3. Two Pointers (Dos Punteros)

La técnica de **Two Pointers (Dos Punteros)** es un método algorítmico muy común y eficiente que se utiliza para resolver problemas en estructuras de datos lineales, como arrays o listas enlazadas. Implica usar dos punteros (o índices) que recorren la estructura de datos bajo ciertas condiciones, reduciendo a menudo la complejidad temporal de $\mathcal{O}(N^2)$ a $\mathcal{O}(N)$ o $\mathcal{O}(N \log N)$.

—

0.1. ¿Qué es la Técnica de Two Pointers?

La idea principal es que en lugar de usar un solo bucle anidado (que generalmente lleva a una complejidad cuadrática), se utilizan dos punteros que avanzan a través de la estructura de datos. Estos punteros pueden moverse en la misma dirección o en direcciones opuestas, dependiendo del problema.

Se distinguen principalmente dos patrones:

1. **Punteros que se Mueven en Direcciones Opuestas:**
   - Un puntero se inicializa al principio de la estructura (`left` o `start`).
   - Otro puntero se inicializa al final de la estructura (`right` o `end`).
   - Ambos punteros se mueven hacia el centro hasta que se cruzan o se encuentran.
   - Este patrón es ideal para problemas en arrays **ordenados** donde se busca una pareja, un subconjunto, o se necesita invertir la estructura.
2. **Punteros que se Mueven en la Misma Dirección (Slow/Fast Pointers o Sliding Window con tamaño variable):**
   - Ambos punteros se inicializan al principio de la estructura.
   - Un puntero (`slow` o `i`) avanza a un ritmo más lento.
   - El otro puntero (`fast` o `j`) avanza a un ritmo más rápido.
   - Este patrón es útil para eliminar duplicados, encontrar ciclos en listas enlazadas, comprimir arrays, o para la técnica de **Sliding Window** (cuando el tamaño de la ventana no es fijo).

—

0.2. ¿Cómo Funciona la Técnica de Two Pointers?

El funcionamiento exacto depende del patrón y del problema específico, pero la lógica subyacente es la de **eliminar eficientemente las posibilidades no válidas** en cada paso, reduciendo así el espacio de búsqueda.

Por ejemplo, si buscas una suma en un array ordenado con punteros opuestos:

- Si la suma de los elementos en `left` y `right` es demasiado pequeña, sabes que necesitas un valor más grande, así que incrementas `left`.

- Si la suma es demasiado grande, necesitas un valor más pequeño, así que decrementas `right`.
- Si la suma es la correcta, encontraste una solución.

Este proceso evita la necesidad de verificar cada par posible, que sería $\mathcal{O}(N^2)$.

—

0.3. Ejemplos Clásicos de Two Pointers en C++

Veamos algunos ejemplos comunes para entender mejor esta técnica.

Encontrar un Par con una Suma Específica (Punteros Opuestos)

Dado un array ordenado y un `target_sum`, encuentra si existe un par de números cuya suma sea igual al `target_sum`.

Listing 1: Encontrar un par con suma específica usando Two Pointers

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::sort si el array no est  ordenado

using namespace std; // Incluyendo namespace std como se solicit

// Funci n para encontrar un par con una suma espec fica usando Two Pointers
bool encontrarSumaPar(const vector<int>& arr, int target_sum) {
    int left = 0;
    int right = arr.size() - 1;

    while (left < right) {
        int current_sum = arr[left] + arr[right];
        if (current_sum == target_sum) {
            cout << "Par encontrado: (" << arr[left] << ", " << arr[right] << ")" << endl;
            return true;
        } else if (current_sum < target_sum) {
            left++; // Necesitamos una suma mayor, movemos 'left' a la derecha
        } else { // current_sum > target_sum
            right--; // Necesitamos una suma menor, movemos 'right' a la izquierda
        }
    }
    cout << "No se encontr  ning n par con la suma objetivo." << endl;
```

```
24      return false;
25  }
26
27  int main() {
28      vector<int> numeros_ordenados = {1, 2, 3, 4, 5, 6, 7,
            8, 9};
29      int suma_objetivo = 10;
30
31      cout << "Array: ";
32      for(int num : numeros_ordenados) {
33          cout << num << " ";
34      }
35      cout << "\nBuscando par que sume " << suma_objetivo
            << ":" << endl;
36      encontrarSumaPar(numeros_ordenados, suma_objetivo);
37
38      suma_objetivo = 15;
39      cout << "\nBuscando par que sume " << suma_objetivo
            << ":" << endl;
40      encontrarSumaPar(numeros_ordenados, suma_objetivo);
41      return 0;
42  }
```

Eliminar Duplicados de un Array Ordenado (Punteros en Misma Dirección - Slow/Fast)

Dado un array ordenado, eliminar los duplicados in-place de tal manera que cada elemento único aparezca solo una vez. La función debe devolver la nueva longitud del array.

Listing 2: Eliminar duplicados de un array ordenado in-place

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm> // No estrictamente necesario aqu
       si el array ya est  ordenado
4
5  using namespace std; // Incluyendo namespace std como se
       solicit
6
7  // Funci n para eliminar duplicados in-place
8  int removerDuplicados(vector<int>& nums) {
9      if (nums.empty()) {
10         return 0;
11     }
12
13     int slow = 0; // Puntero lento (donde se escribe el
           siguiente elemento  nico )
14     // Puntero r pido (recorre todos los elementos)
```

```
    for (int fast = 1; fast < nums.size(); ++fast) {
        if (nums[fast] != nums[slow]) {
            slow++; // Mover el puntero lento
            nums[slow] = nums[fast]; // Escribir el nuevo
                elemento  nico
        }
    }
    // La nueva longitud del array ( ndice  0-basado + 1)
    return slow + 1;
}

int main() {
    vector<int> arr1 = {1, 1, 2, 2, 3, 4, 4, 5};
    cout << "Array original: ";
    for(int x : arr1) cout << x << " ";
    cout << endl;

    int nueva_longitud1 = removerDuplicados(arr1);
    cout << "Nueva longitud del array (sin duplicados): "
        << nueva_longitud1 << endl;
    cout << "Array despu s de remover duplicados: ";
    for (int i = 0; i < nueva_longitud1; ++i) {
        cout << arr1[i] << " ";
    }
    cout << endl;

    vector<int> arr2 = {1, 1, 1, 1, 1};
    cout << "\nArray original: ";
    for(int x : arr2) cout << x << " ";
    cout << endl;
    int nueva_longitud2 = removerDuplicados(arr2);
    cout << "Nueva longitud del array (sin duplicados): "
        << nueva_longitud2 << endl;
    cout << "Array despu s de remover duplicados: ";
    for (int i = 0; i < nueva_longitud2; ++i) {
        cout << arr2[i] << " ";
    }
    cout << endl;
    return 0;
}
```

—

0.4. Análisis de Eficiencia de Two Pointers

- **Complejidad Temporal:** En la mayoría de los casos, la técnica de Two Pointers reduce la complejidad a $\mathcal{O}(N)$ (lineal), ya que cada puntero solo recorre la estructura de datos una vez, o en el peor de los casos, se realiza una única pasada por los datos. Esto es una mejora significativa con

respecto a los enfoques de fuerza bruta de $\mathcal{O}(N^2)$.

- **Complejidad Espacial:** Generalmente, $\mathcal{O}(1)$ (constante), ya que solo se utilizan unas pocas variables para los punteros, independientemente del tamaño de la entrada. Esto la convierte en una técnica muy eficiente en cuanto a memoria.

—

0.5. Aplicaciones Comunes de Two Pointers

La técnica de Two Pointers es increíblemente versátil y se utiliza en una amplia gama de problemas:

- Encontrar pares, tripletes o subconjuntos con propiedades específicas (ej., suma objetivo, producto).
- Invertir arrays o cadenas de caracteres.
- Eliminar duplicados en arrays (ordenados o desordenados, con variaciones).
- Mover todos los ceros al final de un array.
- Comprobar si una cadena es un palíndromo.
- Fusionar dos arrays ordenados.
- Encontrar el inicio de un ciclo en una lista enlazada (Floyd's Cycle-Finding Algorithm).

La técnica de Two Pointers es esencial para optimizar soluciones en arrays y listas, transformando algoritmos ineficientes en soluciones rápidas y de bajo consumo de memoria.