Range Queries

by Alan Martinez

[ΓΩ5]

¿Qué son las Range Queries?

- **Definición**: Consultas que permiten obtener información de un subarreglo o submatriz de una estructura de datos.
- **Aplicaciones comunes**:
  - **Sumas de subarreglos**.
  - **Consultas de mínimos/máximos**.
  - **Problemas de conteo** (como número de elementos en un rango).
- **Objetivo de esta presentación**:
  - Conocer diferentes técnicas de Range Queries y sus aplicaciones.
  - Entender la complejidad de cada técnica y cuándo utilizarlas.

Static Array Queries (Prefix Sums)

- **Objetivo**: Responder consultas de suma de subarreglos en tiempo constante.
- **Concepto**:
  - Calcular un array de sumas acumuladas llamado `prefixSum` donde `prefixSum[i] = A[0] + A[1] + ... + A[i]`.
  - Esto permite que cualquier suma de subarreglo `[L, R]` sea calculada como `prefixSum[R] - prefixSum[L-1]`.
- **Ejemplo**:
  - Dado un arreglo `A = [1, 2, 3, 4, 5]`, el arreglo de sumas acumuladas es `prefixSum = [1, 3, 6, 10, 15]`.

```cpp
vector<int> prefixSum(const vector<int>& arr) {
    int n = arr.size();
    vector<int> prefix(n);
    prefix[0] = arr[0];
    for (int i = 1; i < n; i++) {
        prefix[i] = prefix[i-1] + arr[i];
    }
    return prefix;
}

// Ejemplo de consulta
int rangeSum(int L, int R, const vector<int>& prefix) {
    if (L == 0) return prefix[R];
    return prefix[R] - prefix[L - 1];
}
```

- **Construcción**: O(n)
- **Consulta**: O(1)

Range Queries with Sweep Line

- **Aplicación**: Resolver problemas en una **grilla 2D** mediante consultas de rango 1D.
- **Concepto**:
  - Usar un enfoque de "línea de barrido" para reducir consultas de 2D a consultas de rango en 1D.
  - Cada fila (o columna) se convierte en un evento que se puede procesar usando técnicas de rango 1D.
- **Ejemplo de problema**:
  - Contar puntos dentro de un rectángulo en un plano 2D.

```cpp
struct Event {
    int x, y1, y2, type; // x es el punto, y1 y y2 son el rango vertical
};

// Procesamiento de eventos
void processSweepLine(vector<Event>& events) {
    sort(events.begin(), events.end(), [](Event a, Event b) { return a.x < b.x; });
    // Implementar el procesamiento de eventos aquí
}
```

- **Ordenar eventos**: $O(n \log n)$
- **Procesar eventos**: depende de la estructura de datos usada en `y` (e.g., Segment Tree para $O(\log n)$).

2D Range Queries (Sumas y mínimos en submatrices)

- **Objetivo**: Consultar sumas o valores mínimos/máximos en áreas 2D.
- **Métodos**:
  - **Suma acumulativa en 2D**: Preprocesar la matriz con una matriz de sumas acumuladas.
  - **Segment Tree 2D**: Utilizar Segment Tree para manejar consultas más complejas.
- **Ejemplo**:
  - Para consultar la suma en un rectángulo `[x1, y1]` a `[x2, y2]` en tiempo constante.

```cpp
vector<vector<int>> prefix2D(const vector<vector<int>>& grid) {
    int n = grid.size(), m = grid[0].size();
    vector<vector<int>> prefix(n, vector<int>(m));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            prefix[i][j] = grid[i][j];
            if (i > 0) prefix[i][j] += prefix[i-1][j];
            if (j > 0) prefix[i][j] += prefix[i][j-1];
            if (i > 0 && j > 0) prefix[i][j] -= prefix[i-1][j-1];
        }
    }
    return prefix;
}
```

- **Construcción**: O(n * m)
- **Consulta**: O(1)

Divide & Conquer - Static Range Queries (SRQ)

- **Concepto**: Usar **Divide & Conquer** para resolver consultas de rango en arreglos estáticos.
- **Aplicación**:
  - Divide el arreglo en subarreglos pequeños y resuelve cada subarreglo recursivamente.
- **Complejidad**: Similar a $O(n \log n)$.

Ejemplo: Maximum Subarray usando Divide & Conquer

Problema

Dado un arreglo `arr`, encuentra el subarreglo (una sección continua del arreglo) que tiene la suma más alta.

Enfoque con Divide & Conquer

El problema se puede dividir de la siguiente manera:

1. Encuentra la suma máxima en el subarreglo izquierdo.
2. Encuentra la suma máxima en el subarreglo derecho.
3. Encuentra la suma máxima en un subarreglo que cruza la mitad (parte del subarreglo izquierdo y parte del derecho).

Luego, el resultado será el máximo de los tres valores obtenidos. Esto se realiza recursivamente en subarreglos hasta llegar a arreglos de un solo elemento.

```cpp
int maxCrossingSum(const vector<int>& arr, int left, int mid, int right) {
    int leftSum = INT_MIN, rightSum = INT_MIN; int sum = 0;
    for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);
    }
    sum = 0;
    for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        rightSum = max(rightSum, sum);
    }
    return leftSum + rightSum;
}
int maxSubArraySum(const vector<int>& arr, int left, int right) {
    if (left == right) return arr[left];
    int mid = (left + right) / 2;
    return max({maxSubArraySum(arr, left, mid), maxSubArraySum(arr, mid + 1, right),
    maxCrossingSum(arr, left, mid, right)});
} // (O(n log n))
```

Funcion Main()

```
int main() {
    vector<int> arr = {2, -5, 6, -2, 3, 1, 5, -3};
    int n = arr.size();
    int maxSum = maxSubArraySum(arr, 0, n - 1);
    cout << "La suma máxima del subarreglo es: " << maxSum << endl;
    return 0;
}
```

Square Root Decomposition

- **Objetivo**: Manejar consultas de subarreglos en tiempo `O(sqrt(n))`.
- **Método**:
  - Dividir el arreglo en bloques de tamaño `sqrt(n)`.
  - Precalcular la respuesta en cada bloque.
- **Aplicación**: Útil para problemas donde el arreglo es dinámico y hay múltiples actualizaciones.

```cpp
vector<int> sqrtDecomposition(const vector<int>& arr) {
    int n = arr.size();
    int blockSize = sqrt(n) + 1;
    vector<int> blocks(blockSize, 0);
    for (int i = 0; i < n; i++) {
        blocks[i / blockSize] += arr[i];
    }
    return blocks;
}

int rangeQuery(int L, int R, const vector<int>& arr, const vector<int>& blocks) {
    int blockSize = sqrt(arr.size()) + 1;
    int sum = 0;
    while (L <= R && L % blockSize != 0) sum += arr[L++];
    while (L + blockSize <= R) sum += blocks[L / blockSize], L += blockSize;
    while (L <= R) sum += arr[L++];
    return sum;
} // Complejidad O(sqrt(n))
```

Problemas

- **433B** Kuriyama Mirai's Stones ⮭
- **816B** Karen and Coffee ⮭

Referencias

- Mridul, G. (2024). *Range Operations and Lazy Propagation for Competitive Programming*. Recuperado de https://www.geeksforgeeks.org/range-operations-and-lazy-propagation-for-competitive-programming/
- Jellyman102. (2020). *An Introduction to Range Query Problems [C++]*. Recuperado de https://codeforces.com/blog/entry/79202
- Arora, K. (2021). *Range query data structures*. Recuperado de https://codeforces.com/blog/entry/89730
- Qi, B. et al. (s.f.). *Range Queries with Sweep Line*. https://usaco.guide/plat/range-sweep?lang=cpp
- Laaksonen, A. (2018). *Competitive Programmer's Handbook*. Recuperado de https://cses.fi/book/book.pdf