

Introducción a la Programación

Backtracking

Permutaciones, subconjuntos, N reinas y poda

Gerson Garrido

Semana 11

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Un problema de decisiones: el árbol
2. El esquema: elegir, explorar, deshacer
3. Subconjuntos y permutaciones
4. N reinas: representar el estado y deshacer bien
5. Una solución o todas
6. Poda: cuánto se gana, y cuándo miente
7. El laboratorio: resolver sudokus

> **Lectura**
>
> Apuntes de la semana 11. Código en `code/week-11`.

SECCIÓN 01

El esquema

01

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

- El 5 estaba mal. Hay que **volver**, borrar y probar el 6.

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

- El 5 estaba mal. Hay que **volver**, borrar y probar el 6.
- Eso es **backtracking** *(backtracking)*: probar una elección, explorar lo que sigue y, si no funciona, deshacerla.

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

- El 5 estaba mal. Hay que **volver**, borrar y probar el 6.
- Eso es **backtracking** *(backtracking)*: probar una elección, explorar lo que sigue y, si no funciona, deshacerla.
- ¿Por qué no probar todo? 57 celdas vacías son 9^{57} tableros: un número de 55 cifras.

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

- El 5 estaba mal. Hay que **volver**, borrar y probar el 6.
- Eso es **backtracking** *(backtracking)*: probar una elección, explorar lo que sigue y, si no funciona, deshacerla.
- ¿Por qué no probar todo? 57 celdas vacías son 9^{57} tableros: un número de 55 cifras.
- Hoy resolverán ese tablero en menos de 2 millones de nodos, y luego en menos de 2 mil.

Decidir sin saber si la decisión es buena

Ponen un 5 en una celda de un sudoku, llenan otras diez, y recién ahí una celda se queda sin opciones.

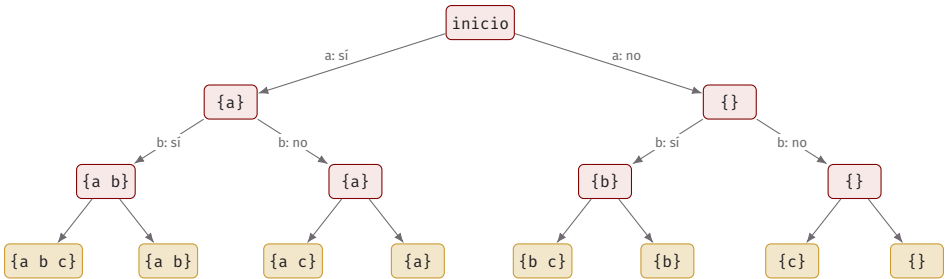
- El 5 estaba mal. Hay que **volver**, borrar y probar el 6.
- Eso es **backtracking** *(backtracking)*: probar una elección, explorar lo que sigue y, si no funciona, deshacerla.
- ¿Por qué no probar todo? 57 celdas vacías son 9^{57} tableros: un número de 55 cifras.
- Hoy resolverán ese tablero en menos de 2 millones de nodos, y luego en menos de 2 mil.

Lo nuevo frente a flood fill

En flood fill, lo marcado queda marcado. Aquí, **lo que se marca, se desmarca al volver**.

El árbol de decisiones

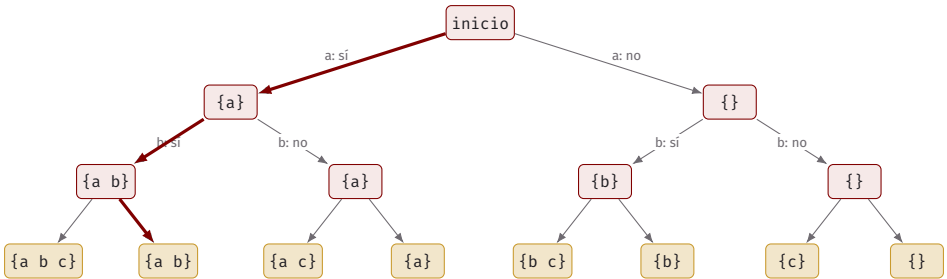
Por cada plato de la carta, *a*, *b* y *c*: ¿lo piden o no?



Nodo: un estado de la búsqueda. **Hoja**: un nodo sin ramas. **Raíz**: el de arriba.

El árbol de decisiones

Por cada plato de la carta, *a*, *b* y *c*: ¿lo piden o no?



En la pila solo está **el camino actual**: 15 nodos en total, pero nunca más de 4 marcos a la vez.

Elegir, explorar, deshacer

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        procesar la solución
        devolver
    fin si
    para opcion desde 0 hasta cantidad_de_opciones - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            buscar(estado)
            quitar opcion de estado
        fin si
    fin para
fin algoritmo
```

Elegir, explorar, deshacer

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        procesar la solución
        devolver
    fin si
    para opcion desde 0 hasta cantidad_de_opciones - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            buscar(estado)
            quitar opcion de estado
        fin si
    fin para
fin algoritmo
```

1. **Elegir**: aplicar la opción.

Elegir, explorar, deshacer

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        procesar la solución
        devolver
    fin si
    para opcion desde 0 hasta cantidad_de_opciones - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            buscar(estado)
            quitar opcion de estado
        fin si
    fin para
fin algoritmo
```

1. **Elegir**: aplicar la opción.
2. **Explorar**: la llamada recursiva. Salto de fe.

Elegir, explorar, deshacer

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        procesar la solución
        devolver
    fin si
    para opcion desde 0 hasta cantidad_de_opciones - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            buscar(estado)
            quitar opcion de estado
        fin si
    fin para
fin algoritmo
```

1. **Elegir**: aplicar la opción.
2. **Explorar**: la llamada recursiva. Salto de fe.
3. **Deshacer**: dejar el estado **exactamente** como estaba.

Elegir, explorar, deshacer

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        procesar la solución
        devolver
    fin si
    para opcion desde 0 hasta cantidad_de_opciones - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            buscar(estado)
            quitar opcion de estado
        fin si
    fin para
fin algoritmo
```

1. **Elegir**: aplicar la opción.
2. **Explorar**: la llamada recursiva. Salto de fe.
3. **Deshacer**: dejar el estado **exactamente** como estaba.

Hay **un solo estado**, compartido por todas las llamadas: se pasa un puntero, no una copia.

La promesa de la función

Promesa

Cuando `buscar` retorna, el estado está **como se lo dieron**.

La promesa de la función

Promesa

Cuando `buscar` retorna, el estado está **como se lo dieron**.

- Es inducción, como en la semana 9: supongan que la llamada interna la cumple, y comprueben que esta también.

La promesa de la función

Promesa

Cuando `buscar` retorna, el estado está **como se lo dieron**.

- Es inducción, como en la semana 9: supongan que la llamada interna la cumple, y comprueben que esta también.
- Quien llamó puede seguir con su siguiente opción como si nada hubiera pasado.

La promesa de la función

Promesa

Cuando `buscar` retorna, el estado está **como se lo dieron**.

- Es inducción, como en la semana 9: supongan que la llamada interna la cumple, y comprueben que esta también.
- Quien llamó puede seguir con su siguiente opción como si nada hubiera pasado.
- Se podría pasar una copia en cada llamada: funciona, y se paga una copia en cada nodo.

SECCIÓN 02

Subconjuntos y permutaciones

02

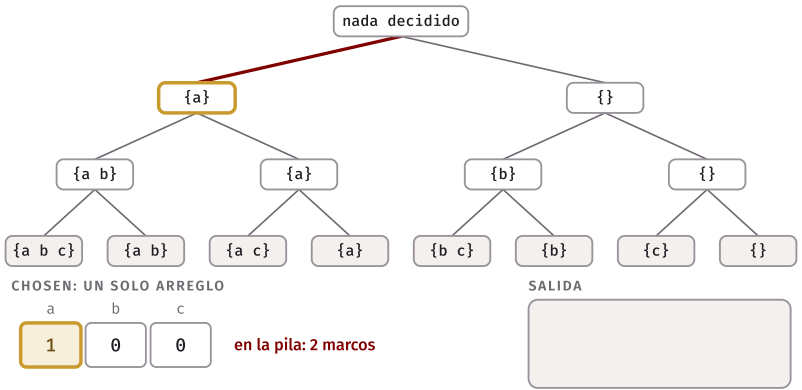
Subconjuntos: dos opciones en cada nivel

subsets.c

```c
static void subsets(const char *items, int *chosen, int n, int index, long *nodes) {
    (*nodes)++;
    if (index == n) {
        print_subset(items, chosen, n);
        return;
    }
    chosen[index] = 1;                              // elegir: se toma
    subsets(items, chosen, n, index + 1, nodes);    // explorar
    chosen[index] = 0;                              // deshacer: no se toma
    subsets(items, chosen, n, index + 1, nodes);    // explorar
}
```

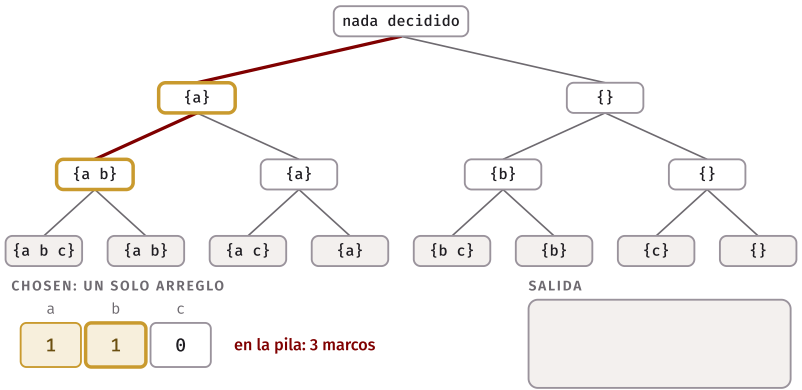
Con dos opciones fijas no hace falta el ciclo. El contador de nodos va por puntero: nada de variables globales.

Sigan chosen paso a paso



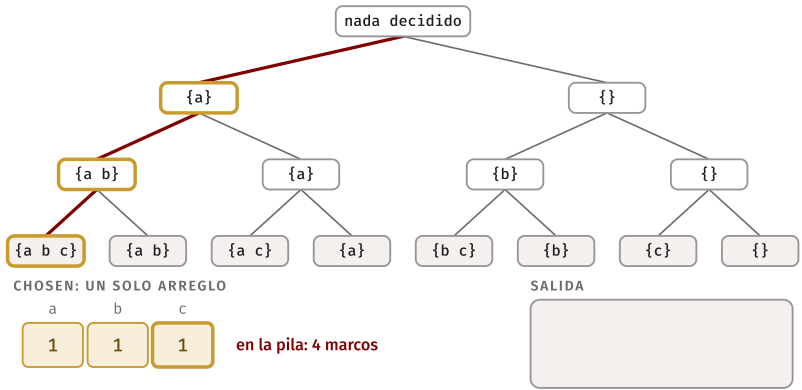
Paso 1. Elige a.

Sigan chosen paso a paso



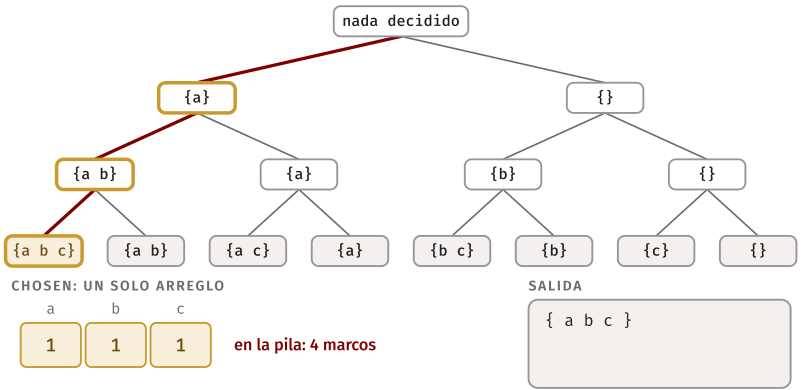
Paso 2. Elige b.

Sigan chosen paso a paso



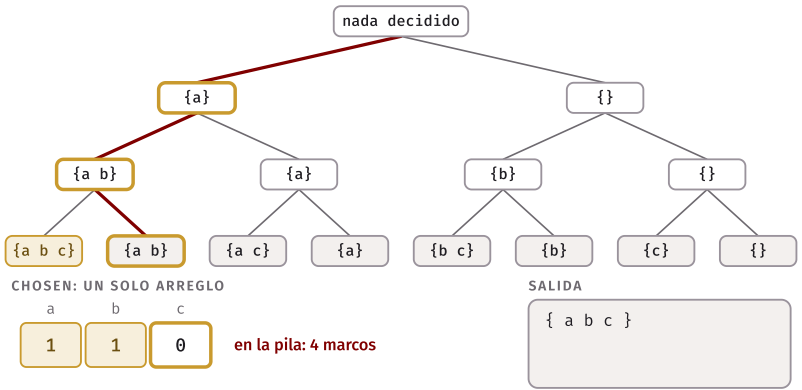
Paso 3. Elige c.

Sigan chosen paso a paso



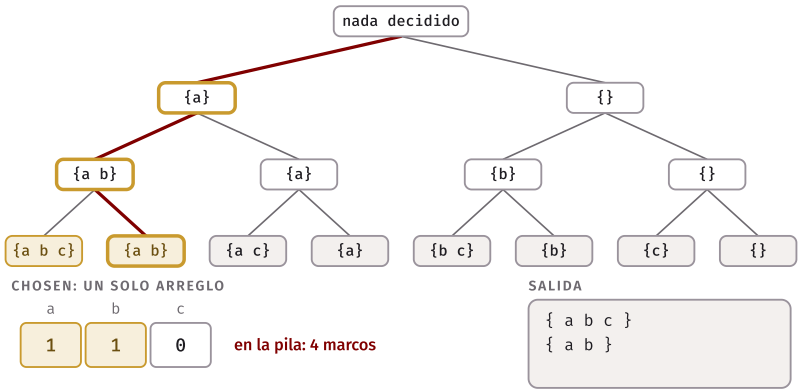
Paso 4. index == 3: escribe { a b c }.

Sigan chosen paso a paso



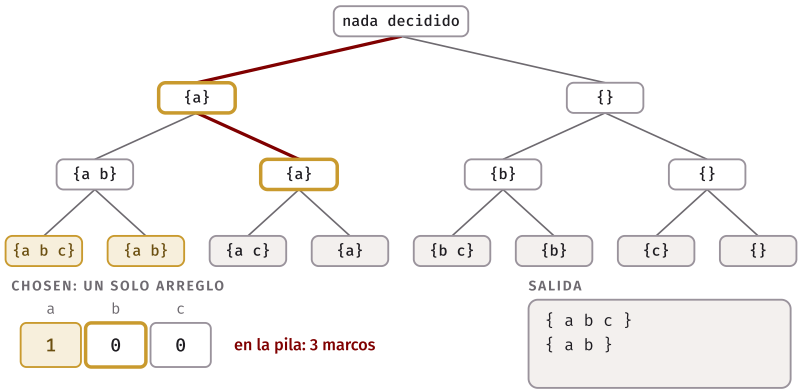
Paso 5. Deshace c, y explora sin él.

Sigan chosen paso a paso



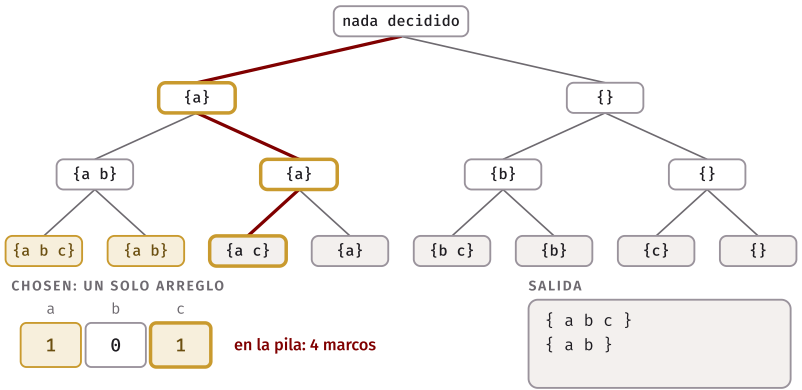
Paso 6. index == 3: escribe { a b }.

Sigan chosen paso a paso



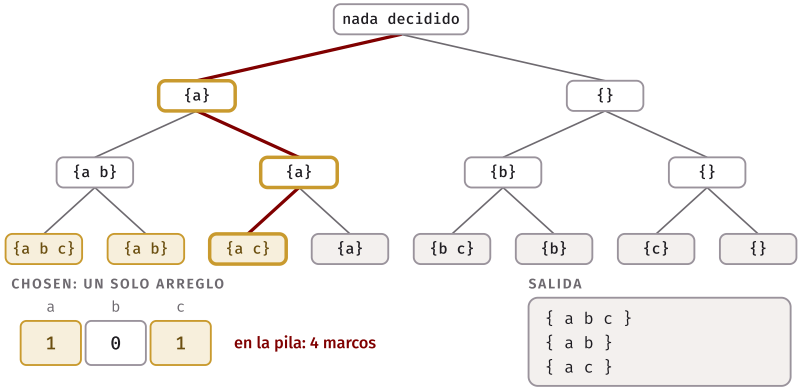
Paso 7. Retorna a la decisión de b: la deshace.

Sigan chosen paso a paso



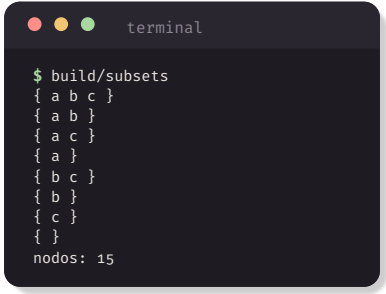
Paso 8. Vuelve a decidir c: lo elige.

Sigan chosen paso a paso

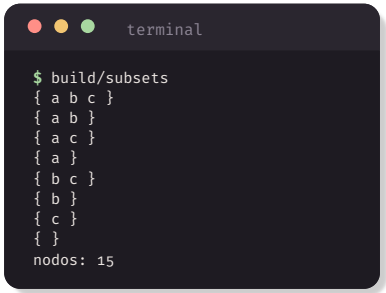


Paso 9. index == 3: escribe { a c }.

Hojas y nodos

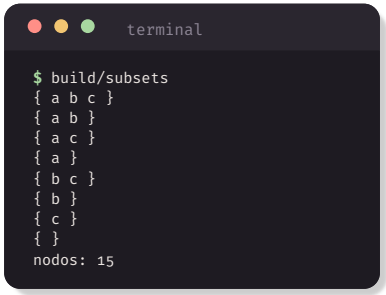


Hojas y nodos



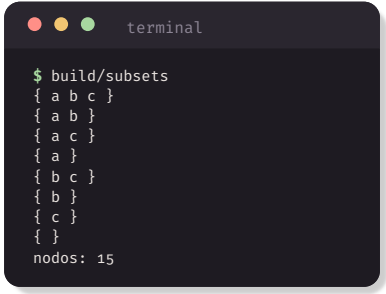
- El orden de las líneas es el de las hojas del dibujo.

Hojas y nodos



- El orden de las líneas es el de las hojas del dibujo.
- Cada elemento duplica los caminos: 2^n **hojas**.

Hojas y nodos



- El orden de las líneas es el de las hojas del dibujo.
- Cada elemento duplica los caminos: 2^n **hojas**.
- Los nodos son $1 + 2 + 4 + 8 = 15$. En general, $2^{n+1} - 1$.

Si olvidan deshacer

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subsets_no_undo.c -o build/subsets_no_undo
$ build/subsets_no_undo
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
```

Si olvidan deshacer

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subsets_no_undo.c -o build/subsets_no_undo
$ build/subsets_no_undo
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
```

- El árbol se recorre completo: llegan las 8 hojas. Pero todas ven el mismo estado.

Si olvidan deshacer

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subsets_no_undo.c -o build/subsets_no_undo
$ build/subsets_no_undo
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
```

- El árbol se recorre completo: llegan las 8 hojas. Pero todas ven el mismo estado.
- Compila sin avisos, y ASan no dice nada: ninguna regla de C se rompió, solo una promesa del programa.

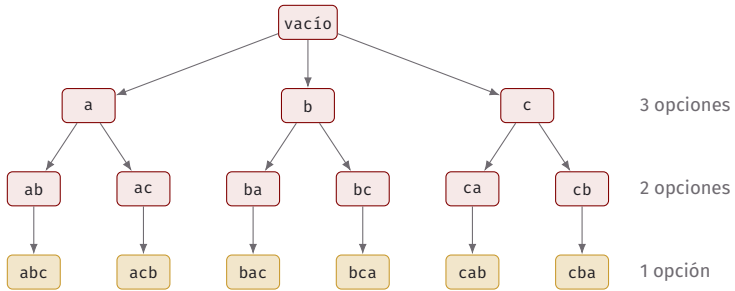
Si olvidan deshacer

terminal

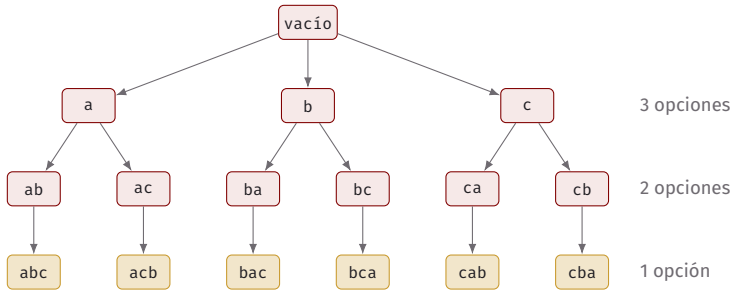
```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/subsets_no_undo.c -o build/subsets_no_undo
$ build/subsets_no_undo
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
{ a b c }
```

- El árbol se recorre completo: llegan las 8 hojas. Pero todas ven el mismo estado.
- Compila sin avisos, y ASan no dice nada: ninguna regla de C se rompió, solo una promesa del programa.
- Sin un caso calculado a mano, este bug no se ve.

Permutaciones: las opciones se acaban

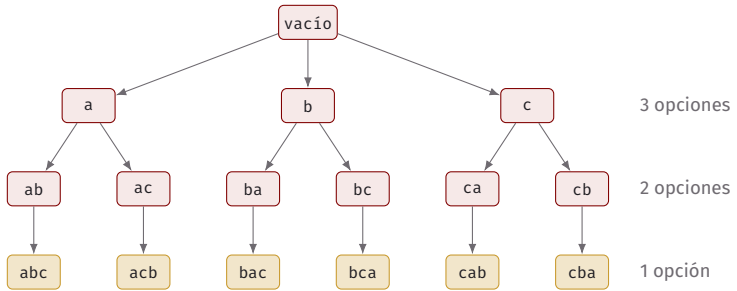


Permutaciones: las opciones se acaban



- $3 \times 2 \times 1 = 6$ hojas. Con n elementos, $n!$ hojas.

Permutaciones: las opciones se acaban



- $3 \times 2 \times 1 = 6$ hojas. Con n elementos, $n!$ hojas.
- No toda opción es válida en cada nodo: hay que saber qué elementos ya se usaron.

En C: un arreglo `used` es parte del estado

permutations.c

```c
(*nodes)++;
if (depth == n) {
    current[n] = '\0';
    printf("%s\n", current);
    return;
}
for (int i = 0; i < n; i++) {
    if (!used[i]) {
        used[i] = 1;                                          // elegir
        current[depth] = items[i];
        permutations(items, n, current, used, depth + 1, nodes);    // explorar
        used[i] = 0;                                          // deshacer
    }
}
```

En C: un arreglo `used` es parte del estado

permutations.c

```c
    (*nodes)++;
    if (depth == n) {
        current[n] = '\0';
        printf("%s\n", current);
        return;
    }
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            used[i] = 1;                                          // elegir
            current[depth] = items[i];
            permutations(items, n, current, used, depth + 1, nodes);    // explorar
            used[i] = 0;                                          // deshacer
        }
    }
```

¿Por qué no se deshace `current[depth]`? Nadie lo lee antes de volver a escribirlo.

En C: un arreglo `used` es parte del estado

permutations.c

```c
    (*nodes)++;
    if (depth == n) {
        current[n] = '\0';
        printf("%s\n", current);
        return;
    }
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            used[i] = 1;                                          // elegir
            current[depth] = items[i];
            permutations(items, n, current, used, depth + 1, nodes);    // explorar
            used[i] = 0;                                          // deshacer
        }
    }
```

¿Por qué no se deshace `current[depth]`? Nadie lo lee antes de volver a escribirlo. **Solo hay que deshacer lo que una decisión posterior va a leer.**

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`
Nivel 2 deshace `c`. Su ciclo termina	`1 1 0`	`abc`

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`
Nivel 2 deshace `c`. Su ciclo termina	`1 1 0`	`abc`
Nivel 1 deshace `b`	`1 0 0`	`abc`

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`
Nivel 2 deshace `c`. Su ciclo termina	`1 1 0`	`abc`
Nivel 1 deshace `b`	`1 0 0`	`abc`
Nivel 1 sigue con `i = 2`: elige `c`	`1 0 1`	`acc`

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`
Nivel 2 deshace `c`. Su ciclo termina	`1 1 0`	`abc`
Nivel 1 deshace `b`	`1 0 0`	`abc`
Nivel 1 sigue con `i = 2`: elige `c`	`1 0 1`	`acc`
Nivel 2: solo `b` está libre	`1 1 1`	`acb`

Después de `abc`, ¿por qué sigue `acb`?

Momento	`used`	`current`
Llega a la hoja	`1 1 1`	`abc`
Nivel 2 deshace `c`. Su ciclo termina	`1 1 0`	`abc`
Nivel 1 deshace `b`	`1 0 0`	`abc`
Nivel 1 sigue con `i = 2`: elige `c`	`1 0 1`	`acc`
Nivel 2: solo `b` está libre	`1 1 1`	`acb`

Fíjense en `current`

Pasa por `acc`, que no es una permutación. No importa: solo se lee en las hojas. Si olvidan `used[i] = 0`, el programa solo escribe `abc`.

¿Cuánto crecen? Se cuenta de verdad

terminal

```
$ build/count_trees
 n | subconjuntos    nodos | permutaciones      nodos
 1 |           2        3 |             1          2
 2 |           4        7 |             2          5
 3 |           8       15 |             6         16
 4 |          16       31 |            24         65
 5 |          32       63 |           120        326
 6 |          64      127 |           720       1957
 7 |         128      255 |          5040      13700
 8 |         256      511 |         40320     109601
 9 |         512     1023 |        362880     986410
10 |        1024     2047 |       3628800    9864101
```

¿Cuánto crecen? Se cuenta de verdad

terminal

```
$ build/count_trees
 n | subconjuntos    nodos | permutaciones      nodos
 1 |            2        3 |             1          2
 2 |            4        7 |             2          5
 3 |            8       15 |             6         16
 4 |           16       31 |            24         65
 5 |           32       63 |           120        326
 6 |           64      127 |           720       1957
 7 |          128      255 |          5040      13700
 8 |          256      511 |         40320     109601
 9 |          512     1023 |        362880     986410
10 |         1024     2047 |       3628800    9864101
```

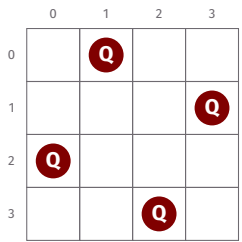
Antes de escribir un backtracking, calculen las hojas. Con 20 elementos, 2^{20} se recorre sin problema. Pero 20! tiene 19 cifras.

SECCIÓN 03

N reinas

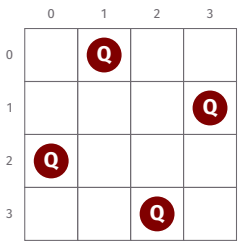
03

N reinas: primero, qué es una elección



N reinas en un tablero de N $\times$ N, sin que ninguna ataque a otra.

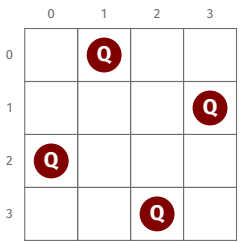
N reinas: primero, qué es una elección



N reinas en un tablero de N $\times$ N, sin que ninguna ataque a otra.

- “Una reina en cualquier casilla”: para N = 8 son 4426165368 formas.

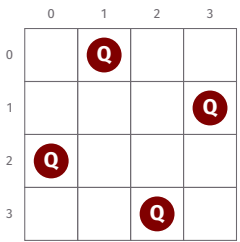
N reinas: primero, qué es una elección



N reinas en un tablero de N $\times$ N, sin que ninguna ataque a otra.

- "Una reina en cualquier casilla": para N = 8 son 4426165368 formas.
- Pero hay **una reina por fila**. Decisión: en qué columna va la reina de esta fila. Son $8^8 = 16777216$ hojas.

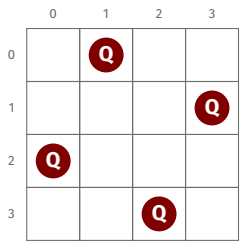
N reinas: primero, qué es una elección



N reinas en un tablero de N $\times$ N, sin que ninguna ataque a otra.

- "Una reina en cualquier casilla": para N = 8 son 4426165368 formas.
- Pero hay **una reina por fila**. Decisión: en qué columna va la reina de esta fila. Son $8^8 = 16777216$ hojas.
- Ya no se pueden representar dos reinas en una fila, igual que en las uniones etiquetadas de la semana 7.

N reinas: primero, qué es una elección



N reinas en un tablero de N $\times$ N, sin que ninguna ataque a otra.

- "Una reina en cualquier casilla": para N = 8 son 4426165368 formas.
- Pero hay **una reina por fila**. Decisión: en qué columna va la reina de esta fila. Son $8^8 = 16777216$ hojas.
- Ya no se pueden representar dos reinas en una fila, igual que en las uniones etiquetadas de la semana 7.

Idea clave

Una buena representación es la primera poda: más de 260 veces menos, antes de escribir una línea.

Sin poda: se revisa solo al final

queens_no_prune.c

```c
static long search(int *column_of_row, int n, int row, long *nodes) {
    if (row == n) {
        return all_safe(column_of_row, n);
    }
    long solutions = 0;
    for (int col = 0; col < n; col++) {
        column_of_row[row] = col;
        solutions += search(column_of_row, n, row + 1, nodes);
    }
    return solutions;
}
```

terminal

```
$ build/queens_no_prune 8
n = 8, sin poda: 92 soluciones, 19173961 nodos
```

Con poda: ¿cómo saber rápido si una casilla está libre?

`row + col`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`row - col + 3`

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

Con poda: ¿cómo saber rápido si una casilla está libre?

`row + col`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`row - col + 3`

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

- Los números iguales forman una diagonal. Hay $2n - 1$ en cada sentido.

Con poda: ¿cómo saber rápido si una casilla está libre?

`row + col`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`row - col + 3`

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

- Los números iguales forman una diagonal. Hay $2n - 1$ en cada sentido.
- El `+ 3`, que es `n - 1`, solo está para que el índice no sea negativo.

Con poda: ¿cómo saber rápido si una casilla está libre?

`row + col`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`row - col + 3`

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

- Los números iguales forman una diagonal. Hay $2n - 1$ en cada sentido.
- El `+ 3`, que es `n - 1`, solo está para que el índice no sea negativo.
- Tres arreglos: columnas, diagonales que bajan y diagonales que suben. Tres lecturas para saber si cabe.

Tres lecturas y tres escrituras

queens.c

```c
static int can_place(const Queens *q, int row, int col) {
    return !q->column_used[col]
        && !q->down_used[row - col + q->n - 1]
        && !q->up_used[row + col];
}

// Pone una reina con value = 1 y la quita con value = 0.
static void mark(Queens *q, int row, int col, int value) {
    q->column_used[col] = value;
    q->down_used[row - col + q->n - 1] = value;
    q->up_used[row + col] = value;
}
```

Tres lecturas y tres escrituras

queens.c

```c
static int can_place(const Queens *q, int row, int col) {
    return !q->column_used[col]
        && !q->down_used[row - col + q->n - 1]
        && !q->up_used[row + col];
}

// Pone una reina con value = 1 y la quita con value = 0.
static void mark(Queens *q, int row, int col, int value) {
    q->column_used[col] = value;
    q->down_used[row - col + q->n - 1] = value;
    q->up_used[row + col] = value;
}
```

Elegir y deshacer son **la misma función** con otro valor: así es difícil olvidarse de una parte.

Con poda: el esquema, tal cual

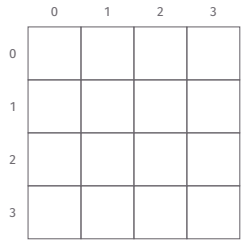
queens.c

```c
static void search(Queens *q, int row) {
    ...
    for (int col = 0; col < q->n; col++) {
        if (can_place(q, row, col)) {
            mark(q, row, col, 1);       // elegir
            search(q, row + 1);         // explorar
            mark(q, row, col, 0);       // deshacer
        }
    }
}
```

terminal

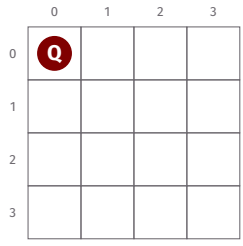
```
$ build/queens 8
n = 8, con poda: 92 soluciones, 2057 nodos
```

4 reinas, siguiendo la búsqueda



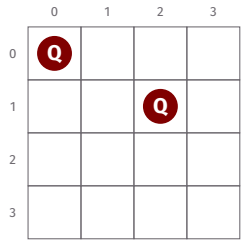
Tablero vacío. La fila 0 tiene cuatro opciones, y se prueba primero la columna 0.

4 reinas, siguiendo la búsqueda



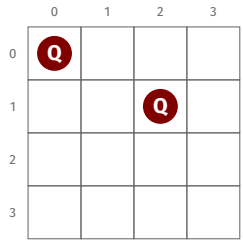
En (0, 0). En la fila 1, la columna 0 está ocupada y la 1 está en la diagonal.

4 reinas, siguiendo la búsqueda



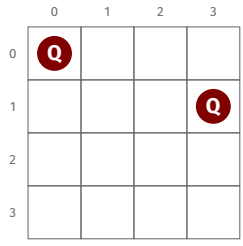
Se prueba (1, 2). Miren la fila 2.

4 reinas, siguiendo la búsqueda



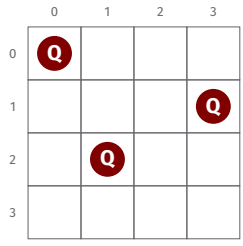
Sin lugar en la fila 2: columnas 0 y 2 ocupadas, 1 y 3 en diagonal. Se vuelve: se deshace (1, 2).

4 reinas, siguiendo la búsqueda



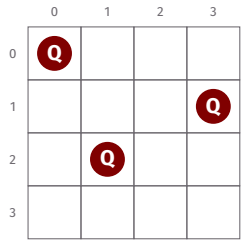
Se prueba (1, 3).

4 reinas, siguiendo la búsqueda



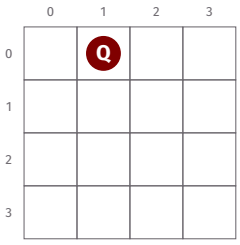
Se prueba (2, 1).

4 reinas, siguiendo la búsqueda



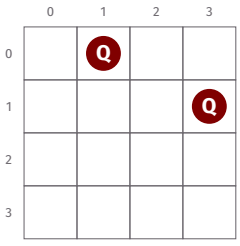
Sin lugar en la fila 3. Se vuelve hasta la fila 0: se deshacen (2, 1), (1, 3) y (0, 0).

4 reinas, siguiendo la búsqueda



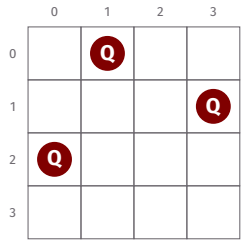
Se prueba (0, 1).

4 reinas, siguiendo la búsqueda



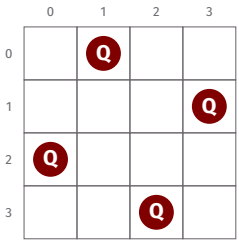
La fila 1 solo admite (1, 3).

4 reinas, siguiendo la búsqueda



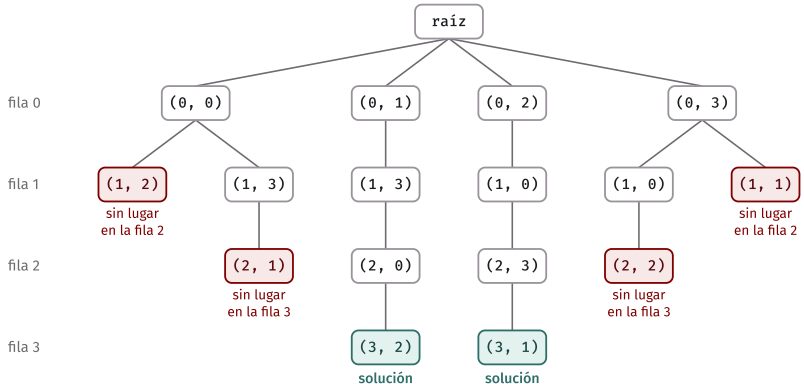
La fila 2 solo admite (2, 0).

4 reinas, siguiendo la búsqueda



Solución. La primera, de dos: el programa completo visita 17 nodos, y sin poda serían 341.

El árbol completo de 4 reinas



(fila, columna) de la reina que se puso. 17 nodos: la raíz y 16 reinas puestas.

Con poda y sin poda

N	Soluciones	Nodos sin poda	Nodos con poda
4	2	341	17
5	10	3906	54
6	4	55987	153
7	40	960800	552
8	92	19173961	2057

Con poda y sin poda

N	Soluciones	Nodos sin poda	Nodos con poda
4	2	341	17
5	10	3906	54
6	4	55987	153
7	40	960800	552
8	92	19173961	2057

- La columna de soluciones no cambia. Eso es lo que hay que exigirle a una poda.

Con poda y sin poda

N	Soluciones	Nodos sin poda	Nodos con poda
4	2	341	17
5	10	3906	54
6	4	55987	153
7	40	960800	552
8	92	19173961	2057

- La columna de soluciones no cambia. Eso es lo que hay que exigirle a una poda.
- Con 8 reinas, unas 9300 veces menos nodos.

Con poda y sin poda

N	Soluciones	Nodos sin poda	Nodos con poda
4	2	341	17
5	10	3906	54
6	4	55987	153
7	40	960800	552
8	92	19173961	2057

- La columna de soluciones no cambia. Eso es lo que hay que exigirle a una poda.
- Con 8 reinas, unas 9300 veces menos nodos.
- Con N = 12, sin poda serían más de 9 millones de millones de nodos. Con poda:

```
n = 12, con poda: 14200 soluciones, 856189 nodos
```

Deshacer a medias

errors/queens_partial_undo.c

```c
q->column_used[col] = 0;    // deshace solo una parte
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/queens_partial_undo.c -o build/queens_partial_undo
$ build/queens_partial_undo 8
n = 8, con poda: 0 soluciones, 11 nodos
```

Deshacer a medias

errors/queens_partial_undo.c

```c
q->column_used[col] = 0;    // deshace solo una parte
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/queens_partial_undo.c -o build/queens_partial_undo
$ build/queens_partial_undo 8
n = 8, con poda: 0 soluciones, 11 nodos
```

- Cero soluciones, en un problema que tiene 92, sin un solo aviso.

Deshacer a medias

errors/queens_partial_undo.c

```
1  q->column_used[col] = 0;    // deshace solo una parte
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/queens_partial_undo.c -o build/queens_partial_undo
$ build/queens_partial_undo 8
n = 8, con poda: 0 soluciones, 11 nodos
```

- Cero soluciones, en un problema que tiene 92, sin un solo aviso.
- Cada reina quitada dejó sus diagonales marcadas: casillas prohibidas que ya nadie ataca.

Deshacer a medias

errors/queens_partial_undo.c

```c
q->column_used[col] = 0;    // deshace solo una parte
```

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/queens_partial_undo.c -o build/queens_partial_undo
$ build/queens_partial_undo 8
n = 8, con poda: 0 soluciones, 11 nodos
```

- Cero soluciones, en un problema que tiene 92, sin un solo aviso.
- Cada reina quitada dejó sus diagonales marcadas: casillas prohibidas que ya nadie ataca.
- Deshacer es **exactamente** lo contrario de elegir, con los mismos índices, en todas las partes del estado.

SECCIÓN 04

Una solución o todas

04

Qué cambia: el retorno

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        devolver 1
    fin si
    para opcion desde 0 hasta total - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            si buscar(estado) = 1 entonces
                devolver 1
            fin si
            quitar opcion de estado
        fin si
    fin para
    devolver 0
fin algoritmo
```

Para buscar **todas**, se procesa cada hoja y se deshace siempre. Para buscar **la primera**:

Qué cambia: el retorno

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        devolver 1
    fin si
    para opcion desde 0 hasta total - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            si buscar(estado) = 1 entonces
                devolver 1
            fin si
            quitar opcion de estado
        fin si
    fin para
    devolver 0
fin algoritmo
```

Para buscar **todas**, se procesa cada hoja y se deshace siempre. Para buscar **la primera**:

- La función devuelve `int`: 1 o 0.

Qué cambia: el retorno

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        devolver 1
    fin si
    para opcion desde 0 hasta total - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            si buscar(estado) = 1 entonces
                devolver 1
            fin si
            quitar opcion de estado
        fin si
    fin para
    devolver 0
fin algoritmo
```

Para buscar **todas**, se procesa cada hoja y se deshace siempre. Para buscar **la primera**:

- La función devuelve `int`: 1 o 0.
- En una hoja válida, devuelve 1.

Qué cambia: el retorno

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        devolver 1
    fin si
    para opcion desde 0 hasta total - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            si buscar(estado) = 1 entonces
                devolver 1
            fin si
            quitar opcion de estado
        fin si
    fin para
    devolver 0
fin algoritmo
```

Para buscar **todas**, se procesa cada hoja y se deshace siempre. Para buscar **la primera**:

- La función devuelve `int`: 1 o 0.
- En una hoja válida, devuelve 1.
- Si la llamada dio 1, retorna **sin deshacer**.

Qué cambia: el retorno

pseudocódigo

```
algoritmo buscar(estado)
    si estado está completo entonces
        devolver 1
    fin si
    para opcion desde 0 hasta total - 1 hacer
        si opcion es válida en estado entonces
            aplicar opcion a estado
            si buscar(estado) = 1 entonces
                devolver 1
            fin si
            quitar opcion de estado
        fin si
    fin para
    devolver 0
fin algoritmo
```

Para buscar **todas**, se procesa cada hoja y se deshace siempre. Para buscar **la primera**:

- La función devuelve `int`: 1 o 0.
- En una hoja válida, devuelve 1.
- Si la llamada dio 1, retorna **sin deshacer**.
- Si se acaban las opciones, devuelve 0.

Primera solución: el estado es la respuesta

queens_first.c

```c
static int search(Queens *q, int row) {
    if (row == q->n) {
        return 1;
    }
    for (int col = 0; col < q->n; col++) {
        if (can_place(q, row, col)) {
            mark(q, row, col, 1);
            q->column_of_row[row] = col;
            if (search(q, row + 1)) {
                return 1;
            }
            mark(q, row, col, 0);
        }
    }
    return 0;
}
```

terminal

```
$ build/queens_first 8
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
n = 8, primera solución: 114 nodos
```

114 nodos, en vez de 2057.

Primera solución: el estado es la respuesta

queens_first.c

```c
static int search(Queens *q, int row) {
    if (row == q->n) {
        return 1;
    }
    for (int col = 0; col < q->n; col++) {
        if (can_place(q, row, col)) {
            mark(q, row, col, 1);
            q->column_of_row[row] = col;
            if (search(q, row + 1)) {
                return 1;
            }
            mark(q, row, col, 0);
        }
    }
    return 0;
}
```

terminal

```
$ build/queens_first 8
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
n = 8, primera solución: 114 nodos
```

114 nodos, en vez de 2057. Si devuelve 1, el estado tiene una solución. Si devuelve 0, está como se lo dieron.

Si no revisan el retorno

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/first_ignored.c -o build/first_ignored
$ build/first_ignored 8
no hay solución
n = 8, primera solución: 2057 nodos
```

Si no revisan el retorno

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/first_ignored.c -o build/first_ignored
$ build/first_ignored 8
no hay solución
n = 8, primera solución: 2057 nodos
```

- En `errors/first_ignored.c`, la línea es `search(q, row + 1);` sola.

Si no revisan el retorno

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/first_ignored.c -o build/first_ignored
$ build/first_ignored 8
no hay solución
n = 8, primera solución: 2057 nodos
```

- En `errors/first_ignored.c`, la línea es `search(q, row + 1);` sola.
- Los 2057 nodos lo delatan: encontró las 92 soluciones, ignoró cada aviso y al final devolvió 0.

Si no revisan el retorno

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/first_ignored.c -o build/first_ignored
$ build/first_ignored 8
no hay solución
n = 8, primera solución: 2057 nodos
```

- En `errors/first_ignored.c`, la línea es `search(q, row + 1);` sola.
- Los 2057 nodos lo delatan: encontró las 92 soluciones, ignoró cada aviso y al final devolvió 0.
- `-Wall -Wextra` no avisa. No usar lo que devuelve una función no es un error.

SECCIÓN 05

Poda

05

¿La poda puede perder una solución?

Sí.

¿La poda puede perder una solución?

Sí.

- Una poda es una afirmación: “debajo de este nodo no hay ninguna solución”.

¿La poda puede perder una solución?

Sí.

- Una poda es una afirmación: “debajo de este nodo no hay ninguna solución”.
- Si es verdadera, ahorra trabajo sin cambiar el resultado.

¿La poda puede perder una solución?

Sí.

- Una poda es una afirmación: “debajo de este nodo no hay ninguna solución”.
- Si es verdadera, ahorra trabajo sin cambiar el resultado.
- Si es falsa, da una respuesta equivocada, rápido y sin avisos.

¿La poda puede perder una solución?

Sí.

- Una poda es una afirmación: “debajo de este nodo no hay ninguna solución”.
- Si es verdadera, ahorra trabajo sin cambiar el resultado.
- Si es falsa, da una respuesta equivocada, rápido y sin avisos.
- En las reinas: dos reinas que se atacan seguirán atacándose, porque nunca se mueven.

¿La poda puede perder una solución?

Sí.

- Una poda es una afirmación: “debajo de este nodo no hay ninguna solución”.
- Si es verdadera, ahorra trabajo sin cambiar el resultado.
- Si es falsa, da una respuesta equivocada, rápido y sin avisos.
- En las reinas: dos reinas que se atacan seguirán atacándose, porque nunca se mueven.

Idea clave

Escriban qué suposición justifica cada poda.

Un menú con presupuesto

Carta con 12 platos y 30 soles. ¿Cuántas combinaciones caben? Si ya se pasó, ningún plato lo arregla: los precios son positivos.

combos.c: count_pruned

```
1       (*nodes)++;
2       if (total > budget) {
3           return 0;
4       }
```

terminal

```
$ build/combos
sin poda: 249 combinaciones, 8191 nodos
con poda: 249 combinaciones, 1265 nodos
```

Un menú con presupuesto

Carta con 12 platos y 30 soles. ¿Cuántas combinaciones caben? Si ya se pasó, ningún plato lo arregla: los precios son positivos.

combos.c: count_pruned

```
1       (*nodes)++;
2       if (total > budget) {
3           return 0;
4       }
```

terminal

```
$ build/combos
sin poda: 249 combinaciones, 8191 nodos
con poda: 249 combinaciones, 1265 nodos
```

`total` va **por valor**: cada marco tiene el suyo, y no hay nada que deshacer.

Cuando la poda miente: un vale de descuento

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/bad_prune.c -o build/bad_prune
$ build/bad_prune
sin poda: 5 combinaciones
con poda: 4 combinaciones
```

Precios `{8, 5, -5}`, presupuesto 6.

Cuando la poda miente: un vale de descuento

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/bad_prune.c -o build/bad_prune
$ build/bad_prune
sin poda: 5 combinaciones
con poda: 4 combinaciones
```

Precios `{8, 5, -5}`, presupuesto 6.

Subconjunto	Total	
{}	0	sí
{8}	8	no
{5}	5	sí
{−5}	−5	sí
{8, 5}	13	no
{8, −5}	3	**sí**
{5, −5}	0	sí
{8, 5, −5}	8	no

Cuando la poda miente: un vale de descuento

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/bad_prune.c -o build/bad_prune
$ build/bad_prune
sin poda: 5 combinaciones
con poda: 4 combinaciones
```

Precios $\{8, 5, -5\}$, presupuesto 6.

Al tomar el 8, ya pasó de 6, y la poda cortó esa rama. Más abajo estaba $\{8, -5\}$.

Subconjunto	Total	
$\{\}$	0	sí
$\{8\}$	8	no
$\{5\}$	5	sí
$\{-5\}$	-5	sí
$\{8, 5\}$	13	no
$\{8, -5\}$	3	**sí**
$\{5, -5\}$	0	sí
$\{8, 5, -5\}$	8	no

Cuando la poda miente: un vale de descuento

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/bad_prune.c -o build/bad_prune
$ build/bad_prune
sin poda: 5 combinaciones
con poda: 4 combinaciones
```

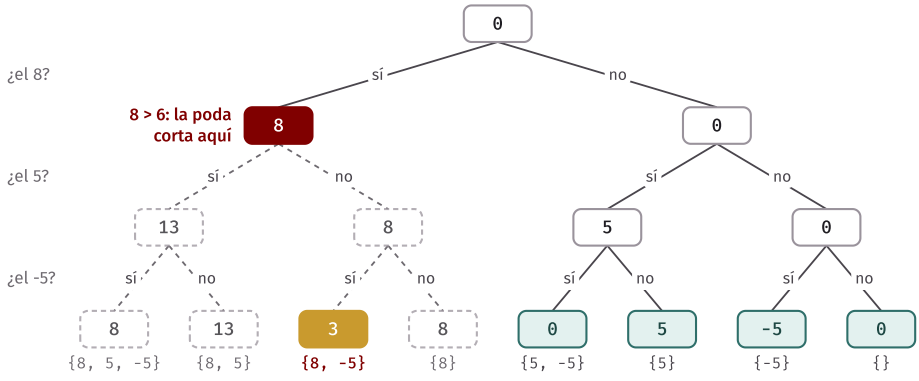
Precios $\{8, 5, -5\}$, presupuesto 6.

Al tomar el 8, ya pasó de 6, y la poda cortó esa rama. Más abajo estaba $\{8, -5\}$.

“Si ya me pasé, no hay arreglo” dependía de que los precios fueran positivos.

Subconjunto	Total	
$\{\}$	0	sí
$\{8\}$	8	no
$\{5\}$	5	sí
$\{-5\}$	-5	sí
$\{8, 5\}$	13	no
$\{8, -5\}$	3	**sí**
$\{5, -5\}$	0	sí
$\{8, 5, -5\}$	8	no

La rama que cortó la poda



Elegir bien también poda

En un sudoku, cada celda vacía se tiene que llenar, y el orden lo elegimos nosotros. ¿Y si elegimos la de **menos opciones**?

Elegir bien también poda

En un sudoku, cada celda vacía se tiene que llenar, y el orden lo elegimos nosotros. ¿Y si elegimos la de **menos opciones**?

- **0 opciones**: este camino no sirve, y se retorna de inmediato.

Elegir bien también poda

En un sudoku, cada celda vacía se tiene que llenar, y el orden lo elegimos nosotros. ¿Y si elegimos la de **menos opciones**?

- **0 opciones**: este camino no sirve, y se retorna de inmediato.
- **1 opción**: no hay nada que decidir, no se abre ninguna rama.

Elegir bien también poda

En un sudoku, cada celda vacía se tiene que llenar, y el orden lo elegimos nosotros. ¿Y si elegimos la de **menos opciones**?

- **0 opciones**: este camino no sirve, y se retorna de inmediato.
- **1 opción**: no hay nada que decidir, no se abre ninguna rama.
- **2 opciones**: la rama se abre lo menos posible.

Elegir bien también poda

En un sudoku, cada celda vacía se tiene que llenar, y el orden lo elegimos nosotros. ¿Y si elegimos la de **menos opciones**?

- **0 opciones**: este camino no sirve, y se retorna de inmediato.
- **1 opción**: no hay nada que decidir, no se abre ninguna rama.
- **2 opciones**: la rama se abre lo menos posible.

¿Pierde soluciones?

No: esa celda hay que llenarla de todas maneras. Es un cambio de **orden**, que hace que los cortes ocurran antes. El precio: cada nodo llama a `can_place` cientos de veces, y no hasta 9.

SECCIÓN 06

El laboratorio

06

Traduzcan el esquema al sudoku

En el esquema	En el sudoku
El estado	`board->cells`, un solo tablero
La decisión	Qué dígito va en una celda vacía
¿Completo?	No quedan celdas vacías
Opción válida	`can_place(board, row, col, digit)`
Elegir	Escribir el dígito
Deshacer	Volver a poner 0

El archivo: 9 líneas de 9 dígitos, con 0 para una celda vacía.

```
430900600
602408005
...
```

Traduzcan el esquema al sudoku

En el esquema	En el sudoku
El estado	`board->cells`, un solo tablero
La decisión	Qué dígito va en una celda vacía
¿Completo?	No quedan celdas vacías
Opción válida	`can_place(board, row, col, digit)`
Elegir	Escribir el dígito
Deshacer	Volver a poner 0

El archivo: 9 líneas de 9 dígitos, con 0 para una celda vacía.

```
430900600
602408005
...
```

Cuidado

Si ningún dígito sirve en la celda elegida, se devuelve 0. Si el ciclo sigue con otra celda, la función devuelve 1 con un tablero incompleto.

Lo que debe dar su programa

terminal

```
$ build/lab/solve lab/data/facil.txt lab/data/medio.txt lab/data/dificil.txt lab/data/sorpresa.txt
lab/data/facil.txt
  orden de lectura: resuelto, 72 nodos
  menos opciones:   resuelto, 38 nodos
lab/data/medio.txt
  orden de lectura: resuelto, 4314 nodos
  menos opciones:   resuelto, 169 nodos
lab/data/dificil.txt
  orden de lectura: resuelto, 1745319 nodos
  menos opciones:   resuelto, 1815 nodos
lab/data/sorpresa.txt
  orden de lectura: resuelto, 815 nodos
  menos opciones:   resuelto, 1116 nodos
```

Lo que debe dar su programa

terminal

```
$ build/lab/solve lab/data/facil.txt lab/data/medio.txt lab/data/dificil.txt lab/data/sorpresa.txt
lab/data/facil.txt
  orden de lectura: resuelto, 72 nodos
  menos opciones:   resuelto, 38 nodos
lab/data/medio.txt
  orden de lectura: resuelto, 4314 nodos
  menos opciones:   resuelto, 169 nodos
lab/data/dificil.txt
  orden de lectura: resuelto, 1745319 nodos
  menos opciones:   resuelto, 1815 nodos
lab/data/sorpresa.txt
  orden de lectura: resuelto, 815 nodos
  menos opciones:   resuelto, 1116 nodos
```

Miren el último: **menos opciones no siempre gana**. Es un criterio que suele ayudar, no una garantía. Hay que medir.

Laboratorio: una sesión, unos 90 minutos

1. Leer `board.h`, `solver.h` y `subset_sum.h`, y responder las preguntas
2. Escribir `subset_sum`, y contar los nodos
3. Escribir `can_place` y el solucionador en orden de lectura
4. Escribir `count_options` y el solucionador con menos opciones. Comparar nodos

La otra sesión es la Práctica 3. El orden deja lo más importante primero: la parte 4 puede quedar para después de la sesión.

Resumen

- **Backtracking**: elegir, explorar, deshacer, sobre un árbol de decisiones. En la pila, solo el camino actual.
- La promesa: al retornar, el estado está como lo recibió.
- Solo hay que deshacer lo que se comparte y lo que una decisión posterior va a leer.
- **Subconjuntos**: 2^n hojas. **Permutaciones**: $n!$ hojas. Con 5 elementos, 32 y 120.
- Olvidar deshacer no da avisos: se ve en la salida, con un caso calculado a mano.
- **Una solución**: se devuelve 1 o 0, y quien llama lo revisa. Con 1, retorna sin deshacer.
- **Poda**: es correcta solo si corta ramas sin soluciones. Con la poda, 8 reinas pasan de 19173961 nodos a 2057.
- Elegir la celda con **menos opciones** no pierde soluciones, pero cada nodo cuesta más, y no siempre gana.

Para la próxima clase

- Terminen el laboratorio: `make test-lab`, sin informes de ASan.
- Resuelvan los ejercicios de los apuntes, sobre todo el 3, el 5 y el 7.
- Respondan las preguntas de repaso de la guía de la semana 11.
- En la semana 12, el backtracker se vuelve genérico: las opciones y la validez se pasan como funciones, y el mismo código resuelve N reinas y sudoku.

Antes de escribir un backtracking

¿Cuántas hojas tiene el árbol? ¿Qué se comparte y hay que deshacer? ¿Qué suposición justifica cada poda?