# **Tutorial 5: Range Minimum Query (RMQ)**

## **1. Ce este RMQ?**

**Range Minimum Query (RMQ)** este o problemă clasică de structuri de date:

* **Dat un tablou A cu n elemente, răspunde rapid la întrebări de forma:** **Care este valoarea minimă din intervalul [L, R]?**

## **2. Soluții naive și eficiente**

* **Soluție naivă:** Pentru fiecare query, parcurgi intervalul [L, R] și găsești minimul.  
   ⮕ Timp per query: **O(R - L + 1)** (lent pentru multe întrebări).
* **Soluții eficiente:** vrem **preprocesare** pentru a răspunde în timp mai bun:  
  + **Sparse Table:** O(n log n) timp de preprocesare, O(1) timp pe query.
  + **Segment Tree:** O(n) sau O(n log n) construcție, O(log n) timp pe query (mai flexibil dacă ai și update-uri).

## **3. Sparse Table pentru RMQ (caz fără update-uri)**

### **3.1 Ideea**

Precalculăm minimele pentru intervale de lungimi puteri ale lui 2.

Exemplu: pentru fiecare i și k, salvăm:

ST[i][k] = minimul pe intervalul [i, i + 2^k - 1]

Apoi pentru un query [L, R]:

* găsim cel mai mare k cu 2^k <= R - L + 1,
* răspundem în O(1) ca fiind min(ST[L][k], ST[R - 2^k + 1][k]).

### **3.2 Cod**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

const int N = 1e5; // capacitate maximă

int A[N];

int ST[N][20]; // sparse table

int n;

// Preprocesare

void build() {

for (int i = 0; i < n; i++)

ST[i][0] = A[i];

for (int k = 1; (1 << k) <= n; k++) {

for (int i = 0; i + (1 << k) <= n; i++) {

ST[i][k] = min(ST[i][k-1], ST[i + (1 << (k-1))][k-1]);

}

}

}

// Query RMQ

int query(int L, int R) {

int k = log2(R - L + 1);

return min(ST[L][k], ST[R - (1 << k) + 1][k]);

}

int main() {

cin >> n;

for (int i = 0; i < n; i++) cin >> A[i];

build();

int q;

cin >> q;

while (q--) {

int L, R;

cin >> L >> R;

cout << query(L, R) << '\n';

}

}

### **3.3 Complexități**

* **Preprocesare:** O(n log n)
* **Query:** O(1)

## **4. Segment Tree pentru RMQ (pentru update-uri)**

### **4.1 Ideea**

Construim un arbore binar, fiecare nod reține minimul într-un interval.

* **Răspuns:** urcăm/descindem în arbore în O(log n).
* **Update:** modificăm un element și actualizăm minimurile tot în O(log n).

### **4.2 Cod**

#include <iostream>

#include <vector>

using namespace std;

const int N = 1e5;

int A[N];

int tree[4\*N];

int n;

// Construire Segment Tree

void build(int node, int l, int r) {

if (l == r) {

tree[node] = A[l];

} else {

int mid = (l + r) / 2;

build(2\*node, l, mid);

build(2\*node+1, mid+1, r);

tree[node] = min(tree[2\*node], tree[2\*node+1]);

}

}

// Query [L, R]

int query(int node, int l, int r, int L, int R) {

if (L > r || R < l) return INT\_MAX;

if (L <= l && r <= R) return tree[node];

int mid = (l + r) / 2;

return min(query(2\*node, l, mid, L, R),

query(2\*node+1, mid+1, r, L, R));

}

// Update A[pos] = val

void update(int node, int l, int r, int pos, int val) {

if (l == r) {

tree[node] = val;

} else {

int mid = (l + r) / 2;

if (pos <= mid)

update(2\*node, l, mid, pos, val);

else

update(2\*node+1, mid+1, r, pos, val);

tree[node] = min(tree[2\*node], tree[2\*node+1]);

}

}

int main() {

cin >> n;

for (int i = 0; i < n; i++) cin >> A[i];

build(1, 0, n-1);

int q;

cin >> q;

while (q--) {

int type;

cin >> type;

if (type == 1) { // query

int L, R;

cin >> L >> R;

cout << query(1, 0, n-1, L, R) << '\n';

} else { // update

int pos, val;

cin >> pos >> val;

update(1, 0, n-1, pos, val);

}

}

}

### **4.3 Complexități**

* **Preprocesare:** O(n)
* **Query:** O(log n)
* **Update:** O(log n)

## **5. Când folosești fiecare varianta?**

| **Problemă** | **Soluție recomandată** |
| --- | --- |
| Doar query-uri fixe, fără modificări | **Sparse Table** |
| Query-uri + modificări dinamice | **Segment Tree** |

## **6. Cod… :)**