ARBORI indexați BINAR

Mihai Scorțaru

Problema determinării sumei elementelor unei subsecvențe a unui șir ale cărui valori se modifică în timp real apare destul de des în diferite aplicații. Ea a apărut, sub diverse forme și la anumite concursuri de programare. În cadrul acestui articol vom prezenta o structură de date care poate fi folosită pentru rezolvarea eficientă a acestei probleme.

Introducere

Prin ***subsecvență*** înțelegem un subșir ale cărui elemente se află pe poziții consecutive în șirul inițial. De exemplu, (2, 3, 4) este o subsecvență a șirului (1, 2, 3, 4, 5) formată din al doilea, al treilea și al patrulea element al șirului, în timp ce (1, 2, 4) nu este o subsecvență deoarece nu este formată din elemente aflate pe poziții consecutive.

În cadrul acestui articol vom identifica o subsecvență prin extremitățile sale (poziția cea mai din stânga și poziția cea mai din dreapta din șir). O subsecvență care începe în poziția *a* și se termină în poziția *b* va fi notată prin <*a*, *b*>. Pentru șirul (1, 2, 3, 4, 5) subsecvența (2, 3, 4) va fi notată prin <2, 4> dacă numerotarea elementelor începe cu 1 sau prin <1, 3> dacă numerotarea începe de la 0.

Deseori, avem nevoie de informații referitoare la subsecvențele unui șir cum ar fi suma elementelor, produsul lor, valoarea minimă, valoarea maximă etc. La prima vedere, rezolvarea acestei probleme pare destul de simplă (de exemplu, pentru sumă se parcurg elementele subsecvenței și se adună). Totuși, acest algoritm este ineficient datorită faptului că necesită parcurgerea întregii subsecvențe. Vom arăta că există algoritmi pentru care o astfel de parcurgere nu este necesară.

Totuși, dacă am efectua o singură dată sau de un număr limitat de ori o astfel de parcurgere, algoritmul ar putea părea performant, viteza de execuție fiind relativ mare. Problema se complică dacă elementele șirului se modifică în timp real.

Modificări în timp real

Să presupunem că există două tipuri de operații care pot fi efectuate asupra unui șir. Primul tip constă în modificarea valorii unui element, în timp ce al doilea reprezintă interogări (cereri de informații) referitoare la anumite subsecvențe ale șirului. De obicei, cele două tipuri de operații sunt "amestecate" în sensul că nu vor fi doar modificări urmate din interogări, ci putem avea o modificare, urmată de două interogări, urmate de cinci modificări, urmate de alte două interogări etc.

Așadar, putem spune că elementele șirului se modifică în timp real și interogările se referă la starea curentă a șirului (cea din momentul cererii de informații).

Enunțul problemei

În cele ce urmează vom prezenta un enunț al problemei, particularizat pentru cazul în care interogările se referă la suma elementelor subsecvențelor.

Se consideră un șir de numere întregi care are toate elementele nule. O modificare a unui element constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei subsecvențe a șirului.

Problema poate fi enunțată în multe alte forme. Una dintre ele ar putea fi următoarea:

Un furnizor lucrează cu N distribuitori; în fiecare moment distribuitorii pot efectua plăți sau pot cumpăra produse. De asemenea, în fiecare moment furnizorul poate cere informații referitoare la suma totală a datoriilor pe care le au magazinele cu numere de ordine cuprinse între două valori date.

Evident, există multe alte enunțuri echivalente. De asemenea, pot apărea enunțuri în care operația de însumare ar putea fi înlocuită cu altele. De exemplu, furnizorul ar putea dori să cunoască datoria maximă a unui magazin care are numărul de ordine cuprins între două valori date.

Exemplu

Vom exemplifica acum evoluția în timp real a unui șir format din cinci elemente, prezentând valorile șirului după fiecare modificare și rezultatul fiecărei interogări.

Operația **Inițializare** constă în setarea la 0 a valorilor tuturor elementelor. Operația **Adună**(*i*, *x*) constă în adunarea valorii *x* la valoarea curentă a celui de-al *i*-lea element.

Operația **Sumă**(*a*, *b*) furnizează suma elementelor subsecvenței <*a*, *b*>.

Operație	*Șir/Rezultat*
Inițializare	0 0 0 0 0
Adună(1, 3)	3 0 0 0 0
Adună(4, 5)	3 0 0 5 0
Sumă(3, 3)	0
Sumă(4, 4)	5
Adună(4, 2)	3 0 0 7 0
Adună(2, 3)	3 3 0 7 0
Sumă(2, 5)	10
Sumă(2, 4)	10
Adună(5, 2)	3 3 0 7 2
Sumă(2, 4)	10
Sumă(2, 5)	12
Adună(3, 1)	3 3 1 7 2
Adună(4, -2)	3 3 1 5 2
Sumă(2, 5)	11
Adună(1, -3)	0 3 1 5 2
Adună(5, 5)	0 3 1 5 7
Sumă(1, 2)	3
Sumă(3, 4)	6
Adună(2, -1)	0 2 1 5 7
Adună(3, 3)	0 2 4 5 7
Sumă(5, 5)	7
Sumă(1, 2)	2
Sumă(1, 5)	18

Cazul unidimensional

Din modul în care a fost enunțată problema, rezultă că operațiile sunt efectuate asupra unui tablou unidimensional; așadar, acesta este cazul unidimensional al problemei. Vom prezenta în continuare trei algoritmi care pot fi folosiți pentru rezolvarea problemei și apoi le vom studia performanțele.

Algoritmul naiv

Cea mai intuitivă metodă de rezolvare constă în păstrarea unui vector cu valorile șirului, modificarea lor atunci când este necesar și calcularea sumelor în momentul în care apar interogări.

Pentru a prezenta algoritmul vom considera că o modificare este codificată prin valoarea 1, iar o interogare prin valoarea 2. Ar fi necesară o a treia operație (codificată prin 3) care ar indica faptul că nu mai există modificări sau interogări, deci execuția poate fi încheiată.

Versiunea în pseudocod este prezentată în continuare:

```
//inițializări
scrie Introduceți numărul de elemente:
citește N      //dimensiunea șirului
pentru i ← 1, N execută
   a_i ← 0      //valorile inițiale sunt nule
sfârșit pentru
scrie Introduceți codul operației:
citește cod
cât timp cod ≠ 3 execută
   dacă cod = 1  //modificare
      atunci
         scrie Introduceți indicele elementului care va fi
               modificat:
         citește ind
         scrie Introduceți valoarea care va fi adunată
               (valoare negativă pentru scăderi):
         citește val
         a_ind ← a_ind + val
      altfel  //interogare
         scrie Introduceți extremitățile subsecvenței:
         citește st, dr
         suma ← 0
         pentru i ← st, dr execută
            suma ← suma + a_i
         sfârșit pentru
         scrie Suma elementelor secvenței este, suma
   sfârșit dacă
   scrie Introduceți codul operației:
   citește cod
sfârșit cât timp
```

Se observă că modificările se efectuează în timp constant deoarece implică doar accesarea unui element al vectorului și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor unei subsecvențe este necesară parcurgerea tuturor elementelor subsecvenței, această operație are ordinul de complexitate $O(l)$, unde l este lungimea subsecvenței.

Trebuie observat faptul că algoritmul este același dacă operația de însumare este înlocuită cu o alta (calcularea produsului, stabilirea minimului etc.).

Vector de sume

Prima încercare de optimizare constă în găsirea unui algoritm mai rapid pentru calcularea sumei elementelor unei subsecvențe. O posibilitate relativ simplă este păstrarea unui vector b a cărui elemente b_i reprezintă suma primelor elemente ale șirului a. Pentru a găsi suma elementelor unei subsecvențe <*st*, *dr*> vom efectua o simplă diferență: $b_{dr} - b_{st-1}$.

Pentru a calcula sumele pentru subsecvențe de forma <1, *dr*> vom avea nevoie de elementul b_0 care va avea întotdeauna valoarea 0. Astfel, pentru o subsecvență de această formă suma elementelor va fi $b_{dr} - b_0 = b_{dr}$.

Așadar, folosind acest artificiu, ordinul de complexitate al unei interogări va fi $O(1)$ pentru că este necesară doar o simplă scădere pentru furnizarea rezultatului. Din nefericire, în momentul efectuării unei modificări pentru elementul i, toate elementele b_j pentru care $j \geq i$ își modifică valoarea.

Ca urmare, operația de modificare a celui de-al i-lea element nu se mai realizează în timp constant, deoarece trebuie modificate $N - i + 1$ valori. Așadar, ordinul de complexitate devine liniar.

Astfel, am reușit să înlocuim algoritmul liniar de determinare a sumei elementelor unei subsecvențe cu un algoritm având ordinul de complexitate $O(1)$ cu prețul creșterii timpului de execuție a algoritmului de modificare de la unul constant la unul liniar.

Se observă că nu mai avem nevoie de șirul *a* folosit pentru algoritmul anterior, fiind suficientă păstrarea valorilor elementelor șirului *b*. Prezentăm versiunea în pseudocod a acestui algoritm, folosind aceleași coduri pentru operațiile efectuate:

```
//inițializări
scrie Introduceți numărul de elemente:
citește N      //dimensiunea șirului
pentru i ← 0, N execută
  b_i ← 0      //valorile inițiale sunt nule
sfârșit pentru
scrie Introduceți codul operației:
citește cod
cât timp cod ≠ 3 execută
  dacă cod = 1 //modificare
    atunci
      scrie Introduceți indicele elementului care va fi
            modificat:
      citește ind
      scrie Introduceți valoarea care va fi adunată
            (valoare negativă pentru scăderi):
      citește val
      pentru i ← ind, N execută
        b_i ← b_i + val
      sfârșit pentru
    altfel //interogare
      scrie Introduceți extremitățile subsecvenței:
      citește st, dr
      scrie Suma elementelor secvenței este b_dr - b_st-1
  sfârșit dacă
  scrie Introduceți codul operației:
  citește cod
sfârșit cât timp
```

Din nou, algoritmul este același dacă operația de însumare este înlocuită cu o alta (calcularea produsului, stabilirea minimului etc.).

Arbore indexat binar

Practic, pentru algoritmii anteriori una dintre cele două operații se efectuează în timp constant, în timp ce cealaltă se efectuează în timp liniar. Așadar, dacă numărul de modificări este aproximativ egal cu cel al interogărilor, timpul de execuție va fi aproximativ același.

Dacă numărul modificărilor este mult mai mare decât cel al interogărilor, atunci este preferabilă folosirea algoritmului naiv.

Dacă, dimpotrivă, numărul interogărilor este mult mai mare decât cel al modificărilor, atunci algoritmul bazat pe vectorul de sume este mai performant. Totuși, în cazul mediu, ambii algoritmi sunt liniari.

În cele ce urmează vom prezenta o structură de date care permite efectuarea în timp logaritmic atât a modificărilor, cât și a interogărilor. Structura de date pe care o propunem este numită ***arbore indexat binar***.

Așadar, vom avea o structură arborescentă care va permite efectuarea interogărilor în timp logaritmic în condițiile în care și modificările se efectuează în timp logaritmic.

Pentru a folosi această structură de date, va trebui să considerăm că elementele șirului sunt numerotate începând cu 1.

Arborele va fi păstrat sub forma unui vector *c* în care fiecare element *i* va conține suma elementelor subsecvenței $<i - 2^k + 1, i>$, unde *k* este numărul zerourilor terminale din reprezentarea binară a lui *i*.

Așadar, elementele de pe pozițiile impare ale arborelui vor păstra suma elementelor unor subsecvențe formate dintr-un singur element (aflat pe o poziție impară în șirul *a*). În elementele de pe pozițiile de forma $4 \cdot k + 2$ (un zero terminal în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din două elemente. În elementele de pe pozițiile de forma $8 \cdot k + 4$ (două zerouri terminale în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din patru elemente.

În general, dacă reprezentarea binară a pozițiilor au *p* zerouri terminale, atunci elementele din arbore vor păstra sume ale elementelor unor subsecvențe cu 2^p elemente.

Vom considera că, la un moment dat, șirul (format din 15 elemente) este (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15).

Valorile vectorului *c* (cel care reprezintă arborele indexat binar) sunt:

Element	*Poziție*	*Semnificație*	*Valoare*
c_1	0001	**Sumă**(1, 1)	1
c_2	0010	**Sumă**(1, 2)	3
c_3	0011	**Sumă**(3, 3)	3
c_4	0100	**Sumă**(1, 4)	10
c_5	0101	**Sumă**(5, 5)	5
c_6	0110	**Sumă**(5, 6)	11
c_7	0111	**Sumă**(7, 7)	7
c_8	1000	**Sumă**(1, 8)	36
c_9	1001	**Sumă**(9, 9)	9
c_{10}	1010	**Sumă**(9, 10)	19
c_{11}	1011	**Sumă**(11, 11)	11
c_{12}	1100	**Sumă**(9, 12)	42
c_{13}	1101	**Sumă**(13, 13)	13
c_{14}	1110	**Sumă**(13, 14)	27
c_{15}	1111	**Sumă**(15, 15)	15

Vom prezenta în continuare modul în care se efectuează modificările, exemplificând pe arborele prezentat anterior. În acest scop, vom modifica valoarea celui de-al doilea element din 2 în 8 (se adună valoarea 6).

Se observă că trebuie modificate toate valorile vectorului *c* care reprezintă sume ale unei subsecvențe care conține al doilea element al șirului; acestea sunt: c_2, c_4 și c_8. Reprezentările binare ale acestor trei poziții sunt `0010`, `0100` și `1000`.

Se observă că, fiecare astfel de reprezentare (evident, cu excepția primeia) poate fi obținută din cea anterioară prin adunarea valorii 2^r, unde *r* reprezintă numărul de zerouri terminale ale reprezentării binare a poziției anterioare.

Așadar, ar trebui să efectuăm astfel de adunări până în momentul în care poziția obținută este mai mare decât dimensiunea șirului și să creștem cu 6 valorile tuturor elementelor de pe pozițiile de la fiecare pas.

Vom exemplifica procedeul și pentru al treilea element. Prima poziție este `0011`; numărul zerourilor terminale este 0, deci vom aduna valoarea $2^0 = 1$. Noua poziție va fi 3 + 1 = 4, deci am ajuns la poziția 4 a cărei reprezentare binară este `0100`. Numărul zerourilor terminale este 2, deci vom aduna valoarea $2^2 = 4$, poziția devenind 4 + 4 = 8. Reprezentarea binară este `1000`, deci avem trei zerouri terminale. Valoarea adunată va fi $2^3 = 8$, deci ajungem în poziția 8 + 8 = 16, așadar am depășit dimensiunea șirului.

Se observă că elementele c_3, c_4 și c_8 sunt cele a căror valoare depinde de valoarea elementului de pe a treia poziție.

Algoritmul care realizează operația de modificare este următorul:

- Se identifică poziția elementului care trebuie modificat.
- Cât timp poziția curentă este cel mult egală cu dimensiunea șirului:
  - Se modifică valoarea elementului de pe poziția curentă.
  - Se determină numărul *k* al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină adunând valoarea 2^k la poziția curentă.

Se observă că numărul elementelor modificate este cel mult egal cu numărul cifrelor reprezentării binare a numărului de elemente din șir, deci operația are ordinul de complexitate $O(\log N)$.

Mai trebuie prezentat modul în care este efectuată operația de interogare. Vom încerca să aplicăm o metodă similară celei propuse în cazul algoritmului care folosește un vector de sume. Pentru aceasta, dacă dorim să determinăm suma elementelor subsecvenței <*st*, *dr*>, vom determina sumele elementelor subsecvențelor <1, *st* - 1> și <1, *dr*>, efectuând apoi o simplă diferență.

Operațiile efectuate sunt, oarecum, inversele celor de la operația de modificare. Vom porni din poziția dată și, la fiecare pas, vom determina următoarea poziție scăzând valoarea 2^k unde *k* reprezintă numărul zerourilor terminale din reprezentarea binară a poziției curente. Ne vom opri în momentul în care poziția curentă devine 0.

De exemplu, pentru a determina suma elementelor subsecvenței <1, 11> vom porni din poziția 11 a cărei reprezentare binară este `1011`. Numărul zerourilor terminale este 0, deci vom scădea valoarea $2^0 = 1$, ajungând în poziția 11 - 1 = 10. Reprezentarea binară a acesteia este `1010`, deci numărul zerourilor terminale este 1. Valoarea scăzută este $2^1 = 2$, deci se ajunge în poziția 10 - 2 = 8, a cărei reprezentare binară este `1000`. De data aceasta avem trei zerouri terminale, deci vom scădea valoarea $2^3 = 8$ ajungând în poziția 8 - 8 = 0.

Așadar, am "trecut" prin pozițiile 11, 10 și 8. Adunând valorile elementelor corespunzătoare (c_{11}, c_{10} și c_8) obținem 11 + 19 + 36 = 66, adică exact valoarea căutată.

Este ușor de observat că c_{11} reprezintă suma elementelor subsecvenței <11, 11>, c_{10} reprezintă suma elementelor subsecvenței <9, 10>, iar c_8 reprezintă suma elementelor subsecvenței <1, 8>, așadar, în momentul calculării sumei, fiecare element de pe o poziție mai mică sau egală cu 11 este adunat o singură dată.

Pentru a determina suma elementelor unei subsecvențe de forma <1, *dr*> vom folosi următorul algoritm:

- Se identifică elementul de pe poziția *dr*.
- Cât timp poziția curentă este diferită de 0:
  - Se adună la suma totală valoarea elementului de pe poziția curentă.
  - Se determină numărul *k* al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină scăzând valoarea 2^k din poziția curentă.

Evident, pentru a determina suma elementelor unei subsecvențe de forma <*st*, *dr*> vom aplica de două ori algoritmul prezentat: o dată pentru subsecvența <1, *dr*> și o dată pentru subsecvența <1, *st*-1>, efectuând la final diferența valorilor obținute.

De exemplu, pentru determinarea sumei elementelor subsecvenței <4, 13> vom calcula mai întâi sumele elementelor subsecvențelor <1, 13> și <1, 3>. Acestea sunt **Sumă**(1, 13) = $c_{13} + c_{12} + c_8 = 13 + 42 + 36 = 91$ și **Sumă**(1, 3) = $c_3 + c_2 = 3 + 3 = 6$, diferența lor fiind 91 - 6 = 85 care este egală cu valoarea **Sumă**(4, 13).

Se observă că numărul elementelor adunate este cel mult egal cu numărul cifrelor reprezentării binare a poziției elementului dat, deci operația are ordinul de complexitate $O(\log N)$.

Din nou, se observă că nu avem nevoie de păstrarea elementelor șirului *a*, fiind suficientă păstrarea valorilor vectorului *c* (cel care reprezintă arborele indexat binar).

Varianta în pseudocod a algoritmului de rezolvare a problemei folosind un arbore indexat binar este următoarea:

```
//inițializări
scrie  Introduceți numărul de elemente:
citește N      //dimensiunea șirului
pentru i ← 0, N execută
   c_i ← 0      //valorile inițiale sunt nule
sfârșit pentru
```

```
scrie Introduceți codul operației:
citește cod
cât timp cod ≠ 3 execută
  dacă cod = 1 //modificare
    atunci
      scrie Introduceți indicele elementului care va fi
            modificat:
      citește ind
      scrie Introduceți valoarea care va fi adunată
            (valoare negativă pentru scăderi):
      citește val
      poz ← 0  //poziția celui mai nesemnficativ bit cu
               //valoarea 1
      cât timp ind ≤ N execută
        c_ind ← c_ind + val
        cât timp ind & 2^poz ← 0  execută
                    //& reprezintă operația de conjucție
                    //logică (ȘI-logic)
          poz ← poz + 1
        sfârșit cât timp
        ind ← ind + 2^poz
        poz ← poz + 1
                    //valoarea poz este incrementată
                    //și nu reinițializată cu 0 deoarece
                    //știm că acum avem cel puțin
                    //poz + 1 zeroruri terminale, prin
                    //adunare adăugându-se cel puțin
                    //un zero terminal
      sfârșit cât timp
    altfel  //interogare
      scrie Introduceți extremitățile subsecvenței:
      citește st, dr
      //calcularea primei sume
      s1 ← 0    //inițializare
      poz ← 0  //poziția celui mai nesemnficativ bit cu
               //valoarea 1
      cât timp dr > 0 execută
        s1 ← s1 +  c_dr
        cât timp dr & 2^poz = 0 execută
          poz ← poz + 1
        sfârșit cât timp
        dr ← dr - 2^poz
        poz ← poz + 1  //prin scădere se adaugă cel
                       //puțin un zero terminal
      sfârșit cât timp
      //calcularea celei de-a doua sume
      st ←  st - 1  //trebuie calculată valoarea pentru
                    //st-1
      s2 ← 0      //inițializare
      poz ← 0  //poziția celui mai nesemnficativ bit cu
               //valoarea 1
      cât timp  st > 0  execută
        s2  ←  s2  +  c_st
        cât timp st & 2^poz = 0  execută
          poz ← poz + 1
        sfârșit cât timp
        st ← st - 2^poz
        poz ← poz + 1
      sfârșit cât timp
      scrie Suma elementelor subsecvenței este: s1 -
            s2
  sfârșit dacă
  scrie Introduceți codul operației:
  citește cod
sfârșit cât timp
```

Și de data aceasta putem înlocui operația de însumare cu alte operații (înmulțire, minim, maxim etc.).

Cazul bidimensional

Problema enunțată poate fi modificată astfel încât să lucrăm în spațiul bidimensional. Enunțul noii probleme ar putea fi următorul:

Se consideră o matrice de numere întregi care are toate elementele nule. O modificare a unui element al matricei constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei submatrice.

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă dreptunghiulară" care face parte din matrice.

Evident, pentru reprezentarea datelor vom folosi tablouri bidimensionale în locul celor unidimensionale (matrice în locul vectorilor).

Din nou, enunțul poate fi reformulat în diferite moduri.

O variantă ar putea fi:

Se consideră un teren de formă dreptunghiulară având lungimea M și lățimea N. Terenul este împărțit în regiuni de formă pătrată având latura egală cu unitatea. În fiecare moment, într-o anumită regiune pot fi plantați sau tăiați pomi. De asemenea, în fiecare moment proprietarul terenului ar putea cere informații referitoare la numărul total al pomilor care se află într-o regiune dreptunghiulară a terenului.

Vom adapta cei trei algoritmi descriși pentru cazul unidimensional pentru a rezolva noua problemă și vom studia apoi performanțele acestora.

De data aceasta, pentru operația de modificare vom avea nevoie de doi parametri care vor reprezenta linia și coloana pe care se află elementul a cărui valoare va fi modificată.

Pentru interogări vom avea nevoie de patru parametri care vor reprezenta coordonatele colțurilor din stânga-sus și dreapta-jos ale unui dreptunghi.

Vom exemplifica acum cazul bidimensional al problemei folosind o matrice cu trei linii și trei coloane.

Operație	*Matrice/Rezultat*
Inițializare	0 0 0
	0 0 0
	0 0 0
Adună(2, 2, 3)	0 0 0
	0 3 0
	0 0 0
Adună(1, 3, 5)	0 0 5
	0 3 0
	0 0 0
Sumă(2, 1, 3, 3)	3
Adună(2, 1, 1)	0 0 5
	1 3 0
	0 0 0
Sumă(1, 2, 2, 3)	8
Adună(2, 2, -1)	0 0 5
	1 2 0
	0 0 0
Sumă(1, 2, 2, 3)	7
Adună(2, 2, 5)	0 0 5
	1 7 0
	0 0 0
Adună(2, 2, -5)	0 0 5
	1 2 0
	0 0 0
Adună(3, 2, 4)	0 0 5
	1 2 0
	0 4 0
Sumă(1, 1, 3, 3)	12

Algoritmul naiv

Primul algoritm descris pentru cazul unidimensional poate fi adaptat foarte ușor pentru a rezolva cazul bidimensional al problemei. Vom păstra valorile matricei, le vom modifica dacă este necesar și vom calcula sumele cerute de interogări.

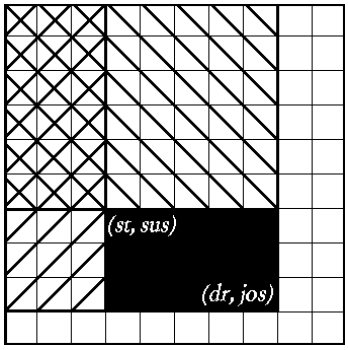
Versiunea în pseudocod este prezentată în continuare:

```
//inițializări
scrie Introduceți dimensiunile matricei:
citește M, N
pentru i ← 1, M execută
  pentru j ← 1, N execută
    a_ij ← 0      //valorile inițiale sunt nule
  sfârșit pentru
sfârșit pentru
scrie Introduceți codul operației:
citește cod
cât timp cod ≠ 3 execută
  dacă cod = 1  //modificare
    atunci
      scrie Introduceți indicii elementului care va fi
            modificat:
      citește indx, indy
      scrie Introduceți valoarea care va fi adunată
            (valoare negativă pentru scăderi):
      citește val
      a_indx,indy ← a_indx,indy + val
    altfel  //interogare
      scrie Introduceți coordonatele colțurilor:
      citește st, sus, dr, jos
      suma ← 0
      pentru i ← sus, jos execută
        pentru j ← st, dr execută
          suma ← suma + a_ij
        sfârșit pentru
      sfârșit pentru
      scrie Suma elementelor dreptunghiului este: suma
  sfârșit dacă
  scrie Introduceți codul operației
  citește cod
sfârșit cât timp
```

Modificările se efectuează tot în timp constant deoarece implică doar accesarea unui element al matricei și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor dintr-un dreptunghi este necesară parcurgerea tuturor elementelor dreptunghiului, această operație are ordinul de complexitate $O(l \cdot h)$, unde l și h sunt lungimile laturilor dreptunghiului.

Matrice de sume



Adaptarea algoritmului care folosește un vector de sume este destul de simplă. Vom păstra o matrice b ale cărei elemente b_{ij} vor conține sumele corespunzătoare dreptunghiurilor cu colțul din stânga-sus în poziția (1, 1) și cel din dreapta-jos în poziția (i, j). Studiind figura alăturată, este ușor de observat că pentru a calcula valoarea **Sumă**(*st*, *sus*, *dr*, *jos*) poate fi folosită relația:

Sumă(*st*, *sus*, *dr*, *jos*) = **Sumă**(1, 1, *dr*, *jos*) -
Sumă(1, 1, *st* - 1, *jos*) -
Sumă(1, 1, *dr*, *sus* - 1) +
Sumă(1, 1, *st* - 1, *sus* - 1).

Se observă că prin scăderea valorii **Sumă**(1, 1, *st*-1, *jos*) se scad valorile tuturor elementelor aflate la dreapta dreptunghiului considerat, iar prin scăderea valorii **Sumă**(1, 1, *dr*, *sus*-1) se scad valorile elementelor aflate deasupra dreptunghiului.

Aceste operații implică scăderea de două ori a tuturor elementelor aflate la stânga și deasupra dreptunghiului considerat, motiv pentru care va trebui să adunăm valoarea **Sumă**(1, 1, *st*-1, *sus*-1).

Folosind acest artificiu avem nevoie de accesarea a patru elemente ale matricei b, deci operația se realizează în timp constant.

Din nou, în momentul modificării valorii unui element de coordonate (x, y), va trebui să modificăm valorile tuturor elementelor matricei b de coordonate (i, j) pentru care $i \geq x$ și $j \geq y$. Așadar, vom modifica $(M - x + 1) \cdot (N - y + 1)$ elemente, ordinul de complexitate devenind $O(M \cdot N)$.

Pentru a putea determina valori pentru dreptghiuri în care colțul din stânga-sus are una dintre coordonate egală cu 1 va trebui să considerăm că valorile elementelor matricei sunt nule dacă cel puțin unul dintre cei doi indici este 0.

Prezentăm versiunea în pseudocod a algoritmului descris:

```
//inițializări
scrie Introduceți dimensiunile matricei:
citește M, N      //dimensiunea șirului
pentru i = 0, M execută
   pentru i = 0, N execută
      b_ij = 0       //valorile inițiale sunt nule
   sfârșit pentru
sfârșit pentru
scrie Introduceți codul operației:
citește cod
cât timp cod ≠ 3 execută
   dacă cod = 1  //modificare
      atunci
         scrie Introduceți indicii elementului care va
               fi modificat:
         citește indx, indy
         scrie Introduceți valoarea care va fi adunată
               (valoare negativă pentru scăderi):
         citește val
         pentru i ← indx, M execută
            pentru i ← indy, N execută
               b_ij ← b_ij + val
            sfârșit pentru
         sfârșit pentru
      altfel //interogare
         scrie Introduceți coordonatele colțurilor:
         citește st, sus, dr, jos
         scrie Suma elementelor secvenței este: b_dr,jos -
               b_st-1,jos - b_dr,sus-1 + b_st-1,sus-1.
   sfârșit dacă
   scrie Introduceți codul operației
   citește cod
sfârșit cât timp
```

Din nou, algoritmul este foare asemănător dacă operația de însumare este înlocuită cu o alta (calcularea produsului, stabilirea minimului etc.).

Arbore de arbori indexați binar

Din nou, avem doi algoritmi în care una dintre operații este eficientă, în timp ce cealaltă necesită un timp de execuție mai mare. Folosind arborii indexați binar, vom putea găsi algoritmi care realizează ambele operații într-un timp de ordinul $O(\log M \cdot \log N)$, unde M și N sunt dimensiunile matricei.

Pentru a rezolva problema vom crea un ***arbore de arbori indexați binar***.

Așadar, vom avea o matrice c ale cărei elemente c_{ij} vor reprezenta sumele elementelor din dreptunghiul care are colțul din stânga-sus în poziția $(i - 2^k + 1, j - 2^l + 1)$, iar cel din dreapta-jos în poziția (i, j). Valoarea k reprezintă numărul zerourilor terminale din reprezentarea binară a lui i, iar l reprezintă numărul zerourilor terminale din reprezentarea binară a lui j.

De exemplu, pentru matricea:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
```

elementele arborelui de arbori indexați binar vor avea următoarele semnficații:

Sumă(1, 1, 1, 1) **Sumă**(1, 1, 1, 2) **Sumă**(1, 3, 1, 3) **Sumă**(1, 1, 1, 4)
Sumă(1, 1, 2, 1) **Sumă**(1, 1, 2, 2) **Sumă**(1, 3, 2, 3) **Sumă**(1, 1, 2, 4)
Sumă(3, 1, 3, 1) **Sumă**(3, 1, 3, 2) **Sumă**(3, 3, 3, 3) **Sumă**(3, 3, 3, 4)
Sumă(1, 1, 4, 1) **Sumă**(1, 1, 4, 2) **Sumă**(1, 3, 4, 3) **Sumă**(1, 1, 4, 4)

Așadar, valorile matricei care reprezintă arborele binar vor fi:

```
1  3  3  6
6  14 10 36
9  19 11 23
28 60 36 136
```

Să presupunem că trebuie să modificăm valoarea elementului de coordonate (2, 1) care face parte dintr-o matrice cu 8 linii și 4 coloane. Vom aplica de două ori principiul descris pentru cazul unidimensional.

Vom identifica linia i pe care se află elementul și vom efectua modificări în toate coloanele de pe acea linie în care acestea trebuie efectuate. Pentru aceasta vom identifica coloana j, vom determina numărul k al zerourilor terminale și apoi vom aduna la j valoarea 2^k. Ajungem într-o nouă poziție și continuăm procedeul până în momentul în care valoarea coloanei curente depășește numărul de coloane al matricei.

În acest moment vom determina numărul zerourilor terminale l din reprezentarea binară a valorii i și ne vom "muta" pe linia $i + 2^l$. Vom reveni la coloana j și vom parcurge aceeași pași ca și la linia anterioară.

Vom continua până în momentul în care valoarea liniei curente va fi mai mare decât numărul de linii ale matricei.

Pe fiecare linie vom modifica cel mult $\log_2 N$ elemente; în plus, vom modifica elemente de pe cel mult $\log_2 M$ linii, așadar, ordinul de complexitate al operației de modificare a unui element este $O(\log M \cdot \log N)$.

Ca urmare, dacă se modifică elementul de coordonate (3, 1), atunci vor trebui modificate următoarele elemente

ale matricei care reprezintă arborele de arbori indexați binar: c_{31}, c_{32}, c_{34}, c_{41}, c_{42}, c_{44}, c_{81}, c_{82} și c_{84}.

În cazul în care dorim să efectuăm o interogare vom folosi aceeași metodă. Singura diferență este, din nou, faptul că valorile de forma 2^k sunt scăzute în loc să fie adunate.

Datorită faptului că, de data aceasta, avem nevoie de patru sume pentru a furniza rezultatul, algoritmul va fi aplicat de patru ori. Ordinul de complexitate al algoritmului va fi $O(\log M \cdot \log N)$.

Vom prezenta în continuare versiunea în pseudocod a algoritmului de rezolvare a problemei în cazul bidimensional, folosind un arbore de arbori indexați binar.

```
//inițializări
scrie Introduceți dimensiunile matricei:
citește M, N
pentru i ← 0, M execută
  pentru j ← 0, N execută
    c_ij ← 0        //valorile inițiale sunt nule
  sfârșit pentru
sfârșit pentru
scrie Introduceți codul operației
citește cod
cât timp cod ≠ 3 execută
  dacă cod = 1  //modificare
    atunci
      scrie Introduceți indicii elementului care va fi
            modificat:
      citește indx, indy
      scrie Introduceți valoarea care va fi adunată
            (valoare negativă pentru scăderi):
      citește val
      pozi ← 0
      cât timp indx ≤ M execută
        pozj ← 0
        j ← indy
        cât timp j ≤ N execută
          c_indx,j ← c_indx,j + val
          cât timp indx & 2^pozj = 0 execută
            pozj ← pozj + 1
          sfârșit cât timp
          j ← j + 2^pozj
          pozj ← pozj + 1
        sfârșit cât timp
        cât timp indx & 2^pozi = 0 execută
          pozi ← pozi + 1
        sfârșit cât timp
        indx ← indx + 2^pozi
        pozi ← pozi + 1
      sfârșit cât timp
    altfel  //interogare
      scrie Introduceți coordonatele colțurilor:
      citește st, sus, dr, jos
      //calcularea primei sume
      s ← 0
      x ← jos
      y ← dr
      pozi ← 0
      cât timp x ≥ 0 execută
        pozj ← 0
        j ← y
        cât timp y ≥ 0 execută
                s1 ← s1 + c_x,j
          cât timp x & 2^pozj = 0 execută
            pozj ← pozj + 1
          sfârșit cât timp
          j ← j - 2^pozj
          pozj ← pozj + 1
        sfârșit cât timp
        cât timp x & 2^pozi = 0 execută
          pozi ← pozi + 1
        sfârșit cât timp
        x ← x - 2^pozi
        pozi ← pozi + 1
      sfârșit cât timp
      s1 ← s
      s ← 0
      //calcularea celei de-a doua sume
      x ← jos
      y ← st - 1
      ...  //se aplică exact aceeași secvență
      s2 ← s
      s ← 0
      //calcularea celei de-a treia sume
      x ← sus - 1
      y ← dr
      ...  //se aplică din nou exact aceeași secvență
      s3 ← s
      s ← 0
      //calcularea celei de-a patra sume
      x ← sus - 1
      y ← dr - 1
      ...  //se aplică din nou exact aceeași secvență
      s4 ← s
      scrie Suma elementelor dreptunghiului este s1
            - s2 - s3 + s4
  sfârșit dacă
  scrie Introduceți codul operației:
  citește cod
sfârșit cât timp
```

Cazul tridimensional

Problema enunțată poate fi modificată astfel încât să lucrăm în spațiul tridimensional. Enunțul ar putea fi următorul:

Se consideră un tablou tridimensional care conține numere întregi și toate elementele sale sunt nule. O modificare a unui element al tabloului constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unui subtablou "paralelipipedic" .

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă paralelipipedică" a tabloului. Evident, pentru reprezentarea datelor vom folosi tablouri tridimensionale.

Și acest enunț poate fi reformulat în diferite moduri. O variantă ar putea fi:

O zonă a spațiului este reprezentată de un paralelipiped format din cuburi cu latura egală cu unitatea. Dimensiunile paralelipipedului sunt M, N și P. În fiecare sector (cub) pot sosi sau pleca nave spațiale. De asemenea, în fiecare moment amiralul flotei poate cere informații referitoare la numărul total de nave spațiale dintr-o regiune paralelipipedică din această zonă a spațiului.

Nu vom mai prezenta pe larg cele trei tipuri de algoritmi; vom face doar câteva precizări care ne vor ajuta să generalizăm problema pentru spații *n*-dimensionale.

În primul rând, în fiecare poziție în care algoritmii pentru cazul bidimensional conțineau două cicluri imbricate, pentru algoritmii care rezolvă cazul tridimensional vom avea trei cicluri. Ca urmare, ordinul de complexitate al primilor doi algoritmi va deveni $O(M \cdot N \cdot P)$. Ordinul de complexitate al celui de-al treilea algoritm va fi $O(\log M \cdot \log N \cdot \log P)$.

În al doilea rând, elementele tabloului tridimensional vor fi identificate prin trei indici. Așadar, pentru o modificare vom avea nevoie de trei parametri "geometrici", iar pentru o interogare de șase astfel de parametri.

În al treilea rând, pentru ultimii doi algoritmi va trebui să calculăm opt sume în loc de patru. Formula de calcul va fi următoarea:

$$\begin{aligned}\textbf{Sumă}(x_1, y_1, z_1, x_2, y_2, z_2) = \ & \textbf{Sumă}(1, 1, 1, x_2, y_2, z_2) - \\ & \textbf{Sumă}(1, 1, 1, x_1 - 1, y_2, z_2) - \\ & \textbf{Sumă}(1, 1, 1, x_2, y_1 - 1, z_2) - \\ & \textbf{Sumă}(1, 1, 1, x_2, y_2, z_1 - 1) + \\ & \textbf{Sumă}(1, 1, 1, x_1 - 1, y_1 - 1, z_2) + \\ & \textbf{Sumă}(1, 1, 1, x_1 - 1, y_2, z_1 - 1) + \\ & \textbf{Sumă}(1, 1, 1, x_2, y_1 - 1, z_1 - 1) - \\ & \textbf{Sumă}(1, 1, 1, x_1 - 1, y_1 - 1, z_1 - 1)\end{aligned}$$

Mai precizăm faptul că pentru cel de-al treilea algoritm vom folosi un tablou tridimensional care va reprezenta un *arbore de arbori de arbori indexați binar*.

Toate celelalte aspecte ale celor trei algoritmi descriși pot fi adaptate foarte simplu pentru a rezolva problema tridimensională.

Cazul n-dimensional

Vom generaliza acum problema pentru a putea fi enunțată într-un spațiu cu un număr oricât de mare de dimensiuni. O variantă a enunțului este:

Se consideră un tablou n-dimensional care conține numere întregi și toate elementele sale sunt nule. O modificare a unui element al tabloului constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unui subtablou.

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă *n*-dimensională" a tabloului. Evident, pentru reprezentarea datelor vom folosi tablouri *n*-dimensionale.

De data această, cele două cicluri imbricate din algoritmii pentru cazul bidimensional (cele trei din algoritmii pentru cazul tridimensional) vor fi înlocuite de *n* cicluri imbricate.

Ca urmare, ordinul de complexitate al primilor doi algoritmi va depinde de produsul celor *n* dimensiuni, iar ordinul de complexitate al celui de-al treilea algoritm va depinde de produsul logaritmilor celor *n* dimensiuni.

Evident, vom avea acum *n* parametri "geometrici" pentru o modificare și $2 \cdot n$ parametri de acest tip pentru o interogare. Numărul sumelor care trebuie calculate pentru o interogare (la al doilea și al treilea algoritm) este, pentru cazul general, 2^n.

Ținând cont de aceste observații, algoritmii prezentați pot fi folosiți pentru rezolvarea unor probleme de acest tip în care numărul dimensiunilor este oricât de mare.

Concluzii

În cadrul acestui articol am prezentat o structură de date care permite rezolvarea eficientă a unei categorii de probleme. Pentru a evidenția performanțele algoritmilor care folosesc o astfel de structură de date, am prezentat și alte două categorii de algoritmi, mult mai ușor de implementat, dar ineficienți.

De asemenea, am prezentat modul în care pot fi generalizați algoritmii folosiți pentru cazul cel mai simplu (cel unidimensional) pentru a rezolva problema pentru cazuri în care numărul dimensiunilor este oricât de mare.

Am exemplificat algoritmii pentru cazul în care interogările se referă la suma anumitor elemente. Cu toate acestea este evident faptul că, în cazul tuturor acestor algoritmi, această operație poate fi înlocuită cu altele (produsul elementelor, minimul sau maximul lor etc.).

Bibliografie

1. **J. Nummenmaa, E. Mäkinen, I. Aho**, *IOI '01 Competition*, Tampere, Finland, 2001
2. **P. M. Fenwick**, *A new data structure for cumulative frequency tables*, Software-Practice and Experience, vol. 24, no. 3, p. 327-336, 1994
3. **T. H. Cormen, C. E. Leiserson, R. R. Rivest**, *Introducere în algoritmi*, Computer Libris Agora, Cluj-Napoca, 2000

Mihai Scorțaru este redactor-șef al GInfo. Poate fi contactat prin e-mail la adresa **skortzy@yahoo.com**.