# Range Minimum Query

Unguru Dragoş-Gabriel – 325CD

Universitatea Politehnică Bucureşti
Facultatea de Automatică şi Calculatoare
unguru.dragos15@gmail.com

**Rezumat** Lucrarea are ca scop compararea mai multor metode de rezolvare ale problemei Range Minimum Query. Mai departe vom analiza eficienţa mai multor abordări atât din punct de vedere al performanţelor de rulare cât şi al eficacităţii gestiunii memoriei.

**Keywords:** Algoritmul lui Mo · Sparse table · Arbori de intervale

## 1 Descrierea problemei rezolvate

### 1.1 Obiectivul problemei

Range Minimum Query (RMQ) este soluţia problemei de a găsi valoarea minimă (sau indexul acestuia) dintr-un subvector de elemente comparabile, între doi indici specifici (queries).

Ca exemplu, având un vector A = | 5 | 2 | 7 | 5 | 1 | 3 | 16 | 0 | 12 | şi query-ul: [2 5], vom căuta valoarea minimă a subvectorului: | 7 | 5 | 1 | 3 |, deci, putem spune că $RMQ_A(2, 5) = 1$

### 1.2 Aplicaţie practică a problemei

RMQ joacă un rol important în problema celui mai lung prefix comun (Longest Common Prefix - LCP) cât şi în procesarea vectorilor de sufixe. Vectorii de sufixe reprezintă o structură de date care înlocuieşte structura de arbori de sufixe. Cu ajutorul acestei structuri de date se realizează căutări de subşiruri aproape la fel de rapid ca şi arborii de sufixe.

Mai mult de atât, RMQ poate fi privit şi ca o versiune restricţionată a problemei celui mai mic strămoş comun (Lowest Common Ancestor - LCA). Efectuând o parcurgere eulerianã a arborelui şi notând într-un vector nivelul fiecărui nod din parcurgere aferent acestuia, tot ce mai rămâne de făcut este aplicarea problemei RMQ având ca şi query indexul primei apariţii nodurilor primite ca input pentru problema LCA. Minimul corespunzător nivelului găsit de RMQ va fi cel mai mic strămos comun.

## 2 Specificarea soluţiilor alese

Rezolvarea "naivă" a acestei probleme este parcurgerea vectorului de-a lungul fiecărui query pentru găsirea minimului, abordare care prezintă o complexitate liniară. Deseori vrem să răspundem la un număr mare de query-uri, caz în care folosirea acestei metode este prea lentă.

Problema RMQ este una dintre cele mai riguros studiate probleme de informatică, deci sunt cunoscute numeroase abordări. Dintre acestea, vom selecta următoarele:

- **Algoritmul lui Mo** sau Query Square Root Decomposition care presupune sortarea query-urilor într-un mod mai avantajos.
- **Sparse table** unde vom folosi o matrice pentru stocarea răspunsurilor preprocesate ale query-urilor.
- **Arbori de intervale** presupune folosirea unei structuri de date mai avantajoase: arbori binari.

## 3 Criterii de evaluare pentru problema propusă

Cum mărimea elementelor din vectorul de input nu influenţează eficienţa problemelor propuse, ne vom axa pe date de intrare cât mai numeroase.

Pentru a putea evidenţia diferenţele de performanţă ale celor trei metode, vom rula seturi de teste cu proprietăţi diferite. Astfel vom putea afla situaţiile în care fiecare algoritm se comportă cel mai eficient. Atât dimensiunea vectorului de intrare cât şi elementele acestuia vor fi generate aleator. Aşadar, vom lua în calcul următoarele format-uri de teste:

- Small: Numărul elementelor ale vectorului va fi maxim $10^3$. Acest set de teste este reprezentat de date aleatorii şi variate. Dependenţa numărului de query-uri faţă de mărimea vectorului este slabă.
- Large: Teste cu un număr mic de query-uri formate din intervale lungi. Pentru aceste teste, vom lua un număr de $\frac{n}{11}$ cele mai lungi query-uri. Un număr considerabil mai mic decât numărul maxim de query-uri. (Mai exact, de $\frac{11(n-1)}{2}$ ori mai mic). Astfel, vom obţine şi o dependinţă liniară între numărul de query-uri şi numărul de elemente. De asemenea, numărul de elemente din vector va fi de cel puţin $10^3$.
- Stress test - O să includem numărul maxim de query-uri posibile. Pentru un vector de $n$ elemente, vom putea forma un maxim de $C_n^2 = \frac{n(n-1)}{2}$ query-uri.
- Modifying array tests - Teste unde vectorul de input se modifică pe parcurs. Am luat date iniţiale (array-ul de început si query-urile) datele testelor de mai sus, având astfel, câte un test cu modificare pentru fiecare tip de test.

Pentru fiecare set de teste menţionate mai sus, vom aplica metoda "Fisher-Yates shuffle" pentru amestecarea query-urilor generate. Folosind "algoritmul lui Knuth", ne asigurăm o amestecare eficientă a datelor.

# 4 Prezentarea soluţiilor alese

## 4.1 Descrierea funcţionalităţii algoritmilor

**Algoritmul lui Mo** (sau Query Square Root Decomposition) se bazează pe preprocesarea răspunsurilor la query-uri pe un interval de lungime $\sqrt{n}$, unde $n$ reprezintă lungimea vectorului de intrare. Acest algoritm prezintă două metode de preprocesare posibile:

- Metoda offline - Metoda pe care, personal, am ales-o. Această abordare nu necesită cunoaşterea query-urilor dinainte. Preprocesarea constă în precomputarea minimului query-urilor $[0, \sqrt(n) - 1]$, $[\sqrt{n}, 2 \cdot \sqrt{n} - 1]$... ş.a.m.d. .
- Metoda online - Această abordare prezintă o metodă avantajoasă de sortare a query-urilor. Complexitatea aşteptată de la algoritmul lui Mo este asigurată prin aranjarea query-urilor ce urmează a fi răspunse după valoarea $a$ (presupunând că query-urile sunt de forma $[a, b]$) în blocuri de lungime $\sqrt{n}$. Query-urile cu valoarea $a$ în intervalul $[0, \sqrt{n} - 1]$, iar acestea sortate la rândul lor crescător după valoarea lui $b$. Procesul se repetă pentru toate blocurile de lungime $\sqrt{n}$.

Pentru ambele metode, ne asigurăm o complexitate avantajoasă de răspuns al unui query prin interogarea valorilor preprocesate plus o parcurgere de maxim $\sqrt{n}$ elemente din vector.

**Sparse Table** vine cu o preprocesare bazată pe o structură de date mai costisitoare - matrici. Cu toate că acest compromis ne oferă un avantaj major când vine vorba de complexitatea răspunsului unui query.
Această abordare presupune preprocesarea răspunsurilor la query-uri de lungime puteri ale lui 2 ($2^i, i \in [0, \log_2 n]$). Aceste valori preprocesate sunt stocate într-o matrice ($matrix \in M_{nxlog(n)}$) cu proprietatea că valoare de pe poziţia $(i, j)$ reprezintă minimul din intervalul de indecşi $(i, 2^j - 1)$.
Complexitatea construirii acestei structuri de date este atenuată de folosirea valorilor computate la paşii anteriori în formarea valorii de la pasul curent. Structura de date este construită pornind doar de la iniţializarea primei coloane cu valorile efective ale vectorului şi continuând într-o manieră ascendentă.
Marele avantaj al acestei preprocesări este faptul că ne permite să răspundem la orice query prin efectuarea a doar unei singure comparaţii de valori preprocesate în matrice. Ca exemplu, pentru a răspunde la query-ul $[0, 7]$, interogăm şi comparăm valorile aflate pe poziţiile $[0, 3]$ şi $[4, 7]$ pentru a obţine minimul.

**Arbori de intervale** prezintă o abordare mai versatilă folosind o structura de date bazată pe arbori binari care au proprietatea că:

- Frunzele reprezintă elementele vectorului;
- Fiecare nod reprezintă minimul tuturor frunzelor descendente lui.

Spunem că această abordare este mai versatilă pentru că ne permite modificarea structurii de date în cazul în care se efectuează schimbări în vectorul iniţial pe parcursul programului, fără a relua tot procesul de la capăt.
Începând construirea arborelui prin iniţializarea nodurilor frunză (ultimele elemente din vector) cu elementele vectorului şi apoi construirea părinţilor "de jos în sus", ne va rezulta mereu un arbore complet, injumătaţind mereu intervalul reprezentativ minimului.
În final, o să avem o structură de date unde rădăcina reprezintă minimul întregului vector, iar, coborând pe descendenţi în jos, fiecare nod reprezentând minimul intervalului anterior înjumătaţit.
Arborele fiind mereu unul complet, această abordare ne permite o implementare uşoară folosind un simplu vector pentru a-l reprezenta, existând relaţii de a obţine părintele cât şi fiecare descendent al fiecărui nod.

### 4.2 Analiza complexităţii algoritmilor

Cum nu există secvenţe recursive, pentru a studia complexităţile algoritmilor, ne vom concentra pe structurile repetitive de-a lungul programelor.

**Algoritmul lui Mo - preprocesare:** Complexitatea părţii de preprocesare a acestui algoritm este dată de secvenţa:

```
23      // Preprocessing:
24      for (i = 0; i < n; i += len) {
25          decomposedMins[i / len] = *std::min_element(arr + i, arr + i + len);
26          processedLength++;
27      }
```

Preprocesare algoritmul lui Mo

La prima vedere pare că se realizează în $\mathcal{O}(\sqrt{n})$ ($len = \sqrt{n}$) dar trebuie luat în calcul şi ce se întâmplă în metoda *min_element*. Apelând-o pentru fiecare bucată de lungime $\sqrt{n}$, această metodă parcurge mereu vectorul "arr" între indecşii $[i, i + \sqrt{n}]$. Apelând în această manieră într-un for ce sare din $\sqrt{n}$ în $\sqrt{n}$, ajungem, de fapt, la parcurgerea întregului vector, rezultându-ne, astfel, o complexitate de $\mathcal{O}(n)$.

**Algoritmul lui Mo - răspuns la query:** Singura parcurgere pe care trebuie s-o efectuăm pentru a răspunde la un query este:

```
47          while (L <= R) { // If our query covers lonely indices (parts of preprocessed intervals):
48              if ((L % len != 0) || (R - L < len)) {
49                  lastStandingMin = MIN(lastStandingMin, arr[L]);
50                  L++;
51              } else { // If our query covers a whole interval of length sqrt(n):
52                  lastStandingMin = MIN(lastStandingMin, processedMins[L / len]);
53                  L += len;
54              }
55          }
```

Aflarea minimului - algoritmul lui Mo

Având preprocesate toate răspunsurile la query-uri de lungime $\sqrt{n}$, ne asigură pentru orice alt viitor query o parcurgere de maxim $\sqrt{n}-1$ înainte de a interoga un query deja preprocesat. Aflarea minimului unui bloc preprocesat peste care se suprapune query-ul nostru se realizează în $\mathcal{O}(1)$ (având datele preprocesate), tot ce mai rămâne de făcut este să comparăm aceste valori cu elementele din vector care nu constituie un bloc de sine stătător de lungime $\sqrt{n}$. Efectuând aceşti paşi pentru un query obţinem complexitatea pe cel mai rău caz $\mathcal{O}(\sqrt{n})$.
Spaţiul necesar pentru a reţine datele preprocesării este $\mathcal{O}(\frac{n}{\sqrt{n}}) = \mathcal{O}(\sqrt{n})$.

**Sparse Table - preprocesare:** Partea de preprocesare a Sparse Table este descrisă de secvenţa de cod:

```
15      // Initialize first column. Minimum of [i, i] = arr[i];
16      for (int i = 0; i < arrSize; i++)
17          table[i][0] = arr[i];
18
19      // Using bitwise operator " << "
20      // knowing that: (1 << a) = 2^a
21
22      for (int i = 1; (1 << i) - 1 < arrSize; i++) {
23          for (int j = 0; j + (1 << i) - 1 < arrSize; j++) {
24              table[j][i] = MIN(table[j][i - 1], table[j + (1 << (i - 1))][i - 1]);
25          }
26      }
```

Preprocesare - Sparse Table

Urmărind codul de mai sus, observăm o parcurgere a vectorului de input şi două structuri repetitive îmbricate. Deşi ambii indecşi merg incremental, observăm că avem ca şi condiţie de oprire $2^i < n$ respectiv $j + 2^i < n$.
Pentru a calcula complexitatea celor doua structuri repetitive, observăm că primul for poate efectua un maxim de $\log_2 n$ iteraţii (pentru $i = \log_2 n \Rightarrow 2^{\log_2 n} = n \Rightarrow$ for-ul încetează) iar condiţia de oprire a celui de-al doilea poate fi exprimată ca $j < n + 1 - 2^i$. Aşadar, obţinem:

Pentru $i = 1 \Rightarrow j = n + 1 - 2$ (j = nr. maxim de iteraţii)
Pentru $i = 2 \Rightarrow j = n + 1 - 2^2$
.
.
.
Pentru $i = \log_2 n \Rightarrow j = n + 1 - 2^{\log_2 n}$
$\_\_\_\_\_(+)$
$T(n) = \sum_{i=1}^{\log_2 n} n + \sum_{i=1}^{\log_2 n} 1 - (2 + 2^2 + 2^3 + ... + 2^{\log_2 n}) \Rightarrow$
$T(n) = n \log_2 n + \log_2 n - \frac{2^{\log_2 n + 1} - 1}{2 - 1} \Rightarrow$
$T(n) = n \log_2 n + \log_2 n - 2n$

Dând un factor comun forţat / considerând doar termenul dominant, ne rezultă complexitatea celor două for-uri ca fiind $\mathcal{O}(n \log_2 n)$.
Luând în calcul şi parcurgerea vectorului de intrare (liniile 16 - 17), ne rezultă o complexitate temporală de preprocesare de $\mathcal{O}(n + n \log_2 n) = \mathcal{O}(n \log_2 n)$.

**Sparse Table - răspuns la query:** Răspunsul la query pentru această abordare este reprezentat de o simplă interogare a valorilor preprocesate anterior:

```
35      for (int i = 0; i < noOfQueries; i++) {
36          int value = myLog(queries[i].b - queries[i].a + 1);
37          int minimum = MIN((table[queries[i].a][value]), (table[queries[i].b - (1 << value) + 1][value]));
38
39          fout << "Minimum of [" << queries[i].a << ", " << queries[i].b << "]: " << minimum << "\n";
40      }
```

Răspuns la query - Sparse Table

Deci avem o complexitate de răspundere la query de $\mathcal{O}(1)$.
Spaţiul necesar pentru Sparse Table (de a reţine datele preprocesării) este de $\mathcal{O}(n \log_2 n)$

**Arbori de intervale - preprocesare:** Pentru preprocesare, avem următoarea secvenţă:

```
13      // Initialize leaves with array values
14      for (int i = arrLength; i < 2 * arrLength; i++) {
15          tree[i] = arr[i - arrLength];
16          tree[i - arrLength] = __INT_MAX__;
17      }
18
19      // Strating from the previously initialised leaves, create parents
20      for (int i = arrLength - 1; i > 0; i--) {
21          tree[i] = MIN(tree[GET_LEFT(i)], tree[GET_RIGHT(i)]);
22      }
```

Preprocesare - Segment Trees

Cum prima structură repetitivă iterează de la $n$ la $2n$, iar a doua de la $n$ la 1, putem spune cu uşurinţă că complexitatea este $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

**Arbori de intervale - modificare vector:**

```
33      while(index > 1) {
34
35          index = GET_PARENT(index);
36
37          tree[index] = MIN(tree[GET_LEFT(index)], tree[GET_RIGHT(index)]);
38      }
```

Update array - Segment Trees

Cum metoda $i = GET\_PARENT(i)$ este echivalentă cu $i = i/2$, structura repetitivă prezentă în imaginea de mai sus poate fi scrisă şi sub forma:

$$for(i = n; i > 1; i/ = 2)$$

care are o complexitate de $\mathcal{O}(\log_2 n)$. Aşadar, complexitatea gestionării unui nou element în structura noastră de date este de $\mathcal{O}(\log_2 n)$

**Arbori de intervale - răspuns la query:** Pentru a răspunde la un query, executăm secvenţa de cod:

```
54          while(treeIdx.a < treeIdx.b) {
55
56              if (treeIdx.a % 2 != 0) {
57                  minimum = MIN(minimum, tree[treeIdx.a]);
58                  treeIdx.a++;
59              }
60              if (treeIdx.b % 2 != 0) {
61                  treeIdx.b--;
62                  minimum = MIN(minimum, tree[treeIdx.b]);
63              }
64
65              treeIdx.a = GET_PARENT(treeIdx.a);
66              treeIdx.b = GET_PARENT(treeIdx.b);
67          }
```

Răspuns la un query - Segment Trees

Considerând cel mai nefavorabil caz (când cel mai apropiat strămoş comun ai nodurilor corespunzătoare celor doi indecşi este chiar rădăcina arborelui), complexitatea acestei secvenţe va fi de $\mathcal{O}(\log_2 n)$ cât timp cei 2 indecşi (treeIdx.a şi treeIdx.b) se împart la 2 pentru fiecare iteraţie.
Din punct de vedere al memoriei, această metodă necesită spaţiu suplimentar de $\mathcal{O}(2n) = \mathcal{O}(n)$.

### 4.3 Avantaje şi dezavantaje

Mai departe, vom analiza performanţele fiecărui algoritm în comparaţie cu ceilalţi doi.

**Algoritmul lui Mo:**

- Avantaje
  - Din punct de vedere al memoriei, comparativ cu celelalte două metode, algoritmul lui Mo căştigă cu o margine fină. Preprocesarea acestui algoritm necesită un spaţiu de $\mathcal{O}(n)$ faţă de $\mathcal{O}(n \log_2 n)$ pentru Sparse Table si $\mathcal{O}(2n)$ (teoretic tot $\mathcal{O}(n)$ dar în practică există o diferenţă) pentru arbori de intervale.
- Dezavantaje
  - Când vine vorba de complexitatea de rulare, acest algoritm răspunde la query-uri în cel mai nefavorabil timp ($\mathcal{O}(\sqrt{n})$).
  - Pentru teste când vectorul de input se modifică pe parcursul rulării, algoritmul lui Mo necesită reluarea executării preprocesării de la bun început.

**Sparse Table:**

- Avantaje
  - Cel mai mare avantaj este reprezentat de complexitatea pe query. Această abordare răspunde la orice query în $\mathcal{O}(1)$. Mai departe vom vedea impactul acestui lucru.
- Dezavantaje
  - Preprocesarea necesită cea mai mare complexitate de memorie, anume $\mathcal{O}(n \log_2 n)$
  - Dificultate de implementare mai mare.
  - Asemănător algoritmului lui Mo, pentru teste când vectorul de input se modifică pe parcursul rulării, acesta necesită reluarea executării preprocesării de la bun început.

**Arbori de intervale:**

- Avantaje
  - Structura de date ne permite să efectuăm modificări în cazul schimbării valorilor vectorului de intrare, fără a relua întreaga preprocesare.
  - Prezintă cea mai bună combinaţie de complexităţi între preprocesare şi răspundere la query: $\mathcal{O}(n)$ respectiv $\mathcal{O}(\log_2 n)$.
  - Dificultate mică de implementare.
- Dezavantaje
  - Comparativ cu ceilalţi doi algoritmi, aparenţele fac să pară că nu există vreun dezavantaj faţă de aceştia. Însă vom vedea că din punct de vedere al timpului de execuţie, această abordare este net mai slabă faţă de Sparse Table.

## 5 Evaluare

### 5.1 Construirea setului de teste

Setul de teste este construit folosind generarea de numere aleatoare atât pentru lungimea vectorului cât şi a elementelor sale.

- Testele 0 - 8: Teste aleatoare. Aceste teste reprezintă cea mai comună formă de date de intrare. Dintre acestea, testul "test8.in" reprezintă un test format din mai multe date de intrare ($inputArr.length() = 45720$). Ne vom folosi de acest test pentru a evidenţia cu usurinta diferenţele de performanţă pentru acest tip de teste.
- Testele 9 - 17: Teste cu dependinţă polinomială între numărul query-urilor si lungimea vectorului. Aceste teste sunt variate, cu n aparţinând intervalului $(200, 8900)$. De asemenea, query-urile acestor teste sunt de lungime foarte mare.
- Testele 18 - 26: Stress tests. Aceste teste sunt formate dintr-un $n$ de maxim $10^3$ (pentru testele 18 - 24) şi $10^4$ (25  26). Aceste teste sunt formate din numărul maxim de query-uri posibile.

– Testele 27 - 29: Teste unde vectorul de input se modifică pe parcurs. Am luat ca date iniţiale (array-ul de început si query-urile) datele testelor 7, 13 si 21. Avand astfel, cate un test cu modificare pentru fiecare tip de test enunţat mai sus. Pentru fiecare, vom avea 7 "seturi" de modificări. Acest proces va fi repetat de 7 ori pentru fiecare din cele 3 teste.

Întrucât perechile de numere ce constituie interogările sunt generate iterativ, după generarea acestora, aplicăm un algoritm de amestecare aleator a acestora. Algoritmul folosit este algoritmul lui Knuth, algoritm implementează metoda Fisher-Yates de generare de permutări a unei secvenţe finite. Lăsându-le neamestecate, testele ar fi reprezentat "best case scenario" pentru sortarea query-urilor algoritmul lui Mo.

### 5.2 Evaluarea timpului de execuţie

Pentru această etapă, testele descrise mai sus au fost rulate pe un sistem compus dintr-un procesor Intel Core I7 7700HQ cu 8GB de memorie RAM. Acest lucru aduce dificultăţi în calcularea efectivă a timpului de rulare, procesorul fiind dotat cu "Smart Caching", frecvenţă variabilă ş.a.m.d. .
Pentru a evita pe cât posibil implicarea acestor factori în datele noastre dar încercând totodată să obţinem date valide, am folosit un script bash care rulează fiecare metodă pe fiecare test de 7 ori şi realizează o medie a timpului de rulare pentru fiecare.
Timpul de execuţie este măsurat folosind "clock-urile" procesorului; mai specific funcţia *clock*() a librăriei $< time.h >$.
Cum am obţinut o valoare de $10^6$ pentru $CLOCKS\_PER\_SECOND$, avem certitudinea că valorile generate cu ajutorul metodei *clock*() sunt exprimate în microsecunde.
Aşadar, în urma rulării unui script bash, obţinem datele necesare.

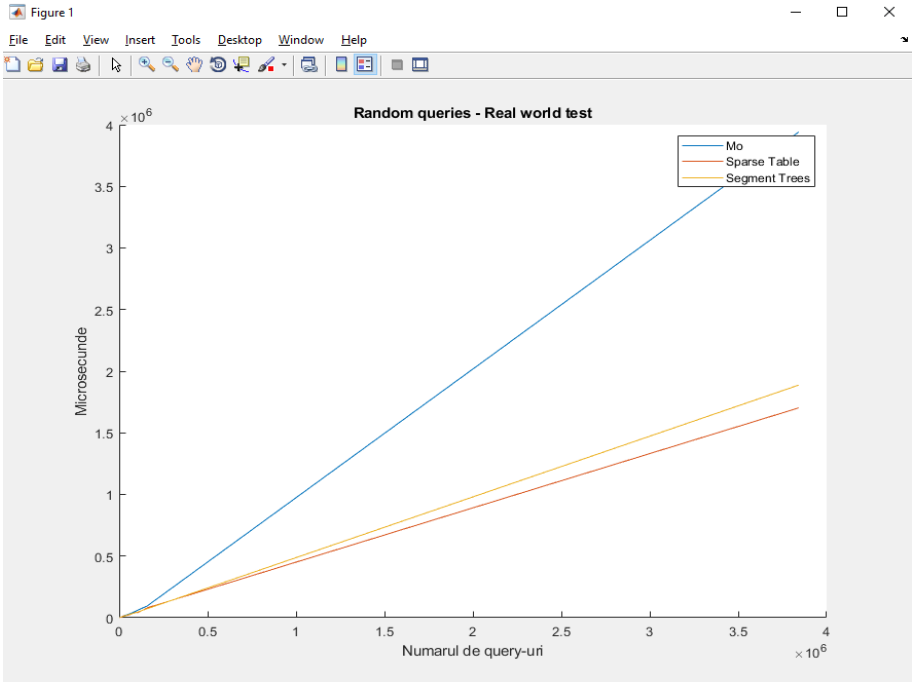
```
#!/bin/bash

NO_OF_TESTS=29
MEAN=0
NR_DE_RULARI=7

#       RUNNING FOR MO

for i in $( seq 0 $NO_OF_TESTS )
do
        for j in $( seq 1 $NR_DE_RULARI )
        do
                OUTPUT=$(./tema mo ./in/test$i.in)
                wait            # Wait for process to finish
                MEAN=$((OUTPUT + MEAN))
        done
        echo $((MEAN / NR_DE_RULARI)) >> runtime_data_MO.txt
        MEAN=0
done
```

Script generare timpi de execuţie

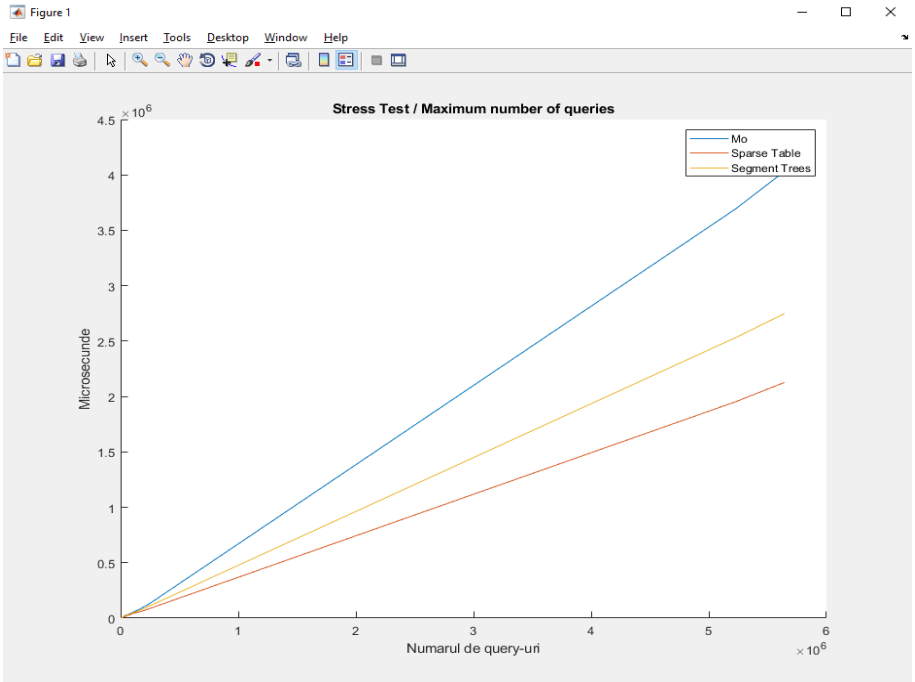
### 5.3 Interpretarea datelor

Folosindu-ne de un mic program MATLAB, putem reprezenta aceste valori grafic, pentru o întelegere mai uşoară.

**Teste aleatoare şi "stress tests" -** În reprezentarea de mai jos se poate observa un rezultat evident. Testele generate complet aleator reprezintă testele cu cea mai ridicată probabilitate de a fi întalnite în practică.
În acest exemplu este ilustrat pur diferenţa între complexităţi pe răspuns la query compusă cu diferenţa complexităţii între preprocesări; Sparse Table şi Arbori de Intervale fiind clar superiori faţă de algoritmul lui Mo. Acest lucru se poate observa atât în primul grafic cât şi în cel de-al doilea.
Deşi în primul grafic putem observa că există o diferenţă infimă între Arbori de Intervale si Sparse table, acest lucru se datorează faptului că aceste teste sunt formate (la început) dintr-un număr mai mic de query-uri. Odată cu creşterea numărului de query-uri creşte şi discrepanţa dintre cei doi algoritmi, simţindu-se impactul răspunsului în $\mathcal{O}(1)$ a algoritmului Sparse Table din ce în ce mai mult.
Întrucât interogările prezente în aceste teste nu au niciun fel de proprietate anume, obţinem un grafic curat: o dependinţă liniară între timpul de execuţie şi numărul de query-uri.
Timpii de execuţie pentru aceste teste includ şi partea de preprocesare a fiecărei metode.

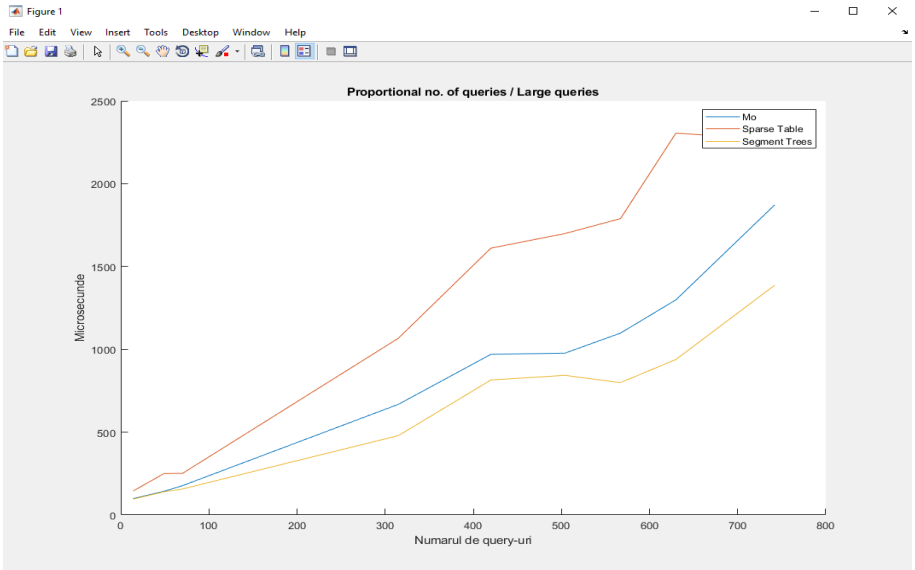



Teste aleatoare (0 - 8)




Teste cu număr maxim de query-uri (18 - 26)

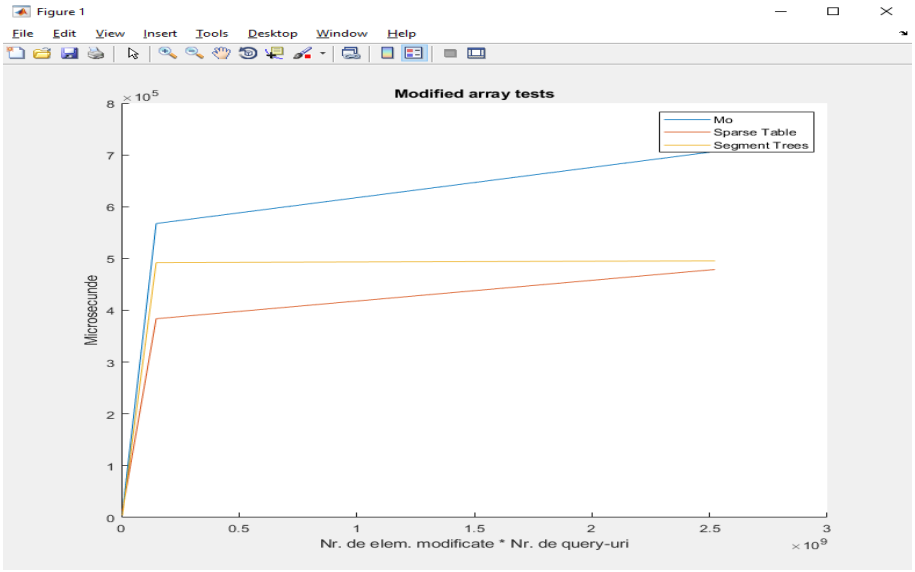
**Sparse tests -** De-a lungul acestor teste observăm încă din procesarea datelor un comportament mai ciudat. Timpii de execuţie nu mai sunt dependenţi liniar faţă de numărul de query-uri:




Teste cu număr mic de query-uri foarte largi

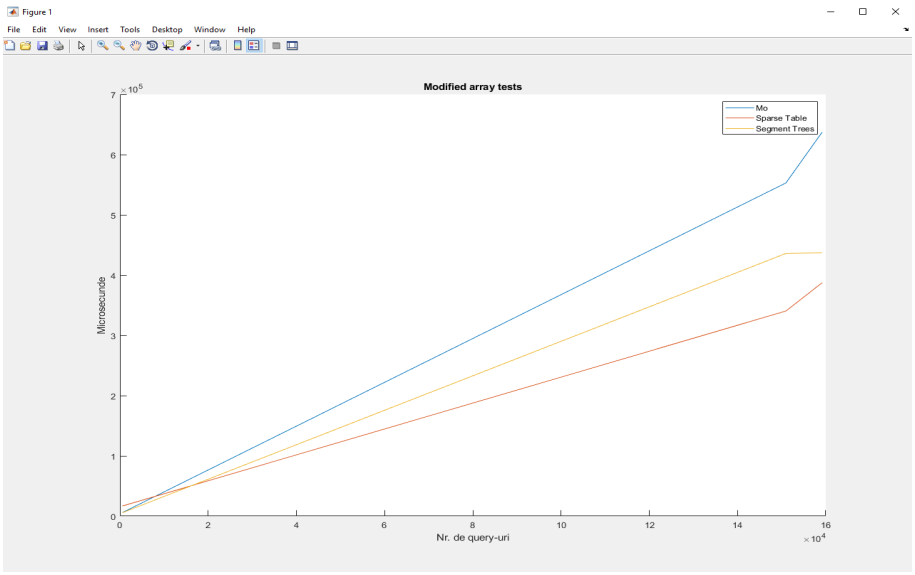
În aceste teste, complexităţile preprocesărilor algoritmilor sunt critici; aceştia făcând diferenţa. Fiind un număr mic de query-uri, puterea abordării Sparse Table de a răspunde la un query în $\mathcal{O}(1)$ este insignifiantă cât timp are complexitatea preprocesării mult mai mare faţă de rivali.($\mathcal{O}(n \log_2 n) > \mathcal{O}(n)$
Cu cât numărul de query-uri creşte, însă, Sparse Table reuşeşte, eventual, să recupereze timpul pierdut pe preprocesare răspunzând la query-uri într-un timp de $\mathcal{O}(1)$ faţă de $\mathcal{O}(\sqrt{n})$ şi $\mathcal{O}(\log_2 n)$.

**Teste cu modificarea vectorului pe parcurs -** Pentru aceste teste, am luat în considerare timpii de rulare ai algoritmilor cu şi fără partea de preprocesare Am făcut acest lucru pentru a studia mai îndeaproape avantajul algoritmului "Segment Trees" de a regestiona structura de date fără a relua tot procesul de la capăt.




Modificarea vectorului pe parcurs - Considerând nr. de modificări

Pentru a obţine un grafic cu o reprezentare mai detaliată, am modificat ca pe axa reală să avem (numărul de query-uri · numarul de elemente modificate).
Conflictul interesant este între Sparse Table şi Segment Trees. Observăm cum Sparse Table pare a fi încă superior, datorită răspunsului rapid la query-uri. Dar pe măsură ce numărul de modificări creşte, Segment Trees îi preia locul.
Superioritatea Sparse Table-ului la început se justifică şi datorită faptului că testele care au fost rulate conţin şi un număr destul de mare de modificări ale vectorului. Acest lucru îngustează diferenţa dintre cei doi algoritmi, când vine vorba de preprocesare. Dacă Sparse Table reia tot procedeul de preprocesare şi Segment Trees are de updatat un număr apropiat de $n$, Sparse Table încă rămâne superior, având un timp rapid de raspuns la interogări cu care să compenseze.




Modificarea vectorului pe parcurs 2

Dacă realizăm o reprezentare în care luăm în calcul doar timpul de execuţie în funcţie de numărul de query-uri (graficul de mai sus), obţinem un rezultat destul de evident: Segment Trees este net superior. Acest lucru se datorează avantajului acestuia de a putea modifica structura de date.
Chit că pentru această reprezentare nu s-a luat în calcul preprocesarea de început a niciunui algoritm, Segment Trees are, indiferent, cel mai eficient timp de rulare.

## 6 Concluzii

În urma analizei realizate mai sus, putem spune cu încredere că metoda Sparse Table este superioară din punct de vedere al complexităţii de timp. Acesta are un avantaj enorm când vine vorba de timpul de răspuns la un query. Acest lucru care, în teorie, părea evident s-a dovedit a fi valid chiar şi în practică.
Însă, de cele mai multe ori, în practică "baza de date" (vectorul nostru de intrare) va fi modificat. Situaţie în care Segment Trees rămâne opţiunea preferată. Algoritmul lui Mo iese în evidenţă şi este de preferat să fie folosit atunci când interogările sunt (aproape de a fi) sortate (pentru atunci când se cunosc query-urile). Această situaţie reprezintă "best case" pentru acest algoritm. Iar când memoria joacă un rol important în dezvoltarea programului, acesta rămâne decizia raţională.

## Bibliografie

1. Niklas Baumstark1, Simon Gog, Tobias Heuer and Julian Labeit - Practical Range Minimum Queries Revisited. Accesat la data de 17 noi. 2018.

2. Cursuri Universitatea Stanford Accesat la data de 17 noi. 2018.
3. Johannes Fischer şi Volker Heun - Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. Accesat la data de 17 noi. 2018.