5. Sliding Window (Ventana Deslizante)

La técnica de **Sliding Window (Ventana Deslizante)** es un patrón algorítmico utilizado para resolver problemas en arrays o listas donde se requiere analizar un *subconjunto continuo* (una "ventana") de elementos. A medida que se recorre la secuencia, esta ventana se "desliza"sobre los datos, añadiendo elementos por un lado y eliminándolos por el otro, en lugar de recalcular el subconjunto completo en cada paso. Esto permite optimizar la complejidad temporal de muchos problemas de $\mathcal{O}(N^2)$ a $\mathcal{O}(N)$.

—

0.1. ¿Qué es la Técnica de Sliding Window?

La técnica de la Ventana Deslizante implica mantener un rango.º "subarreglo"(la ventana) que se mueve a través de una secuencia de datos (como un array o una cadena de caracteres). En lugar de procesar cada posible subarreglo, que sería muy ineficiente, la ventana se ajusta dinámicamente.

Se distinguen principalmente dos tipos de ventanas:

1. **Ventana de Tamaño Fijo:** La ventana tiene una longitud predefinida que no cambia. Por ejemplo, encontrar la suma máxima de un subarreglo de tamaño K. En este caso, cuando se añade un nuevo elemento al final de la ventana, se elimina un elemento del principio para mantener el tamaño K.

2. **Ventana de Tamaño Variable:** La longitud de la ventana puede crecer o encogerse dependiendo de una o más condiciones. Por ejemplo, encontrar el subarreglo más pequeño cuya suma sea mayor que un valor dado. La ventana se expande hasta que la condición se cumple y luego se contrae desde el inicio hasta que la condición deja de cumplirse.

—

0.2. ¿Cómo Funciona la Técnica de Sliding Window?

La técnica generalmente funciona con dos punteros:

- `start` (o `left_pointer`): Marca el inicio de la ventana.
- `end` (o `right_pointer`): Marca el final de la ventana.

El proceso básico es el siguiente:

1. Inicializar los punteros `start` y `end` (a menudo ambos en 0) y cualquier variable para almacenar resultados o estado de la ventana (ej., suma actual, recuento de caracteres).

2. Expandir la ventana: Mover el puntero `end` hacia adelante, incluyendo nuevos elementos en la ventana y actualizando el estado.

3. Comprobar la condición: Si la ventana cumple una condición específica (ej., la suma es mayor que X, la ventana alcanza un tamaño K), se realiza alguna acción (ej., almacenar el resultado, actualizar un máximo/mínimo).

4. Contraer/Deslizar la ventana: Si la ventana excede una condición (ej., el tamaño fijo K se ha alcanzado, la suma es demasiado grande), mover el puntero `start` hacia adelante, eliminando elementos del principio de la ventana y actualizando el estado. Este paso es crucial para "deslizar"la ventana y mantener la complejidad lineal.

5. Repetir los pasos 2 a 4 hasta que el puntero `end` llegue al final de la secuencia.

—

0.3. Ejemplos Clásicos de Sliding Window en C++

Veamos algunos ejemplos comunes para entender mejor esta técnica.

Suma Máxima de un Subarray de Tamaño Fijo K

Dado un array de números enteros y un entero K, encuentra la suma máxima de cualquier subarray de tamaño K.

Listing 1: Suma máxima de un subarray de tamaño fijo K

```cpp
#include <iostream>
#include <vector>
#include <numeric> // Para std::accumulate (opcional para la suma inicial)
#include <algorithm> // Para std::max

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

int maxSumSubarray(const vector<int>& arr, int k) {
    if (k > arr.size()) {
        return 0; // O lanzar una excepci n, el tama o de la ventana es mayor que el array
    }

    int current_sum = 0;
    // Calcular la suma de la primera ventana (tama o k)
    for (int i = 0; i < k; ++i) {
        current_sum += arr[i];
    }

    int max_sum = current_sum;

    // Deslizar la ventana
```

```
    for (int i = k; i < arr.size(); ++i) {
        // A adir el nuevo elemento a la ventana
        current_sum += arr[i];
        // Quitar el elemento que sale de la ventana
        current_sum -= arr[i - k];

        // Actualizar la suma m xima
        max_sum = max(max_sum, current_sum);
    }
    return max_sum;
}

int main() {
    vector<int> nums = {1, 4, 2, 10, 23, 3, 1, 0, 20};
    int k = 4;
    cout << "Array: ";
    for (int x : nums) {
        cout << x << " ";
    }
    cout << "\nTama o de ventana (K): " << k << endl;
    cout << "Suma m xima del subarray de tama o " << k
        << ": " << maxSumSubarray(nums, k) << endl;
    // Subarray {2, 10, 23, 3} tiene suma 38
    return 0;
}
```

Subarray Más Pequeño con Suma Mayor o Igual que un Valor Objetivo

Dado un array de enteros positivos y un entero 'target$_sum$', *encuentralalongituddelsubarraycontinuomáspe*

Listing 2: Subarray más pequeño con suma ¿= valor objetivo

```
#include <iostream>
#include <vector>
#include <algorithm> // Para std::min
#include <limits>    // Para std::numeric_limits

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

int minSubArrayLen(int target_sum, const vector<int>&
    nums) {
    int n = nums.size();
    int min_length = numeric_limits<int>::max(); //
        Inicializar con un valor muy grande
    int current_sum = 0;
    int window_start = 0;

```

```
    for (int window_end = 0; window_end < n; ++window_end) {
        current_sum += nums[window_end]; // Expandir la ventana

        // Contraer la ventana mientras la condici n se cumple
        while (current_sum >= target_sum) {
            min_length = min(min_length, window_end - window_start + 1); // Actualizar longitud m nima
            current_sum -= nums[window_start]; // Quitar elemento del inicio
            window_start++; // Deslizar el inicio de la ventana
        }
    }
    return (min_length == numeric_limits<int>::max()) ? 0 : min_length;
}

int main() {
    vector<int> nums1 = {2, 3, 1, 2, 4, 3};
    int target1 = 7;
    cout << "Array: ";
    for (int x : nums1) {
        cout << x << " ";
    }
    cout << "\nSuma objetivo: " << target1 << endl;
    cout << "Longitud del subarray m s peque o: " << minSubArrayLen(target1, nums1) << endl; // Salida: 2 (para [4,3])

    vector<int> nums2 = {1, 1, 1, 1, 1};
    int target2 = 10;
    cout << "\nArray: ";
    for (int x : nums2) {
        cout << x << " ";
    }
    cout << "\nSuma objetivo: " << target2 << endl;
    cout << "Longitud del subarray m s peque o: " << minSubArrayLen(target2, nums2) << endl; // Salida: 0

    return 0;
}
```

—

0.4. Análisis de Eficiencia de Sliding Window

- **Complejidad Temporal:** $\mathcal{O}(N)$, donde N es el tamaño de la secuencia. Aunque hay un bucle anidado (el `while` dentro del `for` en la ventana variable), cada elemento de la secuencia es visitado y procesado por el puntero `end` una vez y por el puntero `start` a lo sumo una vez. Esto resulta en una complejidad lineal.
- **Complejidad Espacial:** $\mathcal{O}(1)$ (constante), ya que solo se utilizan unas pocas variables para los punteros y el estado de la ventana, independientemente del tamaño de la entrada.

La eficiencia lineal de la Ventana Deslizante es una de sus mayores ventajas, lo que la hace ideal para problemas con grandes conjuntos de datos.

—

0.5. Aplicaciones Comunes de Sliding Window

La técnica de Sliding Window es extremadamente útil para resolver problemas en arrays o cadenas que involucran:

- Encontrar subarrays/subcadenas con sumas, productos o promedios específicos (máximo/mínimo).
- Encontrar el subarray/subcadena más largo/pequeño que cumple una condición (ej., con K caracteres distintos, sin caracteres repetidos).
- Subcadenas que son anagramas o que contienen todas las letras de otra palabra.
- Problemas de conteo que involucran subconjuntos continuos.

La Ventana Deslizante es un patrón fundamental para optimizar algoritmos que operan sobre segmentos contiguos de datos.