Problem Solving

Ignacio Muñoz

ignacio.munoz@progcomp.cl

Problemas y técnicas interesantes

3Sum

Dado un arreglo a de tamaño n y un entero s, la tarea es verificar si existe un trío de números en el arreglo cuya suma sea igual al valor objetivo dado.

Entrada	Salida
`6 24` `12 3 4 1 6 9`	YES
`5 50` `1 2 3 4 5`	NO
`5 0` `0 -1 2 -3 1`	YES

¿Cómo resolverlo? — Two Pointers

- Ordenar el arreglo original.
- Iterar por cada `i` y fijamos `a[i]` como el primero número.
- Utilizamos 2 punteros `left = i + 1` y `right = n - 1` y mientras `left < right`:
  - Calcular la suma actual: `a[i] + a[left] + a[right]`
  - Si la suma es igual a `s`, hemos encontrado un trío.
  - Si la suma es menor que `s`, mover el puntero izquierdo (`left++`) para aumentar la suma.
  - Si la suma es mayor que `s`, mover el puntero derecho (`right--`) para reducir la suma.
- Si al final no se encuentra ningún trío, entonces no existe combinación válida.

¿Qué es Two Pointers?

Técnica utilizada sobre **arreglos ordenados** para recorrer eficientemente combinaciones de elementos.

Consiste en usar **dos índices** (`left` , `right`) que se mueven desde extremos opuestos.

Permite resolver problemas como:

- Encontrar pares o tríos con suma específica.
- Verificar si un arreglo es palíndromo.

Vasya y la cadena

Vasya recibió una cadena de longitud n como regalo de cumpleaños. Esta cadena está compuesta de solo letras `'a'` y `'b'`.

Vasya define la belleza de una cadena como la **longitud máxima de una subcadena (consecutiva)** que contiene solo letras iguales.

Puede cambiar como máximo k caracteres de la cadena original (de `'a'` a `'b'` o viceversa). ¿Cuál es la máxima belleza que puede lograr Vasya?

Link: https://codeforces.com/contest/676/problem/C

Vasya y la cadena — Ejemplos

Entrada	Salida
`4 2` `abba`	4
`8 1` `aabaabaa`	5

2^Sort

Link: https://codeforces.com/problemset/problem/1692/G

2^Sort — Cómo resolver

1. Crear un arreglo auxiliar `b[i]` donde:
   - `b[i] = 1` si `a[i] < 2 * a[i+1]`
   - `b[i] = 0` en caso contrario
2. Queremos contar cuántos subarreglos de tamaño `k` en `b[]` tienen suma total = `k` → Esto indica `k` comparaciones válidas seguidas.
3. Aplicar **Sliding Window** sobre `b[]` con ventana de tamaño `k`.

Sliding Window

1. Inicializa una ventana que cubre los primeros `k` elementos.
2. Recorre la secuencia:
    - **Agrega** el nuevo elemento que entra en la ventana.
    - **Quita** el elemento que ya no está en la ventana.
3. Actualiza la respuesta en cada paso según el estado de la ventana.

Subset Sum

Dado un arreglo a de tamaño n (con n hasta 40) y un entero s, determinar si existe un subconjunto de elementos de a cuya suma sea exactamente s.

Entrada	Salida
`5 10` `1 2 3 4 5`	YES
`4 100` `1 2 3 4`	NO
`6 15` `3 34 4 12 5 2`	YES

Subset Sum — ¿Por qué no fuerza bruta?

- Probar todos los subconjuntos cuesta $O(2^n)$.
- Con $n = 40$, 2^{40} es demasiado grande para iterar directamente.
- Pero 2^{20} (la mitad) sí es manejable: ahí entra **Meet in the Middle**.

Meet in the Middle — Idea

1. Dividir el arreglo en dos mitades: `A` (primeros `n/2`) y `B` (el resto).
2. Generar **todas las sumas posibles** de subconjuntos de `A` y de `B` por separado ($O(2^{n/2})$ cada una).
3. Ordenar el arreglo de sumas de `B`.
4. Para cada suma `x` generada desde `A`, buscar con **búsqueda binaria** si existe `s - x` en las sumas de `B`.
5. Si se encuentra para algún `x`, la respuesta es YES.

Meet in the Middle — Complejidad

- Generar subconjuntos de cada mitad: $O(2^{n/2})$.
- Ordenar las sumas de `B`: $O(2^{n/2} \log(2^{n/2}))$.
- Buscar cada suma de `A` en `B`: $O(2^{n/2} \log(2^{n/2}))$.
- Total: $O(2^{n/2} \cdot n)$, mucho mejor que $O(2^n)$.

Ternary Search

Técnica para encontrar el máximo (o mínimo) de una función **unimodal**: una función que crece y luego decrece (o viceversa), sin otros cambios de dirección.

Ejemplo de función unimodal (tiene un único máximo):

```
f(x): 1  4  8  9  7  5  2
x:    0  1  2  3  4  5  6
```

Ternary Search — Paso a paso (1/3)

1. Tenemos un rango `[lo, hi]` donde sabemos que está el máximo.
2. Calculamos dos puntos que dividen el rango en tercios:
   - `m1 = lo + (hi - lo) / 3`
   - `m2 = hi - (hi - lo) / 3`
3. Evaluamos la función en ambos puntos: `f(m1)` y `f(m2)`.

Ternary Search — Paso a paso (2/3)

4. Comparamos los valores:
   - Si `f(m1) < f(m2)`, el máximo **no puede estar** antes de `m1` → descartamos `[lo, m1)` y hacemos `lo = m1`.
   - Si `f(m1) > f(m2)`, el máximo **no puede estar** después de `m2` → descartamos `(m2, hi]` y hacemos `hi = m2`.
5. Repetimos con el nuevo rango `[lo, hi]`, que ahora es más pequeño.

Ternary Search — Paso a paso (3/3)

6. Cuando `hi - lo` es suficientemente pequeño (o menor a una tolerancia), nos detenemos.
7. El máximo se encuentra en el rango final `[lo, hi]`.

Intuición: en cada iteración descartamos un tercio del rango, así que el espacio de búsqueda se reduce geométricamente, igual que en búsqueda binaria.

Ternary Search — Complejidad

- Cada iteración reduce el rango a $2/3$ de su tamaño.
- Para un rango inicial de tamaño n, se necesitan $O(\log n)$ iteraciones.
- Cada iteración evalúa la función 2 veces $\rightarrow$ $O(\log n)$ evaluaciones en total.

Prefix Function (KMP)

Problema: contar ocurrencias de un patrón

Dado un texto S y un patrón T, contar cuántas veces T aparece como substring de S.

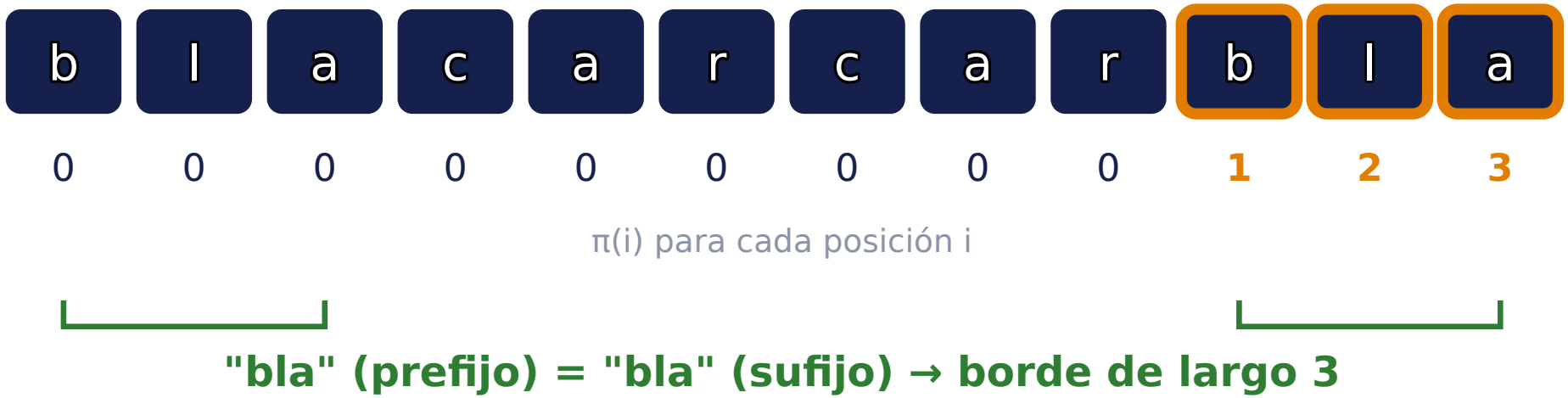
Ejemplo: $S =$ `blacarcarbla`, $T =$ `car` $\rightarrow$ **2** ocurrencias.

A la mala: comparar T contra cada posición de S cuesta $O(|S| \cdot |T|)$; con $|S|, |T| \leq 10^5$, inviable.

Idea de KMP: cuando una comparación falla, no reiniciar desde cero: usar lo que **ya sabemos** que coincide.

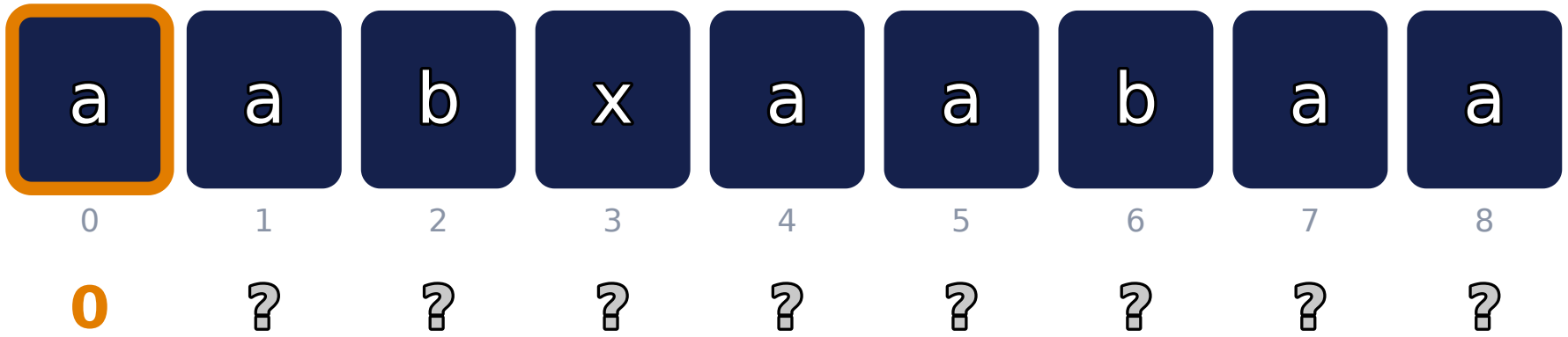
Prefix function: definición

Para un string s, $\pi(i)$ es el largo del mayor **prefijo propio** de $s[0..i]$ que también es **sufijo** de $s[0..i]$.

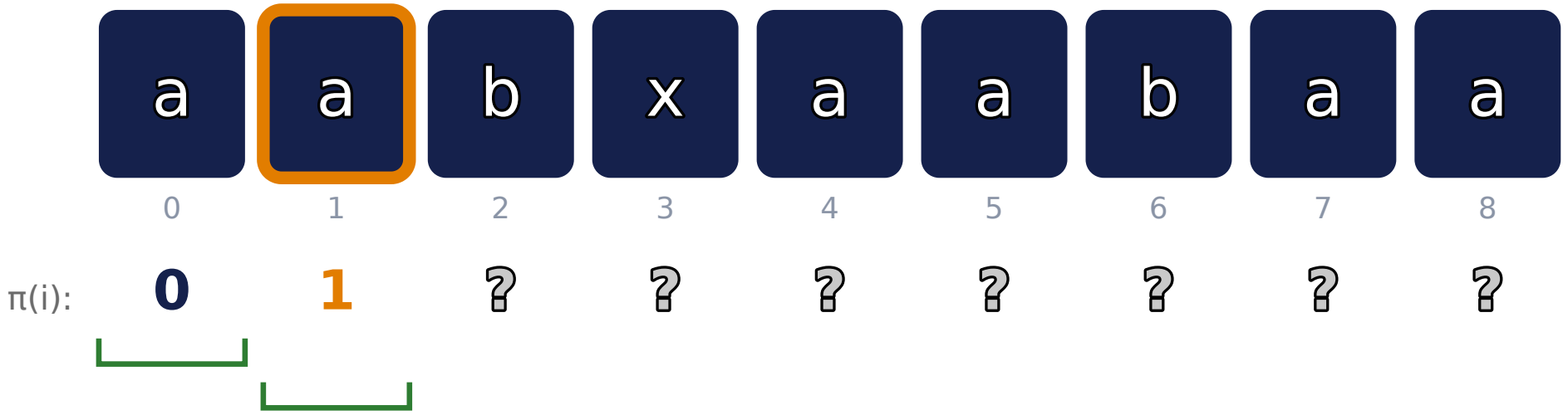


$s =$ `blacarcarbla`: el prefijo `bla` reaparece como sufijo, y π lo detecta al llegar a esas posiciones.

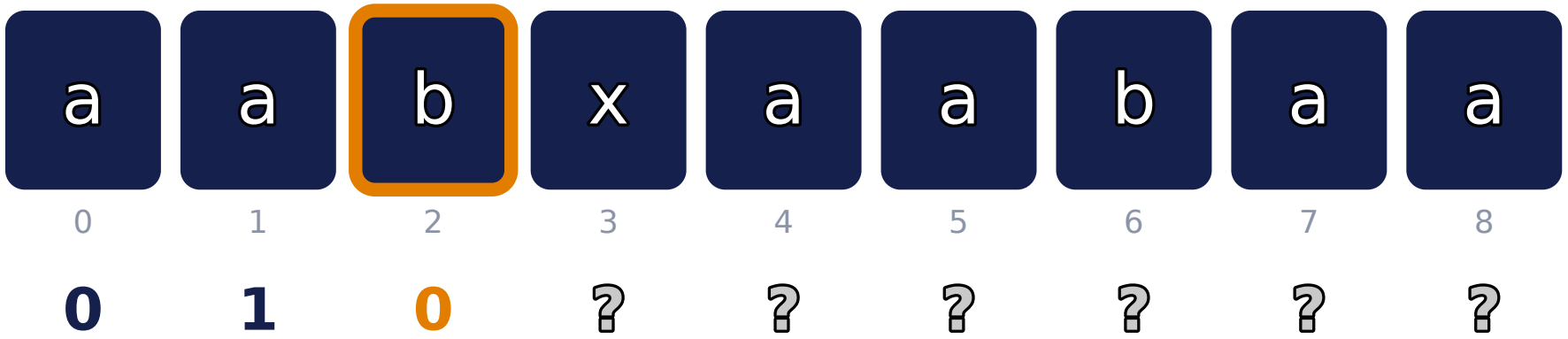
Ejemplo: aabxaabaa



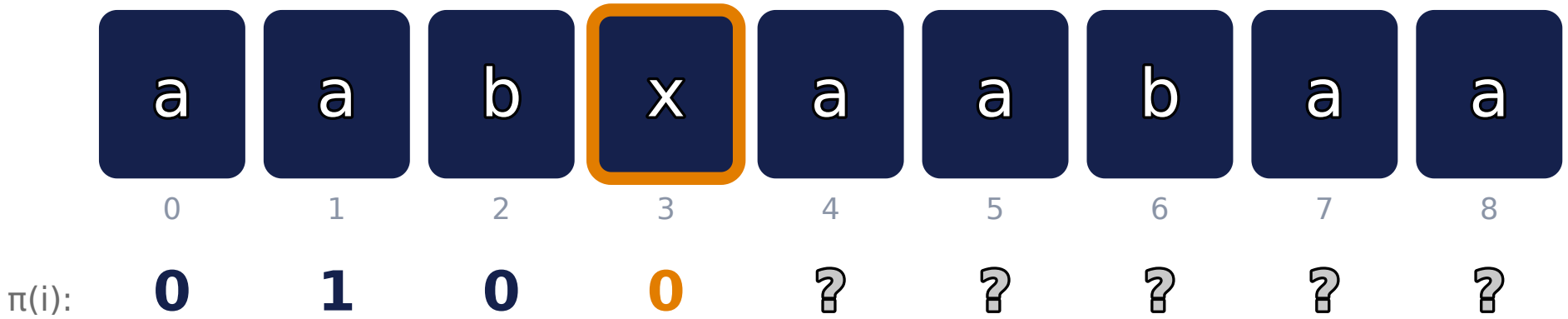
Ejemplo: aabxaabaa



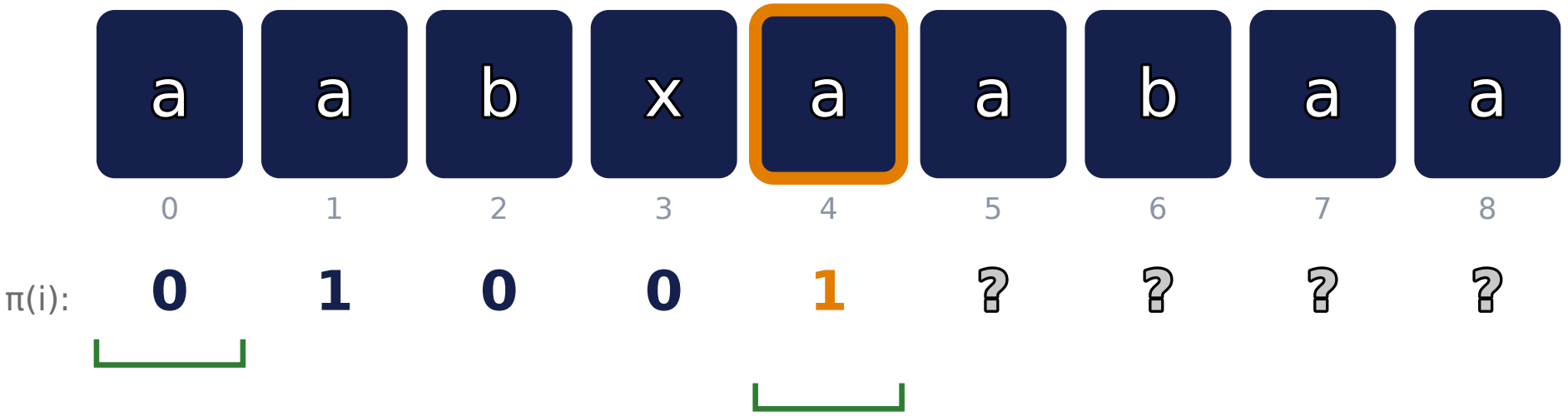
Ejemplo: aabxaabaa



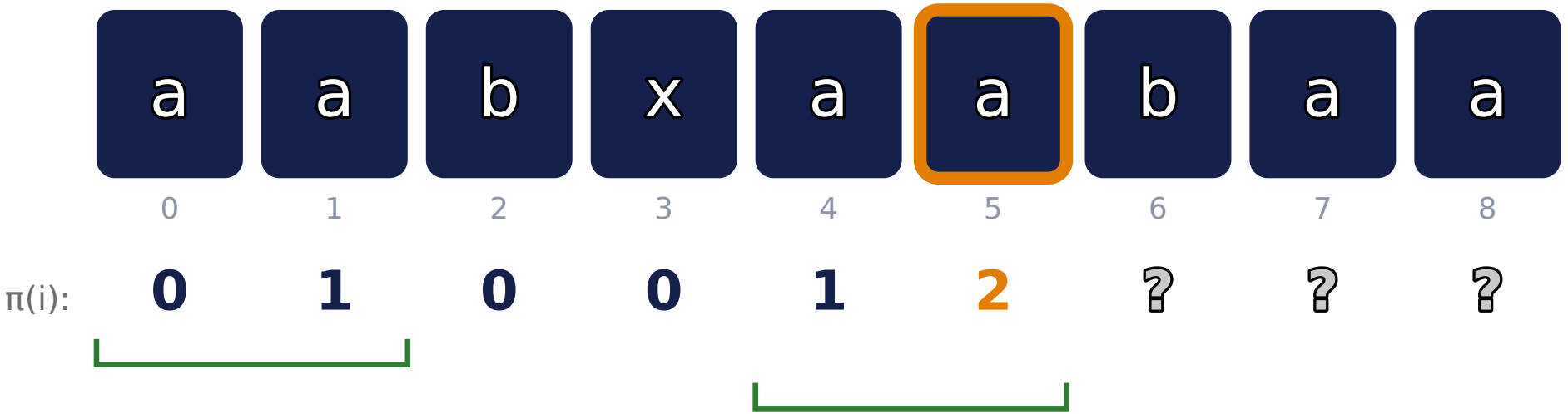
Ejemplo: aabxaabaa



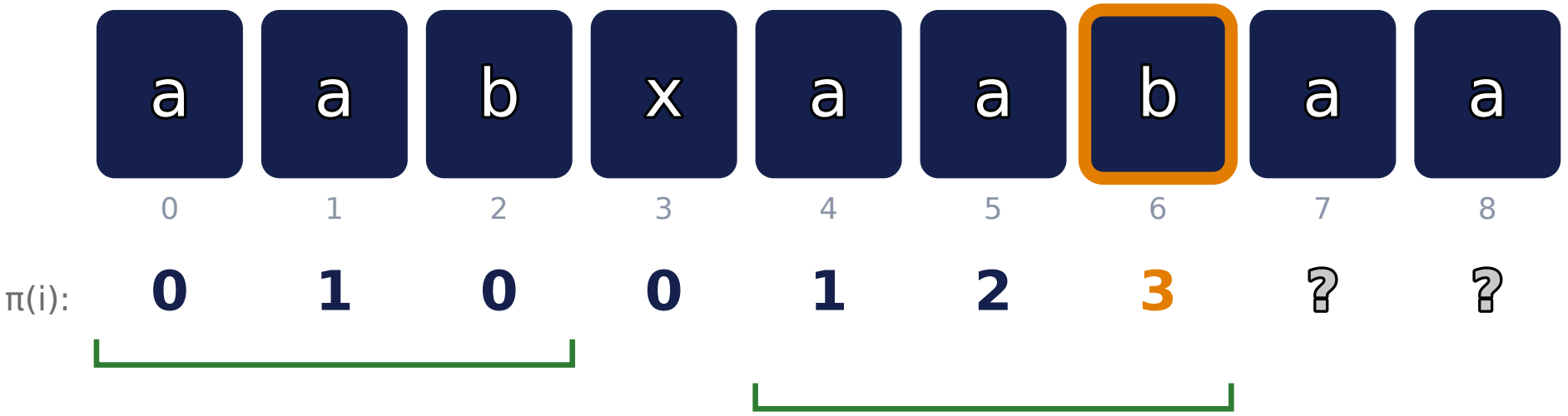
Ejemplo: aabxaabaa



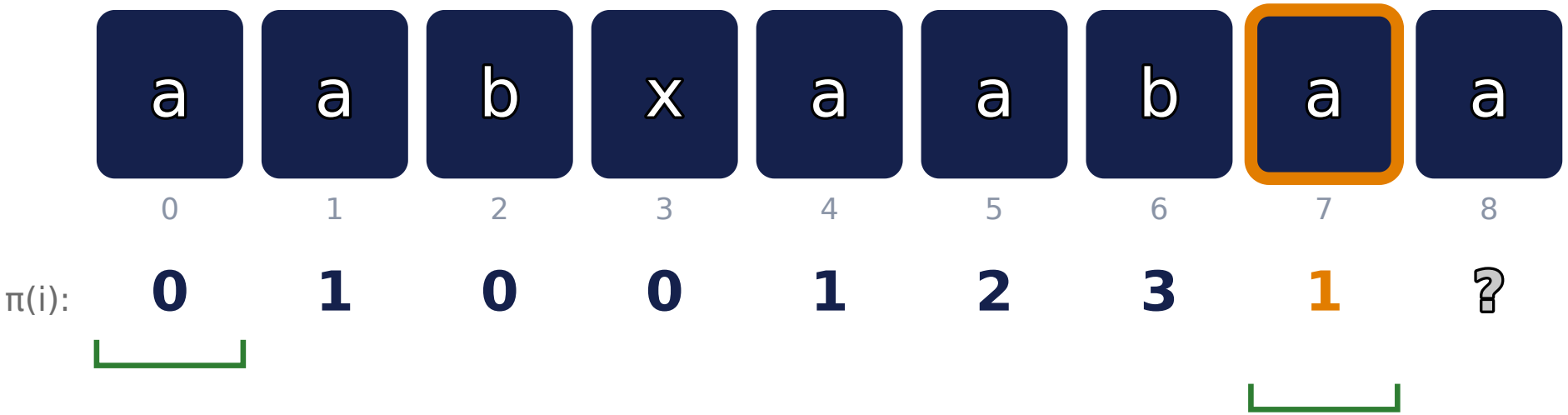
Ejemplo: aabxaabaa



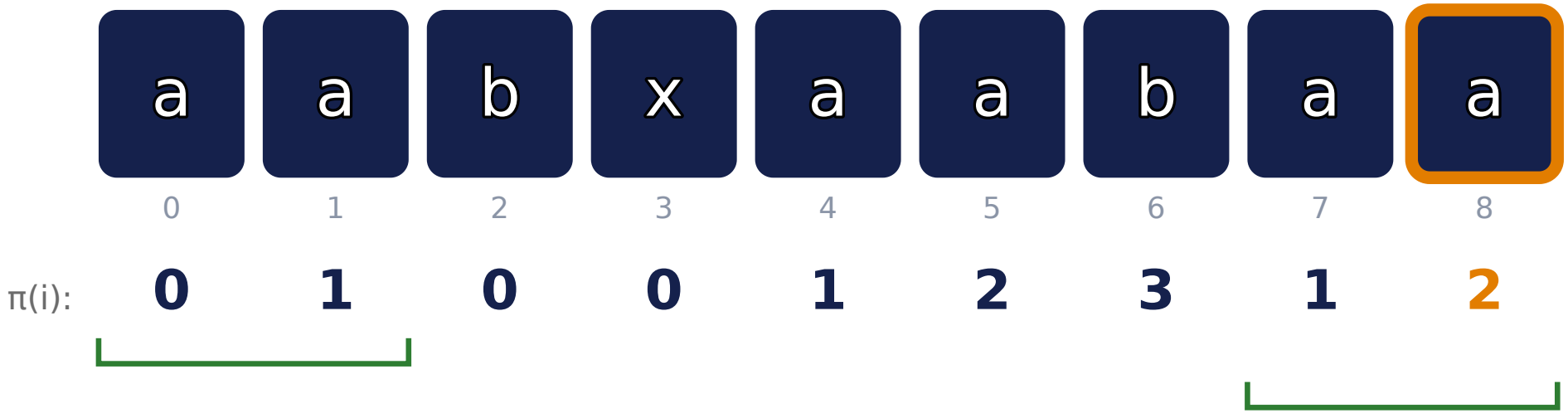
Ejemplo: aabxaabaa



Ejemplo: aabxaabaa

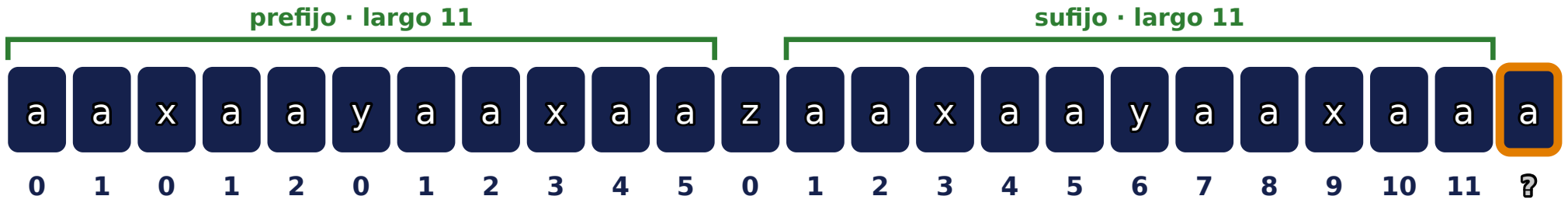


Ejemplo: aabxaabaa

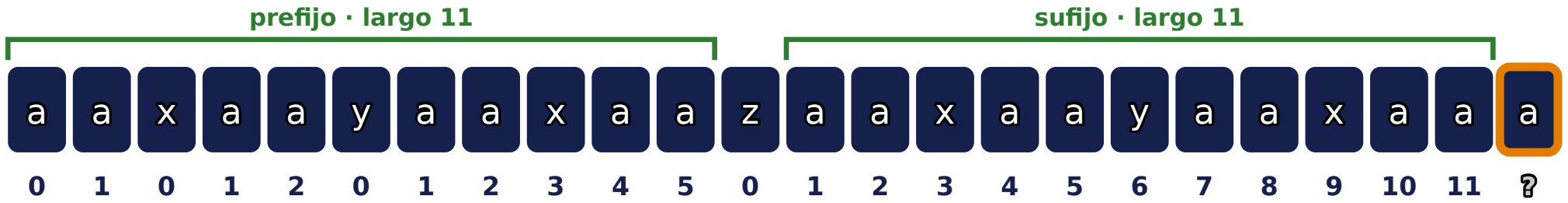


Ejemplo: agregando la última letra

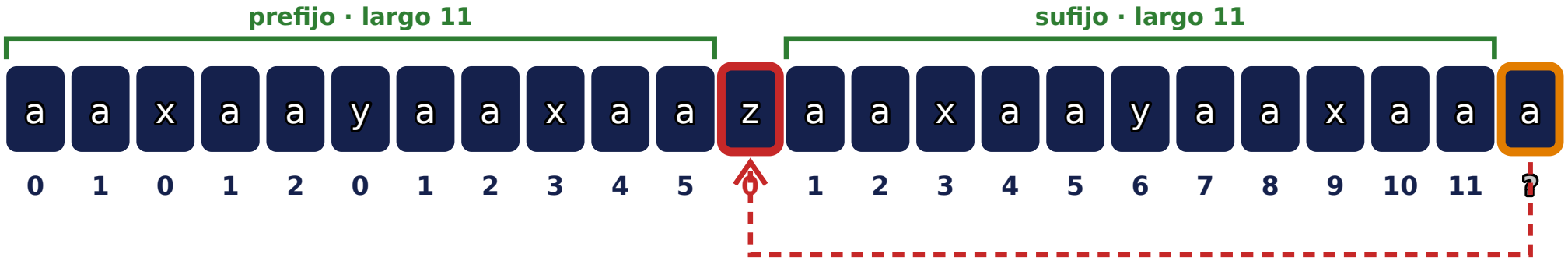
Todo lo demás ya está calculado ($\pi(0)$ a $\pi(22)$). Falta $\pi(23)$.



Candidato: borde de largo 11



¿Coincide? j = 11



Candidato: borde de largo 5

¿Coincide? j = π(10) = 5

Candidato: borde de largo 2

¿Coincide? j = π(4) = 2

Candidato: borde de largo 1

¿Coincide? j = π(1) = 1

$\pi(23) = 1 + 1 = \mathbf{2}$

Cómo se calcula: un puntero que no retrocede

Se mantiene un puntero j (candidato a coincidencia). Al comparar $s[i]$ con $s[j]$:

- Si coinciden, j avanza: la coincidencia crece.
- Si no, j **salta hacia atrás** usando $\pi(j-1)$ (no vuelve a comparar desde cero, reusa un borde ya conocido).

j crece a lo sumo n veces en total (una por iteración de i), así que también puede *decrecer* a lo sumo n veces: el ciclo interno corre $O(n)$ en total, no por iteración.

Código: Prefix Function

```
vector<int> prefix_function(const string &s) {
    int n = s.size();
    vector<int> pi(n, 0);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j])
            j = pi[j - 1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

$O(n)$ amortizado: el `while` interno nunca deshace más de lo que `j` alcanzó a avanzar.

KMP search: buscar el patrón

Truco: concatenar T # S (un separador que no aparece en ninguno) y correr `prefix_function` sobre el resultado. Donde $\pi(i) = |T|$, hay un match.

π(i) = |T| = 3 → match. Posición en S = i − 2·|T| → 3 y 6

E. Compress Words