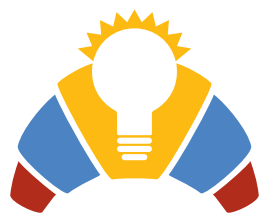
Búsqueda Binaria y Ventanas Deslizantes

Nivel Inicial

Federico Bersano

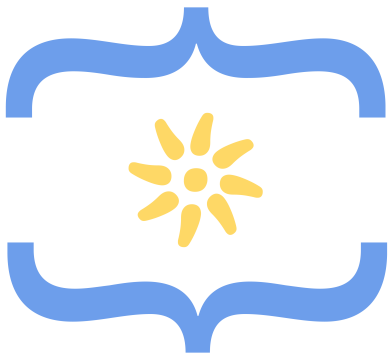
Universidad Nacional de Rosario
Don Gato

Training Camp 2026



¡Gracias sponsors!

Organiza



Facultad de INFORMÁTICA

UNIVERSIDAD NACIONAL DE LA PLATA

Diamond Sponsor

Búsqueda en arreglo ordenado

Problema

Dado un arreglo **ordenado** de n números A, y un número x, queremos saber si x está en A. En caso de estar, buscamos la posición de su **primera aparición**.

Búsqueda en arreglo ordenado

Problema

Dado un arreglo **ordenado** de n números A, y un número x, queremos saber si x está en A. En caso de estar, buscamos la posición de su **primera aparición**.

Ejemplo

$$A = [1,\ 3,\ 4,\ 6,\ 7,\ 8,\ 9,\ 10,\ 13,\ 13,\ 14,\ 15,\ 17,\ 19,\ 21]$$

- $x = 13 \Rightarrow$ primera aparición en la posición 8.
- $x = 5 \Rightarrow$ no está en A.

Solución con búsqueda lineal

Idea: recorrer el arreglo de izquierda a derecha y comparar.

```
int busquedaLineal(int x, vector<int> &A) {
    int n = (int)A.size();
    for (int i = 0; i < n; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Solución con búsqueda lineal

Idea: recorrer el arreglo de izquierda a derecha y comparar.

```
int busquedaLineal(int x, vector<int> &A) {
    int n = (int)A.size();
    for (int i = 0; i < n; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Complejidad: $\mathcal{O}(n)$, en el peor caso hay que recorrer todo el arreglo.

Solución con búsqueda lineal

Idea: recorrer el arreglo de izquierda a derecha y comparar.

```
int busquedaLineal(int x, vector<int> &A) {
    int n = (int)A.size();
    for (int i = 0; i < n; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Complejidad: $\mathcal{O}(n)$, en el peor caso hay que recorrer todo el arreglo.

¿Qué información del arreglo **no** estamos usando?

Solución con búsqueda lineal

Idea: recorrer el arreglo de izquierda a derecha y comparar.

```
int busquedaLineal(int x, vector<int> &A) {
    int n = (int)A.size();
    for (int i = 0; i < n; i++) {
        if (A[i] == x) {
            return i;
        }
    }
    return -1;
}
```

Complejidad: $\mathcal{O}(n)$, en el peor caso hay que recorrer todo el arreglo.

¿Qué información del arreglo **no** estamos usando?

No estamos usando que el arreglo está **ordenado**.

Solución con búsqueda binaria

La idea es mantener **tres zonas** sobre el arreglo:

- **A la izquierda**: una zona donde todo es $< x$.
- **A la derecha**: una zona donde todo es $\geq x$.
- **En el medio**: el rango activo, donde todavía no sabemos nada.

Solución con búsqueda binaria

La idea es mantener **tres zonas** sobre el arreglo:

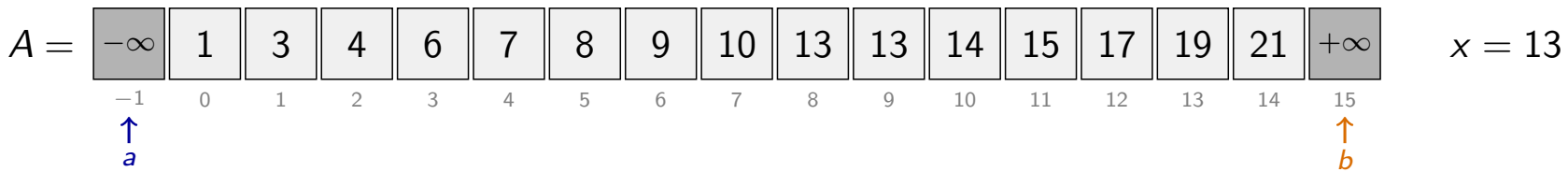
- **A la izquierda**: una zona donde todo es $< x$.
- **A la derecha**: una zona donde todo es $\geq x$.
- **En el medio**: el rango activo, donde todavía no sabemos nada.

En cada paso miramos el **elemento del medio** del rango activo, y eso nos permite **descartar la mitad** del rango activo.

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

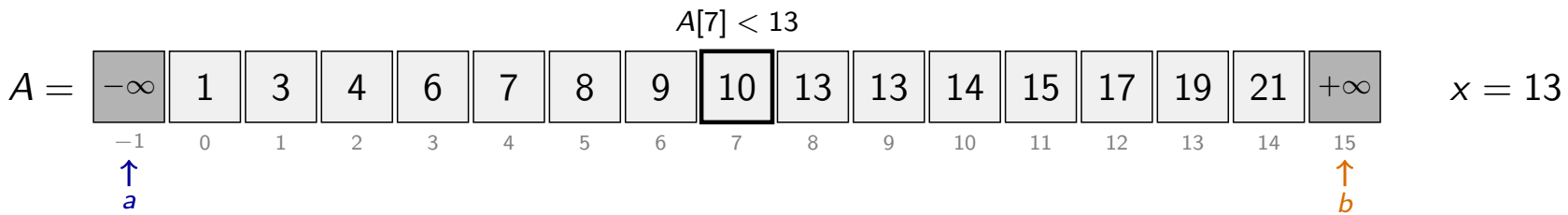


- Arranco con $a = -1$, $b = 15$: todo es rango activo.

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

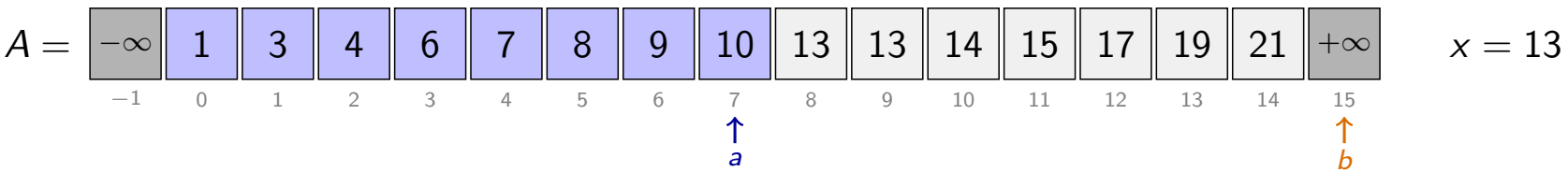


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$?

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

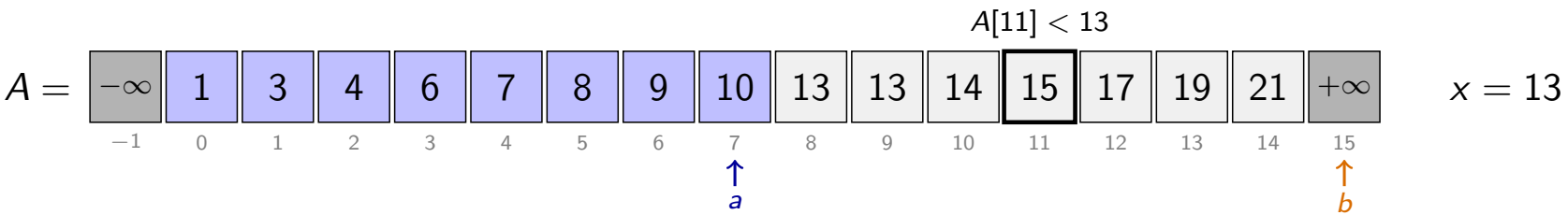


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

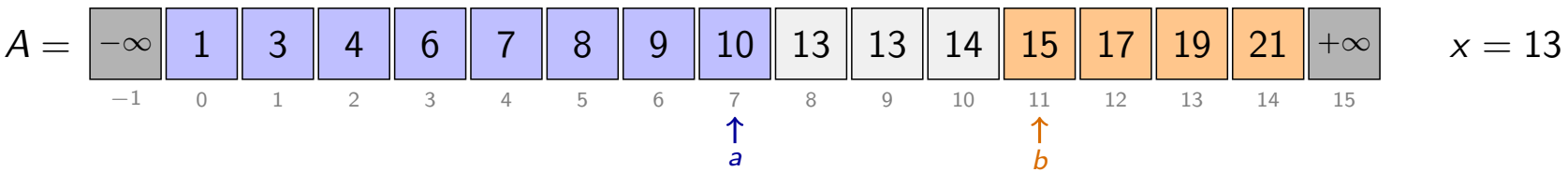


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$?

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

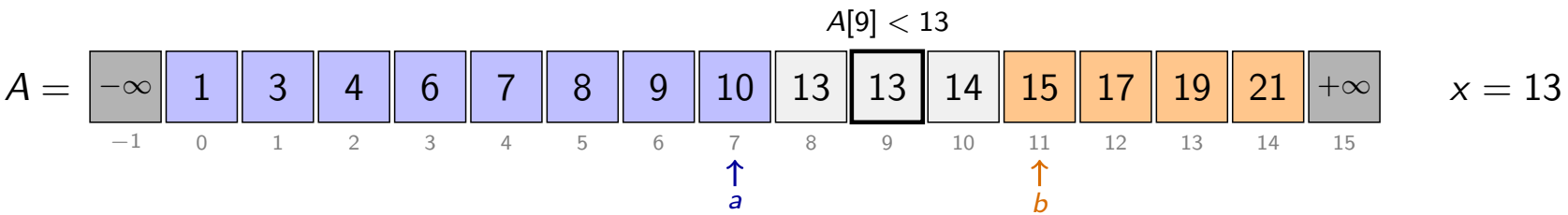


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$? No $\Rightarrow$ $b = 11$.

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

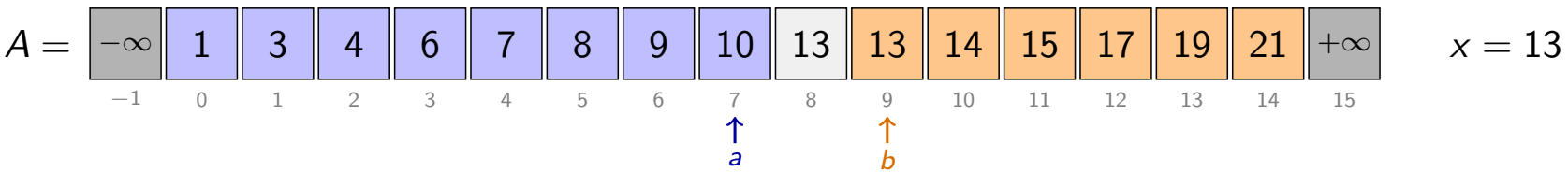


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$? No $\Rightarrow$ $b = 11$.
- $m = 9$: ¿$A[9] = 13 < 13$?

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

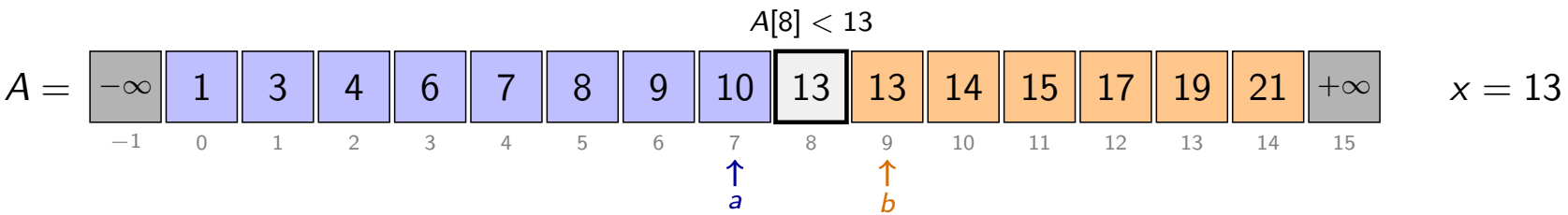


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$? No $\Rightarrow$ $b = 11$.
- $m = 9$: ¿$A[9] = 13 < 13$? No $\Rightarrow$ $b = 9$.

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.

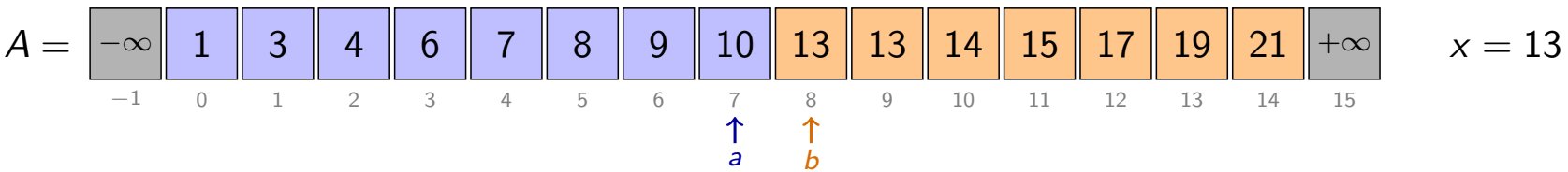


- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$? No $\Rightarrow$ $b = 11$.
- $m = 9$: ¿$A[9] = 13 < 13$? No $\Rightarrow$ $b = 9$.
- $m = 8$: ¿$A[8] = 13 < 13$?

Búsqueda binaria en acción

Invariante

Mantenemos punteros a y b tales que todo lo que está en a o antes es $< x$ y todo lo que está en b o después es $\geq x$. En cada paso achicamos $b - a$ a la mitad.



- Arranco con $a = -1$, $b = 15$: todo es rango activo.
- $m = 7$: ¿$A[7] = 10 < 13$? Sí $\Rightarrow$ $a = 7$.
- $m = 11$: ¿$A[11] = 15 < 13$? No $\Rightarrow$ $b = 11$.
- $m = 9$: ¿$A[9] = 13 < 13$? No $\Rightarrow$ $b = 9$.
- $m = 8$: ¿$A[8] = 13 < 13$? No $\Rightarrow$ $b = 8$. Termino: posición 8.

Implementación en C++

```cpp
int busquedaBinaria(int x, vector<int> &A) {
    int n = (int)A.size();
    int a = -1, b = n;          // invariante: A[a] < x, A[b] >= x
    while (b - a > 1) {         // [a+1 .. b-1] : rango activo
        int m = a + (b - a) / 2;
        if (A[m] < x) { a = m; }
        else          { b = m; }
    }
    if (b < n && A[b] == x) {
        // b: primera aparicion de x
        return b;
    }
    return -1;
}
```

Implementación en C++

```
int busquedaBinaria(int x, vector<int> &A) {
    int n = (int)A.size();
    int a = -1, b = n;          // invariante: A[a] < x, A[b] >= x
    while (b - a > 1) {         // [a+1 .. b-1] : rango activo
        int m = a + (b - a) / 2;
        if (A[m] < x) { a = m; }
        else          { b = m; }
    }
    if (b < n && A[b] == x) {
        // b: primera aparicion de x
        return b;
    }
    return -1;
}
```

Complejidad: $\mathcal{O}(\log n)$: en cada paso el rango activo se reduce a la mitad.

Implementación en C++

```cpp
int busquedaBinaria(int x, vector<int> &A) {
    int n = (int)A.size();
    int a = -1, b = n;          // invariante: A[a] < x, A[b] >= x
    while (b - a > 1) {         // [a+1 .. b-1] : rango activo
        int m = a + (b - a) / 2;
        if (A[m] < x) { a = m; }
        else          { b = m; }
    }
    if (b < n && A[b] == x) {
        // b: primera aparicion de x
        return b;
    }
    return -1;
}
```

Es común escribir `m = (a + b) / 2`, pero esa forma puede resultar en *overflow* antes de dividir por 2.

Complejidad: $\mathcal{O}(\log n)$: en cada paso el rango activo se reduce a la mitad.

Predicados monótonos

En el fondo, lo único que usamos de A fue un **predicado** que depende de un índice k:

$$P(k): \ A[k] \geq x$$

y el hecho de que este predicado es **monótono**: falso, falso, ..., falso, verdadero, verdadero, ..., verdadero.

k	0	1	$\cdots$	a	b	$b+1$	$\cdots$	n
$P(k)$	F	F	$\cdots$	F	V	V	$\cdots$	V

Definición

Decimos que P es un **predicado monótono** si es falso hasta cierto punto y de ahí en adelante siempre verdadero (o al revés). **Buscar dónde cambia** de valor de verdad es justo lo que hace la búsqueda binaria.

El template general

El código general es muy parecido al anterior: solo cambiamos `A[m] < x` por el predicado `P(m)` correspondiente.

```
// P predicado monotono F F ... F V V ... V
int buscarPredicado(int a, int b) {
    // invariante: P(a) = F, P(b) = V
    while (b - a > 1) {
        int m = a + (b - a) / 2;
        if (P(m)) { b = m; }
        else      { a = m; }
    }
    // primer valor donde P es verdadero
    return b;
}
```

El template general

El código general es muy parecido al anterior: solo cambiamos `A[m] < x` por el predicado `P(m)` correspondiente.

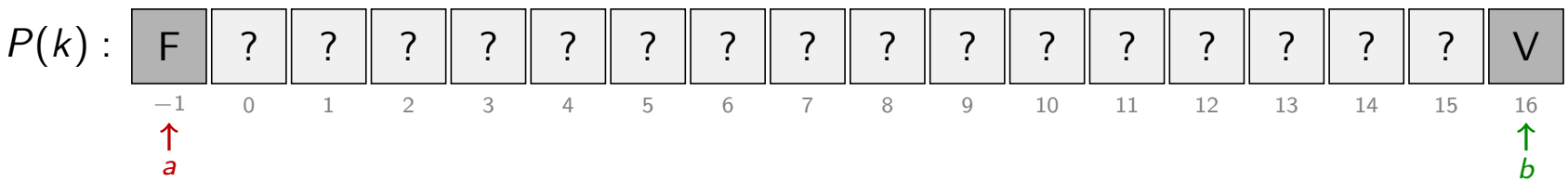
```
// P predicado monotono F F ... F V V ... V
int buscarPredicado(int a, int b) {
    // invariante: P(a) = F, P(b) = V
    while (b - a > 1) {
        int m = a + (b - a) / 2;
        if (P(m)) { b = m; }
        else      { a = m; }
    }
    // primer valor donde P es verdadero
    return b;
}
```

Con esta idea podemos resolver **muchísimos** problemas que no tienen nada que ver con "buscar en un arreglo".

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

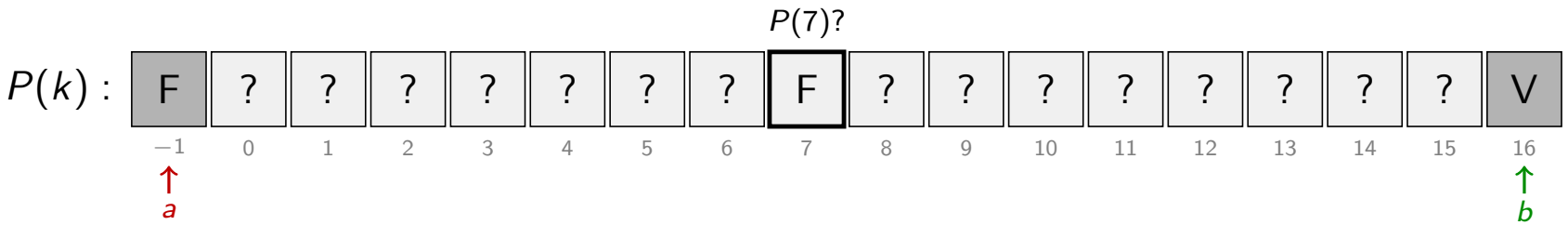


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

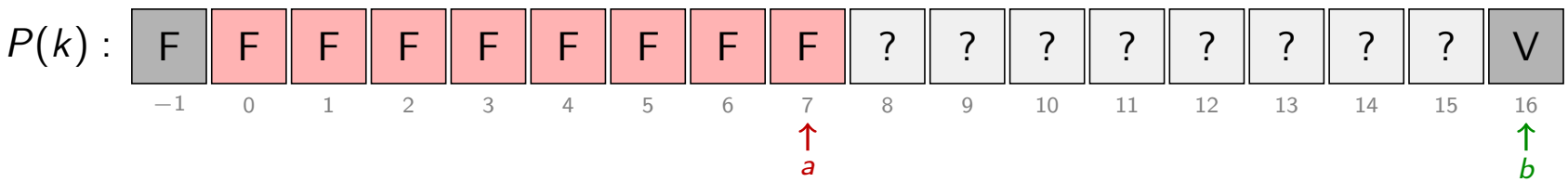


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

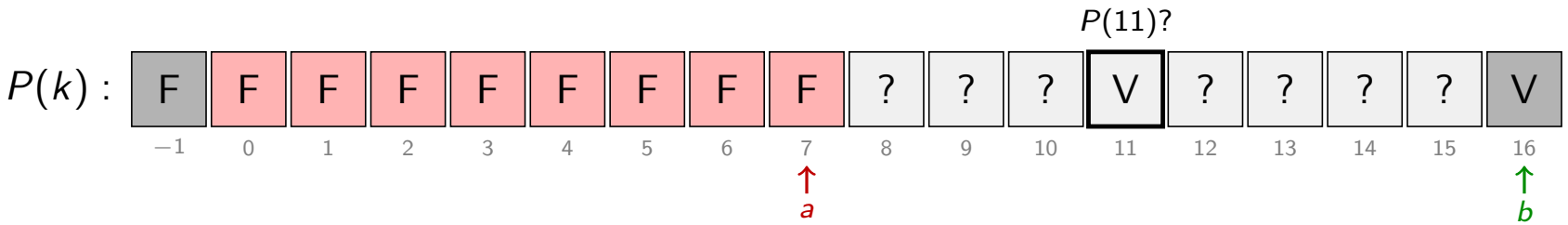


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

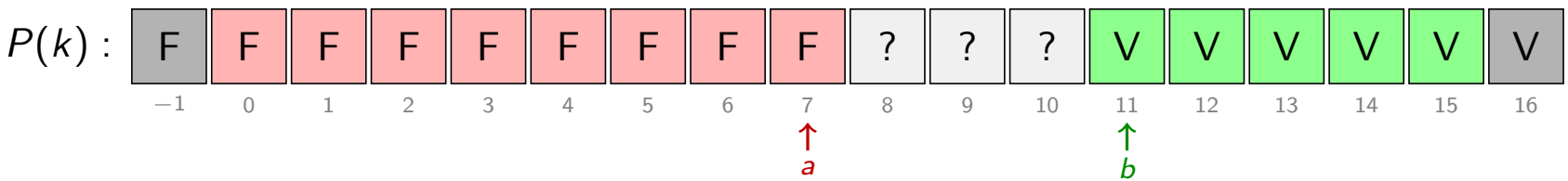


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

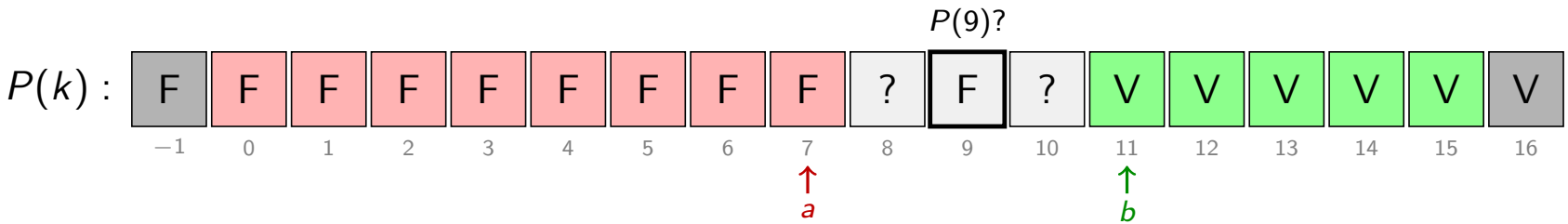


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$, es V $\Rightarrow$ $b = 11$.

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

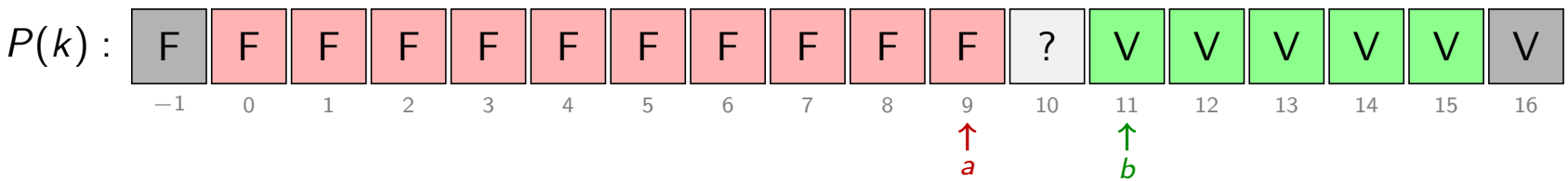


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$, es V $\Rightarrow$ $b = 11$.
- $m = 9$: consulto $P(9)$

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

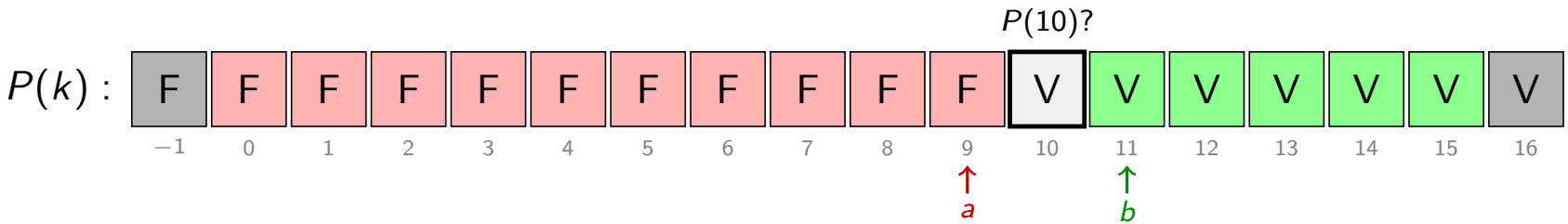


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$, es V $\Rightarrow$ $b = 11$.
- $m = 9$: consulto $P(9)$, es F $\Rightarrow$ $a = 9$.

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.

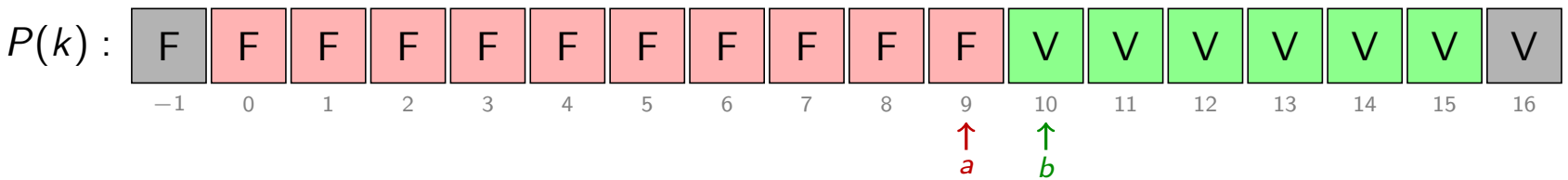


- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$, es V $\Rightarrow$ $b = 11$.
- $m = 9$: consulto $P(9)$, es F $\Rightarrow$ $a = 9$.
- $m = 10$: consulto $P(10)$

El template general en acción

Invariante

Mantenemos punteros a y b tales que $P(a) = \text{F}$ y $P(b) = \text{V}$ (o al revés). En cada paso achicamos $b - a$ a la mitad.



- Arranco con $a = -1$, $b = 16$: todo el rango es desconocido.
- $m = 7$: consulto $P(7)$, es F $\Rightarrow$ $a = 7$.
- $m = 11$: consulto $P(11)$, es V $\Rightarrow$ $b = 11$.
- $m = 9$: consulto $P(9)$, es F $\Rightarrow$ $a = 9$.
- $m = 10$: consulto $P(10)$, es V $\Rightarrow$ $b = 10$. Termino: $b = 10$ es el primer V.

Ejemplo 1: Evolucionando Pokémones

Problema

Ash tiene N pokémones y M caramelos. Para cada pokémon puede hacer una de dos cosas:

- Evolucionarlo, con un costo de X caramelos.
- Venderlo sin evolucionar, con una ganancia de Y caramelos.

¿Cuál es la **máxima** cantidad de pokémones que puede evolucionar?

Ejemplo 1: Evolucionando Pokémones

Problema

Ash tiene N pokémones y M caramelos. Para cada pokémon puede hacer una de dos cosas:

- Evolucionarlo, con un costo de X caramelos.
- Venderlo sin evolucionar, con una ganancia de Y caramelos.

¿Cuál es la **máxima** cantidad de pokémones que puede evolucionar?

Observación

Si decidimos evolucionar k pokémones, podemos **vender** los otros $N - k$. Entonces disponemos de $M + (N - k) \cdot Y$ caramelos, y necesitamos que:

$$k \cdot X \leq (N - k) \cdot Y + M$$

Ejemplo 1: Evolucionando Pokémones

Solución directa: probar todos los valores de k desde 0 hasta N, y quedarnos con el mayor que cumpla $k \cdot X \leq (N - k) \cdot Y + M$. Esto es $\mathcal{O}(N)$.

Ejemplo 1: Evolucionando Pokémones

Solución directa: probar todos los valores de k desde 0 hasta N, y quedarnos con el mayor que cumpla $k \cdot X \leq (N - k) \cdot Y + M$. Esto es $\mathcal{O}(N)$.

¿Podemos hacerlo más rápido? El predicado $P(k)$: "puedo evolucionar k pokémones" es **monótono**:

- Si puedo evolucionar k, también puedo con $k - 1, k - 2, \ldots, 0$ (evoluciono menos).
- Si NO puedo con k, tampoco voy a poder con $k + 1, k + 2, \ldots$

Ejemplo 1: Evolucionando Pokémones

Solución directa: probar todos los valores de k desde 0 hasta N, y quedarnos con el mayor que cumpla $k \cdot X \leq (N - k) \cdot Y + M$. Esto es $\mathcal{O}(N)$.

¿Podemos hacerlo más rápido? El predicado $P(k)$: "puedo evolucionar k pokémones" es **monótono**:

- Si puedo evolucionar k, también puedo con $k - 1, k - 2, \ldots, 0$ (evoluciono menos).
- Si NO puedo con k, tampoco voy a poder con $k + 1, k + 2, \ldots$

$\Rightarrow$ búsqueda binaria sobre k en $\mathcal{O}(\log N)$. (Acá P es verdadero al principio y luego falso: a guarda el último verdadero.)

Ejemplo 1: Evolucionando Pokémones

A veces la desigualdad se puede despejar directamente, sin ni siquiera necesitar la búsqueda binaria:

$$k \cdot X \leq (N - k) \cdot Y + M$$

$$k \cdot X + k \cdot Y \leq N \cdot Y + M \implies k \leq \frac{N \cdot Y + M}{X + Y}$$

$$\text{Respuesta} = \min\left(N, \left\lfloor \frac{N \cdot Y + M}{X + Y} \right\rfloor\right)$$

Ejemplo 1: Evolucionando Pokémones

Moraleja

La búsqueda binaria no siempre es la solución más rápida posible, pero casi siempre es la más **fácil de encontrar** una vez que identificás el predicado monótono.

Ejemplo 2: Vacas Agresivas

Problema

Hay n establos ubicados en posiciones $x_1 < x_2 < \cdots < x_n$ sobre una recta. Queremos ubicar C vacas, una por establo, de forma de **maximizar la mínima distancia** entre dos vacas cualesquiera.

Ejemplo 2: Vacas Agresivas

Problema

Hay n establos ubicados en posiciones $x_1 < x_2 < \cdots < x_n$ sobre una recta. Queremos ubicar C vacas, una por establo, de forma de **maximizar la mínima distancia** entre dos vacas cualesquiera.

Ejemplo

Con establos en $\{1, 2, 4, 8, 9\}$ y $C = 3$ vacas, la mejor opción es ubicarlas en $1, 4, 9$:

Las distancias son 3 y 5, así que la mínima es 3. Otras opciones son peores: por ejemplo, en $2, 8, 9$ las distancias son 6 y 1, y la mínima baja a 1.

Ejemplo 2: Vacas Agresivas

Predicado $P(d)$: "¿se pueden ubicar las C vacas con distancia mínima $\geq d$?"

Ejemplo 2: Vacas Agresivas

Predicado $P(d)$: "¿se pueden ubicar las C vacas con distancia mínima $\geq d$?"

Es monótono: si se puede con distancia d, con la exigencia menor $d-1$ también (sirven las mismas posiciones); y si no se puede con d, menos aún con $d+1$. Entonces $P(d)$ es verdadero para d chico y falso para d grande.

Ejemplo 2: Vacas Agresivas

Predicado $P(d)$: "¿se pueden ubicar las C vacas con distancia mínima $\geq d$?"

Es monótono: si se puede con distancia d, con la exigencia menor $d-1$ también (sirven las mismas posiciones); y si no se puede con d, menos aún con $d+1$. Entonces $P(d)$ es verdadero para d chico y falso para d grande.

¿Cómo saber si $P(d)$ es verdadero? De forma **golosa**, en $\mathcal{O}(n)$: recorremos los establos ordenados y ponemos la primera vaca en el primero. Luego, ubicamos cada vaca siguiente en el primer establo que esté a distancia $\geq d$ de la última que pusimos. Si así logramos colocar las C vacas, $P(d)$ es verdadero.

Ejemplo 2: Vacas Agresivas

Predicado $P(d)$: "¿se pueden ubicar las C vacas con distancia mínima $\geq d$?"

Es monótono: si se puede con distancia d, con la exigencia menor $d-1$ también (sirven las mismas posiciones); y si no se puede con d, menos aún con $d+1$. Entonces $P(d)$ es verdadero para d chico y falso para d grande.

¿Cómo saber si $P(d)$ es verdadero? De forma **golosa**, en $\mathcal{O}(n)$: recorremos los establos ordenados y ponemos la primera vaca en el primero. Luego, ubicamos cada vaca siguiente en el primer establo que esté a distancia $\geq d$ de la última que pusimos. Si así logramos colocar las C vacas, $P(d)$ es verdadero.

Búsqueda binaria sobre d (entre 0 y la distancia máxima) $\Rightarrow$ $\mathcal{O}(n \log(\text{rango de posiciones}))$.

Ejemplo 3: Capacidad del barco

Problema

Hay n paquetes con pesos $w_1, \ldots, w_n$ que se envían **en ese orden**. Cada día, un barco carga paquetes consecutivos sin exceder su capacidad C, y los despacha. Queremos enviar todo en a lo sumo D días. ¿Cuál es la **mínima capacidad** C que lo permite?

Ejemplo 3: Capacidad del barco

Problema

Hay n paquetes con pesos $w_1, \ldots, w_n$ que se envían **en ese orden**. Cada día, un barco carga paquetes consecutivos sin exceder su capacidad C, y los despacha. Queremos enviar todo en a lo sumo D días. ¿Cuál es la **mínima capacidad** C que lo permite?

Ejemplo: pesos $[3, 2, 2, 4, 1, 4]$, $D = 3$ días

Con $C = 6$: los tres días cargan 5, 6 y 5: entran en 3 días. Con $C = 5$ hacen falta 4 días $\Rightarrow$ respuesta 6.

Ejemplo 3: Capacidad del barco

Predicado $P(C)$: "¿se puede enviar todo en $\leq D$ días con capacidad C?"

Ejemplo 3: Capacidad del barco

Predicado $P(C)$: "¿se puede enviar todo en $\leq D$ días con capacidad C?"

Es monótono: con más capacidad nunca hacen falta más días, así que si $P(C)$ es verdadero, $P(C+1)$ también. Entonces $P(C)$ es falso para C chico y verdadero para C grande.

Ejemplo 3: Capacidad del barco

Predicado $P(C)$: "¿se puede enviar todo en $\leq D$ días con capacidad C?"

Es monótono: con más capacidad nunca hacen falta más días, así que si $P(C)$ es verdadero, $P(C+1)$ también. Entonces $P(C)$ es falso para C chico y verdadero para C grande.

¿Cómo saber si $P(C)$ es verdadero? De forma **golosa**, en $\mathcal{O}(n)$: recorremos los paquetes en orden sumándolos a la carga del día actual. Cuando el siguiente paquete no entra sin pasarse de C, cerramos el día y arrancamos uno nuevo con ese paquete. Al final contamos los días usados: $P(C)$ es verdadero si son $\leq D$.

Ejemplo 3: Capacidad del barco

Predicado $P(C)$: "¿se puede enviar todo en $\leq D$ días con capacidad C?"

Es monótono: con más capacidad nunca hacen falta más días, así que si $P(C)$ es verdadero, $P(C+1)$ también. Entonces $P(C)$ es falso para C chico y verdadero para C grande.

¿Cómo saber si $P(C)$ es verdadero? De forma **golosa**, en $\mathcal{O}(n)$: recorremos los paquetes en orden sumándolos a la carga del día actual. Cuando el siguiente paquete no entra sin pasarse de C, cerramos el día y arrancamos uno nuevo con ese paquete. Al final contamos los días usados: $P(C)$ es verdadero si son $\leq D$.

Búsqueda binaria sobre C (entre el peso máximo y la suma de todos los pesos) $\Rightarrow \mathcal{O}(n \log(\text{suma de pesos}))$.

De optimización a decisión

Muchos problemas preguntan el máximo o mínimo valor de k tal que se cumpla algo.

En todos estos ejemplos hicimos el mismo truco: en vez de resolver directamente la optimización, planteamos una **pregunta de sí/no**:

$$P(k): \text{ “¿es posible lograr } k\text{?”}$$

Si P es monótono en k (y muchas veces lo es cuando el problema pide un máximo o un mínimo), **podemos aplicar búsqueda binaria** sobre la respuesta.

¿Preguntas?

Problema motivador

Problema

Dado un arreglo A de n números **positivos** y un número x, ¿existe un subarreglo (contiguo) de A que sume exactamente x?

Problema motivador

Problema

Dado un arreglo A de n números **positivos** y un número x, ¿existe un subarreglo (contiguo) de A que sume exactamente x?

Ejemplo

$$A = [2,\ 3,\ 2,\ 5,\ 1,\ 5,\ 2,\ 3]$$

- $x = 8 \Rightarrow$ el subarreglo $A[2..4] = [2, 5, 1]$ suma 8.
- $x = 4 \Rightarrow$ no hay ningún subarreglo que sume 4.

Problema motivador

Solución fuerza bruta: probar los $\mathcal{O}(n^2)$ subarreglos, usando *prefix sums* para sumar en $\mathcal{O}(1)$ cada uno $\Rightarrow \mathcal{O}(n^2)$.

Problema motivador

Solución fuerza bruta: probar los $\mathcal{O}(n^2)$ subarreglos, usando *prefix sums* para sumar en $\mathcal{O}(1)$ cada uno $\Rightarrow \mathcal{O}(n^2)$.

Solución eficiente: se puede resolver en $\mathcal{O}(n)$ con una **ventana deslizante**.

La idea de la ventana

Mantenemos una ventana $[i, j)$ y su suma actual. Arrancamos con $i = j = 0$ y **suma**$= 0$.

- Si **suma** $< x$: agrandamos la ventana por la derecha (avanza j, sumamos $A[j]$).
- Si **suma** $> x$: achicamos la ventana por la izquierda (avanza i, restamos $A[i]$).
- Si **suma** $= x$: encontramos la respuesta.

¿Por qué funciona?

Porque como todos los números son **positivos**, la suma de la ventana **crece** cuando agrandamos por la derecha y **decrece** cuando achicamos por la izquierda. Nunca tenemos que "volver atrás".

¿Qué es una ventana deslizante?

Idea general

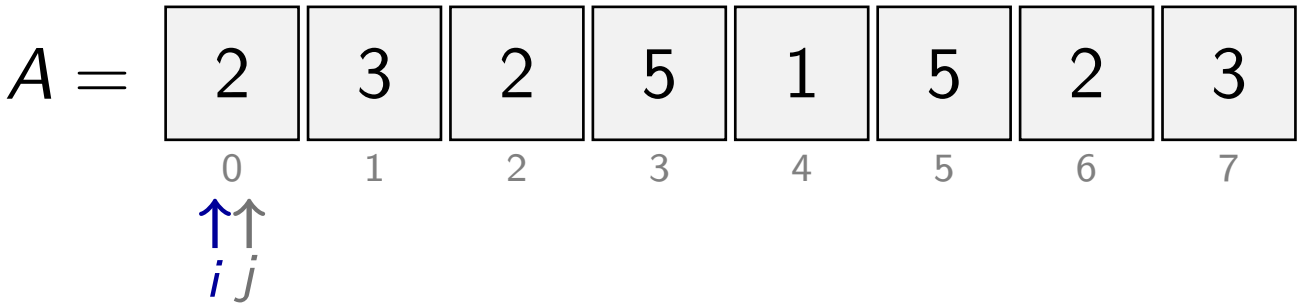
Muchos problemas piden analizar **todos los subarreglos** de un arreglo. Probarlos todos suele costar $\mathcal{O}(n^2)$ o $\mathcal{O}(n^3)$.

La técnica de **ventana deslizante** (*sliding window*) mantiene dos índices $i \leq j$ (los extremos de una "ventana" $[i, j)$) que se mueven **siempre hacia adelante**. En todo momento tenemos calculada alguna información de la ventana actual (por ejemplo, su suma), así que evitamos recalcular todo de cero.

Clave: cada puntero avanza a lo sumo n veces $\Rightarrow$ el algoritmo entero es $\mathcal{O}(n)$, aunque parezca que hay "dos bucles".

Traza sobre el ejemplo

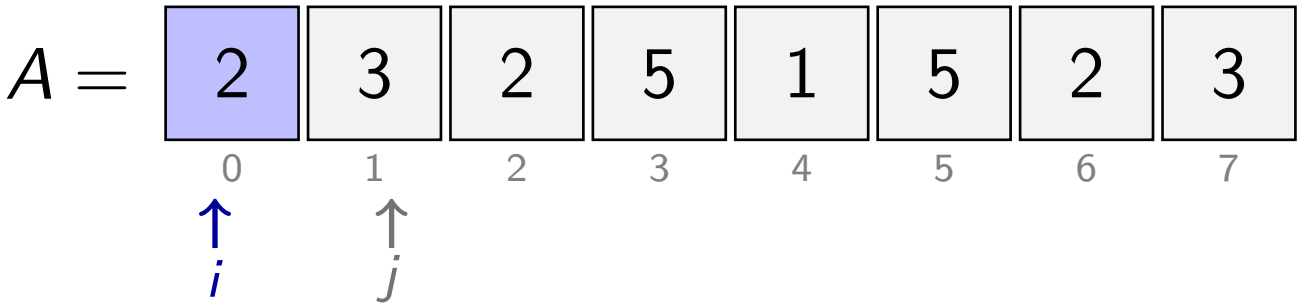
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.

Traza sobre el ejemplo

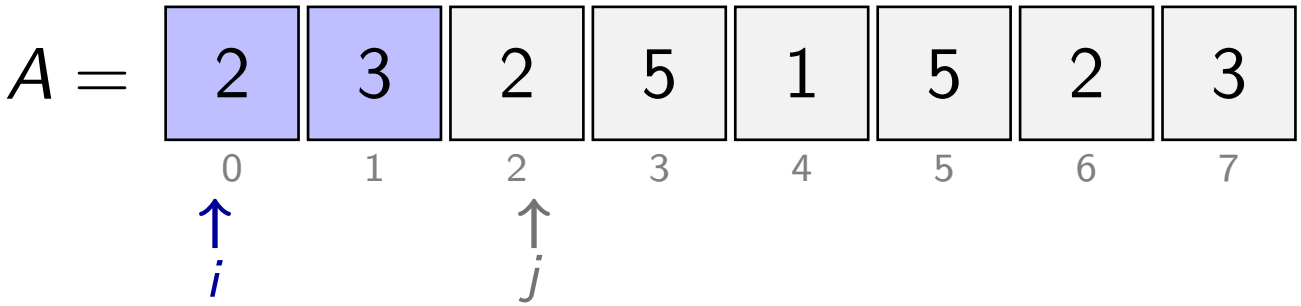
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.

Traza sobre el ejemplo

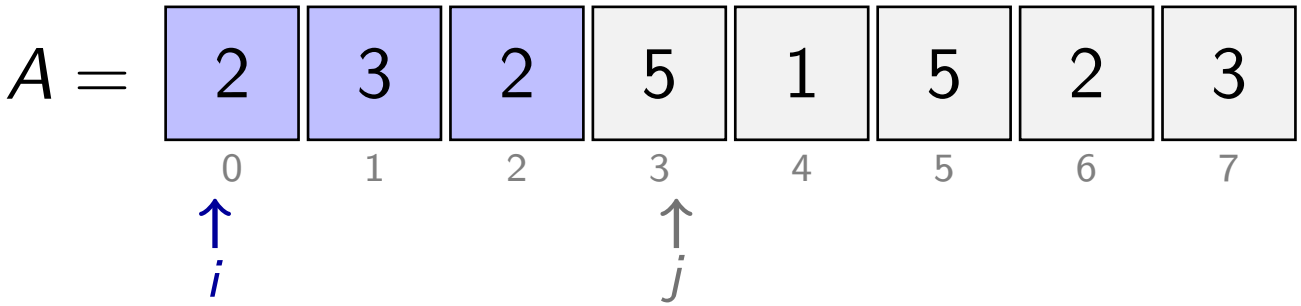
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.

Traza sobre el ejemplo

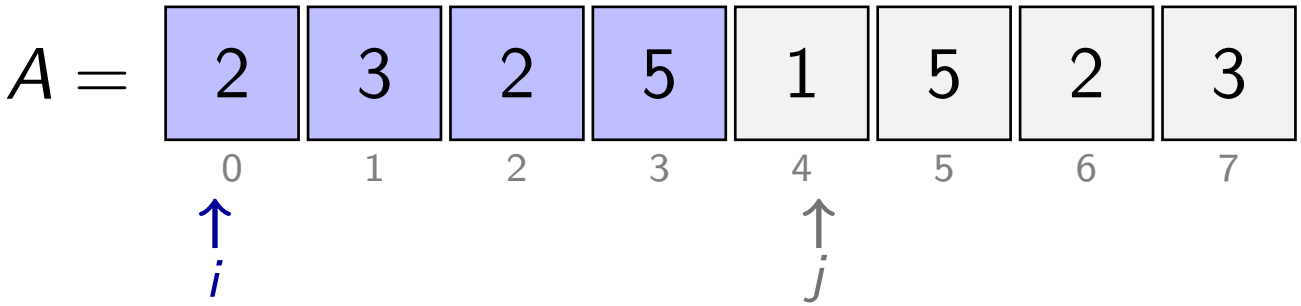
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.
- suma $= 7 < 8 \Rightarrow$ avanza j.

Traza sobre el ejemplo

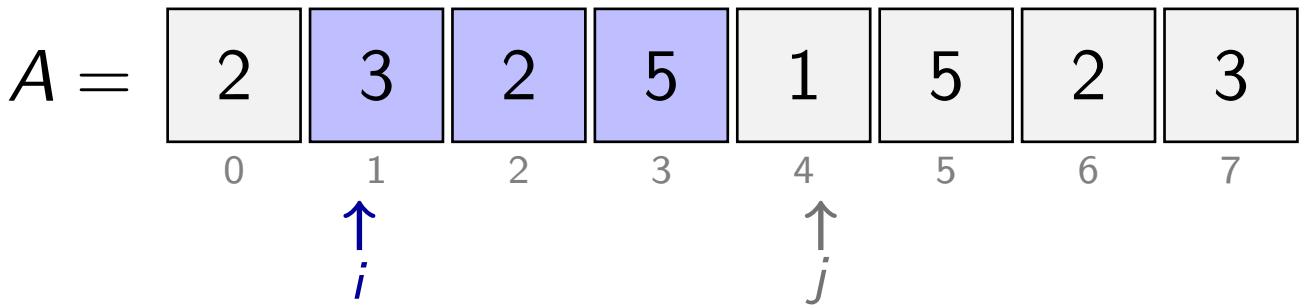
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.
- suma $= 7 < 8 \Rightarrow$ avanza j.
- suma $= 12 > 8 \Rightarrow$ avanza i (sale $A[0]$).

Traza sobre el ejemplo

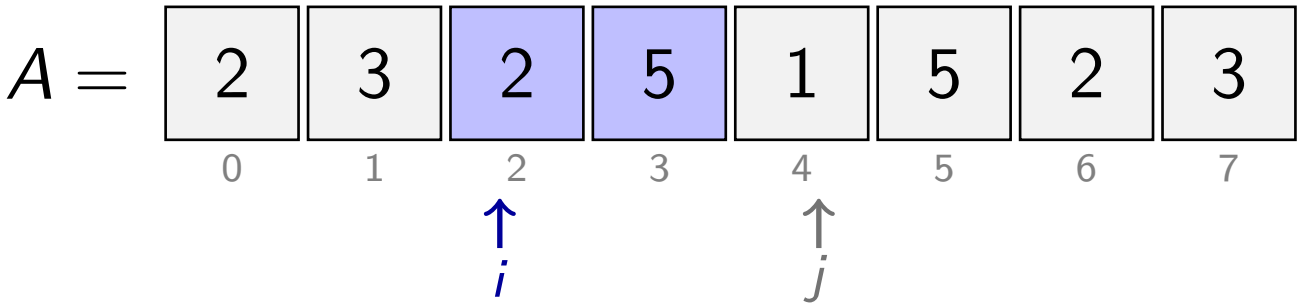
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.
- suma $= 7 < 8 \Rightarrow$ avanza j.
- suma $= 12 > 8 \Rightarrow$ avanza i (sale $A[0]$).
- suma $= 10 > 8 \Rightarrow$ avanza i (sale $A[1]$).

Traza sobre el ejemplo

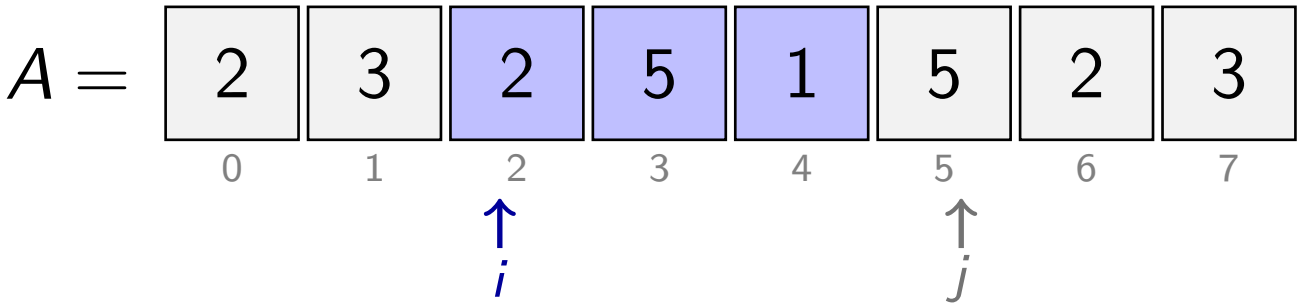
Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.
- suma $= 7 < 8 \Rightarrow$ avanza j.
- suma $= 12 > 8 \Rightarrow$ avanza i (sale $A[0]$).
- suma $= 10 > 8 \Rightarrow$ avanza i (sale $A[1]$).
- suma $= 7 < 8 \Rightarrow$ avanza j.

Traza sobre el ejemplo

Buscamos un subarreglo que sume $x = 8$. La **ventana** $[i, j)$ está pintada.



- $i = 0,\ j = 0$: suma $= 0 < 8 \Rightarrow$ avanza j.
- suma $= 2 < 8 \Rightarrow$ avanza j.
- suma $= 5 < 8 \Rightarrow$ avanza j.
- suma $= 7 < 8 \Rightarrow$ avanza j.
- suma $= 12 > 8 \Rightarrow$ avanza i (sale $A[0]$).
- suma $= 10 > 8 \Rightarrow$ avanza i (sale $A[1]$).
- suma $= 7 < 8 \Rightarrow$ avanza j.
- suma $= 8 = x$: **encontrado**.

Código: ventana deslizante

```
bool subarregloSuma(vector<int> &A, long long x,
                    int &li, int &ri) {
    int n = (int)A.size(), i = 0;
    long long suma = 0;
    for (int j = 0; j < n; j++) {
        suma += A[j];
        while (suma > x) {    // me pase, achico x izq.
            suma -= A[i];
            i++;
        }
        if (suma == x) { li = i; ri = j; return true; }
    }
    return false;
}
```

Complejidad: i y j avanzan a lo sumo n veces cada uno $\Rightarrow \mathcal{O}(n)$ en total

Ejemplo: 2SUM

Problema

Dado un arreglo A de n números y un número x, encontrar dos elementos de A que sumen x (o decir que no existen).

Ejemplo: 2SUM

Problema

Dado un arreglo A de n números y un número x, encontrar dos elementos de A que sumen x (o decir que no existen).

Ejemplo

Con $A = [7,\ 10,\ 4,\ 1,\ 9,\ 5,\ 9,\ 6]$:

- $x = 12 \Rightarrow A[0] + A[5] = 7 + 5 = 12$.
- $x = 4 \Rightarrow$ no hay ningún par que sume 4.

Ejemplo: 2SUM

Solución fuerza bruta: probar los $\mathcal{O}(n^2)$ pares posibles.

Ejemplo: 2SUM

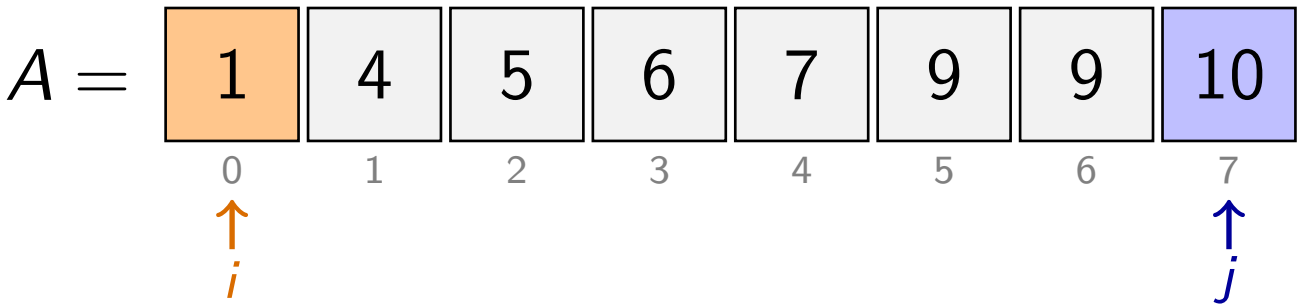
Solución fuerza bruta: probar los $\mathcal{O}(n^2)$ pares posibles.

Observación

El **orden** de los elementos en A no importa para este problema: podemos **ordenarlo** primero sin cambiar la respuesta.

2SUM: variante de punteros opuestos

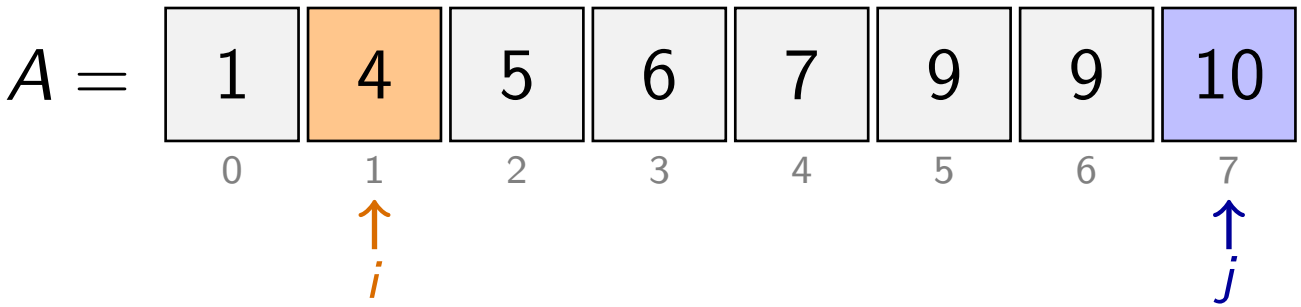
Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.

2SUM: variante de punteros opuestos

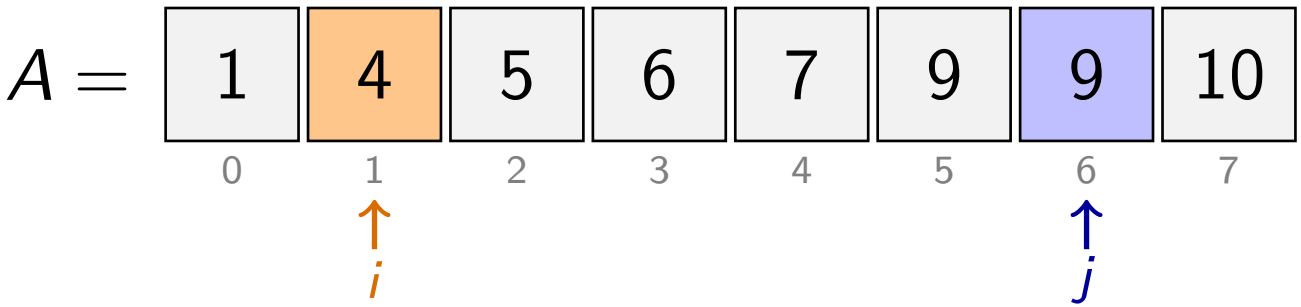
Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.
- $i = 1,\ j = 7$: $4 + 10 = 14 > 12 \Rightarrow$ retrocede j.

2SUM: variante de punteros opuestos

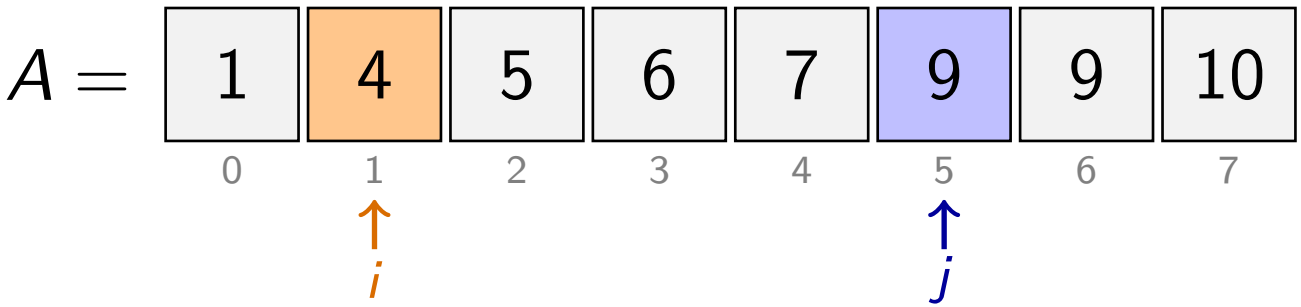
Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.
- $i = 1,\ j = 7$: $4 + 10 = 14 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 6$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.

2SUM: variante de punteros opuestos

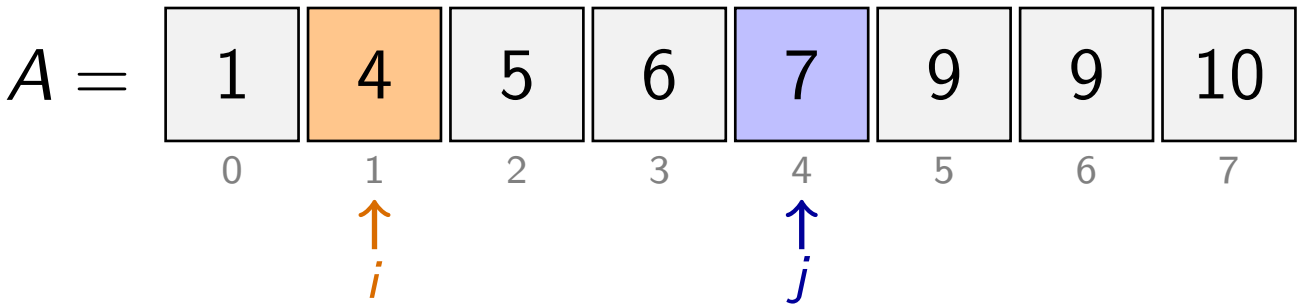
Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.
- $i = 1,\ j = 7$: $4 + 10 = 14 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 6$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 5$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.

2SUM: variante de punteros opuestos

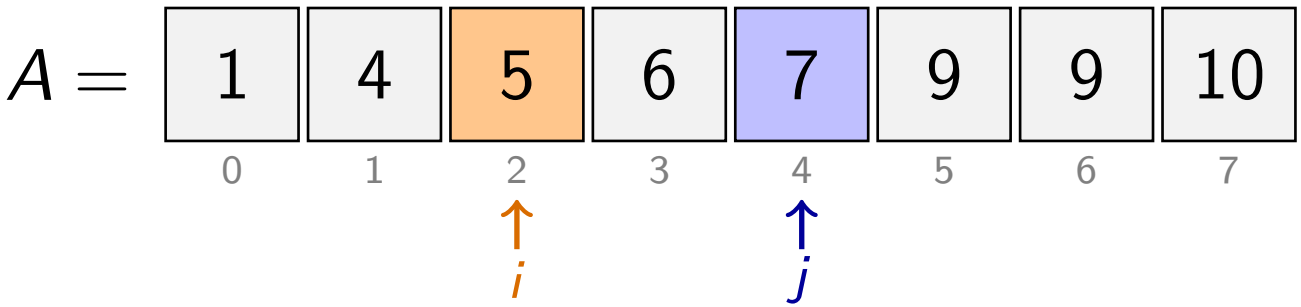
Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.
- $i = 1,\ j = 7$: $4 + 10 = 14 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 6$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 5$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 4$: $4 + 7 = 11 < 12 \Rightarrow$ avanza i.

2SUM: variante de punteros opuestos

Con A ordenado y $x = 12$, usamos **dos punteros que se acercan**: si $A[i] + A[j] < x$ avanzamos i; si $> x$ retrocedemos j; si $= x$, listo.



- $i = 0,\ j = 7$: $1 + 10 = 11 < 12 \Rightarrow$ avanza i.
- $i = 1,\ j = 7$: $4 + 10 = 14 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 6$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 5$: $4 + 9 = 13 > 12 \Rightarrow$ retrocede j.
- $i = 1,\ j = 4$: $4 + 7 = 11 < 12 \Rightarrow$ avanza i.
- $i = 2,\ j = 4$: $5 + 7 = 12 = x$: **encontrado**.

Código: 2SUM

```
bool twoSum(vector<int> A, int x, int &li, int &ri) {
    sort(A.begin(), A.end());
    int i = 0, j = (int)A.size() - 1;
    while (i < j) {
        int s = A[i] + A[j];
        if      (s == x) { li = i; ri = j; return true; }
        else if (s <  x) { i++; }
        else             { j--; }
    }
    return false;
}
```

Complejidad: $\mathcal{O}(n \log n)$ (domina el `sort`; los punteros hacen $\mathcal{O}(n)$)

¿Preguntas?

Contar pares

Problema

Dado un arreglo A de n números y un número x, contar cuántos pares (i, j) con $i < j$ cumplen $A[i] + A[j] \leq x$.

Contar pares

Problema

Dado un arreglo A de n números y un número x, contar cuántos pares (i, j) con $i < j$ cumplen $A[i] + A[j] \leq x$.

Ejemplo

Con $A = [5, 1, 2]$ y $x = 6$: las sumas de pares son $5 + 1 = 6$, $5 + 2 = 7$ y $1 + 2 = 3$. Cumplen ≤ 6 los pares que suman 6 y 3, así que la respuesta es 2.

Contar pares: las dos técnicas

Queremos contar los pares con suma $\leq x$. Sale de dos maneras:

Búsqueda binaria

Ordenamos el arreglo. Después, para cada i, buscamos con **búsqueda binaria** (como al principio) cuántos $A[j]$ con $j > i$ cumplen $A[j] \leq x - A[i]$, y los sumamos. **Complejidad:** $\mathcal{O}(n \log n)$ (sin contar el ordenamiento).

Dos punteros

Como en 2SUM (arreglo **ordenado**): i desde la izquierda, j desde la derecha. Si $A[i] + A[j] \leq x$, todos los del medio también sirven: sumamos $j - i$ y avanzamos i; si no, retrocedemos j. **Complejidad:** $\mathcal{O}(n)$ (sin contar el ordenamiento).

Moraleja

El mismo problema sale con las dos técnicas: ordenar el arreglo y contar, con binaria o con dos punteros.

k-ésima distancia

Problema

Dados n puntos en la recta numérica, y un número k: si listamos **todas** las distancias entre pares de puntos y las ordenamos, ¿cuál queda en el k-ésimo lugar?

k-ésima distancia

Problema

Dados n puntos en la recta numérica, y un número k: si listamos **todas** las distancias entre pares de puntos y las ordenamos, ¿cuál queda en el k-ésimo lugar?

Ejemplo

Puntos $\{-4,\ -1,\ 3,\ 7\}$, $k = 5$.

- Distancias entre todos los pares: $3, 7, 11, 4, 8, 4$.
- Ordenadas: $[3, 4, 4, 7, 8, 11]$. La 5ta es **8**.

k-ésima distancia

Fuerza bruta: calcular las $\mathcal{O}(n^2)$ distancias y ordenarlas $\Rightarrow \mathcal{O}(n^2 \log n)$.
Para $n = 10^5$ ya es demasiado.

k-ésima distancia

Búsqueda binaria sobre la distancia d: definimos

$$P(d): \text{ “la cantidad de pares con distancia } \leq d \text{ es } \geq k\text{”}$$

Es monótono (si hay $\geq k$ pares a distancia $\leq d$, también hay $\geq k$ a distancia $\leq d+1$).

k-ésima distancia

Búsqueda binaria sobre la distancia d: definimos

$$P(d): \text{ “la cantidad de pares con distancia } \leq d \text{ es } \geq k\text{”}$$

Es monótono (si hay $\geq k$ pares a distancia $\leq d$, también hay $\geq k$ a distancia $\leq d+1$).
¿Cómo contamos los pares a distancia $\leq d$? Con los puntos **ordenados**, usando una **ventana deslizante**:

k-ésima distancia

Búsqueda binaria sobre la distancia d: definimos

$$P(d): \text{ “la cantidad de pares con distancia } \leq d \text{ es } \geq k\text{”}$$

Es monótono (si hay $\geq k$ pares a distancia $\leq d$, también hay $\geq k$ a distancia $\leq d+1$).
¿Cómo contamos los pares a distancia $\leq d$? Con los puntos **ordenados**, usando una **ventana deslizante**:

```
long long contarPares(vector<long long> &p, long long d) {
    long long cuenta = 0;
    int i = 0;
    for (int j = 0; j < (int)p.size(); j++) {
        while (p[j] - p[i] > d) { i++; }
        cuenta += (j - i);      // todos los i..j-1 con p[j]
    }
    return cuenta;
}
```

k-ésima distancia

Búsqueda binaria sobre $d \in [0,\ \max(p) - \min(p)]$, evaluando $P(d)$ con la ventana deslizante ($\mathcal{O}(n)$) en cada paso:

$$\underbrace{\mathcal{O}(n \log n)}_{\text{ordenar}} + \underbrace{\mathcal{O}(\log(\text{rango}))}_{\text{búsq. binaria}} \cdot \underbrace{\mathcal{O}(n)}_{\text{ventana}} = \mathcal{O}\big(n(\log n + \log(\text{rango}))\big)$$

La idea para llevarse

Cuando un problema pide "el k-ésimo valor de algo", pensar: *"¿puedo contar cuántos son $\leq d$ más rápido que generándolos todos?"* Si sí, binaria sobre d.

¿Preguntas?

Resumen

Búsqueda binaria

Sirve siempre que exista un predicado $P(k)$ **monótono**. Muy común en problemas de optimización: convertir *"¿cuál es el máximo/mínimo k?"* en *"¿es posible con k?"*.

Ventanas deslizantes

Sirve para evitar recalcular sobre todos los subarreglos, cuando mover un puntero **nunca empeora** lo que ya sabíamos (ej: sumas de positivos, o counts monótonos).

Se combinan

Muchos problemas difíciles son "búsqueda binaria sobre la respuesta, y el predicado se evalúa con una ventana deslizante en $\mathcal{O}(n)$".

Para seguir explorando

Variantes y otros temas que valen la pena:

- **Búsqueda binaria con reales**: cuando la respuesta es un número real, se puede binarizar sobre un intervalo continuo y aproximarla con precisión arbitraria, iterando hasta que el rango activo sea lo suficientemente chico o hasta un número fijo de veces lo suficientemente alto.
- **Búsqueda exponencial**: para listas ordenadas no acotadas. Se avanza a saltos que se duplican en cada paso hasta pasarse del valor buscado, y recién ahí se hace búsqueda binaria dentro de ese rango.
- **Búsqueda ternaria**: para funciones **unimodales** (suben y después bajan, o al revés), encuentra el máximo o mínimo descartando un tercio del rango en cada paso.

Problemas vistos

Los que resolvimos en clase, para implementarlos:

- Pokemon Evolution
- Aggressive Cows
- Capacity To Ship Packages
- Subarray Sums I
- Sum of Two Values
- Number of Pairs — cuenta pares con suma en un rango $[l, r]$, generaliza el que vimos.
- K-th Smallest Pair Distance

Problemas propuestos

Algunos problemas para practicar por su cuenta:

- Finding Square Roots — binaria (fácil).
- Pipeline — binaria (fácil).
- Factory Machines — binaria sobre la respuesta (tiempo mínimo).
- Array Division — binaria sobre la respuesta (minimizar el máximo).
- Mr. Bender and Square — binaria (media).
- Primes on Interval — binaria + ventanas (media).
- Playlist — ventana deslizante + `set` (subarreglo sin repetidos).
- Non-Secret Cypher — ventanas (media/difícil).
- Cubes — binaria + ventanas (media/difícil).
- Sliding Window Maximum — *bonus*: ventana deslizante + `deque` monótono.

Banco de problemas

Para muchísimos más ejercicios (con teoría y ordenados por tema), youkn0wwho Academy:

- youkn0wwho Academy — Binary Search
- youkn0wwho Academy — Two Pointers

Consultas

Cualquier duda pueden preguntarme durante estas semanas o en cualquier momento a través de:

- Email: bersanofede@gmail.com
- Telegram: @Federico_Bersano
- Codeforces: Tainel

Gracias