BIG O - ANÁLISIS COMPLEJIDAD TEMPORAL Y ESPACIAL

BIG O - ANÁLISIS DE COMPLEJIDAD

Daniel Blanco Calviño

¿QUÉ ES EL BIG O?

- **Métrica** que se utiliza en Ingeniería del Software para **medir la eficiencia de nuestros algoritmos.**

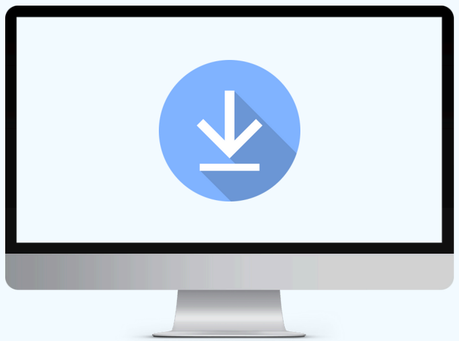
- Es básico para determinar cómo se comportarán nuestros algoritmos bajo una carga de trabajo alta.

Dominar **este tema es esencial** para las entrevistas y para ser un gran desarrollador.

¿QUÉ ES EL BIG O?

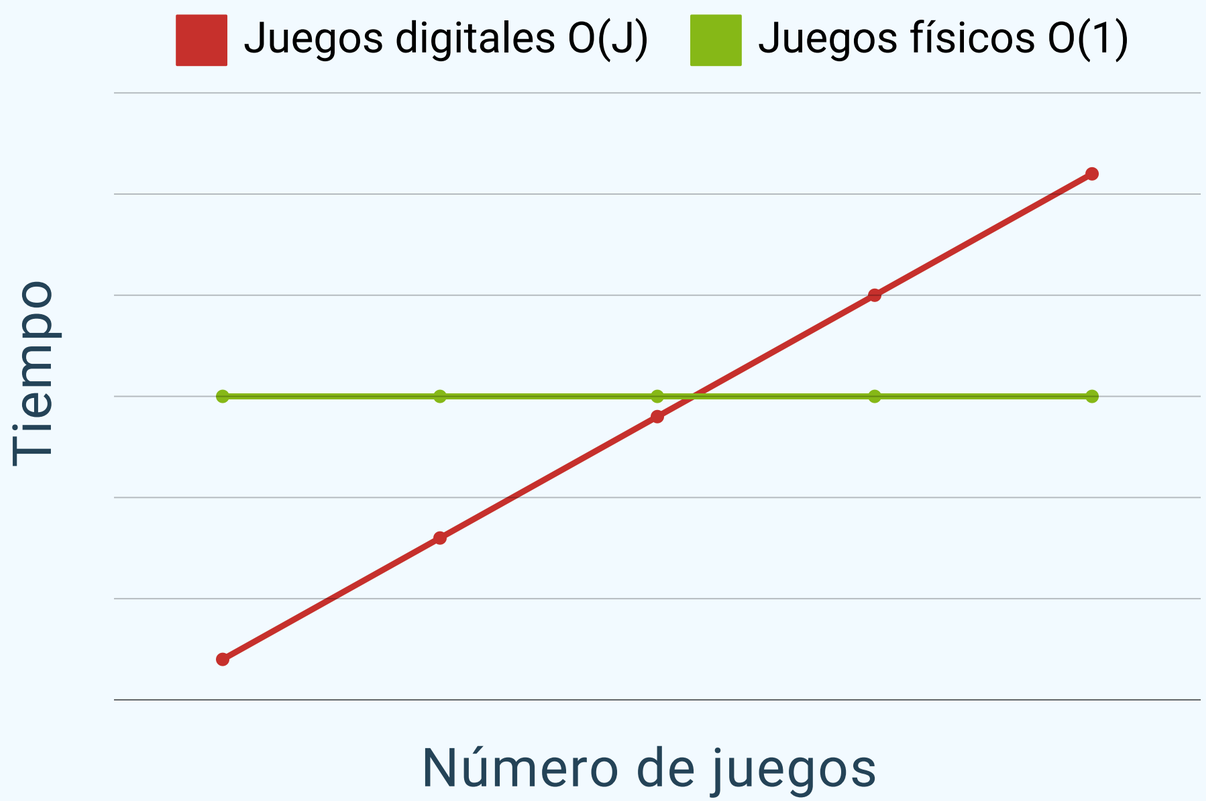
- **Objetivo:** Tener toda la colección de juegos de PS5.

Juego digital

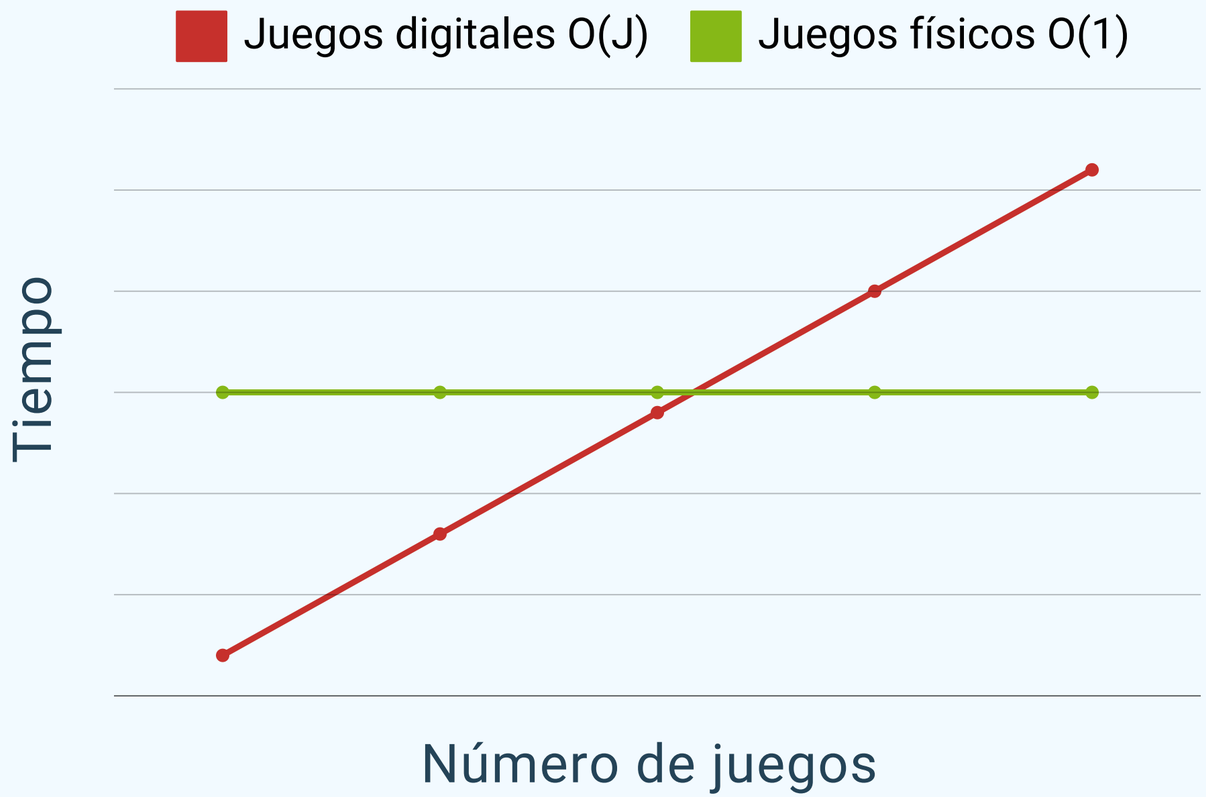


Juego físico

TIEMPO DE EJECUCIÓN ASINTÓTICO



TIEMPO DE EJECUCIÓN ASINTÓTICO



- O(log N), O(N log N), O(N), O(N^2) O(2^N).

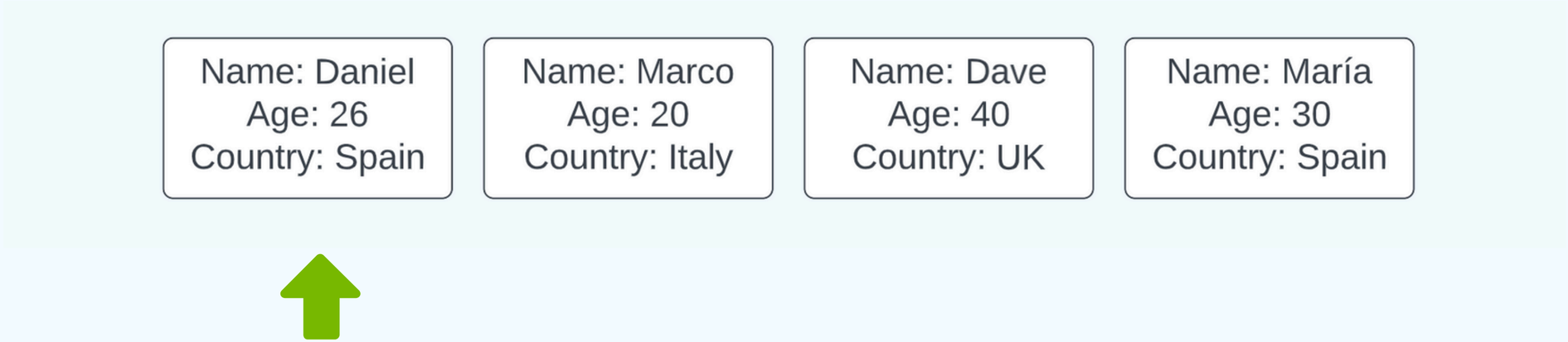
COMPLEJIDAD ESPACIAL

- El espacio utilizado es importante al analizar un algoritmo.
- **Compromisos** entre complejidad temporal y espacial.

EJEMPLO COMPLEJIDAD ESPACIAL

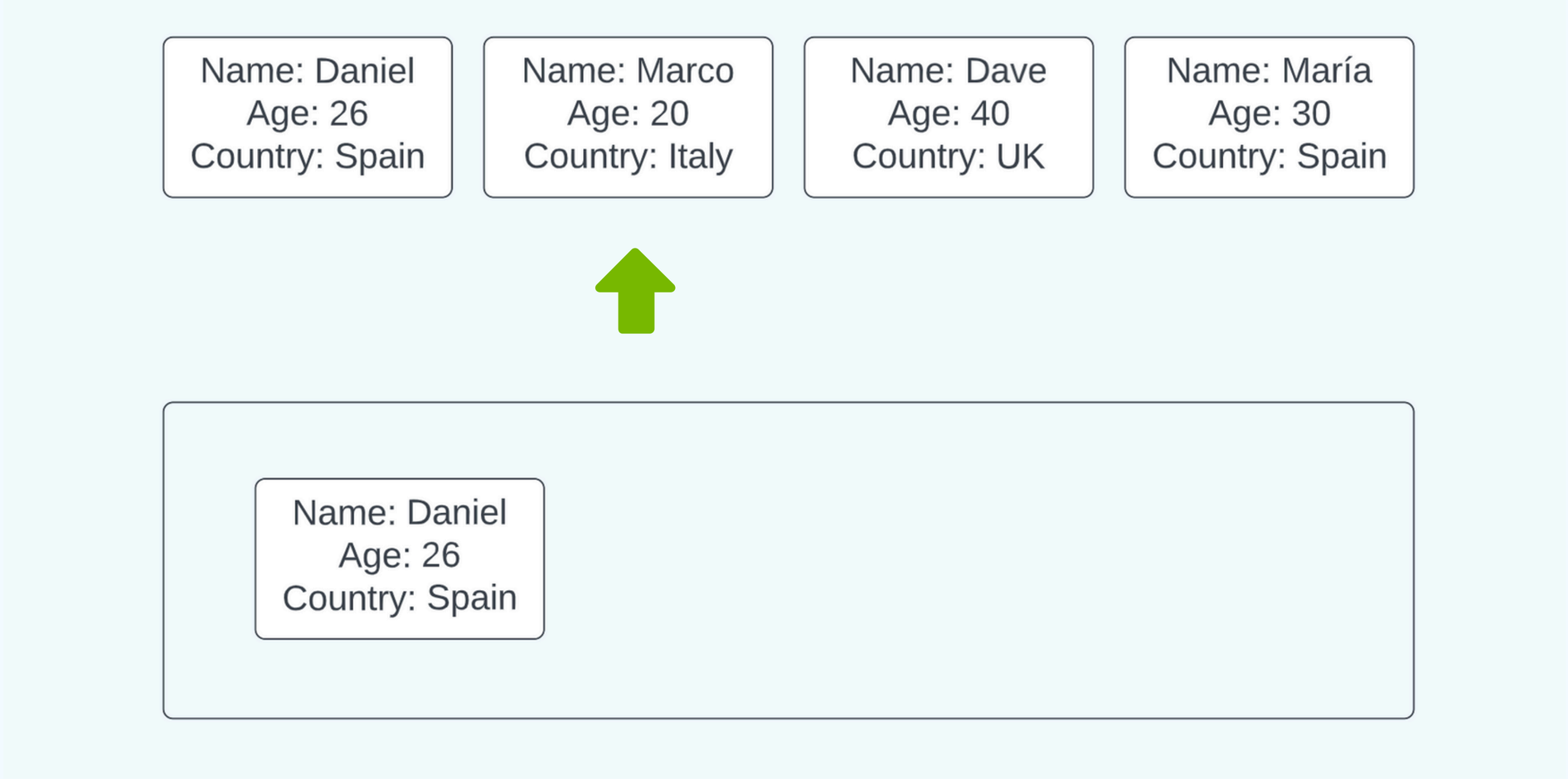
EJEMPLO COMPLEJIDAD ESPACIAL

Input: Spain



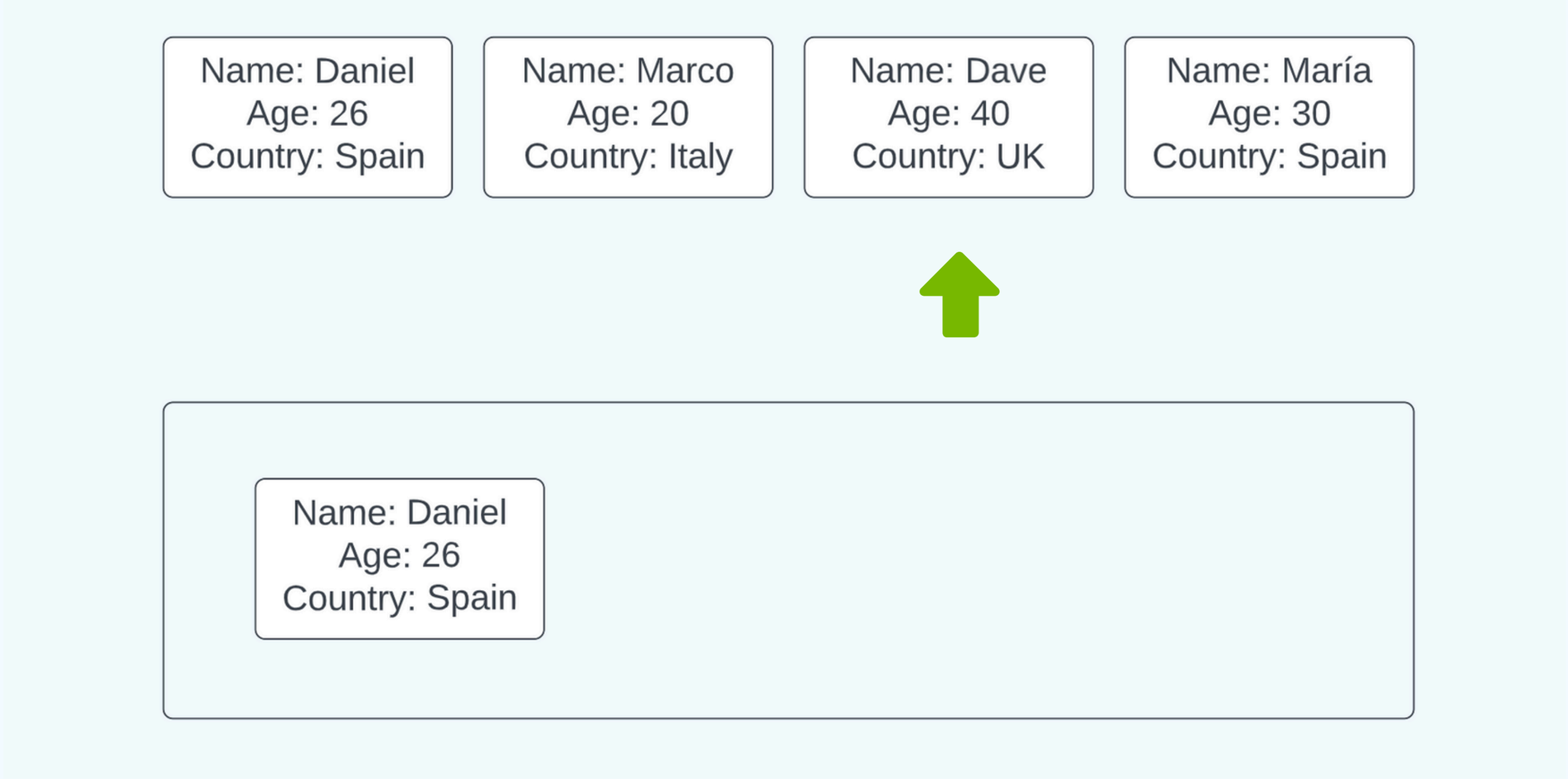
EJEMPLO COMPLEJIDAD ESPACIAL

Input: Spain



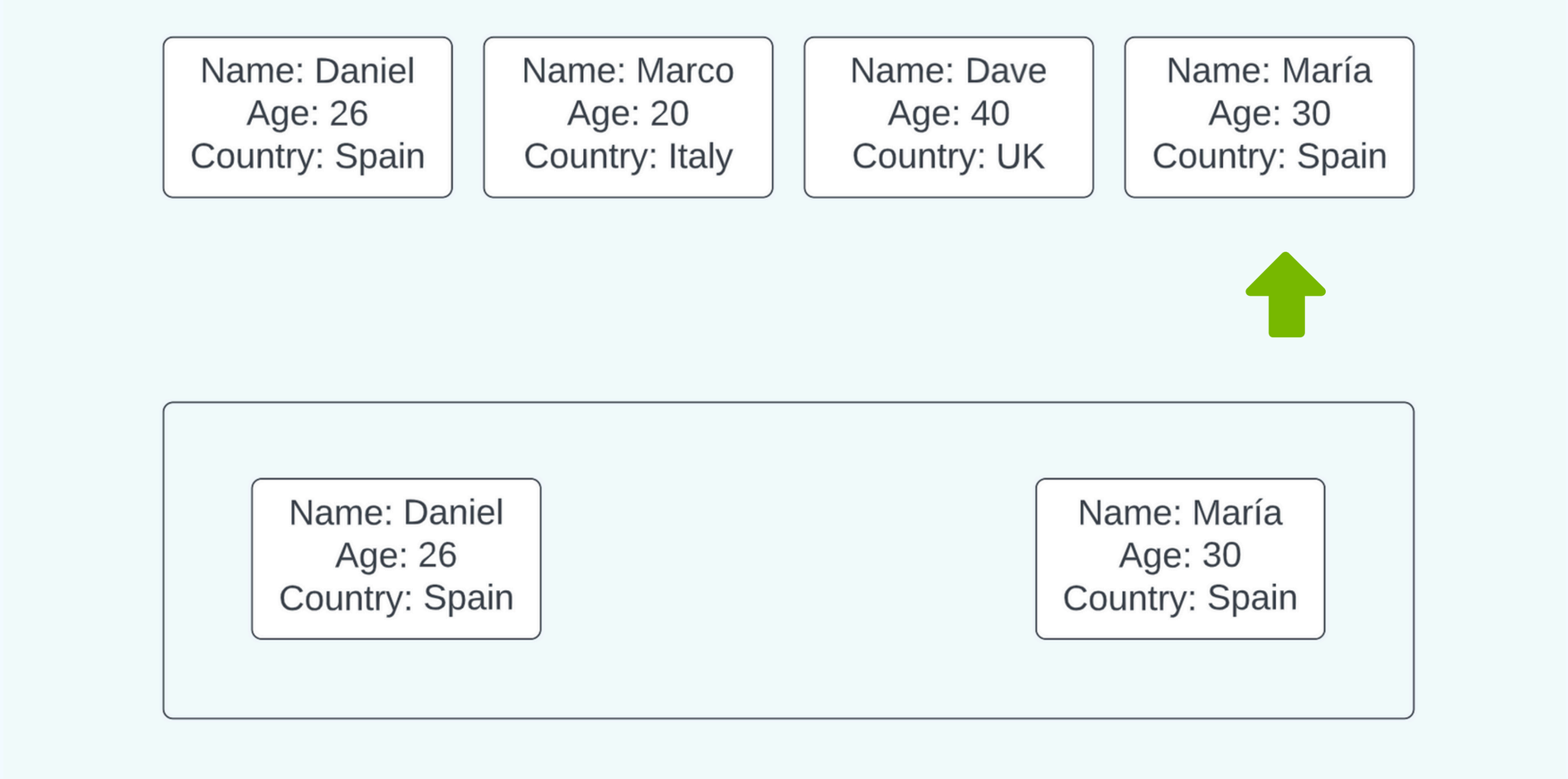
EJEMPLO COMPLEJIDAD ESPACIAL

Input: Spain



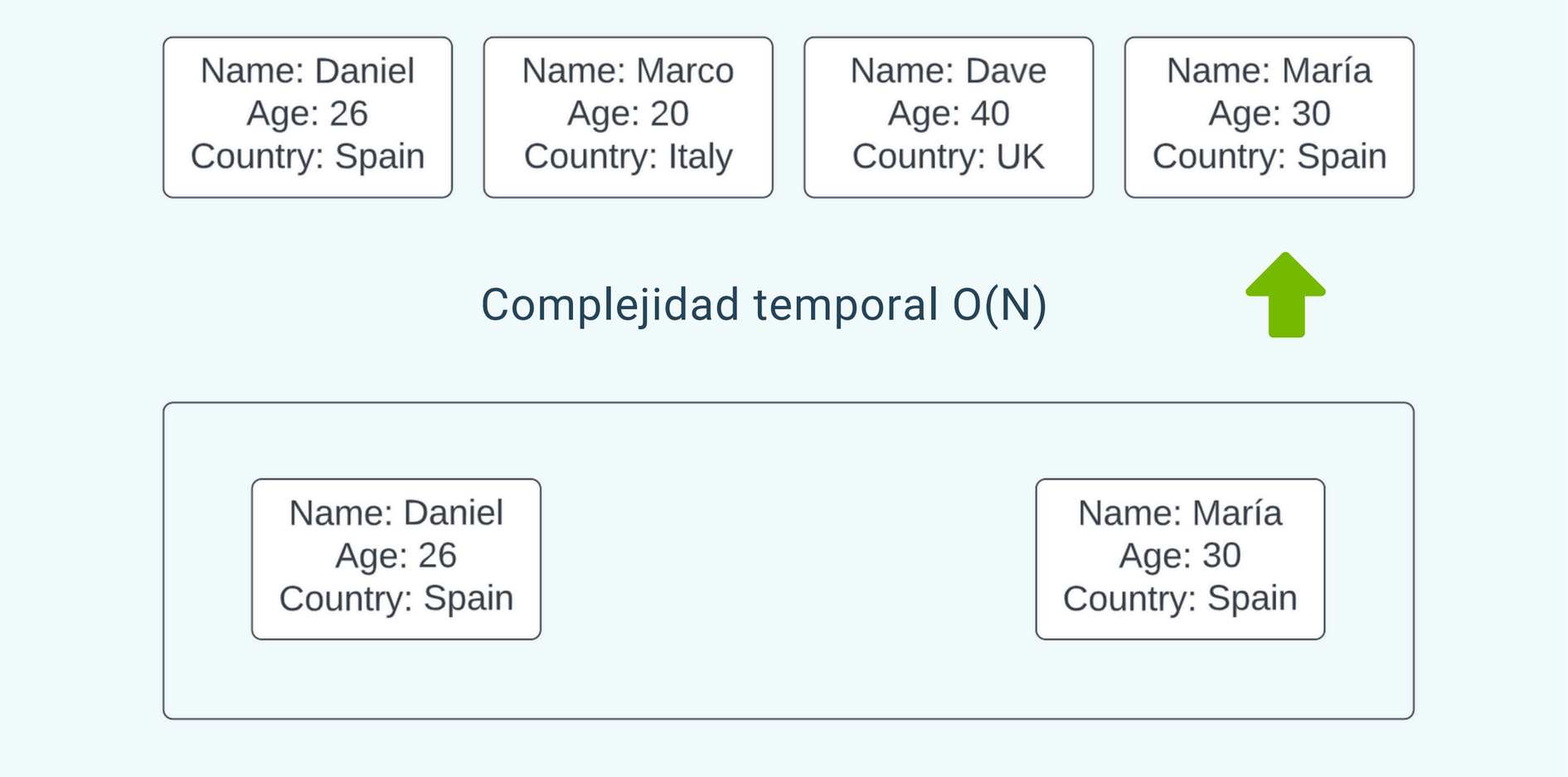
EJEMPLO COMPLEJIDAD ESPACIAL

Input: Spain



EJEMPLO COMPLEJIDAD ESPACIAL

Input: Spain



EJEMPLO COMPLEJIDAD ESPACIAL

EJEMPLO COMPLEJIDAD ESPACIAL

- **Complejidad temporal O(1)** gracias al Hash Map.
- **Complejidad espacial O(N)** debido a la memoria adicional.
- **Compromiso** entre complejidad temporal y espacial.

¡Solo se cuenta la memoria adicional!

CONSTANTES

- Big O describe el **ritmo con el que crece la complejidad** de un algoritmo.
  - O(N) puede ser más rápido que O(1) para valores bajos.

```java
int sum = 0;
int product = 1;

for (int val : array) {
  sum += val;
  product *= val;
}
```

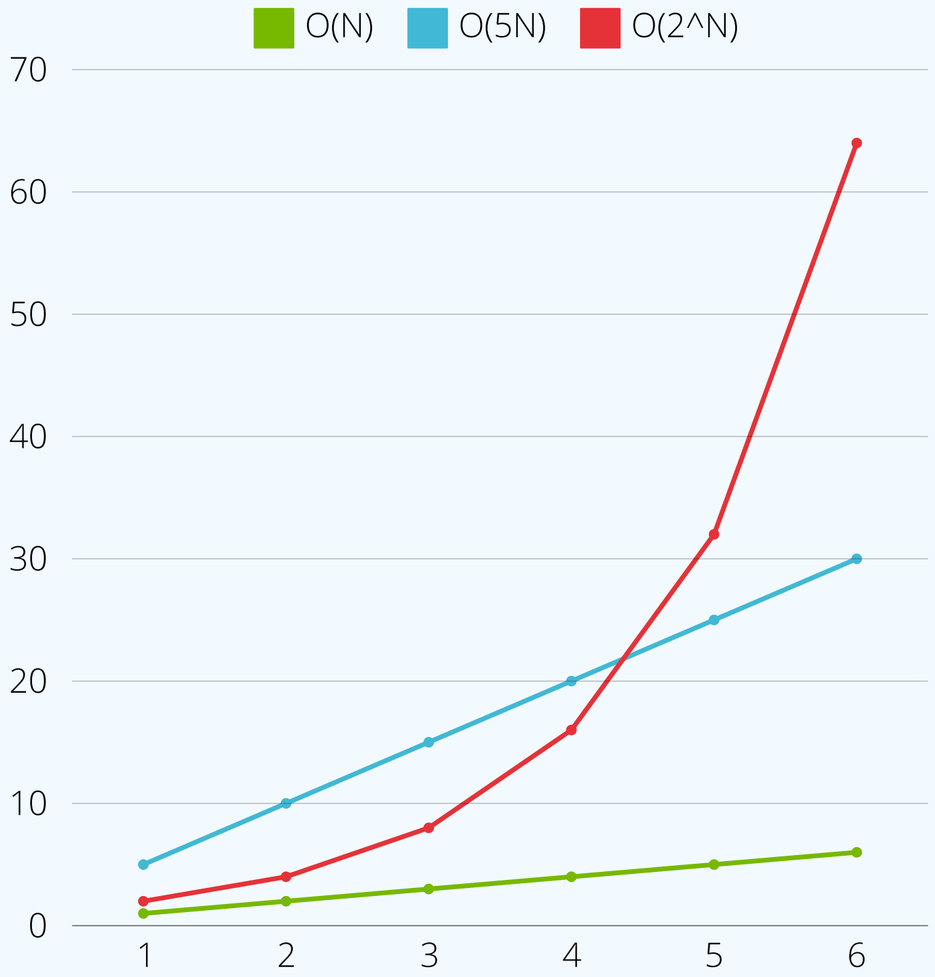
O(N)

```java
int sum = 0;
int product = 1;

for (int val : array) {
  sum += val;
}

for (int val : array) {
  product *= val;
}
```

¿O(2N)?

El Nº de instrucciones no determina la complejidad. Se deben **eliminar las constantes.**

CONSTANTES



TÉRMINOS NO DOMINANTES

¿O(N + N^2)?

O(N)

O(N^2)

```
1  int sum = 0;
2  int result = 0;
3
4  for (int val : array) {
5    sum += val;
6  }
7
8  for (int x : array) {
9    for (int y : array) {
10     result += x * y - sum;
11   }
12 }
```

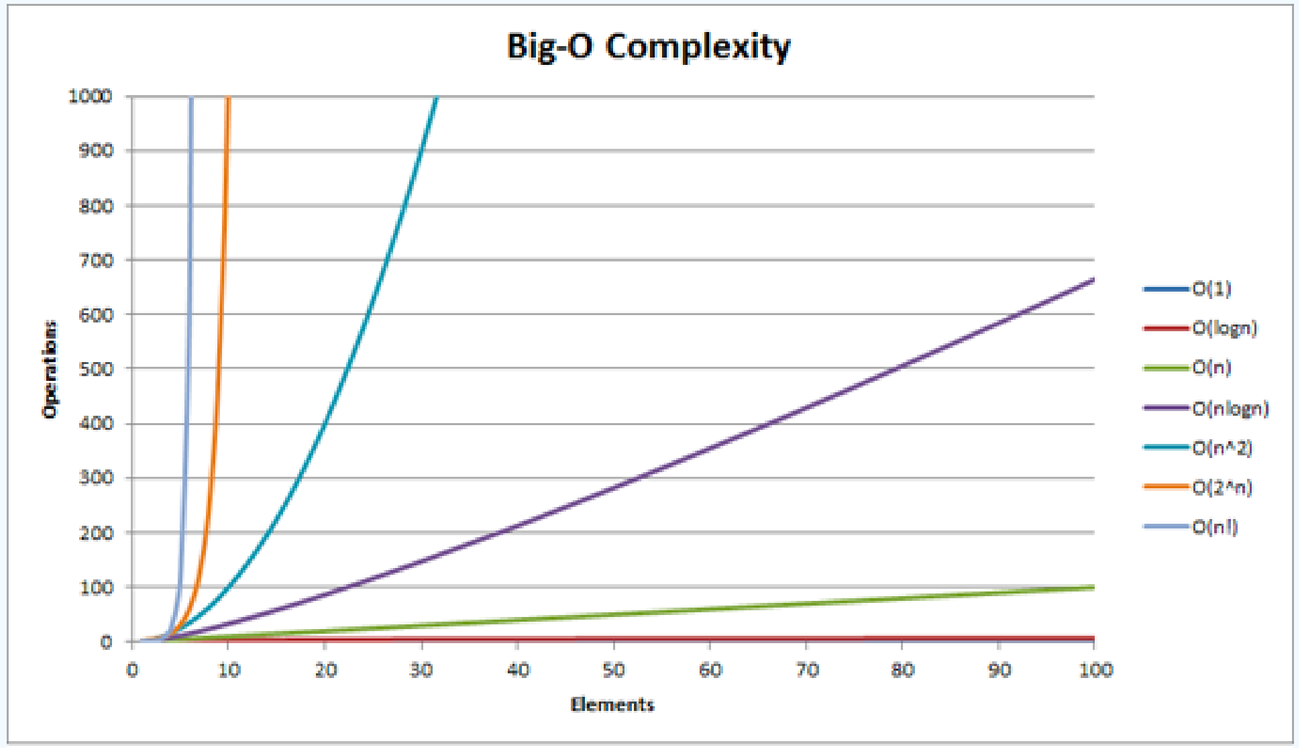
O(2N^2) == O(N^2 + N^2) == O(N^2). Lo mismo para el caso anterior con N. Se deben **eliminar los términos no dominantes.**

"N" DIFERENTES

```java
int sum = 0;

for (int val : arrayA) {
  sum += val;
}

for (int val : arrayB) {
  sum += val;
}
```

A primera vista parece complejidad O(N), pero los arrays **pueden tener tamaños diferentes**, por lo que se deben tener en cuenta para la complejidad total: **O(A + B).**

BIG O MÁS COMUNES



O(1) - COMPLEJIDAD CONSTANTE

```
1   int max(int a, int b) {
2     if (a >= b) return a;
3     return b;
4   }
```

O(LOG N) - COMPLEJIDAD LOGARÍTMICA

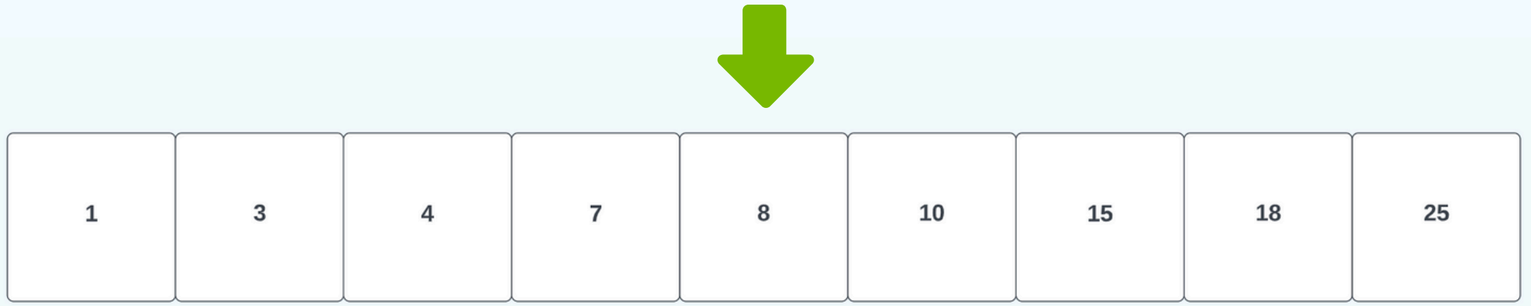
- **Búsqueda binaria**. Búsqueda de un elemento en un conjunto de datos ordenado.
  - Tomamos el elemento central.
  - Si el target == elemento , hemos finalizado.
  - Si el target < elemento, nos quedamos con el subarray izquierdo.
  - Si el target > elemento, nos quedamos con el subarray derecho.
  - Repetimos hasta encontrar el elemento o concluir que no se encuentra en el array.

O(LOG N) - COMPLEJIDAD LOGARÍTMICA

- Target: 15

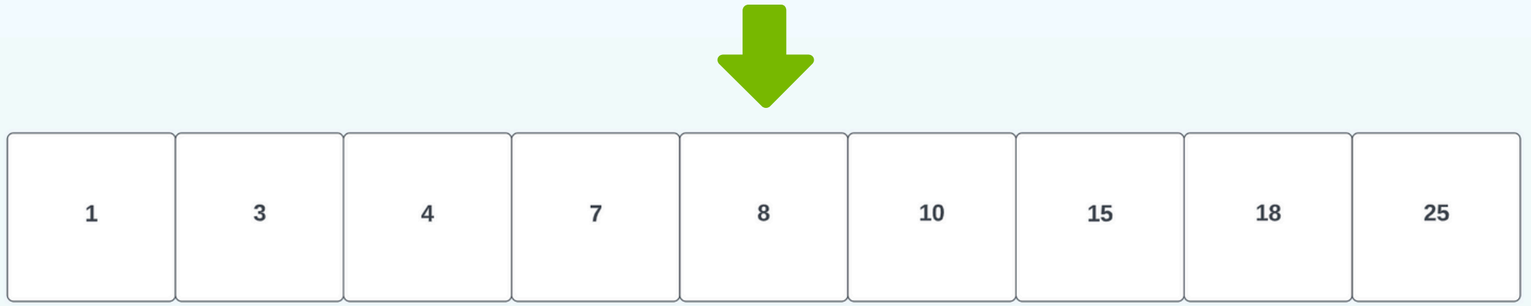
O(LOG N) - COMPLEJIDAD LOGARÍTMICA

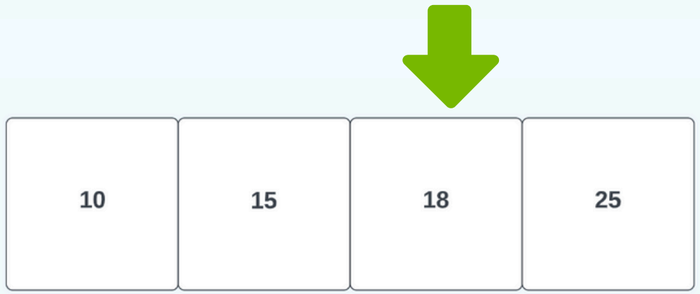
- Target: 15



O(LOG N) - COMPLEJIDAD LOGARÍTMICA

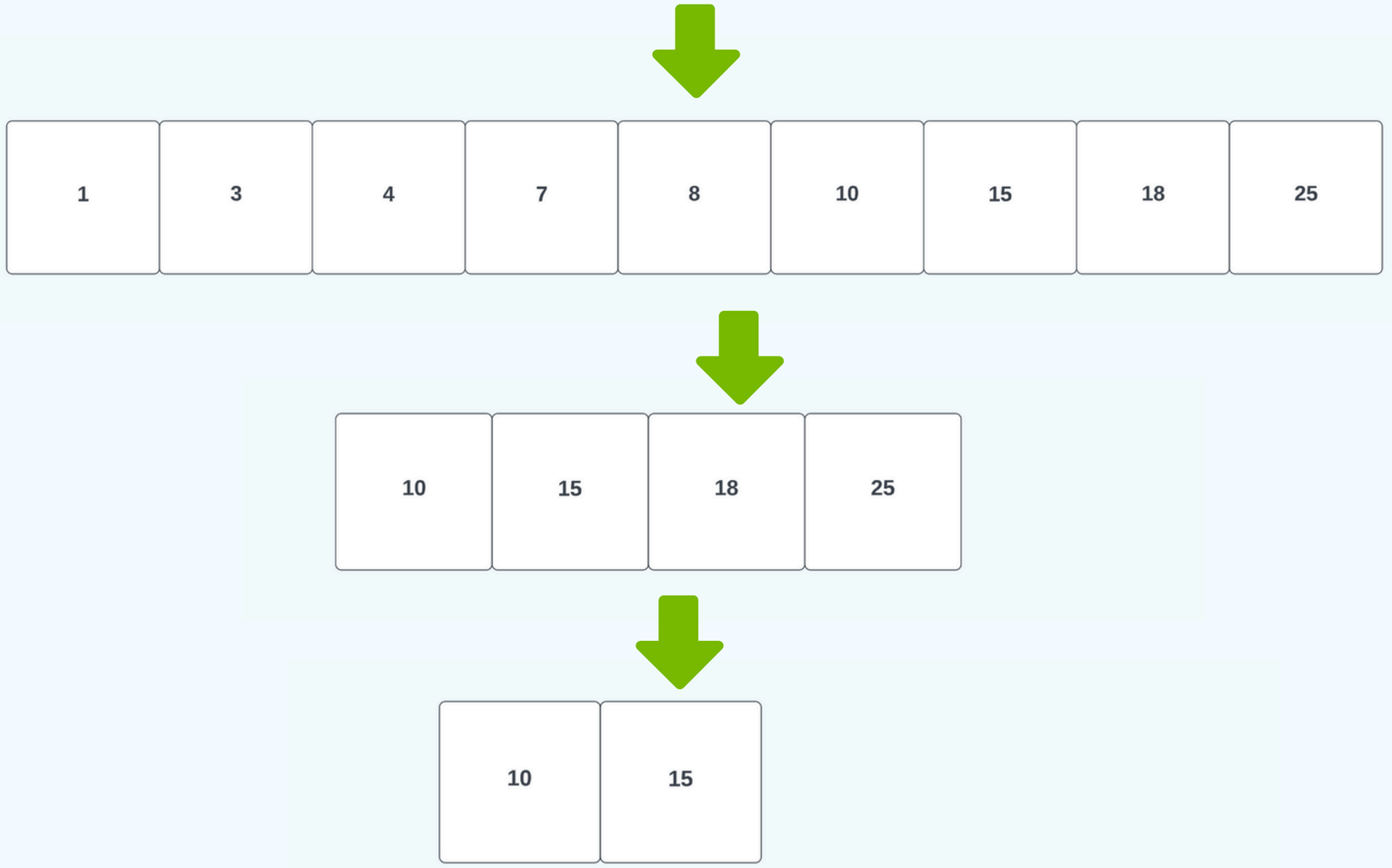
- Target: 15





O(LOG N) - COMPLEJIDAD LOGARÍTMICA

- Target: 15



O(N) - COMPLEJIDAD LINEAL

```java
Employee linearSearch(Employee[] employees, String name) {
  for (Employee e : employees) {
    if (name.equals(e.getName())) return e;
  }

  return null;
}
```

O(N LOG N)

- Los mejores **algoritmos de ordenación** que aplican divide y vencerás:
  - Merge Sort
  - Heap Sort
  - Quick Sort

- N veces búsqueda binaria.

O(N^2)

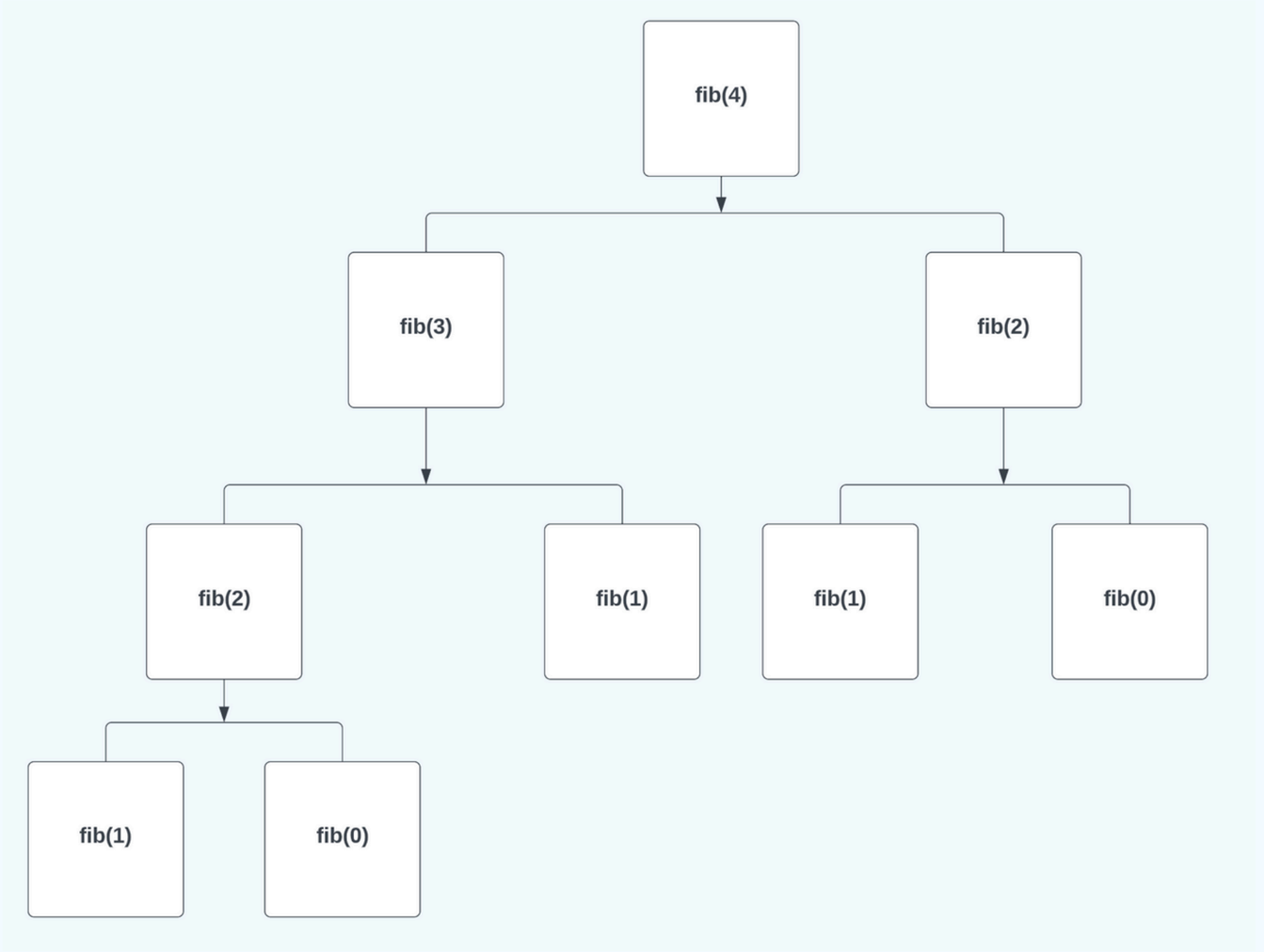
```
1  void printPairs(int[] array) {
2    for (int x : array) {
3      for (int y : array) {
4        System.out.println(x + " " + y);
5      }
6    }
7  }
```

O(2^N)

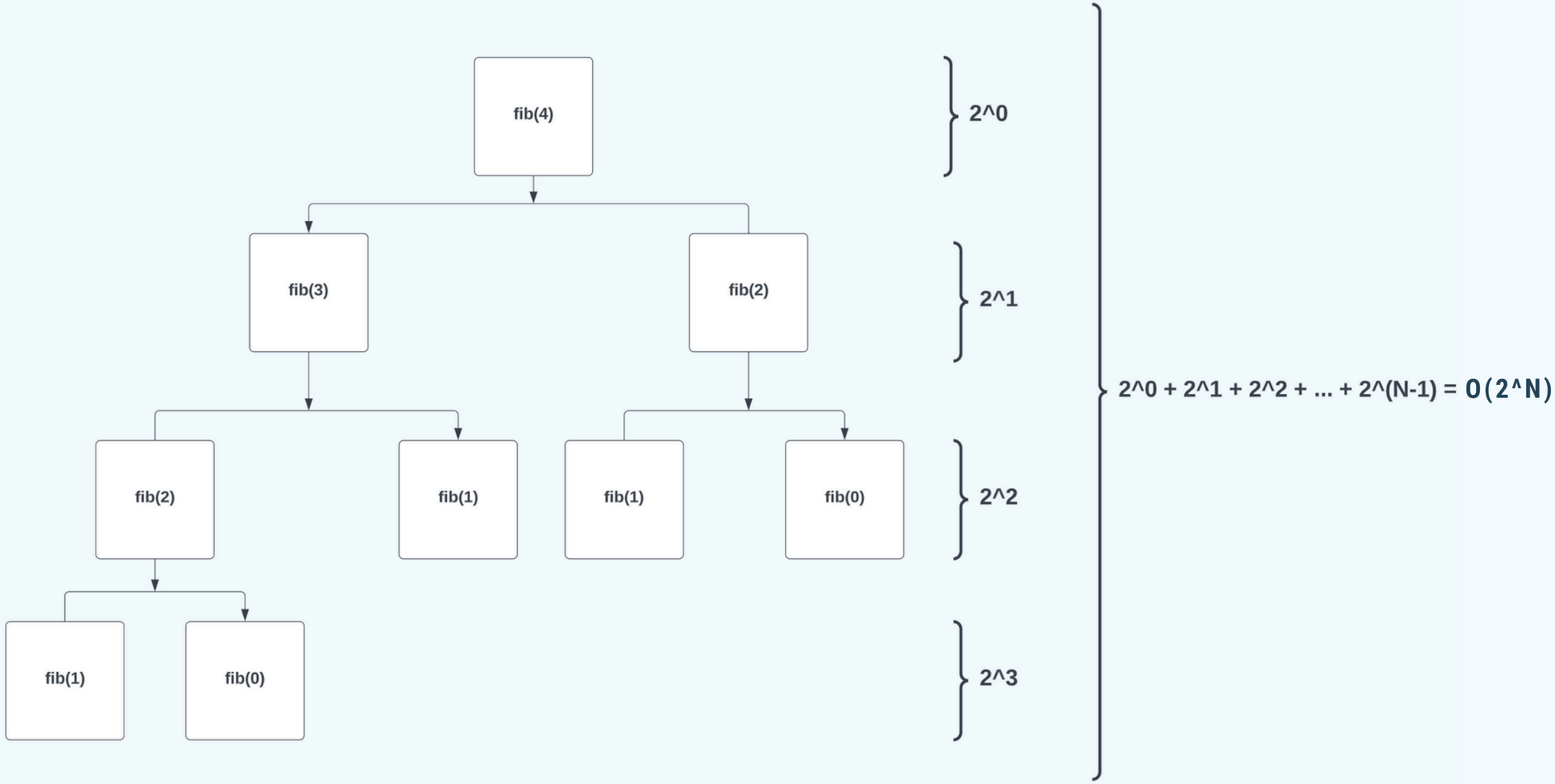
- Secuencia fibonacci:
  - fibonacci(0) = 0 ; fibonacci(1) = 1 ; fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2)
- Ejemplo **fibonacci(3) = fibonacci(2) + fibonacci(1) = 1 + 1 = 2.**

```
1  int fibonacci(int n) {
2    if (n < 2) return n;
3    return fibonacci(n - 1) + fibonacci(n - 2);
4  }
```

O(2^N)



O(2^N)


fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
2^0
2^1
2^2
2^3
2^0 + 2^1 + 2^2 + ... + 2^(N-1) = O(2^N)

O(N!)

- **N! = N * (N -1) * (N - 2) * ... * 3 * 2.**
  - 3! = 3 * 2 = 6
  - 6! = 6 * 5 * 4 * 3 * 2 = 720

```
1  void nFactorialExample(int n) {
2    for (int i = 0; i < n; i++) {
3      nFactorialExample(n - 1);
4    }
5  }
```

CONCLUSIONES TEORÍA

- **No te frustes** si no lo comprendes del todo.
  - Veremos más ejercicios específicos de Big O.
  - **Analizaremos la complejidad en todos los ejercicios** del curso.
- Concepto muy importante que te pedirán en las entrevistas de este tipo.

CONCLUSIONES TEORÍA

- **No te frustes** si no lo comprendes del todo.
  - Veremos más ejercicios específicos de Big O.
  - **Analizaremos la complejidad en todos los ejercicios** del curso.
- Concepto muy importante que te pedirán en las entrevistas de este tipo.

- Big O
  - Mide la **complejidad temporal y espacial** de nuestros algoritmos **en los extremos**.
  - Un algoritmo de mayor complejidad puede ser más rápido para un N pequeño.
    - No nos importa. **Para N pequeños todos los algoritmos son rápidos.**