

UNIVERSIDAD NACIONAL DE INGENIERÍA

Introducción a la Computación

Búsqueda, ordenamiento y eficiencia

Contar pasos, medir tiempos y decidir cuándo ordenar

Gerson Garrido

Semana 13

Hoy

1. Dos formas de buscar: lineal y binaria
2. Contar comparaciones en lugar de medir segundos
3. Cómo crecen N, $\log N$ y N^2
4. Ordenamiento por selección, y `qsort` como caja negra
5. Medir tiempo con `clock`, y medirlo bien
6. Cuándo vale la pena ordenar antes de buscar
7. Recursión: caso base, caso recursivo y factorial

> **Lectura**
>
> Apuntes de la semana 13. Código en `code/week-13`.

SECCIÓN 01

Búsqueda

01

Búsqueda lineal: uno por uno

pseudocódigo

```
algoritmo busqueda_lineal(lista, n, valor)
    para i desde 0 hasta n - 1 hacer
        si lista[i] = valor entonces
            devolver i
        fin si
    fin para
    devolver -1
fin algoritmo
```

linear_search.c

```c
int linear_search(const int a[], int n,
     int value) {
    for (int i = 0; i < n; i++) {
        if (a[i] == value) {
            return i;
        }
    }
    return -1;
}
```

Funciona con cualquier arreglo, ordenado o no. Para decir que un valor no está, mira **todos** los elementos.

Adivinar un número del 1 al 100



Búsqueda binaria ***(binary search)***

Mirar el elemento del medio. Si es menor que el valor, se descarta la izquierda. Si es mayor, la derecha.

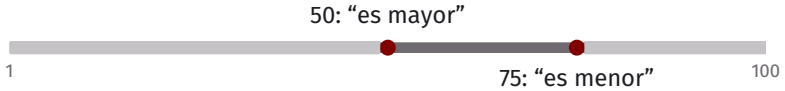
Adivinar un número del 1 al 100



Búsqueda binaria ***(binary search)***

Mirar el elemento del medio. Si es menor que el valor, se descarta la izquierda. Si es mayor, la derecha.

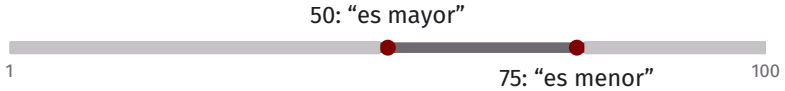
Adivinar un número del 1 al 100



Búsqueda binaria ***(binary search)***

Mirar el elemento del medio. Si es menor que el valor, se descarta la izquierda. Si es mayor, la derecha.

Adivinar un número del 1 al 100



Búsqueda binaria ***(binary search)***

Mirar el elemento del medio. Si es menor que el valor, se descarta la izquierda. Si es mayor, la derecha.

Exige datos ordenados

Al ver un elemento descarta otros sin mirarlos. Eso solo es correcto si el orden garantiza qué hay a cada lado.

Búsqueda binaria: pseudocódigo y C

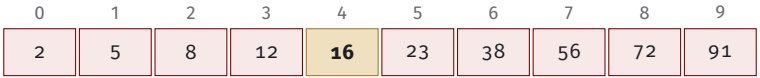
pseudocódigo

```
algoritmo busqueda_binaria(lista, n, valor)
    bajo ← 0
    alto ← n - 1
    mientras bajo ≤ alto hacer
        medio ← bajo + (alto - bajo) div 2
        si lista[medio] = valor entonces
            devolver medio
        sino si lista[medio] < valor entonces
            bajo ← medio + 1
        sino
            alto ← medio - 1
        fin si
    fin mientras
    devolver -1
fin algoritmo
```

binary_search.c

```c
int binary_search(const int a[], int n, int
      value) {
    int low = 0;
    int high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] == value) {
            return mid;
        } else if (a[mid] < value) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

Sigámosla: buscamos el 23



`low = 0, high = 9, mid = 4`. `a[4]` es 16, menor que 23: `low` pasa a 5.

terminal

```
$ echo 23 | build/binary_trace
low = 0, high = 9, mid = 4, a[mid] = 16
low = 5, high = 9, mid = 7, a[mid] = 56
low = 5, high = 6, mid = 5, a[mid] = 23
Resultado: 5
```

Miró 3 elementos de 10.

Sigámosla: buscamos el 23



`low = 5, high = 9, mid = 7`. `a[7]` es 56, mayor que 23: `high` pasa a 6.

terminal

```
$ echo 23 | build/binary_trace
low = 0, high = 9, mid = 4, a[mid] = 16
low = 5, high = 9, mid = 7, a[mid] = 56
low = 5, high = 6, mid = 5, a[mid] = 23
Resultado: 5
```

Miró 3 elementos de 10.

Sigámosla: buscamos el 23



`low = 5, high = 6, mid = 5`. `a[5]` es 23: lo encontró, y devuelve 5.

terminal

```
$ echo 23 | build/binary_trace
low = 0, high = 9, mid = 4, a[mid] = 16
low = 5, high = 9, mid = 7, a[mid] = 56
low = 5, high = 6, mid = 5, a[mid] = 23
Resultado: 5
```

Miró 3 elementos de 10.

Si el valor no está, los índices se cruzan

terminal

```
$ echo 40 | build/binary_trace
low = 0, high = 9, mid = 4, a[mid] = 16
low = 5, high = 9, mid = 7, a[mid] = 56
low = 5, high = 6, mid = 5, a[mid] = 23
low = 6, high = 6, mid = 6, a[mid] = 38
low = 7, high = 6: ya no queda nada
Resultado: -1
```

- Con `low` igual a `high` todavía queda **un** elemento: por eso es `low <= high`.
- Cuando `low` pasa a `high`, la parte que queda está vacía.
- Con valores repetidos, encuentra uno de ellos, pero no promete cuál.

Error 1: datos desordenados

El arreglo de `errors/binary_unsorted.c` tiene 5, 1, 9, 3 y 7. Buscamos cada uno:

terminal

```
$ build/binary_unsorted
El 5 está en el índice 0; binaria responde 0
El 1 está en el índice 1; binaria responde -1
El 9 está en el índice 2; binaria responde 2
El 3 está en el índice 3; binaria responde -1
El 7 está en el índice 4; binaria responde -1
```

No se rompe ni avisa

Responde mal, y a veces bien, que es peor. Con el 1: mira el 9 y descarta la derecha; mira el 5 y descarta lo que queda, **incluido el 1**.

Errores 2 y 3: el ciclo y los índices

`while (low < high)` termina cuando aún queda un elemento, sin mirarlo. Con 10, 20 y 30:

terminal

```
$ build/low_less_high
El 10 está en el índice 0; responde -1
El 20 está en el índice 1; responde 1
El 30 está en el índice 2; responde -1
```

`high = mid`, sin el `- 1`: con `low`, `high` y `mid` iguales y un valor menor, `high` no cambia, y el ciclo no termina.

Cómo evitarlos

Pruebas con los bordes: el primero, el último, un valor menor que todos, uno mayor que todos, uno entre dos, $n = 0$.

Otro cuidado: `(low + high) / 2`

La suma puede no caber en un `int`, aunque el medio sí quepa. Es overflow: **comportamiento indefinido**.

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/mid_overflow.c -o build/mid_overflow
$ echo 2000000000 2100000000 | build/mid_overflow
errors/mid_overflow.c:12:47: runtime error: signed integer overflow: 2000000000 + 2100000000 cannot be represented in type 'int'
Fórmula segura: 2050000000
Fórmula con la suma: -97483648
```

- Ese número negativo salió en esta corrida: el lenguaje no promete nada.
- `low + (high - low) / 2` nunca suma dos índices grandes.
- En nuestros ejemplos no hay arreglos tan grandes. La fórmula segura no cuesta nada, así que siempre la usamos.

SECCIÓN 02

Contar pasos

02

¿Segundos o pasos?

Los segundos dependen de:

- la máquina
- las opciones de compilación
- qué más esté corriendo

Una cuenta de pasos sale igual en cualquier computadora.

Comparación, en este capítulo

Mirar un elemento del arreglo para compararlo con el valor buscado.

La búsqueda binaria hace hasta dos preguntas por elemento, `==` y `<`, y las contamos como una. Si contáramos las dos, el total sería a lo sumo el doble, y la forma de crecer no cambiaría.

Una función puede devolver un `struct`, igual que un `int`

count_search.c

```c
struct search_result {
    int index;         // dónde está, o -1
    int comparisons;   // cuántos elementos se miraron
};

struct search_result linear_count(const int a[], int n, int value) {
    struct search_result result = {-1, 0};
    for (int i = 0; i < n; i++) {
        result.comparisons++;
        if (a[i] == value) {
            result.index = i;
            return result;
        }
    }
    return result;
}
```

Los números del laboratorio

terminal

```
$ build/count_search
       N   lineal  binaria     peor
    1000     1000       10       10
   10000    10000       14       14
  100000   100000       17       17
Promedio con N = 1000, si el valor está: lineal 500.50, binaria 8.99
```

- Buscan un valor mayor que todos. `peor`: el máximo, probando todos. Son exactos.

> **Idea clave**
>
> Con cien veces más datos, la lineal mira **cien veces más**. La binaria mira **7 elementos más**.

De dónde sale el 10

pseudocódigo

```
algoritmo mitades(n)
    cuenta ← 0
    mientras n > 0 hacer
        n ← n div 2
        cuenta ← cuenta + 1
    fin mientras
    devolver cuenta
fin algoritmo
```

Cada comparación deja, como mucho, la mitad, sin decimales.

1000: 500, 250, 125, 62, 31, 15, 7, 3, 1, 0: 10 mitades.

De dónde sale el 10

pseudocódigo

```
algoritmo mitades(n)
    cuenta ← 0
    mientras n > 0 hacer
        n ← n div 2
        cuenta ← cuenta + 1
    fin mientras
    devolver cuenta
fin algoritmo
```

Cada comparación deja, como mucho, la mitad, sin decimales.

1000: 500, 250, 125, 62, 31, 15, 7, 3, 1, 0: 10 mitades.

1 000 000: 500 000, 250 000, 125 000, 62 500, 31 250, 15 625, 7812, 3906, 1953, 976, 488, 244, 122, 61, 30, 15, 7, 3, 1, 0: 20 mitades.

De dónde sale el 10

pseudocódigo

```
algoritmo mitades(n)
    cuenta ← 0
    mientras n > 0 hacer
        n ← n div 2
        cuenta ← cuenta + 1
    fin mientras
    devolver cuenta
fin algoritmo
```

Cada comparación deja, como mucho, la mitad, sin decimales.

1000: 500, 250, 125, 62, 31, 15, 7, 3, 1, 0: 10 mitades.

1 000 000: 500 000, 250 000, 125 000, 62 500, 31 250, 15 625, 7812, 3906, 1953, 976, 488, 244, 122, 61, 30, 15, 7, 3, 1, 0: 20 mitades.

log *N*, para nosotros

Cuántas veces se puede partir *N* a la mitad. Es casi el logaritmo en base 2: difiere en 1 como mucho. Por ejemplo, $2^{10} = 1024$.

Cómo crecen N, $\log N$ y N^2

terminal

```
$ build/growth
        N   log         N             N*N
     1000    10      1000         1000000
     2000    11      2000         4000000
     4000    12      4000        16000000
     8000    13      8000        64000000
   100000    17    100000     10000000000
  1000000    20   1000000   1000000000000
```

Cómo crecen N, $\log N$ y N^2

terminal

```
$ build/growth
        N   log         N            N*N
     1000    10      1000        1000000
     2000    11      2000        4000000
     4000    12      4000       16000000
     8000    13      8000       64000000
   100000    17    100000    10000000000
  1000000    20   1000000  1000000000000
```

Si crece como	Al duplicar N	Ejemplo
$\log N$	suma un poco	búsqueda binaria
N	se duplica	búsqueda lineal
N^2	se multiplica por 4	ordenamiento por selección

Predecir con la forma de crecer

Un programa tarda 2 s con $N = 1000$. ¿Cuánto esperan con $N = 2000$?

Predecir con la forma de crecer

Un programa tarda 2 s con $N = 1000$. ¿Cuánto esperan con $N = 2000$?

Crece como	$N = 1000$	$N = 2000$	$N = 10\,000$
N	2 s	4 s	20 s
N^2	2 s	8 s	200 s

Predecir con la forma de crecer

Un programa tarda 2 s con $N = 1000$. ¿Cuánto esperan con $N = 2000$?

Crece como	$N = 1000$	$N = 2000$	$N = 10\,000$
N	2 s	4 s	20 s
N^2	2 s	8 s	200 s

Idea clave

Con muchos datos importa **cómo crece** el trabajo. Con una computadora el doble de rápida, un programa N^2 solo gana un 41 % más de datos en el mismo tiempo.

Si leen $O(N)$ u $O(N^2)$ en otro lugar, es esta misma idea, con una definición formal que verán en cursos posteriores.

SECCIÓN 03

Ordenar

03

Ordenamiento por selección

pseudocódigo

```
algoritmo ordenar_por_seleccion(lista, n)
    para i desde 0 hasta n - 2 hacer
        menor ← i
        para j desde i + 1 hasta n - 1 hacer
            si lista[j] < lista[menor]
     entonces
                menor ← j
            fin si
        fin para
        guardado ← lista[i]
        lista[i] ← lista[menor]
        lista[menor] ← guardado
    fin para
fin algoritmo
```

selection_sort.c

```c
long long selection_sort(int a[], int n) {
    long long comparisons = 0;
    for (int i = 0; i < n - 1; i++) {
        int smallest = i;
        for (int j = i + 1; j < n; j++) {
            comparisons++;
            if (a[j] < a[smallest]) {
                smallest = j;
            }
        }
        swap_at(a, i, smallest);
        ...
    }
    return comparisons;
}
```

La menor de todas va primera; luego, la menor de las que quedan.

Sigámoslo

terminal

```
$ build/selection_sort
Inicio: 29 10 14 37 13
i = 0, menor en 1: 10 29 14 37 13
i = 1, menor en 4: 10 13 14 37 29
i = 2, menor en 2: 10 13 14 37 29
i = 3, menor en 4: 10 13 14 29 37
Comparaciones: 10
Inicio: 10 13 14 29 37
i = 3, menor en 3: 10 13 14 29 37
Comparaciones: 10
```

La segunda parte está recortada. Ya estaba ordenado, y aun así hizo 10 comparaciones: para saber cuál es el menor de lo que queda, siempre tiene que mirarlo todo.

Cada pasada deja uno en su sitio



Verde: ya tiene su valor definitivo. Oro: el que salió del lugar i.

`4 + 3 + 2 + 1 = 10`

Cuántas comparaciones hace

$$(N-1) + (N-2) + \cdots + 2 + 1 = \frac{N(N-1)}{2}$$

- Exacta, y no depende de los datos.
- Con $N = 5$: $4 + 3 + 2 + 1 = 10$.
- Tiene un $N \times N$: **crece como** N^2.
- Con $N = 100\,000$: $4\,999\,950\,000$. No cabe en un `int`.

`long long`

Un entero que C garantiza que tiene al menos 64 bits. Se escribe con `%lld`.

SECCIÓN 04

Medir

04

Medir tiempo con `clock`

sort_times.c

```c
clock_t start = clock();
long long comparisons = selection_sort(work, n);
clock_t end = clock();
double selection_seconds = (double) (end - start) / CLOCKS_PER_SEC;
```

- `clock`, de `time.h`: tiempo de **procesador** que lleva usado el programa. No es la hora.
- `CLOCKS_PER_SEC`: cuántas de esas unidades tiene un segundo.

Tres cuidados

1. Sin sanitizers: hacen más lento el programa. **2.** Con un nivel de optimización declarado: medimos con `-O2`. **3.** Usando el resultado: si nadie lo usa, `-O2` puede eliminar el cálculo.

El mismo programa, compilado de tres formas

esta máquina, esta corrida

```
$ gcc -Wall -Wextra -std=c17 -O2 sort_times.c -o build/sort_times_o2
$ echo 40000 | build/sort_times_o2
      N    comparac. selección      qsort  ordenado
  40000    799980000    0.6179s    0.0032s  sí
$ gcc -Wall -Wextra -std=c17 -O0 sort_times.c -o build/sort_times_o0
$ echo 40000 | build/sort_times_o0
      N    comparac. selección      qsort  ordenado
  40000    799980000    0.9924s    0.0049s  sí
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined sort_times.c -o build/sort_times_san
$ echo 40000 | build/sort_times_san
      N    comparac. selección      qsort  ordenado
  40000    799980000    3.3838s    0.0061s  sí
```

- La cuenta de comparaciones es la misma, y exacta. El tiempo no.
- Un tiempo sin decir cómo se compiló no significa casi nada.

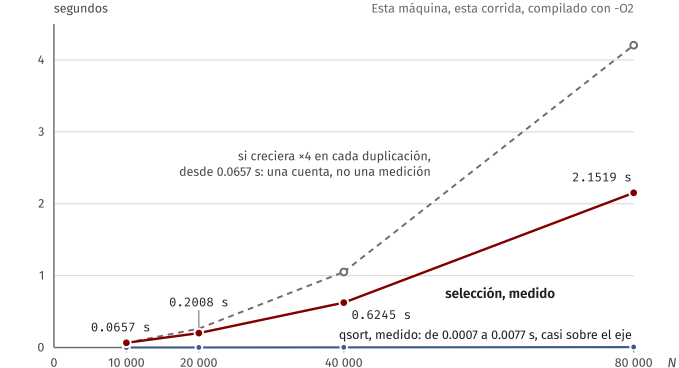
¿La selección crece como N^2?

esta máquina, esta corrida

```
$ echo 10000 20000 40000 80000 | build/fast/sort_times
      N    comparac. selección      qsort  ordenado
  10000     49995000    0.0657s    0.0007s  sí
  20000    199990000    0.2008s    0.0015s  sí
  40000    799980000    0.6245s    0.0032s  sí
  80000   3199960000    2.1519s    0.0077s  sí
```

La columna de comparaciones es exacta. Las dos de tiempo son mediciones.

Los tiempos, a escala



Al duplicar N, cuántas veces creció:

tiempo de selección	×3.06	×3.11	×3.45
comparaciones	×4.00	×4.00	×4.00

La línea punteada es una cuenta. Las otras dos son mediciones, sin arreglar.

Cuántas veces creció cada columna

N pasa de	Comparaciones	Selección	`qsort`
10 000 a 20 000	4.00	3.06	2.14
20 000 a 40 000	4.00	3.11	2.13
40 000 a 80 000	4.00	3.45	2.41

El tiempo no llegó a 4. La fórmula cuenta comparaciones, y aquí cada una costó de 1.3 a 0.7 ns: un tiempo es una medición. Se compara lo predicho con lo que salió, sin arreglar los números.

Idea clave

Los tiempos dependen de la máquina; la tendencia, no: selección crece bastante más que el doble, y `qsort` apenas más que el doble.

SECCIÓN 05

qsort

05



qsort: una caja negra

qsort_demo.c

```c
qsort(grades, N, sizeof(grades[0]), compare_ints);
```

la función de comparar

```c
int compare_ints(const void *first, const void *second)
    {
    int x = *(const int *) first;
    int y = *(const int *) second;
    if (x < y) {
        return -1;
    } else if (x > y) {
        return 1;
    }
    return 0;
}
```

- Recibe arreglo, cuántos, bytes de cada uno y función.
- Su forma la pide `qsort`. **Se copia como receta**.
- `const void *`: una dirección, sin decir de qué tipo.
- C no promete un algoritmo ni una velocidad.

No resten para comparar

errors/compare_subtract.c

```c
return x - y;
```

Parece ingenioso, pero la resta puede no caber en un `int`. Ordenando 2000000000, -2000000000 y 5:

terminal

```
$ gcc -Wall -Wextra -std=c17 -g -fsanitize=address,undefined errors/compare_subtract.c -o build/compare_subtract
$ build/compare_subtract
errors/compare_subtract.c:10:14: runtime error: signed integer overflow: 2000000000 - -2000000000 cannot be represented in type 'int'
 2000000000 -2000000000 5
```

No quedó ordenado, en esta corrida. Es comportamiento indefinido: con otros datos o con otro compilador puede salir otra cosa.

SECCIÓN 06

Cuándo ordenar

06

¿Ordenar, o buscar de una vez?

Hay que buscar k veces en una lista desordenada.

Plan A

k búsquedas lineales. No se prepara nada.

Plan B

Ordenar una vez, y después k búsquedas binarias.

¿Ordenar, o buscar de una vez?

Hay que buscar k veces en una lista desordenada.

Plan A

k búsquedas lineales. No se prepara nada.

Plan B

Ordenar una vez, y después k búsquedas binarias.

Con $N = 100\,000$ y valores que no están, en comparaciones:

- Plan A: $k \times 100\,000$.
- Plan B: como mucho $k \times 17$, más lo que cueste ordenar.
- Si se ordena por selección, son casi $5\,000\,000\,000$ más: gana desde unas $50\,000$ búsquedas.
- Si se ordena con `qsort`, mucho antes.

La medición

esta máquina, esta corrida

```
$ echo 100000 1 10 100 1000 10000 100000 | build/fast/sort_then_search
      k          lineal      binaria   t lineal  t ordenar
      1          100000           17    0.0000s    0.0086s
     10         1000000          167    0.0004s    0.0085s
    100        10000000         1668    0.0036s    0.0085s
   1000       100000000        16688    0.0361s    0.0084s
  10000      1000000000       166891    0.3807s    0.0091s
 100000     10000000000      1668930    3.6900s    0.0134s
```

- `t ordenar` incluye el `qsort`. Las columnas de comparaciones, no.
- **Una** búsqueda: gana el plan A. Ese 0.0000 no significa que no tardó nada.
- Entre 100 y 1000 búsquedas, en esta corrida, empieza a ganar el plan B.
- **100 000** búsquedas: 3.7 s contra 0.0134 s, unas 275 veces menos.

Ordenar es una inversión: se paga una vez y se recupera en cada búsqueda.

SECCIÓN 07

Recursión

07



Una función que se llama a sí misma

pseudocódigo

```
algoritmo factorial(n)
    si n ≤ 1 entonces
        devolver 1
    fin si
    devolver n * factorial(n - 1)
fin algoritmo
```

factorial.c

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

- **Caso base**: tan pequeño que se responde sin más llamadas.
- **Caso recursivo**: reduce el problema y se llama a sí misma.
- `factorial(20)` cabe en 64 bits; `factorial(21)` ya no.

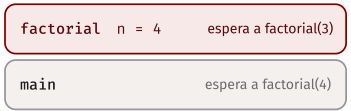
Es una forma distinta de repetir. Se ve a fondo en Introducción a la Programación.

Baja, y luego vuelve

La pila: cada llamada agrega un marco arriba

build/factorial

```
entra factorial(4)
```

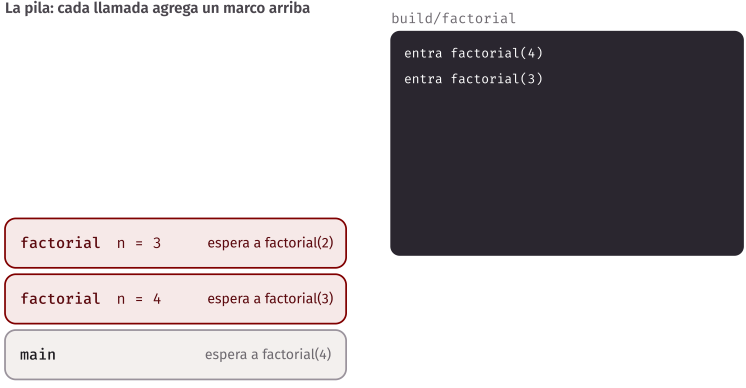


main llama a factorial(4). La llamada agrega un marco, con su propio `n`.

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve

La pila: cada llamada agrega un marco arriba

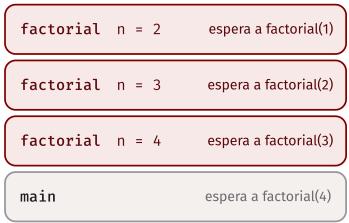


factorial(4) todavía no puede multiplicar: primero llama a factorial(3).

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve

La pila: cada llamada agrega un marco arriba



build/factorial

```
entra factorial(4)
entra factorial(3)
entra factorial(2)
```

factorial(3) llama a factorial(2). Ninguno ha calculado nada aún.

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve

La pila: cada llamada agrega un marco arriba



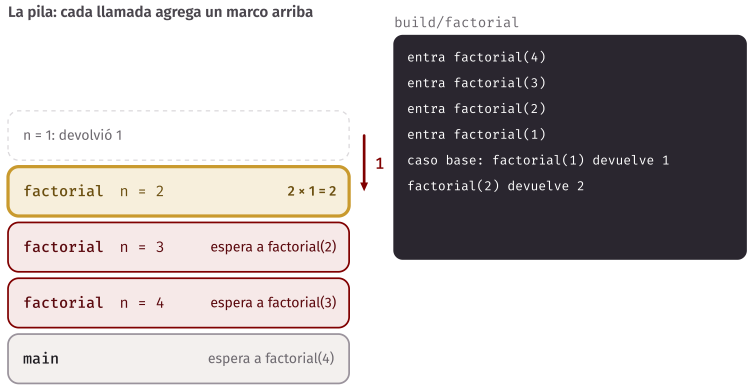
build/factorial

```
entra factorial(4)
entra factorial(3)
entra factorial(2)
entra factorial(1)
caso base: factorial(1) devuelve 1
```

factorial(1) es el caso base: responde sin más llamadas, y devuelve 1.

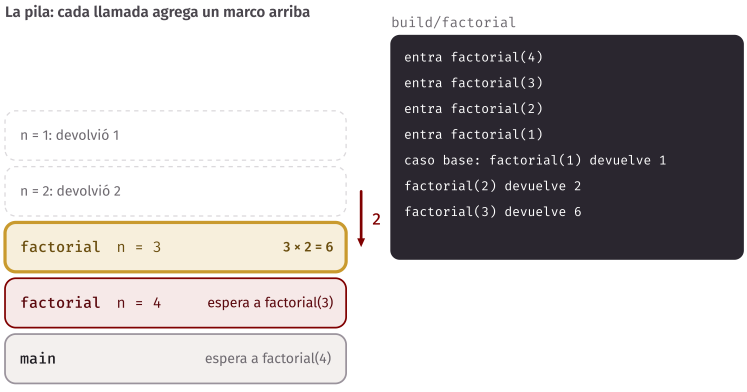
La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve



La pila de la semana 7: cada llamada tiene **su propio** `n`.

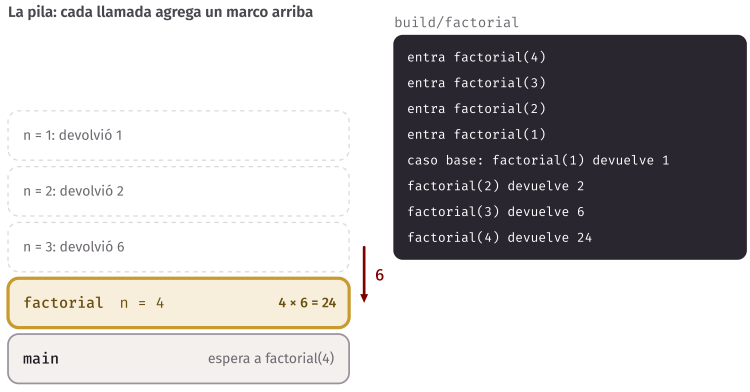
Baja, y luego vuelve



El marco de factorial(2) se retira y entrega 2. factorial(3) multiplica.

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve



El marco de factorial(3) se retira y entrega 6. factorial(4) multiplica.

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Baja, y luego vuelve

La pila: cada llamada agrega un marco arriba



build/factorial

```
entra factorial(4)
entra factorial(3)
entra factorial(2)
entra factorial(1)
caso base: factorial(1) devuelve 1
factorial(2) devuelve 2
factorial(3) devuelve 6
factorial(4) devuelve 24
```

El último marco se retira, y main recibe el 24.

La pila de la semana 7: cada llamada tiene **su propio** `n`.

Sin caso base, se acaba la pila

terminal

```
$ gcc -Wall -Wextra -std=c17 errors/no_base_case.c -o build/no_base_case
errors/no_base_case.c:4:11: warning: infinite recursion detected [-Winfinite-recursion]
errors/no_base_case.c:5:16: note: recursive call
```

esta máquina, esta corrida

```
$ build/no_base_case; echo "estado: $?"
bash: line 1: 15389 Segmentation fault      build/no_base_case
estado: 139
```

- Cada llamada agrega un marco, y la pila tiene un límite: **stack overflow**.
- Tener caso base no basta si es demasiado profunda: `errors/deep_recursion.c` funciona con 1000, y no con 10 millones.

Laboratorio

1. `lab/searching.c`: `linear_search` y `binary_search`, con pruebas `assert`. Los bordes: el primero, el último, un valor que no está, $n = 0$.
2. También ahí: `linear_count` y `binary_count`, y una línea por tamaño, con 1 000, 10 000 y 100 000. **Predigan los números antes de correrlo.**
3. `lab/sorting.c`: `selection_sort`, `is_sorted`, `seconds_selection` y `seconds_qsort`.

Cómo comprobar

`make test-lab` usa un reloj de prueba, con un valor conocido. Para medir de verdad: `make measure`, con `-O2` y sin sanitizers. Escriban el factor que obtuvieron en cada fila.

Para la próxima clase

- Hoy, en el laboratorio: `searching.c` y `sorting.c`.
- Leer los apuntes de la semana 14: internet y una mirada a Python.
- Práctica 4 la próxima semana: `struct`, archivos, búsqueda y eficiencia.

Preguntas

¿Cuántas comparaciones hace como máximo la búsqueda binaria en 1 000 000 de elementos?
Un algoritmo tarda 2 s con $N = 1000$. ¿Cuánto con $N = 2000$, si crece como N^2?
Hay que buscar una sola vez en una lista desordenada. ¿Conviene ordenar primero?