando hacen falta la clave y el valor a la vez, que es casi siempre, el método `items` da las parejas como tuplas, y el `for` las desempaqueta igual que con `enumerate`:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

for palabra, veces in conteo.items():
    print(f"{palabra:>8} aparece {veces} veces")
```

```
      de aparece 20 veces
     que aparece 9 veces
 hidalgo aparece 2 veces
```

Los métodos `keys` y `values` dan solo las claves o solo los valores. El segundo es el útil, porque los valores no se pueden pedir de otro modo:

```
conteo = {"de": 20, "que": 9, "hidalgo": 2}

print(sum(conteo.values()))
print(max(conteo.values()))
```

```
31
20
```

4. El patrón de conteo

Ya podemos escribir el conteo. Para cada palabra, si ya tiene entrada se le suma uno, y si no la tiene se crea con uno:

```
def frecuencias(palabras):
    """Devuelve un diccionario de cada palabra a las veces que aparece."""
    conteo = {}

    for palabra in palabras:
        if palabra in conteo:
            conteo[palabra] += 1
        else:
            conteo[palabra] = 1

    return conteo

print(frecuencias("de los de lanza en astillero".split()))
```

```
{'de': 2, 'los': 1, 'lanza': 1, 'en': 1, 'astillero': 1}
```

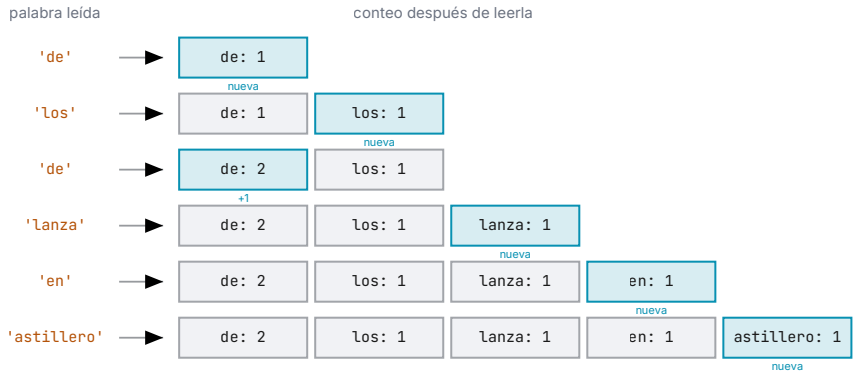


Figura 16. *Cada palabra leída toca una sola casilla: la crea si es nueva o le suma uno si ya estaba. El segundo* `"de"` *no agrega nada, incrementa.*

El `if` y el `else` se pueden fundir en una línea con el método `get`, que devuelve el valor de una clave o, si la clave no está, el valor por defecto que le pases en lugar de dar error:

```
conteo = {"de": 2, "los": 1}

print(conteo.get("de", 0))
print(conteo.get("rocín", 0))
```

```
2
0
```

Con `get`, el conteo entero queda así:

```
def frecuencias(palabras):
    """Devuelve un diccionario de cada palabra a las veces que aparece."""
    conteo = {}

    for palabra in palabras:
        conteo[palabra] = conteo.get(palabra, 0) + 1

    return conteo
```

Las dos versiones hacen lo mismo. La primera dice mejor lo que pasa; la segunda es la que vas a ver escrita en todas partes, y conviene reconocerla.

Contra las listas paralelas

Pongamos las dos soluciones a correr sobre el capítulo entero, con el reloj de la capítulo 3:

```
import time

palabras = palabras_de("quijote-cap1.txt")

inicio = time.perf_counter()
vistas, conteos = frecuencias_con_listas(palabras)
con_listas = time.perf_counter() - inicio
```

```
inicio = time.perf_counter()
conteo = frecuencias(palabras)
con_diccionario = time.perf_counter() - inicio

print(f"{len(palabras)} palabras, {len(conteo)} distintas")
print(f"con dos listas:     {con_listas * 1000:.2f} ms")
print(f"con un diccionario: {con_diccionario * 1000:.2f} ms")
print(f"el diccionario es {con_listas / con_diccionario:.0f} veces más rápido")
```

```
1895 palabras, 719 distintas
con dos listas:     7.09 ms
con un diccionario: 0.24 ms
el diccionario es 29 veces más rápido
```

El resultado es el mismo y el diccionario es decenas de veces más rápido sobre un solo capítulo. La diferencia crece con el texto, porque con las listas cada palabra recorre una lista de vistas que no para de crecer, y con el diccionario no recorre nada. Por qué no recorre nada es la pregunta de la sección 7.

5. Las diez más comunes

Tener el conteo no es tener la respuesta. Falta encontrar las palabras con más apariciones, y eso es el acumulador del capítulo anterior, el que guardaba la palabra más larga vista hasta ahora, cambiando la longitud por el conteo:

```
def mas_comun(conteo):
    """Devuelve la pareja (palabra, veces) con más apariciones."""
    mejor, maximo = "", 0

    for palabra, veces in conteo.items():
        if veces > maximo:
            mejor, maximo = palabra, veces

    return mejor, maximo

print(mas_comun(frecuencias(palabras_de("quijote.txt"))))
```

```
('de', 20)
```

Para las diez primeras basta repetir eso diez veces, sacando del diccionario cada ganador para que la vuelta siguiente encuentre al segundo:

```
def mas_comunes(conteo, k):
    """Devuelve las k parejas (palabra, veces) más frecuentes, de mayor a menor."""
    quedan = dict(conteo)
    resultado = []

    for vuelta in range(k):
        palabra, veces = mas_comun(quedan)
        resultado.append((palabra, veces))
```

```
            del quedan[palabra]

    return resultado

for palabra, veces in mas_comunes(frecuencias(palabras_de("quijote-cap1.txt")), 10):
    print(f"{palabra:>6} {veces:4}")
```

```
    de  122
     y  106
   que   89
     a   44
    el   40
    en   38
    la   35
    su   35
    se   30
   con   27
```

Mira la primera línea de la función. `dict(conteo)` hace una **copia** del diccionario, y es imprescindible: el diccionario se modifica igual que una lista, y si borráramos directamente de `conteo` vaciaríamos el diccionario del que nos llamó. Es la sorpresa de los dos nombres para una misma lista del capítulo 4, ahora con diccionarios.

Este método de buscar el mayor, apartarlo y volver a buscar es una forma de ordenar, y el capítulo siguiente le pone nombre y le contamos las operaciones.

El resultado, de todos modos, decepciona: `de`, `y`, `que`, `a`, `el`. Las palabras más frecuentes de cualquier texto en español son las que no dicen nada. Para encontrar las que hablan del capítulo hay que sacar esas primero, y para eso hace falta una colección nueva.

6. Conjuntos

Un **conjunto** es una colección sin repeticiones y sin orden, como los conjuntos de la matemática. Se escribe entre llaves, sin los dos puntos:

```
vocales = {"a", "e", "i", "o", "u"}
dados = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}

print(len(vocales))
print(dados)
print(4 in dados)
```

```
5
{1, 2, 3, 4, 5, 6, 9}
True
```

El 1, el 3 y el 5 aparecen varias veces en lo que escribimos, y el conjunto los guarda una sola vez. Esa es la operación por la que más se usan: convertir una lista en conjunto elimina los repetidos.

```
palabras = palabras_de("quijote-cap1.txt")

print(len(palabras), "palabras")
print(len(set(palabras)), "palabras distintas")
```

```
1895 palabras
719 palabras distintas
```

Un conjunto vacío se escribe `set()`, porque {} ya estaba tomado por el diccionario vacío. Se le agregan elementos con `add`, y agregar uno que ya estaba no hace nada:

```
vistas = set()
vistas.add("rocín")
vistas.add("hidalgo")
vistas.add("rocín")

print(vistas)
print(type({}))
```

```
{'rocín', 'hidalgo'}
<class 'dict'>
```

Lo que un conjunto no tiene es posiciones. No se puede pedir `vistas[0]`, porque no hay primer elemento, y el orden en que un conjunto imprime sus elementos no tiene por qué ser el orden en que se agregaron:

```
print(vistas[0])
```

```
Traceback (most recent call last):
  File "<stdin>", line 184, in <module>
TypeError: 'set' object is not subscriptable
```

Las operaciones de conjuntos

Los conjuntos traen las operaciones de la matemática, escritas con operadores: `|` para la unión, `&` para la intersección y `-` para la diferencia. Con ellas se comparan vocabularios. Partamos `quijote.txt` en sus tres párrafos, que en el archivo están separados por una línea vacía, y saquemos el vocabulario de cada uno:

```
with open("quijote.txt", encoding="utf-8") as archivo:
    parrafos = archivo.read().split("\n\n")
```

```
primero = set(limpiar(parrafos[0]).split())
tercero = set(limpiar(parrafos[2]).split())

print(len(primero), len(tercero))
print(len(primero | tercero), "en alguno de los dos")
print(sorted(primero & tercero))
print(len(primero - tercero), "solo en el primero")
```

```
50 49
92 en alguno de los dos
['de', 'en', 'hidalgo', 'la', 'los', 'que', 'y']
43 solo en el primero
```

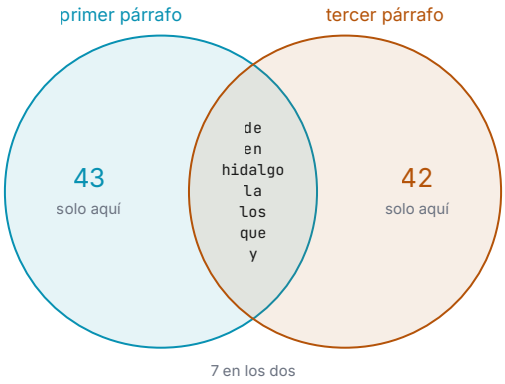


Figura 17. *Los vocabularios del primer y el tercer párrafo. Casi todo lo que comparten son palabras que no dicen nada; la única que habla del texto es* `hidalgo`.

`sorted` devuelve una lista ordenada, y aquí sirve para que el resultado se imprima en un orden fijo. La intersección es reveladora. Lo que dos párrafos cualesquiera tienen en común son las palabras vacías: artículos, preposiciones, conjunciones. Es la misma lista que encabezaba el conteo.

Sacar las palabras vacías

Así que la solución es un conjunto con las palabras que no queremos contar, y una pregunta con `in` antes de contar cada palabra:

```
VACIAS = {
    "a", "al", "así", "bien", "como", "con", "cual", "cuando", "de", "del",
    "dél", "el", "él", "en", "era", "estaba", "fue", "ha", "había", "la", "las",
    "le", "les", "lo", "los", "más", "mas", "me", "mi", "muy", "ni", "no", "o",
    "para", "pero", "por", "porque", "que", "quien", "se", "ser", "si", "sí",
    "sido", "sin", "su", "sus", "tan", "tenía", "todo", "todos", "un", "una",
    "y", "ya",
}

def frecuencias_sin_vacias(palabras):
    """Como `frecuencias`, pero sin contar las palabras de VACIAS."""
    conteo = {}

    for palabra in palabras:
```

```
        if palabra not in VACIAS:
            conteo[palabra] = conteo.get(palabra, 0) + 1

    return conteo

diez = mas_comunes(frecuencias_sin_vacias(palabras_de("quijote-cap1.txt")), 10)

for palabra, veces in diez:
    print(f"{palabra:>9} {veces:4}")
```

```
   nombre   13
caballero   13
      don    6
    razón    6
     vino    6
    rocín    5
  vuestra    5
   amadís    5
      dio    5
   celada    5
```

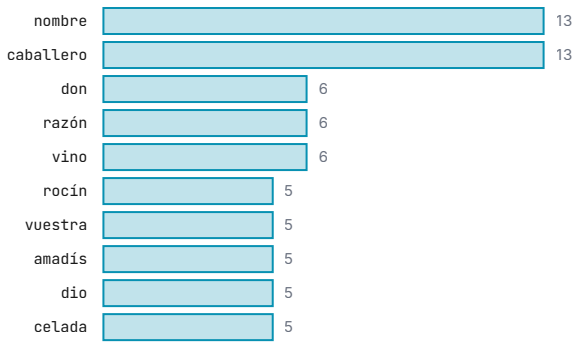


Figura 18. *Las diez palabras más frecuentes del primer capítulo, sin las vacías. El capítulo es la historia de cómo un hidalgo se busca un nombre de caballero, y lo mismo hace con su rocín y con su dama.*

Ahora sí habla el capítulo: el hidalgo pasa días pensando qué **nombre** ponerse como **caballero** andante, lo mismo hace con su **rocín** y con su dama, y el modelo que tiene en la cabeza es **Amadís** de Gaula. La lista `VACIAS` no es una verdad del idioma sino una decisión: la armamos mirando el resultado y agregando lo que estorbaba. Así se trabaja con datos reales.

Fíjate que `VACIAS` es un conjunto y no una lista. Funcionaría igual con corchetes, pero la pregunta `palabra not in VACIAS` se hace una vez por cada palabra del texto, y en un conjunto esa pregunta es mucho más barata. Cuánto, es lo que sigue.

7. Por qué buscar una clave no cuesta nada

Medimos. La función de abajo pregunta muchas veces si un valor está en una colección y devuelve cuánto tardó cada pregunta. Buscamos el `-1`, que no está, así que la lista tiene que mirar todos sus elementos para poder decir que no:

```
import time

def segundos_por_busqueda(coleccion, x, veces):
    """Devuelve el tiempo medio que tarda `x in coleccion`."""
    inicio = time.perf_counter()

    for vuelta in range(veces):
        encontrado = x in coleccion

    return (time.perf_counter() - inicio) / veces

medidas = []

for n in [1_000, 10_000, 100_000, 1_000_000]:
    lista = list(range(n))
    conjunto = set(lista)

    en_lista = segundos_por_busqueda(lista, -1, 50)
    en_conjunto = segundos_por_busqueda(conjunto, -1, 100_000)
    medidas.append((n, en_lista, en_conjunto))

    print(f"{n:>9} elementos: lista {en_lista * 1e3:7.3f} ms,"
          f" conjunto {en_conjunto * 1e9:4.0f} ns")
```

```
     1000 elementos: lista   0.007 ms, conjunto   50 ns
    10000 elementos: lista   0.071 ms, conjunto   32 ns
   100000 elementos: lista   0.832 ms, conjunto   38 ns
  1000000 elementos: lista   9.753 ms, conjunto   33 ns
```

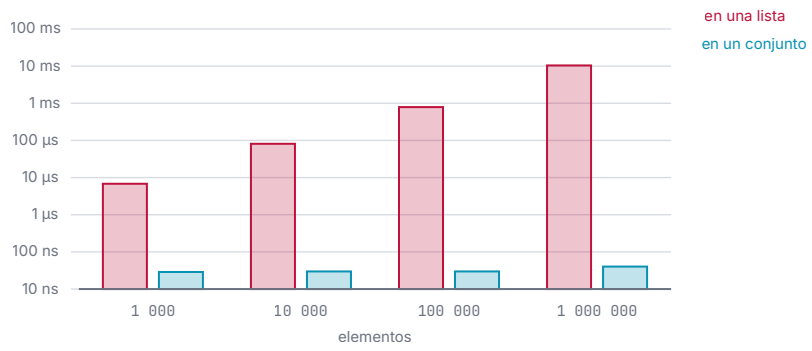


Figura 19. *Tiempo de una búsqueda, medido al construir el libro. La escala es logarítmica: cada raya vale diez veces la de abajo. La lista sube una raya cada vez que el tamaño se multiplica por diez; el conjunto se queda donde estaba.*

Un guion bajo dentro de un número, como en `1_000_000`, no cambia su valor; solo lo hace más fácil de leer.

La lista crece en proporción a su tamaño: diez veces más elementos, diez veces más tiempo. El conjunto tarda lo mismo con mil elementos que con un millón. Un diccionario se comporta exactamente igual al buscar una clave, porque por dentro están hechos de la misma manera.

La idea, sin entrar en cómo se implementa, es esta. Una lista guarda sus elementos en fila y, para encontrar uno, no le queda más remedio que mirarlos de uno en uno. Un conjunto o un

diccionario, en cambio, calcula **a partir del propio valor** un número que dice en qué casillero guardarlo. Al buscar, hace la misma cuenta, va directo a ese casillero y mira solo ahí. Es como un estante de casilleros con el nombre de cada estudiante, donde no hace falta revisar todos los casilleros para encontrar el tuyo: tu nombre te dice cuál es.

Ese número calculado a partir del valor se llama en inglés *hash*, y ahí está la razón de la regla de la sección 2. Si una clave pudiera cambiar después de guardarla, su número cambiaría con ella y el diccionario la buscaría en el casillero equivocado. Por eso una lista, que se modifica, es *unhashable*: no se le puede calcular un casillero fijo. Cómo se calcula ese número y qué pasa cuando dos valores caen en el mismo casillero es materia del segundo semestre.

8. El proyecto completo

El programa pide la ruta del archivo, atiende el error si no existe y reporta las diez palabras más frecuentes, con sus conteos alineados en columna:

```
def reporte(ruta, k=10):
    """Imprime las k palabras más frecuentes del archivo, sin las vacías."""
    palabras = palabras_de(ruta)
    conteo = frecuencias_sin_vacias(palabras)

    print(f"{len(palabras)} palabras, {len(set(palabras))} distintas")

    for palabra, veces in mas_comunes(conteo, k):
        print(f"  {palabra:<10} {veces:3}")

reporte("quijote-cap1.txt")
```

```
1895 palabras, 719 distintas
  nombre      13
  caballero   13
  don          6
  razón        6
  vino         6
  rocín        5
  vuestra      5
  amadís       5
  dio          5
  celada       5
```

```
ruta = input("Archivo: ")

try:
    reporte(ruta)
except FileNotFoundError:
    print(f"No encontré el archivo {ruta}")
```

El `<10` dentro del f-string alinea la palabra a la izquierda en diez espacios, y el `3` alinea el número a la derecha en tres. Es lo que convierte una lista de valores en una tabla legible.

El programa tiene un fallo que dejamos como ejercicio. Si pides más palabras de las que el texto tiene distintas, `mas_comun` recibe un diccionario vacío, devuelve la pareja `("", 0)`, y el `del` de la línea siguiente falla.

9. Resumen

- Un diccionario asocia claves con valores. `d[clave]` lee, `d[clave] = valor` agrega o reemplaza, `del d[clave]` quita, y una clave que no está da `KeyError`.
- `in` en un diccionario pregunta por las claves, nunca por los valores.
- `get(clave, defecto)` devuelve el valor o el defecto sin dar error. El patrón de conteo es `conteo[x] = conteo.get(x, 0) + 1`.
- `for clave, valor in d.items()` recorre las parejas, en el orden en que se agregaron.
- Las claves tienen que ser inmutables: números, cadenas o tuplas. Una lista no sirve.
- Un diccionario se modifica como una lista, así que dos nombres pueden compartirlo. `dict(d)` hace una copia.
- Un conjunto guarda elementos sin repetir y sin orden. `set(lista)` quita los repetidos; `set()` es el conjunto vacío, porque `{}` es el diccionario vacío.
- `|`, `&` y `-` son la unión, la intersección y la diferencia.
- Buscar en un conjunto o en un diccionario cuesta lo mismo con diez elementos que con un millón. Buscar en una lista cuesta en proporción a su tamaño.

Ejercicios

1. Escribe `frecuencia_de_letras(ruta)` que devuelva un diccionario de cada letra a las veces que aparece en el archivo, sin contar espacios ni signos. ¿Cuál es la vocal más frecuente en `quijote-cap1.txt`? ¿Y la consonante?
2. Corrige `mas_comunes` para que, si `k` es mayor que el número de palabras distintas, devuelva todas las que hay en lugar de fallar.
3. Escribe `agrupar_por_longitud(palabras)` que devuelva un diccionario de cada longitud a la lista de palabras distintas de esa longitud. Por ejemplo, la clave `11` del primer capítulo debe incluir `entresemana`.
4. Dos palabras son anagramas si tienen las mismas letras con las mismas repeticiones, como `roma` y `amor`. Escribe `son_anagramas(a, b)` comparando dos diccionarios de conteo de letras. Después escribe `grupos_de_anagramas(palabras)` usando como clave `"".join(sorted(palabra))`, y busca los grupos de más de una palabra en el capítulo.
5. Escribe `invertir(conteo)` que reciba un diccionario de palabras a conteos y devuelva uno de cada conteo a la lista de palabras con ese conteo. ¿Cuántas palabras del capítulo aparecen una sola vez?

6. Tienes la lista de los estudiantes de un grupo y las listas de los que aprobaron el primer y el segundo examen. Con operaciones de conjuntos, di quiénes aprobaron los dos, quiénes aprobaron exactamente uno y quiénes no aprobaron ninguno.
7. Escribe `traducir(frase, diccionario)` que traduzca palabra a palabra con un diccionario como `{"hidalgo": "nobleman", "rocín": "nag"}`, dejando igual las palabras que no estén. Resuélvelo con `get` y sin ningún `if`.
8. Sin ejecutarlos, di qué imprime cada uno de estos tres programas. Después compruébalo.

   (a)

```
d = {"a": 1, "b": 2}
d["a"] = d.get("c", 10) + d["b"]
print(d)
```

   (b)

```
print(len({1, 2, 2, 3, 3, 3}))
print(len({"uno": 1, "uno": 2}))
```

   (c)

```
a = {"x": 1}
b = a
b["x"] = 2
print(a)
```

7

Búsqueda, ordenación y costo

El capítulo 6 terminó con una función que encontraba las diez palabras más frecuentes de un capítulo del Quijote. Lo hacía así: buscar la más frecuente, apartarla, buscar la más frecuente de las que quedan, apartarla, y así diez veces. Al final dijimos que eso era una forma de **ordenar**, y que aquí le pondríamos nombre.

Este capítulo hace tres cosas. Primero, **buscar**: decir si un valor está en una lista y dónde. Segundo, **ordenar**: poner los elementos de una lista de menor a mayor, escrito por nosotros, sin `sort` ni `sorted`. Tercero, **contar**: cuánto trabajo hace cada uno de esos programas, y qué le pasa a ese trabajo cuando la lista crece. Esa tercera parte es la importante. Las dos primeras son la excusa.

Python ya trae `in`, `index`, `sort` y `sorted`, y en un programa de verdad se usan esos. Los escribimos a mano por la misma razón por la que en el capítulo 4 escribimos `sum` y `max`: para saber qué hacen por dentro, y porque los algoritmos de este capítulo son los primeros sobre los que se puede razonar cuánto cuestan.

1. Búsqueda lineal

Dada una lista y un valor, queremos el índice donde está el valor, o `-1` si no está. La manera obvia es mirar los elementos uno por uno:

```
def buscar(lista, x):
    """Devuelve el primer índice donde está x, o -1 si no está."""
    for i in range(len(lista)):
        if lista[i] == x:
            return i
    return -1

edades = [19, 18, 21, 18, 20, 22]
print(buscar(edades, 21))
print(buscar(edades, 18))
print(buscar(edades, 30))
```

```
2
1
-1
```

El `return` dentro del ciclo es el del capítulo 3: en cuanto aparece el valor, la función termina, y el ciclo con ella. Solo si el ciclo se acaba sin encontrarlo se llega al `return -1`.

¿Cuánto trabajo hace? Depende de dónde esté el valor. Si está primero, una comparación. Si está último o no está, tantas comparaciones como elementos tenga la lista. Para verlo, contamos:

```
def buscar_contando(lista, x):
    """Como buscar, pero devuelve también cuántas comparaciones hizo."""
    comparaciones = 0
    for i in range(len(lista)):
        comparaciones += 1
        if lista[i] == x:
            return i, comparaciones
    return -1, comparaciones

numeros = list(range(1_000_000))
print(buscar_contando(numeros, 0))
print(buscar_contando(numeros, 500_000))
print(buscar_contando(numeros, -5))
```

```
(0, 1)
(500000, 500001)
(-1, 1000000)
```

Cuando el valor no está, hay que mirar **todos** los elementos para estar seguro, porque el que no miraste podía ser el que buscabas. Con una lista de un millón son un millón de comparaciones, y con una de dos millones serían dos millones. El trabajo crece en proporción al tamaño de la lista, y por eso se llama búsqueda **lineal**. Es lo que hace `x in lista` por dentro, y es la razón de que en el capítulo 6 preguntar si una palabra estaba en una lista fuera tan lento comparado con un conjunto.

No se puede hacer mejor sin saber algo más de la lista. Si los elementos están en cualquier orden, cualquier elemento que no mires podría ser el que buscas.

2. Búsqueda binaria

Pero a veces sí sabemos algo más: que la lista está **ordenada**. En el capítulo 2 la computadora adivinaba un número entre 1 y 100 en siete preguntas, partiendo siempre el rango a la mitad. Es la misma idea. Si miro el elemento del medio de una lista ordenada y es menor que lo que busco, todo lo que está a su izquierda también es menor, y puedo descartar esa mitad entera sin mirarla.

Llevamos dos índices, `izq` y `der`, que marcan el pedazo de la lista donde el valor todavía puede estar. Al principio es la lista entera:

```
def buscar_binaria(lista, x):
    """Busca x en una lista ordenada. Devuelve su índice, o -1."""
    izq = 0
    der = len(lista) - 1

    while izq <= der:
```

```
        medio = (izq + der) // 2
        if lista[medio] == x:
            return medio
        elif lista[medio] < x:
            izq = medio + 1      # x, si está, está a la derecha de medio
        else:
            der = medio - 1      # x, si está, está a la izquierda de medio

    return -1

ordenada = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
print(buscar_binaria(ordenada, 13))
print(buscar_binaria(ordenada, 2))
print(buscar_binaria(ordenada, 29))
print(buscar_binaria(ordenada, 12))
```

```
5
0
9
-1
```

Haz a mano la búsqueda del 13. Al principio `izq = 0` y `der = 9`, el medio es 4 y `ordenada[4]` es 11, menor que 13: el 13 tiene que estar a la derecha, así que `izq = 5`. Ahora el medio es 7, `ordenada[7]` es 19, mayor: `der = 6`. El medio es 5, `ordenada[5]` es 13, y lo encontramos.

Cuando el valor no está, como el 12, los índices se cruzan: `izq` termina a la derecha de `der`, el pedazo donde podía estar queda vacío y el `while` se acaba.

Ahora contemos:

```
def buscar_binaria_contando(lista, x):
    """Como buscar_binaria, pero devuelve también cuántas vueltas dio."""
    izq, der = 0, len(lista) - 1
    vueltas = 0
    while izq <= der:
        vueltas += 1
        medio = (izq + der) // 2
        if lista[medio] == x:
            return medio, vueltas
        elif lista[medio] < x:
            izq = medio + 1
        else:
            der = medio - 1
    return -1, vueltas

for n in [1_000, 1_000_000, 1_000_000_000]:
    print(f"{n:>13}: {buscar_binaria_contando(range(n), -5)[1]} vueltas")
```

```
         1000: 9 vueltas
      1000000: 19 vueltas
   1000000000: 29 vueltas
```

Le pasamos `range(n)` en lugar de una lista porque un `range` se indexa igual que una lista ordenada y no ocupa memoria; una lista de mil millones de enteros no cabe en la máquina del

laboratorio. Para un millón, la búsqueda lineal hace un millón de comparaciones y la binaria diecinueve vueltas. Multiplicar la lista por mil le añade exactamente diez vueltas, porque cada vuelta parte el pedazo a la mitad, y diez mitades seguidas dividen por $2^{10} = 1024$.

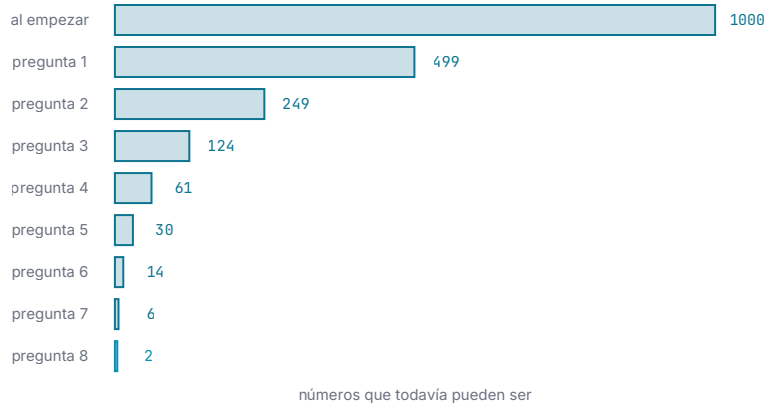


Figura 20. *Buscar un valor que no está en una lista ordenada de 1000 elementos. Cada vuelta del ciclo deja vivo, como mucho, la mitad del pedazo anterior.*

La cantidad de veces que hay que partir n a la mitad hasta llegar a 1 es el **logaritmo en base 2** de n, que se escribe $\log_2 n$. Es la función inversa de 2^k: $\log_2 1024 = 10$, $\log_2 1\,048\,576 = 20$. Crece tan despacio que, a efectos prácticos, una búsqueda binaria no cuesta nada.

La tabla dio 9 vueltas para mil y no 10, y 19 para un millón y no 20. No es un error. Cuando n es par, las dos mitades no son iguales: si quitas el medio de 1000 elementos, quedan 499 a la izquierda y 500 a la derecha. Buscando un valor menor que todos, la búsqueda se va siempre al pedazo izquierdo, el más chico, y llega antes al final. Las vueltas de una búsqueda binaria son $\log_2 n$ redondeado, a veces una menos.

El precio es que la lista tiene que estar ordenada. Si está desordenada, la búsqueda binaria devuelve respuestas falsas sin dar ningún error:

```
print(buscar_binaria([7, 2, 19, 5, 13], 5))
```

```
-1
```

El 5 está en la posición 3, y la función dice que no está: mira el 19 del medio, que es mayor que 5, y descarta toda la derecha, donde estaba el 5. Mirar `lista[medio]` y descartar una mitad solo es correcto si la lista está ordenada. Si no lo está, la función hace lo mismo, pero el resultado ya no significa nada.

Un error plantado

La búsqueda binaria es famosa porque es fácil de escribir mal. Esta versión cambia una sola cosa, `izq = medio` en lugar de `izq = medio + 1`, que parece inofensiva: el medio ya se miró, ¿qué más da dejarlo dentro?

```
def buscar_binaria_mal(lista, x):
    izq, der = 0, len(lista) - 1
    while izq <= der:
        medio = (izq + der) // 2
        if lista[medio] == x:
            return medio
        elif lista[medio] < x:
            izq = medio          # antes era medio + 1
        else:
            der = medio - 1
    return -1
```

Busca el 9 en `[3, 7]`: `izq = 0`, `der = 1`, el medio es 0, `lista[0]` es 3, menor que 9, así que `izq = 0`. No cambió nada. La vuelta siguiente es idéntica a la anterior, y la función no termina nunca. Lo vemos dando solo seis vueltas:

```
lista, x = [3, 7], 9
izq, der = 0, len(lista) - 1
for vuelta in range(6):
    medio = (izq + der) // 2
    if lista[medio] < x:
        izq = medio
    print(f"vuelta {vuelta}: izq={izq} der={der} medio={medio}")
```

```
vuelta 0: izq=0 der=1 medio=0
vuelta 1: izq=0 der=1 medio=0
vuelta 2: izq=0 der=1 medio=0
vuelta 3: izq=0 der=1 medio=0
vuelta 4: izq=0 der=1 medio=0
vuelta 5: izq=0 der=1 medio=0
```

¿Es un caso raro? Lo medimos. Esta versión de la función con el error se rinde a las cien vueltas y devuelve `None`, para que podamos contar cuántas búsquedas se quedan colgadas en una lista de diez números:

```
def buscar_binaria_mal(lista, x):
    izq, der = 0, len(lista) - 1
    vueltas = 0
    while izq <= der:
        vueltas += 1
        if vueltas > 100:
            return None              # se quedó dando vueltas
        medio = (izq + der) // 2
        if lista[medio] == x:
            return medio
        elif lista[medio] < x:
            izq = medio
```

```
        else:
            der = medio - 1
    return -1

pares = list(range(0, 20, 2))       # [0, 2, 4, ..., 18]
colgadas = []
for x in range(-1, 21):
    if buscar_binaria_mal(pares, x) is None:
        colgadas.append(x)

print(len(colgadas), "de", len(range(-1, 21)), "búsquedas colgadas:", colgadas)
```

```
14 de 22 búsquedas colgadas: [1, 3, 5, 6, 7, 9, 10, 11, 13, 15, 17, 18, 19, 20]
```

Más de la mitad, entre ellas valores que sí están en la lista. El error aparece cada vez que el pedazo queda de uno o dos elementos y `lista[medio] < x`: en esos dos casos el medio es el propio `izq`, y `izq = medio` no lo mueve. Cualquier búsqueda de un valor mayor que todos llega a esa situación. Así que este error no es difícil de atrapar, lo atrapa casi cualquier prueba. Lo difícil es acordarse de hacer las pruebas.

Lo que garantiza que un `while` termina es que cada vuelta achique algo, y aquí el pedazo `der - izq` tiene que achicarse. Con `medio + 1` y `medio - 1` siempre se achica, porque el medio queda fuera. Con `izq = medio`, a veces no. Por eso conviene probar siempre con listas de cero, uno y dos elementos, y con valores menores que todos, mayores que todos y que caen en medio, tanto valores que están como valores que no están.

3. Ordenación por selección

Si buscar en una lista ordenada es tanto más barato, vale la pena saber ordenar. El primer método es el del capítulo 6: buscar el menor, ponerlo primero, buscar el menor de los que quedan, ponerlo segundo, y así. Se llama **ordenación por selección**, porque en cada paso se selecciona el menor de lo que falta.

La diferencia con el capítulo 6 es que ahora no apartamos los elementos en otra lista y no borramos nada: los movemos dentro de la misma lista. Después de la pasada `i`, las posiciones `0` a `i` tienen los `i + 1` elementos más pequeños, ya en orden, y el resto está sin ordenar. La pasada `i` busca el menor del resto y lo intercambia con el que está en la posición `i`:

```
def seleccion(lista):
    """Ordena la lista de menor a mayor, modificándola."""
    n = len(lista)
    for i in range(n - 1):
        menor = i
        for j in range(i + 1, n):
            if lista[j] < lista[menor]:
                menor = j
        lista[i], lista[menor] = lista[menor], lista[i]

notas = [4, 2, 5, 3, 5, 2, 3]
```

```
seleccion(notas)
print(notas)
```

```
[2, 2, 3, 3, 4, 5, 5]
```

El intercambio `lista[i], lista[menor] = lista[menor], lista[i]` es el desempaquetado de tuplas del capítulo 4. La función no devuelve nada: modifica la lista que recibió, igual que `sort`, y el cambio se ve desde fuera porque las dos variables nombran la misma lista.

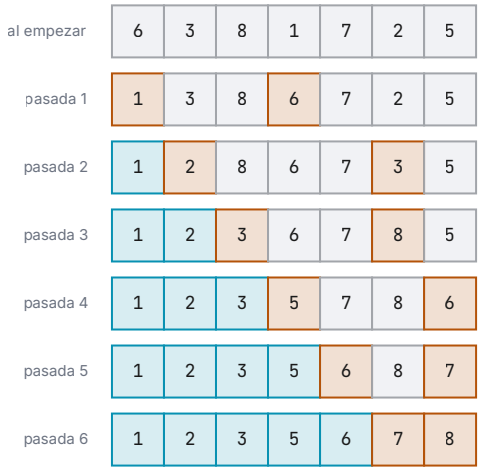


Figura 21. *Ordenación por selección de `[6, 3, 8, 1, 7, 2, 5]`. En azul, la parte ya ordenada. En naranja, los dos elementos que intercambió la pasada. Si el menor ya estaba en su sitio, la pasada no intercambia nada.*

¿Por qué el ciclo de fuera llega solo hasta `n - 2`? Porque cuando quedan los `n - 1` menores en su sitio, el último elemento es el mayor y ya está en su sitio también.

Ahora contemos las comparaciones `lista[j] < lista[menor]`. La primera pasada compara el primero con los otros $n - 1$. La segunda, $n - 2$. La última, 1. En total:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}.$$

No depende de cómo venga la lista. Aunque ya esté ordenada, la selección hace todas esas comparaciones, porque para saber que el primero es el menor tiene que compararlo con todos los demás. Para $n = 1000$ son casi medio millón.

4. Ordenación por inserción

El segundo método es el que usa casi todo el mundo para ordenar las cartas que tiene en la mano. Tomas las cartas una a una, y cada carta nueva la metes en su lugar entre las que ya tienes, que están ordenadas.

En la lista, después de la pasada `i` las posiciones `0` a `i` están ordenadas entre sí, aunque no sean todavía las más pequeñas de la lista. La pasada `i` toma el elemento de la posición `i` y lo va corriendo hacia la izquierda mientras el de su izquierda sea mayor:

```
def insercion(lista):
    """Ordena la lista de menor a mayor, modificándola."""
    for i in range(1, len(lista)):
        x = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > x:
            lista[j + 1] = lista[j]      # corre el mayor un lugar a la derecha
            j -= 1
        lista[j + 1] = x                 # y x cae en el hueco

notas = [4, 2, 5, 3, 5, 2, 3]
insercion(notas)
print(notas)
```

```
[2, 2, 3, 3, 4, 5, 5]
```

El `while` tiene dos condiciones, y su orden tiene que ver con una regla de `and` que no habíamos necesitado hasta ahora: si la primera condición es falsa, Python **no evalúa la segunda**, porque el resultado ya es falso sea lo que sea la segunda. Se llama **evaluación en cortocircuito**, y `or` hace lo mismo cuando la primera es verdadera. Aquí, cuando `j` llega a `-1`, `j >= 0` es falsa y nunca se mira `lista[-1]`. En Python esa lectura no daría error: `lista[-1]` es el último elemento de la lista, que no tiene nada que ver con el hueco que buscamos. Con las dos condiciones en el orden contrario la función seguiría ordenando bien, pero por casualidad: después de mirar ese elemento que no viene al caso, `j >= 0` cortaría el ciclo de todos modos. En otros ciclos el orden decide si el programa funciona. Esta búsqueda se detiene en `x` o al final de la lista:

```
i = 0
while i < len(lista) and lista[i] != x:
    i += 1
```

Con las condiciones al revés, cuando `x` no está, `i` llega a `len(lista)` y `lista[i] != x` se evalúa antes de preguntar si `i` es válido: `IndexError`. Por costumbre, la condición que protege el índice va primero.

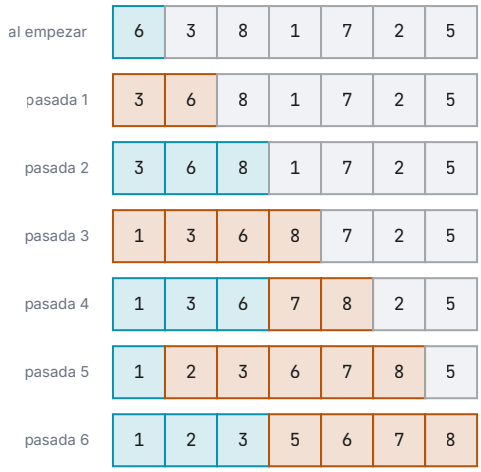


Figura 22. *Ordenación por inserción de la misma lista. En azul, el pedazo ordenado entre sí. En naranja, lo que se movió para hacerle hueco al elemento nuevo. Compárala con la figura de la selección: aquí el azul no son todavía los menores de la lista.*

Compara la figura 22 con la figura 21. En la selección, la parte azul ya es definitiva: esos elementos no se vuelven a mover. En la inserción, la parte azul está ordenada, pero cualquier elemento que llegue después puede meterse en medio.

El trabajo de la inserción depende de la lista. Si ya está ordenada, cada elemento se compara con el de su izquierda, ve que es menor, y no se mueve: $n - 1$ comparaciones en total. Si está al revés, cada elemento nuevo es menor que todos los anteriores y tiene que recorrerlos todos, y salen otra vez $\frac{n(n-1)}{2}$. Lo comprobamos:

```
def insercion_contando(lista):
    """Ordena la lista y devuelve cuántas comparaciones hizo."""
    comparaciones = 0
    for i in range(1, len(lista)):
        x = lista[i]
        j = i - 1
        while j >= 0:
            comparaciones += 1
            if lista[j] <= x:
                break
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = x
    return comparaciones

n = 1000
random.seed(8)
al_azar = [random.randint(1, 10_000) for _ in range(n)]

print("ordenada:   ", insercion_contando(list(range(n))))
print("al revés:   ", insercion_contando(list(range(n, 0, -1))))
print("al azar:    ", insercion_contando(list(al_azar)))
print("n(n-1)/2:   ", n * (n - 1) // 2)
```

```
ordenada:    999
al revés:    499500
```

```
al azar:     248214
n(n-1)/2:    499500
```

Para contar hubo que cambiar la forma del ciclo: la comparación `lista[j] > x` pasó del `while` a un `if` con `break`, para poder sumar uno cada vez que se hace. Lo que hace la función es lo mismo.

La lista al azar se construyó con una forma nueva, que vamos a usar mucho a partir de ahora:

```
al_azar = [random.randint(1, 10_000) for _ in range(n)]
```

Se lee "la lista de `random.randint(1, 10_000)`, una vez por cada vuelta de `range(n)`", y es una abreviatura de lo que ya sabes escribir:

```
al_azar = []
for _ in range(n):
    al_azar.append(random.randint(1, 10_000))
```

Se llama **lista por comprensión**. La variable del ciclo se llama `_` cuando no se usa para nada, que es una convención para decir justo eso. Si se usa, puede aparecer en la expresión: `[k * k for k in range(5)]` es `[0, 1, 4, 9, 16]`. El módulo `random` trae `randint(a, b)`, que devuelve un entero al azar entre `a` y `b`, ambos incluidos, y `random.seed(8)` fija la semilla para que la lista "al azar" sea la misma cada vez que corre el programa, y los números de este libro se puedan repetir.

Con una lista al azar, la inserción hace alrededor de la mitad del peor caso. Tiene sentido: en promedio, cada elemento nuevo tiene la mitad de los anteriores a su derecha. La mitad de $\frac{n(n-1)}{2}$ sigue siendo una cantidad que se multiplica por cuatro cuando n se duplica, y eso es lo que importa, como veremos en la sección 6.

5. Ordenación por intercambio

El tercer método solo mira elementos vecinos. Recorre la lista comparando cada elemento con el siguiente, y si están al revés los intercambia. Al final de una pasada, el mayor de todos ha ido arrastrándose hasta el último lugar. Se repite con el resto hasta que una pasada no tenga nada que intercambiar. Se le llama también **burbuja**, porque los elementos grandes suben hacia el final como burbujas.

```
def burbuja(lista):
    """Ordena la lista de menor a mayor, modificándola."""
    n = len(lista)
    for k in range(n - 1):
        hubo_cambio = False
        for j in range(n - 1 - k):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

```
                hubo_cambio = True
        if not hubo_cambio:
            break

notas = [4, 2, 5, 3, 5, 2, 3]
burbuja(notas)
print(notas)
```

```
[2, 2, 3, 3, 4, 5, 5]
```

El ciclo de dentro llega cada vez un lugar menos lejos, `n - 1 - k`, porque después de `k` pasadas los `k` mayores ya están al final. La bandera `hubo_cambio` permite parar antes: si una pasada entera no intercambió nada, la lista ya está ordenada. Con una lista ordenada, la burbuja hace una sola pasada de $n - 1$ comparaciones. En el peor caso vuelven a ser $\frac{n(n-1)}{2}$.

Antes de seguir, una costumbre que vale para cualquier algoritmo que escribas: probarlo contra uno que sepas que está bien. Aquí ese uno es `sorted`. Probamos los tres métodos con cientos de listas al azar de tamaños pequeños, que es donde están los casos raros, incluidas la lista vacía y la de un elemento:

```
random.seed(1)
for prueba in range(500):
    n = random.randint(0, 12)
    original = [random.randint(1, 5) for _ in range(n)]
    for metodo in [seleccion, insercion, burbuja]:
        copia = list(original)
        metodo(copia)
        if copia != sorted(original):
            print("FALLA", metodo.__name__, original, copia)

print("pruebas terminadas")
```

```
pruebas terminadas
```

Los números van del 1 al 5 a propósito, para que haya muchos repetidos, que es otro caso que se rompe con facilidad.

Fíjate en `for metodo in [seleccion, insercion, burbuja]`. Es una lista de **funciones**, no de resultados: no hay paréntesis después de los nombres, así que no se llama a ninguna. En el capítulo 3 dijimos que una función es un dato más, que se puede guardar en una variable o pasar como argumento, y aquí lo usamos por primera vez: en cada vuelta `metodo` es una de las tres funciones, y `metodo(copia)` la llama. `metodo.__name__` es el nombre con que se definió la función, para que el mensaje de error diga cuál falló.

6. Contar en lugar de cronometrar

Tenemos tres métodos que ordenan. ¿Cuál es mejor? La respuesta tentadora es cronometrarlos, y lo haremos en la sección siguiente, pero primero contamos operaciones. El reloj depende de la máquina, de lo que esté haciendo la máquina en ese momento y de la versión de Python. La cantidad de comparaciones depende solo del algoritmo y de la lista.

Ya lo hicimos una vez. En el capítulo 4, al triplicar el tope, la criba de Eratóstenes triplicaba sus tachones y el método de dividir uno por uno multiplicaba sus divisiones por casi cinco. Aquí hacemos lo mismo duplicando el tamaño de la lista:

```
def seleccion_contando(lista):
    comparaciones = 0
    n = len(lista)
    for i in range(n - 1):
        menor = i
        for j in range(i + 1, n):
            comparaciones += 1
            if lista[j] < lista[menor]:
                menor = j
        lista[i], lista[menor] = lista[menor], lista[i]
    return comparaciones

def burbuja_contando(lista):
    comparaciones = 0
    n = len(lista)
    for k in range(n - 1):
        hubo_cambio = False
        for j in range(n - 1 - k):
            comparaciones += 1
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                hubo_cambio = True
        if not hubo_cambio:
            break
    return comparaciones

random.seed(8)
print(f"{'n':>5} {'lineal':>7} {'binaria':>8} {'selección':>10} "
      f"{'inserción':>10} {'burbuja':>10}")
for n in [100, 200, 400, 800, 1600]:
    lista = [random.randint(1, 10_000) for _ in range(n)]
    lineal = buscar_contando(lista, -5)[1]
    binaria = buscar_binaria_contando(sorted(lista), -5)[1]
    sel = seleccion_contando(list(lista))
    ins = insercion_contando(list(lista))
    bur = burbuja_contando(list(lista))
    print(f"{n:>5} {lineal:>7} {binaria:>8} {sel:>10} {ins:>10} {bur:>10}")
```

```
    n  lineal  binaria  selección  inserción    burbuja
  100     100        6       4950       2573       4947
  200     200        7      19900      10180      19809
  400     400        8      79800      39218      79547
  800     800        9     319600     160691     318610
 1600    1600       10    1279200     639673    1278672
```

Lee la tabla por columnas, de arriba abajo. Cada fila tiene el doble de elementos que la anterior.

- La **búsqueda lineal** duplica su trabajo cuando la lista se duplica. Su trabajo es proporcional a n.
- La **búsqueda binaria** suma una vuelta cuando la lista se duplica. Su trabajo es proporcional a $\log_2 n$.
- Los **tres métodos de ordenación** multiplican su trabajo por cuatro cuando la lista se duplica. Su trabajo es proporcional a n^2. La selección hace exactamente $\frac{n(n-1)}{2}$; la inserción, alrededor de la mitad; la burbuja, prácticamente lo mismo que la selección. Pero los tres se multiplican por cuatro.

Ese "proporcional a" es lo que se llama el **orden de crecimiento** de un algoritmo, y se escribe con una O mayúscula: la búsqueda lineal es $O(n)$, la binaria es $O(\log n)$ y los tres métodos de ordenación son $O(n^2)$. La definición precisa es materia de Estructuras de Datos. Para nosotros, el orden de crecimiento es la respuesta a una pregunta: si la entrada se duplica, ¿por cuánto se multiplica el trabajo? Por 1 y poco, por 2, o por 4.

Las constantes, como el $\frac{1}{2}$ de $\frac{n(n-1)}{2}$, deciden cuál de dos métodos del mismo orden es más rápido. El orden de crecimiento decide algo más importante: hasta qué tamaño de entrada el programa sirve.

7. El reloj

Ahora sí, el reloj. Medimos el tiempo de los tres métodos sobre las mismas listas al azar, duplicando el tamaño:

```
def cronometrar(metodo, lista):
    """Segundos que tarda metodo en ordenar una copia de la lista."""
    copia = list(lista)
    inicio = time.perf_counter()
    metodo(copia)
    return time.perf_counter() - inicio

random.seed(8)
tamanos = [500, 1000, 2000, 4000]
tiempos = {"selección": [], "inserción": [], "burbuja": []}

for n in tamanos:
    lista = [random.randint(1, 1_000_000) for _ in range(n)]
    tiempos["selección"].append(cronometrar(seleccion, lista))
    tiempos["inserción"].append(cronometrar(insercion, lista))
    tiempos["burbuja"].append(cronometrar(burbuja, lista))

print(f"{'n':>5} {'selección':>10} {'inserción':>10} {'burbuja':>10}")
for k, n in enumerate(tamanos):
    print(f"{n:>5} {tiempos['selección'][k]:>9.3f}s {tiempos['inserción'][k]:>9.3f}s "
          f"{tiempos['burbuja'][k]:>9.3f}s")
```

```
     n  selección  inserción    burbuja
   500     0.009s     0.004s     0.009s
  1000     0.018s     0.022s     0.039s
  2000     0.072s     0.072s     0.213s
  4000     0.564s     0.772s     1.176s
```

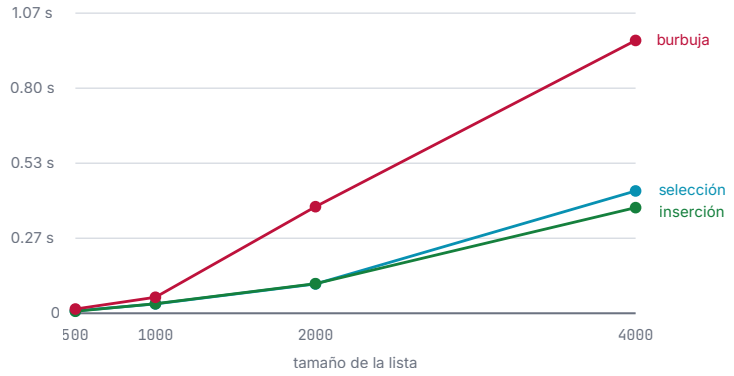


Figura 23. *Los segundos de la tabla anterior. Las tres curvas se doblan hacia arriba: cada vez que la lista se duplica, el tiempo se multiplica por cuatro.*

La burbuja es la más lenta, aunque haga prácticamente las mismas comparaciones que la selección. Hace muchos más intercambios, y un intercambio cuesta más que una comparación. Las constantes, otra vez. Pero las tres curvas tienen la misma forma, la de figura 23, y esa forma es la del n^2. Lo vemos dividiendo cada tiempo por el anterior:

```python
for metodo, ts in tiempos.items():
    razones = [ts[k + 1] / ts[k] for k in range(len(ts) - 1)]
    print(f"{metodo:>10}: " + "  ".join(f"×{r:.1f}" for r in razones))
```

```
 selección: ×3.7  ×3.8  ×4.3
 inserción: ×4.5  ×4.3  ×4.3
   burbuja: ×4.7  ×4.6  ×4.2
```

La línea que imprime une las razones con `"  ".join(...)`, y lo que va dentro del `join` es una lista por comprensión sin corchetes. Dentro de una llamada se pueden omitir: Python va produciendo los elementos uno a uno y se los pasa al `join`, sin construir la lista entera. El resultado es el mismo.

Las razones no salen exactamente 4, porque el reloj mide también otras cosas, pero rondan el 4. Con eso podemos hacer una predicción. Si 4000 elementos tardan lo que dice la tabla, un millón son 250 veces más elementos, y por tanto unas $250^2 = 62\,500$ veces más tiempo:

```python
for metodo, ts in tiempos.items():
    estimado = ts[-1] * (1_000_000 / 4000) ** 2
    print(f"{metodo:>10}: unas {estimado / 3600:.1f} horas para un millón")
```

```
  selección: unas 5.0 horas para un millón
  inserción: unas 5.3 horas para un millón
    burbuja: unas 12.6 horas para un millón
```

Horas, para un millón de números. Y un millón no es una cantidad rara: La Habana tiene más habitantes que eso. Y no se arregla con una computadora más rápida. Una diez veces más rápida pasa de horas a decenas de minutos, y con diez millones de elementos el problema vuelve, multiplicado por cien.

Para que se vea que no es un problema sin solución, esto es lo que tarda `sorted`, la función de Python, con ese mismo millón:

```
random.seed(8)
millon = [random.randint(1, 1_000_000) for _ in range(1_000_000)]
inicio = time.perf_counter()
ordenado = sorted(millon)
print(f"sorted: {time.perf_counter() - inicio:.2f} s")
```

```
sorted: 0.37 s
```

Menos de un segundo. `sorted` no es un n^2 escrito en un lenguaje más rápido; es otro algoritmo, con otro orden de crecimiento. Cómo se ordena un millón de números en segundos es el tema del capítulo 9.

8. El proyecto: las palabras por frecuencia

Volvemos al Quijote. En el capítulo 6 contamos las palabras del primer capítulo con un diccionario. Ahora queremos **todas** las palabras ordenadas de la más frecuente a la menos frecuente, escritas con un método nuestro.

El diccionario da parejas `(palabra, veces)`. Para ordenar parejas por su segundo elemento, y de mayor a menor, basta cambiar la comparación de la inserción:

```
def ordenar_por_frecuencia(parejas):
    """Ordena parejas (palabra, veces) de más a menos frecuente, modificando la lista."""
    for i in range(1, len(parejas)):
        x = parejas[i]
        j = i - 1
        while j >= 0 and parejas[j][1] < x[1]:
            parejas[j + 1] = parejas[j]
            j -= 1
        parejas[j + 1] = x

conteo = frecuencias(palabras_de("quijote-cap1.txt"))
parejas = list(conteo.items())
ordenar_por_frecuencia(parejas)

print(len(parejas), "palabras distintas")
```

```
for palabra, veces in parejas[:8]:
    print(f"  {palabra:<8} {veces:3}")
```

```
719 palabras distintas
  de       122
  y        106
  que       89
  a         44
  el        40
  en        38
  la        35
  su        35
```

Lo único que cambió es la condición del `while`: `parejas[j][1] < x[1]` compara los conteos, y el `<` en lugar de `>` pone primero los mayores. El resto de la función es la inserción de la sección 4 línea por línea. Un algoritmo de ordenación no sabe qué está ordenando; solo necesita saber cuándo un elemento va antes que otro.

Hay un detalle que vale la pena notar. Las palabras con el mismo conteo quedan en el mismo orden en que estaban en el diccionario, que como vimos en el capítulo 6 es el orden en que aparecieron por primera vez en el texto, porque la inserción nunca pasa un elemento por encima de otro igual: el `while` se detiene ante uno que no es estrictamente menor. Un método con esa propiedad se llama **estable**, y la tiene también `sorted`. La selección no la tiene; el ejercicio 4 te pide encontrar una lista que lo demuestre.

La segunda parte del proyecto es buscar. Queremos un programa al que le das una palabra y te dice cuántas veces aparece. Con el diccionario del capítulo 6 eso ya sale directo, pero vamos a hacerlo con lo de este capítulo para ver la búsqueda binaria trabajando: ordenamos las palabras distintas alfabéticamente, y buscamos en esa lista.

```
alfabetico = list(conteo)
insercion(alfabetico)
print(alfabetico[:6])
print(alfabetico[-4:])

for palabra in ["hidalgo", "rocinante", "molino"]:
    i = buscar_binaria(alfabetico, palabra)
    if i == -1:
        print(f"{palabra}: no aparece")
    else:
        print(f"{palabra}: posición {i}, aparece {conteo[palabra]}")
```

```
['a', 'acabar', 'acabándolos', 'acomodada', 'acomodársele', 'acontece']
['él', 'ésta', 'ídolo', 'ínsula']
hidalgo: posición 337, aparece 4
rocinante: posición 585, aparece 1
molino: no aparece
```

La inserción ordena las cadenas sin cambiar una línea de su código, porque `<` entre cadenas compara letra por letra, en un orden que se parece al alfabético. Ese orden es el de los códigos de los caracteres, y en esos códigos las letras con tilde y la `ñ` van después de la `z`: por eso `él` e `ínsula` salen al final. Si una palabra empieza por `á`, la búsqueda binaria la encuentra igual, porque la lista y la búsqueda usan la misma comparación.

9. Resumen

- La **búsqueda lineal** mira los elementos uno por uno. En el peor caso mira todos: su trabajo es proporcional a n.
- La **búsqueda binaria** descarta la mitad del pedazo en cada vuelta. Necesita la lista ordenada, y si no lo está devuelve respuestas falsas sin ningún error. Su trabajo es proporcional a $\log_2 n$: diecinueve o veinte vueltas para un millón.
- En la búsqueda binaria, `izq = medio + 1` y `der = medio - 1` garantizan que el pedazo se achica en cada vuelta. Con `izq = medio`, el ciclo se queda dando vueltas en cuanto el pedazo tiene uno o dos elementos y el valor buscado es mayor que el del medio.
- La **selección** busca el menor de lo que falta y lo intercambia a su lugar. Hace siempre $\frac{n(n-1)}{2}$ comparaciones.
- La **inserción** mete cada elemento nuevo en su lugar dentro de la parte ya ordenada. Es rápida con listas casi ordenadas y lenta con listas al revés. Es estable.
- La **burbuja** intercambia vecinos que están al revés, y para cuando una pasada no cambia nada.
- Los tres métodos son de orden n^2: duplicar la lista multiplica su trabajo por cuatro. Para un millón de elementos tardan horas.
- Contar operaciones dice cómo crece el trabajo sin depender de la máquina. El reloj lo confirma.
- Todo algoritmo nuevo se prueba contra uno que se sabe correcto, con muchas entradas pequeñas al azar, incluidas la vacía y la de un elemento.

Ejercicios

1. Escribe `buscar_ultimo(lista, x)` que devuelva el **último** índice donde aparece `x`, o `-1`. Hazlo de dos maneras: recorriendo desde el principio, y recorriendo desde el final. ¿Cuál hace menos comparaciones, y en qué casos?
2. La búsqueda binaria de la sección 2 devuelve **algún** índice donde está `x`, no necesariamente el primero. Encuentra una lista con repetidos donde no devuelva el primero. Después escribe `buscar_binaria_primero(lista, x)` que siempre devuelva el primero, sin dejar de partir a la mitad: cuando encuentres `x`, no termines, sigue buscando en la mitad izquierda.

3. Escribe `insertar_ordenado(lista, x)` que reciba una lista ordenada y le agregue `x` en su lugar, de manera que siga ordenada. Usa la búsqueda binaria para encontrar el lugar. ¿Cuántas comparaciones hace? ¿Y cuántos elementos tiene que mover?
4. Encuentra una lista de parejas `(nombre, nota)` donde ordenar por nota con la selección cambie el orden de dos estudiantes con la misma nota. Es decir, demuestra que la selección no es estable. ¿Puedes arreglarla?
5. Modifica la selección para que en cada pasada busque a la vez el menor y el mayor de lo que falta, y los ponga uno al principio y otro al final. ¿Cuántas pasadas hace? ¿Cuántas comparaciones? ¿Cambia el orden de crecimiento?
6. Cuenta, además de las comparaciones, cuántas veces se **escribe** en la lista en cada uno de los tres métodos, para listas al azar de 100, 200 y 400 elementos. ¿Cuál escribe menos? Con eso explica por qué la burbuja es más lenta en el reloj que la selección aunque haga casi las mismas comparaciones.
7. Construye una lista de 1000 elementos con la que la inserción haga **menos** de 2000 comparaciones, y otra con la que la burbuja haga exactamente 999. Después construye una lista de 1000 elementos casi ordenada, en la que solo el último elemento esté fuera de su lugar: es el menor de todos. ¿Cuántas pasadas hace la burbuja? ¿Y la inserción?
8. Sin usar `sorted`, `sort` ni ningún método de este capítulo, escribe `esta_ordenada(lista)` que diga si una lista está ordenada de menor a mayor. ¿Cuál es su orden de crecimiento? Úsala para que `buscar_binaria` dé un error claro cuando la lista no está ordenada. ¿Qué pasa con el costo de la búsqueda binaria si hace esa comprobación cada vez que se la llama?

PARTE II

Estrategias de solución y recursividad

8

Recursión

Hay tres varillas clavadas en una tabla. En la primera hay ocho discos de tamaños distintos, ensartados del más grande abajo al más pequeño arriba. Hay que pasarlos todos a la tercera varilla con dos reglas: se mueve un disco cada vez, el de arriba de alguna varilla, y nunca se pone un disco sobre otro más pequeño. Es el juego de las **Torres de Hanói**.

Con un disco es un movimiento. Con dos, tres: el pequeño a la del medio, el grande a la tercera, el pequeño encima. Con tres discos ya hay que pensar un rato, y con ocho casi nadie lo hace a mano sin equivocarse. Intenta escribir un programa con `while` y `for` que imprima los movimientos para `n` discos, y verás que no es nada obvio qué repetir.

Al final del capítulo lo resolvemos en cinco líneas. Para llegar ahí hace falta una sola idea nueva: **una función que se llama a sí misma**. El capítulo tiene dos mitades. En la primera vemos funciones que se llaman a sí mismas una vez, y descubrimos que esas son ciclos escritos de otra manera. En la segunda vemos funciones que se llaman a sí mismas más de una vez, que es donde la recursión resuelve lo que los ciclos no resuelven fácil.

1. Una función que se llama a sí misma

En los ejercicios del capítulo 3 escribiste el máximo común divisor. El algoritmo de Euclides dice que el `mcd` de `a` y `b` es el mismo que el de `b` y el resto de dividir `a` entre `b`, y que el `mcd` de cualquier número con 0 es el propio número. Se puede escribir tal cual:

```python
def mcd(a, b):
    """Máximo común divisor de a y b, por el algoritmo de Euclides."""
    if b == 0:
        return a
    return mcd(b, a % b)

print(mcd(84, 36))
print(mcd(17, 5))
```

```
12
1
```

Dentro de `mcd` se llama a `mcd`. Es correcto, y no da vueltas para siempre, por dos razones que conviene separar:

- Hay un **caso base**, `b == 0`, que se responde sin llamar a nadie.

- El **caso recursivo** llama a la función con un problema más pequeño: el segundo argumento pasa de `b` a `a % b`, que es menor que `b`. Como los números naturales no pueden achicarse para siempre, en algún momento el segundo argumento llega a 0.

Son las mismas dos piezas que hacen terminar a un `while`, la condición de salida y algo que se achica en cada vuelta, con otros nombres. Veamos qué pasa con `mcd(84, 36)`:

```
def mcd_traza(a, b):
    print(f"mcd({a}, {b})")
    if b == 0:
        return a
    return mcd_traza(b, a % b)

print(mcd_traza(84, 36))
```

```
mcd(84, 36)
mcd(36, 12)
mcd(12, 0)
12
```

Tres llamadas, cada una con números más pequeños, hasta que la última encuentra el caso base y devuelve 12. Ese 12 es la respuesta de la penúltima llamada, que lo devuelve sin tocarlo, y de la anterior, y así hasta la primera.

2. La búsqueda binaria, otra vez

En el capítulo 2 la computadora adivinaba un número partiendo el rango a la mitad, y en el capítulo 7 escribimos la búsqueda binaria con un `while` y dos índices. Se puede pensar de otra forma: buscar `x` entre `izq` y `der` es mirar el medio y, si no es, **buscar `x` en una de las dos mitades**. Buscar en una mitad es el mismo problema, más pequeño.

```
def buscar_binaria(lista, x, izq, der):
    """Busca x en lista[izq..der], que está ordenada. Devuelve su índice, o -1."""
    if izq > der:
        return -1                       # el pedazo está vacío
    medio = (izq + der) // 2
    if lista[medio] == x:
        return medio
    elif lista[medio] < x:
        return buscar_binaria(lista, x, medio + 1, der)
    else:
        return buscar_binaria(lista, x, izq, medio - 1)

primos = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
print(buscar_binaria(primos, 13, 0, len(primos) - 1))
print(buscar_binaria(primos, 12, 0, len(primos) - 1))
```

```
5
-1
```

Compárala con la del capítulo 7. Lo que allá era `izq = medio + 1` aquí es una llamada con `medio + 1` en el lugar de `izq`. La condición del `while`, `izq ⩽ der`, aquí es su contraria, `izq > der`, en el caso base. Es el mismo algoritmo.

Tener que escribir `0, len(primos) - 1` en cada llamada es incómodo. Lo normal es darles a los dos índices un valor por defecto, como en el capítulo 3. Para `izq` es fácil, `izq=0`. Para `der` no, porque el valor que queremos, `len(lista) - 1`, depende de otro parámetro, y un valor por defecto no puede depender de los otros parámetros. El truco es darle a `der` el valor `None` y calcular el de verdad dentro de la función:

```
def buscar_binaria(lista, x, izq=0, der=None):
    """Busca x en lista[izq..der], que está ordenada. Devuelve su índice, o -1."""
    if der is None:
        der = len(lista) - 1            # no nos dijeron dónde termina: hasta el final
    if izq > der:
        return -1
    medio = (izq + der) // 2
    if lista[medio] == x:
        return medio
    elif lista[medio] < x:
        return buscar_binaria(lista, x, medio + 1, der)
    else:
        return buscar_binaria(lista, x, izq, medio - 1)

print(buscar_binaria(primos, 29))
```

```
9
```

`None` es un valor de Python que significa "no hay valor". Es lo que devuelve una función que no tiene `return`, como viste en el capítulo 3. Para preguntar si una variable vale `None` se escribe `der is None`, no `der == None`; las dos funcionan aquí, pero `is` es la forma que vas a ver en todo el código de Python. Solo la primera llamada recibe `der = None`. Las recursivas ya le pasan un número.

3. La pila de llamadas

El factorial de `n` es $n! = 1 \cdot 2 \cdot 3 \cdots n$, y $0! = 1$. También se puede definir así: $0! = 1$, y $n! = n \cdot (n-1)!$. Esa segunda definición se escribe en Python sin cambiar nada:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(4))
print(factorial(20))
```

```
24
2432902008176640000
```

Hay una diferencia con `mcd` que no se ve a primera vista. En `mcd`, la llamada recursiva era lo último que pasaba: su resultado se devolvía tal cual. En `factorial`, cuando `factorial(4)` llama a `factorial(3)`, todavía le falta algo por hacer: multiplicar lo que le devuelvan por 4. Así que `factorial(4)` no puede terminar. Se queda **esperando**, con su `n = 4` guardado, mientras `factorial(3)` trabaja. Y `factorial(3)` se queda esperando a `factorial(2)`, y así hasta `factorial(0)`, que devuelve 1 sin esperar a nadie.

En ese momento hay cinco llamadas a `factorial` vivas a la vez, cada una con su propia `n`. Python las guarda en orden, cada una a continuación de la que la llamó, en lo que se llama la **pila de llamadas**. En la figura de abajo cada llamada nueva baja un escalón. Cuando `factorial(0)` devuelve 1, sale de la pila, y la llamada que la esperaba, un escalón más arriba, multiplica `1 * 1` y devuelve 1; la siguiente hacia arriba multiplica `2 * 1`, y así hasta la primera.

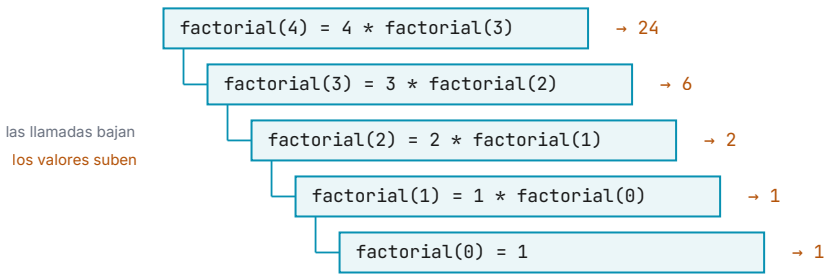


Figura 24. *La pila de `factorial(4)`. Cada llamada baja un escalón y se queda esperando. A la derecha, lo que devuelve cada una cuando la de debajo le contesta: el valor cambia en cada escalón, porque cada llamada todavía tenía una multiplicación pendiente.*

Lo importante de la figura 24 es que **cada llamada tiene sus propias variables**. La `n` de `factorial(4)` vale 4 y la de `factorial(2)` vale 2, a la vez, y no se pisan. Es lo mismo que viste en el capítulo 3 sobre las variables locales: cada vez que se llama una función, sus variables se crean de nuevo. Que la función que llama sea la misma no cambia nada.

4. Recursión de cola

Ahora escribe el factorial de otra manera. En lugar de dejar la multiplicación pendiente, la hacemos antes de llamar, y le pasamos el resultado parcial a la llamada siguiente en un parámetro nuevo, `acumulado`:

```
def factorial_cola(n, acumulado=1):
    if n == 0:
        return acumulado
    return factorial_cola(n - 1, acumulado * n)
```

```
print(factorial_cola(4))
print(factorial_cola(20))
```

```
24
2432902008176640000
```

El parámetro `acumulado` es el acumulador del capítulo 2, el que llevaba la suma o el producto parcial en un ciclo. Ahora la llamada recursiva es otra vez lo último que hace la función, como en `mcd`, y el valor que devuelve el caso base sube por la pila sin que nadie lo toque:

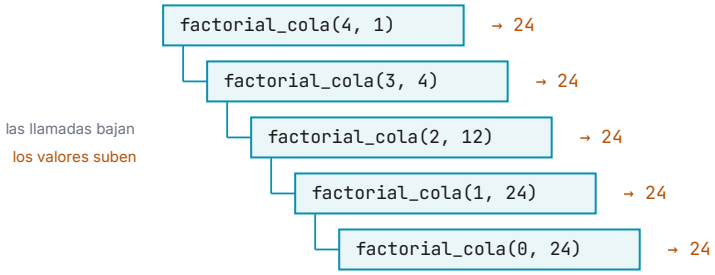


Figura 25. *La pila de `factorial_cola(4)`. El trabajo se hace al bajar, en el segundo argumento. Al subir, todas las llamadas devuelven el mismo 24: ninguna tenía nada pendiente.*

Una función donde la llamada recursiva es lo último que se hace, y su resultado se devuelve sin tocarlo, se llama **recursiva de cola**. `mcd` lo es, la búsqueda binaria recursiva lo es, y `factorial_cola` lo es. `factorial` no lo es, por la multiplicación pendiente.

Compara las dos figuras. En la figura 25, las llamadas de arriba no aportan nada cuando les llega la respuesta, así que no hacía falta guardarlas. Una función recursiva de cola se puede convertir en un ciclo de forma mecánica: los parámetros pasan a ser variables, el caso base es la condición de salida, y la llamada recursiva es una asignación de todos los parámetros a la vez.

```
def factorial_ciclo(n):
    acumulado = 1
    while n != 0:
        n, acumulado = n - 1, acumulado * n
    return acumulado

def mcd_ciclo(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(factorial_ciclo(20), mcd_ciclo(84, 36))
```

```
2432902008176640000 12
```

Pon `factorial_cola` y `factorial_ciclo` una al lado de la otra. La línea `return factorial_cola(n - 1, acumulado * n)` es `n, acumulado = n - 1, acumulado * n` con otra ropa. La asignación simultánea de la capítulo 4 importa aquí: si escribieras primero `n = n - 1` y después `acumulado = acumulado * n`, multiplicarías por el `n` nuevo, que es otro número.

Esa es la conclusión de la primera mitad: **una recursión de cola es un ciclo escrito de otra manera**. A veces se lee mejor, como en `mcd`, donde la versión recursiva es casi la definición matemática. Pero no hace nada que el ciclo no haga.

5. El límite de la pila

Y en Python, además, la versión recursiva tiene un problema que el ciclo no tiene:

```
print(len(str(factorial_ciclo(1500))), "cifras")

try:
    factorial_cola(1500)
except RecursionError as error:
    print(type(error).__name__, error)
```

```
4115 cifras
RecursionError maximum recursion depth exceeded
```

El `try` y el `except` son los del capítulo 5, con una cosa nueva: `as error` le pone nombre al error que se atrapó, para poder mirarlo. `type(error).__name__` es el nombre de su tipo, y al imprimir `error` sale el mensaje que lo acompaña. Así vemos el error sin que el programa se muera.

El ciclo calcula $1500!$, un número de más de cuatro mil cifras, sin problema. La versión recursiva falla con `RecursionError`, aunque sea de cola. La pila de llamadas tiene un tope, y Python lo pone por defecto en mil llamadas:

```
print(sys.getrecursionlimit())
```

```
1000
```

El tope está para proteger a la máquina de una recursión que nunca llega al caso base, el equivalente del ciclo infinito. Fíjate en lo que pasa si olvidas el caso base, o si el problema no se achica:

```
def factorial_roto(n):
    return n * factorial_roto(n - 1)    # sin caso base

try:
```

```
    factorial_roto(4)
except RecursionError as error:
    print(type(error).__name__, error)
```

```
RecursionError maximum recursion depth exceeded
```

Un ciclo infinito se queda colgado hasta que aprietas Ctrl+C. Una recursión infinita se para sola al llegar a mil llamadas, con un error que dice qué pasó.

Algunos lenguajes, como Scheme, reconocen las llamadas de cola y las convierten en ciclos por su cuenta, sin gastar pila. Python decidió no hacerlo, para que el error, cuando ocurre, muestre todas las llamadas que lo causaron. Así que en Python la recursión de cola tiene las desventajas de la recursión y ninguna ventaja sobre el ciclo, salvo a veces la claridad.

Con eso queda abierta la pregunta: si la recursión que hemos visto es un ciclo disfrazado, ¿para qué sirve? La segunda mitad la contesta.

6. Más de una llamada: las Torres de Hanói

Volvemos a los ocho discos. La idea que lo resuelve es pensar en el disco más grande. Para moverlo de la primera varilla a la tercera, los otros siete tienen que estar fuera de su camino, todos en la del medio. Así que el plan es:

1. Mover los siete discos de arriba de la primera varilla a la del medio.
2. Mover el disco grande de la primera a la tercera.
3. Mover los siete discos de la del medio a la tercera.

Los pasos 1 y 3 son el mismo problema con un disco menos y otras varillas. El disco grande no estorba, porque está abajo de todo y cualquier disco puede ponerse sobre él. Con cero discos no hay nada que hacer.

```
def hanoi(n, origen, destino, auxiliar):
    """Imprime los movimientos para pasar n discos de origen a destino."""
    if n == 0:
        return
    hanoi(n - 1, origen, auxiliar, destino)
    print(f"disco {n}: {origen} -> {destino}")
    hanoi(n - 1, auxiliar, destino, origen)

hanoi(3, "A", "C", "B")
```

```
disco 1: A -> C
disco 2: A -> B
disco 1: C -> B
disco 3: A -> C
disco 1: B -> A
```

```
disco 2: B -> C
disco 1: A -> C
```

Son cinco líneas, y resuelven el problema para cualquier cantidad de discos. Sigue los movimientos con tres monedas de tamaños distintos y comprueba que ninguno pone una moneda grande sobre una pequeña.

Lo que hace esta función distinta de todo lo anterior es que **se llama a sí misma dos veces**. La primera llamada no es la última cosa que se hace: después de ella viene un `print` y otra llamada. No es de cola, y no se puede convertir en un ciclo cambiando parámetros, porque después de resolver los `n - 1` discos de arriba hay que acordarse de que todavía falta mover el grande y los otros `n - 1`. Esa memoria es justo lo que guarda la pila de llamadas.

¿Cuántos movimientos hace? Contamos en lugar de imprimir:

```python
def movimientos(n):
    if n == 0:
        return 0
    return movimientos(n - 1) + 1 + movimientos(n - 1)

for n in range(1, 11):
    print(n, movimientos(n), 2 ** n - 1)
```

```
1 1 1
2 3 3
3 7 7
4 15 15
5 31 31
6 63 63
7 127 127
8 255 255
9 511 511
10 1023 1023
```

La tercera columna es $2^n - 1$, y coincide. Cada disco más duplica los movimientos y suma uno. Con ocho discos son 255. Hay una leyenda de un templo donde unos monjes mueven 64 discos de oro, un disco por segundo, y el mundo se acaba cuando terminen:

```python
segundos = 2 ** 64 - 1
print(f"{segundos / (3600 * 24 * 365.25):.3e} años")
```

```
5.845e+11 años
```

Más de quinientos mil millones de años, unas cuarenta veces la edad del universo.

El árbol de llamadas

En el factorial las llamadas formaban una fila, cada una esperando a la siguiente. En Hanói cada llamada hace dos, y las llamadas forman un **árbol**:

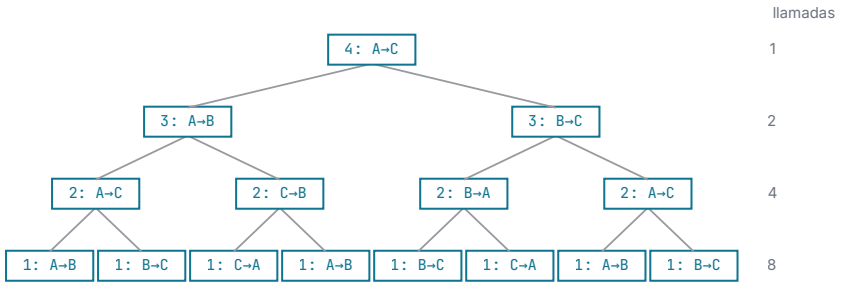


Figura 26. *Las llamadas de* `hanoi(4, "A", "C", "B")` *que mueven algo. Cada nodo dice cuántos discos mueve y entre qué varillas. Cada llamada hace un movimiento propio, el del disco grande, y le encarga el resto a sus dos hijas. A la derecha, cuántas llamadas hay en cada nivel.*

La figura 26 da otra manera de contar los movimientos. Cada llamada hace un solo movimiento propio, así que hay tantos movimientos como llamadas. En el primer nivel hay 1 llamada, en el segundo 2, en el tercero 4, en el cuarto 8: cada nivel duplica al anterior, y en total son $1 + 2 + 4 + 8 = 15 = 2^4 - 1$. Contar las llamadas nivel por nivel es la herramienta con la que el capítulo 9 va a explicar por qué la ordenación por mezcla es tan rápida.

7. Recursión sobre listas y cadenas

Antes del último problema volvemos un momento a funciones con una sola llamada, para que la recursión no quede asociada solo a los números. La recursión no es solo para números. Una lista que no está vacía es su primer elemento seguido de una lista más corta, y eso basta para recorrerla:

```
def suma(lista):
    if len(lista) == 0:
        return 0
    return lista[0] + suma(lista[1:])

print(suma([4, 2, 5, 3]))
```

```
14
```

La rebanada `lista[1:]` es la lista sin su primer elemento, y es el problema más pequeño. Esta suma tiene el mismo defecto que el factorial: no es de cola, y con más de mil elementos se rompe. Y además cada rebanada copia la lista, como viste en el capítulo 4. Para sumar una lista, el ciclo es mejor.

Lo mismo vale para cadenas. Una palabra es un palíndromo si la primera letra es igual a la última y lo que queda en medio es un palíndromo:

```python
def es_palindromo(palabra):
    if len(palabra) <= 1:
        return True
    return palabra[0] == palabra[-1] and es_palindromo(palabra[1:-1])

for palabra in ["reconocer", "anilina", "radar", "recursion"]:
    print(palabra, es_palindromo(palabra))
```

```
reconocer True
anilina True
radar True
recursion False
```

El caso base es `len(palabra) <= 1` y no `== 0`, porque cada llamada quita dos letras: si la palabra tiene una cantidad impar de letras, se llega a una sola letra, nunca a la cadena vacía.

Lo que un ciclo no hace fácil: aplanar

Ahora un problema donde la recursión sí gana. Una lista puede contener otras listas, que a su vez contienen otras, a cualquier profundidad:

```python
anidada = [1, [2, 3], [4, [5, 6, [7]], 8], [[[9]]]]
```

Queremos una lista **plana** con los números en el mismo orden: `[1, 2, ..., 9]`. Con un ciclo recorres el primer nivel; con dos ciclos anidados, dos niveles. Pero no sabes cuántos niveles hay hasta que el programa corre, y no puedes escribir una cantidad de ciclos anidados que todavía no conoces. Es el mismo callejón del capítulo 3, donde no se podía escribir una cantidad de variables que todavía no se conocía.

Con recursión es directo. Recorremos la lista, y cada elemento que sea a su vez una lista lo aplanamos con la misma función:

```python
def aplanar(lista):
    plana = []
    for x in lista:
        if isinstance(x, list):
            plana.extend(aplanar(x))     # x es una lista: aplánala y agrégala
        else:
            plana.append(x)
    return plana

print(aplanar(anidada))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`isinstance(x, list)` pregunta si `x` es una lista, y `extend` agrega al final todos los elementos de otra lista, uno por uno. El caso base no está escrito con un `if` aparte: es una lista sin listas dentro, para la que el ciclo nunca entra a la rama recursiva.

Fíjate en que aquí la recursión está **dentro de un ciclo**: una llamada puede hacer cero, una o muchas llamadas, tantas como sublistas tenga. Cada llamada recuerda en la pila por dónde iba su ciclo, y eso es justo lo que con ciclos tendrías que llevar a mano.

8. Fibonacci y el reloj

La sucesión de Fibonacci empieza con 0 y 1, y cada término es la suma de los dos anteriores: 0, 1, 1, 2, 3, 5, 8, 13… La definición se escribe sola:

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

print([fib(n) for n in range(12)])
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Es correcta, y es una de las funciones más lentas que vas a escribir este año. Medimos:

```
tiempos = []
for n in range(20, 33, 2):
    inicio = time.perf_counter()
    fib(n)
    tiempos.append((n, time.perf_counter() - inicio))

for k, (n, t) in enumerate(tiempos):
    razon = f"×{t / tiempos[k - 1][1]:.1f}" if k > 0 else ""
    print(f"fib({n}): {t:7.3f} s  {razon}")
```

```
fib(20):   0.001 s
fib(22):   0.003 s  ×2.7
fib(24):   0.008 s  ×2.8
fib(26):   0.019 s  ×2.5
fib(28):   0.049 s  ×2.6
fib(30):   0.132 s  ×2.7
fib(32):   0.369 s  ×2.8
```

La expresión `a if condicion else b` vale `a` si la condición es verdadera y `b` si no. Es un `if` que se puede usar dentro de una expresión, y aquí sirve para no imprimir razón en la primera fila, que no tiene una anterior con qué compararse.

Cada dos términos más, el tiempo se multiplica por algo más de dos y medio. Con eso se puede estimar cuánto tardarían el término 40 y el 50:

```
n, t = tiempos[-1]
razon = (tiempos[-1][1] / tiempos[0][1]) ** (1 / (len(tiempos) - 1))
for objetivo in [40, 50]:
    estimado = t * razon ** ((objetivo - n) / 2)
    print(f"fib({objetivo}): unos {estimado:.0f} segundos, {estimado / 60:.1f} minutos")
```

```
fib(40): unos 20 segundos, 0.3 minutos
fib(50): unos 2804 segundos, 46.7 minutos
```

La segunda línea calcula la razón media. Del primer tiempo al último hubo seis saltos de dos términos, y en total el tiempo se multiplicó por `tiempos[-1][1] / tiempos[0][1]`. Si cada salto multiplica por la misma razón r, entonces r^6 es ese total, y r es su raíz sexta. En Python la raíz sexta se escribe como potencia: `** (1 / 6)`. Después, para llegar de 32 a 40 faltan cuatro saltos más, y se multiplica por r^4.

La razón está en el árbol de llamadas. `fib(5)` llama a `fib(4)` y a `fib(3)`, pero `fib(4)` también llama a `fib(3)`, que se calcula dos veces desde cero:

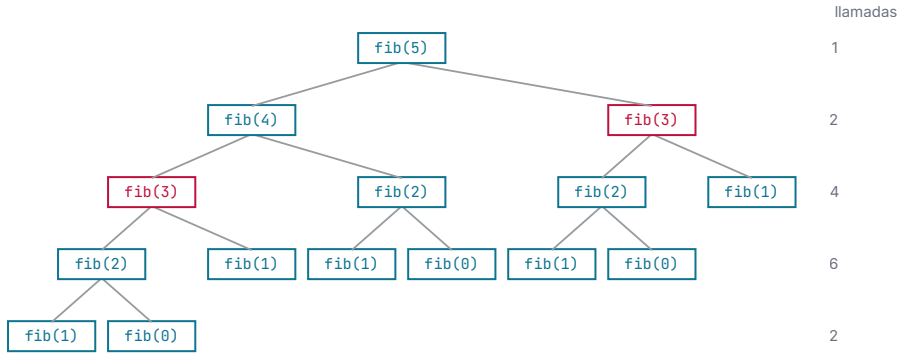


Figura 27. *Las llamadas de `fib(5)`. En rojo, las dos veces que se calcula `fib(3)`, cada una con todo su subárbol. A la derecha, cuántas llamadas hay en cada nivel.*

En la figura 27, `fib(3)` aparece dos veces, `fib(2)` tres veces y `fib(1)` cinco. Contemos cuántas llamadas hace `fib(n)` en total:

```
def llamadas(n):
    if n < 2:
        return 1
    return 1 + llamadas(n - 1) + llamadas(n - 2)

for n in [10, 20, 30]:
    print(f"fib({n}) = {fib(n)}, con {llamadas(n)} llamadas")
```

```
fib(10) = 55, con 177 llamadas
fib(20) = 6765, con 21891 llamadas
fib(30) = 832040, con 2692537 llamadas
```

Para `fib(30)` son más de dos millones de llamadas para calcular un número que con un ciclo sale en treinta sumas. El problema no es la recursión: es que el árbol repite trabajo. Cada vez que se pide `fib(3)`, se calcula entero otra vez, aunque ya se haya calculado antes. Hanói también tiene un árbol que crece como 2^n, pero ahí no hay trabajo repetido: los $2^n - 1$ movimientos son todos necesarios. En Fibonacci casi todas las llamadas sobran.

Cómo arreglarlo, recordando cada respuesta en lugar de volverla a calcular, es el tema de programación dinámica, en el segundo semestre. Por ahora queda la observación: **una definición recursiva correcta puede ser un programa inútil**, y lo que lo decide es el árbol de llamadas.

9. Resumen

- Una función recursiva se llama a sí misma. Necesita un **caso base**, que se responde sin llamar a nadie, y un **caso recursivo** que llama con un problema más pequeño.
- Cada llamada tiene sus propias variables. Las llamadas que esperan se guardan en la **pila de llamadas**.
- Una función es **recursiva de cola** si la llamada recursiva es lo último que hace y su resultado se devuelve sin tocarlo. `mcd`, la búsqueda binaria y `factorial_cola` lo son; `factorial` no.
- Una recursión de cola se convierte en un ciclo de forma mecánica: los parámetros son las variables y la llamada es una asignación simultánea. Un acumulador convierte muchas recursiones en recursiones de cola.
- Python pone un tope de mil llamadas a la pila, y no elimina las llamadas de cola. Pasar el tope da `RecursionError`, igual que olvidar el caso base.
- Una función que se llama **más de una vez** hace un árbol de llamadas. Es lo que resuelve Hanói y lo que aplana una lista anidada, cosas que un ciclo no hace fácil.
- Contar las llamadas nivel por nivel en el árbol da el costo de una función recursiva: $2^n - 1$ movimientos en Hanói.
- Si el árbol repite subproblemas, como en Fibonacci, la recursión puede ser correcta y a la vez inútil.

Ejercicios

1. Escribe `potencia(a, n)` que calcule a^n con recursión, usando $a^0 = 1$ y $a^n = a \cdot a^{n-1}$. Después escribe la versión de cola con un acumulador, y después la versión con un ciclo que sale de convertir la de cola.
2. Escribe `contar_digitos(n)` y `suma_digitos(n)` recursivas, para un entero `n >= 0`. ¿Qué es el problema más pequeño? Pista: `n // 10` y `n % 10`.

3. Escribe `invertir(cadena)` recursiva, que devuelva la cadena al revés. Después escribe `maximo(lista)` recursiva, sin usar `max`. ¿Cuál es el caso base de `maximo`, si una lista vacía no tiene máximo?
4. La búsqueda binaria recursiva tiene tantas llamadas en la pila como vueltas daba la del `while`. ¿Con qué tamaño de lista daría `RecursionError`? Justifícalo sin ejecutarla.
5. Modifica `hanoi` para que no imprima los movimientos sino que devuelva una lista de tuplas `(disco, origen, destino)`. Después escribe `comprobar(n, movimientos)` que simule las tres varillas con tres listas y diga si la secuencia es legal y termina con todos los discos en la tercera. Úsala para comprobar tu `hanoi` hasta 10 discos.
6. Escribe `profundidad(lista)` que diga cuántos niveles de anidamiento tiene una lista: `[1, 2]` tiene profundidad 1, `[1, [2]]` tiene 2 y la lista `anidada` de la sección 7 tiene 4. ¿Qué profundidad tiene la lista vacía?
7. Escribe `fib_ciclo(n)` con un ciclo y dos variables, y compara su tiempo con el de `fib` para `n = 30`. ¿Cuántas sumas hace cada una?
8. Escribe una versión de `suma(lista)` que sea de cola, con un acumulador y un índice en lugar de rebanadas. Conviértela en un ciclo. ¿Cuál de las tres versiones puede sumar una lista de un millón de elementos?

9

Divide y vencerás

El capítulo 7 terminó con una cuenta. La selección, la inserción y la burbuja multiplican su trabajo por cuatro cada vez que la lista se duplica, y con un millón de números tardarían horas. Y sin embargo `sorted` ordenaba ese mismo millón en menos de un segundo. Dijimos que `sorted` no era un n^2 escrito en un lenguaje más rápido, sino otro algoritmo.

En este capítulo escribimos dos algoritmos de esa otra clase, la **ordenación por mezcla** y la **ordenación rápida**, y los ponemos a ordenar el millón. Los dos usan la herramienta de el capítulo anterior, la función que se llama a sí misma más de una vez, y los dos siguen el mismo esquema, que tiene nombre: **divide y vencerás**. Partir el problema en pedazos, resolver cada pedazo con la misma función, y combinar las soluciones.

En los dos de este capítulo la división es evidente, porque una lista se parte en dos mitades. El capítulo siguiente trata problemas donde partir o combinar requiere pensar más.

1. Mezclar dos listas ordenadas

Empezamos por una pieza que no es recursiva. Tienes dos listas, cada una ya ordenada, y quieres una sola lista ordenada con todos los elementos de las dos.

Piensa en dos montones de exámenes, cada uno ordenado por nota de menor a mayor, boca arriba. El examen de nota más baja de todos es uno de los dos que están arriba: no puede estar debajo en un montón, porque ahí todos son mayores que el de arriba. Lo tomas, lo pones en el montón final, y repites con los dos que quedan arriba. Cuando un montón se acaba, el otro va entero al final, porque ya está ordenado.

```
def mezclar(a, b):
    """Recibe dos listas ordenadas y devuelve una lista ordenada con todos sus elemen-
tos."""
    resultado = []
    i, j = 0, 0                # el de arriba de cada montón

    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            resultado.append(a[i])
            i += 1
        else:
            resultado.append(b[j])
            j += 1

    resultado.extend(a[i:])    # lo que quede de a, que ya está ordenado
    resultado.extend(b[j:])    # o lo que quede de b; uno de los dos está vacío
    return resultado
```

```
print(mezclar([2, 5, 8, 9], [1, 3, 4, 10, 12]))
```

```
[1, 2, 3, 4, 5, 8, 9, 10, 12]
```

Cada vuelta del `while` pone un elemento en `resultado`, así que hay como mucho `len(a) + len(b)` vueltas. **Mezclar cuesta tanto como la cantidad de elementos.** No hay un ciclo dentro de otro, como en la selección: cada elemento se mira una vez, se coloca, y no se vuelve a tocar.

Fíjate en el $\leqslant$ de la comparación. Con elementos iguales, se toma primero el de `a`. Eso hace que la mezcla conserve el orden de los iguales, la propiedad que en el capítulo 7 llamamos estabilidad.

2. Ordenación por mezcla

Ahora la idea. Si sabes mezclar dos listas ordenadas, ordenar una lista es:

1. Partirla en dos mitades.
2. Ordenar cada mitad.
3. Mezclar las dos mitades ordenadas.

El paso 2 es el mismo problema con la mitad de elementos. Ahí entra la recursión. El caso base es una lista de cero o un elemento, que ya está ordenada.

```
def ordenar_mezcla(lista):
    """Devuelve una lista nueva con los elementos de lista, ordenados."""
    if len(lista) <= 1:
        return list(lista)
    medio = len(lista) // 2
    izquierda = ordenar_mezcla(lista[:medio])
    derecha = ordenar_mezcla(lista[medio:])
    return mezclar(izquierda, derecha)

print(ordenar_mezcla([38, 27, 43, 3, 9, 82, 10, 5]))
```

```
[3, 5, 9, 10, 27, 38, 43, 82]
```

A diferencia de los métodos del capítulo 7, esta función no modifica la lista que recibe: construye y devuelve una nueva, como `sorted`. Por eso el caso base devuelve `list(lista)`, una copia, y no `lista`: si devolviera la misma lista que recibió, con cero o un elemento el resultado y la original serían la misma lista con dos nombres, la sorpresa del capítulo 4.

Lo primero, como siempre, es probarla contra algo que sabemos que está bien:

```
random.seed(1)
for prueba in range(500):
    n = random.randint(0, 12)
    original = [random.randint(1, 5) for _ in range(n)]
    if ordenar_mezcla(original) != sorted(original):
        print("FALLA", original)

print("pruebas terminadas")
```

```
pruebas terminadas
```

Para ver qué pasa por dentro, miramos el árbol de llamadas del capítulo 8. Las llamadas parten la lista hacia abajo hasta llegar a listas de un elemento:

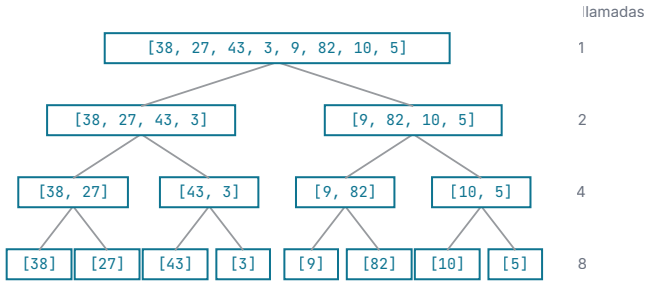


Figura 28. *Cada llamada a* `ordenar_mezcla` *con la lista que recibe. Las llamadas parten la lista a la mitad hasta llegar a listas de un elemento, que ya están ordenadas. A la derecha, cuántas llamadas hay en cada nivel.*

Y cuando las llamadas vuelven, cada una mezcla lo que le devolvieron sus dos hijas. El mismo árbol, ahora con lo que **devuelve** cada llamada:

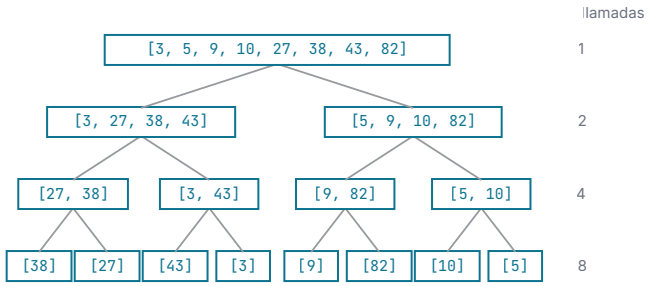


Figura 29. *El mismo árbol que en la figura anterior, con lo que devuelve cada llamada. Se lee de abajo arriba: cada nodo es la mezcla de sus dos hijos.*

3. Por qué es tan rápida

Contemos el trabajo con el árbol, nivel por nivel, como contamos los movimientos de Hanói en el capítulo 8.

En cada nivel se mezclan, en total, n elementos. En el nivel de arriba se mezclan dos listas de 4 en una de 8. En el siguiente, dos mezclas de dos listas de 2: otra vez 8 elementos en total. En el siguiente, cuatro mezclas de listas de 1: otra vez 8. Los pedazos se achican, pero hay más pedazos, y la suma es siempre la lista entera.

Hay $\log_2 n$ niveles. Cada nivel parte las listas a la mitad, y la pregunta de cuántas veces se puede partir n a la mitad hasta llegar a 1 es la de la búsqueda binaria: $\log_2 n$. Para 8 elementos las figuras tienen 4 niveles, pero el de abajo, el de las listas de un elemento, no mezcla nada: son 3 niveles de mezclas, y $\log_2 8 = 3$.

Entonces el trabajo total es n por nivel por $\log_2 n$ niveles: $n \log_2 n$. Lo comprobamos contando las comparaciones de `mezclar`:

```
comparaciones = 0

def mezclar_contando(a, b):
    global comparaciones
    resultado = []
    i, j = 0, 0
    while i < len(a) and j < len(b):
        comparaciones += 1
        if a[i] <= b[j]:
            resultado.append(a[i])
            i += 1
        else:
            resultado.append(b[j])
            j += 1
    resultado.extend(a[i:])
    resultado.extend(b[j:])
    return resultado

def ordenar_mezcla_contando(lista):
    if len(lista) <= 1:
        return lista
    medio = len(lista) // 2
    izquierda = ordenar_mezcla_contando(lista[:medio])
    derecha = ordenar_mezcla_contando(lista[medio:])
    return mezclar_contando(izquierda, derecha)

random.seed(8)
print(f"{'n':>6} {'mezcla':>8} {'n log n':>8} {'selección':>10}")
for n in [100, 200, 400, 800, 1600, 3200]:
    lista = [random.randint(1, 1_000_000) for _ in range(n)]
    comparaciones = 0
    ordenar_mezcla_contando(lista)
    print(f"{n:>6} {comparaciones:>8} {n * math.log2(n):>8.0f} {n * (n - 1) // 2:>10}")
```

```
     n   mezcla  n log n  selección
   100      545      664       4950
   200     1278     1529      19900
   400     2961     3458      79800
   800     6739     7715     319600
  1600    15047    17030    1279200
  3200    33290    37260    5118400
```

El `global comparaciones` le dice a Python que la función va a modificar la variable de afuera y no a crear una local con el mismo nombre. Es el error de alcance de la capítulo 3. Aquí lo usamos a propósito, porque el contador tiene que sobrevivir a todas las llamadas recursivas, pero no es una costumbre recomendable fuera de un experimento como este.

Lee la tabla por columnas. Cuando n se duplica, la selección multiplica sus comparaciones por cuatro, y la mezcla las multiplica por poco más de dos. Ese "poco más" es el $\log_2 n$, que sube de uno en uno cuando n se duplica. La mezcla hace un poco menos de $n \log_2 n$ comparaciones, porque cuando un montón se acaba el otro se copia sin comparar.

Para $n = 3200$, la selección hace más de cinco millones de comparaciones y la mezcla unas treinta y tres mil. Y la diferencia crece con n: es la diferencia entre multiplicar por n y multiplicar por $\log_2 n$, que para un millón es un millón contra unos veinte.

4. El esquema

La ordenación por mezcla tiene tres partes, y cualquier algoritmo de divide y vencerás las tiene:

- **Dividir** el problema en subproblemas del mismo tipo y más pequeños. Aquí, partir la lista por la mitad.
- **Vencer** cada subproblema con la misma función, recursivamente. Si es lo bastante pequeño, se resuelve directo: es el caso base.
- **Combinar** las soluciones de los subproblemas en la solución del problema. Aquí, mezclar.

La búsqueda binaria del capítulo 8 también encaja, con una particularidad: divide la lista en dos mitades pero solo necesita resolver **una**, la que puede contener el valor, y combinar no cuesta nada, porque la respuesta de la mitad es la respuesta. Por eso tiene $\log_2 n$ niveles con una sola llamada cada uno, en lugar de $\log_2 n$ niveles con n de trabajo cada uno.

Que un problema se pueda partir no basta para que divide y vencerás sea rápido. El costo total depende de cuánto cuestan las tres partes. En la mezcla, dividir es casi gratis y combinar cuesta n por nivel. El algoritmo siguiente hace lo contrario.

5. Ordenación rápida

La mezcla parte la lista sin mirarla y trabaja al combinar. La **ordenación rápida** trabaja al partir, para que combinar sea gratis.

Escoge un elemento de la lista, el **pivote**, y reparte los demás en tres listas: los menores que el pivote, los iguales y los mayores. Después ordena los menores y los mayores, recursivamente. Y ya está: todos los menores van antes que todos los iguales, que van antes que todos los mayores, así que basta pegar las tres listas.

```
def ordenar_rapida(lista):
    """Devuelve una lista nueva con los elementos de lista, ordenados."""
    if len(lista) <= 1:
        return list(lista)
    pivote = lista[0]
    menores, iguales, mayores = [], [], []
    for x in lista:
        if x < pivote:
            menores.append(x)
        elif x == pivote:
            iguales.append(x)
        else:
            mayores.append(x)
    return ordenar_rapida(menores) + iguales + ordenar_rapida(mayores)

print(ordenar_rapida([38, 27, 43, 3, 9, 82, 10, 5]))

random.seed(1)
for prueba in range(500):
    original = [random.randint(1, 5) for _ in range(random.randint(0, 12))]
    if ordenar_rapida(original) != sorted(original):
        print("FALLA", original)
print("pruebas terminadas")
```

```
[3, 5, 9, 10, 27, 38, 43, 82]
pruebas terminadas
```

El + entre listas las pega una detrás de otra, en una lista nueva. La lista `iguales` nunca está vacía, porque contiene al menos el pivote, y por eso las dos llamadas recursivas reciben listas más pequeñas que la original. Si pusiéramos el pivote junto con los menores, habría listas que se llamarían a sí mismas con la misma lista una y otra vez, hasta chocar con el tope de mil llamadas del capítulo 8 y dar `RecursionError`; el ejercicio 4 te pide encontrarlas.

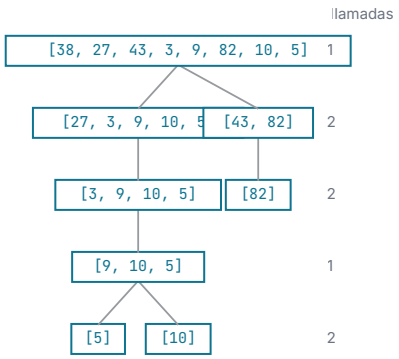


Figura 30. *Las llamadas de* `ordenar_rapida` *sobre la misma lista. El pivote es el primer elemento de cada lista, y no aparece en los hijos porque se queda en* `iguales`*. No se dibujan las llamadas con la lista vacía.*

Compara la figura 30 con la figura 28. La mezcla parte siempre exactamente a la mitad. La rápida parte donde caiga el pivote: el 38 deja cinco elementos a la izquierda y dos a la derecha. Si el pivote cae cerca del medio, el árbol tiene unos $\log_2 n$ niveles y en cada nivel se reparten, en total, unos n elementos: otra vez $n \log_2 n$. Pero eso depende de dónde caiga el pivote.

6. El pivote

¿Qué pasa si el pivote cae siempre en un extremo? Con el primer elemento como pivote, basta darle una lista **ya ordenada**. El primero es el menor, `menores` queda vacía, y `mayores` tiene todos los demás. Cada nivel quita un solo elemento:

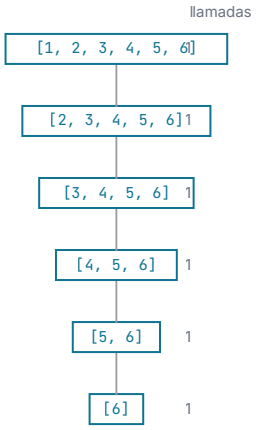


Figura 31. `ordenar_rapida([1, 2, 3, 4, 5, 6])`. *El pivote es siempre el menor, y cada llamada le pasa a la siguiente la lista con un elemento menos. El árbol es una fila, como la pila del factorial.*

El árbol de la figura 31 tiene n niveles en lugar de $\log_2 n$, y el nivel k reparte $n-k$ elementos. En total, $n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$, casi la misma suma que la de la selección y del mismo orden: otra vez cuadrática. Y peor aún, con n niveles de recursión se llega al tope de la pila de la capítulo 8:

```
ordenada = list(range(2000))
try:
    ordenar_rapida(ordenada)
except RecursionError as error:
    print(type(error).__name__, error)
```

```
RecursionError maximum recursion depth exceeded
```

La lista ya ordenada es justo el caso que más se da en la práctica: datos que alguien ya ordenó, o que llegan casi en orden. Un algoritmo de ordenación que se rompe con una lista ordenada no sirve.

El arreglo es escoger el pivote **al azar**. `random.choice(lista)` devuelve un elemento cualquiera de la lista:

```
def ordenar_rapida_azar(lista):
    if len(lista) <= 1:
        return list(lista)
    pivote = random.choice(lista)
    menores, iguales, mayores = [], [], []
```

```
    for x in lista:
        if x < pivote:
            menores.append(x)
        elif x == pivote:
            iguales.append(x)
        else:
            mayores.append(x)
    return ordenar_rapida_azar(menores) + iguales + ordenar_rapida_azar(mayores)

print(ordenar_rapida_azar(ordenada) == ordenada)
```

```
True
```

Ahora no existe ninguna lista que sea mala siempre. El pivote cae en un extremo con probabilidad pequeña, y para que el árbol quede como una fila tendría que caer en un extremo en casi todas las llamadas, lo que con listas grandes no pasa en la práctica. Lo que se puede demostrar, y se demuestra en Diseño y Análisis de Algoritmos, es que el trabajo **esperado** es proporcional a $n \log n$ con cualquier lista de entrada.

7. El reloj

Ahora sí, medimos. Primero contra los métodos del capítulo 7, con las mismas listas y los mismos tamaños:

```
def cronometrar(metodo, lista):
    copia = list(lista)
    inicio = time.perf_counter()
    metodo(copia)
    return time.perf_counter() - inicio

random.seed(8)
tamanos = [500, 1000, 2000, 4000]
tiempos = {"selección": [], "inserción": [], "mezcla": [], "rápida": []}
for n in tamanos:
    lista = [random.randint(1, 1_000_000) for _ in range(n)]
    tiempos["selección"].append(cronometrar(seleccion, lista))
    tiempos["inserción"].append(cronometrar(insercion, lista))
    tiempos["mezcla"].append(cronometrar(ordenar_mezcla, lista))
    tiempos["rápida"].append(cronometrar(ordenar_rapida_azar, lista))

print(f"{'n':>5}" + "".join(f"{m:>11}" for m in tiempos))
for k, n in enumerate(tamanos):
    print(f"{n:>5}" + "".join(f"{tiempos[m][k]:>10.3f}s" for m in tiempos))
```

```
    n  selección  inserción     mezcla     rápida
  500     0.004s     0.004s     0.001s     0.001s
 1000     0.021s     0.025s     0.004s     0.001s
 2000     0.085s     0.079s     0.004s     0.003s
 4000     0.302s     0.296s     0.007s     0.005s
```

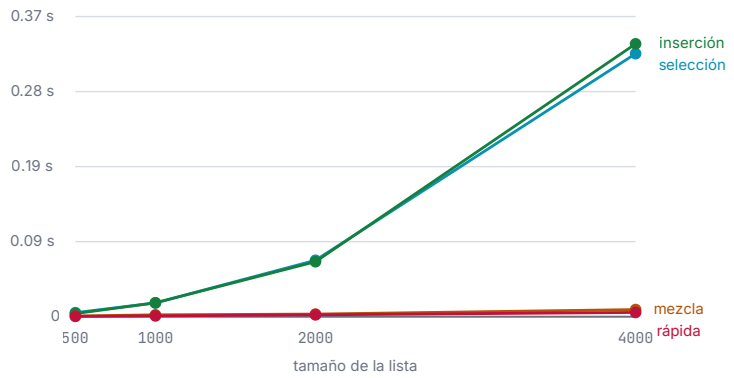


Figura 32. *Los segundos de la tabla anterior. Las dos curvas del capítulo 7 se doblan hacia arriba; la mezcla y la rápida se quedan pegadas al fondo.*

En la figura 32, la mezcla y la rápida apenas se separan del cero. Con 4000 elementos ya son más de diez veces más rápidas, y la ventaja crece con cada duplicación.

Y ahora, el millón que en el capítulo 7 habría tardado horas:

```
random.seed(8)
millon = [random.randint(1, 1_000_000) for _ in range(1_000_000)]

for nombre, metodo in [("mezcla", ordenar_mezcla), ("rápida", ordenar_rapida_azar),
                       ("sorted", sorted)]:
    inicio = time.perf_counter()
    resultado = metodo(millon)
    print(f"{nombre:>7}: {time.perf_counter() - inicio:5.2f} s  "
          f"{'bien' if resultado == sorted(millon) else 'MAL'}")
```

```
 mezcla:  4.14 s  bien
 rápida:  2.88 s  bien
 sorted:  0.37 s  bien
```

Segundos en lugar de horas, y escrito en Python, con listas y `append`, sin ningún truco. Lo que cambió no es la máquina ni el lenguaje: es el orden de crecimiento.

`sorted` sigue siendo varias veces más rápida que las nuestras, por dos razones. Está escrita en C, no en Python, y eso le da una constante mucho menor. Y su algoritmo, que se llama Timsort, es una ordenación por mezcla con mejoras: aprovecha los pedazos que ya vienen ordenados en la entrada y usa la inserción para los pedazos pequeños, donde es más rápida que la recursión. Pero es de la misma familia que la que escribimos aquí, y su orden de crecimiento es el mismo.

8. Resumen

- **Divide y vencerás**: dividir el problema en subproblemas más pequeños del mismo tipo, resolverlos recursivamente, y combinar sus soluciones.

- **Mezclar** dos listas ordenadas cuesta tanto como la cantidad de elementos: cada elemento se mira una vez.
- La **ordenación por mezcla** parte la lista a la mitad sin mirarla, ordena cada mitad y mezcla. Hace $\log_2 n$ niveles con n de trabajo cada uno: $n \log_2 n$. Es estable.
- La **ordenación rápida** reparte la lista en menores, iguales y mayores que un pivote, ordena los menores y los mayores, y pega. Trabaja al partir, y combinar es gratis.
- Con el primer elemento como pivote, una lista ya ordenada deja el árbol como una fila: trabajo cuadrático y `RecursionError` con listas de algunos miles. Con el pivote al azar, el trabajo esperado es $n \log n$ con cualquier entrada.
- Contar el trabajo de un algoritmo recursivo es contar el árbol de llamadas: cuántos niveles tiene, y cuánto trabajo hay en cada nivel.
- La búsqueda binaria es divide y vencerás con un solo subproblema.
- Pasar de n^2 a $n \log n$ es pasar de horas a segundos con un millón de elementos.

Ejercicios

1. Escribe `mezclar_tres(a, b, c)` que mezcle tres listas ordenadas en una. Hazlo de dos maneras: llamando dos veces a `mezclar`, y con un solo ciclo que mire los tres elementos de arriba. ¿Cuál hace menos comparaciones?
2. Modifica `ordenar_mezcla` para que parta la lista en **tres** pedazos en lugar de dos, usando el ejercicio anterior. ¿Cuántos niveles tiene el árbol? ¿Es más rápida? Mídelo.
3. Escribe `ordenar_mezcla_ciclo(lista)` **sin recursión**: empieza con cada elemento como una lista de uno, y mezcla de dos en dos hasta que quede una sola. Es el árbol de la figura 29 recorrido de abajo arriba.
4. Cambia `ordenar_rapida` para que el pivote vaya en `menores` en lugar de en `iguales`, es decir, usando `x ⩽ pivote`. Encuentra la lista más corta con la que la función dé `RecursionError`, y explica qué parte del razonamiento de la sección 5 deja de ser cierta.
5. Escribe `contar_rapida(lista)` que cuente las comparaciones de `ordenar_rapida` con el primer elemento como pivote, y úsala con listas ordenadas de 100, 200 y 400 elementos. ¿Se parece a la suma $\frac{n(n+1)}{2}$ de la sección 6? ¿Por qué no hace falta que la lista esté ordenada de menor a mayor para el peor caso? Encuentra otra lista mala.
6. Usa `ordenar_mezcla` para ordenar las parejas `(palabra, veces)` del Quijote de la capítulo 7 de más a menos frecuente. Tienes que cambiar la comparación en `mezclar`. Comprueba que las palabras con el mismo conteo quedan en el mismo orden que con la inserción del capítulo 7.
7. La **mediana** de una lista es el elemento que quedaría en el medio si se ordenara. Escribe `seleccionar(lista, k)` que devuelva el elemento que quedaría en la posición `k`, usando la idea de la rápida pero con **una sola** llamada recursiva: después de repartir, sabes en cuál de las tres listas está la posición `k`. ¿Cuánto trabajo hace, si el pivote parte siempre más o menos a la mitad?

8. Con un millón de elementos, la ordenación rápida con el primer elemento como pivote funciona bien si la lista está al azar. Calcula cuántos niveles tendría el árbol si la lista estuviera ordenada, y cuánto tiempo tardaría, usando lo que mediste en la capítulo 7 para un algoritmo cuadrático. En Python, con el tope de mil llamadas, ¿qué pasa en realidad: termina, o da `RecursionError`? ¿Con qué tamaño de lista empieza a fallar?

10

Divide y vencerás, cuando no es evidente

En el capítulo anterior las tres partes de divide y vencerás se veían a simple vista. Para ordenar, partir es cortar la lista por la mitad, cada mitad es el mismo problema, y combinar es mezclar, o pegar. Nadie tuvo que preguntarse qué hacer con cada pedazo.

Este capítulo trata tres problemas donde eso deja de ser obvio. En el primero partir sigue siendo fácil, pero combinar exige pensar un caso que no está en ninguna de las dos mitades, y para combinar barato hay que pedirle a cada mitad más de lo que el problema pide. En el segundo lo que no es obvio es qué partir. En el tercero hay que pedirle a cada mitad que, además de contar, ordene. Los tres tienen la misma pregunta detrás, y es la que hay que aprender a hacerse: **¿qué tengo que saber de cada mitad para poder combinar?**

1. La mejor racha

Un kiosco anota cada día cuánto ganó o perdió, en pesos. Estos son dieciséis días:

```
dias = [3, -5, 4, -1, 2, 6, -4, 2, 3, -2, 5, 1, -7, 4, -3, 2]
```

Queremos la **mejor racha**: los días seguidos, al menos uno, en los que el kiosco ganó más en total. Numeramos los días desde 0, como los índices de la lista. No vale saltarse días: si la racha va del día 2 al día 5, cuenta todo lo de en medio, sea ganancia o pérdida. En otros libros este problema se llama el **subarreglo de suma máxima**.

Si todos los números fueran positivos, la respuesta sería la lista entera. Los negativos lo complican: a veces vale la pena atravesar un día malo para llegar a días buenos, y a veces no.

Fuerza bruta

La solución obvia prueba todas las rachas: para cada día de inicio `i`, extiende la racha día por día y lleva la suma.

```
def mejor_racha_bruta(v):
    """La mayor suma de un pedazo de días seguidos de v, probando todos."""
    mejor = v[0]
    for i in range(len(v)):
        suma = 0
        for j in range(i, len(v)):
            suma += v[j]              # suma de v[i..j]
            if suma > mejor:
```

```
                mejor = suma
    return mejor

print(mejor_racha_bruta(dias))
```

```
16
```

Hay un ciclo dentro de otro, y el de dentro da `n - i` vueltas: en total $\frac{n(n+1)}{2}$, el mismo orden n^2 que la selección del capítulo 7. Funciona, y es el oráculo contra el que vamos a probar todo lo demás.

2. Partir por la mitad

Probemos divide y vencerás. Partimos los días por la mitad y buscamos la mejor racha de cada mitad, recursivamente. ¿La mejor racha de todo es la mejor de las dos?

No necesariamente. La mejor racha puede **cruzar** el punto de corte: empezar en la mitad izquierda y terminar en la derecha. Esa racha no es una racha de ninguna de las dos mitades, así que ninguna de las dos llamadas recursivas la va a encontrar. Así que hay tres candidatas:

1. La mejor racha que está entera en la mitad izquierda: la da una llamada recursiva.
2. La mejor racha que está entera en la mitad derecha: la da la otra.
3. La mejor racha que cruza el medio: hay que calcularla aparte.

La tercera parece tan difícil como el problema original, pero no lo es. Una racha que cruza el medio está formada por un pedazo que **termina** justo antes del medio y un pedazo que **empieza** justo en el medio, y los dos se pueden escoger por separado. La mejor racha que cruza es la mejor de las que terminan en el medio, pegada a la mejor de las que empiezan en el medio. Y cada una de esas es un solo ciclo: desde el medio hacia la izquierda, llevando la suma, y desde el medio hacia la derecha.

```
def mejor_cruce(v, izq, medio, der):
    """La mejor suma de un pedazo de v[izq:der] que cruza el medio."""
    suma, mejor_izq = 0, v[medio - 1]
    for i in range(medio - 1, izq - 1, -1):     # del medio hacia la izquierda
        suma += v[i]
        mejor_izq = max(mejor_izq, suma)

    suma, mejor_der = 0, v[medio]
    for j in range(medio, der):                 # del medio hacia la derecha
        suma += v[j]
        mejor_der = max(mejor_der, suma)

    return mejor_izq + mejor_der

def mejor_racha(v, izq=0, der=None):
    """La mayor suma de un pedazo de días seguidos de v[izq:der]."""
    if der is None:
        der = len(v)
```

```
    if der - izq == 1:
        return v[izq]                           # un solo día
    medio = (izq + der) // 2
    return max(mejor_racha(v, izq, medio),
               mejor_racha(v, medio, der),
               mejor_cruce(v, izq, medio, der))

print(mejor_racha(dias))
```

```
16
```

En lugar de rebanadas usamos los índices `izq` y `der`, como en la búsqueda binaria recursiva del capítulo 8, con `der` fuera del pedazo, igual que en `range` y en las rebanadas. Así ninguna llamada copia la lista. El `range(medio - 1, izq - 1, -1)` cuenta hacia atrás: de `medio - 1` hasta `izq`, porque el final de un `range` no se incluye.

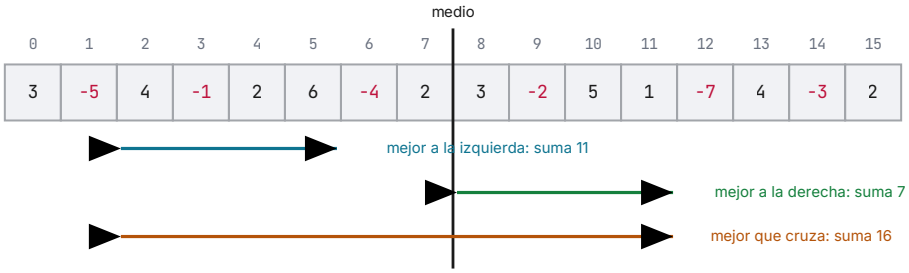


Figura 33. *Las tres candidatas en el primer nivel de la recursión. La que cruza el medio no la encuentra ninguna de las dos llamadas recursivas, y en este ejemplo es la que gana.*

En la figura 33 la mejor racha de la izquierda suma 11 y la de la derecha 7, pero la que cruza suma 16: atraviesa el día malo del índice 6 para juntar los días buenos de los dos lados. Si solo hubiéramos combinado las dos respuestas de las mitades, el programa habría devuelto 11 sin ningún error.

Probamos contra la fuerza bruta, con muchas listas pequeñas que tengan negativos:

```
random.seed(1)
for prueba in range(2000):
    v = [random.randint(-10, 10) for _ in range(random.randint(1, 12))]
    if mejor_racha(v) != mejor_racha_bruta(v):
        print("FALLA", v)
print("pruebas terminadas")
```

```
pruebas terminadas
```

Cuánto cuesta

Es el árbol de la ordenación por mezcla. Cada llamada parte su pedazo a la mitad, así que hay $\log_2 n$ niveles, y en cada nivel los cruces recorren, entre todos, cada día una vez: n de

trabajo por nivel. Total, $n \log_2 n$, igual que la mezcla. Aquí `mejor_cruce` hace el papel de `mezclar`.

3. Devolver más de lo que se pide

Se puede hacer mejor, y la manera de hacerlo es la idea más importante del capítulo.

`mejor_cruce` recorre el pedazo entero porque necesita dos cosas que las llamadas recursivas no le dieron: la mejor suma que **termina** en el borde derecho de la mitad izquierda, y la mejor que **empieza** en el borde izquierdo de la mitad derecha. Las llamadas recursivas recorrieron esos mismos días, y podrían haber calculado esas sumas por el camino, pero solo devolvían la mejor racha.

Así que hacemos que cada llamada devuelva **cuatro** números sobre su pedazo, en lugar de uno:

- `total`, la suma de todo el pedazo;
- `prefijo`, la mejor suma de un pedazo que empieza en el borde izquierdo;
- `sufijo`, la mejor suma de un pedazo que termina en el borde derecho;
- `mejor`, la mejor racha en cualquier lugar, que es la respuesta.

La pregunta es si los cuatro números del pedazo entero se pueden calcular a partir de los cuatro de cada mitad, sin mirar los días. Se puede, y cada uno se razona en una línea:

- El `total` es la suma de los dos totales.
- El mejor `prefijo` del todo o se queda en la mitad izquierda, y entonces es el prefijo de la izquierda, o la atraviesa entera y sigue en la derecha: el total de la izquierda más el prefijo de la derecha.
- El `sufijo` es lo mismo al revés.
- La `mejor` racha está en la izquierda, en la derecha, o cruza. Y la que cruza es el sufijo de la izquierda más el prefijo de la derecha, que ya los tenemos.

```
def resumen(v, izq, der):
    """Devuelve (total, prefijo, sufijo, mejor) de v[izq:der]."""
    if der - izq == 1:
        x = v[izq]
        return x, x, x, x
    medio = (izq + der) // 2
    t1, p1, s1, m1 = resumen(v, izq, medio)
    t2, p2, s2, m2 = resumen(v, medio, der)
    total = t1 + t2
    prefijo = max(p1, t1 + p2)
    sufijo = max(s2, t2 + s1)
    mejor = max(m1, m2, s1 + p2)
    return total, prefijo, sufijo, mejor

def mejor_racha_rapida(v):
    return resumen(v, 0, len(v))[3]
```

```
print(resumen(dias, 0, len(dias)))

random.seed(2)
for prueba in range(2000):
    v = [random.randint(-10, 10) for _ in range(random.randint(1, 12))]
    if mejor_racha_rapida(v) != mejor_racha_bruta(v):
        print("FALLA", v)
print("pruebas terminadas")
```

```
(10, 14, 12, 16)
pruebas terminadas
```

Ahora combinar cuesta lo mismo tenga el pedazo dos días o un millón: unas pocas sumas y `max`. El árbol tiene n hojas, una por día, y $n - 1$ nodos internos: cada nodo interno junta dos pedazos en uno, y para pasar de n pedazos a uno solo hacen falta $n - 1$ juntas. Así que hay $2n - 1$ llamadas, cada una con trabajo constante. El total es proporcional a n, mejor que $n \log n$.

Medimos las tres versiones:

```
def cronometrar(funcion, v):
    inicio = time.perf_counter()
    funcion(v)
    return time.perf_counter() - inicio

random.seed(8)
print(f"{'n':>5} {'bruta':>9} {'mitades':>9} {'resumen':>9}")
for n in [500, 1000, 2000, 4000]:
    v = [random.randint(-100, 100) for _ in range(n)]
    print(f"{n:>5} {cronometrar(mejor_racha_bruta, v):>8.3f}s "
          f"{cronometrar(mejor_racha, v):>8.3f}s "
          f"{cronometrar(mejor_racha_rapida, v):>8.3f}s")
```

```
    n     bruta   mitades   resumen
  500    0.006s    0.001s    0.000s
 1000    0.025s    0.001s    0.000s
 2000    0.105s    0.003s    0.001s
 4000    0.437s    0.006s    0.002s
```

Con estos tamaños la fuerza bruta ya está lejos, pero las dos versiones de divide y vencerás tardan tan poco que no se distinguen. Para ver la diferencia entre $n \log n$ y n hacen falta listas más grandes, sin la fuerza bruta:

```
random.seed(8)
print(f"{'n':>7} {'mitades':>9} {'resumen':>9}")
for n in [10_000, 20_000, 40_000, 80_000]:
    v = [random.randint(-100, 100) for _ in range(n)]
    print(f"{n:>7} {cronometrar(mejor_racha, v):>8.4f}s "
          f"{cronometrar(mejor_racha_rapida, v):>8.4f}s")
```

```
      n   mitades   resumen
  10000   0.0150s   0.0048s
  20000   0.0320s   0.0094s
  40000   0.0677s   0.0196s
  80000   0.1408s   0.0389s
```

Lee las columnas de arriba abajo. Las dos se duplican, más o menos, con cada duplicación de n, porque $\log_2 n$ crece tan despacio que entre 10 000 y 80 000 solo pasa de 13 a 16. La diferencia está en la constante: `mejor_racha` recorre el pedazo entero en cada llamada interna, y `resumen` solo hace unas sumas.

Lo que cambió entre `mejor_racha` y `resumen` no es la división: las dos parten igual. Es lo que devuelve cada llamada. Pedirle a cada mitad un poco más de lo que el problema pide, el total, el prefijo y el sufijo, es lo que deja combinar sin volver a mirar los datos. Es un truco que se repite: cuando combinar es caro, pregúntate qué información sobre cada mitad lo haría barato, y haz que la recursión la devuelva.

4. La potencia rápida

El segundo problema parece no tener nada que partir. Queremos calcular a^n. La manera obvia hace n multiplicaciones:

```python
def potencia_lenta(a, n):
    resultado = 1
    for _ in range(n):
        resultado *= a
    return resultado

print(potencia_lenta(3, 13))
```

```
1594323
```

No hay ninguna lista aquí. Lo que se parte es el **exponente**. Si n es par, $a^n = a^{n/2} \cdot a^{n/2}$, y basta calcular $a^{n/2}$ una vez y multiplicarlo por sí mismo. Si n es impar, $a^n = a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a$. En Python, en los dos casos la mitad es `n // 2`:

```python
def potencia(a, n):
    if n == 0:
        return 1
    mitad = potencia(a, n // 2)
    if n % 2 == 0:
        return mitad * mitad
    return mitad * mitad * a

print(potencia(3, 13))
print(all(potencia(a, n) == a ** n for a in range(-3, 6) for n in range(40)))
```

```
1594323
True
```

El `all(...)` de la última línea pregunta si todas las comparaciones son ciertas, para 9 bases y 40 exponentes. Cada llamada divide el exponente entre dos, así que hay unos $\log_2 n$ niveles, cada uno con una o dos multiplicaciones. Contamos:

```python
multiplicaciones = 0

def potencia_contando(a, n):
    global multiplicaciones
    if n == 0:
        return 1
    mitad = potencia_contando(a, n // 2)
    if n % 2 == 0:
        multiplicaciones += 1
        return mitad * mitad
    multiplicaciones += 2
    return mitad * mitad * a

for n in [10, 1000, 1_000_000, 10 ** 18]:
    multiplicaciones = 0
    potencia_contando(1, n)
    print(f"n = {n:>19}: {multiplicaciones:>3} multiplicaciones en lugar de {n}")
```

```
n =                  10:   6 multiplicaciones en lugar de 10
n =                1000:  16 multiplicaciones en lugar de 1000
n =             1000000:  27 multiplicaciones en lugar de 1000000
n = 1000000000000000000:  84 multiplicaciones en lugar de 1000000000000000000
```

Usamos la base 1 para que el resultado no crezca y la cuenta se haga en un instante. Para $n = 10^{18}$ la potencia lenta necesitaría un trillón de multiplicaciones; la rápida, menos de cien.

Un error que deshace todo

Esta versión parece la misma idea y cambia una sola línea: en lugar de guardar la mitad en una variable, la calcula donde la necesita.

```python
def potencia_mal(a, n):
    global multiplicaciones
    if n == 0:
        return 1
    if n % 2 == 0:
        multiplicaciones += 1
        return potencia_mal(a, n // 2) * potencia_mal(a, n // 2)
    multiplicaciones += 2
    return potencia_mal(a, n // 2) * potencia_mal(a, n // 2) * a

for n in [16, 256, 4096]:
    multiplicaciones = 0
```

```
    potencia_mal(1, n)
    print(f"n = {n:>4}: {multiplicaciones} multiplicaciones")
```

```
n =   16: 47 multiplicaciones
n =  256: 767 multiplicaciones
n = 4096: 12287 multiplicaciones
```

Da el resultado correcto y hace **más** multiplicaciones que la potencia lenta. Ahora cada llamada hace dos llamadas, y el árbol tiene unos $\log_2 n$ niveles, cada uno con el doble de llamadas que el anterior: $1 + 2 + 4 + \cdots$ hasta pasar de n. La tabla dice que salen unas tres multiplicaciones por cada unidad del exponente. Es el árbol de Fibonacci de la capítulo 8, que calculaba lo mismo varias veces. La potencia rápida es rápida porque resuelve **un** subproblema y lo usa dos veces, como la búsqueda binaria, que también tenía un solo subproblema. Dividir no ayuda si después resuelves los pedazos repetidos.

Para qué sirve

Con números de Python, que no tienen límite de tamaño, `a ** n` ya usa esta idea por dentro. Donde se nota es cuando solo interesa el resto de dividir entre algo, por ejemplo los últimos tres dígitos de $7^{10^{18}}$. Basta tomar el resto en cada multiplicación, para que los números nunca crezcan:

```
def potencia_resto(a, n, m):
    """a ** n % m, sin calcular a ** n."""
    if n == 0:
        return 1 % m          # y no 1: si m es 1, todo resto es 0
    mitad = potencia_resto(a, n // 2, m)
    if n % 2 == 0:
        return mitad * mitad % m
    return mitad * mitad * a % m

print(potencia_resto(7, 10 ** 18, 1000))
print(potencia_resto(7, 12345, 1000) == 7 ** 12345 % 1000)
```

```
1
True
```

El primer resultado es 1: $7^{10^{18}}$ termina en `001`. No es casualidad: 7^{20} ya termina en `001`, y como 10^{18} es múltiplo de 20, $7^{10^{18}}$ es una potencia de 7^{20}, y un número que termina en `001` multiplicado por sí mismo sigue terminando en `001`. La segunda línea comprueba la función contra `**` con un exponente que todavía se puede calcular entero.

Es el cálculo central de RSA, uno de los sistemas de criptografía de clave pública con los que se construyó `https`, que usa exponentes y restos de cientos de cifras. Los sistemas más usados hoy en `https` trabajan con curvas elípticas en lugar de restos, pero con la misma idea de duplicar y sumar. Con la potencia lenta no terminaría nunca.

5. Contar inversiones

El tercer problema: ¿cuán desordenada está una lista? Una medida natural es contar las **inversiones**, los pares de posiciones `i < j` donde `v[i] > v[j]`, es decir, los pares que están al revés. Una lista ordenada tiene 0 inversiones. Una lista al revés tiene todas las posibles, $\frac{n(n-1)}{2}$.

Sirve, por ejemplo, para comparar dos clasificaciones. Si dos personas ordenan las mismas diez canciones según su gusto, se numeran las canciones según la primera persona, se escriben en el orden de la segunda, y las inversiones cuentan los pares de canciones en que no están de acuerdo.

La fuerza bruta mira todos los pares:

```python
def inversiones_bruta(v):
    total = 0
    for i in range(len(v)):
        for j in range(i + 1, len(v)):
            if v[i] > v[j]:
                total += 1
    return total

print(inversiones_bruta([3, 1, 4, 1, 5, 9, 2, 6]))
```

```
8
```

Con divide y vencerás, partimos por la mitad. Las inversiones son de tres clases: las que tienen los dos elementos en la izquierda, las que los tienen en la derecha, y las que **cruzan**, con uno en cada mitad. Las dos primeras las cuentan las llamadas recursivas. Las que cruzan, otra vez, son el problema.

Contarlas par por par es n^2 otra vez. Pero fíjate en una cosa: para las inversiones que cruzan, el orden **dentro** de cada mitad no importa. Solo importa qué elemento de la izquierda es mayor que qué elemento de la derecha. Así que podemos pedirle a cada llamada recursiva que, además de contar, **devuelva su mitad ordenada**. Y con dos mitades ordenadas, las inversiones que cruzan se cuentan mientras se mezclan.

Mira la mezcla del capítulo 9. Cuando el elemento que se toma viene de la derecha, `b[j]`, es porque es menor que `a[i]`, y como `a` está ordenada, también es menor que todos los que quedan en `a` después de `a[i]`. Cada uno de esos forma una inversión con `b[j]`. Son `len(a) - i`, y se suman de una vez:

```python
def ordenar_contando(v):
    """Devuelve (v ordenada, cantidad de inversiones de v)."""
    if len(v) <= 1:
        return v, 0
    medio = len(v) // 2
```

```
    a, inv_a = ordenar_contando(v[:medio])
    b, inv_b = ordenar_contando(v[medio:])

    mezcla, cruzan = [], 0
    i, j = 0, 0
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            mezcla.append(a[i])
            i += 1
        else:
            mezcla.append(b[j])
            cruzan += len(a) - i      # b[j] es menor que a[i], a[i+1], ...
            j += 1
    mezcla.extend(a[i:])
    mezcla.extend(b[j:])
    return mezcla, inv_a + inv_b + cruzan

print(ordenar_contando([3, 1, 4, 1, 5, 9, 2, 6]))

random.seed(3)
for prueba in range(2000):
    v = [random.randint(1, 6) for _ in range(random.randint(0, 12))]
    if ordenar_contando(v)[1] != inversiones_bruta(v):
        print("FALLA", v)
print("pruebas terminadas")
```

```
([1, 1, 2, 3, 4, 5, 6, 9], 8)
pruebas terminadas
```

El $\leqslant$ importa otra vez. Con dos elementos iguales se toma primero el de la izquierda, y no se cuenta inversión, que es lo correcto: dos iguales no están al revés. Con <, cada par de iguales que cruza contaría como inversión, y las pruebas, que usan números del 1 al 6 para que haya muchos repetidos, lo atraparían.

La función es la ordenación por mezcla con un contador: $n \log_2 n$. Contra la fuerza bruta:

```
random.seed(8)
print(f"{'n':>5} {'bruta':>9} {'mezcla':>9}")
for n in [500, 1000, 2000, 4000]:
    v = [random.randint(1, 1_000_000) for _ in range(n)]
    print(f"{n:>5} {cronometrar(inversiones_bruta, v):>8.3f}s "
          f"{cronometrar(ordenar_contando, v):>8.3f}s")
```

```
    n     bruta    mezcla
  500    0.005s    0.001s
 1000    0.021s    0.002s
 2000    0.084s    0.003s
 4000    0.342s    0.009s
```

Fíjate en que la versión rápida resuelve un problema **más grande** que el que se le pidió: cuenta y además ordena. Es lo mismo que pasó con la mejor racha. Ordenar no era parte del problema, pero sin la lista ordenada no había manera barata de contar lo que cruza.

6. La pregunta

Los cinco problemas de los dos últimos capítulos, puestos juntos:

PROBLEMA	DIVIDIR	QUÉ DEVUELVE CADA MITAD	COMBINAR
Ordenar por mezcla	por la mitad, gratis	la mitad ordenada	mezclar, n
Ordenación rápida	alrededor de un pivote, n	la parte ordenada	pegar, gratis
Mejor racha	por la mitad	total, prefijo, sufijo y mejor	unas sumas y `max`
Potencia	el exponente a la mitad	$a^{n/2}$, una sola vez	una o dos multiplicaciones
Inversiones	por la mitad	la mitad ordenada y su cuenta	mezclar y contar, n

La columna del medio es la que hay que inventar, y la pregunta que lleva a ella es siempre la misma: **¿qué necesito saber de cada mitad para poder combinar sin volver a mirar todo?** Si la respuesta es "solo la respuesta del problema", el diseño es directo. Si no, se le pide más a la recursión, y casi siempre ese más es poco: una lista ordenada, una suma, un prefijo.

Y hay una advertencia que la potencia dejó clara: el costo lo decide el árbol de llamadas. Contar cuántos niveles tiene y cuánto trabajo hay en cada uno da la respuesta, y si un subproblema se resuelve dos veces, el árbol se llena de trabajo repetido.

7. Resumen

- En divide y vencerás, partir puede ser fácil y combinar no. La mejor racha puede **cruzar** el medio, y ninguna llamada recursiva la ve.
- La mejor racha que cruza es la mejor que termina en el medio más la mejor que empieza en el medio, y se calcula con dos ciclos: $n \log_2 n$ en total.
- Si cada llamada devuelve también el total, el mejor prefijo y el mejor sufijo, combinar cuesta lo mismo con cualquier tamaño y el total baja a n.
- La **potencia rápida** parte el exponente: a^n con unas $2 \log_2 n$ multiplicaciones. Calcular la mitad dos veces en lugar de guardarla lleva el costo de vuelta a n.
- Una **inversión** es un par de posiciones que están al revés. Se cuentan en $n \log_2 n$ si cada mitad devuelve su lista ordenada, contando mientras se mezcla.
- La pregunta que guía el diseño: qué hay que saber de cada mitad para combinar. A veces la respuesta es resolver un problema más grande que el pedido.

- Todo algoritmo de divide y vencerás se prueba contra la fuerza bruta, con muchas entradas pequeñas al azar.

Ejercicios

1. Modifica `mejor_racha_bruta` y `resumen` para que devuelvan, además de la suma, el primer y el último día de la mejor racha. ¿Qué más tiene que devolver cada llamada de `resumen`?
2. ¿Qué devuelve `mejor_racha` si todos los días son pérdidas? ¿Y si se permitiera la racha vacía, de cero días? Cambia las funciones para ese caso y prueba las dos contra una fuerza bruta escrita para la nueva regla.
3. Existe una solución de la mejor racha con **un solo ciclo** que no es divide y vencerás: recorrer los días llevando la mejor suma de una racha que termine en el día actual. Escríbela, pruébala contra la fuerza bruta y mídela contra `resumen`. Pista: la mejor racha que termina hoy es la de hoy sola, o la mejor que terminaba ayer más hoy.
4. Escribe `potencia` sin recursión, mirando los bits del exponente: $a^{13} = a^8 \cdot a^4 \cdot a^1$, porque 13 es `1101` en binario. Lleva una variable que vaya valiendo $a, a^2, a^4, a^8, \ldots$
5. Usa `potencia_resto` para encontrar los últimos cuatro dígitos de $2^{2^{100}}$. Antes de ejecutarlo, estima cuántas multiplicaciones hace.
6. Dos personas ordenan las canciones `A` a `J` así: la primera `ABCDEFGHIJ`, la segunda `CABDFEJGIH`. Usa `ordenar_contando` para medir en cuántos pares no están de acuerdo. ¿Cuál es el máximo posible con diez canciones?
7. El **par más cercano** en una recta: dada una lista de números, encuentra los dos más cercanos entre sí. Resuélvelo por fuerza bruta, y después con divide y vencerás: parte la lista **ordenada** por la mitad. ¿Qué par puede cruzar el medio? ¿Cuántos candidatos tiene? ¿Hacía falta divide y vencerás, una vez que la lista está ordenada?
8. Escribe `mayoritario(v)` que diga si algún valor aparece en más de la mitad de las posiciones, y cuál. Con divide y vencerás: si un valor es mayoritario en `v`, tiene que ser mayoritario en al menos una de las dos mitades. ¿Por qué? ¿Qué devuelve cada mitad, y cuánto cuesta combinar?