


UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Programación

# Recursión en árbol

## Flood fill, cuadrículas, profundidad de la pila y memoización

**Gerson Garrido**

Semana 10

# Hoy

1. Cuando una llamada hace varias llamadas: Fibonacci
2. Flood fill: pintar una región
3. El estado “visitado”, y por qué sin él no termina
4. Contar regiones y buscar un camino
5. Cuán profunda puede ser la recursión, y una pila propia
6. Memoización

> **Lectura**
>
> Apuntes de la semana 10. Código en `code/week-10`.

SECCIÓN 01

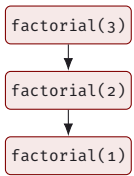
# Recursión en árbol

01

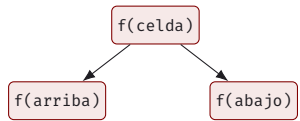


# De una cadena a un árbol

**Semana 9**: cada llamada hace una.




**Hoy**: cada llamada hace varias.




**Recursión en árbol** *(tree recursion)*

Recursión en la que una llamada hace varias llamadas. El dibujo de las llamadas ya no es una cadena: es un árbol.

# Fibonacci: dos llamadas

pseudocódigo

```
algoritmo fib(n)
    si n < 2 entonces
        devolver n
    fin si
    devolver fib(n - 1) + fib(n - 2)
fin algoritmo
```

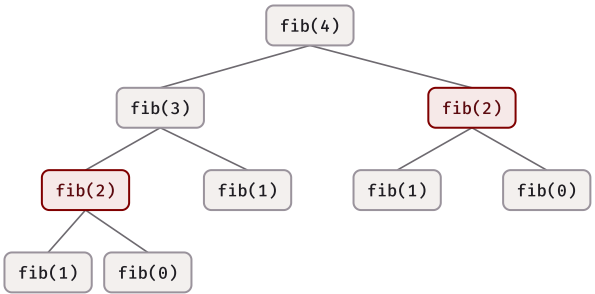
fib_trace.c

```c
int fib(int n, int depth) {
    printf("%*sfib(%d)\n", depth * 4, "", n);
    if (n < 2) {
        return n;
    }
    return fib(n - 1, depth + 1) + fib(n - 2,
      depth + 1);
}
```

- Correcta por inducción, como la semana pasada: `n` baja hasta un caso base.
- Lo nuevo es lo que cuesta ejecutarla.
- `%*s` toma el ancho de un argumento: escribe una sangría.

# El árbol de `fib(4)`

ÁRBOL DE LLAMADAS




LA PILA




9 llamadas, y como máximo 4 marcos de fib a la vez. fib(2) se calculó dos veces, y las dos dieron lo mismo.

# Dos medidas distintas

terminal

```
$ build/fib_trace
fib(4)
    fib(3)
        fib(2)
            fib(1)
            fib(0)
        fib(1)
    fib(2)
        fib(1)
        fib(0)
resultado: 3
```

- **Cuántas llamadas:** las líneas. Mide el trabajo total.
- **Cuán profundo:** la mayor sangría. Mide cuántos marcos hay en la pila a la vez.

**Idea clave**

Las llamadas dicen cuánto tarda. La profundidad dice cuánta pila usa. Más adelante falla la segunda.

# Contar llamadas de verdad

fib_count.c

```c
long long fib(int n, long long *calls) {
    (*calls)++;
    if (n < 2) {
        return n;
    }
    return fib(n - 1, calls) + fib(n - 2, calls)
      ;
}
```

terminal

```
$ build/fib_count
  n        fib(n)      llamadas
  4             3             9
  5             5            15
 10            55           177
 20          6765         21891
 30        832040       2692537
 35       9227465      29860703
 36      14930352      48315633
 37      24157817      78176337
 40     102334155     331160281
```

Sin variables globales: el contador llega por puntero.

Cada `n` más cuesta unas 1.6 veces más.

`fib(40)` con ASan: unos 2 segundos aquí. El tiempo depende de la máquina. Las llamadas, no.

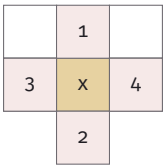
SECCIÓN 02

# Flood fill

02

# Flood fill: la herramienta del balde

- Se hace clic en un punto y se pinta toda la zona del mismo color.
- Una **región** es un grupo de celdas del mismo valor, conectadas.
- **Cuatro vecinas**: arriba, abajo, izquierda, derecha. Las diagonales no conectan.




**La recursión aparece sola**

Pintar la región desde una celda es pintar esa celda, y luego pintar la región desde cada vecina.

# En pseudocódigo

pseudocódigo

```
algoritmo flood_fill(imagen, x, y, viejo, nuevo)
    si (x, y) está fuera de la imagen entonces
        devolver
    fin si
    si imagen[x, y] ≠ viejo entonces
        devolver
    fin si
    imagen[x, y] ← nuevo
    flood_fill(imagen, x, y - 1, viejo, nuevo)
    flood_fill(imagen, x, y + 1, viejo, nuevo)
    flood_fill(imagen, x - 1, y, viejo, nuevo)
    flood_fill(imagen, x + 1, y, viejo, nuevo)
fin algoritmo
```

En el laboratorio lo escribirán para imágenes PGM. En clase, sobre una cuadrícula de caracteres, para leer el resultado en la terminal.

# En C, sobre una cuadrícula

grid_fill.c

```c
void fill(char grid[][COLS + 1], int row, int col,
      char old_mark, char new_mark) {
    if (row < 0 || row >= ROWS || col < 0 || col >=
      COLS) {
        return;
    }
    if (grid[row][col] != old_mark) {
        return;
    }
    grid[row][col] = new_mark;
    fill(grid, row - 1, col, old_mark, new_mark);
    fill(grid, row + 1, col, old_mark, new_mark);
    fill(grid, row, col - 1, old_mark, new_mark);
    fill(grid, row, col + 1, old_mark, new_mark);
}
```

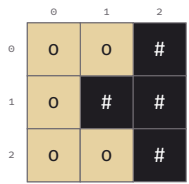
terminal

```
$ build/grid_fill
....#.......
....#..###..
#####..#.#..
..#....###..
..#.........
..#.........

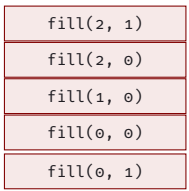
....#ooooooo
....#oo###oo
######oo#.#oo
..#oooo###oo
..#ooooooooo
..#ooooooooo
```

El . del cuadrado es del mismo valor, pero no está conectado.

# Paso a paso: desde `(0, 1)`




Pila




Orden: arriba, abajo, izquierda, derecha.

5 celdas, 5 marcos. Cada celda hace 4 llamadas, y una más para empezar: **21 llamadas**.

# Lo mismo, en la máquina

terminal

```
$ build/fill_trace
pinta (0, 1)
  pinta (0, 0)
    pinta (1, 0)
      pinta (2, 0)
        pinta (2, 1)
oo#
o##
oo#
5 celdas pintadas, 21 llamadas
```

## ¿Por qué termina?

`row` y `col` suben y bajan: ningún argumento se achica.

Lo que baja es **cuántas celdas valen `old_mark`**: cada llamada que sigue acaba de pintar una.

Es el razonamiento de la semana 9, con otra medida. Y el trabajo crece como la región: nunca se repite una rama.

SECCIÓN 03

# Visitado

03

# Pintar demasiado tarde

errors/fill_no_mark.c

```c
fill(grid, row - 1, col);
fill(grid, row + 1, col);
fill(grid, row, col - 1);
fill(grid, row, col + 1);
grid[row][col] = 'o';
```

La línea que pinta hace dos trabajos: es el resultado, y es la **marca de visitado**.

`(0, 0)` llama a `(1, 0)`, que mira arriba: `(0, 0)` sigue sin pintar. Y otra vez.

terminal

```
$ build/fill_no_mark
==201==ERROR: AddressSanitizer: stack-overflow on address 0xffffc9846ff0 (pc 0xaaaab0ea14e8 ...)
    #0 0xaaaab0ea14e8 in fill errors/fill_no_mark.c:7
    #1 0xaaaab0ea1694 in fill errors/fill_no_mark.c:14
    #3 0xaaaab0ea16cc in fill errors/fill_no_mark.c:15
SUMMARY: AddressSanitizer: stack-overflow errors/fill_no_mark.c:7 in fill
```

Compila sin advertencias. Informe recortado. Las líneas 14 y 15 se alternan.

# Pintar del mismo color

errors/fill_same_mark.c

```
1  fill(grid, 0, 0, grid[0][0], '.');
```

- La celda se “pinta”, pero sigue valiendo `old_mark`: la marca no marca nada. Mismo final, `stack-overflow`.
- Por eso `fill` exige `new_mark != old_mark`, y su `flood_fill` revisará ese caso.
- Y un error más: leer `grid[row][col]` antes de revisar los límites es comportamiento indefinido.

**Idea clave**

Marquen la celda **antes** de llamar a las vecinas, y asegúrense de que la marca se note.

SECCIÓN 04

# Regiones y caminos

04

# Contar regiones: un ciclo y un flood fill

count_islands.c

```c
int count_islands(char map[][COLS + 1]) {
    int islands = 0;
    for (int row = 0; row < ROWS; row++) {
        for (int col = 0; col < COLS; col++) {
            if (map[row][col] == '#') {
                islands++;
                sink(map, row, col);
            }
        }
    }
    return islands;
}
```

terminal

```
$ build/count_islands
##......#.....
##.....###..#.
........#...#.
...#........#.
....#...##....
.........##...
islas: 6
```

`sink` es un `fill` que cambia `#` por `~`. Las de las filas 3 y 4 se tocan por una esquina: son dos islas.

**Un costo:** destruye el mapa. En el laboratorio, `count_regions` recibe un `const Image *`.

# Dos formas de recordar por dónde se pasó

## Marcar en los datos

`fill`, `sink`, `flood_fill`.

Es corto, pero cambia los datos.

## Llevar las marcas aparte

`count_regions`: un bloque `visited` con su owner.

Cuesta memoria, y deja intacto lo que se recorre.

La pregunta de diseño: ¿quien llama espera que sus datos cambien?

# Buscar un camino: cada llamada responde

maze.c

```c
int find_path(char maze[][COLS + 1], int row, int col) {
    ...
    if (maze[row][col] == 'E') {
        return 1;
    }
    if (maze[row][col] != ' ') {
        return 0;
    }
    maze[row][col] = '+';
    if (find_path(maze, row - 1, col) || find_path(maze, row + 1, col)
        || find_path(maze, row, col - 1) || find_path(maze, row, col + 1)) {
        return 1;
    }
    maze[row][col] = 'x';
    return 0;
}
```

# El camino que encontró

terminal

```
$ build/maze
hay camino: 1
###########
#+xx#xxxxx#
#+#x#x###x#
#+#xxx#+++#
#+#####+#+#
#+++++++#E#
###########
```

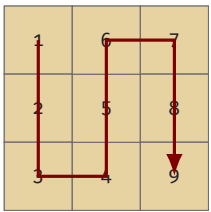
- `+` es el camino, `x` son los callejones.
- Con `||`, si una vecina tiene camino, las otras no se ejecutan.
- En `(3, 9)` probó primero arriba, recorrió todo el pasillo, y recién luego fue abajo.
- No es el camino más corto: es el primero en ese orden.
- Volver atrás anticipa el backtracking de la semana 11.

SECCIÓN 05

# Profundidad

05

# ¿Cuántos marcos a la vez? Una cuadrícula vacía




Desde `(0, 0)`, con el orden arriba, abajo, izquierda, derecha:

- baja mientras pueda,
- al llegar abajo, va a la derecha,
- sube mientras pueda, y así.

**Ninguna llamada ha terminado** cuando se pinta el 9: la región entera está en la pila.

**No son N marcos**

En una imagen de N × N, el peor caso es **N² marcos**. Con 1000 de lado, un millón.

# Medido: $N^2 + 1$

terminal

```
$ build/deep_fill 3
N = 3: 9 celdas pintadas, 10 marcos apilados como máximo
$ build/deep_fill 10
N = 10: 100 celdas pintadas, 101 marcos apilados como máximo
$ build/deep_fill 100
N = 100: 10000 celdas pintadas, 10001 marcos apilados como máximo
```

- El marco de más es la llamada de la última celda hacia afuera, que termina en el caso base.
- `deep_fill` lleva la cuenta en un `struct` que recibe por puntero.
- En el entorno del curso, `ulimit -s` dice 8192: son 8 MB de pila.

# 1000 × 1000: un programa correcto que cae

terminal

```
$ build/deep_fill 1000
==152==ERROR: AddressSanitizer: stack-overflow on address 0xfffffe174ff0 (pc 0xaaaabbe02668 ...)
    #0 0xaaaabbe02668 in fill deep_fill.c:15
    #1 0xaaaabbe02e04 in fill deep_fill.c:24
    #2 0xaaaabbe02e04 in fill deep_fill.c:24
SUMMARY: AddressSanitizer: stack-overflow deep_fill.c:15 in fill
$ build/deep_fill_plain 1000; echo "estado: $?"
bash: line 7:   159 Segmentation fault      build/deep_fill_plain 1000
estado: 139
```

Recortado: las direcciones y los números de proceso cambian, y en las rutas quitamos la carpeta.

- Sin sanitizers no hay pista. Solo `Segmentation fault`.
- No hay ningún bug: el caso base está bien, y el bloque de un millón de celdas se pidió sin problema.
- Lo que no alcanzó fue la **pila**.

# ¿Hasta dónde llega?

| Programa | Compilado | N máximo | Bytes por marco |
|---|---|---|---|
| `deep_fill` | ASan | 361 | $\approx 64$ |
| `deep_fill_plain` | `-O0` | 417 | $\approx 48$ |
| `deep_fill_o2` | `-O2` | 417 | $\approx 48$ |

Medido en este entorno: aarch64, 8 MB. **Depende de la máquina.** Los bytes se estiman: 8 MB / $N^2$.
Con más pila, el mismo programa termina:

terminal

```
$ (ulimit -s 400000; build/deep_fill_plain 1000)
N = 1000: 1000000 celdas pintadas, 1000001 marcos apilados como máximo
```

**Agrandar la pila no lo arregla:** quien corra el programa tendrá otro límite. Si la profundidad depende de la entrada, hay un límite escondido.

SECCIÓN 06

# Una pila propia

06

# Lo que queda por hacer, en memoria dinámica

pseudocódigo

```
algoritmo flood_fill_con_pila(imagen, x, y)
    pintar (x, y) y ponerla en la pila
    mientras la pila no esté vacía hacer
        sacar una celda de la pila
        para cada vecina de esa celda hacer
            si la vecina está dentro y sin pintar entonces
                pintar la vecina y ponerla en la pila
            fin si
        fin para
    fin mientras
fin algoritmo
```

- Un marco guarda lo que queda por hacer. Eso se puede guardar en otra parte.
- En el heap, el límite es mucho mayor.
- Si se acaba, `malloc` devuelve `NULL`: el programa puede avisar.

# fill_stack.c

fill_stack.c

```
push_if_free(cells, size, stack, &top, 0, 0);
while (top > 0) {
    ...
    top--;
    Cell cell = stack[top];
    painted++;
    push_if_free(cells, size, stack, &top, cell.
      row - 1, cell.col);
    push_if_free(cells, size, stack, &top, cell.
      row + 1, cell.col);
    push_if_free(cells, size, stack, &top, cell.
      row, cell.col - 1);
    push_if_free(cells, size, stack, &top, cell.
      row, cell.col + 1);
}
```

`push_if_free` pinta la celda al apilarla. Una celda pintada no vuelve a entrar: la pila nunca pasa de N × N.

**Error común**

Marcar al sacar: la misma celda entra varias veces.

# Con ASan y el mismo límite de 8 MB

terminal

```
$ build/fill_stack 10
N = 10: 100 celdas pintadas, 46 pendientes como máximo
$ build/fill_stack 300
N = 300: 90000 celdas pintadas, 44851 pendientes como máximo
$ build/fill_stack 1000
N = 1000: 1000000 celdas pintadas, 499501 pendientes como máximo
```

| | Recursión | Pila propia |
|---|---|---|
| Código | Corto, parecido a la definición | Más largo |
| Límite | La pila del sistema, unos MB | La memoria dinámica |
| Si no alcanza | El programa cae | `malloc` devuelve `NULL` |
| Cuándo | Profundidad pequeña y conocida | Profundidad que crece con la entrada |

**La pregunta:** ¿cuán profunda puede ser, y quién decide eso?

SECCIÓN 07

# Memoización

07



# Memoización: no calcular dos veces

fib_memo.c

```c
long long fib_memo(int n, long long
      memo[], long long *calls) {
    (*calls)++;
    if (n < 2) {
        return n;
    }
    if (memo[n] != NOT_KNOWN) {
        return memo[n];
    }
    memo[n] = fib_memo(n - 1, memo,
      calls) + fib_memo(n - 2, memo,
      calls);
    return memo[n];
}
```

La tabla la pide y la libera `fib_with_table`, su owner.

terminal

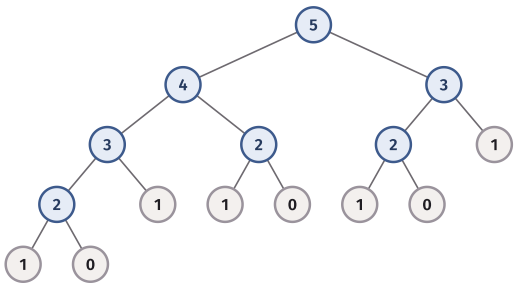
```
$ build/fib_memo
   n        fib(n)     sin tabla     con tabla
   5             5            15             9
  10            55           177            19
  20          6765         21891            39
  30        832040       2692537            59
  35       9227465      29860703            69
fib(90) = 2880067194370816120, con 179 llamadas
```

Con tabla: 2n - 1 llamadas. El árbol se volvió casi una cadena.

# `fib(5)`, sin tabla y con tabla

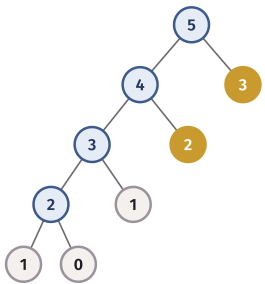
**Sin tabla**

15 llamadas




**Con tabla**

**9 llamadas**




calcula  caso base  responde desde la tabla

# ¿Qué se puede memoizar?

1. **Mismo argumento, mismo resultado.** Nada de archivos, horas ni números aleatorios.
2. **Sin efectos que importen.** Quien responde desde la tabla no escribe ni cambia nada.
3. **Argumentos que sirvan de índice.** Con un `int` pequeño, basta un arreglo.
4. Y para que valga la pena, **los subproblemas se repiten**.

Flood fill no: su trabajo es cambiar la imagen, lo que hace depende de lo ya pintado, y nunca repite una celda.

> **Idea clave**
>
> Memoizar es cambiar memoria por tiempo.

SECCIÓN 08

# Laboratorio

08

# Laboratorio, sesión 1: regiones y tablero

1. Leer el código dado: `image.h`, `regions.h`, `board.h` y las pruebas.
2. `flood_fill`, usando la imagen solo por su interfaz.
3. `count_regions`: recibe `const Image *`, con sus marcas en un bloque aparte.
4. `board_new` y `board_free`: un tipo opaco `Board`, con dos bloques.
5. `count_neighbors`: las ocho vecinas de cada celda.

> **Para probar**
>
> `build/lab/fill_demo lab/images/islands.pgm 2 2 255`, y `make test-lab`.

# Laboratorio, sesión 2: abrir celdas

pseudocódigo

```
algoritmo abrir(tablero, fila, columna)
    si (fila, columna) está fuera del tablero entonces
        devolver
    fin si
    si la celda ya está abierta o tiene bandera entonces
        devolver
    fin si
    abrir la celda
    si la celda es una mina entonces
        anotar que explotó
        devolver
    fin si
    sumar 1 a las celdas abiertas
    si la celda no tiene minas vecinas entonces
        para cada una de sus ocho vecinas hacer
            abrir(tablero, fila de la vecina, columna de la vecina)
        fin para
    fin si
fin algoritmo
```

- Es un flood fill con ocho vecinas. Las celdas con número son la frontera.
- `board_state`: victoria y derrota. Las pruebas usan tableros fijos y una semilla conocida.
- **Al final:** 1000 × 1000 sin minas.

**Nuestra solución**

Con 8 MB cayó con ASan y sin sanitizers. Con más pila, terminó bien.

# Resumen

- En una **recursión en árbol**, las llamadas dicen cuánto tarda, y la profundidad, cuánta pila usa.
- `fib(40)` hace más de 331 millones de llamadas, porque repite los mismos cálculos.
- **Flood fill** pinta una celda y sigue por sus vecinas. Termina porque cada vez queda menos por pintar.
- La marca de **visitado** va antes de llamar a las vecinas, y tiene que cambiar algo.
- Las marcas van en los datos o aparte, si los datos no se pueden cambiar.
- En una imagen de N × N se pueden apilar **N² marcos**. Un programa correcto puede caer.
- Si la profundidad crece con la entrada: una **pila propia**, marcando al apilar.
- **Memoización**: guardar resultados, si la función depende solo de sus argumentos y no tiene efectos que importen.

# Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan, y sus respuestas sobre el tablero de 1000 × 1000.
- Resuelvan los ejercicios de los apuntes, sobre todo el 1, el 3, el 8 y el 10.
- Respondan las preguntas de repaso de la guía de la semana 10.
- La semana 11 es backtracking: elegir, explorar y deshacer.

**Antes de escribir una recursión sobre una cuadrícula**

¿Dónde está la marca de visitado? ¿Cuán profunda puede llegar?