Range Minimum Query

Zanficu Madalina-Valentina

Universitatea Politehnica din Bucuresti, Facultatea de Automatica si Calculatoare

Abstract. Aceasta documentatie urmareste analiza problemei gasirii elementului minim dintr-un interval, cunoscuta si sub denumirea "Range Minimum Query". Se vor oferi trei solutii eficiente pentru rezolvarea cerintei si vom interpreta solutiile oferite din punct de vedere al complexitatii temporale si spatiale pe diferite date de intrare.

Keywords: Segment Tree Algorithm · Sparse Table Algorithm · Square Root Decomposition.

1 Introducere

1.1 Descrierea probelemei rezolvate

In Computer Science, problema **Range Minimum Query (RMQ)** consta in determinarea eficienta a elementului minim ce se afla intre doi indici specificati ai unui vector. Cu alte cuvinte: detereminarea elementului minim dintr-un subtablou al tabloului dat.

Input : Se da un vector V ce contine N elemente numere intregi, si M interogari de forma: "Care este elementul minim din intervalul inchis corepunzator indicilor I, J din vectorul V, unde $I <= X <= J <= N$, iar X – pozitia elementului minim din interval ce trebuie determinata.

Output : Cu ajutorul RMQ, vom afla in mod eficient pozitia elementului minim din intervalul [I,J] pentru fiecare interogare M din input.

1.2 Exemple de aplicatii practice

Printre aplicatiile practice dezvoltate in urma identificarii elementului minim dintr-un interval dat [I, J] intr-un tablou V cu N elemente se afla determinarea celui mai mic stramos comun dintr-un arbore (Lowest Common Ancestor). De asemenea problema celui mai lung prefix intr-un sir de caractere (LCP) poate fi redusa in timp constant la problema RMQ.

RMQ in viata reala: Un exemplu concret din viata de zi cu zi in care ar putea fi folsit algoritmul RMQ este determinarea zilei dintr-un interaval de o saptamana, o luna sau un an in care rata de incidenta a cazurilor SARS-COV-19 a fost cea mai mica.

RMQ in viata reala - date care se actualizeaza constant: Un alt scenariu al aplicarii problemei RMQ ar putea fi determinarea celei mai ieftine monede dintr-un set de valute disponibile la o banca (tinand cont ca nu se doreste schimbarea in orice moneda disponibila, ci doar in cele din intervalul din query). Tabloul initial ar contine cursul pentru fiecare moneda ce corespunde unui indice i, si in fiecare zi valoarea monedei se poate modifica.

1.3 Specificarea solutiilor alese

SparseTable Algorithm Utilizeaza o structura de date ce necesita pre-procesarea datelor de intrare pentru a putea oferi raspunsul la interogari de interavale. Poate raspunde la majoritatea interogarilor din interval in O(logN), dar adevarata sa putere se manifesta in utilizarea lui pentru determinarea elementului minim din interval (Range Minimum Query), timpul de raspuns fiind de O(1).

Segment Tree Algorithm Foloseste un arbore binar complet drept structura de date ajutatoare pentru a oferi raspunsul la o interogare intr-un timp logaritmic. Nodurile interne sunt formate de jos in sus prin furinzarea informatiilor din nodurile sale copii. Astfel fiecare nod corespunde unui interval al tabloului initial.

Square Root Decomposition Reprezinta una dintre cele mai frecvente metode de optimizare pentru interogari folosita in lumea programarii competitive. Aceasta tehnica reduce complexitatea temporala cu un factor de $\sqrt{N}$.

1.4 Criterii de evaluare pentru solutiile propuse

Algoritmii vor fi evaluati in functie de comportamentul lor pe diferite seturi de date menite sa evidentieze avantajele si dezavantajele acestora raportate la eficienta temporala si spatiala. Pentru a evalua fiecare algoritm din punct de vedere al performatelor sale, acestia vor fi rulati pe un set de **45 de teste** structurate astfel:

- 10 teste oficiale preluate de pe Infoarena care testeaza corectitudinea rezultatelor algoritmilor
- 35 de teste personale - generate cu ajutorul unui program generator, care verifica de asemenea corectitudinea rezultatelor, dar au fost construite in special pentru a permite analizarea performantelor algoritmilor in functie de mai multi factori precum:

1. Dimensiunea array-ului (N), unde N apartine $[10^1, 10^6]$
2. Numarul de query-uri (M), unde M apartine $[10^1, 10^8]$
3. Numarul de query-uri mult mai mare decat dimensiunea array-ului
=> M >> N
4. Numarul de query-uri proportional cu dimensiunea array-ului

=> M ≈ N
5. Numarul de query-uri mult mai mic decat dimensiunea array-ului
=> M << N
6. Ordinea elementelor din array. Au fost concepute teste care contin array-uri sortate crescator dar si teste care contin array-uri sortate descrescator, astfel se va determina daca ordinea in care apar elementele influenteaza timpii de cautare.
7. Pentru o analiza mai riguroasa pe cazul general a algoritmilor s-a optat si pentru generarea unor teste random (cu N, M – aleatorii – din intervalul $[10^1, 10^6]$).

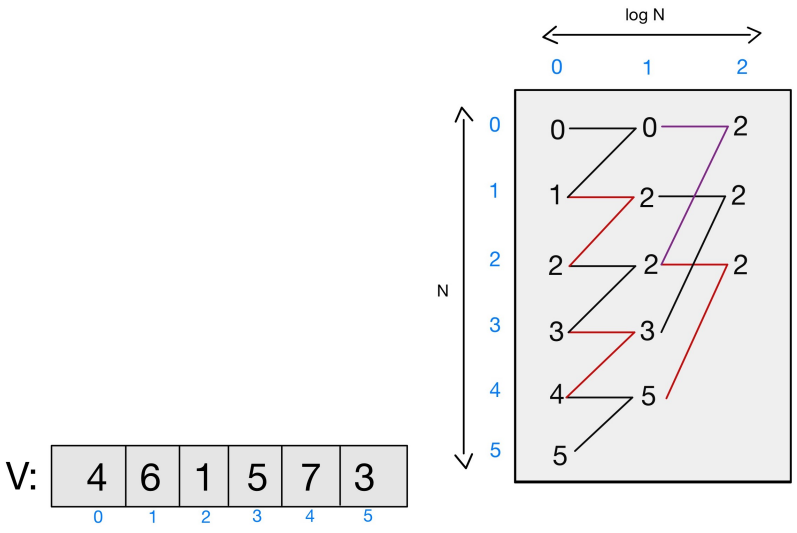
Testele sunt diversificate din punct de vedere al dimenisunilor dar si al cazurilor abordate, astfel se va putea oferi o analiza mai buna a complexitatii in functie de relatiile dintre dimensiuni, dar si din perspectiva ordinii elementelor.

2 Prezentarea solutiilor

2.1 Modul de functionare al algoritmilor alesi

Sparse Table Algorithm se bazeaza pe construirea intr-o maniera dinamica a unei structuri de date si utilizarea acesteia pentru fiecare interogare M. Pentru implementarea structurii se utilizeaza un tablou bidimensional cu N linii, si logN coloane.

Ideea din spatele tabloului consta in determinarea in avans a minimului din toate sub-tablourile de dimensiune 2^j, unde $j \in [0, logN]$.
Pentru claritate am atasat exemplul urmator:



Pe prima coloana a Sparse Tablelului se vor obtine pozitiile elementelor minime din vector pentru intervale de 1 element.
Pe a doua coloana vor fi stocate pozitiile elementelor minime din vectorul initial comparate in perechi de cate 2^1 elemente consecutive, pentru intervale de cate 2 elemente.
Pentru a construi urmatoarea coloana, ne putem folosi de coloana anterioara si vom compara elementele in grupuri de cate 2^2 elemente consecutive din vectorul initial.

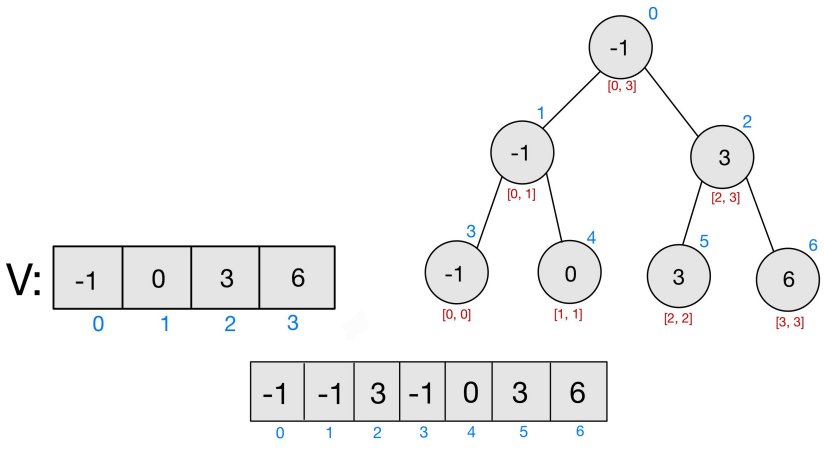
Segment Tree Algorithm utilizeaza conceptual ca structura de date un arbore binar complet in care fiecare nod retine valoarea minima corespunzatoare fiecarui subinterval.

Pentru construirea arborelui se opteaza pentru folosirea unui vector, in care accesul spre copii al unui nod se face direct la indicii $2 \cdot i + 1$ si $2 \cdot i + 2$, unde i = nodul parinte iar accesul catre parinte se poate face la indicele (j – 1) / 2, unde j = nodul copil.

Dimensiunea tabloului construit se calculeaza astfel:

- Daca numarul inital de elemente N este o putere a lui 2 => tabloul va avea $2 \cdot N$–1 elemente.
- Daca N nu este o putere a lui 2 => tabloul va avea un numar maxim de $2 \cdot M - 1$ elemente, unde M = cea mai mica putere a lui 2 mai mare decat N;

Exemplu:



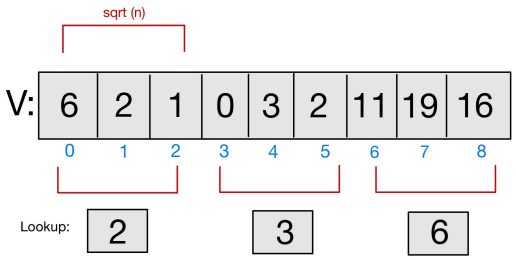
Square Root Decomposition este una dintre cele mai cunoscute metode de optimizare a interogarii in randul programarii competitive.

Algoritmul se imparte in 2 etape:
1. Preprocesarea elementelor presupune:

- Impartirea tabloului din input (V de dimensiune N) in blocuri de dimensiune $\sqrt{N}$, de unde rezulta $\sqrt{N}$ blocuri de dimensiune $\sqrt{N}$, pentru N – patrat perfect. Pe caz general, daca N – nu este patrat perfect, ultimul bloc va contine $\sqrt{N} - 1$ elemente
- Calcularea minimului corespunzator fiecarui bloc si stocarea rezultatelor intr-un vector de dimensiune $\sqrt{N}$.

2. Determinarea valorii minime din intervalul [L, R]. Pentru aceasta etapa se cauta minimul in toate blocurile de dimensiune $\sqrt{N}$ ce se incadreaza in intervalul din query. Pentru blocurile din extremitati, exista posibilitatea sa se suprapuna partial cu intervalul [L, R], in acest caz se vor scana liniar elementele corespunzatoare acestor blocuri si se va determina minimul.

Exemplu:



2.2 Analiza complexitatii algoritmilor

Metoda	Spatiu	Build	Query	Update
Sparse Table	O(Nlog N)	O(NlogN)	O(1)	O(NlogN)
Segment Tree	O(N)	O(N)	O(logN)	O(logN)
Square Root Decomposition	O($\sqrt{N}$)	O(N)	O($\sqrt{N}$))	O(1)

Sparse Table Complexitatea in etapa de pre-procesare a datelor de intrare: In implementarea structurii se utilizeaza un tablou bidimensional pentru calcularea in prealabil a raspunsurilor query-urilor si stocarea acestora. Dimensiunile tabloului sunt: N linii si logN coloane, de unde rezulta folosirea unui spatiu suplimenatar de O(NlogN). De asemenea complexitatea temporala a acestei etape este de: O(NlogN).

• Complexitatea in etapa de interogare a celor M intervale: pentru o interogare raspunsul se obtine intr-un timp constant => O(1) datorita pre-procesarii elementelor.

• Complexitatea operatiei de update: pentru realizarea unei modificari asupra elementelor vectorului din input intre query-uri, este absolut necesara refacerea etapei de pre-procesare. Tinand cont de acest aspect, timpul de update pentru metoda Sparse Table este de O(NlogN).

Segment Tree Complexitatea in etapa de pre-procesare a datelor de intrare: arborle va fi stocat tot sub forma unui array, ce contine 2 * N – 1 elemente. Astfel memoria suplimentara folosita este de $2 * 2^{(logN-1)}$, datorita relatiilor de indecsi dintre parinte si nodurile sale copii. Din punct de vedere al complexitatii, tabloul este realizat in timp liniar => O(N) si solicita un spatiu suplimantar de O(N).

• Complexitatea in etapa de interogare a celor M intervale: se parcurge recursiv arborele din nodul radacina si se verifica daca intervalul [L, R] corespunzator unui query se suprapune total, partial, sau nu se suprapune deloc cu intervalul corespunzator nodului curent din parcurgere. Astfel fiecare query necesita un timp logaritmic O(logN), in cel mai rau caz tot arborele va fi parcurs.

• Complexitatea operatiei de update: modificarea tabloului initial intre query-uri are un cost de O(log N) pentru a face update structurii de date ajutatoare – Segment Tree.

Square Root Decomposition Complexitatea in etapa de pre-procesare a datelor de intrare: se imparte tabloul din input in $\sqrt{N}$ blocuri, se calculeaza minimul corespunzator fiecarui bloc de dimensiune $\sqrt{N}$ si se stocheaza rezultatul. Toate aeste operatii au o complexitate in timp de O($\sqrt{N}$ * $\sqrt{N}$) => O(N), si solicita un spatiu suplimentar de O($\sqrt{N}$) pentru formarea blocurilor.

• Complexitatea in etapa de interogare a celor M intervale: rezolvarea unui query este implementata in 3 pasi: gasirea minimului in blocul din extremitatea stanga care se suprapune partial cu intervalul din query, gasirea minimului in blocurile care apar intre intervalul de index (L, R), gasirea minimului in ultimul bloc – extremitatea dreapta care se suprapune partial cu intervalul din query. Aceasta etapa are o complexitate temporala de O($\sqrt{N}$).

• Complexitatea operatiei de update: Daca se aplica modificari asupra tabloului initial intre query-uri, pentru a face udpate asupra blocurilor – se cauta blocul corespunzator indicelui dat si se actualizeaza valoarea la acel indice in particular in timp constant O(1).

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile alese

Avantaje si Dezavantaje Sparse Table
Aceasta metoda reduce complexitatea temporala necesara unui query la O(1), insa necesita mai multe resurse (atat de memorie cat si de timp) pentru etapa de construire. Structura de date functioneaza optim doar pentru array-uri imutabile (array-ul nu poate fi modificat intre query-uri). Daca apar schimbari in tabloul initial, este necesara o noua pre-procesare, astfel se consuma foarte mult timp. Algoritmul poate fi folosit la un potential maxim pentru cazurile cu un numar foarte mare de query-uri, insa trebuie respectata conditia de imutabilitate a tabloului.

Avantaje si Dezavantaje Segment Tree
Abordarea SegmentTree este una foarte puternica datorita flexibilitatii structurii de date folosite, astfel cu ajutorul Segment Tree se poate rezolva si verisiunea dinamica a problemei RMQ, in care elementele array-ului se pot modifica intre query-uri. Segment Tree ofera un timp rezonabil de pre-procesare O(N), iar operatiile de query si de update sunt rapide, realizandu-se in timp logaritmic O(logN).

Avantaje si Dezavantaje Square Root Decomposition
Timpul de update pentru aceasta metoda este constant O(1), astfel Square Root este cea mai buna varianta pentru rezolvarea versiunii dinamice a problemei RMQ, pentru array-uri supuse modificarilor intre query-uri.
In randul programarii competitive, un alt avantaj al acestei abordari consta in rapiditatea cu care poate fi scris algoritmul.

3 Evaluare

3.1 Descrierea modalitatii de construire a setului de tese folosite pentru validare

Pentru realizarea unui set personal de teste am construit un program generator ce genereaza teste cu input-uri atat random, dar si teste care respecta anumite proprietati, cu o dimensiune a datelor de pana la 10^8. Testele vizeaza corectitudinea rezultatelor generate de algoritmi, dar si performantele acestora.
Tabelul 1 contine detalii semnificative despre fiecare test in parte.

Construirea testelor si rolul acestora:
Testele 1 - 10: au fost preluate de pe Infoarena.
Testele 11 - 30: au fost generate total random (N, M, elementele din tabloul, query-urile sunt numere aleatore). Concepute cu scopul de a forma o persctiva generala asupra complexitatii algoritmilor in circumstante intamplatoare.
Testele 31, 32, 33: respecta proprietatea: N $>>$ M
Testele 34, 35, 36: respecta proprietatea: N $\approx$ M
Testele 37, 38, 39: respecta proprietatea: N $<<$ M
Testele 40, 41, 42: generate cu scopul de a oferi un caz favorabil pentru cautarea minimului din intreval, astfel elementele tabloului sunt sortate crescator.
Testele 43, 44, 45: elementele tabloului sunt sortate in ordine descrescatoare.

3.2 Specificatiile sistemului de calcul pe care au fost rulate testele

Sistemul de calcul pe care au fost rulate testele are urmatoarele specificatii:
Sistemul de operare: Ubuntu 20.04.3 LTS
Procesor: Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Memorie RAM: 15.4 GB

Table 1. Teste

Nr. test	N	M	Relatia N-M	Ordinea elementelor
1	10	20	N ≈ M	- random
2	100	200	N ≈ M	- random
3	5000	1000	N ≈ M	- random
4	50000	80000	N ≈ M	- random
5	50000	100000	N < M	- random
6	60000	700000	N < M	- random
7	80000	700000	N < M	- random
8	100000	800000	N ≈ M	- random
9	100000	900000	N ≈ M	- random
10	100000	1000000	N < M	- random
11	8631	4328	N ≈ M	- random
12	5018	4728	N ≈ M	- random
13	6648	6648	N ≈ M	- random
14	519	1862	N < M	- random
15	4768	7128	N ≈ M	- random
16	7494	8515	N ≈ M	- random
17	2602	892	N > M	- random
18	7352	8546	N ≈ M	- random
19	6643	2173	N > M	- random
20	9834	2402	N > M	- random
21	99905	71593	N > M	- random
22	17502	30578	N < M	- random
23	45960	3682	N > M	- random
24	54423	70038	N ≈ M	- random
25	94134	76842	N ≈ M	- random
26	22301	49093	N < m	- random
27	59768	61241	N ≈ M	- random
28	10165	43723	N < M	- random
29	50531	65240	N ≈ M	- random
30	7755	56192	N < M	- random
31	90000	900	N >> M	- random
32	100000	1000	N >> M	- random
33	100000	10	N >> M	- random
34	10000	10000	N = M	- random
35	30000	30000	N = M	- random
36	100000	100000	N = M	- random
37	1000	1000000	N << M	- random
38	1000	1000000	N << M	- random
39	1000	10000000	N << M	- random
40	90000	900	N >> M	- sortate crescator
41	10000	1000	N >> M	- sortate crescator
42	100	1000000	N << M	- sortate crescator
43	100	1000000	N << M	- sortate descrescator
44	90000	900	N >> M	- sortate descrescator
45	100000	100000	N = M	- sortate descrescator

3.3 Rezultatele evaluarii solutiilor pe setul de teste

Fig. 1. Variatia timpului de executie

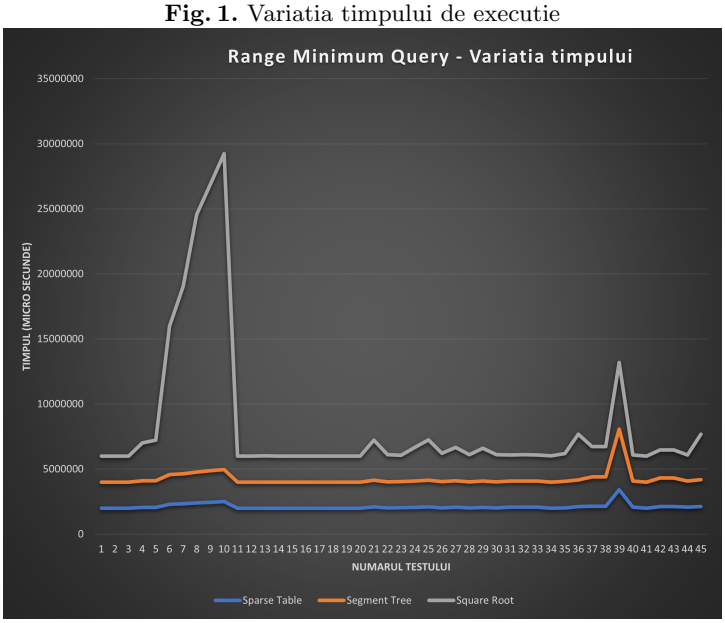


Fig. 2. Variatia timpului de executie pentru teste cu valori random

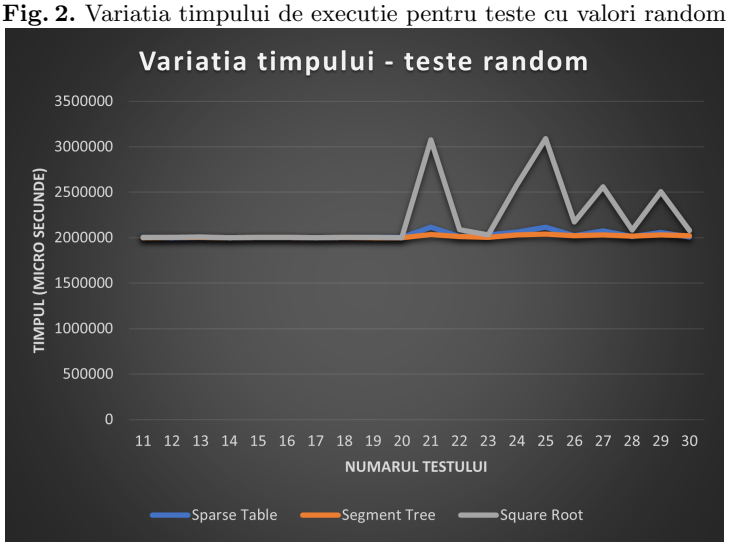


Fig. 3. Variatia timpului de executie pentru teste ce respecta conditia N >> M

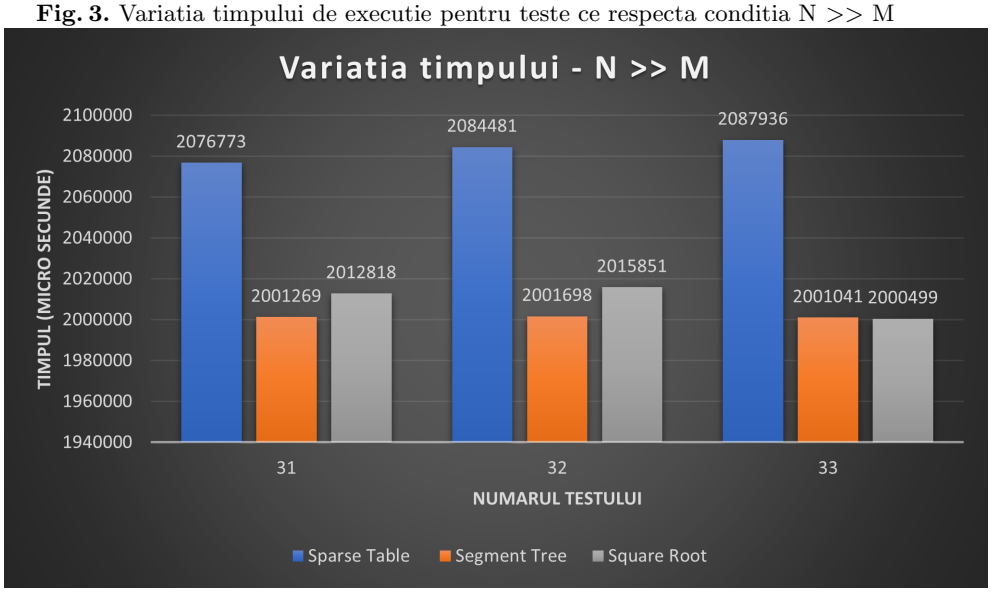


Fig. 4. Variatia timpului de executie pentru teste ce respecta conditia N ≈ M

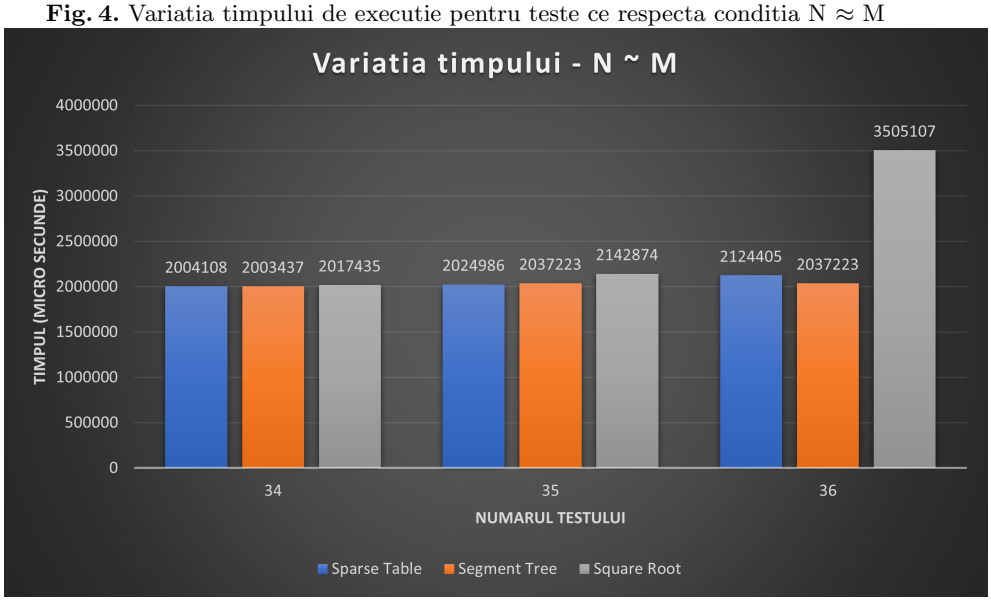


Fig. 5. Variatia timpului de executie pentru teste ce respecta conditia N << M

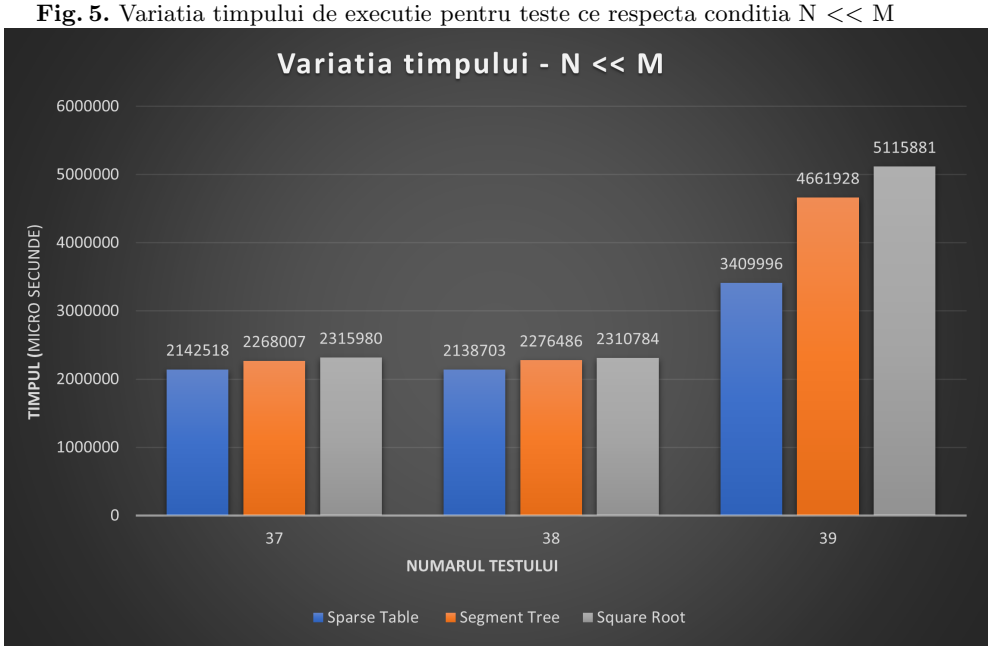


Fig. 6. Variatia timpului de executie pentru teste cu elemente sortare crescator

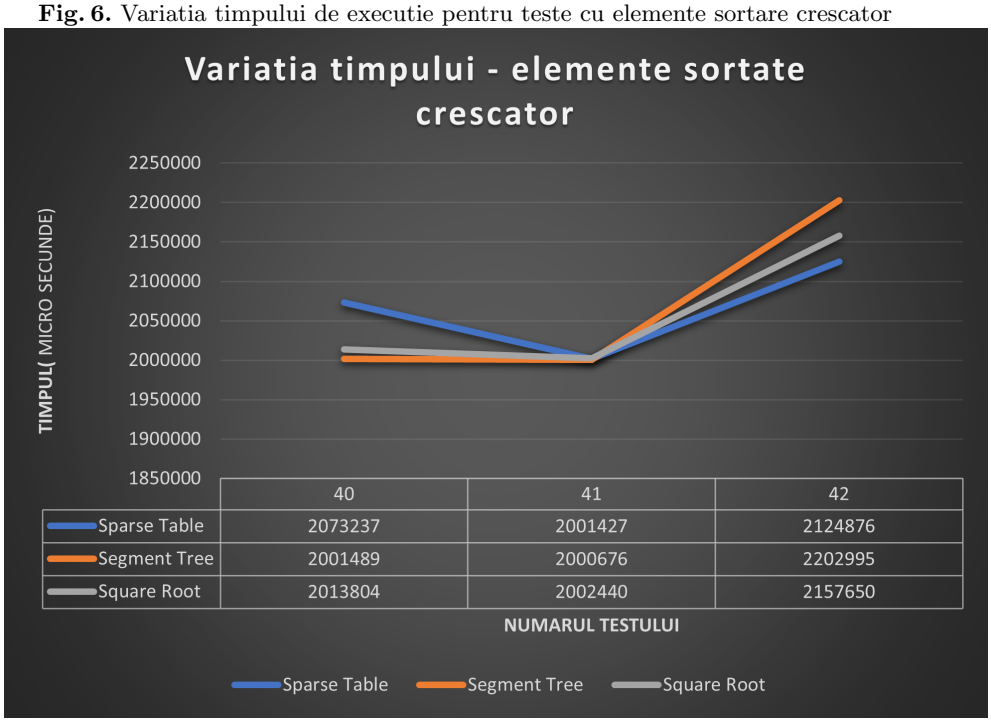
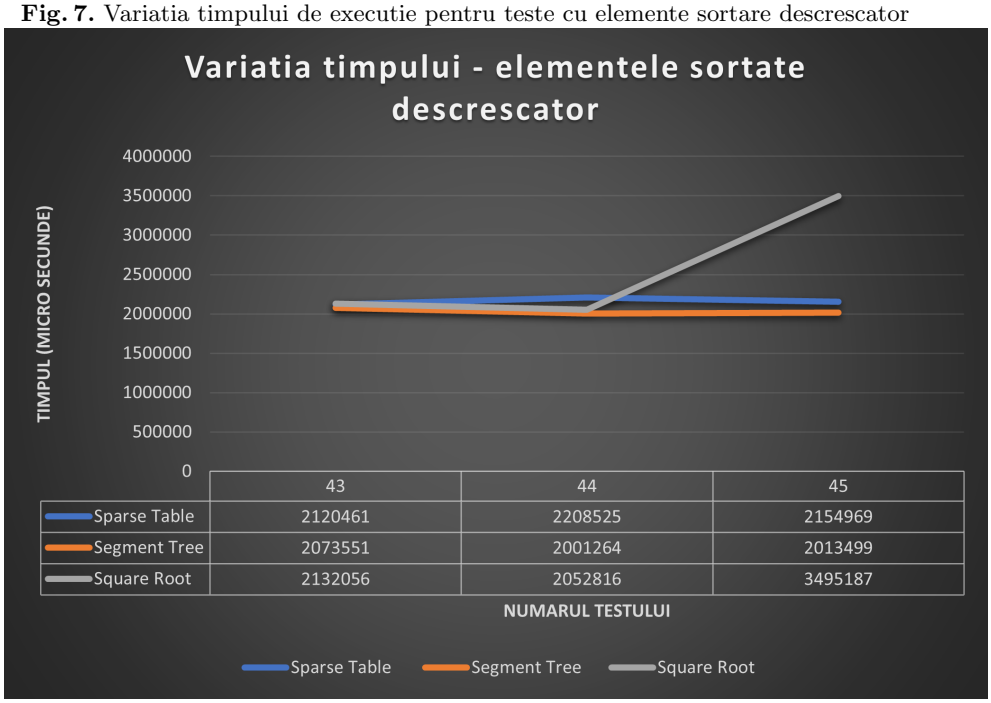


Table 2. Timpii de rulare

Nr test	Sparse Table (µs)	Segment Tree (µs)	Square Root (µs)
1	2000102	2000125	2000072
2	2000161	2000284	2000282
3	2000771	2000921	2002868
4	2068861	2039226	2896326
5	2061388	2044345	3123377
6	2294151	2292020	11383622
7	2346271	2305685	14406269
8	2411279	2361918	19793065
9	2452726	2426875	22017500
10	2502543	2462239	24275754
11	2002329	2001394	2002759
12	2001450	2001870	2002587
13	2002305	2004932	2006667
14	2000385	2001110	2000985
15	2001715	2002496	2003850
16	2002481	2004464	2003908
17	2000381	2001130	2000678
18	2002339	2003186	2002727
19	2001739	2001109	2001508
20	2001598	2001458	2000824
21	2112996	2036836	3077848
22	2010907	2011372	2087187
23	2031351	2002646	2028622
24	2062519	2030937	2583667
25	2113638	2038350	3089650
26	2020223	2020322	2173585
27	2074919	2030917	2559758
28	2012464	2017150	2083879
29	2056952	2030423	2508437
30	2010374	2019760	2080601
31	2076773	2001269	2012818
32	2084481	2001698	2015851
33	2087936	2001041	2000499
34	2004108	2003437	2017435
35	2024986	2037223	2142874
36	2124405	2048448	3505107
37	2142518	2268007	2315980
38	2138703	2276486	2310784
39	3409996	4661928	5115881
40	2073237	2001489	2013804
41	2001427	2000676	2002440
42	2124876	2202995	2157650
43	2120461	2208525	2154969
44	2073551	2001264	2013499
45	2132056	2052816	3495187

Fig. 7. Variatia timpului de executie pentru teste cu elemente sortare descrescator



3.4 Prezentarea valorilor obtinute pe teste

Graficul 1 ilsutreaza performantele algoritmilor pe toate cele 45 de teste. La prima vedere se poate observa ca Algoritmul Sparse Table are cea mai buna performanta din punct de vedere al timpului, Algoritmul Square Root are cea mai slaba performanta, iar Algoritmul Segment Tree are o performanta medie. Insa la o analiza mai detaliata exista diferente pentru timpii de executie.

Pentru testele random (graficul 2), Segment Tree este putin mai rapid decat Sparse Table. In Graficul 3, se poate observa ca performanta Sparse Table este cea mai slaba. Acest fapt este cauzat de numarul foarte mare de elemente ale tabloului, ceea ce implica un timp mai mare de executie pentru etapa de pre-procesare.

Pentru teslete 34, 35 unde N ≈ M (graficul 4), alogoritmii au performante asemanatoare iar pentru testul 36 Square Root are un timp mai mare de executie. Pentru testele cu un numar mult mai mare de query-uri decat numarul de elemente (37, 38, 39), Sparse Table este cel mai potrivit avand complexitate O(1) pentru operatia de query - acest lucru se poate observa din graficul 5.

Pentru graficul 6, s-a calculat timpul mediu de rulare, astfel pentru metoda Sparse Table este de 206651333 µs, pentru Segment Tree 206838667 µs iar pentru Square Root 205796467 µs. Ceea ce indica o performanta usor mai buna pentru Square Root. Pentru tesele ce contin elemente sortate descrescator (graficul 7) variatia timpului e mult mai buna pentru metodele Sparse Table si Segment Tree decat pentru Square Root.

4 Concluzii

In urma analizei se poate face o ierarhie a algoritmilor din punct de vedere al performantelor. Astfel, in practica as aplica algoritmii in felul urmator:

Pentru analiza unor date, precum in exemplul "RMQ in viata reala" care vizeaza un numar foarte mare de interogari a unor date care nu se pot modifica (presupun ca avem un tablou ce retine numarul de cazuri covid pe parcursul unui an care deja a trecut), prin urmare nu necesita update-uri, as utiliza algoritmul Sparse Table, deoarece etapa de pre-procesare a datelor se va face o singura data, iar complexitatea operatiei de query O(1) reprezinta un avantaj pentru un numar mare de interogari.

Dintr-o alta perspectiva daca tabolul initial ar avea foarte multe update-uri, as opta pentru solutia Segment Tree deoarece ofera un timp bun atat pentru operatia de query cat si pentru update. Aceasta abordare ar fi potrivita exemplului 2 din viata "RMQ in viata reala", deoarece fiecare moneda isi actualizeaza constant valoarea. Si Algoritmul Square Root se preteaza pe aceasta situatie, unde datele se actualizeaza constant, oferind un timp constant de update, insa pentru un numar mai mare de update-uri si query-uri as utiliza in continuare Segment Tree.

References

1. https://iq.opengenus.org/range-minimum-query-square-root-decomposition/
2. https://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/
3. https://www.geeksforgeeks.org/range-minimum-query-for-static-array//
4. http://www.topcoder.com/thrive/articles/Range
5. https://iq.opengenus.org/sparse-table/
6. $https://cp-algorithms.com/data_structures/sparse-table.html$
7. $https://cp-algorithms.com/data_structures/segment_tree.html$
8. $https://cp-algorithms.com/data_structures/sqrt_decomposition.html$

Site-urile web au fost ultima data accesate pe 16 decembrie 2021