

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

Recursión

Caso base, salto de fe, recursión e iteración, y el límite de la pila

Gerson Garrido

Semana 9

Hoy

1. Resolver con un problema más pequeño: cada llamada, su marco
2. El salto de fe: razonar sin seguir las llamadas
3. Recursión en arreglos y cadenas
4. Búsqueda binaria y contar llamadas
5. Ordenar por mezcla
6. Recursión e iteración
7. La pila no es infinita, y cómo depurar

> **Lectura**
>
> Apuntes de la semana 9. Código en `code/week-09`.

SECCIÓN 01

Un problema más pequeño

01

Recursión *(recursion)*

$5! = 5 \times 4!$. Para saber el factorial de n, basta saber el de $n - 1$.

- **Caso base** *(base case)*: se responde sin otra llamada.
- **Caso recursivo**: reduce el problema y se llama a sí misma.

factorial.c

```c
long long factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    long long rest = factorial(n - 1);
    return n * rest;
}
```

En Introducción a la Computación lo vieron como un tema corto. Hoy, a fondo: sostiene toda esta parte del curso.

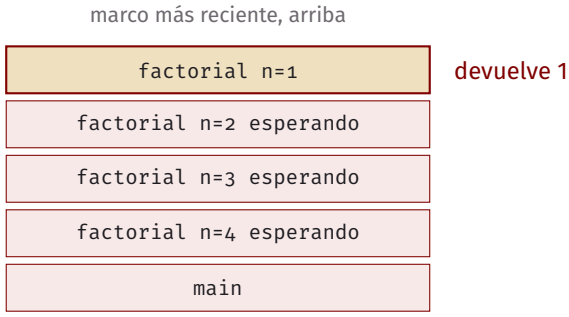
Primero baja, después calcula

terminal

```
$ build/factorial_trace
entra factorial(4)
  entra factorial(3)
    entra factorial(2)
      entra factorial(1)
      caso base: devuelve 1
    2 * 1 = 2
  3 * 2 = 6
4 * 6 = 24
resultado: 24
```

- **Al bajar**, cada llamada entra y queda esperando. No ha multiplicado nada.
- **En el caso base**, aparece la primera respuesta.
- **Al volver**, ocurren las multiplicaciones, de la más interna a la más externa.

Cada llamada tiene su marco



- Hay varias `n`, en varios marcos, con valores distintos. Comparten solo el nombre.
- Cada marco recuerda dónde se quedó, y sigue ahí al volver.

Se ven en las direcciones

frame_addresses, compilado con -g -O0

```
$ build/frame_addresses_plain
n = 3 está en 0xfffff42252cc
n = 2 está en 0xfffff422529c
n = 1 está en 0xfffff422526c
n = 0 está en 0xfffff422523c
al volver, n sigue siendo 1
al volver, n sigue siendo 2
al volver, n sigue siendo 3
niveles: 3
```

Las direcciones cambian entre corridas y máquinas.

- Son cuatro variables `n` distintas.
- Las direcciones bajan: aquí, la pila crece hacia abajo.
- Están a 0x30, es decir a 48 bytes. Es el costo de cada llamada, en esta máquina.
- Sin sanitizers: ASan cambia dónde quedan estas variables.

En gdb: un breakpoint condicional

gdb, extracto; compilado con -g -O0

```
(gdb) break factorial if n == 1
Breakpoint 1 at 0x7b4: file factorial.c, line 5.
(gdb) run
Breakpoint 1, factorial (n=1) at factorial.c:5
(gdb) backtrace
#0  factorial (n=1) at factorial.c:5
#1  0x0000aaaaaaaa07d4 in factorial (n=2) at factorial.c:8
#2  0x0000aaaaaaaa07d4 in factorial (n=3) at factorial.c:8
#3  0x0000aaaaaaaa07d4 in factorial (n=4) at factorial.c:8
#4  0x0000aaaaaaaa07fc in main () at factorial.c:13
(gdb) frame 2
#2  0x0000aaaaaaaa07d4 in factorial (n=3) at factorial.c:8
(gdb) info args
n = 3
```

- `break factorial if n == 1`: se detiene solo en la llamada que nos interesa.
- Los marcos intermedios están en la línea 8: ninguno ha terminado esa línea.

SECCIÓN 02

El salto de fe

02

Razonar sin bajar

array_examples.c

```c
static size_t count_negative(const int *a,
      size_t n) {
    if (n == 0) {
        return 0;
    }
    size_t rest = count_negative(a + 1, n - 1)
      ;
    if (a[0] < 0) {
        return rest + 1;
    }
    return rest;
}
```

Salto de fe *(leap of faith)*: suponer que la llamada recursiva acierta.

1. **¿Los casos base son correctos?** Sí: un arreglo vacío tiene 0 negativos.
2. **¿Cada llamada se acerca a un caso base?** Sí: `n` baja hasta 0.
3. **Si la llamada recursiva acierta, ¿esta acierta?** Sí: los del resto, más uno si `a[0]` es negativo.

Con esas tres respuestas, es correcta para cualquier `n`, sin seguir ninguna llamada.

No es magia: es inducción

Inducción	Función recursiva
Vale para 0	El caso base
Si vale para $n - 1$, vale para n	El salto de fe
Se llega de cualquier n a 0	El problema se achica

- La cadena: acierta con 0, por lo tanto con 1, por lo tanto con 2...
- Sin la segunda pregunta no se prueba nada: la función no termina.

Error común

Tratar de entender bajando mentalmente tres o cuatro niveles. Escriban qué **promete** la función, y úsenlo en la llamada, como si fuera `strlen`.

Termina si algo baja

Busquen un **entero no negativo que baje en cada llamada**.

Función	Lo que baja	Hasta dónde
`factorial`	`n`	1 o menos
`count_negative`	elementos que quedan	0
palíndromo, por los extremos	caracteres que quedan, de 2 en 2	0 o 1
búsqueda binaria	elementos del rango	0

Si baja de 2 en 2, el caso base debe atajar los dos finales posibles.

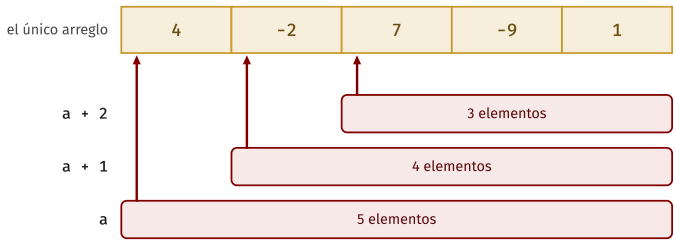
SECCIÓN 03

Arreglos y cadenas

03



El resto del arreglo es un arreglo



- `a + 1` con `n - 1` elementos es **el resto**: aritmética de punteros de la semana 4.
- No se copia nada. Todas las llamadas miran el mismo arreglo. Solo los marcos ocupan memoria.
- Por eso el pseudocódigo usa un índice, `suma(a, i, n)`, y C no lo necesita.

El caso base depende de la pregunta

array_examples.c

```c
static int array_min(const int *a, size_t n) {
    if (n == 1) {
        return a[0];
    }
    int rest = array_min(a + 1, n - 1);
    if (a[0] < rest) {
        return a[0];
    }
    return rest;
}
```

- La suma de nada es 0. Contar nada da 0.
- El mínimo de nada no existe. Por eso: requiere `n >= 1`, y el caso base es un elemento.
- Devolver 0 con el arreglo vacío “parece más general”, y es un bug. Lo veremos al depurar.

Dos casos base, y de a dos

array_examples.c

```c
static int is_symmetric(const int *a, size_t n) {
    if (n <= 1) {
        return 1;
    }
    if (a[0] != a[n - 1]) {
        return 0;
    }
    return is_symmetric(a + 1, n - 2);
}
```

- Termina de dos formas sin llamar: se acabaron los elementos, o hay un par distinto.
- `n` baja de 2 en 2: el caso base ataja 0 y 1.
- Con `n == 0` solo, y `n = 1`: `n - 2` da un número enorme, porque `size_t` no tiene negativos.

En una cadena, `s + 1` es otra cadena

array_examples.c

```c
static size_t string_length(const char *s) {
    if (s[0] == '\0') {
        return 0;
    }
    return 1 + string_length(s + 1);
}
```

terminal

```
$ build/array_examples
negativos: 2
mínimo: -9
data simétrico: 0
mirror simétrico: 1
largo de "Chosica": 7
largo de "": 0
```

La cadena misma dice dónde termina. El caso base es encontrar el `'\0'`.

Un palíndromo

pseudocódigo

```
algoritmo es_palindromo(s, i, j)
    si i ≥ j entonces
        devolver 1
    fin si
    si s[i] ≠ s[j] entonces
        devolver 0
    fin si
    devolver es_palindromo(s, i + 1, j - 1)
fin algoritmo
```

Con `abcba`:

i	j		
0	4	a a	sigue
1	3	b b	sigue
2	2		i ≥ j: devuelve 1

¿Termina? Baja `j - i`, de 2 en 2. `i` ≥ `j` lo detiene, con largo par o impar.

La función pública y la auxiliar

lab/counted.c

```c
static long search_from(const int *a, size_t n, size_t
      i, int value, long *calls) {
    *calls = *calls + 1;
    if (i == n) {
        return -1;
    }
    if (a[i] == value) {
        return (long) i;
    }
    return search_from(a, n, i + 1, value, calls);
}

long linear_search(const int *a, size_t n, int value,
     long *calls) {
    return search_from(a, n, 0, value, calls);
}
```

- Quien usa la función no quiere saber de `i`.
- La **auxiliar** `static` lleva los parámetros de la recursión.
- La **pública** prepara la primera llamada.

SECCIÓN 04

Búsqueda binaria y contar llamadas

04

Búsqueda binaria, en un rango `[lo, hi)`

pseudocódigo

```
algoritmo buscar(a, lo, hi, valor)
    si lo ≥ hi entonces
        devolver -1
    fin si
    mid ← lo + (hi - lo) div 2
    si a[mid] = valor entonces
        devolver mid
    sino si a[mid] < valor entonces
        devolver buscar(a, mid + 1, hi, valor)
    sino
        devolver buscar(a, lo, mid, valor)
    fin si
fin algoritmo
```

- `lo` entra y `hi` no. El rango vacío es `lo = hi`.
- Las cuentas no restan 1, algo que importa con `size_t`.
- Baja `hi - lo`: las dos llamadas tienen un rango más corto.

Error común

Con `mid` en vez de `mid + 1`, el rango `[lo, lo + 1)` no se achica, y se acaba la pila.

Buscar el 10

0	1	2	3	4	5	6
7	9	14	18	22	25	31

`[2, 3)`, `mid` = 2: 14 es mayor, sigue a la izquierda

Cuatro llamadas en 7 elementos, y ningún elemento mirado dos veces. El arreglo debe estar ordenado.

Contar sin globales: un puntero al contador

lab/counted.c

```c
long long power_slow(long long base, int n, long *
    calls) {
    *calls = *calls + 1;
    if (n == 0) {
        return 1;
    }
    return base * power_slow(base, n - 1, calls);
}
```

- Todos los marcos reciben **la misma dirección**: es un solo contador.
- Lo crea quien llama: `long calls = 0;`
- Aquí no calculamos fórmulas. Solo contamos: `n + 1` llamadas.

La versión rápida

pseudocódigo

```
algoritmo potencia_rapida(base, n)
    si n = 0 entonces
        devolver 1
    fin si
    mitad ← potencia_rapida(base, n div 2)
    resultado ← mitad * mitad
    si n mod 2 = 1 entonces
        resultado ← resultado * base
    fin si
    devolver resultado
fin algoritmo
```

`potencia_rapida(2, 10)`, en el orden en que termina cada llamada:

n	mitad	×	devuelve
0			1
1	1	1	2
2	2	4	4
5	4	16	32
10	32	1024	1024

5 llamadas, contra 11 de la lenta.

Las llamadas, contadas

build/lab/demo, con el laboratorio terminado

```
power(2, 10): lenta 1024 con 11 llamadas, rápida 1024 con 5 llamadas
power(2, 62): lenta 4611686018427387904 con 63 llamadas, rápida 4611686018427387904 con 7 llamadas
power(1, 1024): lenta 1 con 1025 llamadas, rápida 1 con 12 llamadas
buscar 19998 en 10000: lineal da 9999 con 10000 llamadas, binaria da 9999 con 13 llamadas
buscar 7 en 10000: lineal da -1 con 10001 llamadas, binaria da -1 con 14 llamadas
```

- `power(1, 1024)`: `n` pasa por 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 y 0. Son 12 llamadas.
- Y la búsqueda binaria en 10 000 elementos hace 13 o 14, frente a unas 10 000 de la lineal.

¿Por qué con base 1?

Las llamadas no dependen de la base, y 2^{1024} no cabe en un `long long`.

terminal

```
$ build/power_too_big
errors/power_too_big.c:8:17: runtime error: signed integer overflow: 2 * 4611686018427387904 cannot be represented in type 'long long int'
2^62 = 4611686018427387904
2^1024 = 0
```

- Ese 0 salió en esta corrida. Es comportamiento indefinido, como en la semana 7: no significa nada.
- Contar las llamadas de `power(2, 1024)` tiene sentido. Usar su resultado, no.

Un atajo que no ahorra nada

errors/power_twice.c

```c
long long result = power_twice(base, n / 2, calls) * power_twice(base, n / 2, calls);
```

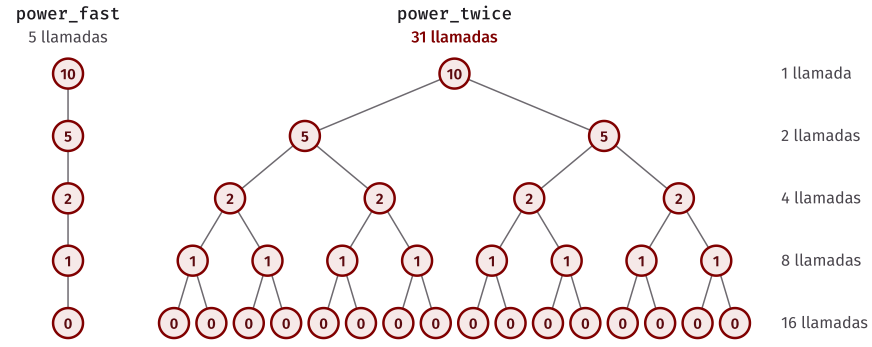
terminal

```
$ build/power_twice
power_twice(1, 10) = 1, llamadas: 31
power_twice(1, 100) = 1, llamadas: 255
power_twice(1, 1024) = 1, llamadas: 4095
```

- El mismo resultado, con 4095 llamadas contra 1025 de la lenta y 12 de la rápida.
- Cada llamada hace otras dos: un árbol, no un camino. Es el tema de la semana 10.

Guardar el resultado en una variable no es un detalle de estilo.

Un camino o un árbol



Cada círculo es una llamada, con su n.

SECCIÓN 05

Ordenar por mezcla

05

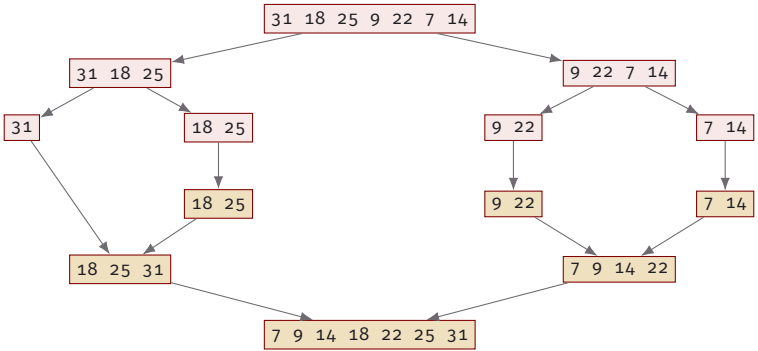
Ordenar por mezcla *(merge sort)*

pseudocódigo

```
algoritmo ordenar(a, lo, hi, scratch)
    si hi - lo < 2 entonces
        devolver
    fin si
    mid ← lo + (hi - lo) div 2
    ordenar(a, lo, mid, scratch)
    ordenar(a, mid, hi, scratch)
    mezclar(a, lo, mid, hi, scratch)
fin algoritmo
```

- Dos llamadas, cada una con la mitad.
- Aquí el **salto de fe** se ve mejor: nadie necesita saber cómo se ordena cada mitad.
- Todo el trabajo real está en `mezclar`.

El árbol de llamadas



Al bajar se parte. Al volver se mezcla. Para un millón de elementos, unos 20 niveles apilados.

Mezclar dos mitades ordenadas

Siempre se mira el primero de cada fila, y sale el menor.

`a[izq]`	`a[der]`	Sale	`scratch`
18	7	7	`7`
18	9	9	`7 9`
18	14	14	`7 9 14`
18	22	18	`7 9 14 18`
25	22	22	`7 9 14 18 22`
25		se acabó la derecha	`7 9 14 18 22 25 31`

Si se escribe sobre `a`, se pierden datos que aún no se han leído. Por eso hace falta `scratch`, y al final se copia de vuelta.

¿Quién es owner de `scratch`?

lab/merge_sort.h

```
1  int merge_sort(int *a, size_t n);
```

- `merge_sort` reserva una vez `n` enteros, llama a la recursión, y libera.
- `sort_range` y `merge` solo **usan** el bloque: no lo reservan ni lo liberan.
- Menos de 2 elementos: no pide memoria. `malloc(0)` puede devolver `NULL` sin que falte nada.
- Si `malloc` falla, devuelve 0, sin fugas.

Idea clave

Es la misma pregunta de la semana 5: cada bloque tiene un owner. La recursión no cambia eso.

SECCIÓN 06

Recursión e iteración

06

De ciclo a recursión

convert.c

```c
static int gcd_recursive(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd_recursive(b, a % b);
}
```

convert.c

```c
static int gcd_loop(int a, int b) {
    while (b != 0) {
        int next_b = a % b;
        a = b;
        b = next_b;
    }
    return a;
}
```

- Las variables del ciclo se vuelven **parámetros**.
- La condición negada es el **caso base**. La llamada es la vuelta del ciclo.
- Es directo porque la llamada es lo último que se hace: al volver, no queda nada pendiente.

Si queda algo pendiente: un acumulador

convert.c

```c
    return n + sum_recursive(n - 1);
```

convert.c

```c
static long long sum_accumulator(int n,
     long long total) {
    if (n <= 0) {
        return total;
    }
    return sum_accumulator(n - 1, total +
      n);
}
```

convert.c

```c
static long long sum_loop(int n) {
    long long total = 0;
    while (n > 0) {
        total = total + n;
        n = n - 1;
    }
    return total;
}
```

- `sum_recursive`: falta sumar **después** de la llamada.
- El **acumulador** lo lleva por adelantado. `total` es el mismo del ciclo.

Antes y después de la llamada

digits.c

```c
static void print_backward(int n) {
    printf("%d", n % 10);
    if (n >= 10) {
        print_backward(n / 10);
    }
}

static void print_forward(int n) {
    if (n >= 10) {
        print_forward(n / 10);
    }
    printf("%d", n % 10);
}
```

terminal

```
$ build/digits
6202
2026
```

- Lo de antes de la llamada ocurre **al bajar**. Lo de después, **al volver**.
- Con un ciclo habría que guardar las cifras en un arreglo. Aquí están en los marcos: la pila hace de arreglo.

¿Ciclo o recursión?

	Ciclo	Recursión
Memoria	fija	un marco por nivel
Muy profundo	no pasa nada	se acaba la pila
Volver atrás	hay que guardar los datos	lo hacen los marcos
Varias ramas	una pila a mano	natural

- **Ciclo**: solo avanza de uno en uno. Sumar un arreglo, contar en una cadena. La profundidad sería el tamaño de la entrada.
- **Recursión**: se parte en subproblemas, o es poco profunda. Búsqueda binaria, ordenar por mezcla, la semana 10.

En esta semana harán con recursión cosas que en un programa real se harían con un ciclo. Es para practicar.

SECCIÓN 07

La pila no es infinita

07

Sin caso base

terminal, extracto

```
$ gcc -Wall -Wextra -std=c17 errors/no_base_case.c -o build/no_base_case
errors/no_base_case.c:4:18: warning: infinite recursion detected [-Winfinite-recursion]
errors/no_base_case.c:5:16: note: recursive call
```

ASan: informe recortado; las direcciones cambian

```
==122==ERROR: AddressSanitizer: stack-overflow on address 0xffffcb335fe0 (pc 0xaaaab2750d28 bp 0xffffcb336010 sp 0xffffcb336010 T0)
    #0 0xaaaab2750d28 in sum_down errors/no_base_case.c:4
    #1 0xaaaab2750d64 in sum_down errors/no_base_case.c:5
    #2 0xaaaab2750d64 in sum_down errors/no_base_case.c:5
    ...
SUMMARY: AddressSanitizer: stack-overflow errors/no_base_case.c:4 in sum_down
```

- `gcc` solo avisa en casos así de directos.
- **Qué** pasó: `stack-overflow`, la pila se llenó.
- **Dónde**: la misma función y la misma línea, una y otra vez. Eso se reconoce de inmediato.

En gdb, con cuidado

gdb, extracto; sin sanitizers, con -g -O0

```
Program received signal SIGSEGV, Segmentation fault.
sum_down (n=0) at errors/no_base_case.c:4
(gdb) backtrace 3
#0  sum_down (n=0) at errors/no_base_case.c:4
#1  0x0000aaaaaaaa07c8 in sum_down (n=-174726) at errors/no_base_case.c:5
#2  0x0000aaaaaaaa07c8 in sum_down (n=-174725) at errors/no_base_case.c:5
(More stack frames follow...)
(gdb) backtrace -3
#174731 0x0000aaaaaaaa07c8 in sum_down (n=4) at errors/no_base_case.c:5
#174732 0x0000aaaaaaaa07c8 in sum_down (n=5) at errors/no_base_case.c:5
#174733 0x0000aaaaaaaa07e8 in main () at errors/no_base_case.c:9
```

- `backtrace` solo querría imprimirlos todos. Usen `backtrace 3` y `backtrace -3`.
- Los parámetros cuentan la historia: `n` empezó en 5 y va por -174726.
- $8192 \times 1024/48 = 174762$. Los marcos de 48 bytes llenan los 8192 KB de `ulimit -s`. Depende de la máquina.

Con un caso base que nunca llega

errors/skipped_base.c

```c
static int half_steps(int n) {
    if (n == 0) {
        return 0;
    }
    return 1 + half_steps(n - 2);
}
```

- Con 6: 6, 4, 2, 0. Bien.
- Con 7: 7, 5, 3, 1, −1, −3… Nunca toca el 0.
- `gcc` no avisa: hay un camino que termina.
- Falla la segunda pregunta. Se corrige con `n <= 0`.

Correcta, y demasiado profunda

deep_sum sin sanitizers; depende de la máquina

```
$ build/deep_plain recursive 170000
14450085000
$ build/deep_plain recursive 180000; echo "estado: $?"
bash: line 1:    10 Segmentation fault      build/deep_plain recursive 180000
estado: 139
$ build/deep_plain loop 10000000
50000005000000
```

terminal

```
$ build/deep_o2 recursive 10000000
50000005000000
```

- Sin ningún error de lógica: solo es profunda.
- El ciclo llega a 10 millones, sin un marco más.
- Con `-O2`, `gcc` eliminó la recursión.
- Y sin caso base, con `-O2`, se queda dando vueltas en vez de caer.
- No cuenten con eso: el lenguaje no lo garantiza.

SECCIÓN 08

Depurar

08

Primero, los casos base

errors/min_base_zero.c

```c
static int array_min(const int *a, size_t n
    ) {
    if (n == 0) {
        return 0;
    }
    int rest = array_min(a + 1, n - 1);
    if (a[0] < rest) {
        return a[0];
    }
    return rest;
}
```

terminal

```
$ build/min_base_zero
{5}: 0
{4, 2, 7}: 0
{4, -2, 7}: -2
```

- Compila sin avisos, no se cae, y la última línea es correcta.
- El caso de un elemento lo delata: el mínimo de `{5}` no puede ser 0.

Forma de trabajar: escriban la promesa, prueben vacío, uno y dos, busquen el caso más pequeño que falle, y úsenlo en `gdb`.

Lo que dice cada síntoma

Síntoma	Causa probable
`stack-overflow`, misma línea repetida	Falta caso base, o los parámetros no se acercan a él
Falla solo con pares o impares	Un caso base que se puede saltar
`heap-buffer-overflow`	Índice fuera de rango, o un `size_t` que bajó de 0
Resultado siempre del mismo tipo	Un caso base con un valor incorrecto
Funciona con pocos datos y cae con muchos	Demasiado profundo: conviértanla en ciclo
Correcto pero muy lento	Se calcula lo mismo más de una vez: cuenten llamadas

SECCIÓN 09

Laboratorio

09

Laboratorio: recursión en arreglos y cadenas

Sesión 1

1. Dibujar los marcos de `power(2, 3)` y comprobarlos en `gdb`.
2. `array_sum`, `array_max`, `count_char`, `is_palindrome` y `reverse_string`, sin ciclos y con pruebas.

Sesión 2

3. `binary_search` y `power_fast`, contando llamadas.
4. `merge_sort` sin fugas, incluso si `malloc` falla.
5. Provocar un stack overflow y leer el informe.

Antes de escribir cada función

¿Cuál es el caso base? ¿Qué baja en cada llamada? ¿Qué hace esta llamada con la respuesta de la siguiente?

Resumen

- Cada llamada tiene **su marco**. Al bajar, esperan. Al volver, cada uno sigue donde se quedó.
- **Salto de fe**: tres preguntas en vez de seguir las llamadas. Es inducción.
- **Termina** si un entero no negativo baja en cada llamada. Vigilen los saltos de 2 en 2.
- `a + 1` con `n - 1` es el resto, sin copiar. Una auxiliar `static` lleva los parámetros de más.
- `power(1, 1024)`: 1025 llamadas la lenta, 12 la rápida, 4095 si se llama dos veces. Y 2^{1024} no cabe.
- **Ordenar por mezcla**: partir, ordenar, mezclar. Una sola reserva, con un solo owner.
- Ciclo si avanza de uno en uno. Recursión si el problema se parte.
- La pila es finita: `stack-overflow` en ASan, `backtrace 3` en `gdb`.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 2, el 4 y el 7.
- Respondan las preguntas de repaso de la guía de la semana 9.
- La semana 10 es recursión en árbol: cuando cada llamada hace varias.

Antes de escribir una función recursiva

¿Qué promete? ¿Cuál es el caso más pequeño? ¿Qué baja?