Range Minimum Query

Elementul Minim dintr-un interval

Dumitrescu Alexandra
323CA

[1] Universitatea Politehnica București, Facultatea de Automatică și Calculatoare, RO
[2] adumitrescu2708@stud.acs.upb.ro

Rezumat Se propune un studiu comparativ între 3 algoritmi ce rezolvă problema identificării elementului minim dintr-un interval, analizați ulterior din punct de vedere al complexității și al aplicabilității.

Keywords: Range Minimum Query · Square Root Decomposition · Sparse Table · Segment Trees · Dynamic Programming

1 Introducere

1.1 Descrierea Problemei

Definiția problemei Se consideră un vector A de N elemente numere întregi și M interogări (x_i, y_i). Ne propunem să răspundem la întrebări de forma: *Care este cel mai mic element din intervalul delimitat de pozițiile x_i și y_i în vectorul A?*.

$$RMQ(x_i, y_i) = min(A[x_i], A[x_i + 1], A[x_i + 2]...A[y_i]) \tag{1}$$

1.2 Exemplificare Practică a Problemei

Auto Completion [2] Auto-Completarea reprezintă o sarcină indispensabilă în proiectarea unui număr larg de aplicații precum: siteuri de social networking, browsere web, baze de date, IDE-uri, interpretatoare command-line, aplicații de mobile text și multe altele.
Problema se poate reduce la alegerea primelor K completări dintr-un dicționar de stringuri S care au același prefix cu string-ul introdus de utilizator și care au cel mai mare scor raportat la un clasament statistic.
În majoritatea cazurilor practice, este nevoie de o structură de date care să se comporte exemplar din punct de vedere al eficienței de timp deoarece o data ce utilizatorul a introdus un caracter se dorește un update instant al celor K completări din setul S. Printre numeroasele soluții descoperite se numără și o structura de trie, referită RMQ TRIE.
Pe baza dicționarului de stringuri se construiește trie-ul și se face o asociere bijectivă între fiecare string și un array de scoruri, pe care se aplică algoritm de RMQ inversat (cel mai mare scor va fi gândit ca minimul din intervalul de căutare) pentru a găsi cele mai mari K scoruri.

Problema celui mai mic strămoș comun [6] Dându-se un arbore T având N noduri și M interogări de forma (x_i, y_i), se cere strămoșul nodurilor x_i și y_i cu cea mai mare adâncime în arborele T.
În teoria grafurilor, problema *Lowest Common Ancestor - LCA* este folosită preponderent în string processing și în ierarhizări precum cele utilizate în sistemele dedicate programării orientate pe obiect.
Problema LCA se poate rezolva folosind RMQ drept subproblemă. Ideea ce stă la baza acestei reduceri constă în parcurgerea într-un tur Eulerian a arborelui și reținerea informațiilor în 3 vectori auxiliari, unde E[i] reprezintă nodul vizitat la pasul i, L[i] nivelul pe care se află nodul E[i] și H[i] prima apariție a nodului i în E[i]. Se aplică RMQ pe vectorul de nivele L[i], întrucât se caută nivelul minim, între pozițiile de descoperire ale nodurilor interogate, H[x_i] și H[y_i].

1.3 Soluțiile supuse comparației

Precum s-a observat în exemplele practice, setul de date inițial poate avea dimensiuni mari și nu este modificat deloc în timpul interogărilor. Astfel, majoritatea soluțiilor RMQ preprocesează datele de intrare pentru a asigura ulterior răspunsul eficient la interogări.

Square Root Decomposition Primul algoritm are la bază ideea de a sparge vectorul A în blocuri de dimensiune $sqrt(N)$ și de a reține într-un vector auxiliar preprocesat minimul pentru fiecare bloc. Astfel, în loc să căutăm minimul liniar în interval la fiecare interogare, ne vom folosi de minimul blocurilor de dimensiune $sqrt(N)$ incluse în interval.

$$PREP[i] = min(A[k]), k = (i \times \sqrt{N}) : (i \times \sqrt{N} + \sqrt{N} - 1) \tag{2}$$

Sparse Table Algoritmul al doilea își propune să preproceseze datele într-o structură auxiliară de date, o matrice rară, construită progresiv printr-o metodă de programare dinamică. Reținem în matricea auxiliară minimul pentru toate subintervalele ce încep în vectorul A de la poziția i și au dimensiunea de 2^j. Fiecare interogare se va soluționa prin împărțirea intervalului în 2 subintervale, iar răspunsul va fi minimul dintre cele 2 minime precalculate în matrice.

$$PREP[i][j] = min(A[k]), k = i : (i + 2^j - 1) \tag{3}$$

Arbori de Intervale Se preprocesează datele într-un arbore binar ce are frunzele elementele vectorului A, iar nodurile interne reprezintă minimul tuturor frunzelor descendente. Un nod codifică minimul pentru un interval $[i, j]$, iar fii săi vor reține minimul pentru $[i, (i + j)/2]$ și $[(i + j)/2 + 1, j]$. Arborele va fi reținut în memorie printr-un vector de $2 \times 2^{ceil(log(N))} - 1$ elemente. Interogările se rezolvă printr-o căutare în arbore.

1.4 Criterii de Evaluare

Pentru evaluare vom genera teste atât manual, cât și random folosind funcțiile srand() și rand() din biblioteca time.h [4]. Pentru testarea corectitudinii vom testa gradual fiecare algoritm cu un checker ce rezolvă problema într-un mod banal și compară outputurile, iar în final vom verifica outputurile celor 3 soluții care trebuie să coincidă, urmând să le comparăm eficiența.

Raport date de intrare - interogări Vom compara cum se comportă programele raportat la numărul de interogări și de date de intrare pentru $M > N$, $N > M$ și M proporțional cu N, urmând să luăm o decizie în care situație este mai eficient să folosim un anume program, raportat la memoria auxiliară utilizată pentru preprocesare și numărul de interogări M. Numerele N și M o sa varieze de la ordin mic pană la ordin mare, pentru a nota cum se comportă algoritmii pe vectori de dimensiuni mari, respectiv mici.

Organizare date de intrare Propunem seturi de date în care elementele sunt ordonate diversificat. Vom avea teste în care datele sunt sortate crescător, astfel încat elementul minim dintr-un interval dat să se afle la începutul acestuia, sortate descrescător, unde minimul se va afla la sfârșitul intervalului și o organizare alternantă, în care minimul se va putea găsi și central. De altfel, vor fi teste în care elementele vor fi generate aleator precum s-a descris anterior.

Tipuri de interogări Vom testa corectitudinea prin intervale (x_i, y_i) cât mai variate. Spre exemplu, vrem să aflăm cum se comportă algoritmii din punct de vedere al timpului de execuție pentru intervale cât mai răstrânse sau cât mai permisive, ajungând de la teste de forma (x_i, x_i) până la teste $(1, N)$.

2 Prezentarea soluțiilor

2.1 Descrierea Algoritmilor

Square Root Decomposition *Preprocesare* Algoritmul sparge vectorul inițial în $ceil(\sqrt{N})$ blocuri a câte $floor(\sqrt{N})$ elemente și reține într-un vector auxiliar minimul pentru fiecare bloc. Astfel, în loc să fim nevoiți să parcurgem tot vectorul inițial pentru a găsi minimul pentru un interval interogat, ne vom folosi de un vector preprocesat codificat astfel:

$$PREP[i] = min(A[k]), k = (i \times \sqrt{N}) : (i \times \sqrt{N} + \sqrt{N} - 1)$$

Interogări Fiecare interogare se rezolvă prin găsirea minimului dintre toate blocurile întregi conținute în intervalul interogat (x_i, y_i) și eventual dintre blocurile de început și de final dacă acestea sunt trunchiate. Blocurile vor corespunde pozițiilor din intervalul $(\frac{x_i}{\sqrt{(N)}}, \frac{y_i}{\sqrt{(N)}})$ în vectorul preprocesat. Pentru cazul în care x_i și y_i nu sunt multiplii ai lui $\sqrt{(N)}$ înseamnă că trebuie să parcurgem liniar blocurile fragmentate de început și de final.

Update În cazul în care se dorește modificarea unui element din vectorul inițial este nevoie de a recalcula minimul blocului în care se găsește elementul respectiv. După ce s-a dat update valorii în vectorul inițial, căutăm indexul blocului ce cuprinde elementul respectiv și recalculăm minimul său, parcurgând cele $\sqrt{N}$ elemente din bloc.

Algorithm 1 Interogări (x_i, y_i) Square Root Decomposition

while x_i *not multiple of* $\sqrt{(N)}$ **do**
  find min for the first block in query
  x_i++
end while

for x = $\frac{x_i}{\sqrt{(N)}}$; x <= $\frac{y_i}{\sqrt{(N)}}$; x ++ **do**
  find min of blocks using $PREP[x]$
end for

while x != y_i **do**
  find min for the last block in query
  $x++$
end while

Algorithm 2 Preprocesare Square Root Decomposition

```
for x = 1; x <= N; x ++ do
    find min for each block and store in PREP
end for
```

Sparse Table *Preprocesare* Algoritmul propune preprocesarea datelor de intrare într-o matrice rară ce va reține toate minimele pentru subintervalele din vectorul dat începând de la o poziție i și având lungimea de 2^j. Vom codifica o matrice de preprocesare:

$$PREP[i][j] = min(A[k]), k = i : (i + 2^j - 1) \tag{4}$$

Vom demonstra în cele ce urmează că orice interval interogat (x_i, y_i) se poate scrie ca reuniunea a două intervale codificate în matricea preprocesată.

Demonstrație. [7] Fie intervalul interogat (x_i, y_i), unde știm că $x_i \leq y_i$. Fie p cel mai mare număr întreg astfel încat $x_i + 2^p \leq y_i$. Astfel, vom considera intervalele $(x_i, x_i + 2^p)$ și $(y_i - 2^p, y_i)$. Dacă cele două intervale nu s-ar suprapune ar implica $x_i + 2^p \leq y_i - 2^p$, de unde rezultă că $x_i + 2^{p+1} \leq y_i$. De aici rezultă contradicție deoarece am presupus inițial că p este cea mai mare putere astfel $x_i + 2^p \leq y_i$.

Matricea de preprocesare va reține pe prima coloana minimul pentru fiecare interval de dimensiune 1, având deci N linii. Cât despre numărul de coloane, vom codifica toate minimele pornind de la i cu lungime 1, 2, 4, 8, până ajungem la capătul vectorului. Vom avea astfel, $log(N)$ coloane.

Matricea se construiește progresiv, pe coloane, folosind o metodă de programare dinamică. Astfel, pentru a afla minimul unui interval ce începe de la poziția i având dimensiunea de 2^j, vom considera minimul dintre minimele codificate de intervalele $(i, i+2^{j-1})$ și $(i+2^{j-1}, i+2^j-1)$ ce au fost calculate la un pas anterior.

Interogări Pentru orice interval (x_i, y_i) se caută cel mai mare număr p astfel încât $2^p \leq j - i + 1$, urmând ca rezultatul să fie minimul dintre $(i, j + 2^p - 1)$ și $(j - 2^p + 1, j)$. Astfel, obținem răspunsul direct bazându-ne pe demonstrația anterioară.

Update Algoritmul descris se comportă ideal pentru interogări pe vectori immutabili, care nu se modifică. Dacă dorim, însă, modificarea unui element din vector matricea trebuie refăcută.
Modificarea unui element implică modificarea unui interval de un element, a 2 intervale a câte 2 elemente, a 4 intervale a câte 4 elemente și așa mai departe. Altfel zis, pe prima coloana se modifică un singur element, pe a 2a 2 elemente, pe a 3a 4 elemente etc. Elementele care se modifică la un pas i sunt dictate de un element modificat la pasul interior $i - 1$.

Algorithm 3 Preprocesare Sparse Table

for i = 0; i < N; i ++ **do**
  PREP[i][0] = v[i]
end for

for j = 1; j $\leq log(N)$; j++ **do**
  for i = 0; i + 2^j < N; i++ **do**
    $PREP[i][j] = min(PREP[i + 2^{j-1}][j-1], PREP[i][j-1])$
  end for
end for

Algorithm 4 Interogări Sparse Table

return $min(PREP[x_i][k], PREP[y_i - 2^k + 1][k])$

Segment Trees *Preprocesare* Preprocesăm minimele pentru fiecare interval într-un arbore de intervale, reținut sub forma unui arbore binar echilibrat. Rădăcina arborelui va codifica minimul întregului vector $min(A[k]), k = 0 : N$, subarborele stăng minimul pentru prima jumătate a intervalului din nodul părinte $min(A[k]), k = 0 : N/2$, iar subarborele drept minimul pentru a doua jumătate $min(A[k]), k = N/2 : N$. Se procedează identic recursiv pentru fiecare nod până se ajunge la frunze, ce vor codifica minimul pentru intervale de un singur element, adică elementul respectiv din vector. Astfel, fiecare nod intern codifică minimul tuturor frunzelor sale descendente.
Întrucat la fiecare apel se împarte intervalul în alte 2 intervale, arborele binar obținut va fi unul de tip întreg, echilibrat, cu înălțimea de $log(N)$. Astfel, numărul total de noduri va fi $2^{log(N)} - 1$. Arborele va fi reținut sub forma de vector.
Construcția arborelui are la bază un apel recursiv. Condiția de oprire a recursivității este de a ajunge la un interval de dimensiune 1, însemnand că am ajuns la o frunză, caz în care arborele reține elementul din vector. Pentru un nod intern, mai întai se apelează recursiv constructorul pentru subarborele stang și subarborele drept, urmând să reținem în nodul dat minimul dintre cele 2 valori calculate anterior.

Interogări Pentru a răspunde la interogări ne folosim de un apel recursiv ce pornește din rădăcină și se oprește când intervalul codificat de nod este inclus in cel interogat. Pentru fiecare apel din interogare distingem următoarele cazuri:

A. Intervalul codificat de nod este cuprins în totalitate în intervalul căutat
B. Intervalul codificat de nod este cuprins parțial in intervalul căutat
C. Intervalul interogat nu se află în intervalul codificat de nod

Cazuri ce se soluționează astfel:

A. Se returnează valoarea din nodul respectiv
B. Se continuă apelul recursiv pentru subarborii stang și drept

C. Se returnează o valoare MAX

Update O operație de update se soluționează prin găsirea și modificarea frunzei în cadrul arborelui ce codifică elementul pe care vrem să îl modificăm și apoi să dăm update printr-o abordare bottom-up a tuturor intervalelor codificate de nodurile din drumul de la frunză către rădăcină.

Algorithm 5 Preprocesare Arbori de Intervale

if $left = right$ **then**
  $node = value\ in\ array$
else
  $node = min(find - min(left\ interval),\ find - min(right\ interval))$
end if

Algorithm 6 Interogare Arbori de Intervale

if $interval(node) \subset (x_i, y_i)$ **then**
  $return\ value(node)$
end if

if $interval(node) \not\subset (x_i, y_i)$ **then**
  $return\ MAX$
end if

$RMQ(left, middle)$
$RMQ(middle, right)$

2.2 Analiza Complexității soluțiilor

Square Root Decomposition Algoritmul prezentat este, de fapt, un caz favorabil al algormtului de *Block Decomposition*.[1] Considerăm cazul general în care împărțim vectorul dat în $\frac{N}{b}$ blocuri de dimensiune b.
Indiferent de cum alegem valoarea lui b complexitatea preprocesării va fi de $O(N)$ deoarece trebuie să iterăm prin întreg vectorul pentru a găsi minimul pentru fiecare bloc. Mai exact, avem nevoie de b pași pentru a găsi minimul unui singur bloc, iar în total avem $\frac{N}{b}$ blocuri.
Cat despre complexitatea interogărilor, pentru a parcurge blocul inițial și blocul final avem nevoie de câte b pași, iar pentru a itera prin blocurile întregi conținute în interval vom avea maxim $\frac{N}{b}$ pași. Astfel, numărul total de pași este:

$$b + \frac{N}{b} + b = 2 \times b + \frac{N}{b} \tag{5}$$

Căutăm să obținem minimul acestei funcții, pentru a minimiza complexitatea algoritmului, deci o derivăm în funcție de b și impunem derivatei să fie nulă. Astfel obținem:

$$2 - \frac{N}{b^2} = 0 => b^2 = \frac{N}{2} => b = \sqrt{\frac{N}{2}} \tag{6}$$

Se aproximează b ca fiind $\sqrt(N)$ pentru algoritmul Square Root Decomposition. În final, $O(2 \times b + \frac{N}{b})$ pentru b specificat anterior va da $O(\sqrt(N))$. Astfel, complexitatea per interogare este de $O(\sqrt{N})$
În cazul unei operații de update avem nevoie de $\sqrt{N}$ operații pentru a parcurge blocul ce conține indicele la care facem update.
Pentru toate cele trei operații descrise anterior folosim un vector auxiliar având $\sqrt{N}$ elemente.
Obținem astfel rezultatele:

Complexitate Preprocesare	Complexitate Interogare	Complexitate Update	Complexitate Memorie Suplimentară
$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N})$

Sparse Table Pentru preprocesarea avem maxim $log(N)$ intervale pentru fiecare poziție de început aleasă, în total fiind N variante, deci avem complexitate $O(Nlog(N))$ De altfel, aceeași complexitate poate fi calculată urmărind pseudocodul prezentat anterior. Primul for care fixează rezultatele pentru fiecare interval a câte un singur element are $O(N)$ operații. Pentru următoarele foruri imbricate contorizăm pentru fiecare j câte $N - 2^j$ operații. Obținem, astfel, complexitatea:

$$\sum_{j=1}^{logN}(N - 2^j) = \sum_{j=1}^{log(N)} N - \sum_{j=1}^{log(N)} 2^j = Nlog(N) - N \subset O(Nlog(N))$$

Pentru interogare, în schimb, avem timp constant $O(1)$ urmărind logica prezentată anterior.
Pentru un update este nevoie de modificarea unui element pe prima coloana, 2 elemente pe a 2a coloana, 2^2 elemente pe a 3a coloana și tot așa. Astfel, numărul maxim de elemente modificate va fi:

$$\sum_{j=0}^{logN} 2^j = 2 \times 2^{log(N)} - 1 = 2 \times N - 1 \subset O(N)$$

În toate cele 3 operații avem nevoie de o matrice suplimentară de N linii și $log(N)$ coloane, de unde rezultă o complexitate de $O(Nlog(N))$. Complexitățile finale sunt:

Complexitate Preprocesare	Complexitate Interogare	Complexitate Update	Complexitate Memorie Suplimentară
$O(Nlog(N))$	$O(1)$	$O(N)$	$O(Nlog(N))$

Segment Trees Pentru operația de preprocesare, parcurgem și asignăm fiecărui nod din arbore minimul pentru intervalul codificat. În total sunt $2 \times N - 1$ noduri, deci complexitatea pentru preprocesare este $O(N)$.
Pentru o operație de update se parcurge drumul de la frunză la rădăcină. Cum înălțimea arborelui este $log(N)$, fiind unul echilibrat, vom avea complexitatea $O(log(N))$ per update.
Pentru operația de interogare, la fiecare apel avem cel mult al 2 alte apeluri recursive de funcție, în careînaintăm în adâncimea arborelui cu un nivel. Cum arborele are adâncimea de $log(N)$, rezultă că complexitatea unei interogări este de $O(logN)$.
Pentru cele 3 operații folosim un vector auxiliar de dimensiune $2 * N - 1$ pentru a reține arborele, de unde rezultă că avem complexitate de spațiu $O(N)$

Complexitate Preprocesare	Complexitate Interogare	Complexitate Update	Complexitate Memorie Suplimentară
$O(N)$	$O(log(N))$	$O(log(N))$	$O(N)$

Rezultate finale

Complexitate Totală			Complexitate Memorie		
Square Root	Sparse Table	Segment Trees	Square Root	Sparse Table	Segment Trees
$O(\sqrt{N} + M\sqrt{N})$	$O(Nlog(N) + M)$	$O(N + Mlog(N))$	$O(\sqrt{N})$	$O(N)$	$O(Nlog(N))$

2.3 Avantaje și Dezavantaje

Square Root Decomposition *Avantaje:* Un prim avantaj al acestei metode constă în simplitatea și ușurința înțelegerii codului. Comparativ cu celelalte soluții propuse, această metodă nu necesită o structura de date auxiliară sau cunoștințe de programare dinamică, ideea ce stă la bază fiind ușor de dedus și rapid de implementat.
Un al doilea avantaj constă în memoria auxiliară relativ redusă comparativ cu celelalte două metode, algoritmul fiind, de altfel, cel care se comportă cel mai bine din punct de vedere al complexității de spațiu. Deducem, astfel, că este de preferat să folosim această metodă în situația unui număr mult mai mare de elemente N, comparativ cu numărul de interogări M, N >> M. În practică s-a observat că metoda răspunde rapid la teste ce cuprind date de intrare N și M de dimensiune redusă.
Dezavantaje Un dezavantaj se poate observa în orice alt test propus în cadrul studiului, în care N și M cresc de la dimensiuni mici la dimensiuni medii sau chiar mari, teste în care algoritmul eșuează ca timp.

Sparse Table *Avantaje:* Printre avantajele matricelor sparse se numără răspunderea la interogări în timp constant $O(1)$. Astfel, algoritmul este ideal în situațiile în care numărul de interogări este mai mare decat numărul de elemente din vector, $N << M$.
Dezavantaje: Deși răspunde rapid la interogări, acest tip de matrice are nevoie de un timp de preprocesare mai costisitor $O(Nlog(N))$ și de o memorie auxiliară ridicată comparativ cu celelalte metode propuse $O(Nlog(N))$. Deducem, astfel, că această metodă nu poate fi folosită în cazuri în care $N >> M$. Un alt dezavantaj ar fi al timpului mare necesar unei singure operații de update.

Segment Trees *Avantaje:* Principalul avantaj al acestui algoritm constă în cel mai bun comportament pentru operația de update. În plus, în practică, s-a observat un caracter echilibrat asupra celorlalte operații, respectiv timpi mai ridicați decat cei propuși de Sparse Table, dar mai reduși semnificativ decat Square Root Decomposition.
Dezavantaje: Un dezavantaj al acestei metode este complexitatea soluției, fiind nevoie de cunoștințe în structuri de date arborescente, aspect ce face soluția mai dificil de implementat relativ la celelalte două propuse.

3 Evaluare

3.1 Construirea setului de teste

Programul Generator de Teste Ne propunem generarea unui număr mare de teste pe un set de date în care valorile variază exponențial atat ca dimensiune, cat și ca valoare. Pentru a ne asigura corectitudinea indiferent de complexitatea datelor de intrare, am implementat în generator și o soluție banală, care deși ne dezavantajează ca timp, asigură corectitudinea datelor de ieșire.
Astfel, programul propus generează date de intrare random folosind functia *srand*, manipulate ulterior de noi, le scrie in fișierele corespunzătoare, iar pentru scrierea rezultatului în fișierele de ieșire, rezolvă problema propusă prin metoda banală.În plus, această decizie a facilitat și testarea graduală a algoritmilor, ajungând în final ca toți cei 4 algoritmi propuși să ofere aceleași rezultate, confirmând corectitudinea fiecăruia.

Criterii de comparație

1. Dimensiunea datelor de intrare (N, M)

2. Ordinea elementelor în vectorul dat (crescător, descrescător, aleator)

3. Dimensiunea intervalelor interogate

4. Corner Cases

   În categoria corner case-urilor trecem testul 77, în care elementele din vector sunt constante și testul 76 în care vectorul este nul. În plus, am avut în vedere ca în cazul testelor [16, 30] valorile din vector să fie de tip SHORT, iar N și M de tip MEDIUM pentru a impune repetiția elementelor din vector. Aceste teste sunt menite să ajute la verificarea corectitudinii algoritmului.

Dimensiuni date

1. SHORT = 10^2
2. MEDIUM = 10^4
3. LARGE = 10^6

3.2 Specificațiile sistemului

Algoritmii au fost rulați și testați pe un sistem de operare Ubuntu 8.0 pe o mașină având următoarele specificații de sistem:

1. Procesor Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz, Arhitectură x86, Little-Endian, 4 Core-uri
2. Total Mem 8022984 KB, Cache Size 8192 KB

Observaţii sistem Întrucât timpii de execuție ai unui program nu depind în totalitate doar de programul în sine, ci și de timpul de loading al CPU-ului, pentru a asigura acuratețea rezultatelor, ne-am folosit de un BASH Script ce rulează un algoritm pe același input de 100 de ori și calculează media aritmetică a timpilor obținuți. Rezultatele anterioare reprezintă rezultatul final al măsurătorilor.

3.3 Generarea Rezultatelor. Grafice. Tabele

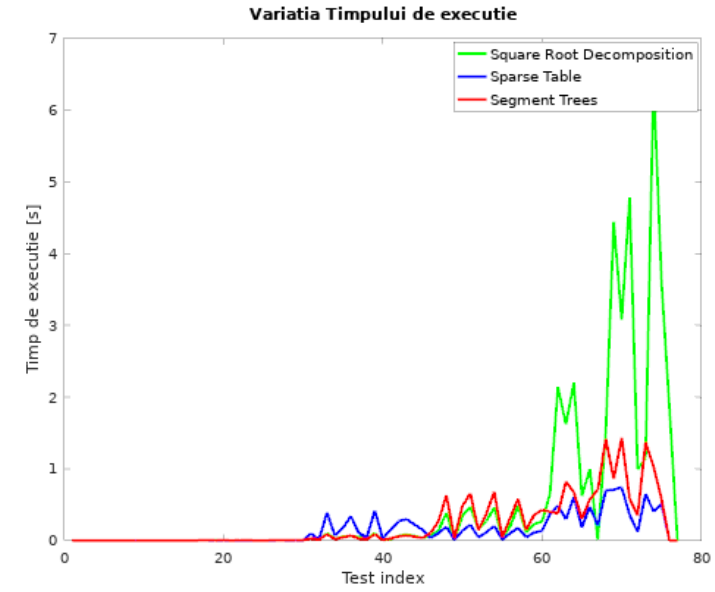
Ierarhie Teste

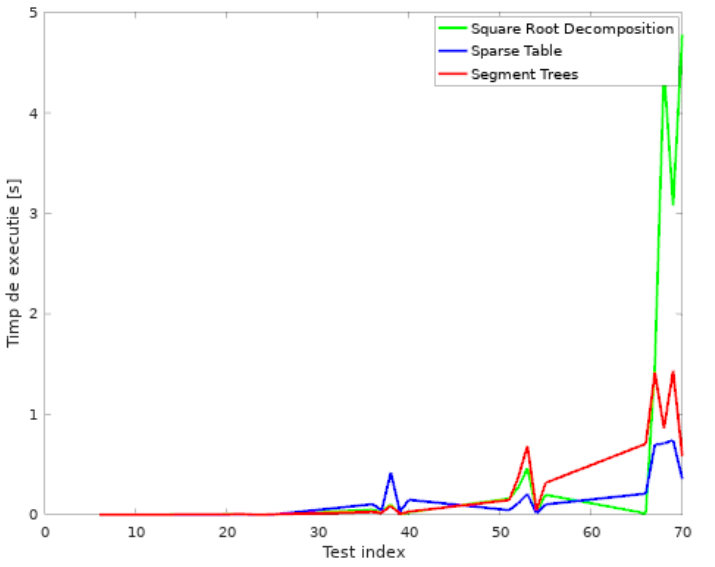
Nr. Test	Dimensiuni Date	Ordinea elementelor	Dimensiune intervale
[1, 15]	N = SHORT M = SHORT	random [1, 5]	mici [1] mari [2]
		crescător [6, 10]	mici [6] mari [7]
		descrescător [11, 15]	mici [6] mari[7]
[16, 30]	N = MEDIUM M = MEDIUM	random [16, 20]	mici [16] mari [17]
		crescător [21, 25]	mici [21] mari [22]
		descrescător [26, 30]	mici [26] mari[27]
[31, 45]	N = LARGE M = MEDIUM	random [31, 35]	mici [31] mari [32]
		crescător [36, 40]	mici [36] mari [37]
		descrescător [41, 45]	mici [41] mari[42]
[46, 60]	N = MEDIUM M = LARGE	random [46, 50]	mici [46] mari [47]
		crescător [51, 55]	mici [51] mari [52]
		descrescător [56, 60]	mici [56] mari[57]
[61, 75]	N = LARGE M = LARGE	random [61, 65]	mici [61] mari [62]
		crescător [66, 70]	mici [566] mari [67]
		descrescător [71, 75]	mici [71] mari[72]

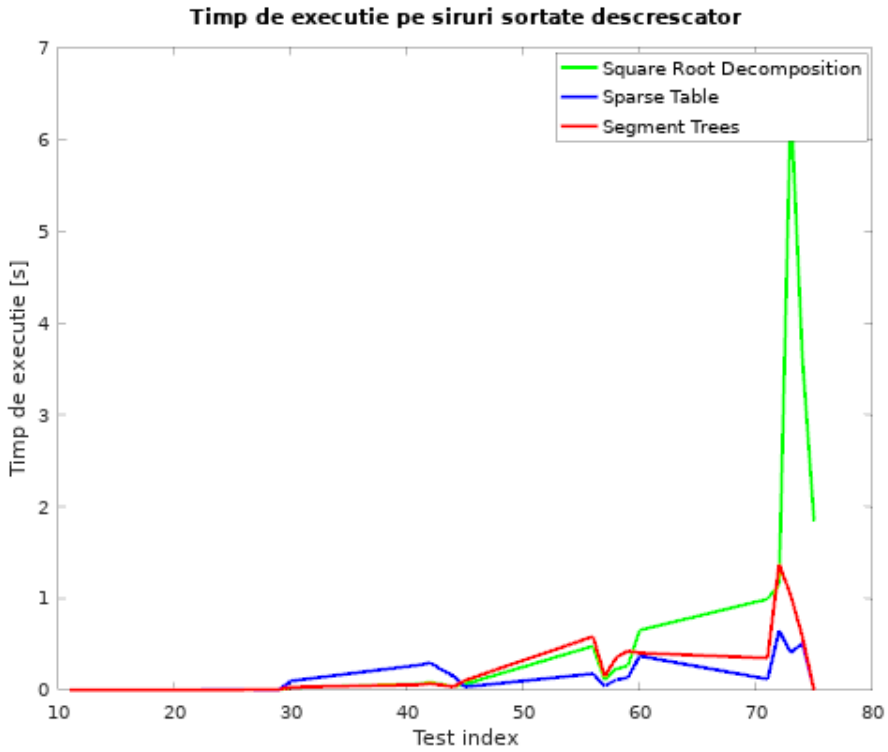
Nr. Test	N	M	Ordine Elemente	Timp de execuție s		
				Square Root	Sparse Table	Segment Trees
1	18	98	aleator	0.0002352	0.0001630	0.0004515
2	42	11	aleator	0.0001949	0.0001876	0.0003176
3	91	17	aleator	0.0001301	0.0002245	0.0003022
4	32	82	aleator	0.0000050	0.0001660	0.0002731
5	96	78	aleator	0.0003787	0.0002943	0.0003254
6	74	59	crescător	0.0001562	0.0002792	0.0002818
7	47	46	crescător	0.0001481	0.0002847	0.0003413
8	82	22	crescător	0.0001624	0.0002433	0.0002590
9	11	1	crescător	0.0002950	0.0002005	0.0002404
10	84	43	crescător	0.0003061	0.0001840	0.0003212
11	98	5	descrescător	0.0002845	0.0002165	0.0002564
12	13	98	descrescător	0.0001591	0.0002666	0.0002487
13	68	34	descrescător	0.0001610	0.0002181	0.0002040
14	97	35	descrescător	0.0002282	0.0002896	0.0002633
15	36	18	descrescător	0.0001516	0.0002237	0.0002852
16	2617	1212	aleator	0.0011870	0.0015296	0.0011799
17	4122	4929	aleator	0.0045301	0.0040247	0.0041254
18	9386	5512	aleator	0.0048347	0.0036410	0.0058886
19	4789	2885	aleator	0.0043198	0.0024700	0.0029590
20	7530	7382	aleator	0.0068460	0.0037145	0.0074180
21	3198	1575	crescător	0.0018516	0.0010568	0.0015677
22	2848	8380	crescător	0.0071399	0.0027336	0.0063440
23	5781	6755	crescător	0.0050462	0.0032543	0.0064632
24	1894	366	crescător	0.0005201	0.0005826	0.0006039
25	5578	382	crescător	0.0008396	0.0020636	0.0008953
26	312	8746	descrescător	0.0031196	0.0021050	0.0049396
27	7886	3297	descrescător	0.0039842	0.0046582	0.0035747
28	3350	9630	descrescător	0.0062041	0.0030963	0.0084160
29	1585	4397	descrescător	0.0031623	0.0021830	0.0044943
30	2776	9047	descrescător	0.0063167	0.0027001	0.0071833
31	246740	9619	aleator	0.0309455	0.1000790	0.0323782
32	49375	7963	aleator	0.0191996	0.0169455	0.0126368
33	802475	8886	aleator	0.1024340	0.3914130	0.0892134
34	162403	4405	aleator	0.0260655	0.0680767	0.0212660

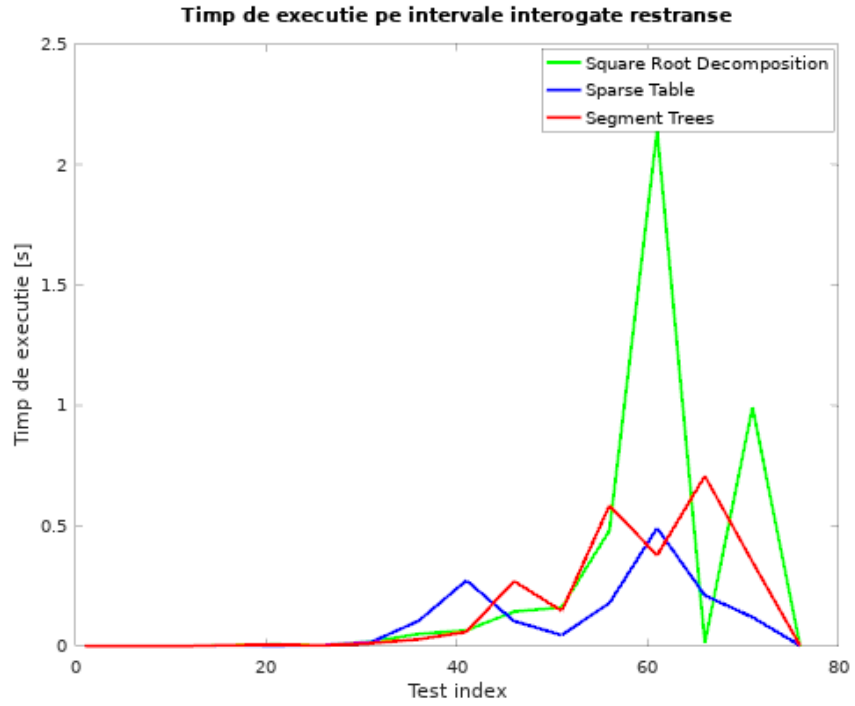
Nr. Test	N	M	Ordine Elemente	Timp de execuție s		
				Square Root	Sparse Table	Segment Trees
35	374232	8269	aleator	0.0565213	0.1773020	0.0465889
36	686196	7811	crescător	0.0634584	0.3402690	0.0709951
37	246185	7505	crescător	0.0510010	0.1053300	0.0287679
38	111516	8955	crescător	0.0264768	0.0442523	0.0201467
39	861483	8575	crescător	0.1056150	0.4206850	0.0887802
40	92611	544	crescător	0.0076372	0.0324749	0.0092113
41	322871	1503	descrescător	0.0249822	0.1493340	0.0318272
42	567782	3957	descrescător	0.0658885	0.2731340	0.0593486
43	599412	9977	descrescător	0.0857633	0.2978180	0.0724140
44	473625	7580	descrescător	0.0639738	0.2217350	0.0549711
45	323483	4131	descrescător	0.0363068	0.1526170	0.0365863
46	2619	154441	aleator	0.0652734	0.0356515	0.1079100
47	281	486561	aleator	0.1436600	0.1046240	0.2701490
48	2136	831242	aleator	0.3867720	0.1943150	0.6333520
49	683	50700	aleator	0.0181485	0.0110956	0.0325022
50	6238	576148	aleator	0.3439210	0.1399180	0.4823370
51	7572	942522	crescător	0.4726480	0.2209760	0.6581440
52	9164	193964	crescător	0.1612860	0.0460482	0.1463440
53	4378	470018	crescător	0.2696210	0.1152270	0.3731300
54	4100	857622	crescător	0.4629300	0.2077410	0.6855970
55	8130	47959	crescător	0.0344539	0.0139880	0.0442821
56	4295	444202	descrescător	0.1989640	0.1032600	0.3176040
57	3025	813311	descrescător	0.4803100	0.1792860	0.5840460
58	4981	184151	descrescător	0.1092600	0.0459697	0.1496660
59	2846	464765	descrescător	0.2325800	0.1134120	0.3564490
60	2164	575017	descrescător	0.2657990	0.1357100	0.4271270
61	488535	349329	aleator	0.6528310	0.3724750	0.4048030
62	835245	261110	aleator	2.1406300	0.4908000	0.3775630
63	165136	632197	aleator	1.6183100	0.2985000	0.8229550
64	876378	380980	aleator	2.2027500	0.6053230	0.6727210
65	240327	197583	aleator	0.6211610	0.1823020	0.3021680
66	531981	519463	crescător	0.9970870	0.4734040	0.5777880
67	44401	804847	crescător	0.0141311	0.2110530	0.7065340
68	610829	931383	crescător	1.4973600	0.6983470	1.4163000
69	973124	511735	crescător	4.4439000	0.7093460	0.8599570
70	692258	963659	crescător	3.0756000	0.7422840	1.4356100
71	352435	550984	descrescător	4.7821000	0.3578250	0.5842100
72	109153	329726	descrescător	0.9921520	0.1204120	0.3500610
73	647554	846988	descrescător	1.1497100	0.6487730	1.3720900
74	337434	683602	descrescător	6.4323900	0.4084830	1.0314700
75	875816	297213	descrescător	3.6098300	0.5102890	0.5836320
76	9151	0	null	1.8379900	0.0032793	0.0010299
77	36	5268	constant	0.0008135	0.0027314	0.0022502

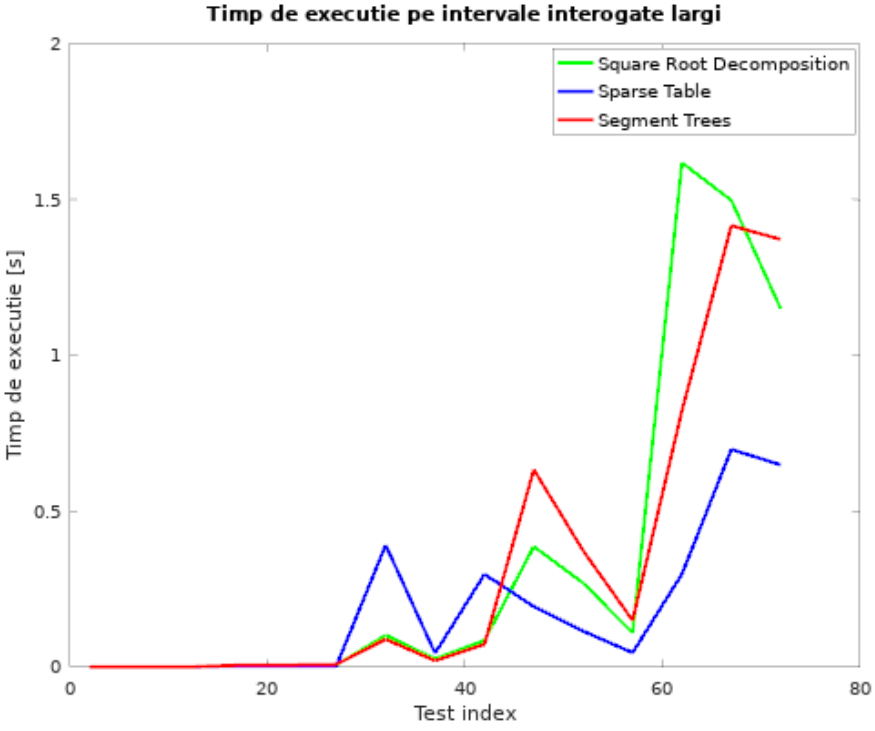
Variația Timpului de execuție în funcție de organizarea datelor de intrare Pentru început, vom ilustra comparativ cum se comportă cei trei algoritmi descriși pe toate cele 77 de teste generate.Întrucât am impus restricțiile prezentate anterior testelor, vom ilustra separat rezultatele pentru fiecare caz studiat.





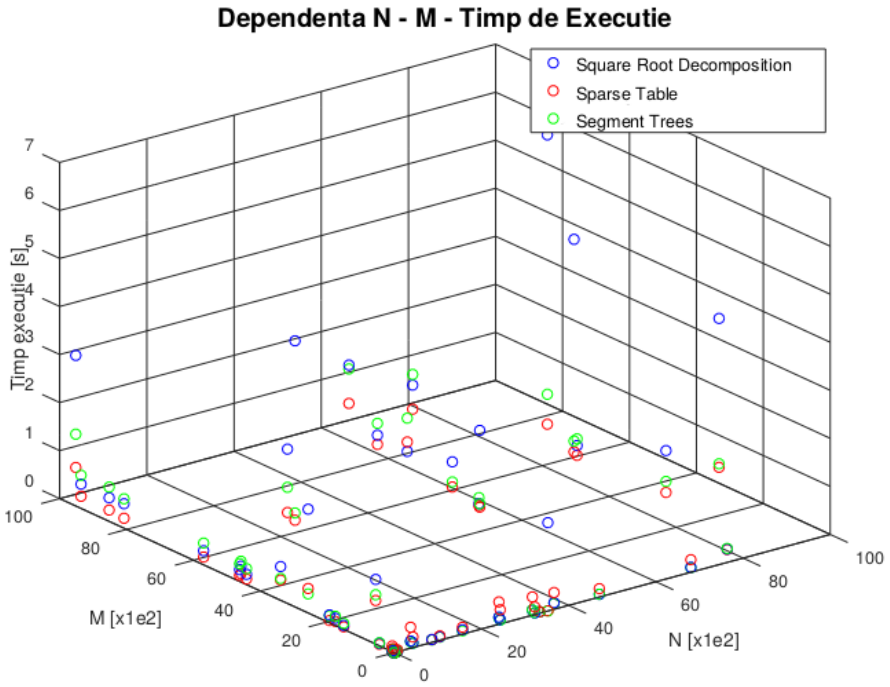

Timp de executie pe siruri sortate descrescator
Square Root Decomposition
Sparse Table
Segment Trees
Timp de executie [s]
Test index
0
1
2
3
4
5
6
7
10
20
30
40
50
60
70
80


Timp de executie pe intervale interogate restranse
Square Root Decomposition
Sparse Table
Segment Trees
Timp de executie [s]
Test index
0
0.5
1
1.5
2
2.5
0
20
40
60
80


Timp de executie pe intervale interogate largi
Square Root Decomposition
Sparse Table
Segment Trees
Timp de executie [s]
Test index
0
0.5
1
1.5
2
0
20
40
60
80

Nr. Test	N	M	Intervale	Timp de execuție s		
				Square Root	Sparse Table	Segment Trees
1	18	98	mici	0.0002352	0.0001630	0.0004515
2	42	11	mari	0.0001949	0.0002943	0.0003254
6	74	59	mici	0.0001562	0.0002792	0.0002818
7	47	46	mari	0.0001481	0.0001840	0.0003212
11	98	5	mici	0.0002845	0.0002165	0.0002564
12	13	98	mari	0.0001591	0.0002237	0.0002852
16	2617	1212	mici	0.0011870	0.0015296	0.0011799
17	4122	4929	mari	0.0045301	0.0037145	0.0074180
21	3198	1575	mici	0.0018516	0.0010568	0.0015677
22	2848	8380	mari	0.0071399	0.0020636	0.0008953
26	312	8746	mici	0.0031196	0.0021050	0.0049396
27	7886	3297	mari	0.0039842	0.0027001	0.0071833
31	246740	9619	mici	0.0309455	0.1000790	0.0323782
32	49375	7963	mari	0.0191996	0.0169455	0.0126368
36	686196	7811	mici	0.0634584	0.3402690	0.0709951
37	246185	7505	mari	0.0510010	0.1053300	0.0287679
41	322871	1503	mici	0.0249822	0.1493340	0.0318272
42	567782	3957	mari	0.0658885	0.2731340	0.0593486
46	2619	154441	mici	0.0652734	0.0356515	0.1079100
47	281	486561	mari	0.1436600	0.1046240	0.2701490
51	7572	942522	mici	0.4726480	0.2209760	0.6581440
52	9164	193964	mari	0.1612860	0.0460482	0.1463440
56	4295	444202	mici	0.1989640	0.1032600	0.3176040
57	3025	813311	mari	0.4803100	0.1792860	0.5840460
61	488535	349329	mici	0.6528310	0.3724750	0.4048030
62	835245	261110	mari	2.1406300	0.4908000	0.3775630
66	531981	519463	mici	0.9970870	0.4734040	0.5777880
67	44401	804847	mari	0.0141311	0.2110530	0.7065340

Variația Timpului de execuție în funcție dimensiunile N și M Întrucât nu putem trage o concluzie directă asupra variației timpului de execuție în funcție de un singur parametru (N sau M), propunem o serie de grafice ce ilustrează în paralel dependentă N - M - timp execuție.



3.4 Interpretarea Rezultatelor

Din graficele precedente putem distinge comparativ cum evoluează fiecare algoritm din punct de vedere al timpului de execuție gradual de la teste cu date de intrare mici, până la date mari.
Se poate distinge cum o dată ce datele de intrare cresc, algoritmul Square Root Decomposition, având cea mai mare complexitate eșuează comparativ cu ceilalți algoritmi.
Inițial, datele de intrare fiind reduse, cei 3 algoritmi se comportă asemănător.
De menționat este faptul că se poate observa în intervalul testelor 30 - 40 o tendință a algoritmului Sparse Table de a impune timpi mai ridicați dar acest lucru se argumentează prin impunerea restricției ca N să fie cu un ordin de 1e3 mai mare decat M. Algoritmul compensează la rezultatele timpilor pe testele cu indexii 40-60, unde N este cu ordin 1e3 mai mic decat M. Se poate observa cum în cazul $N >> M$ acesta este cel mai rapid.

Soluția ce utilizează segment trees reușește pe teste cu inputuri mari să obțină timpi considerabili mai buni decat cei ai lui Square Root Decomposition. Putem considera că acest algoritm se comportă echilibrat, un avantaj major al său fiind cel al update-urilor, aspect ce va fi discutat în secțiunile următoare.

3.5 Discuţie dimensiuni N M

Bazându-ne pe rezultatele anterioare dezbatem următoarele cazuri:

1. N >> M
2. N << M
3. N ≈ M

1. Pentru acest caz ar putea fi de preferat să folosim algoritmul Square Root Decomposition întrucât are atât complexitatea de timp de preprocesare mai redusă comparativ cu ceilalți algoritmi, cat și complexitatea de memorie este mai mică. Desigur, această afirmație ar putea fi contrabătută de un set considerabil de queries, caz în care am putea prefera algoritmul ce folosește Segment Trees. O altă observație ce rezultă din grafice se bazează pe eșecul algoritmului Sparse Table pe acest caz, a se urmări evoluția timpilor pe testele cu indexii cuprinși între 30-40.

2. Întrucat algoritmul Sparse Table are complexitatea de timp per interogare O(1), este de preferat să fie folosit acesta, a se urmări evoluția timpilor pe teste cu indexii cuprinși între 45-60. Algoritmul are un timp de preprocesare costisitor, însă având un număr mult mai ridicat de interogări se compensează.

3. Putem translata tabelul final al complexităților de timp astfel:

Complexitate Totală			Complexitate Memorie		
Square Root	Sparse Table	Segment Trees	Square Root	Sparse Table	Segment Trees
$O(N\sqrt{N})$	$O(Nlog(N))$	$O(Nlog(N))$	$O(\sqrt{N})$	$O(N)$	$O(Nlog(N))$

Știind că $log(N) < \sqrt{N}$ tragem concluzia că ar fi de preferat să folosim algoritmul ce se bazează pe Sparse Table pentru acest caz, aspect confirmat de altfel și în graficul timpilor de execuție, pentru testele cuprinse între indexii 61-75.

3.6 Problema Updateurilor

Detalii despre update-uri În cadrul cercetării, am urmărit problema update-urilor singulare, fără update-uri pe intervale sau adăugări, respectiv eliminări ale elementelor din vectori. Astfel, complexitatea de memorie a celor 3 algoritmi rămâne neschimbată. În contextul propus, M reprezintă numărul total de comenzi introduse de utilizator, comenzi codificate astfel:

1. $U(a, b)$ - Update al valorii b la indexul a
2. $Q(a, b)$ - Interogare în intervalul a b

Vom nota în cele ce urmează m_u, numărul de update-uri și m_q numărul de interogări, în total fiind $m_u + m_q = m$.

Rezultate Generale

Complexitate Totală		
Square Root	Sparse Table	Segment Trees
$O(\sqrt{N} + m_q\sqrt{N} + m_u\sqrt{(N)})$	$O(Nlog(N) + m_q + m_uN)$	$O(N + m_qlog(N) + m_ulog(N))$

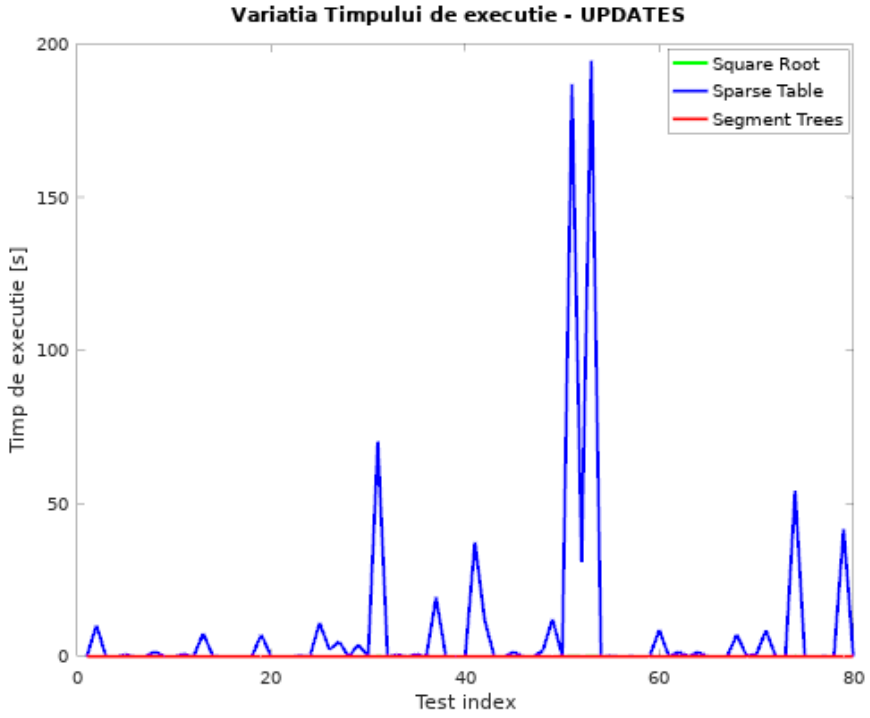
Întrucât algoritmul Sparse Table eșuează în acest context am decis să generăm teste cu date de intrare mai reduse dimensional.

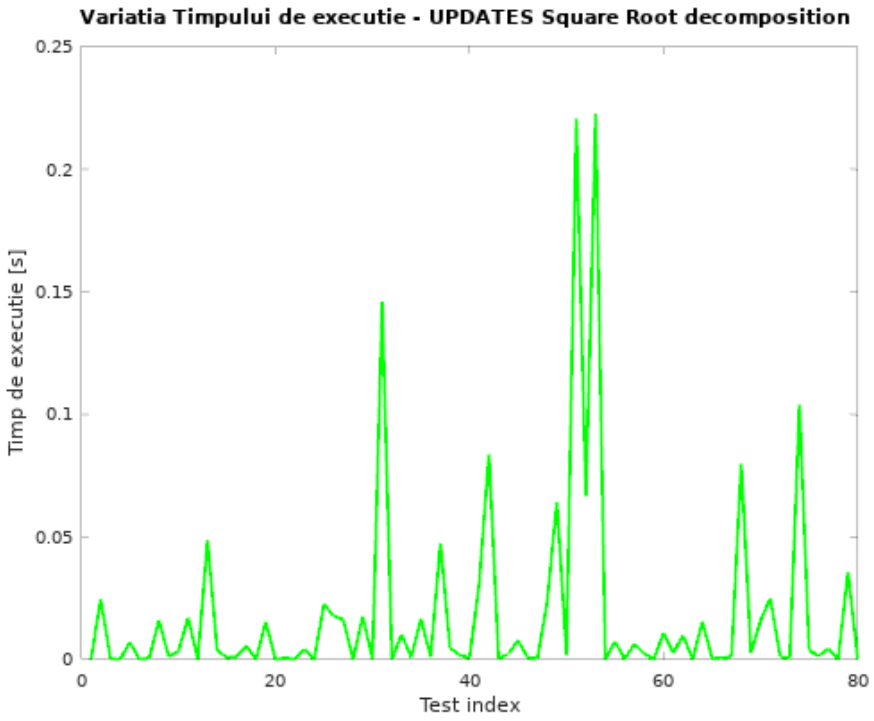
Dimensiuni date

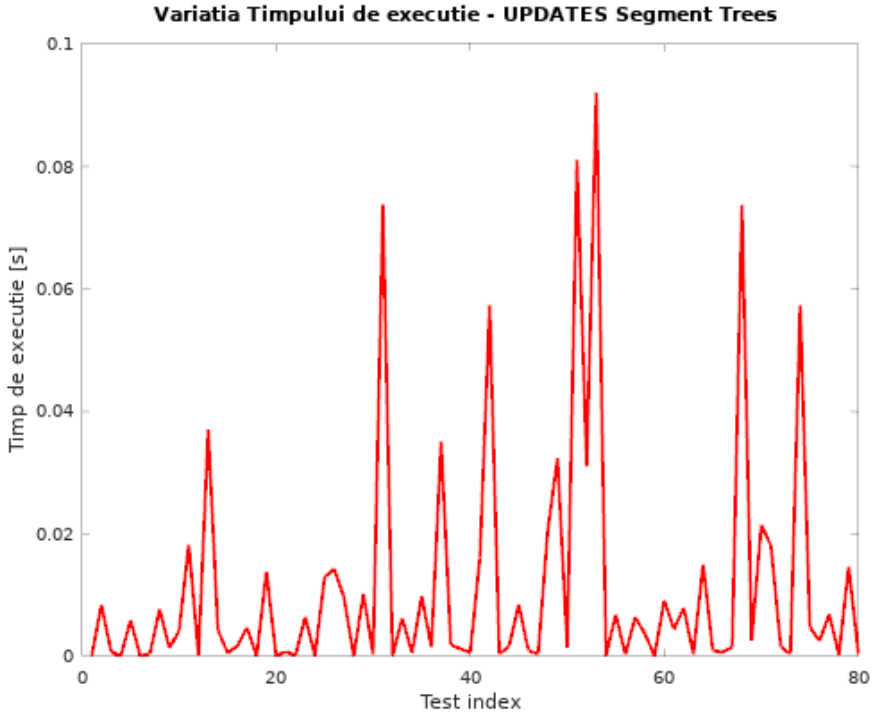
1. SHORT = $1e2$
2. MEDIUM = $1e3$
3. MDEIUM2 = $1e4$
4. LARGE = $1e5$

Concluzii update-uri Deși soluționează problema generală în O(1) per interogare, în situația în care se doresc update-uri, Sparse Table generează timpi care pot ajunge la 200 secunde pe teste cu un număr considerabil de update-uri și un ordin ridicat de mărime pentru datele de intrare. Se pot urmări rezultatele pentru indexii testelor 41-50. Algoritmul care generează, în schimb cei mai optimi timpi este Segment Trees, iar cu o diferență mică , de aproape 0.1 secunde, se află Square Root Decomposition.

Interpretarea rezultatelor obținute *Variatia timpului de executie*




Variatia Timpului de executie - UPDATES Square Root decomposition
Timp de executie [s]
Test index
0.25
0.2
0.15
0.1
0.05
0
0
20
40
60
80


Variatia Timpului de executie - UPDATES Segment Trees
Timp de executie [s]
Test index
0.1
0.08
0.06
0.04
0.02
0
0
20
40
60
80

Nr. Test	Dimensiuni Date	Ordinea elementelor	U ? Q
[1, 10]	N = SHORT M = SHORT	random [1, 4]	<< [2] >> [4]
		crescător [5, 7]	<< [6] >> [7]
		descrescător [8, 10]	<< [9] >> [10]
[11, 20]	N = MEDIUM M = MEDIUM	random [11, 14]	<< [12] >> [13]
		crescător [11, 20]	<< [12] >> [13]
		descrescător [15, 17]	<< [16] >>[17]
[21, 30]	N = MEDIUM2 M = MEDIUM	random [21, 24]	<< [22] >> [23]
		crescător [25, 27]	<< [26] >> [27]
		descrescător [28, 30]	<< [29] >>[30]
[31, 40]	N = MEDIUM1 M = MEDIUM2	random [31, 34]	<< [32] >> [33]
		crescător [35, 37]	<< [36] >> [37]
		descrescător [38, 40]	<< [39] >>[40]
[41, 50]	N = MEDIUM2 M = MEDIUM2	random [41, 44]	<< [42] >> [43]
		crescător [45, 47]	<< [46] >> [47]
		descrescător [48, 50]	<< [49] >>[50]
[51, 60]	N = MEDIUM2 M = LARGE	random [51, 54]	<< [52] >> [53]
		crescător [55, 57]	<< [56] >> [57]
		descrescător [58, 60]	<< [59] >>[60]
[61, 70]	N = LARGE M = LARGE	random [61, 64]	<< [62] >> [63]
		crescător [65, 67]	<< [66] >> [67]
		descrescător [68, 70]	<< [69] >>[70]
[71, 80]	N = LARGE M = LARGE	random [71, 74]	<< [72] >> [73]
		crescător [75, 77]	<< [76] >> [77]
		descrescător [78, 80]	<< [79] >>[80]

4 Concluzii

Pentru a trage concluziile acestui studiu, propunem o viziune ierarhizată asupra celor trei algoritmi propuși.
Deși reușește să rezolve problema într-un timp relativ redus pentru date de intrare N, M mici, algoritmul *Square Root Decomposition* eșuează pe orice alt test. Astfel, în practică ar putea fi folosit exclusiv în situații în care datele de intrare sunt în număr redus.
În schimb, metoda *Sparse Table* răspunde în timp constant la interogări, aspect ce face algoritmul să fie ideal în situația în care numărul de interogări este mult mai mare decat numărul de elemente din vectorul de intrare, iar vectorul este imutabil.
Segment Trees s-a dovedit a fi o soluție echilibrată, ce are un avantaj major al complexității reduse pentru update-uri.
Propunem în cele ce urmează un studiu practic asupra problemei în secțiunea 1, cea a Auto Completion- ului. În primă instanța, putem alege să folosim algoritmul *Sparse Table* întrucăt numărul de interogări dintr-o bază de date comună unui număr larg de utilizatori este foarte ridicat. Etapa de preprocesare ar putea fi făcută înainte de a fi făcut publică baza astfel încat utilizatorul să beneficieze de timp constant per interogare. Pe de altă parte, baza de date își poate da update în funcție de rezultate statistice asupra search-urilor utilizatorilor, caz în care algoritmul *Segment Trees* ar fi ideal.
Tragem concluzia că nu există un algoritm fără dezavantaje, soluțiile avantajoase depind de contextul în care sunt utilizate.

5 Referințe

Bibliografie

1. Standford University, CS166, Data Structures Lectures, Range Minimum Queries, Part One and Two http://web.stanford.edu/class/cs166/
2. Giuseppe Ottaviano, BJ Hsu, Space-Efficient Data Structures for Top-K Completion, 2013
3. Geeksforgeeks Homepage, https://www.geeksforgeeks.org/range-minimum-query-for-static-array/. Last accessed: 15 Oct 2021
4. Geeksforgeeks Homepage, https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/. Last accessed: 15 Dec 2021
5. Topcoder Homepage, https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/
6. Geeksforgeeks Homepage, https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/. Last accessed: 15 Dec 2021
7. Brilliant Homepage, https://brilliant.org/wiki/sparse-table/. Last accessed: 15 Dec 2021
8. Opengenus Homepage, https://iq.opengenus.org/range-minimum-query-segment-tree/. Last accessed: 15 Dec 2021