Sorting, greedy y búsqueda binaria

Blaz Korecic

Sorting (ordenamiento)

Sorting

Para ordenar un vector podemos usar la función `std::sort`

Para ordenar un vector podemos usar la función `std::sort`

```
vector<int> a = {5, 2, 6, 2, 1, 8};
sort(a.begin(), a.end());
// a queda como {1, 2, 2, 5, 6, 8}
```

Para ordenar un vector podemos usar la función `std::sort`

```
vector<int> a = {5, 2, 6, 2, 1, 8};
sort(a.begin(), a.end());
// a queda como {1, 2, 2, 5, 6, 8}
```

Complejidad temporal $O(n \log n)$ si $|a| = n$

Para ordenar un vector podemos usar la función `std::sort`

```
vector<int> a = {5, 2, 6, 2, 1, 8};
sort(a.begin(), a.end());
// a queda como {1, 2, 2, 5, 6, 8}
```

Complejidad temporal $O(n \log n)$ si $|a| = n$

¿Qué son `a.begin()` y `a.end()`?

Función de comparación

Podemos cambiar la *función de comparación* para ordenar de la forma que queramos. Por ejemplo, para ordenar de mayor a menor:

```
bool es_mayor(int a, int b) {
  // La función de comparación debe retornar true
  // cuando "a" va antes que "b" en nuestro ordenamiento
  return a > b;
}
```

```
sort(a.begin(), a.end(), es_mayor);
```

Función de comparación

La librería estándar trae `greater<T>` que funciona como `es_mayor`:

```
sort(a.begin(), a.end(), greater<int>());
```

Función de comparación

La librería estándar trae `greater<T>` que funciona como `es_mayor`:

```
sort(a.begin(), a.end(), greater<int>());
```

También podemos usar funciones anónimas o *lambdas*:

```
sort(a.begin(), a.end(), [](int a, int b) {
  return a > b;
});
```

Algoritmos greedy

Programando eventos

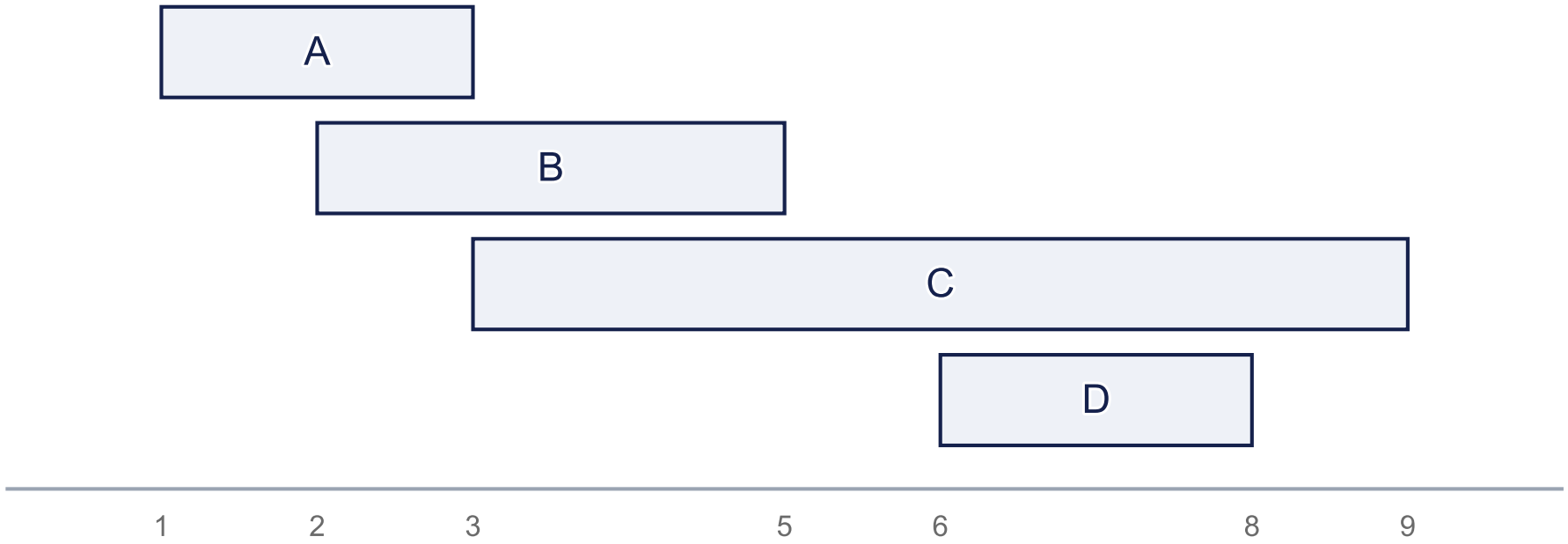
Dadas n películas en un cine con sus tiempos de inicio y fin, encuentra un horario que permita ver la **mayor cantidad de películas**:

- Solo se puede ver una a la vez.
- No se puede ver una película parcialmente.

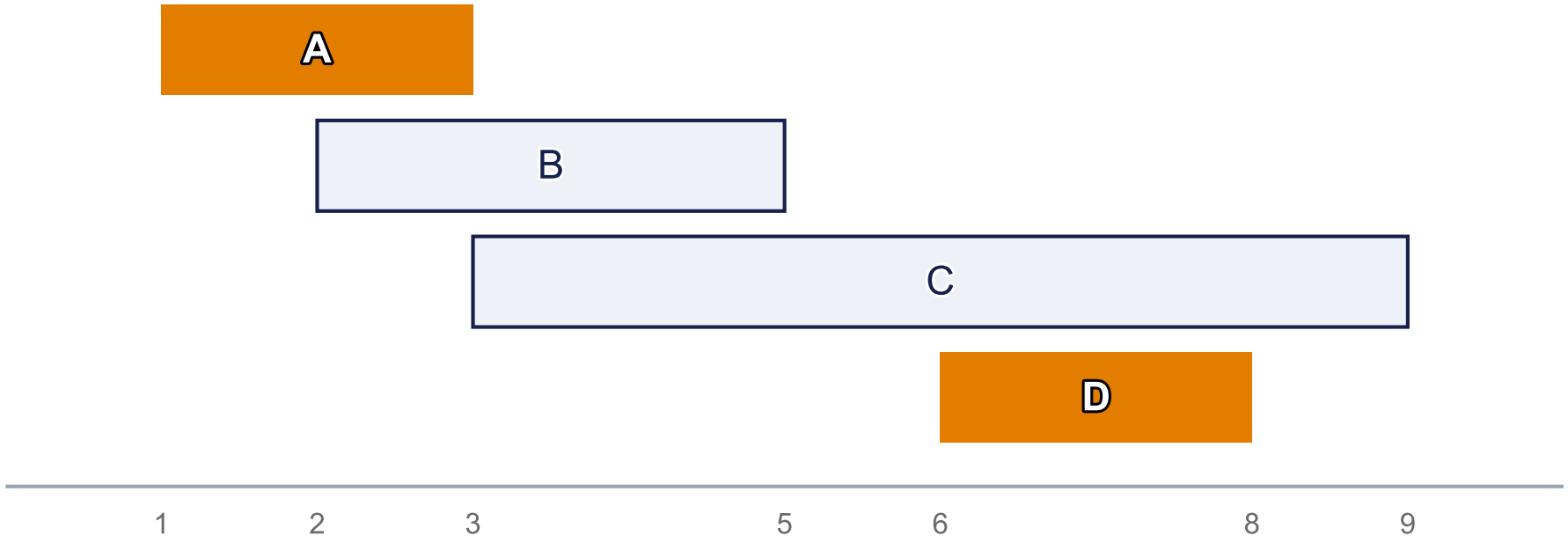
película	tiempo inicio	tiempo fin
A	1	3
B	2	5
C	3	9
D	6	8

Programando eventos

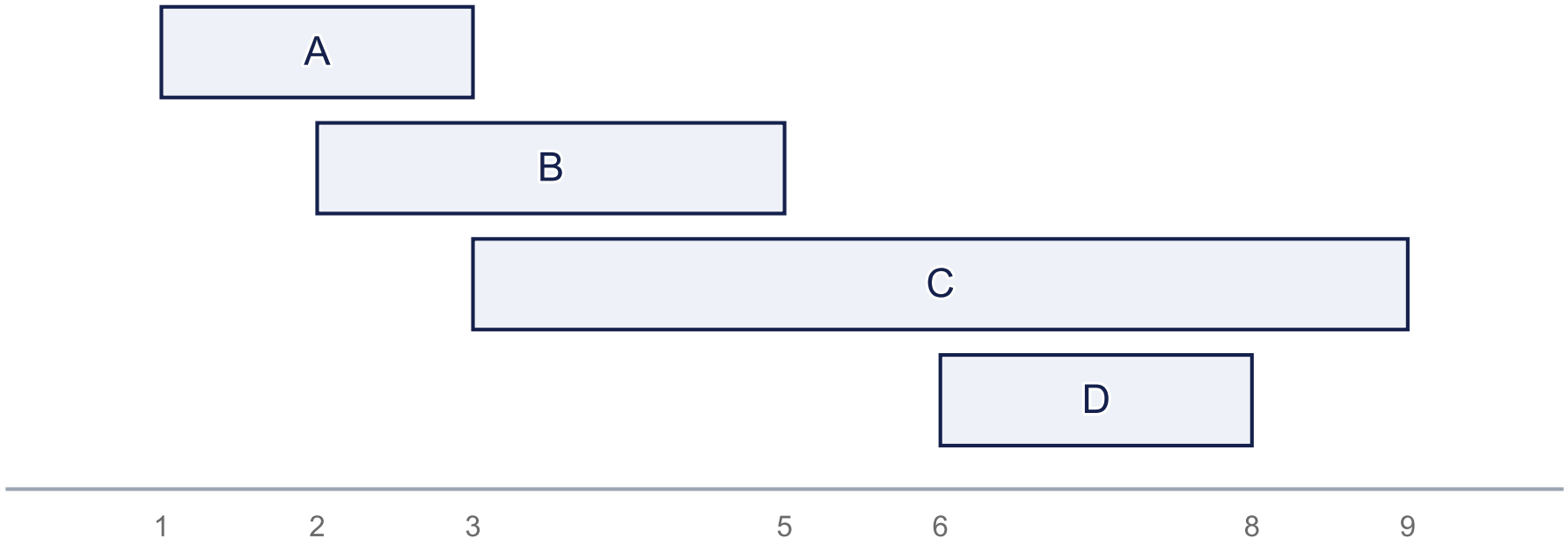
¿Cuántas películas podemos ver como máximo?



Solución al ejemplo: **2 películas** (por ejemplo A y D).

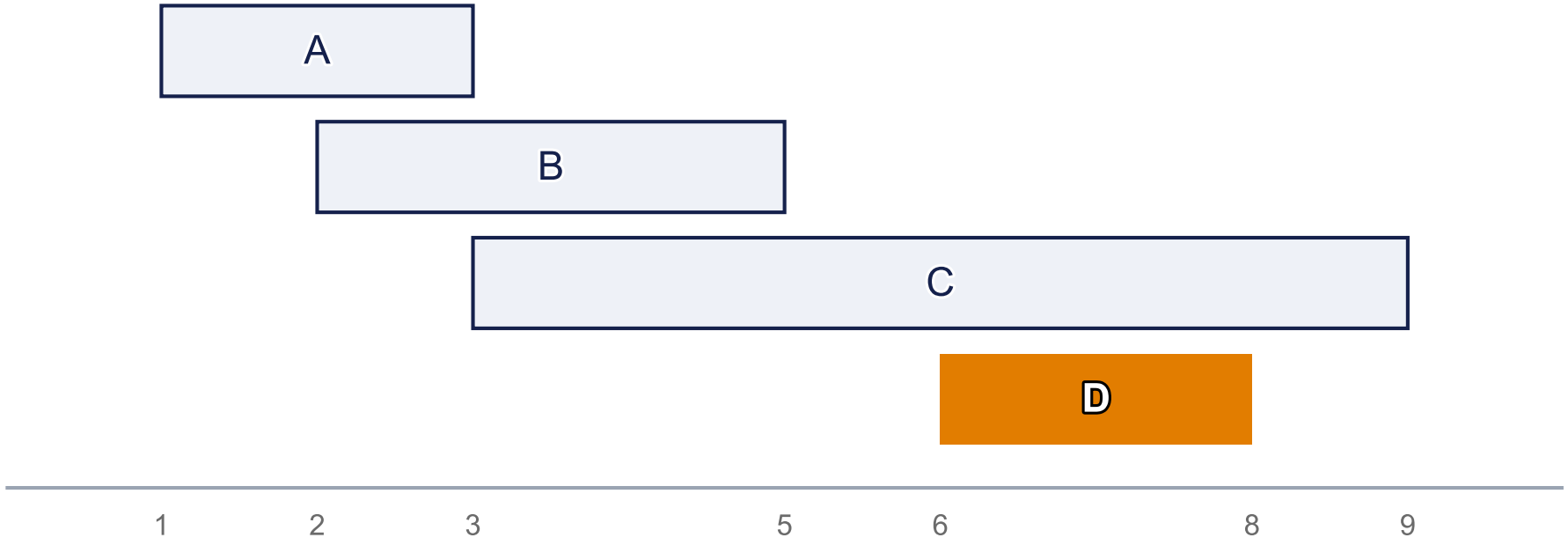


Idea greedy 1: los escogemos del **más corto al más largo**.



Programando eventos

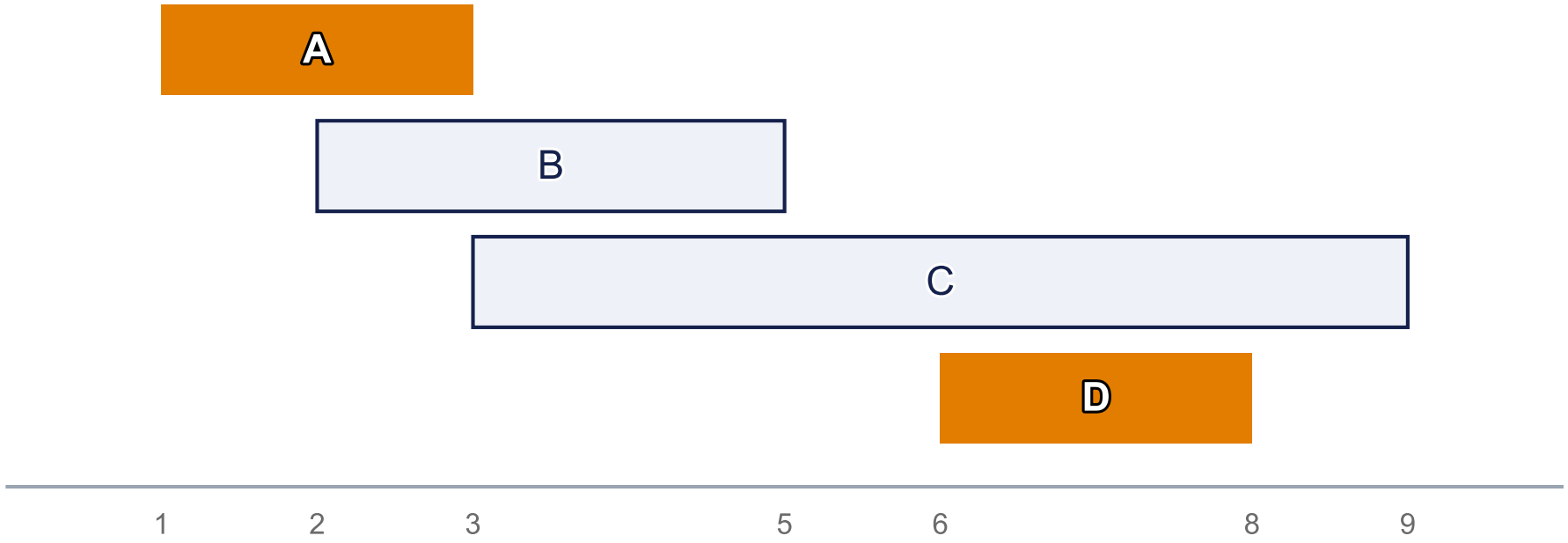
Idea greedy 1: los escogemos del **más corto al más largo**.



El más corto es D.

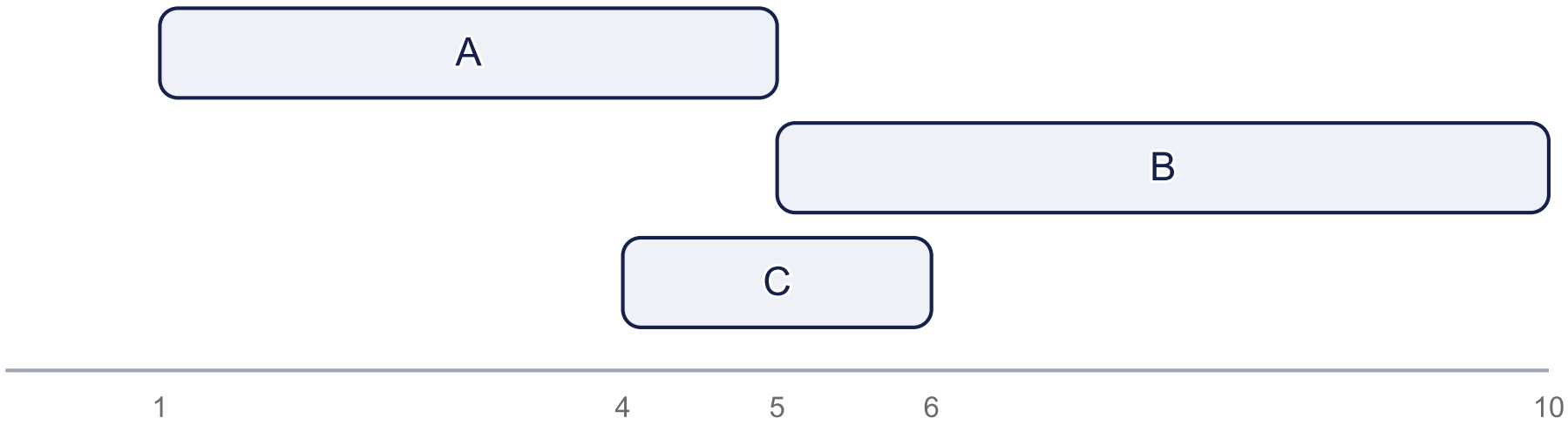
Programando eventos

Idea greedy 1: los escogemos del **más corto al más largo**.

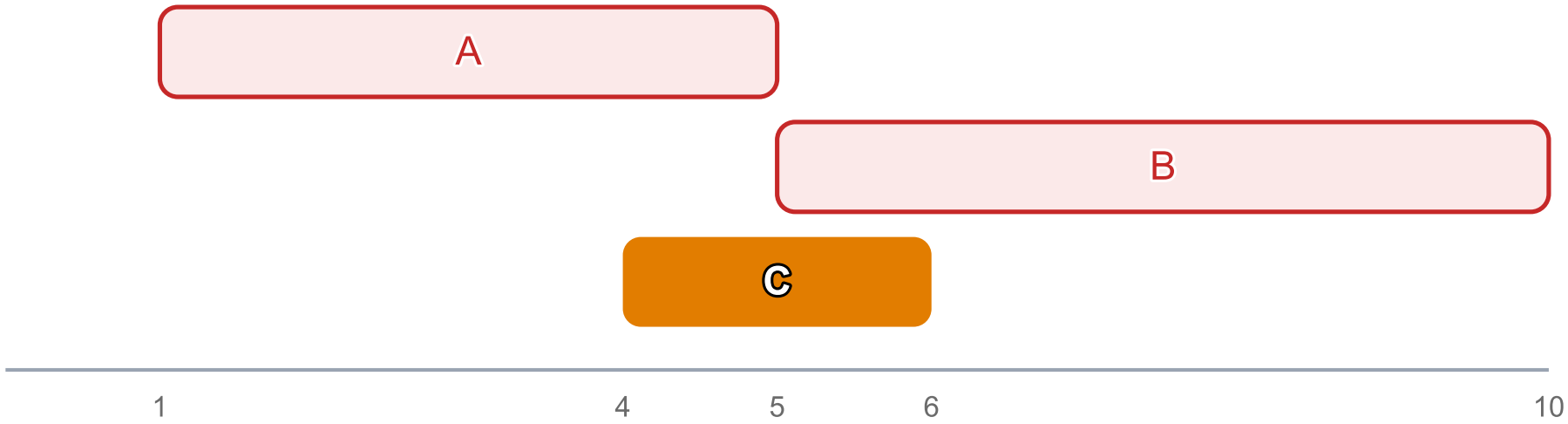


Luego A: 2 películas. Aquí funciona...

¿El más corto funciona siempre?

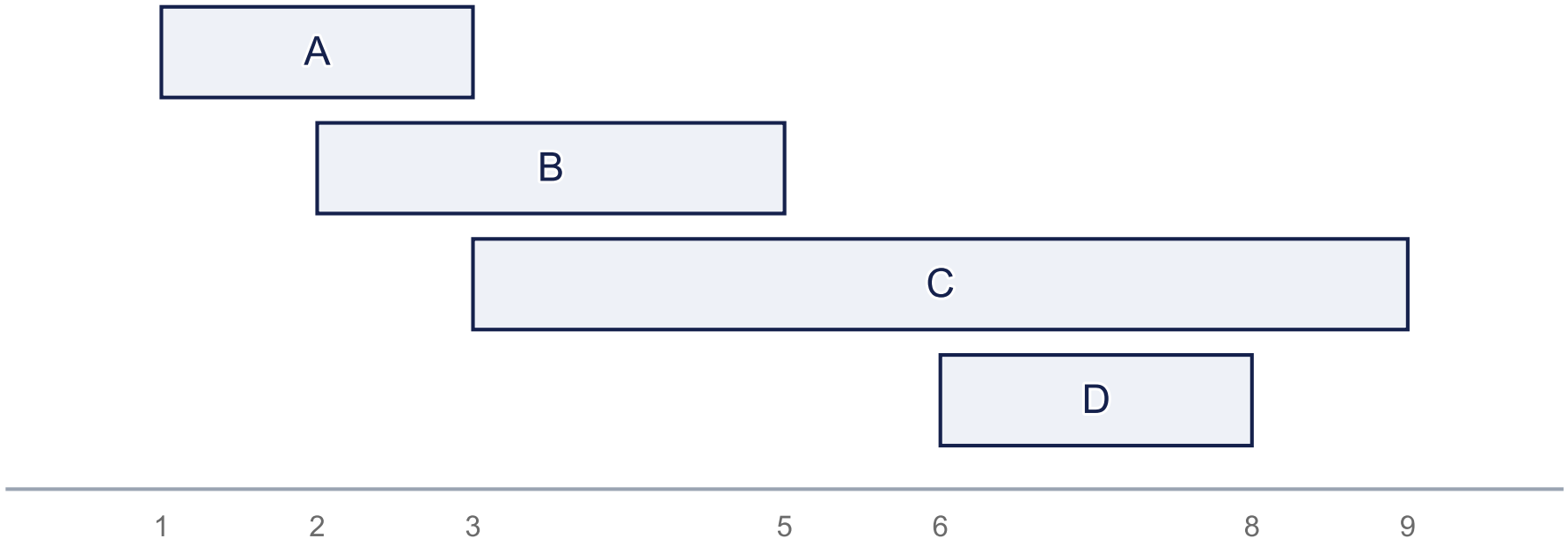


No funciona 😕



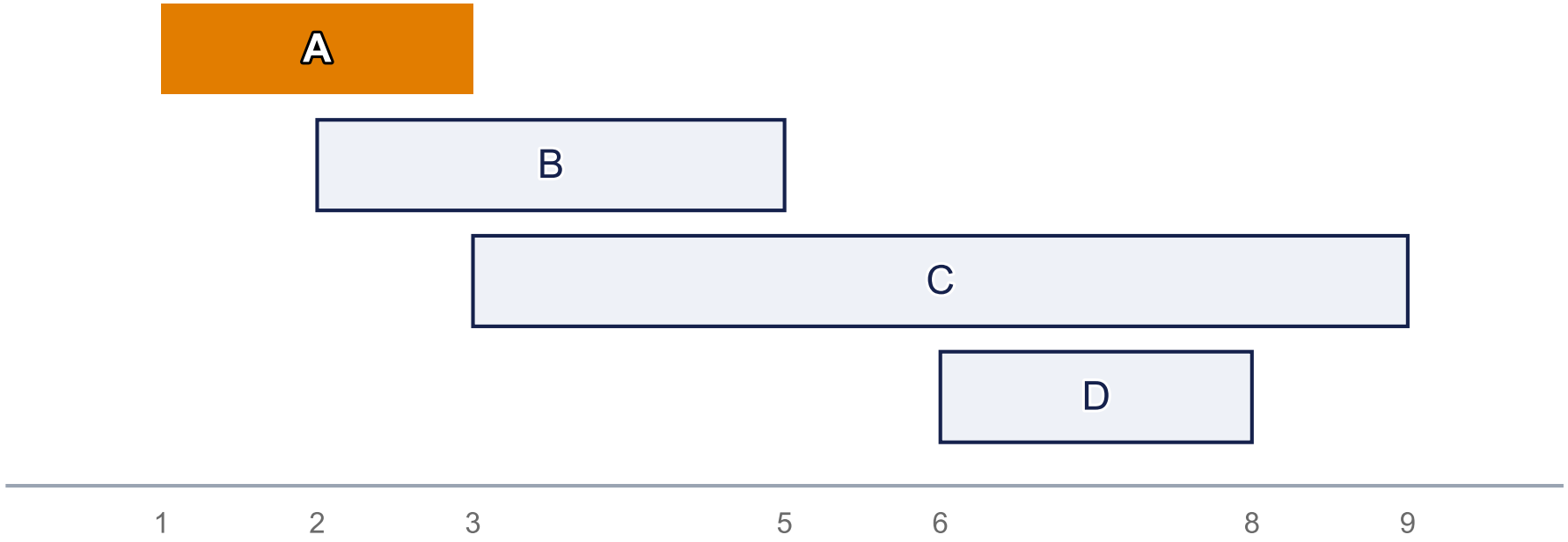
Escogemos C (el más corto) y bloquea a A y B: solo **1**. El óptimo son **2** (A y B).

Idea greedy 2: escogemos en orden de cuál **comienza primero**.



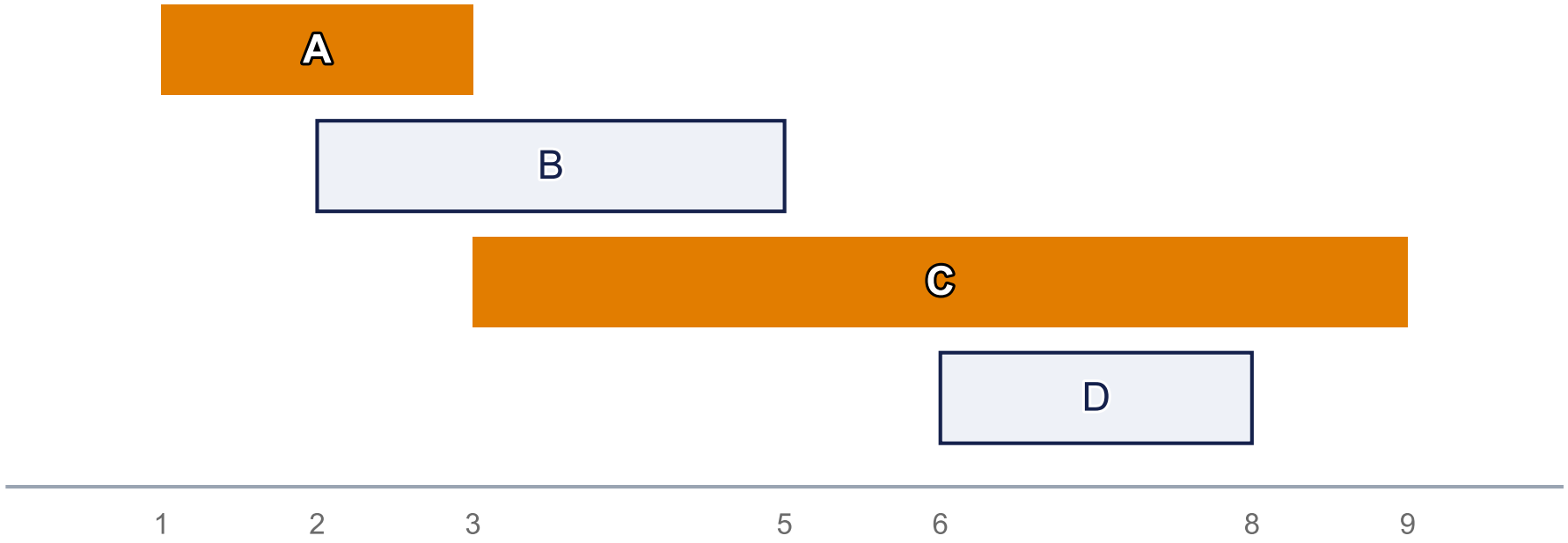
Programando eventos

Idea greedy 2: escogemos en orden de cuál **comienza primero**.



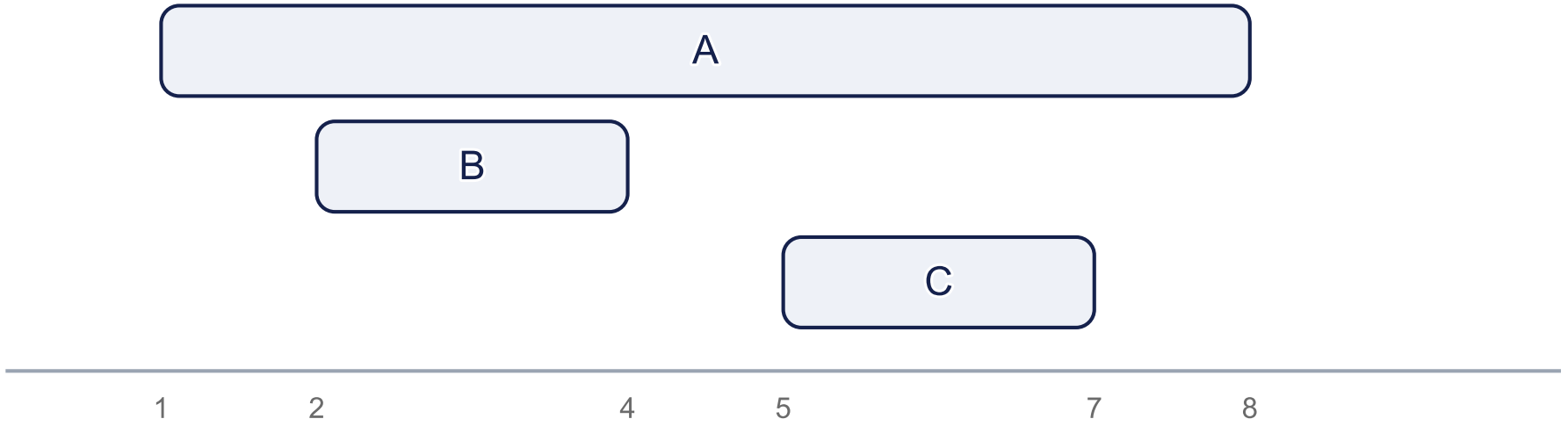
El primero en comenzar es A.

Idea greedy 2: escogemos en orden de cuál **comienza primero**.

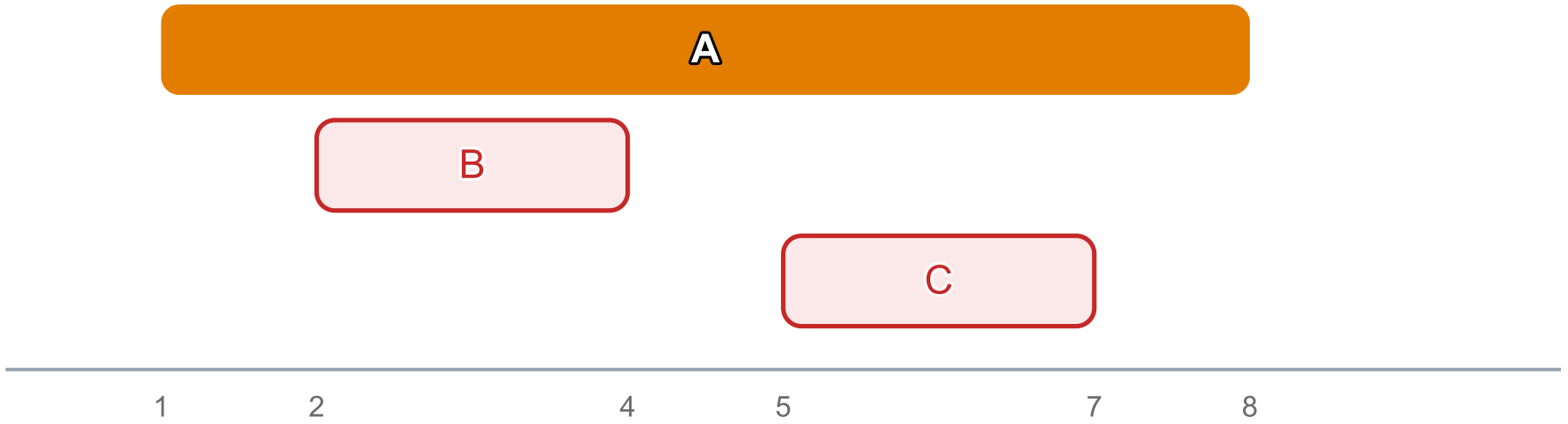


Luego C: 2 películas. Aquí funciona...

¿Comenzar primero funciona siempre?

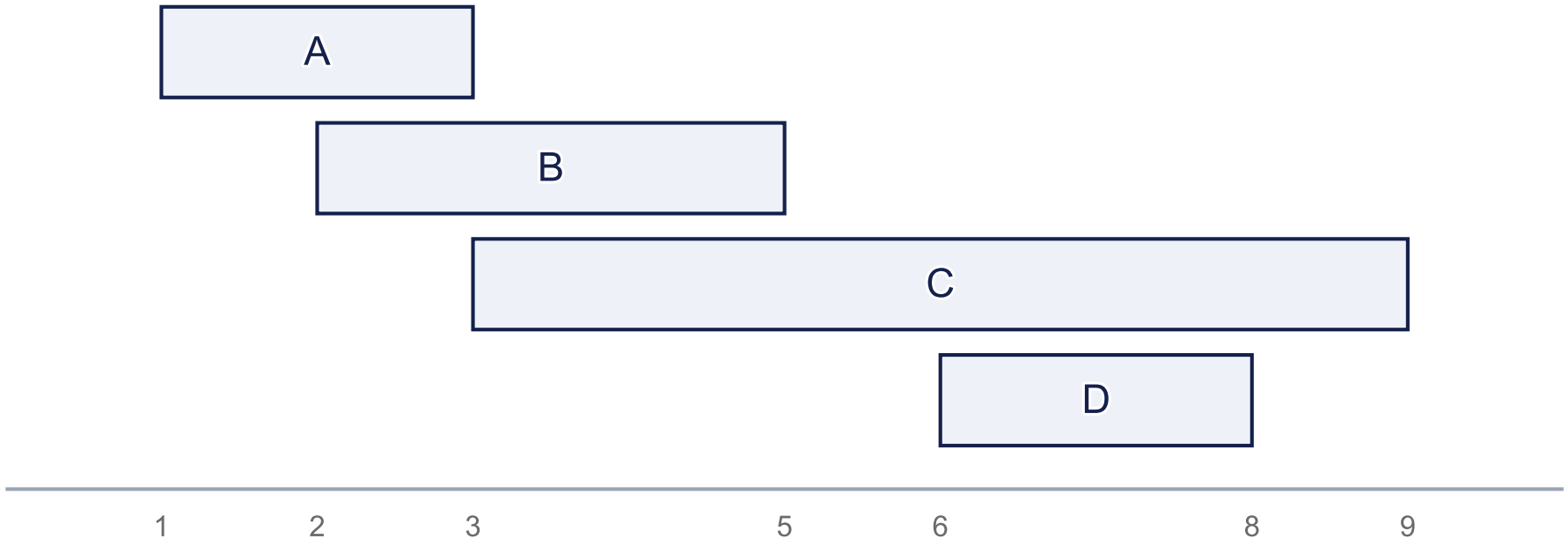


Tampoco funciona 😕 😕



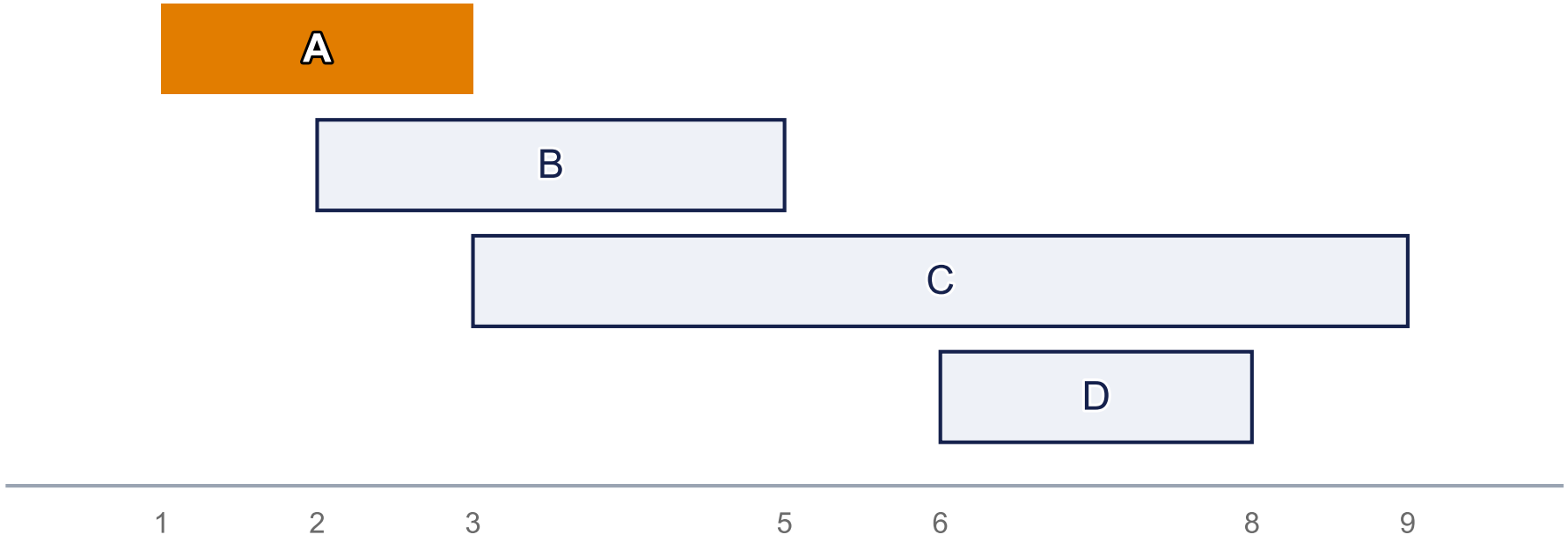
Escogemos A (comienza primero) y bloquea a B y C: solo **1**. El óptimo son **2** (B y C).

Idea greedy 3: escogemos en orden de cuál **termina primero**.



Programando eventos

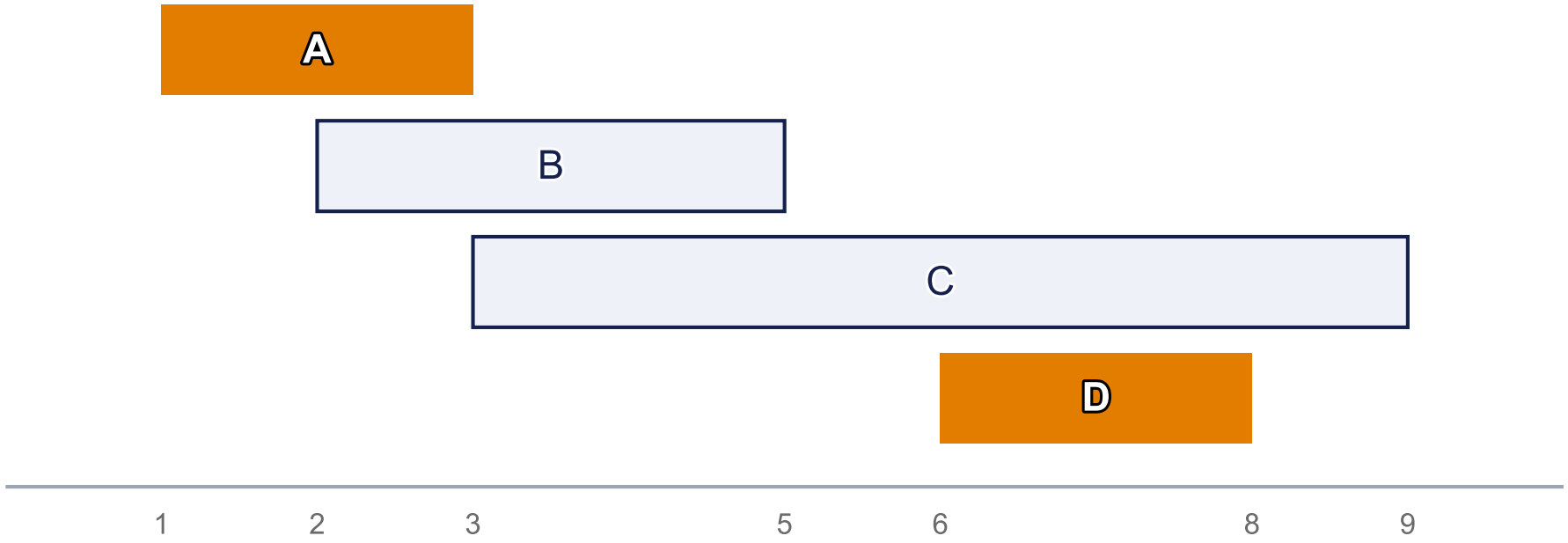
Idea greedy 3: escogemos en orden de cuál **termina primero**.



El primero en terminar es A.

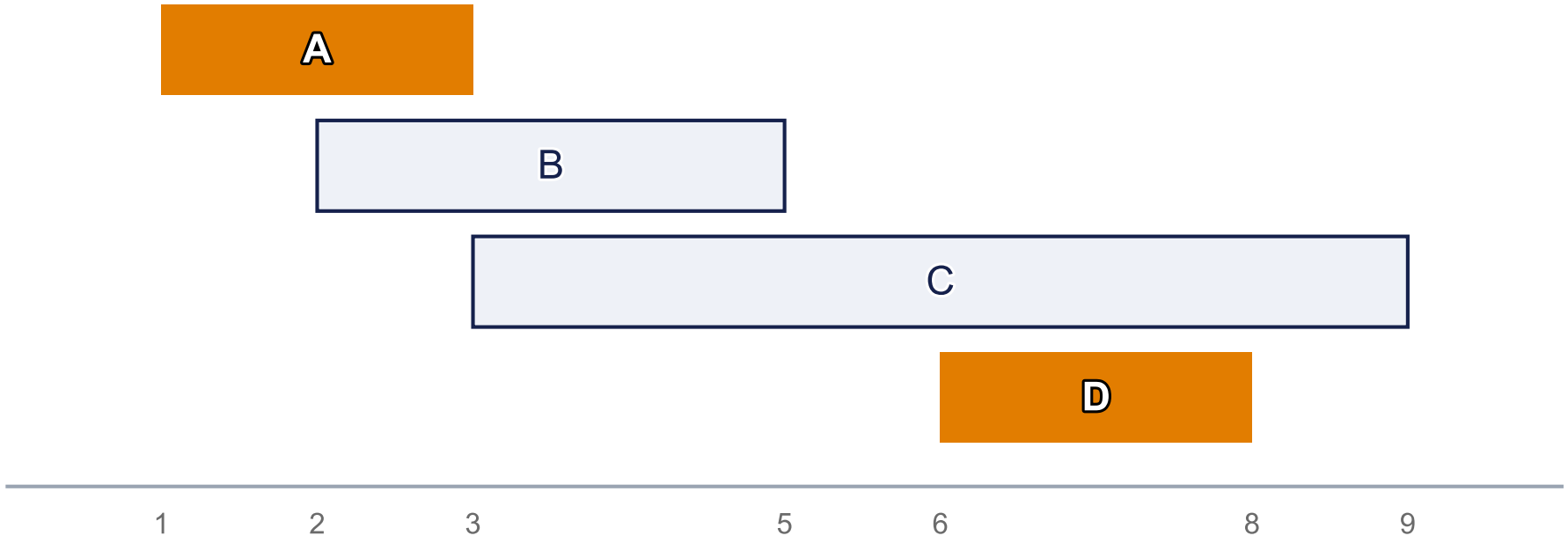
Programando eventos

Idea greedy 3: escogemos en orden de cuál **termina primero**.



Luego D: 2 películas.

Idea greedy 3: escogemos en orden de cuál **termina primero**.



¡Funciona siempre!

Algoritmos greedy

Un algoritmo greedy es uno que...

- Toma siempre la "mejor" decisión **local**, con la esperanza de construir una solución óptima **global**.
- No es un algoritmo específico, sino una **estrategia** o **forma de pensar**.
- Son **peligrosos**, porque es fácil convencerse de que funciona cuando no.
- Experiencia, intuición, jugársela.

Dragones y caballeros

Tenemos n dragones y m caballeros. Los dragones tienen fuerzas $a_1, a_2, \ldots, a_n$ y los caballeros $b_1, b_2, \ldots, b_m$. Un caballero puede pelear con un solo dragón y le gana si tiene **igual o más fuerza**.

¿Escogiendo las peleas, es posible derrotar a todos los dragones?

Dragones y caballeros

Tenemos n dragones y m caballeros. Los dragones tienen fuerzas $a_1, a_2, \ldots, a_n$ y los caballeros $b_1, b_2, \ldots, b_m$. Un caballero puede pelear con un solo dragón y le gana si tiene **igual o más fuerza**.

¿Escogiendo las peleas, es posible derrotar a todos los dragones?

Entrada

```
2 3
5 4
7 8 4
```

Salida

```
YES
```

Entrada

```
2 1
5 5
10
```

Salida

```
NO
```

Bit flips

Tenemos dos secuencias a y b de n bits. Queremos convertir a en b usando la siguiente operación:

- Escoger un substring de a e invertir todos los bits en él.

¿Cuál es la menor cantidad de operaciones para convertir a en b?

Bit flips

Tenemos dos secuencias a y b de n bits. Queremos convertir a en b usando la siguiente operación:

- Escoger un substring de a e invertir todos los bits en él.

¿Cuál es la menor cantidad de operaciones para convertir a en b?

Entrada

```
7
1000100
0011100
```

Salida

```
2
```

Búsqueda binaria

Buscar un número en un arreglo ordenado

Buscamos el 2 en este arreglo **ordenado**:

-5	0	1	2	3	3	5	5	7	9	10	15

El algoritmo

- Inicializamos nuestro espacio de búsqueda $[l, r]$.
- En cada iteración, consultamos el elemento al medio del espacio de búsqueda $\lfloor (l + r)/2 \rfloor$ y descartamos una de las mitades, achicando nuestro espacio de búsqueda.
- Cantidad de iteraciones hasta que el espacio de búsqueda tenga tamaño 1:
- $O(\log n)$

Generalización

La búsqueda binaria no sirve solo para buscar un número en un arreglo. Para generalizarla, necesitamos dos conceptos:

- **Función binaria**: es una función f que retorna true o false.
- **Función monótona**: toma un valor hasta cierto punto, en donde cambia al otro valor y se mantiene en ese otro valor.
- TTTTTTTTTTFFFFFFFF es monótona
- FFTTTTTTTTTTT es monótona
- FFFFFFFFFFFFFFFF es monótona
- FFFFTTTTTFFFFFFF **no** es monótona

Aplicando la generalización

Buscamos el 2 en este arreglo **ordenado**.

-5	0	1	2	3	3	5	5	7	9	10	15

- **Función binaria:** $f(x) = (x \geq 2)$.
- **¿Es monótona?** Sí, porque el arreglo está ordenado.

Aplicando la generalización

Buscamos el 2 en este arreglo **ordenado**. Función binaria: $f(x) = (x \geq 2)$, que es **monótona**.

Basta encontrar el **primer** V: ese es el 2.

Implementación

Dado un entero n calcula

$$\lfloor \sqrt{n} \rfloor$$

Raíz cuadrada

Dado un entero n calcula

$$\lfloor \sqrt{n} \rfloor$$

Entrada	Salida
9	3

Entrada	Salida
15	3

Suma de cuadrados

Dado un entero c ($0 \leq c \leq 2^{31} - 1$), di si existen dos enteros a y b tales que $a^2 + b^2 = c$.

Suma de cuadrados

Dado un entero c ($0 \leq c \leq 2^{31} - 1$), di si existen dos enteros a y b tales que $a^2 + b^2 = c$.

Entrada

```
5
```

Salida

```
YES
```

Entrada

```
3
```

Salida

```
NO
```

Fábrica de figuritas

Tienes n impresoras 3D, la i-ésima tarda a_i segundos en imprimir una figurita ($1 \leq n \leq 2 \cdot 10^5, 1 \leq a_i \leq 10^9$).

¿Cuál es el tiempo mínimo requerido para imprimir k figuritas? ($1 \leq k \leq 10^9$)

Fábrica de figuritas

Tienes n impresoras 3D, la i-ésima tarda a_i segundos en imprimir una figurita ($1 \leq n \leq 2 \cdot 10^5, 1 \leq a_i \leq 10^9$).

¿Cuál es el tiempo mínimo requerido para imprimir k figuritas? ($1 \leq k \leq 10^9$)

Entrada

```
n k
a1 a2 a3
```

Salida

```
tiempo_minimo
```

Entrada

```
3 7
3 2 5
```

Salida

```
8
```