Elementul minim dintr-un interval(RMQ)

Bogdan Alexandra-Lăcrămioara, 325CD

[1] Universitatea Politehnică din București

Abstract. În această lucrare voi studia Problema elementului minim dintr-un interval (RMQ), dar și trei dintre soluțiile acesteia, analizând atât performanța cât și corectitudinea lor. Rezolvarea acestei probleme constă în găsirea valorii minime într-un anumit interval dintr-un vector dat. De precizat este că elementele vectorului nu se schimbă. Astfel, soluțiile și structurile de date concludente pentru această problemă, pe care am ales să le studiez în lucrarea aceasta sunt Solutia banală, Algoritmul lui Mo, Arbore de segment (Segments Tree) și Sparese Table.

Keywords: Range Minimum Query, RMQ. Elementul minim dintr-un interval, Algoritmul lui Mo, Soluția banală, Segments Tree, Arbore de segment, Sparse Table.

1 Introducere

1.1 Descrierea problemei rezolvate

Primul pas în ceea ce privește orice rezolvare este întelegerea clară a enunțului.
Asadar, *Problema Elementului minim dintr-un interval* sau *Range Minimum Query Problem* afirmă următoarele: fiind dat un vector $\vec{a}$, $\vec{a} = (a_1 a_2 \dots a_n)$, cu n elemente numere întregi și m interogări de forma (x,y), reprezentând poziții din secvență, răspunsul acestor secvențe returnează valoarea elementului minim din intervalul mărginit inferior de x și superior de y, adică

$$\mathbf{argmin}_{\mathbf{x_i \le j \le y_i}} \mathbf{a_j},$$

însă fără ca elementele vectorului să se modice.

Dacă vă gândiți că Range Minimum Query Problem are aplicabilitate doar în domeniul științelor exacte vă voi prezenta câteva aplicații practice din viața cotidiană în care își are aplicabilitatea. O aplicație practică care poate fi redusă la problema propusă spre analiză în prezenta lucrare o reprezintă *Problema Strămoșului Comun a Două Noduri din Arbore* sau *Lowest Common Ancestor (LCA).* Această problemă își are

aplicabilitatea în domeniul analizei genetice, mai exact în analizarea AND-ului a două specii pentru a afla strămoșul comun din care au evoluat.
Problema *Elementului minim dintr-un interval* reprezintă de altfel și fundamentul altor operații, precum operația de determinare a *celui mai mare divizor comun pentru o subsecvență*.

1.2 Specificarea soluțiilor alese

Pentru rezolvarea *Problemei elementului minim dintr-un interval* am ales trei algoritmi, pe care îi voi prezenta mai jos:

- *Soluția banală*

  Soluția banală presupune stocarea elementelor din vector și la primirea unei interogări iterarea prin toate elementele din interval și returnarea elementului minim dintre elementele intervalului.
  Pentru cazul când M este mult mai mic decât N, această metodă se va comporta bine, deoarece nu trebuie să parcurgem toate elementele vectorului pentru a găsi elementul minim din interval. În schimb, pentru cazul când M este mult mai mare decât N, această metodă va avea o performanță slabă, deoarece trebuie să parcurgem toate elementele vectorului pentru a găsi elementul minim din interval.
  Dacă utilizatorul poate solicita schimbarea valorii unui element de la o anumită poziție în timpul interogărilor, această metodă nu se va comporta bine, deoarece trebuie să parcurgem întreg intervalul pentru a găsi elementul minim, chiar dacă valoarea unui element a fost schimbată.

- *Algoritmul lui Mo*

  În cazul acestui algoritm vom reține minimul pe bucăti de lungime $\sqrt{n}$. În momentul primirii unei interogări vom găsi minimul dintre toate bucățile de lungime $\sqrt{n}$ incluse complet în intervalul dat, apoi pur si simplu vom parcurge valorile rămase din interval, pe care nu le-am luat în considerare.
  Pentru cazul când M este mult mai mic decât N, această metodă se va comporta bine, deoarece nu trebuie să parcurgem toate elementele vectorului pentru a găsi elementul minim din interval. În schimb, pentru cazul când M este mult mai mare decât N, această metodă va avea o performanță slabă, deoarece trebuie să parcurgem toate elementele vectorului pentru a găsi elementul minim din interval.
  Dacă utilizatorul poate solicita schimbarea valorii unui element de la o anumită poziție în timpul interogărilor, această metodă nu se va comporta bine, deoarece trebuie să parcurgem întregul vector pentru a găsi elementul minim din interval, chiar dacă valoarea unui element a fost schimbată.

- *Arbore de segment (Segments Tree)*

  Această soluție presupune generarea unui arbore binar în care fiecare nod poate avea asociată o structură auxiliară, bazată pe valorile nodurilor copil. Fiecărui nod îi este alocat un interval, iar intervalele nodurilor copil se termină și încep la valoarea de mijloc a nodului părinte.
  Pentru cazul când M este mult mai mic decât N, această metodă se va comporta bine, deoarece vom parcurge doar o parte din arbore pentru a găsi elementul minim din interval. În schimb, pentru cazul când M este mult mai mare decât N, această metodă poate avea o performanță slabă, deoarece vom parcurge tot arborele pentru a găsi elementul minim din interval.
  Dacă utilizatorul poate solicita schimbarea valorii unui element de la o anumită poziție în timpul interogărilor, problema se poate modifica în sensul că trebuie să actualizăm arborele de intervale pentru a reflecta schimbarea valorii elementului. Actualizarea arborelui de intervale poate avea o complexitate de O(log N), astfel că în acest caz, problema devine mai dificilă.

- *Sparse Table*

  Aceasta este o metodă care folosește o tabelă suplimentară pentru a stoca informații despre elementele din vector, astfel încât să putem găsi elementul minim din intervalul (x,y) într-un timp mai scurt. În acest caz, vom construi o tabelă în care fiecare element stochează valoarea minimă dintr-un interval de dimensiune crescătoare. Astfel, putem găsi elementul minim dintr-un interval prin accesarea elementelor din tabelă.
  Pentru cazul când M este mult mai mic decât N, această metodă se va comporta bine, deoarece vom accesa doar câteva elemente din tabelă pentru a găsi elementul minim din interval. În schimb, pentru cazul când M este mult mai mare decât N, această metodă poate avea o performanță slabă, deoarece vom accesa toate elementele din tabelă pentru a găsi elementul minim din interval.
  Dacă utilizatorul poate solicita schimbarea valorii unui element de la o anumită poziție în timpul interogărilor, problema se poate modifica.

1.3 Evaluarea soluțiilor

Primul pas pentru evaluarea fiecărui algoritm îl reprezintă întocmirea unor seturi de teste relevante pentru verificarea atât a performanței, a eficientei în ceea ce privește memoria utilizată, cât și a eficienței în vederea rezolvării procesului de interogare.
Deoarece soluțiile sunt independente de dimensiunea fiecărui element din vector, testele vor include vectori cu n elemente numere întregi, reprezentabile pe 32 de biți.

Așadar, algoritmii vor fi analizați pe baza numărului de elemente din vectorul $\vec{a}$, și anume N și a numărului de interogări Q.
Prin urmare, testele generate vor prezenta fiecare pereche posibilă formată din N și Q, cu precizarea că $N \epsilon \{10,10^3, 10^4, 10^5, 10^6\}$, iar $Q \in \{10, 10^3, 10^4, 10^6\}$.
Așadar am generat 20 de teste posibile. Pentru fiecare test generat am făcut media aritmetică a timpului de rulare a cinci rulări.
Interogările au fost generate utilizând o distribuție uniformă.
Definim structura de date de tip RMQ ca având complexitatea în timp de ⟨p(n), q(n)⟩, unde

- p(n) reprezintă timpul maxim în care este realizată preprocesarea
- q(n) durata maximă de realizare a interogărilor.

2 Analizarea soluțiilor prezentate

1. Soluția banală

Implementarea acestei metode a fost necesară pentru validarea testelor. Aceasta este cea mai simplă variantă de rezolvare a problemei. În acest caz, vom itera prin toate numere din intervalul dat, vom găsi valoarea minima dintre ele și o vom returna.

Complexitatea metodei
Această metodă nu necesită preprocesare, deci are o complexitate a acestei faze de O(1). Fiecare interogare ar putea itera prin întregul vector, deci operația de interogare are o complexitate de O(n). De la procesare timpul este proporțional cu lungimea intervalului dat, deci are complexitate θ(n). Operația de actualizare înlocuiește doar o valoare din tabloul original, deci aceasta are o complexitate de O(1). Această metodă nu utilizează spațiu suplimentar în afară de variabile temporare, deci complexitatea în spațiu este O(1).

2. Algoritmul lui Mo

Ideea din spatele Algoritmului lui Mo este de a împărți vectorul în bucăți de lungime, să zicem b, calcularea valoarii minime pentru fiecare bucată și compararea tuturor celorlalte elemente cu valoarea precalculată.

Complexitatea algoritmului

În ceea ce privește preprocesarea, complexitatea este O(n), independentă de valoarea lui b, deoarece necesită o iterație prin întreagul vector de lungime n și stocarea valoarii minime a fiecărui element în bucata desemnată.

O interogare parcurge elementele de la început până la începutul unei bucăți sau sfârșitul interogării, apoi parcurge toate bucățile până când ajunge la bucată care conține sfârșitul intervalului și apoi parcurge elementele până când se ajunge la acest sfârșit. De aici rezultă că sunt cel mult $b + \frac{n}{b} + b$ operații, adică $2b + \frac{n}{b}$

Deoarece dorim să minimizăm ecuația $2b + \frac{n}{b}$ o vom integra, mai apoi o vom egala cu zero.

Așadar, $\int 2b + \frac{n}{b} = 0 \Rightarrow 2 - \frac{n}{b^2} = 0 \Rightarrow b^2 = \frac{n}{2}, deci\ b \sqrt{\frac{n}{2}} \approx \sqrt{n}.$

Înlocuind oricare dintre aceste substituții în ecuația inițială, obținem o complexitate de O(√n) pe interogare.

Actualizarea necesită recalcularea valorii minime a unei singure bucăți, astfel că, folosind mărimea bucății dată de ecuațiile de mai sus, avem o complexitate O(√n). Trebuie să stocăm valorile minime pentru fiecare bucată și numărul de bucăți dacă n/b, astfel că, cu b proporțional cu √n, avem o complexitate de spațiu O(√n).

3. Arbori de segment (Segment Tree)

Un segment tree este o structură de date care permite efectuarea rapidă a interogărilor de tipul "cea mai mică valoare dintr-un interval dat?" pentru un set de numere. Fiecare nod din arbore conține valoarea minimă a unui interval specific din setul dat de numere, radacina fiind valoarea minimă a întregului set. Nodul din stânga conține valoarea minimă a primei jumătăți a setului, iar nodul din dreapta conține valoarea minimă a celei de-a doua jumătăți, iar apoi continuăm să împărțim fiecare interval până rămânem cu noduri frunză care conțin elemente individuale.

Complexitatea algoritmului

Se poate demonstra că arborele conține aproximativ 2n noduri, dar din cauza indexării vom aloca spațiu pentru cel mult 4n noduri. Fiind un arbore binar echilibrat, înălțimea sa este proporțională cu log(n). Construirea arborelui constă în a merge de la nodurile frunză în sus spre rădăcină și a calcula valoarea minimă pe parcurs pentru fiecare nod. Versiunea iterativă a acestui algoritm completează de asemenea setul original cu +∞ pentru a face indexarea ușoară pentru nodurile frunză. Numărul total de noduri este proporțional cu n, astfel că avem o complexitate temporală pentru faza de prelucrare de O(n).

Interogarea implică a merge de la rădăcină în jos în algoritmul recursiv și dacă intervalul curent al nodului este inclus în intervalul pe care dorim să-l interogăm, returnează valoarea sa. Altfel, se verifică dacă intervalele din stânga sau din dreapta conțin elemente din intervalul dat și se păstrează minimul. Pentru a dovedi că fiecare interogare are o complexitate temporală de O(log(n)), vom arăta că există doar un singur nod cu proprietatea că nodurile sale stâng și drept sunt parțial incluse în intervalul dat, adică niciunul dintre ele nu returnează în timp constant dacă sunt interogate. Să presupunem că dorim să interogăm intervalul [L, R] și nodul cu proprietatea menționată mai sus acoperă intervalul [P, M] ∪ [M + 1, Q], cu intervalurile nodurilor din stânga și din dreapta împărțite la indexul M. Dacă proprietatea menționată mai sus se îndeplinește, atunci P < L ≤ M < R < Q. În timp ce ne ducem pe nodul din stânga, dacă avem intervalul $[P_0, M_0] \cup [M_0 + 1, Q_0]$ cu $Q_0 \leq$ M < R, atunci această proprietate nu va fi niciodată valabilă pentru orice copil al nodului din stânga, deoarece avem nevoie de $Q_0 >$ R. Similar, dacă ne descendem pe nodul din dreapta cu intervalul $[P_0, M_0] \cup [M_0 + 1, Q_0]$ cu L ≤ M < P_0 și avem nevoie de $P_0 <$ L. Prin urmare, vom verifica cel mult 2 log(n) noduri și astfel avem o complexitate temporală de O(log(n)) per interogare. Actualizarea unei valori implică coborârea de la nodul frunze asociat acelui element și în sus spre rădăcină, actualizarea valorii fiecărui nod, ceea ce face ca această operație să aibă o complexitate temporală de O(log(n)). În afară de copac în sine și variabila temporară, nu este necesară nicio memorie suplimentară, deci avem o complexitate spațială de O(n).

4. Sparse Table

Sparse table este o structură de date utilizată pentru a stoca și a accesa rapid informații de agregare (cum ar fi minimul) pe intervale din secvențe de lungime variabilă. În mod obișnuit, o tabelă sparsă este construită în preprocesare, iar apoi poate fi utilizată pentru a răspunde rapid la interogări de agregare pe intervale în timpul execuției.

Complexitatea algoritmului

Complexitatea în timp pentru construirea unui sparse table este O(n log n), deoarece trebuie să parcurgem fiecare element din secvență și să actualizăm valorile pentru intervale de lungimi diferite. După ce tabela este construită, complexitatea timpului pentru a răspunde la o interogare de agregare pe un interval dat este O(1), deoarece putem accesa direct valorile precalculate din tabelă. Complexitatea spațială a unui sparse table depinde de modul în care este construită tabela. Dacă se alocă o matrice completă pentru a stoca valorile, atunci complexitatea spațială va fi O(n^2). În schimb, dacă se utilizează o reprezentare mai compactă a tabelei (cum ar fi o tabelă sparsă optimizată pentru spațiu), atunci complexitatea spațială poate fi redusă la O(nlog n) sau chiar mai mică. Vom considera în cazul de față complexitatea în timp și

spațiu O(nlog n). Complexitatea interogării este O(1). Actualizarea acestei structuri de date este costisitoare și necesită actualizarea a un interval de lungime 1, a două intervale de lungime 2, a patru intervale de lungime 4 și așa mai departe până când ajungem la lungimea n. Numărul de elemente actualizate este astfel proporțional cu n, deci avem o complexitate temporală de actualizare de O(n).

3 Evaluarea

3.1 Evaluarea pe teste concrete

Așa cum este descris în prima parte, valorile numerelor din vector nu afectează performanța niciuneia dintre soluțiile testate în cazul cel mai rău. Prin urmare, definim variabila aleatoare V având o distributie uniformă peste intervalul $[-10^9, 10^9]$: V ~ Uniform($[-10^9, 10^9)$. Ceea ce contează si poate afecta performanța este lungimea fiecărei interogări. Pentru versiunea online a problemei, vom analiza performanta în cazul general, atunci când intervalele sunt răspândite cât mai mult, în timp ce adăugăm și operatii de update, numărul cărora este un procent din numarul total de interogări. Solutiile care permit executarea operatiei de update sunt Segment Tree și Algoritmul lui Mo.

Testele generate se regăsesc în folderul in. De menționat este că testele au fost rulate de pe un i5-7200U CPU @ 2.50GHz 2.70 GHz, 8GB de RAM, cu Visual Studio Code, pe 64 de biți.

Așadar am prelevat datele despre 4 algoritmi, și anume Soluția banală, Segment Tree, Algoritmul lui Mo și Sparse Table, ai căror timpi de rulare au fost comparați pe 20 de teste, după 5 rulări, în cele ce urmează.

	Algoritm					
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table		
Test Index	Timp (s)				N	M
1	0.000194	0.000145	0.000138	0.000379	10	10
2	0.000632	0.000571	0.000641	0.000554	10	1000
3	0.004312	0.004546	0.004523	0.00425	10	10000
4	0.425461	0.466227	0.454385	0.409325	10	1000000
5	0.00027	0.000474	0.000266	0.000733	1000	10
6	0.001165	0.001073	0.000917	0.004617	1000	1000
7	0.00872	0.00684	0.007092	0.004336	1000	10000
8	0.852058	0.658156	0.59685	0.428399	1000	1000000
9	0.001322	0.004205	0.001967	0.002372	10000	10
10	0.005898	0.0044	0.002331	0.003091	10000	1000
11	0.045676	0.019925	0.011367	0.007544	10000	10000
12	4.364087	0.764789	0.970911	0.442613	10000	1000000
13	0.011741	0.042597	0.012291	0.029949	100000	10
14	0.060101	0.042501	0.015945	0.027238	100000	1000
15	0.405884	0.051229	0.041888	0.030013	100000	10000
16	42.77113	0.925304	2.161822	0.542299	100000	1000000
17	0.114428	0.487793	0.131438	0.295587	1000000	10
18	0.487646	0.527405	0.143337	0.437646	1000000	1000
19	4.475126	0.525562	0.202787	0.42733	1000000	10000
20	394.3593	1.66081	6.538821	1.158364	1000000	1000000

Fig.1 Datele experimentale obținute în urma primei rulări

	Algoritm					
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table		
Test Index	Timp (s)				N	M
1	0.00019	0.000138	0.000165	0.000158	10	10
2	0.000692	0.000797	0.000594	0.000605	10	1000
3	0.00452	0.005212	0.004427	0.004076	10	10000
4	0.415899	0.489892	0.452868	0.390185	10	1000000
5	0.00036	0.000432	0.000259	0.000733	1000	10
6	0.001086	0.001248	0.000877	0.00435	1000	1000
7	0.008965	0.007254	0.006334	0.004447	1000	10000
8	0.856811	0.651574	0.619927	0.422197	1000	1000000
9	0.001313	0.00373	0.001349	0.002305	10000	10
10	0.005951	0.004557	0.002301	0.002803	10000	1000
11	0.072257	0.011492	0.010915	0.006418	10000	10000
12	4.401937	0.77471	1.044645	0.42731	10000	1000000
13	0.011795	0.051677	0.012587	0.029984	100000	10
14	0.051102	0.04333	0.015278	0.025247	100000	1000
15	0.403822	0.064469	0.035205	0.033657	100000	10000
16	43.219903	0.945341	2.240523	0.520829	100000	1000000
17	0.113942	0.500649	0.129247	0.322459	1000000	10
18	0.485851	0.510005	0.144207	0.433175	1000000	1000
19	4.412069	0.509648	0.198391	0.388026	1000000	10000
20	396.231	1.658723	6.505195	1.269681	1000000	1000000

Fig. 2 Datele experimentale obținute în urma celei de a doua rulări

	Algoritm					
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table		
Test Index	Timp (s)				N	M
1	0.000206	0.000139	0.00014	0.000165	10	10
2	0.000564	0.00061	0.000585	0.000523	10	1000
3	0.004396	0.012206	0.004537	0.004084	10	10000
4	0.428156	0.456764	0.440605	0.400998	10	1000000
5	0.000302	0.000472	0.000298	0.000762	1000	10
6	0.001119	0.001086	0.00088	0.004405	1000	1000
7	0.008701	0.007492	0.006219	0.004447	1000	10000
8	0.857464	0.658662	0.632825	0.413226	1000	1000000
9	0.001242	0.003717	0.001393	0.002494	10000	10
10	0.005918	0.004367	0.002423	0.002777	10000	1000
11	0.046249	0.012587	0.011423	0.006852	10000	10000
12	4.404402	0.794007	0.976184	0.431121	10000	1000000
13	0.011957	0.055526	0.01225	0.039012	100000	10
14	0.050573	0.042465	0.014552	0.025438	100000	1000
15	0.402163	0.054359	0.035754	0.029193	100000	10000
16	43.774112	0.934498	2.242634	0.029044	100000	1000000
17	0.124622	0.60251	0.129905	0.399389	1000000	10
18	0.479054	0.490688	0.155642	0.433256	1000000	1000
19	4.098094	0.511589	0.192159	0.455287	1000000	10000
20	399.224	1.683044	6.434736	1.202411	1000000	1000000

Fig.3 Datele experimentale obținute dupa cea de a treia rulare

	Algoritm					
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table		
Test Index	Timp (s)				N	M
1	0.000194	0.000139	0.000139	0.000161	10	10
2	0.000584	0.000636	0.000594	0.000562	10	1000
3	0.004486	0.004595	0.005294	0.00411	10	10000
4	0.422263	0.449966	0.436792	0.419955	10	1000000
5	0.000268	0.000433	0.000258	0.000749	1000	10
6	0.001088	0.00106	0.000824	0.004559	1000	1000
7	0.008745	0.007212	0.006543	0.004374	1000	10000
8	0.850883	0.638464	0.647711	0.404241	1000	1000000
9	0.00127	0.003827	0.00136	0.002401	10000	10
10	0.005906	0.004468	0.002331	0.00272	10000	1000
11	0.046496	0.01136	0.010978	0.007221	10000	10000
12	4.546808	0.775701	0.972378	0.43455	10000	1000000
13	0.011929	0.043264	0.012277	0.025478	100000	10
14	0.05315	0.05791	0.020429	0.025269	100000	1000
15	0.408423	0.051263	0.036007	0.032179	100000	10000
16	42.552123	1.043261	2.213181	0.531092	100000	1000000
17	0.119973	0.508181	0.125124	0.408723	1000000	10
18	0.509911	0.53419	0.134138	0.405427	1000000	1000
19	4.237363	0.507262	0.200023	0.458156	1000000	10000
20	394.765	1.674961	6.597644	1.226504	1000000	1000000

Fig.4 Datele experimentale obținute dupa cea de-a patra rulare

	Algoritm					
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table		
Test Index	Timp (s)				N	M
1	0.000161	0.000137	0.000139	0.000222	10	10
2	0.000604	0.000713	0.000558	0.000565	10	1000
3	0.004314	0.004769	0.005294	0.004421	10	10000
4	0.422138	0.450198	0.461009	0.395874	10	1000000
5	0.000306	0.000481	0.00028	0.000784	1000	10
6	0.001086	0.001115	0.000891	0.004348	1000	1000
7	0.008673	0.006785	0.006455	0.004386	1000	10000
8	0.8811	0.673041	0.628695	0.408365	1000	1000000
9	0.003214	0.003678	0.001434	0.002387	10000	10
10	0.005876	0.004469	0.003753	0.00277	10000	1000
11	0.047103	0.011458	0.011561	0.006484	10000	10000
12	4.417093	0.764339	0.978645	0.482735	10000	1000000
13	0.011329	0.040986	0.01233	0.025125	100000	10
14	0.05906	0.041874	0.014901	0.025815	100000	1000
15	0.425871	0.042189	0.033611	0.04311	100000	10000
16	40.806337	0.934494	2.477809	0.509144	100000	1000000
17	0.12643	0.496181	0.138043	0.399389	1000000	10
18	0.568527	0.482782	0.12917	0.437667	1000000	1000
19	4.1177	0.502459	0.19048	0.419568	1000000	10000
20	398.571847	1.540195	6.44861	1.277487	1000000	1000000

Fig. 5 Datele experimentale obținute în urma celei de-a cincea rulări

	Algoritm						
	Solutia banala	Segment Tree	Algoritmul lui Mo	Sparse Table			
Test Index	Timp (s)				Best Algorithm	N	M
1	0.000189	0.0001396	0.0001442	0.000217	Segment Tree	10	10
2	0.0006152	0.0006654	0.0005944	0.0005618	Sparse Table	10	1000
3	0.0044056	0.0062656	0.004815	0.0041882	Sparse Table	10	10000
4	0.4227834	0.4626094	0.4491318	0.4032674	Sparse Table	10	1000000
5	0.0003012	0.0004584	0.0002722	0.0007522	Algoritmul lui Mo	1000	10
6	0.0011088	0.0011164	0.0008778	0.0044558	Algoritmul lui Mo	1000	1000
7	0.0087608	0.0071166	0.0065286	0.004398	Sparse Table	1000	10000
8	0.8596632	0.6559794	0.6252016	0.4152856	Sparse Table	1000	1000000
9	0.0016722	0.0038314	0.0015006	0.0023918	Algoritmul lui Mo	10000	10
10	0.0059098	0.0044522	0.0026278	0.0028322	Algoritmul lui Mo	10000	1000
11	0.0515562	0.0133644	0.0112488	0.0069038	Sparse Table	10000	10000
12	4.4268654	0.7747092	0.9885526	0.4436658	Sparse Table	10000	1000000
13	0.0117502	0.04681	0.012347	0.0299096	Solutia banala	100000	10
14	0.0547972	0.045616	0.016221	0.0258014	Algoritmul lui Mo	100000	1000
15	0.4092326	0.0527018	0.036493	0.0336304	Sparse Table	100000	10000
16	42.624721	0.9565796	2.2671938	0.4264816	Sparse Table	100000	1000000
17	0.119879	0.5190628	0.1307514	0.3651094	Solutia banala	1000000	10
18	0.5061978	0.509014	0.1412988	0.4294342	Algoritmul lui Mo	1000000	1000
19	4.2680704	0.511304	0.196768	0.4296734	Algoritmul lui Mo	1000000	10000
20	396.6302294	1.6435466	6.5050012	1.2268894	Sparse Table	1000000	1000000

Fig. 6 Media aritmetica a celor cinci rulări pentru fiecare test, respectiv algoritm

3.2 Interpretarea rezultatelor obținute

Cazul in care M este mult mai mic decat N:

- Segment Tree este o metodă eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(log N) și actualizarea elementelor într-o complexitate O(log N).
- Algoritmul lui Mo este o metodă de asemenea eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(√N) și actualizarea elementelor într-o complexitate O(√N).
- Sparse Table este o metodă mai puțin eficientă în acest caz, deoarece construirea tabelului sparse necesită o complexitate O(N log N), dar interogările RMQ pot fi realizate într-o complexitate O(1) după construirea tabelului.

Cazul în care M este proporțional cu N:

- Algoritmul lui Mo este o metodă eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(√N) si actualizarea elementelor într-o complexitate O(√N).
- Segment Tree este o metodă de asemenea eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(log N) și actualizarea elementelor într-o complexitate O(log N).
- Sparse Table este o metodă la fel de eficientă în acest caz, deoarece construirea tabelului sparse necesită o complexitate O(N log N), dar interogarile RMQ pot fi realizate intr-o complexitate O(1) dupa construirea tabelului.

Cazul în care M este mult mai mare decat N:

- Segment Tree este o metodă eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(log N) și actualizarea elementelor într-o complexitate O(log N).
- Algoritmul lui Mo este o metodă de asemenea eficientă în acest caz, deoarece permite realizarea interogărilor RMQ într-o complexitate O(sqrt(N)) și actualizarea elementelor într-o complexitate O(sqrt(N)).
- Sparse Table este metodă cea mai eficientă, deoarece construirea tabelului sparse necesită o complexitate O(N log N), dar interogările RMQ pot fi realizate intr-o complexitate O(1) după construirea tabelului.

4 Concluzie

În timp ce unele soluții au performat mai bine decât altele în testele generate, trebuie să stabilim o ierarhie pentru a decide care algoritm se potriveste fiecărei situatii. Segment Tree, deși a performat bine pe numărul de interogările foarte mici, s-a comportat foarte prost în toate celelalte teste.

Două soluții versatile care functionează bine chiar și atunci cand aplicația necesită actualizarea vectorului sunt cele care folosesc Algoritmul lui Mo și Sparse Table. Construirea rapidă a structurilor de date necesare, împreună cu operațiile eficiente de interogare și actualizare fac aceste solutii cele mai versatile analizate până acum. Cazul ideal, așa cum este indicat în teste, este atunci când problema necesită operații de actualizare rapidă a vectorului care nu sunt neglijabile, astfel încât reconstruirea structurii de date este impracticabilă.

Soluția cea mai comună pentru această problemă este metoda Sparse Table. Deși relativ ușor de implementat, performanța sa este cea mai bună dintre toate algoritmele. Principalul dezavantaj al acestei solutii este stadiul lent de preprocesare. Precalcularea valorilor minime pentru fiecare interval necesar este costisitoare, dar se compensează prin permiterea unei operatii de interogare extrem de eficiente. Cazul ideal pentru aceasta metodă este atunci când sunt multe interogări comparativ cu mărimea vectorului, astfel incât procesul de construcție devine insignifiant, și de asemenea atunci când sunt puține sau deloc operații de actualizare necesare.

În final, putem afirma că problema elementului minim dintr-un interval(RMQ Problem) are multe soluții eficiente, care depind foarte mult de cazul de utilizare. Alegerea algoritmului potrivit înseamnă studierea atentă a problemei practice folosind atât complexitatea teoretică cât și rezultatele testelor practice înainte de a decide care este cea mai bună soluție. Nu există o metodă general valabilă care să funcționeze în toate cazurile, însă majoritatea problemelor se încadreaza intr-o anumită categorie din cele prezentate în lucrarea propusă spre analiză.

Referințe bibliografice

1. ILIE GÂRBACEA, RĂZVAN ANDONIE, 1995, Algoritmi Fundamentalio perspectiva C++, Editura Libris
2. DIMA, M., CETERCHI, R.: Efficient Range Minimum Queries using Binary Indexed Trees. OLYMPIADS IN INFORMATICS. 9, 39-44 (2015).
3. Bender, M. A., & Farach-Colton, M. (1997). The LCA problem revisited. In Mathematical Foundations of Computer Science, Springer Berlin Heidelberg.
4. Tarjan, R. E. (1983). Data structures and network algorithms. Society for Industrial and Applied Mathematics.
5. Range Minimum Query and Low Common Ancestor , https://www.topcoder.com/thrive/articles/Range%20Minimum%20Query%20and%20Lowest%20Common%20Ancestor[Accesat in 19.11.2022]
6. Range Minimum Query, https://www.infoarena.ro/problema/rmq [Accesat ultima data 19.11.2022]
7. Kogler, J., Mykhailova, M., Huang, A. and Singh, K. (2019). Range Minimum Query - Competitive Programming Algorithms. [online] Cp-algorithms.com. Available at: https://cp-algorithms.com/sequences/rmq.html [17.12.2022]
8. https://codeforces.com/blog/entry/78931 [Accesat in 16.11.2022]
9. Sparse Table, https://cp-algorithms.com/data_structures/sparse-table.html#range-minimum-queries-rmq [Accesat 18.12.2022]