Tema Analiza Algoritmilor
Etapa III

Alexandru Olteanu, grupa 322CA
alexandru.olteanu01@stud.acs.upb.ro

December 17, 2021

Range Minimum Query (RMQ)

1 Introducere

1.1 Descrierea Problemei Rezolvate

Problema de fata presupune gasirea in timp si spatiu optim a elementului minim dintr-o regiune continua de elemente ale unui sir. Solutia pare sa fie triviala la prima vedere, o simpla parcurgere in intervalul [x, y] ne gaseste elementul minim. Problema apare in momentul in care dupa o prima intrebare de acest tip apare inca una, si inca una pana cand ne aflam in dificultatea de a raspunde eficient unei multimi mai mare de astfel de intrebari. De ce? Ei bine asta presupune sa parcurgem o sectiune mare de elemente de fiecare data, rezultand astfel la un algoritm polinomial de grad 2. Din fericire, metode mult mai optime au fost gasite de-a lungul timpului, metode care cresc eficienta calculelor dar nexesita mai multa memorie, algoritmi ce vor fi analizati in continuare.

1.2 Aplicatii practice ale acestei probleme

Desi problema este interesanta si incercarea de a o rezolva reprezinta o atractie, nu ar avea atat de mult farmec daca nu am avea aplicatii practice ce ne ajuta in viata cotidiana. S-a descoperit ca aflarea valorii minime poate fi folosita in multe cazuri precum gasirea celui mai mic stramos comun, algoritm ce este folosit in realizarea sistemelor programate precum clase de obiecte. Compilatoarele ce stau la baza tuturor sistemelor in zilele noastre sunt mai eficiente astazi datorita acestei abilitati de a afla cel mai apropiat stramos comun a doua obiecte, clase sau functionalitati. Algoritmii pentru RMQ sunt de asemenea folositi in grafica 3D unde scenariile sunt impartite in cuburi sub forma unui arbore si din nou, algoritmul ajuta la gasirea celui mai aproiat cub de informatie ce contine 2 cuburi mai mici! Algoritmul RMQ este folosit in gasirea secventelor comune a doua siruri de caractere astfel ca o ultima aplicatie pe care doresc sa o mentionez este in cadrul procentului de compatibilitate intre doua cuvinte, functionalitate de baza pe care este sustinut intregul Google Search. Astfel sunt gasite aproape constant articolele ce contin cuvinte comune sau asemanatoare cu cele scrise de utilizator sau corecteaza scrierea gresita a unui cuvant!

1.3 Specificarea Solutiilor alese

Pentru a rezolva eficient aceasta problema am ales trei algoritmi diferiti ce se comporta unic in functie de dimensiunile input-ului si posibilitatile utilizatorului de interactiune cu sirul de elemente si cu numarul de intrebari puse in total. Desi nu sunt neaparat niste algoritmi ci mai mult metode de rezolvare a problemei, cei 3 algoritmi alesi sunt Segment Tree, Sparse Table si Square Root Decomposition. Segment Tree se bazeaza pe construirea unui arbore binar in care fiecare nod contine minumul corespondent unui interval bine determinat, astfel vom putea afla date din acest arbore in mod eficient. De asemenea, Segment Tree este metoda ce suporta si modificarea elementelor din sirul initial datorita acestei structuri arborescente in care frunzele sunt chiar elementele sirului. Sparse Table este a doua metoda despre care vom discuta. Spre deosebire de Segment Tree, Sparse Table

va fi reprezentata in memorie ca fiind o matrice unde st[i][j] mentine valoarea minima pe intervalul [i, i + 2^j - 1]. Construirea acestei matrice este finala, asta insemnand ca nu putem modifica eficient elemente in sirul initial pe masura ce adresam intrebarile. Cu toate acestea, Sparse Table ne permite sa raspundem intrebarilor in O(1). Ultimul algoritm ales, Square Root Decomposition se bazeaza pe impartirea sirului in blocuri de dimensiune $\lceil\sqrt{n}\rceil$. Preprocesarea are complexitate O(n) si pentru a raspunde fiecarei intrebari vom avea complexitatea O($\sqrt{n}$). Vom vedea mai tarziu ca aceasta metoda necesita pe alocuri si calcularea triviala anumitor parti din intervalul dorit!

1.4 Criterii de evaluare pentru solutii propuse

Pentru evaluarea algoritmilor de mai sus mi-am propus sa generez teste de dimensiuni diferite cu ajutorul unui script de generare. Vreau sa generez atat teste in care dimensiunea sirului (N) si numarul de interogari (M) sunt de dimensiune mica sau ambele sunt foarte mari, etc! De asemenea voi genera si teste care cuprind aceeasi interogare pe intervalul [1, n] pentru a testa in cel mai nefavorabil scenariu eficienta celor trei algoritmi. Pe langa aceste teste generate pentru a putea verifica corectitudinea solutiilor voi realiza un script ce compara doua solutii pentru a evidentia posibile diferente in rezultate. Astfel voi putea construi o solutie banala de complexitate O(n) per interogare cu ajutorul careia sa verific corectitudinea celor 3 algoritmi pe teste de dimensiune relativ mica.

1.5 Referinte legate de RMQ

Range Query (Data Structure)

Fischer, Johannes; Heun, Volker (2007). A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array

Fischer, J. and V. Heun (2006). "Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE"

2 Prezentarea Solutiilor

2.1 Descrierea modului in care functioneaza algoritmii alesi

2.1.1 Algoritmul Segment Tree

Acest algoritm se bazeaza pe construirea in memorie a unui arbore binar reprezentat cu ajutorul unui vector. Pentru construirea acestui arbore binar este nevoie sa adaugam valorile din sirul dat in nodurile frunza ale acestui arbore binar. Urmatorul pas este cel de construire al intregului arbore pornind de la aceste noduri frunza, pentru aceasta fiind folosita metoda de build din cadrul declararii acestui algoritm. Intrebarea este, desigur, ce reprezinta aceste noduri din graf? Conceptul din spatele algoritmului Segment Tree este unul simplu, fiecare nod corespunde valorri functiei de minim pe un anumit interval. Mai exact, copiii unui nod X corespund primii jumatati respectiv celei de a doua din intervalul retinut de parintele acestora. In acest mod, calculul minimului pe un interval reprezinta de fapt impartirea acestuia in intervale cu dimensiune cat mai mare reprezentate in totalitate de noduri din arbore. Se aplica apoi functia de minim intre toate aceste noduri, rezultatul fiind cel final, minimul din intervalul dorit!

2.1.2 Algoritmul Sparse Table

Acest algoritm dispune de o abordare cu ajutorul unei matrice ce retine valori ale functiei de minim pentru fiecare pozitie, pentru fiecare pozitie retinand minimul pentru toate intervalele de lungime puteri ale lui 2 pornind din respectiva pozitie. Astfel, valoarea lui matrix[i][j] reprezinta rezultatul functiei de minim pe intervalul [i, i + 2^j - 1]. In continuare vom intelege cum este construita aceasta matrice in mod eficient. Pentru inceput vor fi puse pentru toate pozitiile valoarea lui mat[i][0], valori ce reprezinta sirul initial in sine. Apoi, se va itera prin puterile lui 2 consecutiv plecand de la 1. Astfel, cunoscand valorile din matrice pentru puterea anterioara, valoarea curenta se poate calcula dupa urmatoarea formula: mat[i][j] = min(mat[i][j - 1], mat[i + 2^{j-1}][j - 1]). Astfel, matricea se

construieste treptat de la valorile pentru puterile mici la cele mari. Odata construita matricea trebuie obtinut rezultatul pentru un interval oarecare, interval ce nu este o putere a lui 2 in majoritatea cazurilor, cum procedam? Pentru acest calcul avem doua optiuni, cea de a doua fiind mai putin evidenta dar mai eficienta. Prima optiune este de a imparti intervalul dat intr-o reuniune de intervale ce sunt puteri ale lui 2 de lungime maxima. In acest mod, putem calcula minimul dintre toate aceste intervale componente. Cea de a doua optiune este oferita de urmatoarea observatie: Un interval nu poate contine doua alte intervale de putere maxima a lui 2 fara ca acestea sa se interesecteze (in caz contrar puterea lui 2 aleasa nu ar mai fi maxima). Astfel, este de ajuns sa calculam minimul intre cel mai mare interval cu dimensiune putere a lui 2 ce pleaca din capatul stang al intervalului si cel mai mare interval de aceeasi lungime ce se termina in capatul drept al intervalului. Astfel, intreg intervalul este acoperit si minimul este obtinut. De mentionat este ca aceasta metoda nu merge mereu indiferent de functie (Spre exemplu, daca vrem sa calculam suma pe interval, trebuie scazuta apoi zona de intersectie)

2.1.3 Algoritmul Square Root Decomposition

Acest algoritm are unele aspecte comune cu algoritmul Sparse Table deorece acesta imparte sirul dat in lungimi aproximativ egale de lungime $\sqrt{n}$ pe cand cel de Sparse Table impartea in puteri ale lui 2. Cu toate acestea, cele doua difera, SQD salvand aceste informatii intr-un vector. Ideea ce sta la baza solutiei este de a memora valoarea functiei pentru aceste blocuri de lungime egala si apoi adaugarea a cator mai multe astfel de blocuri in intervalul ce trebuie calculat. Astfel, aceasta solutie asigura ca nu va ramane o portiune mai mare sau egala cu $\sqrt{n}$ ce nu a fost inclusa in calcule. Pentru aceasta portiune ramasa va trebui sa o parcurgem manual, element cu element si sa calculam minimul final!

2.2 Analiza complexitatii solutiilor

Pentru calculul complexitatilor celor 3 algoritmi vom analiza pas cu pas operatiile efectuate de acestia pentru obtinerea rezultatelor.

2.2.1 Algoritmul Segment Tree

Pentru citirea si initializarea sirului din care nodurile frunza ale arborelui isi vor prelua datele complexitatea este O(N).

Pentru realizarea constructiei arborelui final trebuie accesat fiecare nod din acesta (fiind aproximativ 4N noduri), complexitatea acestei operatii fiind O(N).

Pentru a raspunde unei singure interogari din cele M, numarul de operatii necesar este in general $\log_2 N$ deoarece vom gasi mereu nodurile ce corespund celor mai mari intervale cuprinse in intervalul dorit (Acest proces continand aproximativ $\log_2 N$ intervale). Astfel, complexitatea pentru rezolvarea interogarilor este O(M * $\log_2 N$).

Complexitatea algoritmului este altfel O(N + M * $\log_2 N$) pentru N si M de dimensiuni diferite (unul valoare mica si altul mare) dar se poate considera O(N * $\log_2 N$) pentru N si M aproximativ egale.

Din punct de vedere al memoriei, acest algoritm utilizeaza doi vectori pentru stocarea informatiilor, unul de dimensiune N si altul de dimnensiune 4 * N. Astfel, memoria totala folosita este de aproximnativ 5 * N variabile de tip intreg, respectiv 20 * N bytes de memorie.

2.2.2 Algoritmul Sparse Table

Pentru citirea si initializarea sirului din care se primeste valori pentru matrix[i][0] complexitatea este O(N).

Pentru realizarea constructiei matricei finale complexitatea este O(N * $log_2 N$) deoarece pentru fiecare putere trebuie parcurse toate pozitiile din sir.

Pentru a raspunde unei singure interogari din cele M, complexiatea este constanta, O(1), deoarece trebuie calculat doar minimul dintre doua intervale preprocesate. Astfel, complexitatea pentru rezolvarea interogatiilor este O(M).

Complexitatea finala a algoritmului este O(M + N * $log_2 N$). Din punct de vedere al memoriei, acest algoritm utilizeaza 2 vectori de dimensiune N si o matrice de dimensiuni N * $log_2 N$, astfel memoria totala folosita este de 8N + 4N * $log_2 N$ bytes de memorie.

2.2.3 Algoritmul Square Root Decomposition

Pentru citirea si initializarea sirului din care se primeste valori complexitatea este O(N)

Pentru realizarea constructiei vectorului ce retine minimul pe intervale de lungime $\sqrt{n}$ complexitatea este O(N) deoarece trebuie trecut prin fiecare pozitie din sir si updatat la fiecare pas blocul din care pozitia respectiva face parte.

Pentru a raspunde unei singure interogari din cele M, complexitatea este O($\sqrt{N}$) in cel mai nefavorabil caz. Astfel, complexitatea pentru rezolvarea interogatiilor este O(M * $\sqrt{N}$).

Complexitatea finala a algoritmului este astfel O(N + M * $\sqrt{N}$) pentru N si M diferite dar se poate considera O(N * $\sqrt{N}$) pentru N si M aproximativ egale.

Din punct de vedere al memoriei, acest algoritm necesita un vector de dimensiune N si altul de diensiune $\sqrt{N}$, astfel memoria totala folosita este de 4N + 4$\sqrt{N}$ bytes de memorie.

2.2.4 Analiza complexitatilor pentru diferite cazuri de diferenta intre N si M ai celor trei algoritmi

Precedent am analizat complexitatile celor trei algoritmi si am obtinut urmatoarele rezultate:

- Complexitate Segment Tree : O(N + M * $log_2 N$)
- Complexitate Sparse Table : O(M + N * $log_2 N$)
- Complexitate Square Root Decomposition : O(N + M * $\sqrt{N}$)

Astfel, putem observa diferente de optimizare intre cele trei solutii in cazuri speciale. Atunci cand M este mult mai mic decat N, complexitatea pentru Segment Tree si Square Root Decomposition devine aproximativ O(N), pe cand cea pentru Sparse Table este O(N * $log_2 N$). In acest caz Sparse Table este defavorizat si va avea o performanta mai slaba decat ceilalti doi. In cazul in care N este mult mai mic decat M, complexitatea pentru toti cei trei algoritmi devine O(M), caz in care acestia ajung la fel de optimi in majoritatea cazurilor de acest tip. Ultimul caz este cel in care N este proportional cu M, caz in care cei doi algoritmi, Segment Tree si Sparse Table pastreaza o complexitate comuna si optima, pe cand algoritmul Square Root Decomposition aduce o performanta cu complexitatea sa O(N * $\sqrt{N}$)

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

2.3.1 Algoritmul Segment Tree

Un avantaj major pe care il prezinta acest algoritm este capabilitatea de a face update-uri in sir in timpul interogarilor cu aceeasi complexitate ca rezolvarea minimului pe un interval. Din cei trei algoritmi acesta este singurul care suporta acest tip de modificari, reprezentand astfel un avantaj unic al acestui algoritm. Un dezavantaj pe care il are acest algoritm este gradul mai ridicat de dificultate in ceea ce priveste implementarea in comparatie cu ceilalti doi algoritmi. Per total, algoritmul Segment Tree ofera o complexitate si utilizare de memorie buna insa nu este de preferat datorita implementarii costisitoare ce are sanse mai mari sa aiba erori.

2.3.2 Algoritmul Sparse Table

Principalul avantaj pe care il ofera acest algoritm este procesarea interogarilor in O(1). Astfel, Sparse Table este de obicei de preferat datorita calitatii de a lucra cu o cantitate mare de interogari foarte eficient. Pe de alta parte, dezavantajul il reprezinta insusi incapabilitatea de a trata update-uri in timpul interogarilor. Astfel, el poate fi folosit doar in cazurile cand sirul ramane neschimbat.

2.3.3 Algoritmul Square Root Decomposition

Marele avantaj de care dispune acest algoritm este memoria considerabil de mica pe care o foloseste. In principal este un algoritm preferat in cazul testelor mici cand memoria de care dispunem este limitata. Astfel, pentru o cantitate mica de memorie putem spori complexitatea de la solutia banala cu complexitate O(N) per interogare la O($\sqrt{N}$). Dezavantajele insa sunt multe, acest algoritm nu suporta update-uri iar complexitatea lui este cu mult mai neeficienta decat a celorlalti algoritmi. Tragem de aici concluzia ca nu este deloc un algoritm de preferat cand intampinam o gama larga de date.

3 Evaluare

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

Pentru a genera cele 20 de teste folosite in analiza corectitudinii celor 3 algoritmi am folosit un script de generare random pentru numere atat pozitive cat si negative. Pentru a obtine teste cu diferite dimensiuni am modificat pe parcurs intervalul de alegere al numerelor random. Astfel am creat teste cu N si M de dimensiuni mici, medii si foarte mari. Cateva teste au fost generate cu N si M primind dimensiunea maxima iar un test a fost generat asa cum am amintit anterior cu toate interogarile pe intervalul [1, N] pentru a analiza cum se comporta cei 3 algoritmi pe cazul cel mai nefavorabil.

3.2 Mentionati specificatiile sistemului de calcul pe care ati rulat testele(procesor, memorie disponibila)

Testele au fost rulate pe laptop-ul personal cu urmatoarele specificatii:

- Procesor : AMD Ryzen 7 4800H, 16 cores, 4.2 Ghz
- Placa video : Radeon RX 5500M, 4 Gb
- Memorie totala RAM : 16 Gb
- Memorie RAM folosita de sistem : 5.5 Gb
- Memorie disponibila : 10.5 Gb
- Sistem de operare : Windows 10 Pro

3.3 Ilustrarea, folosind grafice / tabele, a rezultatelor evaluarii solutiilor pe setul de teste

Prezentarea rezultatelor celor trei algoritmi pe testele generate in timp de secunde in functie de dimensiunea sirului si a interogarilor:

Nr.Test	N	M	Algoritmul Segment Tree	Algoritmul Sparse Table	Algoritmul Square Root Decomposition
1	9	15	0.003238 s	0.002901 s	0.003357 s
2	10	15	0.002063 s	0.002236 s	0.00235 s
3	7	11	0.003141 s	0.002979 s	0.003865 s
4	6	11	0.002009 s	0.002361 s	0.002302 s
5	9	9	0.001733 s	0.001503 s	0.001618 s
6	576	4299	0.004036 s	0.003541 s	0.004179 s
7	6407	576	0.003246 s	0.004129 s	0.00377 s
8	5887	4702	0.006637 s	0.005973 s	0.008191 s
9	5646	8787	0.007892 s	0.007017 s	0.011258 s
10	6744	7321	0.007173 s	0.007055 s	0.010701 s
11	433048	606163	0.595819 s	0.648449 s	3.69393 s
12	303918	434652	0.433322 s	0.392864 s	2.25046 s
13	133401	260135	0.222942 s	0.230452 s	0.923741 s
14	91819	531646	0.412513 s	0.382036 s	1.54816 s
15	226670	547179	0.413191 s	0.49891 s	2.47356 s
16	1000000	1000000	1.10483 s	1.24087 s	9.35986 s
17	1000000	1000000	1.11867 s	1.24554 s	9.35507 s
18	1000000	1000000	1.09809 s	1.25761 s	9.39647 s
19	1000000	1000000	1.09911 s	1.24779 s	9.38876 s
20	1000000	1000000	0.511421 s	0.863486 s	7.08597 s

3.4 Prezentarea, succinta, a valorilor obtinute pe teste. Daca apar valori neasteptate, incercati sa oferiti o explicatie.

In urma analizei comportamentului celor trei algoritmi pe testele generate putem afirma, intr-adevar, ca algoritmul Segment Tree si algoritmul Sparse Table au complexitate similara. Ambii au rulat tot setul de teste fara sa depaseasca 1.26 secunde, o limita de timp potrivita pentru seturi largi de date precum ultimele 5 teste. Pe de alta parte, algoritmul Square Root Decomposition s-a comportat cum era de asteptat, pentru primele 10 teste unde valorile au fost relativ mici acesta a obtinut un timp foarte bun, aproximativ identic cu ceilalti doi. Diferentele majore au inceput cu testul 11 in continuare unde acesta a ajuns sa ruleze pana la aproximativ 9.4 secunde pe un singur test, de 10 ori mai incet decat Segment Tree si Sparse Table. Avand in vedere aceste date vom incerca sa infaptuim un clasament in functie de numarul de teste unde fiecare a rulat in cel mai scurt timp, astfel:

- Algoritmul Segment Tree : 11
- Algoritmul Sparse Table : 9
- Algoritmul Square Root Decomposition : 0

Putem evidentia astfel concluzia ca primii doi algoritmi sunt cu siguranta considerati la fel de buni, cu precizarea ca pe testele de dimensiuni mai mari Segment Tree obtine timpi putin mai buni in comparatie cu Sparse Table. Un alt punct asupra caruia vreau sa fac referire este legat de ultimul test, test care in teorie trebuia sa obtina cel mai nefavorabil timp pentru cei 3 algoritmi, cu toate acestea timpul pentru acesta este considerabil mai mic decat cel pentru testele 15 - 19. Explicatia ar fi urmatoarea: Pentru Segment Tree nu mai este nevoie de $log_2 N$ operatii cu aproximare pentru fiecare interogare deoarece toate acestea se dau pe intreg sirul, interval retinut chiar de primul nod. Astfel, cautarea se incheie mereu in O(1), complexitatea fiind considerabil mai mica. Pentru Sparse Table nu am intocmai o explicatie clara, probabil unui program ii este mai usor sa afiseze acelasi numar de mai multe ori decat diferite numere. In cadrul Algoritmului Square Root Decomposition se observa ca in cadrul unei interogari [1, N], aceasta reprezinta intocmai modul de parcurgere initial cand au fost construite blocurile. Astfel numarul de bucati neacoperite este diminuat si astfel se obtine o complexitate mai buna.

4 Concluzii

4.1 Precizati, in urma analizei facute, cum ati aborda problema in practica; in ce situatii ati apela pentru una din solutiile alese.

In urma analizei realizate am ajuns la concluzia ca fiecare algoritm are scopul sau unic. Algoritmul Segment Tree are o implementare mai dificila insa performantele sale sunt foarte bune, acesta suporta modificari ale elementelor pe parcursul interogarilor ceea ce reprezinta un motiv principal pentru care as folosi acest algoritm in practica. Astfel, in cadrul unor date largi si a modificarilor elementelor, Segment Tree este o optiune de dorit.

Algoritmul Sparse Table poate fi cu siguranta folosit in mai multe cazuri in care nu este necesara modificarea elementelor. Implementarea rapida si complexitatea foarte buna reprezinta o optiune de preferat pentru testing rapid si rezolvarea problemelor comune ce implica calcule pe intervale.

Algoritmul Square Root Decomposition este cu siguranta un algoritm ce nu trebuie neglijat. In detrimentul faptului ca nu a fost cel mai optim algoritm pe niciunul dintre teste, se poate trage concluzia ca a avut un timp de rulare foarte bun pana la N, M aproximativ $< 10\ 000$ si un timp acceptabil pentru testing pentru valori N, M pana la 500 000. Marele avantaj pe care il ofera acest algoritm este cel al memoriei cum am mentionat mai sus. De asemenea, este cel mai usor de implementat dintre cei trei, astfel fiind o alegere foarte buna in cadrul unor teste relativ de dimensiune mica ce trebuie testate intr-o scurta perioada de timp sau cu memorie limitata.

5 Bibliografie

https://cp-algorithms.com/data$_s tructures/sqrt_d ecomposition.html$
https://cp-algorithms.com/data$_s tructures/sparse - table.html$
https://cp-algorithms.com/data$_s tructures/segment_t ree.html$
https://codeforces.com/blog/entry/83248