RMQ

Propun să plecăm de la următoarea problema: se dă un șir de n numere naturale și m query-uri de forma să calculeze minimul dintre numerele aflate pe pozițiile . Cum putem rezolva problema în **timp** minim?

**Varianta 1 (brut)**

Calculăm brut minimul pe intervalul [l,r]. Această abordare are complexitate-timp O(N) pe query. Având M astfel de query-uri, ajungem la o complexitate-timp finală și cea de spațiu auxiliar

void VariantaBrut()  
{  
 while(M--)  
 {  
 cin >> leftQ >> rightQ;  
 int Min = INT\_MAX; // constanta predefinita in C++, 2^31 - 1  
  
 for(int i = leftQ; i <= rightQ; ++i)  
 Min = min(Min, A[i]);  
  
 cout << Min << '\n';  
 }  
}

**Varianta 2 (precalculare cu matrice și calcul de minim brut)**

Se observă că nu avem operații de update care să ne modifice

structura șirului, lucru care ne permite să ne gândim la o metodă de precalculare (memoizare, cache-uire a calculelor).

Concret, pentru fiecare element de la primul la ultimul, putem calcula minimul dintre el și, pe rând, următorii, stocând rezultatul într-o matrice de tipul .

Presupunând că generăm toate perechile (left, right) cu left <= right și calculăm minimul direct pe intervalele respective, o să avem o complexitate-timp pentru precalculare și pe fiecare dintre cele M query-uri, ceea ce ne aduce în final la o complexitate-timp de . Complexitatea-spațiu auxiliar a soluției este , fapt datorat stocării matricei de precalculare.

void VariantapreCalcMatriceFaraMemoizareMinim()  
{  
 vector<vector<int>> preCalc(N + 1, vector<int> (N + 1));  
 //echivalent cu a scrie preCalc[N + 1][N + 1]. Se umple automat cu zerouri  
  
 for(int i = 1; i <= N; ++i)  
 for(int j = i + 1; j <= N; ++j)  
 {  
 int Min = INT\_MAX;  
 for(int k = i; k <= j; ++k)  
 Min = min(Min, A[k]);  
 preCalc[i][j] = Min;  
 }  
  
 while(M--)  
 {  
 cin >> leftQ >> rightQ;  
 cout << preCalc[leftQ][rightQ] << '\n';  
 }  
}

**Varianta 2.5 (precalculare cu matrice și calcul de minim optim)**

Soluția precedentă poate fi îmbunătățită prin modificarea modului de calculare al minimului. Deoarece minimul este o funcție agregată (se aplică pe o mulțime de valori și returnează o valoare) ceea ce ne permite să reformulăm, de exemplu, . Complexitatea-timp scade la .

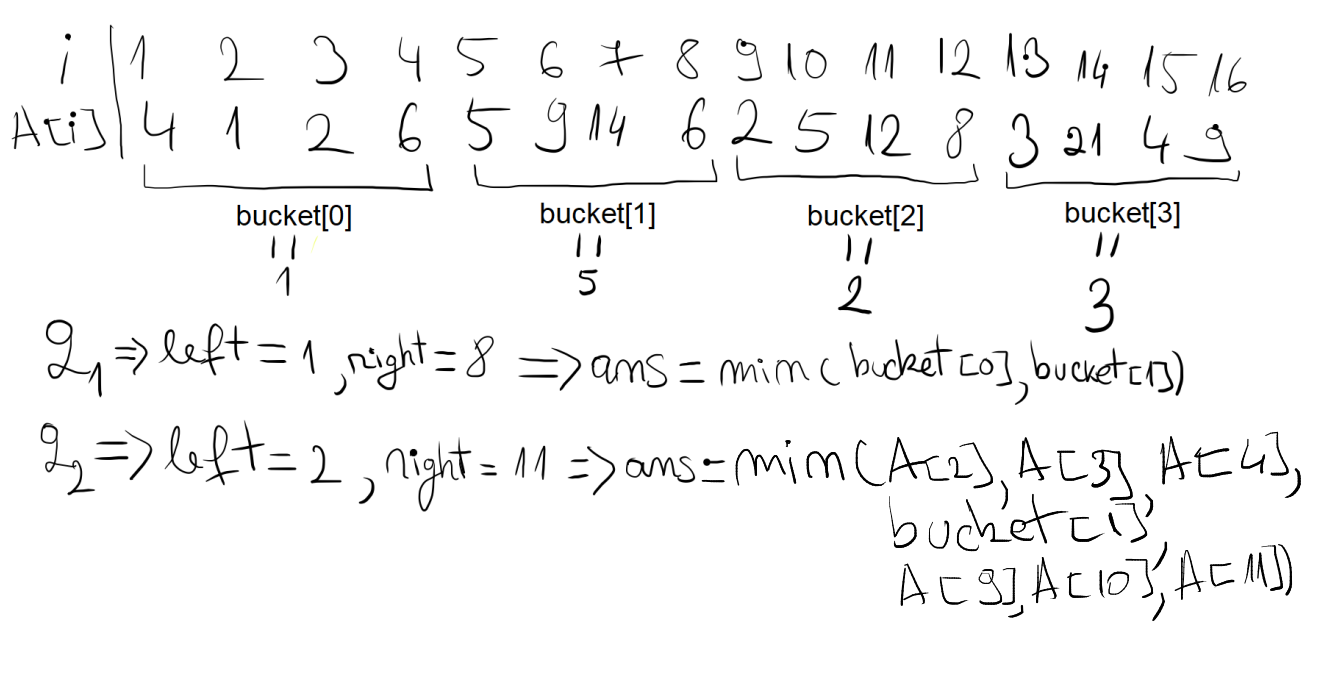
void VariantapreCalcMatriceCuMemoizareMinim()  
{  
 vector<vector<int>> preCalc(N + 1, vector<int> (N + 1));  
 //echivalent cu a scrie preCalc[N + 1][N + 1]. Se umple automat cu zerouri   
  
 for(int i = 1; i <= N; ++i)  
 {  
 preCalc[i][i] = A[i];  
 for(int j = i + 1; j <= N; ++j)  
 preCalc[i][j] = min(preCalc[i][j - 1], A[j]);  
 }  
  
 while(M--)  
 {  
 cin >> leftQ >> rightQ;  
 cout << preCalc[leftQ][rightQ] << '\n';  
 }  
}

**Varianta 3 (Sqrt Decomposition/Șmenul lui Batog)**

O altă metodă de a face preprocesare este Sqrt Decomposition care implică să spargem șirul în blocuri de mărime și să reținem în ele query-urile precalculate pentru a le reuni într-un query mai mare.

În figura (sperăm, inteligibilă) de mai jos vom ilustra un exemplu pe problema noastră. Se obervă două cazuri de evaluare, anume:

1. Capetele intervalului de query se suprapun perfect cu bucket-urile noastre, fiind nevoie să facem doar o reuniune. Complexitatea-timp pentru aceasta este (Reiese din faptul că avem bucket-uri și nu putem parcurge mai mult de atât.)
2. Capetele intervalului de query nu se suprapun bine cu bucket-urile și trebuie să calculăm brut porțiunile respective, reunindu-le apoi cu rezultatele precalculate din bucket-uri. În cel mai rău caz, am putea să parcurgem două porțiuni de mărime și să parcurgem blocuri. Complexitatea-timp ar fi egala cu:



Complexitatea-timp pentru preprocesare va fi , iar cea pentru un query ceea ce ne va aduce la. Complexitatea-spațiu auxiliară va fi fiindcă vectorul de blocuri are doar elemente.

void VariantaSqrtDecomposition()  
{  
 int blockSize = sqrt(N);  
 int blocks[blockSize] = {INT\_MAX};  
  
 for(int i = 1; i <= N; ++i)  
 {  
 int pozBlock = (i - 1) / blockSize; // calculam asa din cauza indexarii de la 1 ca sa nu punem  
 // elementele pe pozitii multiplu de blockSize in urmatorul bloc  
 blocks[pozBlock] = min(blocks[pozBlock], A[i]);  
 }  
  
 while(M--)  
 {  
 cin >> leftQ >> rightQ;  
  
 int ans = INT\_MAX;  
  
 while(leftQ <= rightQ && (leftQ - 1) % blockSize)  
 ans = min(ans, A[leftQ++]);  
 while(leftQ + blockSize - 1 <= rightQ)  
 {  
 ans = min(ans, blocks[leftQ / blockSize]);  
 leftQ += blockSize;  
 }  
 while(leftQ <= rightQ)  
 ans = min(ans, A[leftQ++]);  
  
 cout << ans << '\n';  
 }  
}

**Varianta 4 (Sparse Table)**

Sparse table-urile sunt o unealtă puternică pentru probleme de query fără update-uri (modificarea valorii unui element din structura de date). Găsiți o explicație bună aici: [Sparse Table & RMQ (Range Minimum Query) - YouTube](https://www.youtube.com/watch?v=0jWeUdxrGm4).

Complexitatea-timp va fi , iar cea spațiu .

void VariantaRMQ()  
{  
 vector<vector<int>> dp(N + 1, vector<int> (int(log2(N) + 10), 0));  
 for(int i = 0; i < N; ++i)  
 dp[i][0] = A[i];  
  
 for(int j = 1; (1 << j) <= N; ++j)  
 for(int i = 0; i + (1 << j) <= N; ++i)  
 dp[i][j] = min(dp[i][j - 1], dp[i + (1 << (j - 1))][j - 1]);  
  
 while(M--)  
 {  
 cin >> leftQ >> rightQ;  
 leftQ--, rightQ--;  
 int len = log2(r - l + 1);  
 cout << min(dp[l][len], dp[r - (1 << len) + 1][len]) << '\n';  
 }  
}