SEPI

Societatea pentru Excelență și Performanță în Informatică

Infobits Academy

F1! Algoritmi și structuri de date

Fi! Algoritmi și structuri de date

Marius NICOLI
Cătălin FRÂNCU
Mircea ROTAR
Eugen NODEA
Zoltan SZABÓ

Emanuela CERCHEZ
Daniela LICA
Dan PRACSIU
Ștefan DĂSCĂLESCU
Andrei ONUȚ

INFOBITS ACADEMY

Tehnoredactare

Cătălin Frâncu, Emanuela Cerchez

Coperta

Emanuela Cerchez, Vlad Tudor
(pe baza unui design generat de Midjourney AI)

ISBN

978-630-6559-05-3

Materialul este distribuit

Spor la studiu!

Date de contact

Societatea pentru Excelență și Performanță în Informatică
www.sepi.ro
contact@sepi.ro

Infobits Academy
https://ebooks.infobits.ro
hello@infobits.ro

Cuprins

Cuvânt înainte

Informatica este, probabil, știința cu dinamica cea mai accelerată și dovada se află peste tot în jurul nostru. Viteza cu care se schimbă lumea în care muncim, comunicăm, trăim este o reflexie a dinamicii tehnologiei și, în special, a tehnologiei informației. Olimpiadele de informatică reflectă și ele, evident, această dinamică. Problema cea mai dificilă de la Olimpiada Internațională de Informatică din 1994 din Suedia – „The triangle" – este, astăzi, cea mai simplă problemă pe care ilustrăm la clasă modul de funcționare a programării dinamice. În 2001, la Olimpiada Internațională de Informatică din Finlanda, a fost propusă, pentru prima dată, o problemă a cărei soluție se baza pe arbori indexați binar, problemă cu grad foarte înalt de dificultate la acel moment. Astăzi, elevi de gimnaziu utilizează această structură de date, fără emoție și cu o profundă înțelegere.

România a fost, permanent, în avangarda competițiilor de informatică, atât prin rezultatele obținute de elevi în competiții, cât și prin inițierea unor concursuri internaționale de referință în peisajul competițional actual – BOI (Balkan Olympiad in Informatics) în 1993, CEOI (Central European Olympiad in Informatics) în 1994, RMI (Romanian Masters in Informatics) în 2014 sau, mai nou, IIOT (International Informatics Olympiad in Teams). În ceea ce privește rezultatele, România continuă să se mențină pe locul al doilea „all times" în Hall of Fame-ul Olimpiadei Internaționale de Informatică, cu 127 de medalii (33 aur, 58 argint, 36 bronz), la egalitate cu Polonia, pe locul I situându-se China.

În anul școlar 2023-2024, programa Olimpiadei Naționale de Informatică a fost publicată într-un format nou, care detaliază tematica, specificând metode de programare și structuri de date care apar în problemele de concurs, dar care nu sunt obligatoriu incluse și în programa școlară. *F1! Algoritmi și structuri de date* este o colecție de articole scrise de membri ai Societății pentru Excelență și Performanță în Informatică (SEPI), ca auxiliar al programei. Autorii sunt profesori cu experiență în pregătirea de performanță a elevilor, atât la clasă, cât și în Centre de Excelență sau Centre de pregătire pentru performanță în Informatică. Temele abordate au un grad mai înalt de dificultate sau doar de noutate, pentru fiecare temă fiind incluse, pe lângă considerentele teoretice, probleme rezolvate, precum și probleme propuse spre rezolvare. Scopul acestui demers este de a veni în sprijinul elevilor pasionați de programarea competitivă, precum și al profesorilor care îi coordonează pe drumul lor către performanță. Mult succes!

Partea I

Structuri de date arborescente

Capitolul 1

Arbori indexați binar

prof. Daniela Lica
Colegiul Național „Ion Luca Caragiale" Ploiești
Centrul Județean de Excelență Prahova

1.1 Arbori indexați binar unidimensionali - AIB 1D

Definiția 1.1. Arborii indexați binar reprezintă o structură de date care permite efectuarea eficientă a următoarelor operații în complexitatea de timp $\mathcal{O}(\log N)$ pentru un vector V ce conține N elemente:

- ADD(x, y) - adună la elementul de pe poziția x din vectorul V valoarea y: `V[x] += y` (unde $1 \leq x \leq N$);
- SUM(x) - calculează suma primelor x elemente din vectorul V (se presupune că V este indexat de la 1 la N): $V[1] + V[2] + \cdots + V[x]$ (unde $1 \leq x \leq N$).

Operațiile descrise mai sus pot fi efectuate, în mod normal, în complexitatea menționată folosind un arbore de intervale. Totuși, folosirea unui arbore indexat binar oferă anumite avantaje, cum ar fi:

- implementarea este mai ușoară, cu mai puține linii de cod;
- se folosesc exact N locații de memorie pentru stocarea arborelui în memorie (adică, arborele indexat binar folosește ca suport un vector de N elemente);
- operațiile se pot extinde relativ ușor pentru a rezolva aceeași problemă în mai multe dimensiuni (în complexitatea de timp $\mathcal{O}((\log_2 N)^d)$, unde d reprezintă dimensiunea problemei; de exemplu, $d = 2$, în cazul în care problema se referă la o matrice).

Din păcate, această structură de date nu oferă aceeași flexibilitate precum un arbore de intervale, existând și seturi de operații care pot fi efectuate în complexitatea logaritmică dorită folosind doar arbori de intervale.

Eficiența acestei structuri de date nu are legătură cu echilibrarea arborelui, precum în cazul altor structuri de date arborescente, ci cu modul de indexare a elementelor, ținând cont de reprezentarea binară a indicilor (de unde și numele de arbore indexat binar).

Ideea de bază este că, așa cum un număr pozitiv n poate fi exprimat în mod unic ca sumă de puteri întregi ale lui 2, și un interval $[1 \dots n]$ $(n \geq 1)$ poate fi exprimat ca reuniunea unor (sub)intervale de lungimi întregi egale cu puteri ale lui 2.

Pentru exemplificare, vom trata intervalul $[1 \dots 11]$:

$$11 = 2^3 + 2^1 + 2^0 = 1011_{(2)}$$

Eliminând cel mai puțin semnificativ bit de 1 din reprezentarea lui 11 în baza 2 (adică, bitul 0), constatăm că suma pentru intervalul $[1 \dots 11]$ poate fi calculată astfel:

$$\text{SUM}([1 \dots 11]) = \text{SUM}([1 \dots 10]) + \text{SUM}([11 \dots 11])$$

Figura 1.1

$$\text{SUM}([1 \dots 10]) = \text{SUM}([1 \dots 8]) + \text{SUM}([9 \dots 10])$$

Figura 1.2

Figura 1.3

$\text{SUM}([1 \dots 8]) =$ `AIB[8]` , deoarece 8 are un singur bit de 1 în reprezentarea binară.

Astfel, dacă se asociază fiecărui index x din arbore un interval $[x - 2^k + 1 \dots x]$, unde k reprezintă numărul zerourilor terminale din reprezentarea binară a lui x (k poate fi și 0, în cazul în care x este un număr impar), fiecare interval de forma $[1 \dots x]$ poate fi exprimat ca reuniunea a cel mult $\log_2 N$ intervale, folosind procedeul (descris mai sus) de eliminare succesivă a celui mai nesemnificativ bit de 1. Având stabilită această structură, operația $\text{SUM}(x)$ se efectuează însumând suma intervalelor din descompunerea lui $[1 \dots x]$.

Considerăm un arbore indexat binar cu 15 elemente (distribuția elementelor este reprezentată în Figura 1.4). Pentru a actualiza valoarea unui element x (operația $\text{ADD}(x, y)$), se actualizează prima dată intervalul asociat lui x ($[x - 2^k + 1 \dots x]$) cât și restul intervalelor care îl conțin pe x. Acest lucru se efectuează adăugând succesiv cel mai nesemnificativ bit de 1. Spre exemplu, pentru a actualiza elementul de pe poziția 9, se modifică următoarele elemente (se consideră că sunt 15 elemente):

- 9 $(9 = 1001_{(2)})$ - $[9 \dots 9]$
- 10 $(10 = 1010_{(2)})$ - $[9 \dots 10]$
- 12 $(12 = 1100_{(2)})$ - $[9 \dots 12]$

Prezentăm o justificare a faptului că plecând de la x adunând succesiv cel mai nesemnificativ bit de 1 se actualizează toate intervalele ce conțin poziția x. Să considerăm o valoare oarecare y mai mare strict decât x. Fie b primul bit care diferă în reprezentările binare ale lui x și y, începând de la cel mai semnificativ bit (corespunzător celei mai mari puteri a lui 2 din scrierea binară). Primul caz este atunci când bitul b este egal cu 1 pentru x și egal cu 0 pentru y. Cele două valori arată, scrise în baza 2, astfel:

$$x = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 1\ a_{b-1}\ \ldots\ a_0$$
$$y = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 0\ \ldots$$

Se observă că y este mai mic decât x, prin urmare intervalul lui y nu îl poate conține pe x. Al doilea caz este atunci când bitul b este egal cu 0 pentru x și egal cu 1 pentru y. Valorile arată astfel:

$$x = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 0\ a_{b-1}\ \ldots\ a_0$$
$$y = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 1\ \ldots$$

Vom demonstra că pentru ca x să aparțină intervalului lui y, este necesar ca b să fie cel mai nesemnificativ bit al lui y.

Dacă x nu conține niciun bit de 1 mai nesemnificativ decât b, atunci $y - 2^b + 1 > x$, deci este necesar ca x să conțină măcar un bit de 1 mai nesemnificativ decât b.

Presupunem că există cel puțin un bit al lui y mai nesemnificativ decât b. Fie y' valoarea pe care obținem dacă eliminăm toți biții lui y mai nesemnificativi decât b. Datorită presupunerii $y' \leq y - 2^k$, deci $y' < y - 2^k + 1$ unde k e cel mai nesemnificativ bit al lui y. În acest caz, avem:

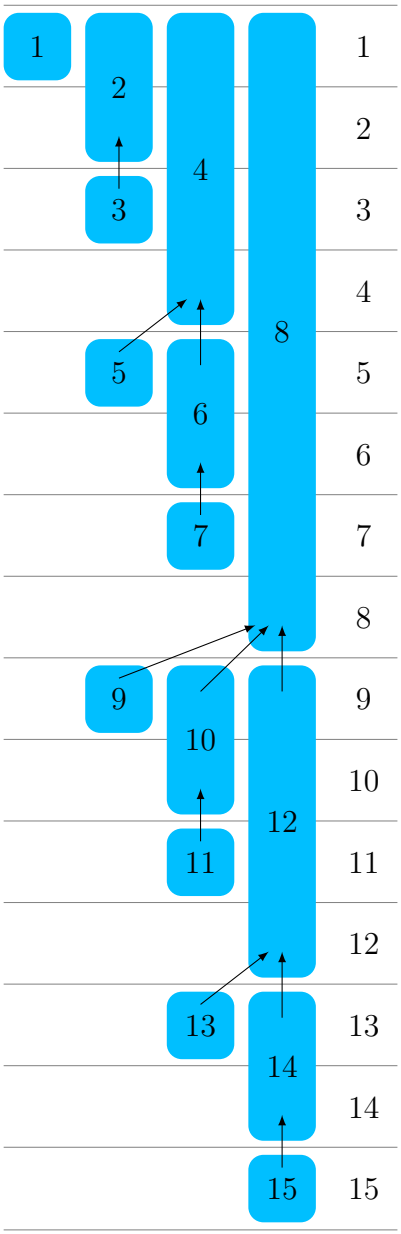

Figura 1.4

$$x = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 0\ a_{b-1}\ \ldots\ a_0$$
$$y' = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 1\ 0\ \ldots\ 0$$

Se observă că $x < y'$, de unde reiese că $x < y - 2^k + 1$, deci x nu aparține intervalului lui y. Prin urmare, pentru ca x să aparțină intervalului lui y, cel mai nesemnificativ bit al lui y trebuie să fie b. Vom avea:

$$x = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 0\ a_{b-1}\ \ldots\ a_0$$
$$y = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 1\ 0\ \ldots\ 0\ 0$$
$$y - 2^k + 1 = a_n\ a_{n-1}\ \ldots\ a_{b+1}\ 0\ 0\ \ldots\ 0\ 1$$

De aici reiese că $y-2^k+1 \leq x$, deci x aparține intervalului lui y pentru toate aceste valori. Trebuie să găsim toate valorile cu proprietatea că sunt formate selectând un bit b care este egal cu 0 în reprezentarea lui x, eliminând toți biții de 1 ai lui x mai nesemnificativi (de notat că este necesar să existe măcar unul) și transformând bitul b în 1. Adunând la x cel mai nesemnificativ bit k al său, de fiecare dată vom transforma întreaga subsecvență de biți de 1 ce îl conține pe k în biți de 0, iar primul bit de 0 mai semnificativ decât k va deveni egal cu 1. Prin urmare, acest procedeu găsește toate intervalele dorite. Facem observația că intervalul corespunzător lui x $[x - 2^{ub(x)} + 1, x]$ va fi actualizat la început.

Pentru a determina în $\mathcal{O}(1)$ cel mai nesemnificativ bit de 1 al unui număr, se poate folosi expresia: $(x \ \& \ (-x))$.

Implementare

```
#include <bits/stdc++.h>
int aib[nmax + 5];

int ub(int x)
{
  return (x & (-x));
}

void add(int x, int y)
{
  for (int i=x; i<=n; i+=ub(i))
    {
      aib[i] += y;
    }
}

int sum(int x)
{
  int rez = 0;
  for (int i=x; i>=1; i-=ub(i))
    {
      rez += aib[i];
    }
  return rez;
}
```

Deși această structură de date nu este la fel de flexibilă ca un arbore de intervale, ea poate fi extinsă să suporte și alte operații:

- SUMA(x, y) - suma elementelor cu poziții între x și y: se poate rescrie ca suma elementelor din intervalul $[1 \ldots y]$ minus suma elementelor din intervalul $[1 \ldots x-1]$;
- UPDATE(x, y) - actualizează valoarea elementului de pe poziția x la maximul dintre aceasta și y;
- MAXIM(x) - determină maximul din primele x elemente.

Aceste operații pot fi implementate înlocuind operația de adunare („+”) cu cea de maxim.

1.2 Arbori indexați binar bidimensionali - AIB 2D

Datorită structurii lor, arborii indexați binar pot fi extinși pentru a suporta următoarele operații în complexitate $\mathcal{O}((\log_2 N)^2)$ pe o matrice a de $N \times N$ elemente:

- ADD(x, y, val) - adună la elementul de pe poziția (x, y) din matrice valoarea val;
- SUM(x, y) - calculează suma elementelor din submatricea având colțurile (1,1) și (x, y).

Pentru a realiza aceste operații, vom construi inițial un arbore indexat binar pe linii care va menține informația de pe mai multe linii consecutive. Un exemplu este o submatrice de 11×11, care va fi descompusă pe linii ca în Figura 1.5.

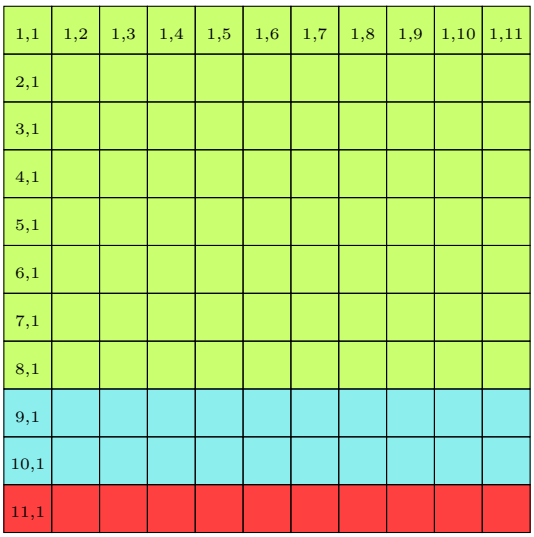

Figura 1.5: Descompunerea pe linii a unui AIB bidimensional

Pentru fiecare interval de linii, ca să putem menține sumele corecte, vom folosi un nou arbore indexat binar `AIB`, în care `AIB[j]` va menține suma celulelor cuprinse între cele două linii și, mai mult, cuprinse între coloanele j și $j - 2^k + 1$, unde k e cel mai nesemnificativ bit al lui j. Descompunerea pe linii, apoi pe coloane, arată ca în Figura 1.6.

Mai mult, deducem că:

$$\text{AIB}[i][j] = \sum_{x=i-k+1}^{i} \sum_{y=j-p+1}^{j} a[x][y], \text{ unde } k = 2^{ub(i)} \text{ și } p = 2^{ub(j)}$$

Implementarea funcțiilor ADD și SUM este prezentată în continuare:

```
#include <bits/stdc++.h>
int aib[nmax + 5][nmax + 5];

int ub(int x)
{
    return (x & (-x));
```

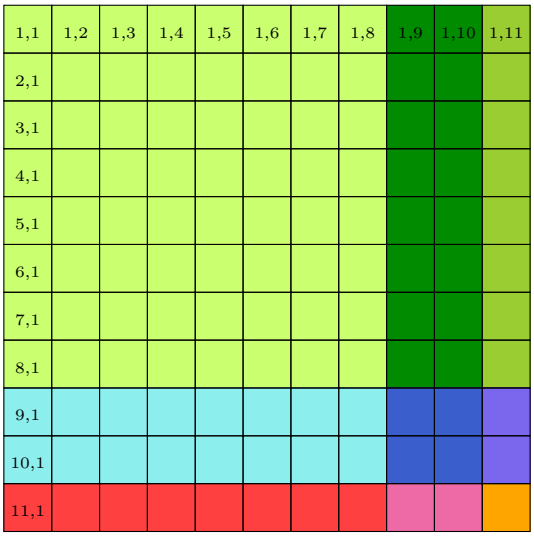

Figura 1.6: Descompunerea pe linii și coloane a unui AIB bidimensional

```
}

void add(int x, int y, int val)
{
    for(int i=x; i<=n; i+=ub(i))
    {
        for(int j=y; j<=n; j+=ub(j))
        {
            aib[i][j] += val;
        }
    }
}

int sum(int x, int y)
{
    int rez = 0;
    for(int i=x; i>=1; i-=ub(i))
    {
        for(int j=y; j>=1; j-=ub(j))
        {
            rez += aib[i][j];
        }
    }
    return rez;
}
```

Bineînțeles, ne putem folosi de aceste funcții pentru a afla suma pe o submatrice oarecare. Mai mult, putem extinde arborii indexați binar pe oricâte dimensiuni.

1.3 Probleme

1.3.1 Problema Inv

Se dă un șir S de lungime n cu numere întregi. Numim o inversiune o pereche de indici (i, j) astfel încât $1 \leq i < j \leq n$ și $S[i] > S[j]$.

Cerință

Să se determine câte inversiuni sunt în șirul dat.

Date de intrare

Fișierul de intrare `inv.in` conține pe prima linie numărul natural n. Pe următoarea linie se găsesc n numere întregi, reprezentând în ordine elementele șirului S.

Date de ieșire

În fișierul de ieșire `inv.out` se va afișa un singur număr, reprezentând restul împărțirii numărului de inversiuni din șir la valoarea 9917.

Restricții

- $2 \leq n \leq 100\,000$

Exemple

inv.in	inv.out
5 3 4 1 2 5	4

Soluție

În primul rând, vom considera vectorul normalizat (fiecare valoare este înlocuită cu poziția ei în vector, dacă acesta ar fi sortat crescător). Se observă că numărul de inversiuni rămâne același. Să considerăm o soluție cu complexitate $\mathcal{O}(n^2)$. Pentru fiecare element $v[i]$, rezultatul va fi mărit cu numărul de elemente $v[j]$, cu $j < i$, care au proprietatea că $v[j] > v[i]$. Fie variabila `rez` rezultatul problemei. Dacă parcurgem vectorul de la stânga la dreapta și reținem un vector de frecvență `fr`, pentru orice element $v[i]$, vom modifica rez astfel:

```
rez += fr[v[i] + 1] + fr[v[i] + 2] + ... + fr[N]
```

Ulterior, vom mări cu 1 valoarea lui `fr[v[i]]`. Putem calcula suma cu ușurință folosind un al doilea `for`.

Observăm că pe vectorul de frecvență `fr` vom aplica două operații:

- mărim valoarea de la poziția $v[i]$ cu 1;
- calculăm suma pe intervalul $[v[i] + 1, N]$.

Dar aceste operații pot fi realizate în complexitate $\mathcal{O}(\log_2 N)$ cu ajutorul unui arbore indexat binar. Prin urmare, vom înlocui vectorul de frecvență cu un arbore indexat binar care va reține frecvențele. Acum, vom modifica rez astfel: `rez += SUM(N) - SUM(v[i])`.

Vom modifica AIB-ul folosind `ADD(v[i], 1)`. Complexitatea soluției este $\mathcal{O}(N \log_2 N)$.

Implementare

```
#include <bits/stdc++.h>
using namespace std;
const int nmax = 1e5;
int n, m;
int aib[nmax + 5];

int ub(int x)
{
  return (x & (-x));
}

void add(int x, int val)
{
  for(int i=x; i<=n; i+=ub(i))
    {
      aib[i] += val;
    }
}
int sum(int x)
{
  int rez = 0;
  for(int i=x; i>=1; i-=ub(i))
    {
      rez += aib[i];
    }
  return rez;
}

int main()
{
  cin>>n>>m;
  for(int i=1; i<=m; i++)
    {
      int tip;
      cin >> tip;
      if(tip == 0)
        {
          /// update
          int x, y, val;
          cin >> x >> y >> val;
          add(x, +val);
          add(y+1, -val);
        }
      else
        {   /// query
          int x;
          cin >> x;
          cout << sum(x) << '\n';
```

```
        }
    }
  return 0;
}
```

1.3.2 Căutare binară pe arbori indexați binar. Problema aib

Se dă un vector A cu N elemente naturale. Asupra lui se vor face M operații, codificate astfel în fișierul de intrare:

- `0 a b` - Valorii elementului de pe poziția a i se va adăuga valoarea b.
- `1 a b` - Să se determine suma valorilor elementelor intervalului $[a, b]$.
- `2 a` - Să se determine poziția minimă k astfel încât suma valorilor primilor k termeni să fie exact a.

Date de intrare

Pe prima linie a fișierului de intrare se află N și M. Pe următoarea linie se găsesc cele N elemente ale vectorului, iar următoarele M linii descriu operația care trebuie efectuată.

Date de ieșire

Pentru fiecare operație de tip 1 se va afișa pe câte o linie suma valorilor elementelor pentru intervalul cerut (în ordinea cerută în fișierul de intrare), iar pentru fiecare operație de tip 2 se va afișa poziția k cerută. Dacă nu există o astfel de poziție se va afișa `-1` pentru operația respectivă.

Restricții

- $1 \leq N \leq 100\,000$
- $1 \leq M \leq 150\,000$
- $1 \leq A_i \leq 10\,000$, pentru orice $1 \leq i \leq N$
- Pentru operația de tip 0: $1 \leq a \leq N$ și $1 \leq b \leq 10\,000$
- Pentru operația de tip 1: $1 \leq a \leq b \leq N$
- Pentru operația de tip 2: $0 \leq a \leq 2^{31}$
- Rezultatul pentru fiecare operație se va încadra pe 32 de biți

Exemple

aib.in	aib.out
8 6 25 42 8 15 1 55 16 67 0 5 12 2 25 2 90 1 7 7 1 2 8 2 241	1 4 16 216 8

Soluție

Pentru primele două tipuri de operații, ne vom folosi de un arbore indexat binar pentru a le realiza în timp $\mathcal{O}(\log_2 N)$. Pentru al treilea tip de operație, o primă idee ar fi să ne folosim de căutarea binară pentru a afla poziția. Fie mij mijlocul intervalului de căutare $[st, dr]$. Dacă suma elementelor de la 1 la mij este mai mică decât a, vom continua căutarea în partea dreaptă, dacă suma este mai mare decât a, vom continua în partea stângă, iar dacă suma este egală cu a, vom opri căutarea. Suma elementelor de la 1 la mij este egală cu SUM(mij).

Complexitatea pentru al treilea tip de operație este de $\mathcal{O}((\log_2 N)^2)$. În continuare, vom studia structura arborilor indexați binar pentru a încerca să găsim o soluție mai rapidă.

Să considerăm un algoritm de căutare binară pe biți, în care încercăm să determinăm poziția-rezultat adăugând pe rând biții acesteia, de la cel mai semnificativ până la cel mai nesemnificativ.

Fie poz poziția curentă, b bitul pe care dorim să îl adăugăm și $s = suma(1, poz)$, unde $suma(a, b)$ este suma elementelor din intervalul (a, b). Suntem interesați să aflăm dacă $suma(1, poz + 2^b)$ este în continuare mai mică decât a. Dar $suma(1, poz + 2^b) = s + suma(poz + 1, poz + 2^b)$.

Observăm că cel mai nesemnificativ bit al lui $poz + 2^b$ este chiar b, deoarece toti biții lui poz sunt mai mari decât b, prin urmare, `AIB[poz + (1<<b)]` va reține chiar $suma(poz + 2^b - 2^b + 1, poz + 2^b)$, adică $suma(poz + 1, poz + 2^b)$.

Prin urmare, $suma(1, poz + 2^b) = s +$ `AIB[poz + (1<<b)]`, deci o putem calcula în $\mathcal{O}(1)$. Dacă $suma(1, poz + 2^b)$ este mai mică decât a, vom mări s cu `AIB[poz + (1<<b)]` și poz cu 2^b. Rezultatul, dacă există, se va afla în $poz + 1$.

Complexitatea acestui algoritm este $\mathcal{O}(\log_2 N)$. Deoarece putem rezolva toate tipurile de operații în $\mathcal{O}(\log_2 N)$, complexitatea totală a soluției este $\mathcal{O}(N \log_2 N)$.

Implementare

```
#include <bits/stdc++.h>
using namespace std;
const int nmax = 1e5;
```

```
int n,m;
int v[nmax + 5];

int aib[nmax + 5];

int ub(int x)
{
  return (x & (-x));
}

void add(int x, int y)
{
  for(int i=x; i<=n; i+=ub(i))
    {
      aib[i] += y;
    }
}

int sum(int x)
{
  int rez = 0;
  for(int i=x; i>=1; i-=ub(i))
    {
      rez += aib[i];
    }
  return rez;
}

int main()
{
  freopen("aib.in","r",stdin);
  freopen("aib.out","w",stdout);
  cin >> n >> m;
  for(int i=1; i<=n; i++)
    {
      cin >> v[i];
      add(i, v[i]);
    }
  for(int i=1; i<=m; i++)
    {
      int tip;
      cin >> tip;
      if(tip == 0)
        {
          int poz, val;
          cin >> poz >> val;
          add(poz,val);
        }
      else if(tip == 1)
        {
          int a, b;
          cin >> a >> b;
          cout<<sum(b) - sum(a - 1)<<'\n';
        }
      else
        {
          int a, s = 0, poz = 0;
```

```
          cin>>a;
          for(int b=17; b>=0; b--)
            {
              if(poz + (1<<b) <= n && s + aib[poz + (1<<b)] < a)
                {
                  s += aib[poz + (1<<b)];
                  poz += (1<<b);
                }
            }
          if(poz + 1 > n || sum(poz + 1) != a)
            {
              cout << -1 << '\n';
            }
          else
            {
              cout<< poz + 1 << '\n';
            }
        }
    }
  return 0;
}
```

1.4 Utilizarea AIB în probleme cu arbori

Se dau un arbore cu N noduri cu rădăcina în nodul 1, fiecare nod având asociată o valoare $v[i]$, și Q operații care pot fi de două tipuri:

- `update x y`: valoarea $v[x]$ a nodului x se mărește cu y
- `query x`: să se determine suma valorilor nodurilor de pe drumul de lungime minimă dintre nodul x și rădăcină.

Soluție

Deși problemele de acest tip necesită în general soluții puțin mai complexe, cu ajutorul algoritmului *Heavy Path Decomposition*, în acest caz cele două tipuri de operații se pot reduce cu ușurință exact la operațiile pe care le suportă un arbore indexat binar: modificarea unei valori și calcularea sumei pe un prefix. Acest lucru se datorează faptului că suntem interesați de suma de pe drumul dintre un nod și rădăcină, nu de suma de pe drumul dintre două noduri arbitrare.

În primul rând, vom considera parcurgerea Euler a arborelui (în care adăugăm un nod în listă atunci când algoritmul DFS ajunge pentru prima dată în el și atunci când algoritmul DFS se întoarce în nodul tată). Fie *poz* prima apariție a unui nod x în parcurgere. Pe prefixul $(1, poz)$ al listei, fiecare nod apare:

- de 0 ori, dacă până in acest moment algoritmul DFS nu a ajuns în acest nod;
- de 2 ori, dacă algoritmul DFS a ajuns în nod și a revenit în nodul părinte;
- o singură dată, dacă algoritmul DFS a ajuns în nod, dar nu a revenit încă în tată.

Observația care ne va ajuta să rezolvăm problema este că nodurile aflate în ultima situație sunt exact nodurile de pe drumul dintre x și rădăcină.

Pentru fiecare nod x din lista parcurgerii Euler, fie p_1 și p_2 prima și, respectiv, a doua apariție a nodului în listă. Vom reține un vector auxiliar l, în care $l[p_1] = v[x]$ și $l[p_2] = -v[x]$. Datorită proprietății de mai sus, suma $l[1] + l[2] + \cdots + l[p_1]$ va fi egală cu suma valorilor de pe drumul dintre x și rădăcină, deoarece toate celelalte valori se anulează.

Prin urmare, cele doua tipuri de operații din enunțul problemei se reduc la:

- `update x y`: $l[p_1] += y$ și $l[p_2] += -y$;
- `query x`: $l[1] + l[2] + \cdots + l[p_1]$.

Putem rezolva cu ușurință atât update-urile, cât și query-urile, folosind un arbore indexat binar. Fiecare operație va avea complexitate $\mathcal{O}(\log_2 N)$, iar complexitatea totală a soluției este $\mathcal{O}(N \log_2 N)$.

Implementare

```
#include <bits/stdc++.h>
using namespace std;
const int nmax = 1e5;

int n,q;
int v[nmax + 5];

vector<int> G[nmax + 5];

int p[nmax + 5], u[nmax + 5];

int aib[2 * nmax + 5];

int cnt = 0;

int ub(int x)
{
  return (x & (-x));
}

void add(int x, int y)
{
  for(int i=x; i<=2*n; i+=ub(i))
    {
      aib[i] += y;
    }
}

int sum(int x)
{
  int rez = 0;
  for(int i=x; i>=1; i-=ub(i))
    {
      rez += aib[i];
    }
  return rez;
```

```
}

void euler(int nod, int dad = 0)
{
  p[nod] = (++cnt);
  for(auto it : G[nod])
    {
      if(it==dad)
        {
          continue;
        }
      euler(it, nod);
    }
  u[nod] = (++cnt);
}

int main()
{
  cin>>n>>q;
  for(int i=1; i<n; i++)
    {
      int x,y;
      cin>> x >> y;
      G[x].push_back(y);
      G[y].push_back(x);
    }
  euler(1);
  for(int i=1;i<=n;i++)
    {
      cin>>v[i];
      add(p[i],v[i]);
      add(u[i],-v[i]);
    }
  for(int i=1; i<=q; i++)
    {
      int tip;
      cin >> tip;
      if(tip == 0)
        {
          /// update
          int nod, val;
          cin >>nod >> val;
          add(p[nod], val);
          add(u[nod], -val);
        }
      else if(tip == 1)
        {
          /// query
          int nod;
          cin >> nod;
          cout << sum(p[nod]) << '\n';
        }
    }
  return 0;
}
```

1.5 Probleme propuse

- Problema Aib
- Problema Aib2d
- Problema Arbori indexați binar
- Problema Inv
- Problema Swap
- Problema Determinanta
- Problema Permsort
- Problema Costperm
- Problema Evantai

1.6 Bibliografie

[1] Marius Nicoli, *Arbori indexați binar*, URL: https://cppi.sync.ro/materia/arbori_indexati_binar.html.

[2] Mihnea Giurgea, *Arbori indexați binar*, URL: https://www.infoarena.ro/aib.

[3] *Understanding Fenwick Trees / Binary Indexed Trees*, URL: https://codeforces.com/blog/entry/57292.

Capitolul 2

Arbori de intervale

prof. Marius Nicoli
Colegiul Național „Frații Buzești” Craiova
Centrul de Pregătire pentru Performanță în Informatică Craiova

Arborii de intervale sunt o structură de date (o modalitate de a organiza informații cu scopul de a realiza mai rapid un anumit tip de calcule).

2.1 Problema inițială

Pornim de la următorul enunț:

Se dă un tablou unidimensional (vector) V cu n numere naturale, asupra căruia se pot face două tipuri de operații:

- interogare pentru minim pe o secvență de elemente;
- actualizarea unui element (schimbarea valorii sale).

Să considerăm că vectorul poate avea poate avea cel mult 100 000 de elemente și că sunt cel mult 100 000 de interogări. De asemenea, valorile sale sunt de tip `int`.

În această problemă actualizările și interogările nu sunt separate, ele pot apărea amestecat și la fiecare interogare răspundem considerând configurația vectorului din acel moment.

Exemple

Intrare	Ieșire
11 1 6 4 3 9 2 7 8 0 5 4 3 1 2 4 2 3 2 1 2 4	3 2

Semnificația datelor de intrare de mai sus: vectorul are 11 elemente, acestea sunt inițial 1 6 4 3 9 2 7 8 0 5 4 (să considerăm indexarea de la 1). Sunt 3 operații. Mai întâi o interogare (simbolizată prin prezența valorii 1 prima pe linia ce o descrie), care ne cere să aflăm minimul din intervalul de indici de la 2 la 4. Acesta are valoarea 3. A doua operație este o actualizare (linia începe cu 2). Adică elementul de pe poziția 3 devine 2. Efectul este că vectorul va fi: 1 6 2 3 9. A treia operație este iarăși o interogare pe intervalul de indici de la 2 la 4. Acum minimul este 2. Așadar ieșirea acestui program ar trebui să fie 3 2.

Rezolvarea în mod brut, adică simularea a ceea ce este descris în enunț, presupune actualizarea în mod direct (operația de tip `2 p x` se rezolvă printr-o singură atribuire, $v[p] \leftarrow x$). În schimb, operația de tip interogare de minim pe intervalul de indici $[a, b]$, simbolizată prin `1 a b` presupune realizarea unei parcurgeri de la a la b, iar acesta, pe cazul cel mai defavorabil tinde să aibă spre n pași. Această metodă face actualizarea în timp constant iar interogarea în timp liniar. Așadar, dacă numărul de interogări este mare, iar acestea sunt mereu pe intervale mari, timpul de calcul ajunge să fie $\mathcal{O}(n \cdot m)$.

Dacă interogările ar fi toate la final, fără ca printre ele să mai apară actualizări, se poate folosi structura de date numită RMQ (*Range Minimum Query*) pentru a face precalculări de minime pe anumite intervale și apoi se folosesc aceste minime pentru a răspunde în timp constant la o anumită interogare. Precalcularea elementelor structurii necesită timp și memorie de ordin $\mathcal{O}(n \log_2 n)$. Din păcate actualizarea sa la modificarea valorii unui element nu se mai face în timp logaritmic și astfel folosirea pentru cazul în care avem actualizări și interogări amestecate devine neeficientă.

Să revenim la problema enunțată inițial. Anunțăm de pe acum ce vom obține la final, urmând să arătăm apoi pașii:

- vom calcula (și stoca) minime din anumite intervale ale vectorului V (intervale stabilite clar de la început);
- intervalele pentru care avem minimele calculate vor fi suficiente ca pe baza lor să determinăm minimul din oricare alt interval. Mai mult, pentru un interval oarecare dat, numărul de intervale necesare, dintre cele cu rezultatul stocat, este de ordin $\log_2 n$;
- la actualizarea unui element din vectorul V numărul de intervale care trebuie actualizate este de ordin $\log_2 n$.

Astfel, vom obține timp de ordin logaritmic pe fiecare dintre dele două operații.

Mai întâi să vedem cum alegem setul de intervale fixat, pentru care calculăm minimele. În primul rând observăm că numărul total de intervale este de ordin n^2 (ne imaginăm că împerechem fiecare număr de la 1 la n, reprezentând capătul stâng al unui interval cu fiecare număr mai mare decât el, reprezentând capătul drept).

Pentru datele noastre, este nepractic să calculăm informații pentru toate aceste intervale (atât ca timp dar și ca memorie avem n^2, iar pentru $n = 100\,000$ asta înseamnă foarte mult).

Pentru a găsi modalitatea de obținere a setului de intervale care ne interesează pornim de la următoarea funcție recursivă:

```
void rec(int st, int dr) {
  if (st == dr) {
    /// operație directă pe intervalul format
    /// cu un singur element (v[st])
  } else {
    /// intervalul de indici de la st la dr este
    /// împărțit în două intervale, folosind
    /// un indice de la mijloc, iar apoi facem
    /// autoapel în cele două intervale

    int mid = (st+dr)/2;
    rec(st, mid);
    rec(mid+1, dr);
  }
}
```

Apel inițial `rec(1, n)`.

Pentru codul de mai sus avem:

- n = numărul de elemente ale vectorului dat (indexat, cum spuneam, de la 1);
- parametrii funcției recursive au semnificația de indici din vectorul dat.

De exemplu, pentru $n = 11$, apelul de mai sus generează următoarele perechi de indici (st, dr): (1, 11), (1, 6), (1, 3), (1, 2), (1, 1), (2, 2), (3, 3), (4, 6), (4, 5), (4, 4), (5, 5), (6, 6), (7, 11), (7, 9), (7, 8), (7, 7), (8, 8), (9, 9), (10, 11), (10, 10), (11, 11)

Citindu-le pur și simplu ne este greu să găsim repede o semnificație a acestor numere, dar să vedem mai departe câteva moduri de a le imagina:

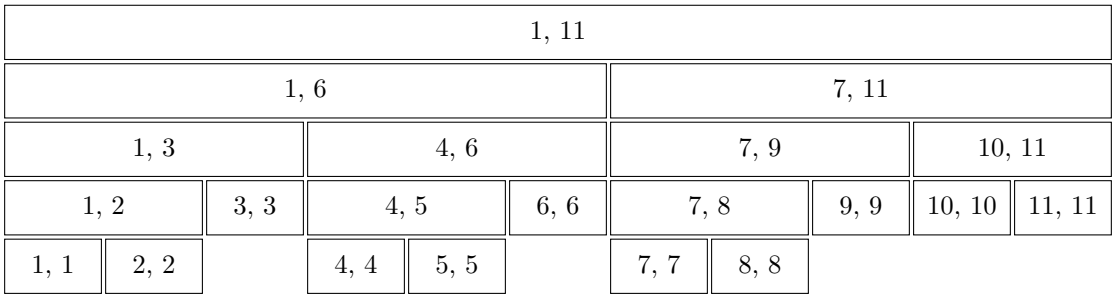

Figura 2.1: Descompunerea în intervale a apelului `rec(1, n)`.

Pentru $n = 11$ acestea vor fi intervalele pentru care vom ține în orice moment minimele (adică, după fiecare operație vom avea stocată valoarea minimă din fiecare dintre aceste intervale). Va trebui să determinăm câte astfel de intervale sunt, cum actualizăm aceste intervale în timp logaritmic și cum putem determina minimul din oricare alt interval folosind doar dintre acestea, în număr de operații de ordin logaritmic. Bineînțeles, exemplul nostru este pentru $n = 11$, dar descompunerea este valabilă în general, folosind template-ul de funcție recursivă de mai sus, împreună cu apelul inițial `rec(1, n)`.

O primă observație este că numărul de linii ale tabelului de mai sus este de ordin $\log_2 n$. Justificare: Dacă intervalele de pe un rând au lungimea L, cele de pe rândul următor au lungimea $L/2$ sau $L/2+1$ (am semnificat prin „/” câtul împărțirii întregi). Dar numărul

de înjumătățiri după care se ajunge la intervale de lungime 1 (celule de jos, care nu mai pot fi împărțite) este de ordin logaritmic.

Să vedem cum păstrăm informații despre aceste intervale. Pentru aceasta, vom mai introduce un parametru la funcția `rec`. Îl notăm cu `nod` și alegem să îl scriem primul (poziția lui între parametrii funcției recursive nu este importantă, însă încercăm să păstrăm o ordine întâlnită în multe materiale despre arborii de intervale). Avem așadar:

```
void rec(int nod, int st, int dr) {
  if (st == dr) {
    /// operație directă pe intervalul format
    /// cu un singur element (v[st])
  } else {
    /// intervalul de indici de la st la dr este împărțit
    /// în două intervale, folosind un indice de la mijloc,
    /// iar apoi facem autoapel în acestea

    int mid = (st+dr)/2;
    rec(2*nod, st, mid);
    rec(2*nod+1, mid+1, dr);
  }
}
```

Apel inițial: `rec(1, 1, n)`.

Urmăriți modul în în care realizăm autoapelurile.

Mai departe să vedem cum arată celulele din tabel, cu noul parametru.

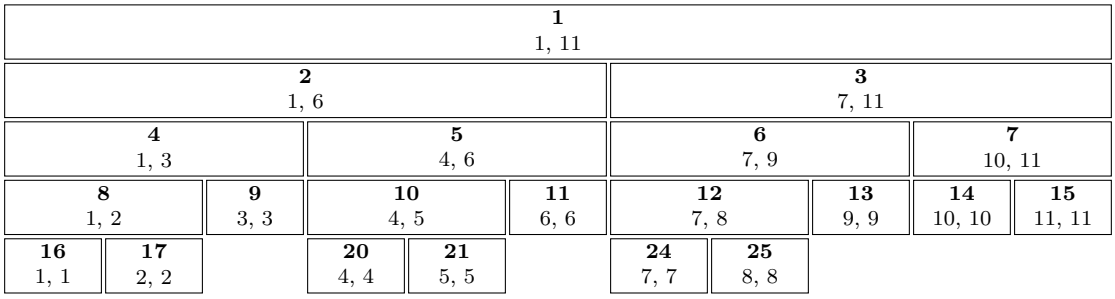

Figura 2.2: Descompunerea în intervale a apelului `rec(1, 1, n)`.

Valorile lui *nod* sunt figurate deasupra, boldat. Așadar, în fiecare celulă avem un triplet (nod, st, dr) unde $[st, dr]$ reprezintă un interval de indici din vectorul V. Dar ce semnificație poate avea *nod*?

Citind de sus în jos și de la stânga la dreapta, valorile sale sunt consecutive începând cu 1, excepție făcând câteva aflate mai la final, care lipsesc.

Acum să estimăm numărul de intervale (celule) care apar în tabel. Pentru asta, să considerăm că n este o putere de 2. Pentru simplitate luăm $n = 8$. Tabelul nostru va arăta ca în Figura 2.3.

Acum se observă ușor:

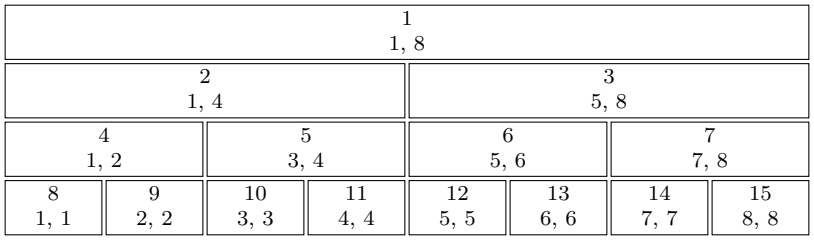

Figura 2.3: Descompunerea în intervale pentru $n = 8$.

- valorile *nod* sunt toate numerele de la 1 la $2n - 1$ (deci tot acesta este numărul de intervale care ne interesează);
- înălțimea este $1 + \log_2 n$;
- dacă n nu ar fi putere de 2, ne dăm seama ușor că tabelul nu este mai înalt decât pentru cazul că am majora pe n la următoarea putere de 2, iar numărul său de elemente este tot $2n - 1$.

În general, la tehnica noastră numărul de intervale care contează este $2n - 1$. Valoarea *nod* o vom folosi ca indice în alt vector, pe care îl vom nota cu A și care este de fapt structura noastră de date.

Astfel, pentru un triplet (nod, st, dr) generat de funcția de mai sus, avem: $A[nod] =$ minimul elementelor din V aflate pe indici între st și dr inclusiv. Putem deci spune că la un triplet (nod, st, dr), nod este indice din A, iar st și dr sunt indici din V.

Din cele analizate mai sus deducem că sunt importante $2n - 1$ elemente din A, însă observăm că ele nu sunt plasate chiar pe toți indicii consecutiv de la 1 la $2n - 1$ indiferent de valoarea lui n.

Gândindu-ne că la trecerea unui n peste o putere de 2 se merge până la a se dubla numărul de elemente (la următoarea putere de 2 fiecare frunză se expandează în alte două) deducem că $4n$ elemente pentru vectorul A este o limitare superioară suficientă. Nu face obiectul acestui articol, dar precizăm că există formule de calcul pentru indicii din A, altele decât cea folosită de noi (`mid = (st+dr)/2`) care permit reducerea semnificativă a memoriei.

Date fiind cele de mai sus, să trecem către implementare. Primul pas ar fi să calculăm valorile inițiale din A. Adică minimele, conform elementelor date inițial în V, înainte de update-uri. Pentru asta, vom folosi următoarea funcție recursivă (o variantă a funcției `rec` pe care o vom numi `build`).

```
void build(int nod, int st, int dr) {
  if (st == dr) {
    A[nod] = V[st];
  } else {
    int mid = (st+dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}
```

Apel: `build(1, 1, n)`;

Este un principiu **divide et impera** pe care îl putem interpreta așa:

- Dacă intervalul are lungimea 1, valoarea din el este chiar cea din V de pe acea poziție.
- Altfel, cei doi fii sunt de fapt cele două jumătăți ale sale și valoarea din interval este minimul valorilor din cele două jumătăți.

Iată cum arată vectorul A dar și tabelul de mai sus în care acum completăm cu valorile calculate în noduri de apelul `build(1, 1, n)`.

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$V[x]$	1	6	4	3	9	2	7	8	0	5	4														
$A[x]$	0	1	0	1	2	0	4	1	4	3	2	7	0	5	4	1	6	X	X	3	9	X	X	7	8

Tabela 2.1: Valorile din V și A pentru datele din exemplu.

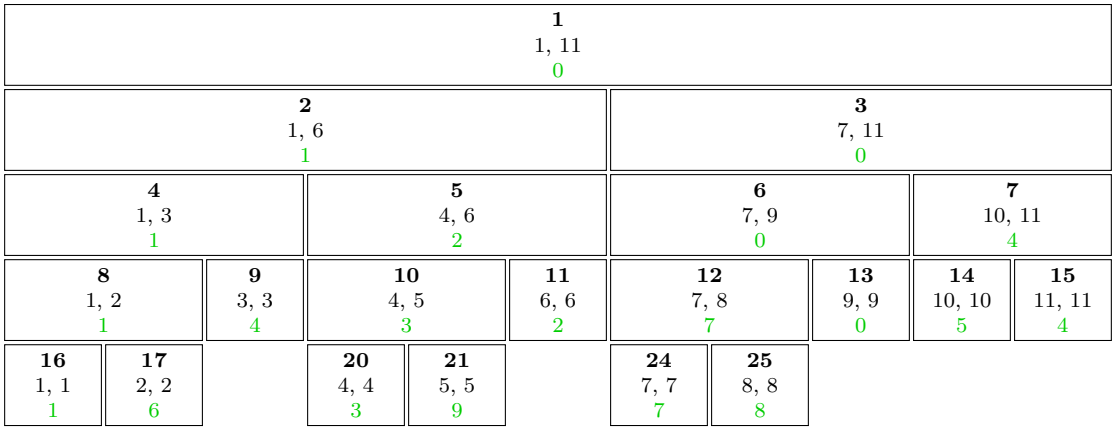

Figura 2.4: Valorile lui A după apelul `build(1, 1, n)`.

De remarcat că timpul de calcul pentru rularea acestui cod, `build(1, n)` este de ordin n (sunt $2n-1$ valori care se calculează). Asta ne poate duce puțin în eroare gândindu-ne că noi urmărim să obținem cod care rulează în timp logaritmic. Nu este nicio problemă întrucât apelul acesta se face o singură dată, nu la fiecare operație, iar el are aceeași complexitate în timp ca și citirea datelor.

Pornind de la observația că nodurile frunză (cele care corespund unor intervale de lungime 1) se parcurg de la stânga la dreapta (acest lucru se întâmplă pentru că noi facem mai întâi autoapel în subintervalul stâng), și că valorile nodurilor se completează de jos în sus (deoarece instrucțiunea `A[nod] = min(A[2*nod], A[2*nod+1])` se află după autoapeluri), putem face citirea datelor de intrare chiar în funcție, pe ramura fără autoapel, adică:

```
void build(int nod, int st, int dr) {
  if (st == dr) {
    fin>>A[nod];
  } else {
    int mid = (st+dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);
```

```
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}
```

Am prezentat această variantă pentru a ajuta la înțelegerea mai bună a modului de funcționare, dar nu o recomandăm, considerăm că este mai important să facem lucrurile separat, și pentru a le avea mai clare.

Pentru că tot am vorbit de frunze, noduri, dar și ținând cont și de la denumirea structurii (arbore de intervale), ne putem imagina și o reprezentare a datelor printr-un arbore binar (în care un nod este un interval iar cei doi fii ai săi sunt cele două jumătăți care îl compun). Astfel, fiecare celulă de tabel este un nod iar cei doi fii sunt cele două celule de sub ea, corespunzând celor două jumătăți în care se descompune intervalul pe care ea îl reprezintă.

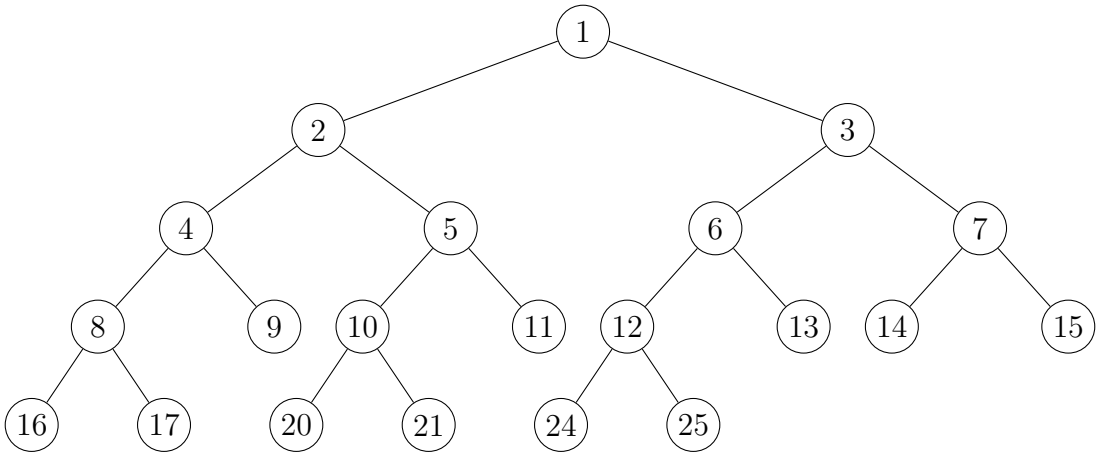

Figura 2.5: Echivalența între un arbore de intervale și un arbore convențional.

2.1.1 Operația de actualizare a unui element

Noi am codificat-o prin `2 p x`, cu semnificația: elementul de pe poziția p din vectorul dat devine x. Să vedem ce modificări implică această operație în vectorul A

- se va modifica valoarea frunzei corespunzătoare intervalului de lungime 1 $[p, p]$;
- pentru tot drumul de la această frunză până la rădăcină se vor calcula valorile din celula curentă în funcție de cele din cele două celule fiu (cele două jumătăți ale intervalului reprezentat de celula curentă) - care la revenire sunt deja calculate.

Figura 2.6 ilustrează cele explicate aici considerând exemplul inițial și actualizarea $V[8] \leftarrow 5$ (adică valoarea de pe poziția 8, în loc de 8, va deveni 5).

Celulele desenate cu background roșu sunt cele în care se recalculează valoarea (pentru unele ea se poate modifica iar pentru altele nu).

Pentru a realiza acestea, facem o altă funcție recursivă (o vom numi `update`), în care avem ca parametri, pe de o parte pe cei 3 care identifică tripletele (nod, st, dr) și încă doi specifici operației: p și x.

Reamintim că este esențial să generăm la toate funcțiile care operează pe arbore, aceleși triplete.

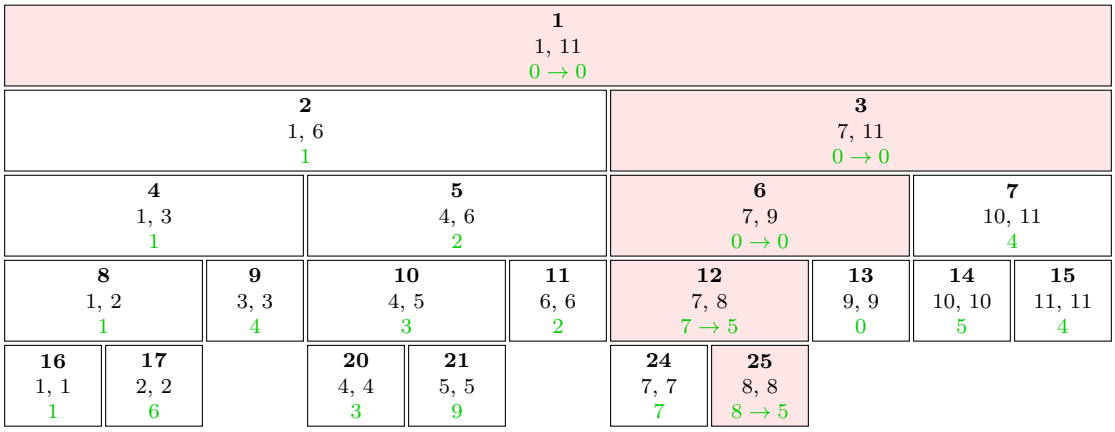

Figura 2.6: Actualizarea lui A după modificarea $V[8] \leftarrow 5$.

```
void update(int nod, int st, int dr, int p, int x) {
  if (st == dr) {
    A[nod] = x;
  } else {
    int mid = (st+dr)/2;
    if (p <= mid)
      update(2*nod, st, mid, p, x);
    if (p > mid)
      update(2*nod+1, mid+1, dr, p, x);
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}
```

Apel: `update(1, 1, n, p, x)`.

Câteva observații legate de codul funcției de mai sus:

- din nodul curent avansăm exact în unul dintre cei doi fii (cel care va conține poziția p);
- întrucât înălțimea arborelui este de ordin logaritmic iar noi coborâm un nivel la fiecare autoapel, ajungem la complexitatea dorită;
- cele două `if`-uri puteau fi scrise mai compact printr-un `if-else`, însă am preferat această variantă pentru a introduce un standard ce va fi folosit și la alte operații pe arbore; odată stăpânit modul în care funcționează structura, este la alegerea celui care codează să scrie cum i se pare mai potrivit;
- p și x se transmit cu aceeași valoare către autoapeluri dar am ales să îi punem parametri pentru claritate; dacă dorim, îi putem folosi ca variabile globale.

2.2 Operația de interogare

O codificăm așadar prin `1 a b` cu semnificația dată (să aflăm minimul dintre valorile: $V[a], V[a+1], \ldots, V[b]$).

De această dată vom scrie întâi codul care realizează interogarea și vom veni cu explicații asupra lui.

```
void query(int nod, int st, int dr, int a, int b) {
  if (a<=st && dr<=b) {
    sol = min(sol, A[nod]);
  } else {
    int mid = (st+dr)/2;
    if (a <= mid)
      query(2*nod, st, mid, a, b);
    if (b >= mid+1)
      query(2*nod+1, mid+1, dr, a, b);
  }
}

sol = -INF;
query(1, 1, n, a, b);
fout<<sol;
```

O explicație sumară a acestui cod ar fi următoarea: dacă pentru intervalul curent $[st, dr]$, intervalul de interogare $[a, b]$ are elemente și într-o jumătate și în alta, facem autoapel în amândouă jumătățile, în caz contrar facem apel în jumătatea în care el conține elemente. Când ajungem la un interval $[st, dr]$ complet inclus în $[a, b]$, comparăm cu minimul global (păstrat de noi în variabila `sol`) valoarea $A[nod]$ (corespunzătoare lui $[st, dr]$).

Această explicație sumară nu justifică nici eficiența (de exemplu, ne punem întrebarea, „dacă facem apel în ambii fii, nu putem ajunge la timp de ordin n, precum la `build`?") și nici nu ajută la înțelegerea temeinică a modului în care se face descompunerea (iar această bună înțelegere este esențială și pentru priceperea modului de funcționare a structurii atunci când avem actualizare pe interval de indici - să nu uităm că aici noi am făcut explicație doar pentru update pe un singur element).

Așadar, să ne concentrăm pe modul în care funcționează funcția `query`, scrisă mai sus.

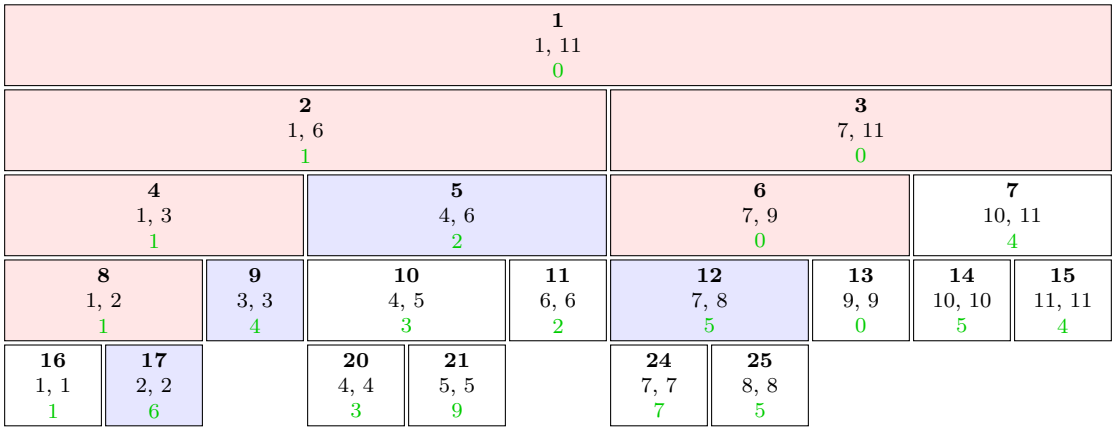

Figura 2.7: Interogarea arborelui de intervale pentru minimul pe intervalul [2, 8].

În Figura 2.7 sunt evidențiate (cu alt background decât alb) nodurile prin care se trece la un apel pentru intervalul $[a, b] = [2, 8]$.

Nodurile figurate cu roșu au intersecție nevidă cu intervalul $[a, b]$ însă nu sunt complet incluse în el, așa că pentru ele nu se răspunde cu informația $A[nod]$ ci se autoapelează în fii. Din unele dintre ele se fac două autoapeluri iar din altele doar unul, depinde de care dintre jumătăți au intersecție nevidă cu $[a, b]$.

De exemplu, când pornim din nodul de sus, cu apelul inițial, adică $[st, dr] = [1, 11]$, se fac autoapeluri în ambii fii, iar când ajungem în [7, 11], se mai face autoapel doar în fiul stâng, [7, 9], pentru că doar el are intersecție nevidă cu intervalul $[a, b]$.

Cu albastru am figurat intervalele complet incluse în $[a, b]$. Deci, când se ajunge cu autoapel în ele se returnează valoarea din A corespunzătoare fără să se mai facă autoapeluri. Se observă pe figură că, dacă pe acestea albastre le concatenăm de la stânga la dreapta, obținem chiar intervalul $[a, b]$.

Acum ne punem problema: cât de multe pot fi intervalele prin care trece un autoapel (adică în total cele roșii și cele albastre)? De numărul lor depinde timpul de executare, iar noi ne dorim ca el să fie de ordin logaritmic.

Din această figură pare că numărul de intervale implicate este mare raportat la numărul total de intervale. Dar asta pentru că avem un n mic iar intervalele aflate mai sus sunt desenate „mai late". Dar să vedem o figură mai în ansamblu.

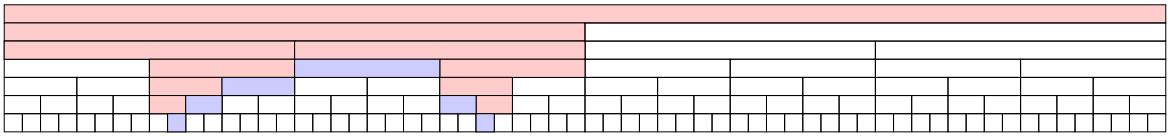

Figura 2.8: Descompunerea în intervale pentru $n = 8$.

Este esențial de observat că sub intervalele colorate cu albastru nu mai avem culori. Semnificația este că nu se mai ajunge cu autoapelul în intervale de sub acestea. **Practic, pentru fiecare valoare din intervalul de query se ajunge doar până la cel mai înalt nod din AINT care o conține și care este complet inclus în intervalul de query** $[a, b]$. Reținem în mod special această observație, esențială în înțelegerea situației cu update pe interval, despre care am mai amintit și pe care o vom explica ulterior (tehnica numită „**lazy update**").

Observăm că este o singură linie (nivel) pe care se află două dreptunghiuri roșii. Celelalte linii pe care sunt două dreptunghiuri colorate au proprietatea că unul dintre ele este colorat cu albastru, adică cel dinspre mijloc. Ce se întâmplă de fapt ? Odată făcută ramificarea pe un nivel, pe nivelele de mai jos în care se mai fac două autoapeluri, se întâmplă că unul dintre cele două (cel dinspre mijloc, adică din dreapta dacă am mers la ramificare pe ramura din stânga, și cel din stânga, dacă am mers la ramificare pe ramura din dreapta) reprezintă un interval al AINT-ului complet inclus în $[a, b]$, deci din el se va răspunde direct cu valoarea ($A[nod]$) fără a se mai face autoapeluri.

Dacă am figura mai contractat nodurile, am observa și mai bine ce se întâmplă de fapt.

Nu e obligatoriu ca ramificarea să se facă de pe primul nivel, dar de acolo unde se face, se va merge în jos pe două ramuri, iar pe fiecare dintre acestea va fi cel mult un nod în care se mai fac două autoapeluri și în plus, în fiecare astfel de caz, unul dintre autoapeluri returnează direct valoarea.

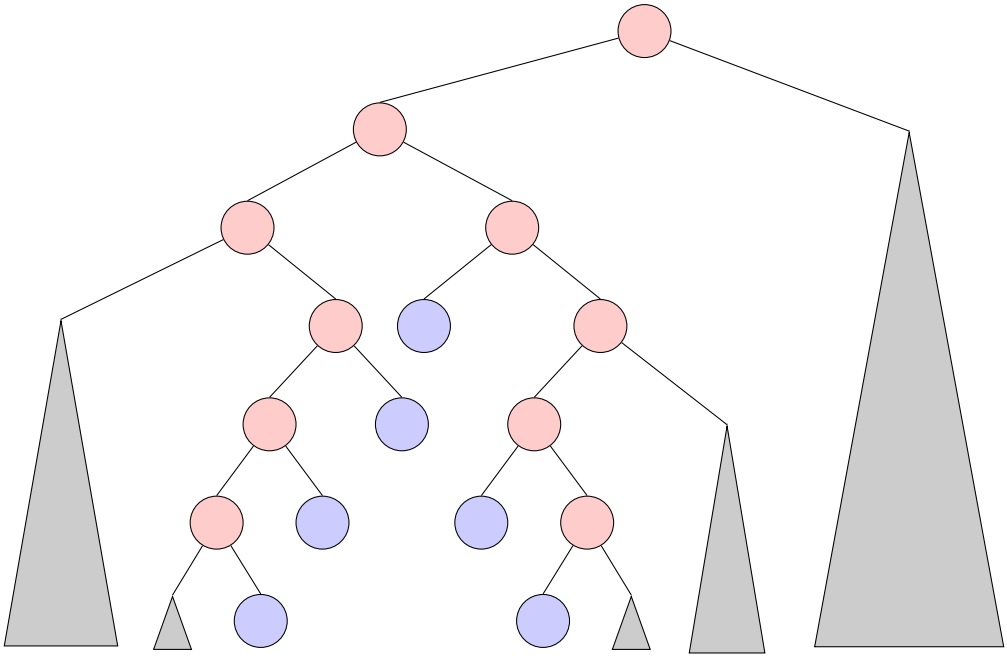

Figura 2.9: Pe fiecare nivel sunt vizitate cel mult patru noduri.

Astfel, de pe fiecare nivel se pot vizita maxim 4 noduri. Pe de altă parte, numărul de niveluri este maxim $1 + \log_2 n$. Obținem timp de calcul logaritmic.

Prezentăm mai jos o sursă care rezolvă problema enunțată la început. Un enunț complet al acesteia se află pe pbinfo.

```
#include <fstream>
#include <climits>
#define DIM 100010
#define INF INT_MAX

using namespace std;

ifstream fin ("aemi.in");
ofstream fout("aemi.out");

int A[4*DIM];
int n, m, a, b, t, i, minim;

void build(int nod, int st, int dr) {
  if (st == dr) {
    fin>>A[nod];
  } else {
    int mid = (st + dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}

void query(int nod, int st, int dr, int a, int b) {
  if (a <= st && dr <= b) {
    minim = min(minim, A[nod]);
```

```
  } else {
    int mid = (st + dr)/2;
    if (a <= mid)
      query(2*nod, st, mid, a, b);
    if (b > mid)
      query(2*nod+1, mid+1, dr, a, b);
  }
}

void update(int nod, int st, int dr, int a, int b) {
  if (st == dr) {
    A[nod] = b;
  } else {
    int mid = (st + dr)/2;
    if (a <= mid)
      update(2*nod, st, mid, a, b);
    if (a > mid)
      update(2*nod+1, mid+1, dr, a, b);
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}

int main () {

  fin>>n;
  build(1, 1, n);
  fin>>m;
  for (int i=1;i<=m;i++) {
    fin>>t>>a>>b;
    if (t == 1) {
      minim = INF;
      query(1, 1, n, a, b);
      fout<<minim<<"\n";
    } else {
      update(1, 1, n, a, b);
    }
  }
  return 0;
}
```

2.3 Folosirea arborilor de intervale atunci când avem update pe un interval cu mai mult de un element (tehnica „lazy update”)

La problema anterioară, chiar dacă interogarea se poate face pe un interval, operația de actualizare a fost dată pentru un singur element. Acum rezolvăm problema în care operația de actualizare este de forma: `2 a b x`, cu semnificația: „toate elementele aflate în V pe poziții de la a la b inclusiv primesc valoarea x”.

Pornind de la ce avem deja, am putea folosi funcția `update` (scrisă mai sus) pentru fiecare element din intervalul $[a, b]$.

```
for (i=a;i<=b;i++)
  update(1, 1, n, i, x);
```

Avem astfel $\mathcal{O}(n \log_2 n)$ pe update, dar noi am promis $\mathcal{O}(\log_2 n)$.

Altă variantă ar fi ca la funcția `update` să facem cam ca la `query` și să mergem până la frunze.

```
void update(int nod, int st, int dr, int a, int b, int x) {
  if (st == dr) {
    A[nod] = x;
  } else {
    int mid = (st+dr)/2;
    if (a <= mid)
      update(2*nod, st, mid, a, b, x);
    if (b >= mid+1)
      update(2*nod+1, mid+1, dr, a, b, x);
    A[nod] = min(A[2*nod], A[2*nod+1]);
  }
}
```

Practic avansăm peste tot până ajungem la frunze, având grijă să reconstituim nodurile interioare, adică după autoapeluri. Din păcate, ajungem să actualizăm fiecare frunză individual, iar numărul acestora este de ordin n.

Soluția optimă se obține pornind de la cea de mai sus plus câteva observații pentru a nu mai ajunge la frunze. Acest lucru se obține oprindu-ne mereu cu actualizarea la intervalele complet incluse în $[a, b]$, adică la cele figurate cu albastru pe desenele anterioare, la explicațiile despre operația de interogare.

Imediat apare însă întrebarea: dar ce se întâmplă dacă după un astfel de update apare un query (sau un alt update) strict în interiorul unui astfel de interval pentru care nu am mers și în fii?

Mai precis, dacă pe exemplul inițial facem o operație de tipul `2 3 8 4` (adică toate valorile din intervalul de indici [3, 8] devin 4), arborele nostru ar arăta:

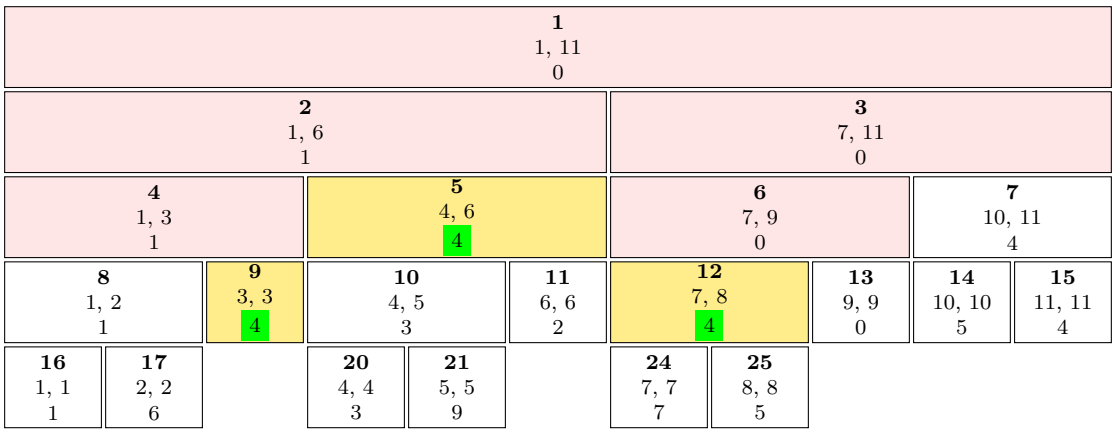

Figura 2.10: Actualizarea $V[3 \ldots 8] \leftarrow 4$.

Adică valorile setate pentru intervalele în care se descompune $[a, b]$ sunt actualizate la 4, însă cele de sub ele rămân cu valorile anterioare. Acest lucru ar face ca un query pentru poziția 5, spre exemplu, să dea valoare greșită, 9 (acest query ar ajunge chiar în acea frunză pentru care valoarea acum nu mai este corectă).

Pentru a rezolva problema, vom reține suplimentar în fiecare nod al arborelui de intervale un *flag* (valoare care poate fi 0 sau 1) iar când acesta este marcat la 1 semnifică faptul că s-a făcut în intervalul reprezentat de acel nod o actualizare care nu a mai fost trimisă fiilor. În momentul în care se trece în jos printr-un nod, vom verifica valoarea flagului și în cazul în care el este 1, înainte să facem autoapel în unul sau în ambii fii ai nodului curent, vom transmite valoarea din nod în ambii săi fii și vom seta la 1 flagul din aceștia, făcând totodată 0 flagul din nodul prin care am trecut.

Aceste operații se execută în timp constant, deci nu afectează complexitatea, iar prezența flagului anunță că valorile din nodurile de sub un nod cu flagul 1 nu sunt neapărat reale, dar ele pot fi actualizate când este nevoie de ele, întrucât orice apel pornește de sus în jos și poate afla valoarea flag-ului pentru nodurile prin care trece.

Arborele nostru, după update-ul de mai sus, pentru care figurăm în fiecare nod și minimul și flagul, arată astfel:

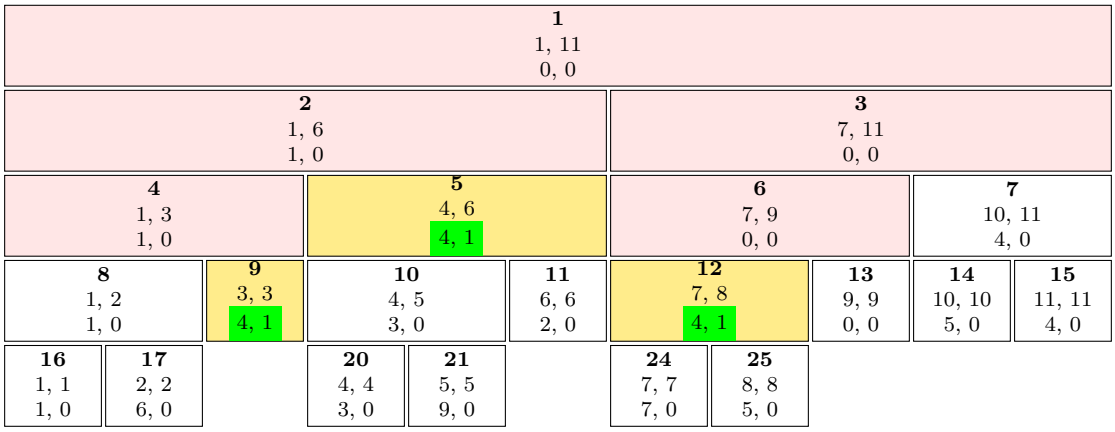

Figura 2.11: Actualizarea $V[3 \ldots 8] \leftarrow 4$ cu informația *lazy*.

Să presupunem că avem mai departe un query pentru valoarea din nodul 6 (adică de forma `1 6 6`). Un apel al funcției query are nevoie de nodul 11 din A, care nu are valoarea corectă însă pentru a ajunge cu autoapelul la el se va trece prin tatăl său, nodul de pe poziția 5, care are flagul setat. Chiar dacă este necesar să se avanseze doar spre fiul său drept, la trecerea prin nodul 5 se va seta valoarea și flagul către ambii fii ai săi și se va deseta valoarea sa. Astfel rezultă arborele din figura 2.12.

Pe scurt, artificiul nostru se realizează în următorii pași:

- se operează înainte de autoapeluri;
- **atât la query cât și la update**, dacă se întâlnește un interval cu flagul 1 pentru care sunt necesare autoapeluri:
  - se setează la 0 flagul;
  - se trimite la fii valoarea sa;

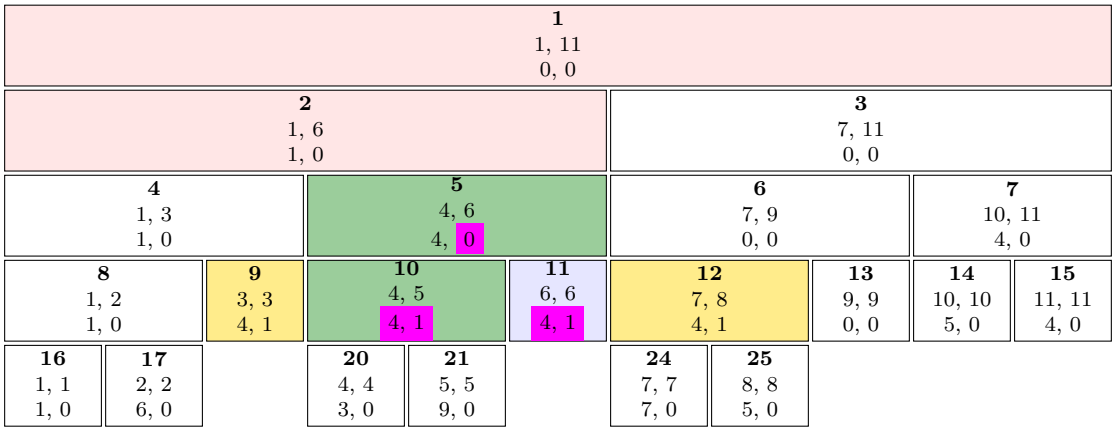

Figura 2.12: Interogarea $V[6]$ și propagarea informației *lazy*.

  - se setează la 1 flagul fiilor;

- la update, dacă se întâlnește un interval $[st, dr]$ complet inclus în $[a, b]$:
  - se actualizează valoarea nodului corespunzător;
  - se setează flagul la 1;
  - nu se mai face autoapel în fii.

Prezentăm mai jos rezolvarea problemei (actualizare pe interval și interogare de minim pe interval). Enunțul complet se găsește pe pbinfo.

```
#include <fstream>
#include <climits>
#define DIM 100010
#define INF INT_MAX

using namespace std;

ifstream fin ("aimi.in");
ofstream fout("aimi.out");

struct nod {
  int value;
  int flag;
};


nod A[4*DIM];
int n, m, a, b, t, i, minim, x;

void build(int nod, int st, int dr) {
  if (st == dr) {
    fin>>A[nod].value;
    A[nod].flag = 0;
  } else {
    int mid = (st + dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);
    A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
```

```
    A[nod].flag = 0;
  }
}

void query(int nod, int st, int dr, int a, int b) {
  if (a <= st && dr <= b) {
    minim = min(minim, A[nod].value);
  } else {
    if (A[nod].flag == 1) {
      A[2*nod].value = A[nod].value;
      A[2*nod].flag = 1;
      A[2*nod+1].value = A[nod].value;
      A[2*nod+1].flag = 1;
      A[nod].flag = 0;
    }

    int mid = (st + dr)/2;
    if (a <= mid)
      query(2*nod, st, mid, a, b);
    if (b > mid)
      query(2*nod+1, mid+1, dr, a, b);

    A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
  }
}

void update(int nod, int st, int dr, int a, int b, int x) {
  if (a <= st && dr <= b) {
    A[nod].value = x;
    A[nod].flag = 1;
  } else {

    if (A[nod].flag == 1) {
      A[2*nod].value = A[nod].value;
      A[2*nod].flag = 1;
      A[2*nod+1].value = A[nod].value;
      A[2*nod+1].flag = 1;
      A[nod].flag = 0;
    }

    int mid = (st + dr)/2;
    if (a <= mid)
      update(2*nod, st, mid, a, b, x);
    if (b > mid)
      update(2*nod+1, mid+1, dr, a, b, x);
    A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
  }
}

int main () {

  fin>>n;
  build(1, 1, n);
  fin>>m;
  for (int i=1;i<=m;i++) {
    fin>>t>>a>>b;
    if (t == 1) {
```

```
      minim = INF;
      query(1, 1, n, a, b);
      fout<<minim<<"\n";
    } else {
      fin>>x;
      update(1, 1, n, a, b, x);
    }
  }
  return 0;
}
```

Întrucât se fac modificări în noduri și în cazul unui query (atunci când se trece prin cele cu flagul 1), acestea pot afecta și minimele din nodurile de deasupra lor, așadar, și la revenirea din autoapeluri trebuie pusă instrucțiunea de actualizare a valorii din nodul curent în funcție de valoarea din fii:

```
A[nod].value = min(A[2*nod].value, A[2*nod+1].value);
```

Semnificația flag-ului a fost descrisă conform cu specificul problemei de mai sus. În funcție de caz, putem da acestuia și alte roluri.

2.4 Folosirea arborilor de intervale pentru operații mai complexe

Problema: Avem un șir de numere întregi asupra căruia putem face două tipuri de operați:

- aflarea minimului urmată de ștergerea sa (dacă minimul apare de mai multe ori se șterge doar o apariție, cea de la indicele mai mic); lungimea șirului se actualizează (scade cu 1);
- schimbarea valorii unui element dat.

Enunțul complet se află pe pbinfo.

Dacă ne spune cineva că soluția este folosind arbori de intervale, ne putem gândi că apare o problemă prin faptul că se modifică lungimea vectorului de la o operație la alta (la ștergere). Noi știm că trebuie să formăm același set de triplete (nod, st, dr) la fiecare funcție recursivă. Dar modificându-se n, lungimea vectorului, lucrurile se dereglează.

Pentru a evita acest lucru apelăm la următorul truc: nu ștergem efectiv elementele ci doar le marcăm drept șterse. Procedând astfel, alt lucru de care trebuie ținut cont este că la operațiile de actualizare se dă poziția din momentul curent, ori în urma ștergerilor anterioare ea nu mai corespunde neapărat cu cea inițială, deci trebuie aflată. De asemenea, în afară de minim ne mai interesează și cea mai din stânga poziție pe care el se află.

Pentru a îndeplini toate cele de mai sus, vom ține, pentru fiecare nod din AINT (adică pentru fiecare interval) trei valori:

```
struct nod {
  int minim;
  int poz;
  int cnt;
};
```

- `minim` reprezintă valoarea minimă din intervalul corespunzător nodului;
- `poz` reprezintă cea mai din stânga poziție pe care se află minimul din interval; această valoare este relativă la dimensiunea inițială a vectorului;
- `cnt` reprezintă numărul de elemente nesterse încă în intervalul corespunzător lui nod (deci, dacă acest interval este $[st, dr]$, $dr - st + 1 - A[nod].cnt$ reprezintă numărul de elemente șterse).

Construim mai întâi arborele de intervale corespunzător vectorului dat:

```
void build(int nod, int st, int dr) {
  if (st == dr) {
    /**
     * pentru intervalele de lungime 1:
     * - minimul este chiar valoarea de la intrare;
     * - numărul de valori nesterse este 1 (inițial nu e nimic șters);
     * - poziția minimului este chiar cea care se dă la intrare;
     **/
    fin>>A[nod].minim;
    A[nod].poz = st;
    A[nod].cnt = 1;
  } else {
    int mid = (st + dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);

    /**
     * Dacă minimul provine din jumătatea stângă ținem în poz, pentru
     * nodul curent, valoarea lui poz din fiul stâng (chiar dacă am
     * avea aceeași valoare minimă și în fiul drept)
     **/
    A[nod].minim = A[2*nod].minim;
    A[nod].poz = A[2*nod].poz;
    if (A[2*nod+1].minim < A[nod].minim) {
      A[nod].minim = A[2*nod+1].minim;
      A[nod].poz = A[2*nod+1].poz;
    }
    /**
     * - inițial toate valorile sunt nesterse;
     * - altfel, puteam scrie: A[nod].cnt = dr-st+1;
     **/
    A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;

  }
}
```

Pentru operația de interogare lucrurile sunt relativ simple: În rădăcină avem și minimul și poziția sa. Astfel, putem afișa direct valoarea $A[1].minim$.

Acum trebuie să ștergem elementul de pe poziția $A[1].poz$. Noi am stabilit că vom face o ștergere logică, adică doar vom marca drept șters elementul de pe această poziție. Această marcare provoacă modificarea unor informații din AINT (scade cu 1 valoarea *cnt* din nodurile care conțin poziția de pe care ștergem, dar se poate modifica și minimul sau poziția sa în aceste noduri).

Practic această ștergere este o actualizare mai specială a valorii unui element (vom pune în el o valoare foarte mare, dar față de actualizarea de tipul 2, vom mai și seta la 0 valoarea *cnt* din nodul frunză corespunzător nodului marcat ca șters).

Vom folosi deci același cod atât pentru ștergerea unui element cât și pentru actualizarea valorii unui element.

```
void update(int nod, int st, int dr, int poz, int x) {
  if (st == dr) {
    A[nod].minim = x;
    A[nod].poz = st;
    if (x == INF)
      A[nod].cnt = 0; /// pentru ștergere
    else
      A[nod].cnt = 1; /// pentru actualizare
  } else {
    int mid = (st + dr)/2;
    if (poz <= mid)
      update(2*nod, st, mid, poz, x);
    if (poz > mid)
      update(2*nod+1, mid+1, dr, poz, x);

    A[nod].minim = A[2*nod].minim;
    A[nod].poz = A[2*nod].poz;

    if (A[2*nod+1].minim < A[nod].minim) {
      A[nod].minim = A[2*nod+1].minim;
      A[nod].poz = A[2*nod+1].poz;
    }

    A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
  }
}
```

Pentru ștergere facem apelul:

```
update(1, 1, n, A[1].poz, INF);
```

Pentru actualizare facem apelul:

```
fin>>p>>x;
update(1, 1, n, ?, x);
```

Acum mai rămâne de văzut cine este valoarea **?**.

Cum spuneam, la intrare ni se dă o poziție dinamică, adică ținând cont de ștergerile anterioare, pe când noi păstrăm elementele nesterse pe pozițiile lor inițiale. Trebuie deci

să aflăm pe ce poziție se află la un moment dat al p-lea element nesters. Pentru asta ne vom folosi de câmpul *cnt* al nodurilor din AINT.

Vom mai face o altă funcție care operează pe AINT:

```
getPoz(int nod, int st, int dr, int p)
```

cu semnificația: returnează al p-lea nod nesters din intervalul $[st, dr]$, cu informații păstrate în A pe poziția *nod*.

Codul este mai jos:

```
int getPoz(int nod, int st, int dr, int p) {
  if (st == dr) {
    return st;
  } else {
    int mid = (st + dr)/2;
    if (A[2*nod].cnt >= p)
      return getPoz(2*nod, st, mid, p);
    else
      return getPoz(2*nod+1, mid+1, dr, p-A[2*nod].cnt);
  }
}
```

Modul de funcționare este ca la o căutare binară: Al p-lea nod nesters din intervalul $[st, dr]$ este al p-lea nod nesters din intervalul $[st, mid]$ dacă în partea stângă sunt cel puțin p noduri nesterse, respectiv al x - lea nod din intervalul $[mid+1, dr]$ dacă în stânga sunt mai puțin de p noduri nesterse (unde x este egal cu p minus numărul de noduri nesterse în fiul stâng al lui *nod*). Practic la fiecare pas se coboară un nivel în arbore. Toate aceste funcții au timp de calcul de ordin logaritmic.

Programul complet este aici:

```
#include <fstream>
#define INF 2000000001
#define DIM 100010
using namespace std;

ifstream fin ("aesm.in");
ofstream fout("aesm.out");

struct nod {
  int minim;
  int poz;
  int cnt;
};

nod A[DIM * 4];
int n, m, x, t, p;

void build(int nod, int st, int dr) {
  if (st == dr) {
    fin>>A[nod].minim;
    A[nod].poz = st;
```

```
    A[nod].cnt = 1;
  } else {
    int mid = (st + dr)/2;
    build(2*nod, st, mid);
    build(2*nod+1, mid+1, dr);

    A[nod].minim = A[2*nod].minim;
    A[nod].poz = A[2*nod].poz;

    if (A[2*nod+1].minim < A[nod].minim) {
      A[nod].minim = A[2*nod+1].minim;
      A[nod].poz = A[2*nod+1].poz;
    }
    A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
  }
}

void update(int nod, int st, int dr, int poz, int x) {
  if (st == dr) {
    A[nod].minim = x;
    A[nod].poz = st;
    if (x == INF)
      A[nod].cnt = 0;
    else
      A[nod].cnt = 1;
  } else {
    int mid = (st + dr)/2;
    if (poz <= mid)
      update(2*nod, st, mid, poz, x);
    if (poz > mid)
      update(2*nod+1, mid+1, dr, poz, x);

    A[nod].minim = A[2*nod].minim;
    A[nod].poz = A[2*nod].poz;

    if (A[2*nod+1].minim < A[nod].minim) {
      A[nod].minim = A[2*nod+1].minim;
      A[nod].poz = A[2*nod+1].poz;
    }

    A[nod].cnt = A[2*nod].cnt + A[2*nod+1].cnt;
  }
}

int getPoz(int nod, int st, int dr, int p) {
  if (st == dr) {
    return st;
  } else {
    int mid = (st + dr)/2;
    if (A[2*nod].cnt >= p)
      return getPoz(2*nod, st, mid, p);
    else
      return getPoz(2*nod+1, mid+1, dr, p-A[2*nod].cnt);
  }
}

int main () {
```

```
  fin>>n;

  build(1, 1, n);

  fin>>m;
  for (int i=1;i<=m;i++) {
    fin>>t;
    if (t == 1) {
      fout<<A[1].minim<<"\n";
      update(1, 1, n, A[1].poz, INF);
    } else {
      fin>>p>>x;
      update(1, 1, n, getPoz(1, 1, n, p), x);
    }
  }
  return 0;
}
```

2.5 Probleme propuse

- Problema Actualizare element, minim interval
- Problema Actualizare element, CMMDC interval
- Problema Descompunere în intervale
- Problema Actualizare element, ștergere minim
- Problema Rest
- Problema SequenceQuery
- Problema Actualizare interval, minim interval
- Problema Hotel
- Problema Biscuiți
- Problema Eliminare
- Problema Intersecție segmente
- Problema Zoo
- Problema Demolish

2.6 Bibliografie

[1] Daniela Lica, *Arbori de intervale și aplicații în geometria computațională*, URL: https://www.infoarena.ro/arbori-de-intervale.

[2] Ștefan Istrate, *Algoritmi de baleiere*, URL: https://www.infoarena.ro/algoritmi-de-baleiere.

[3] *Lazy Propagation in Segment Tree*, URL: https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree.

[4] *Segment Tree*, URL: https://cp-algorithms.com/data_structures/segment_tree.html.

Capitolul 3

Determinarea celui mai apropiat strămoș comun a două noduri dintr-un arbore

prof. Mircea Rotar
Colegiul Național „Samuil Vulcan”, Beiuș

Definiția 3.1. Fie $T = (V, U)$ un arbore cu rădăcină, unde V este mulțimea nodurilor, iar U mulțimea muchiilor și $x, y \in V$ două noduri din T. Numim **cel mai apropiat strămoș comun** (engl. *lowest common ancestor*, abr. LCA) al nodurilor x și y acel nod z cu proprietatea că z este cel mai îndepărtat nod de la rădăcină care are ambele noduri ca descendenți.

Notație. Cel mai apropiat strămoș comun al nodurilor x și y este z se notează $LCA(x, y) = z$.

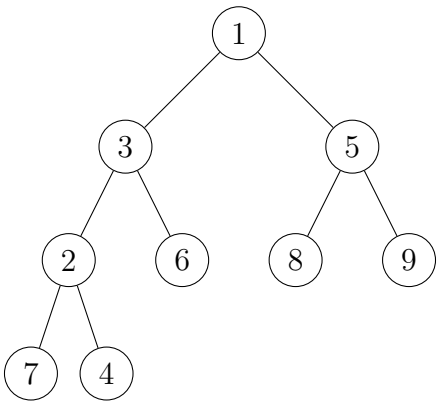

Figura 3.1: În acest arbore, $LCA(4, 6) = 3$, $LCA(8, 9) = 5$, $LCA(1, 2) = 1$, $LCA(6, 9) = 1$

De reținut!

Dacă notăm cu $nivel[x]$ adâncimea nodului x, lungimea drumului minim dintre două noduri x și y este: $nivel[x] + nivel[y] - 2 \times nivel[LCA(x, y)]$.

Aplicații

LCA are aplicații în multe domenii, cum ar fi sisteme de management al bazelor de date, procesarea limbajului natural, platforme pentru controlul versiunilor (de exemplu, GitHub, GitLab, BitBucket).

3.1 Tehnici de determinare a LCA

1. **Metoda de căutare simplă (naivă).** Pornind de la un nod x, se poate construi o stivă cu toți strămoșii săi. Se face același lucru pentru nodul y. Cel mai apropiat strămoș comun se va găsi comparând aceste stive. Această metodă are o complexitate de $\mathcal{O}(n)$ pentru arborele cu n noduri.
2. **Metoda preprocesării și a interogării rapide.** Folosind tehnici precum *Sparse Table* sau *Segment Tree*, se poate determina LCA în timp constant după o preprocesare în $\mathcal{O}(n \log n)$.
3. **Algoritmul *Tarjan's offline LCA*.** O metodă offline pentru găsirea LCA pentru mai multe perechi de noduri simultan având complexitatea $\mathcal{O}(n \log n + Q \log n)$.

3.1.1 Metoda de căutare simplă (naivă)

Descriere

Putem observa din definiție că LCA a două noduri x și y nu este altceva decât nodul de intersecție a lanțurilor de la x și y la nodul rădăcină. În arborele din Figura 3.1, lanțurile de la 7 și 6 la nodul rădăcină au prima lor intersecție la 3. Prin urmare, $LCA(7, 6) = 3$. Putem calcula lanțurile folosind DFS și găsim intersecția folosind o abordare bazată pe stivă sau folosind o abordare recursivă. Aceasta este soluția generală (naivă) și are o complexitate $\mathcal{O}(n)$ timp și $\mathcal{O}(n)$ spațiu.

Vectorul de tați asociat arborelui este:

i	1	2	3	4	5	6	7	8	9
$adj[i]$	0	3	1	2	1	3	2	5	5

Tabela 3.1: Vectorul de tați *adj* pentru arborele DFS din Figura 3.1.

Drumurile de la x la rădăcină, respectiv de la y la rădăcină se construiesc în stivele următoare:

Cât timp vârfurile celor două stive coincid, se consideră că respectivul vârf este LCA.

Ultimul vârf eliminat este $LCA(x, y)$.

Mai jos este codul pentru abordarea iterativă folosind stive.

Implementare

```
int findLCA(int x, int y, vector<int>& adj) {
  // adj[i] reprezintă nodul părinte a lui i
```

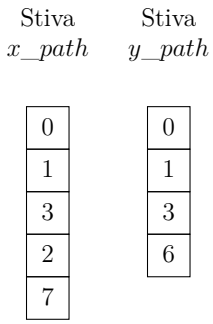

Figura 3.2: Stivele de strămoși pentru nodurile 7 și 6.

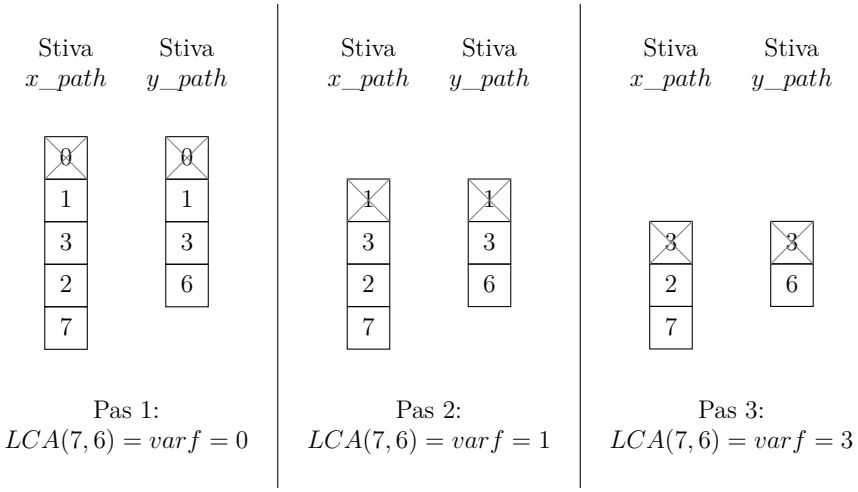

Figura 3.3: Determinarea LCA pentru nodurile 7 și 6.

```cpp
int root = 0;
stack<int> x_path, y_path;

// găsește drumul de la x la rădăcină
while (x != root) {
  x_path.push(x);
  x = adj[x];
}
x_path.push(x);

// găsește drumul de la y la rădăcină
while (y != root) {
  y_path.push(y);
  y = adj[y];
}
y_path.push(y);

int lca = -1;
// Găsește ultimul nod comun
while ((!x_path.empty() && !y_path.empty()) &&
       (x_path.top() == y_path.top())) {
  lca = x_path.top();
  x_path.pop();
  y_path.pop();
}
```

```
  return lca;
}
```

Complexitatea algoritmului

Construirea lanțului de la nodul x la rădăcină are o complexitate de $\mathcal{O}(n)$ în cazul cel mai defavorabil, unde n este adâncimea maximă a arborelui. Construirea lanțului de la nodul y la rădăcină are, de asemenea, o complexitate de $\mathcal{O}(n)$ în cazul cel mai defavorabil.

Compararea celor două stive pentru a găsi ultimul nod comun are o complexitate de $\mathcal{O}(n)$ în cazul cel mai defavorabil, deoarece fiecare nod este comparat o singură dată.

Prin urmare, complexitatea totală a timpului pentru funcția `findLCA` este $\mathcal{O}(n)$.

Complexitatea spațiu

Stiva *x_path* poate avea maximum n elemente. Stiva *y_path* poate avea, de asemenea, maximum n elemente. Prin urmare, complexitatea totală a spațiului pentru funcția `findLCA` este $\mathcal{O}(n)$.

3.1.2 Metoda preprocesării și a interogării rapide

Descriere

Algoritmul utilizează tehnica tabelului rar (*Sparse Table*) pentru a determina cel mai mic strămoș comun (LCA) al două noduri într-un arbore. În faza de preprocesare, se construiește o matrice `parent`, unde `parent[i][j]` indică al 2^i-lea strămoș al nodului j. Aceasta permite să „sărim” în arbore cu pași exponențiali, optimizând căutarea.

În funcția LCA, se ajustează adâncimile nodurilor u și v pentru a fi egale, apoi se parcurg strămoșii acestora în paralel până când se găsește un strămoș comun. Datorită preprocesării, interogările LCA sunt răspunse rapid, în timp logaritmic.

Implementare

```
const int MAXN = 100005;
const int LOG = 20;

// Matricea parent va stoca strămoșii pentru fiecare nod.
// parent[i][j] reprezintă al 2^i-lea strămoș al nodului j.
int parent[LOG][MAXN];

// Șirul depth va stoca adâncimea fiecărui nod în arbore.
int depth[MAXN];

// Funcția de preprocesare construiește matricea parent.
void preprocess(int n) {
  for (int i = 1; i < LOG; i++)
    for (int j = 1; j <= n; j++)
      if (parent[i - 1][j])
        parent[i][j] = parent[i - 1][parent[i - 1][j]];
}
```

```
// Funcția LCA returnează cel mai mic strămoș comun al nodurilor u și v.
int LCA(int u, int v) {
  // ne asigurăm că u este nodul cu adâncimea cea mai mare.
  if (depth[u] < depth[v]) swap(u, v);

  // Ridicăm u la aceeași adâncime cu v.
  for (int i = LOG - 1; i >= 0; i--)
    if (depth[u] - (1 << i) >= depth[v])
      u = parent[i][u];

  // Dacă u și v sunt identice, atunci u este LCA.
  if (u == v) return u;

  // Căutăm cel mai mic strămoș comun al lui u și v.
  for (int i = LOG - 1; i >= 0; i--) {
    if (parent[i][u] && parent[i][u] != parent[i][v]) {
      u = parent[i][u];
      v = parent[i][v];
    }
  }
  return parent[0][u];
}
```

Complexitatea algoritmului

Funcția `preprocess` are două bucle imbricate, fiecare având o complexitate de $\mathcal{O}(\log n)$ și $\mathcal{O}(n)$ respectiv. Complexitatea totală pentru `preprocess` este $\mathcal{O}(n \log n)$.

Funcția `LCA` are două bucle separate, fiecare având o complexitate de $\mathcal{O}(\log n)$. Complexitatea totală pentru `LCA` este $\mathcal{O}(\log n)$. Prin urmare, după o preprocesare inițială de $\mathcal{O}(n \log n)$, putem răspunde la interogările LCA în $\mathcal{O}(\log n)$.

3.1.3 Algoritmul *Tarjan's offline LCA*

Descriere

Algoritmul *Tarjan's offline LCA* identifică cel mai mic strămoș comun (LCA) pentru mai multe perechi de noduri într-un arbore. Acesta folosește tehnica „*Union-Find*" pentru a gestiona seturile disjuncte de noduri. Algoritmul presupune că toate interogările LCA sunt cunoscute în avans (de aceea se numește „*offline*").

În timpul parcurgerii arborelui, fiecare nod este marcat ca vizitat. Când se ajunge la un nod, se verifică dacă nodurile asociate cu interogările sale au fost vizitate. Dacă au fost vizitate, LCA pentru acea pereche de noduri poate fi determinat.

Algoritmul se bazează pe o parcurgere în adâncime a arborelui, explorând fiecare nod și muchie o singură dată. Pe măsură ce fiecare nod este vizitat, acesta este adăugat la structura *Union-Find* și strămoșul său este actualizat. Când ambele noduri dintr-o pereche de interogare au fost vizitate, LCA-ul lor poate fi identificat imediat folosind informațiile stocate în structura *Union-Find*.

Algoritmul este eficient și poate procesa simultan mai multe interogări LCA.

Implementare

```cpp
#include <vector>
#include <map>
using namespace std;

const int MAXN = 100005;

// Arborele reprezentat prin liste de adiacență; adj[i] reține vecinii
// nodului i
vector<int> adj[MAXN];

// Map care stochează interogările LCA. Pentru fiecare pereche de noduri
//(u, v), map-ul va conține LCA-ul lor după ce algoritmul este executat.
map<pair<int, int>, int> queries;

// Vector care stochează părintele direct al fiecărui nod în structura //Union-Find.
int parent[MAXN];

// Vector care indică dacă un nod a fost vizitat în timpul parcurgerii DFS.
bool visited[MAXN];

// Vector care stochează strămoșul direct al fiecărui nod în arbore.
int ancestor[MAXN];


// Funcția find pentru structura de date Union-Find.
// Returnează reprezentantul setului în care se află nodul x.
int find(int x) {
  if (parent[x] != x) {
    parent[x] = find(parent[x]);
  }
  return parent[x];
}

// Funcția union pentru structura de date Union-Find.
// Unește seturile a două noduri x și y.
void unionSet(int x, int y) {
  int rootX = find(x);
  int rootY = find(y);
  if (rootX != rootY) {
    parent[rootY] = rootX;
  }
}

// Algoritmul Tarjan LCA.
// Calculează LCA pentru toate perechile de noduri specificate în map-ul //queries.
void tarjanLCA(int u) {
  visited[u] = true;
  ancestor[u] = u;

  // Parcurgerea în adâncime a arborelui.
  for (int v : adj[u]) {
    if (!visited[v]) {
      tarjanLCA(v);
      unionSet(u, v);
      ancestor[find(u)] = u;
```

```
    }
  }

  // Verificarea interogărilor pentru nodul curent.
  for (int v : queries[u]) {
    if (visited[v]) {
      // (u, v) este o pereche de interogare și LCA-ul lor este ancestor[find(v)]
      queries[{u, v}] = ancestor[find(v)];
    }
  }
}

int main() {
  // Inițializări și construirea arborelui și a interogărilor.
  // Codul specific pentru această parte nu este furnizat.
  tarjanLCA(1);
  return 0;
}
```

Complexitatea algoritmului

Funcția `find` are o complexitate medie de $\mathcal{O}(\log n)$ datorită compresiei căii, dar în practică este foarte rapidă și se apropie de $\mathcal{O}(1)$ pentru majoritatea cazurilor. Funcția `unionSet` are, de asemenea, o complexitate medie de $\mathcal{O}(\log n)$. Funcția `tarjanLCA` parcurge fiecare nod și fiecare muchie o singură dată, având o complexitate de $\mathcal{O}(n+m)$, unde n este numărul de noduri și m este numărul de muchii (în cazul unui arbore, $m = n-1$).

Pentru fiecare nod, se pot verifica toate interogările asociate acestuia, ceea ce adaugă o complexitate de $\mathcal{O}(Q)$, unde Q este numărul total de interogări.

Prin urmare, complexitatea totală a algoritmului este $\mathcal{O}(n \log n + Q \log n)$ în cazul cel mai defavorabil, dar în practică, datorită eficienței structurii de date *Union-Find*, este mult mai rapidă și se apropie de $\mathcal{O}(N + Q)$.

3.2 Probleme

3.2.1 Problema binary_tree (pbinfo)

Se dau un arbore binar complet infinit cu rădăcina în nodul 1 în care pentru orice nod i fiii săi sunt nodurile $2i$, respectiv $2i+1$ și Q perechi de numere (u, v). Se cere să se afle pentru fiecare pereche lungimea celui mai scurt drum (ca număr de muchii) dintre nodurile u și v din arbore.

Date de intrare

Programul citește de la tastatură numărul Q, iar apoi Q perechi de numere naturale, fiecare pereche pe câte o linie.

Date de ieșire

Programul va afișa pe ecran pentru fiecare pereche (u, v) distanța dintre nodurile u și v din arbore.

Restricții și precizări

- $1 \leq Q \leq 100\,000$
- $1 \leq u, v \leq 2\,000\,000\,000$

Soluție

Pentru un arbore binar complet infinit, relația dintre un nod și fiii săi este dată de faptul că, pentru un nod i, fiul stâng este $2i$ și fiul drept este $2i + 1$. Pentru a găsi cel mai mic strămoș comun (LCA) dintre două noduri, se folosește faptul că părintele unui nod i este $i/2$. Soluția propusă urmărește următoarea logică: pentru două noduri u și v, dacă u este mai mare decât v, le inversăm. Apoi, mergem la părintele nodului v și incrementăm un contor. Repetăm acest proces până când nodurile devin identice, adică am găsit LCA. Numărul de muchii dintre u și v este egal cu numărul de pași făcuți pentru a ajunge de la u și v la LCA.

Implementare

```cpp
#include <iostream>
using namespace std;

/* Funcția lca determină cel mai mic strămoș comun (LCA) dintre două noduri u
   și v și returnează numărul de muchii dintre u și v.*/

int lca(int u,int v){
  int sol=0;          // Contor pentru numărul de muchii
  while(u!=v){          // Cât timp nodurile nu sunt identice
    if(v<u)          // Dacă v este mai mare decât u, le inversăm
      swap(u,v);
    v=v/2;          // Mergem la părintele nodului v
    ++sol;          // Incrementăm numărul de muchii
  }
  return sol;          // Returnăm numărul total de muchii dintre u și v
}

int main(){
  ios_base::sync_with_stdio(false); // Optimizări pentru citire si scriere
  cin.tie(nullptr); cout.tie(nullptr);
  int q,u,v;
  cin>>q;                  // Citim numărul de interogări
  while(q--)
    {cin>>u>>v; cout<<lca(u,v)<<'\n';}
  return 0;
}
```

Complexitatea

Complexitatea algoritmului este $\mathcal{O}(q \log n)$.

3.2.2 Problema Triplet Min Sum (csacademy)

Avem un arbore cu N noduri. Răspundeți la Q interogări de tipul: date fiind trei noduri distincte A B C, găsiți nodul D astfel încât suma distanțelor de la D la A, B și C să fie minimă.

Date de intrare

Prima linie conține două numere întregi, N și Q. Fiecare dintre următoarele $N-1$ linii conține două numere întregi, reprezentând două noduri care împart o muchie. Fiecare dintre următoarele Q linii conține trei numere întregi A, B și C.

Date de ieșire

Pentru fiecare interogare, afișați două numere pe o linie distinctă: nodul D și suma distanțelor de la D la A, B și C.

Indicații

Este important să alegem nodul D astfel încât să se afle pe toate cele trei drumuri între $A-B$, $A-C$ și $B-C$. Există întotdeauna un singur nod care respectă această proprietate, deci răspunsul este unic.

Dacă alegem o rădăcină pentru arborele nostru (să spunem nodul 1, alegerea nu contează cu adevărat), nodul D va fi cel mai mic strămoș comun (LCA) pentru cel puțin una dintre cele trei perechi. De obicei există două LCA-uri distincte, în acest caz ne interesează cel de la un nivel mai mare.

După ce găsim nodul D, mai trebuie să găsim distanțele de la D la A, B și C. Distanța dintre două noduri X și Y este egală cu $Nivel[X] + Nivel[Y] - 2 \cdot Nivel[LCA(X,Y)]$.

Implementare

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int Nmax = 1e5 + 5;
const int logNmax = 18;

int depth[Nmax];           // depth[i] stochează adâncimea nodului i în arbore
int pos[Nmax];              // pos[i] stochează poziția nodului i în arborele Euler.
int euler[2 * Nmax]; // ordinea nodurilor vizitate în timpul parcurgerii DFS.
int rmq[logNmax][2 * Nmax]; // tabelul pentru Range Minimum Query.
int l[2 *Nmax];            // l[i] stochează logaritmul în baza 2 al numărului i
int euler_count;            // Câte noduri au fost vizitate în timpul DFS.
vector<int> v[Nmax]; // v[i] este lista de adiacență a nodului i.

/**
```

```cpp
 * Funcția DFS se folosește pentru a construi arborele Euler și adâncimea
 * fiecărui nod.
 *
 * - Șirul euler stochează ordinea nodurilor vizitate.
 * - Șirul pos stochează poziția fiecărui nod în arborele Euler.
 *
 * Adâncimea fiecărui nod este calculată în funcție de adâncimea părintelui
 * său.
 **/
void DFS(int x, int p) {
  euler[++euler_count] = x;
  pos[x] = euler_count;
  if (p) {
    depth[x] = depth[p] + 1;
  }
  for (auto &it: v[x]) {
    if (it != p) {
      DFS(it, x);
      euler[++euler_count] = x;
    }
  }
}

/**
 * Funcția LCA este pentru a găsi cel mai mic strămoș comun (LCA) dintre două
 * noduri folosind arborele Euler și RMQ (Range Minimum Query).
 *
 * Se folosește tehnica de a "sări" în arbore cu pași exponențiali pentru a
 * găsi LCA într-un mod eficient.
 **/
int LCA(int x, int y) {
  if (pos[x] > pos[y]) {
    swap(x, y);
  }
  int dif = pos[y] - pos[x] + 1;
  int log_dif = l[dif];

  int best = rmq[log_dif][pos[x]];
  if (depth[ rmq[log_dif][pos[x] + dif - (1 << log_dif)] ] < depth[best]) {
    best = rmq[log_dif][pos[x] + dif - (1 << log_dif)];
  }
  return best;
}

// Funcție pentru a calcula distanța dintre două noduri
int dist(int x, int y) {
  return depth[x] + depth[y] - 2 * depth[LCA(x, y)];
}

/**
 * Inițializare pentru LCA folosind RMQ (Range Minimum Query).
 *
 * Construiește arborele Euler și pregătește tabelul RMQ pentru interogări
 * rapide.
 **/
void initLCA() {
  DFS(1, 0);
```

```
  for (int i = 2; i <= euler_count; ++i) {
    l[i] = l[i / 2] + 1;
  }
  for (int i = 1; i <= euler_count; ++i) {
    rmq[0][i] = euler[i];
  }
  for (int j = 1; (1 << j) <= euler_count; ++j) {
    for (int i = 1; i <= euler_count; ++i) {
      rmq[j][i] = rmq[j - 1][i];
      if (i + (1 << (j - 1)) <= euler_count) {
        if (depth[rmq[j-1][i + (1 << (j-1))]] < depth[rmq[j][i]]) {
          rmq[j][i] = rmq[j - 1][i + (1 << (j - 1))];
        }
      }
    }
  }
}

int main() {
  int n, q;
  cin >> n >> q;

  // Construirea listei de adiacență a arborelui
  for (int i = 1; i < n; ++i) {
    int x, y;
    cin >> x >> y;
    v[x].push_back(y);
    v[y].push_back(x);
  }

  // Inițializare pentru LCA
  initLCA();

  // Procesarea interogărilor
  while (q--) {
    int a, b, c;
    cin >> a >> b >> c;
    /**
     * Pentru fiecare interogare, se calculează LCA și distanța pentru fiecare
     * combinație posibilă dintre nodurile A, B și C pentru a determina nodul
     * D cu suma distanțelor minimă
     **/
    int best = LCA(a, b);
    int best_sum = dist(best, a) + dist(best, b) + dist(best, c);
    int crt = LCA(a, c);
    int crt_sum = dist(crt, a) + dist(crt, b) + dist(crt, c);
    if (crt_sum < best_sum) {
      best = crt;
      best_sum = crt_sum;
    }
    crt = LCA(b, c);
    crt_sum = dist(crt, a) + dist(crt, b) + dist(crt, c);
    if (crt_sum < best_sum) {
      best = crt;
      best_sum = crt_sum;
    }
    cout << best << " " << best_sum << "\n";
```

```
  }
  return 0;
}
```

Complexitate

Complexitatea algoritmului este $\mathcal{O}(N \log N + Q \log N)$.

3.3 Bibliografie

[1] M. A. Bender și M. Farach-Colton, ``The LCA Problem Revisited'', în *Latin American Symposium on Theoretical Informatics* (2000), pp. 88–94.

[2] R. E. Tarjan, ``Applications of Path Compression on Balanced Trees'', în *Journal of the ACM (JACM)* 26(4) (1979), pp. 690–715.

[3] D. Harel și R. E. Tarjan, ``Fast Algorithms for Finding Nearest Common Ancestors'', în *SIAM Journal on Computing* 13(2) (1984), pp. 338–355.

[4] M. R. Schmid, *Algorithms and Data Structures: The Basic Toolbox*, Springer Science and Business Media, 2010.

[5] Rohit Taparia, *Lowest Common Ancestor*, URL: https://leetcodethehardway.com/tutorials/graph-theory/lca.

Capitolul 4

Range minimum query (RMQ)

prof. Marius Nicoli
Colegiul Național „Frații Buzești” Craiova
Centrul de Pregătire pentru Performanță în Informatică Craiova

4.1 Prezentarea problemei

Se dă un șir de valori numerice și mai multe interogări ce presupun aflarea minimului pentru secvențe oarecare din șir.

Notăm cu v tabloul în care se memorează elementele șirului și cu n numărul de elemente. Presupunem că acestea sunt stocate începând cu poziția 1. Notăm cu m numărul de interogări și considerăm că fiecare dintre acestea este precizată prin indicii de început și de final ai secvenței, notați st și dr ($st \leq dr$).

Observăm că întrebările se pun fără ca între timp șirul să se mai modifice. Încadrăm deci problema la capitolul „operații statice pe șiruri”.

Soluție

Soluția **brută** presupune ca la fiecare interogare să determinăm minimul din secvența dată, printr-o parcurgere de la indicele st la indicele dr.

```
fin>>n>>m;
for (i=1; i<=n; i++)
    fin>>v[i];
for (i=1; i<=m; i++) {
    fin>>st>>dr;
    minim = INF;
    for (j=st; j<=dr; j++)
        if (v[j] < minim)
            minim = v[j];
    fout<<minim<<"\n";
}
```

Timpul de calcul este de ordinul $\mathcal{O}(n \times m)$, iar pe date mari programul va rula lent.

O soluție **mai bună** se poate obține folosind arbori de intervale. Structura de date necesită însă cunoștințe de programare mai avansată și oferă timp de calcul de $\mathcal{O}(\log n)$ pentru fiecare interogare, precum și timp de calcul de $\mathcal{O}(n \log n)$ pentru pregătirea structurii.

Prezentăm în continuare o soluție care necesită o preprocesare cu $\mathcal{O}(n \log n)$ timp și memorie, dar pentru care rezultatul la fiecare întrebare se poate obține în timp constant.

Întrucât elementele vectorului nu se mai modifică în timpul interogărilor, putem reorganiza de la început datele într-un mod convenabil tipului nostru de interogări (minim pe secvențe).

Probabil ne gândim mai întâi la sume parțiale. Dacă am fi avut nevoie de sume pe secvențe, puteam precalcula un șir de sume parțiale ($s[i]$ = suma elementelor din v de la poziția 1 la poziția i) și la fiecare interogare am fi răspuns în timp constant ($s[dr] - s[st - 1]$). Calculul lui s ar necesita timp constant, realizându-se la citire printr-o instrucțiune suplimentară de atribuire ($s[i] = s[i-1] + v[i]$).

Fiind vorba despre minime, realizarea de „minime parțiale" nu ne-ar ajuta. Minimul din secvența de la 1 la i s-ar obține ușor, $minim[i] = \min(v[i], minim[i-1]))$ dar nu ne-ar fi util pentru a obține minimul din secvența de la st la dr (nu putem decide minimul din secvență folosind $minim[dr]$ și $minim[st]$, ca în cazul sumei).

Vom folosi altă abordare, care ne oferă timp de executare constant pentru fiecare interogare. Pentru fiecare poziție i din v, vom calcula mai multe minime. Pentru fiecare secvență de lungime putere de 2 care începe la poziția i, vom calcula minimul dintre elementele ei.

De exemplu, pentru datele de intrare: $n = 11$ și $v = (4, 2, 7, 8, 3, 5, 9, 1, 6, 2, 9)$, și pentru poziția 3 (acolo unde este valoarea 7) vom calcula 4 valori:

- minimul unei secvențe de lungime 1 ce începe la poziția 3 (secvența: 7), cu valoarea 7;
- minimul unei secvențe de lungime 2 ce începe la poziția 3 (secvența: 7, 8), cu valoarea 7;
- minimul unei secvențe de lungime 4 ce începe la poziția 3 (secvența: 7, 8, 3, 5), cu valoarea 3;
- minimul unei secvențe de lungime 8 ce începe la poziția 3 (secvența: 7, 8, 3, 5, 9, 1, 6, 2), cu valoarea 1.

Deci, pentru fiecare poziție din v vom avea maximum $\log_2 n$ minime de calculat (atâtea puteri de 2 sunt mai mici sau egale cu n).

Astfel, vom obține o matrice $r[p][i]$ = minimul din v pentru elementele dintr-o secvență de lungime 2^p care începe la poziția i.

Valorile de pe linia 0 a acestei matrice vor fi minimele unor secvențe de lungime 1 care încep în șirul dat la indicele din dreptul valorii, valorile de pe linia 1 reprezintă minimele unor secvențe de lungime 2, cele de pe linia 2 reprezintă minimele pentru secvențe de lungime 4 etc.

Pentru vectorul v exemplificat mai sus, matricea r ar fi:

r \ v	4	2	7	8	3	5	9	1	6	2	9
2^0	4	2	7	8	3	5	9	1	6	2	9
2^1	2	2	7	3	3	5	1	1	2	2	9
2^2	2	2	3	3	1	1	1	1	2	2	9
2^3	1	1	1	1	1	1	1	1	2	2	9

Tabela 4.1: Matricea r corespunzătoare vectorului V.

Deci indicii de coloană reprezintă indici din v, adică poziții unde încep secvențele de lungime putere de 2 pentru care se calculează minime. Indicii de linie reprezintă exponenți ai puterii de 2 ce dă lungimea secvențelor pentru care se stochează informații pe acea linie. O altă observație: elementele de pe linia 0 a lui r sunt chiar elementele șirului dat.

Promitem așadar că dacă am reușit să memorăm minime pentru un număr de $\mathcal{O}(n \log n)$ secvențe dintre cele aproximativ n^2 câte sunt în total, am putea apoi să aflăm minimul din oricare altă secvență în timp constant (fără repetiții suplimentare la fiecare interogare).

Să vedem mai departe cum realizăm acestea. Mai întâi vom arăta cum construim valorile din r, iar în etapa a doua vom arăta cum ne folosim de r pentru a realiza interogările.

Modul de calcul pentru r

Linia 0 o inițializăm cu valorile șirului de la intrare. În practică putem face asta direct de la citire, astfel nici nu mai e nevoie de declararea vectorului v.

```
int r[17][100010];
fin>>n;
for (i=1; i<=n; i++)
    fin>>r[0][i];
```

Observăm modul în care declarăm matricea în care structurăm datele pentru RMQ (numărul de linii este de $\mathcal{O}(\log n)$ unde n este numărul de coloane).

Pentru a construi elementele de pe celelalte linii este esențială următoarea observație: orice secvență pentru care calculăm minimul pe linia i (deci care are lungimea 2^i) se obține concatenând două secvențe pentru care avem informații pe linia $i-1$ (deci de lungime 2^{i-1}).

Pentru o mai bună înțelegere este utilă analizarea datelor din următorul tabel:

r \ v	4	2	7	8	3	5	9	1	6	2	9
2^0	4	2	7	8	3	5	9	1	6	2	9
2^1	2	2	7	3	3	5	1	1	2	2	9
2^2	2	2	3	3	1	1	1	1	2	2	9
2^3	1	1	1	1	1	1	1	1	2	2	9

Tabela 4.2: Relația între valorile din matricea r.

Concret, elementul $r[3][2]$ trebuie să memoreze minimul din toată secvența hașurată sus, adică de lungime 2^3 și care începe la poziția 2. Această secvență este formată din cea de lungime 2^2 care începe la poziția 2 și din cea de lungime 2^2 care începe la poziția 6. Minimele din aceste două jumătăți ale ei se află în elementele $r[2][2]$ respectiv $r[2][6]$. Așadar, valoarea $r[3][2] = \min(r[2][2], r[2][6])$.

Privind la general, un element de pe linia p se calculează ca minimul a două elemente de pe linia $p-1$.

$$r[p][i] = \min(r[p-1][i], r[p-1][i+2^p-1])$$

Având deja linia 0 cunoscută nu ne rămâne decât să calculăm valorile linie cu linie, în ordinea crescătoare a indicilor de linii. Valoarea 2^{p-1} se poate calcula direct folosind operatorul de deplasare la stânga pe biți (`1 << (p-1)`).

O ultimă observație înainte de a prezenta secvența de cod: la determinarea minimului dintre cele două elemente de pe linia anterioară trebuie să ne asigurăm că există cel din dreapta, fiind suficient să testăm ca `i + (1 << (p-1))` să fie mai mic sau egal cu n.

```
for (p=1; (1<<p) <= n; p++)
    for (i=1; i<=n; i++) {
        r[p][i] = r[p-1][i];
        if (i +(1<<(p-1))<=n && r[p][i] > r[p-1][i+ (1<<(p-1))])
            r[p][i] = r[p-1][i + (1<<(p-1))];
}
```

Ca detaliu de implementare, am putea stoca într-o variabilă valoarea coloanei elementului din dreapta de pe linia anterioară, astfel obținem un cod mai clar și mai rapid.

```
for (p=1; (1<<p) <= n; p++)
    for (i=1; i<=n; i++) {
        r[p][i] = r[p-1][i];
        int j = i +(1<<(p-1));
        if (j<=n && r[p][i] > r[p-1][j])
            r[p][i] = r[p-1][j];
}
```

Este acum mai vizibil că am construit o structură de date care ocupă memorie de $\mathcal{O}(n \log n)$ și timpul de executare pentru acest lucru este de același ordin.

Trecem acum la **etapa a doua**, să vedem cum folosim structura construită. Noi primim la fiecare interogare capetele unui interval, $[st, dr]$ care, evident, nu este neapărat de lungime putere de 2.

O primă idee ar fi să descompunem numărul $L = dr - st + 1$ (lungimea intervalului) în baza 2, adică scriindu-l ca sumă de puteri de 2, putem să aflăm minimul său folosind minimele din intervalele corespunzătoare, acestea fiind de lungime putere de 2 (deci le regăsim în r). Descompunerea unui număr în baza 2 are însă timp de calcul de ordin logaritmic, ori noi ne-am propus să răspundem la fiecare interogare în timp constant.

Pentru acest lucru, să considerăm cea mai mare putere de 2 mai mică sau egală decât L (notăm cu *len* această valoare). Această putere de 2 este totodată mai mare decât

jumătatea lungimii intervalului (altfel nu ea ar fi cea mai mare putere de 2 mai mică sau egală cu L, ci dublul ei). Acum să considerăm, intervalul de lungime len care începe la poziția st și pe cel de lungime len care se termină la poziția dr.

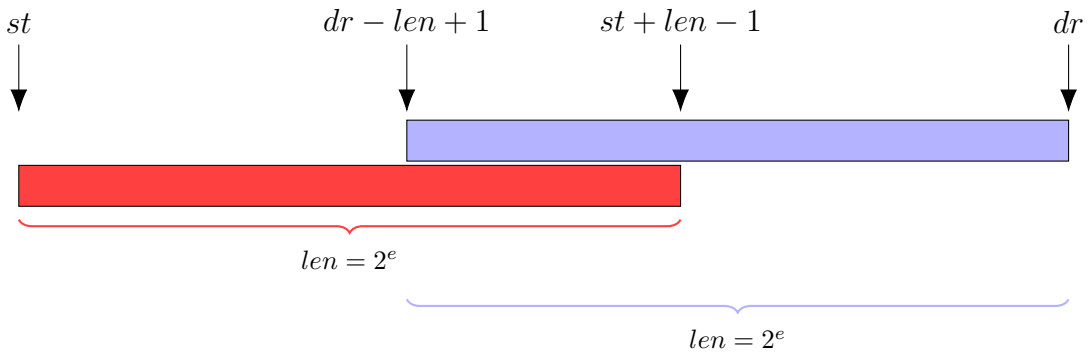

Figura 4.1: Acoperirea unui interval cu blocuri de dimensiune putere a lui 2.

Aceste două intervale acoperă în întregime intervalul $[st, dr]$, iar faptul că se suprapun la mijloc nu ne încurcă. Pentru fiecare dintre cele două intervale noi avem câte un element în r unde este stocat minimul secvenței corespunzătoare.

Să notăm cu e exponentul puterii lui 2 egală cu len: $2^e = len$.

Elementul din r care stochează minimul din secvența care începe la poziția st și are lungimea 2^e este $r[e][st]$ iar elementul care stochează minimul din secvența de lungime len care se termină la poziția dr este $r[e][dr - 2^e + 1]$. Formula provine din faptul că noi trebuie să indicăm începutul secvenței (care are lungimea len și se termină la poziția dr) - nu uităm că len este o putere de 2. Astfel, la o interogare de forma: care este minimul din secvența care începe la poziția st și se termină la poziția dr, răspunsul este: $\min(r[e][st], r[e][dr - 2^e + 1])$, adică minimul dintre două valori precalculate deja anterior.

Ultimul lucru care ne mai încurcă este: cum aflăm valoarea e (exponentul celei mai mari puteri de 2 mai mică sau egală cu o valoare dată).

O primă soluție ar fi să ne folosim de funcții de calcul al logaritmului care se găsesc în biblioteca de funcții matematice. Acestea implică operații cu numere reale și sunt în general lente, lucru vizibil mai ales dacă sunt apelate de multe ori.

Soluția pe care noi o alegem este să precalculăm valorile e pentru fiecare număr de la 1 la n. Astfel, vom construi un vector E, în care $E[i]$ = exponentul celei mai mari puteri de 2 mai mică sau egală cu i. $E[i] = 1 + E[i/2]$. Intuitiv este destul de evident (la dublarea valorii, exponentul puterii lui 2 crește cu 1). Pornind cu $E[1] = 0$, putem calcula dinainte vectorul E ca o etapă de precalculare:

```
E[1] = 0;
for (i=2; i<=n; i++)
    E[i] = 1 + E[i/2];
```

Implementare

Programul complet este:

```
#include <fstream>
#define INF 1000000000
using namespace std;
ifstream fin ("rmq.in");
ofstream fout("rmq.out");
int r[17][100010];
int E[100010];
int n, m, i, j, st, dr, minim, p, e, len;
int main () {
    fin>>n>>m;
    for (i=1;i<=n;i++)
        fin>>r[0][i];

    for (p=1; (1<<p) <= n; p++)
        for (i=1;i<=n;i++) {
            r[p][i] = r[p-1][i];
            j = i + (1<<(p-1));
            if (j <= n && r[p][i] > r[p-1][j])
                r[p][i] = r[p-1][j];
        }
    E[1] = 0;
    for (i=2;i<=n;i++)
        E[i] = 1 + E[i/2];

    for (i=1;i<=m;i++) {
        fin>>st>>dr;
        e = E[dr-st+1];
        len = (1<<e);
        fout<<min(r[e][st], r[e][dr-len+1])<<"\n";
    }
}
```

4.2 RMQ 2D

Vom aplica tehnica folosind o problemă de pe infoarena.ro, devenită clasică: Plantație.

Avem dată o matrice și interogările sunt submatrice pătratice ale ei pentru care trebuie să determinăm elementul maxim (evident că structurile pot fi folosite la fel și pentru maxim și pentru minim).

Vom nota cu A matricea dată (pentru simplitate, în problema plantație ea este pătratică, cu n linii și n coloane). Pentru o interogare vom folosi notația (i, j, lat) cu semnificația: să se determine valoarea maximă din submatricea pătratică având colțul stânga-sus (i, j) și latura lat.

În etapa de precalculare, pentru fiecare poziție (i, j) din matricea dată A, vom calcula maximele din toate submatricele pătratice care au colțul stânga sus în (i, j) și care au latura putere de 2. Vor fi $\log_2 n$ astfel de submatrice.

Așadar vom avea o matrice tridimensională de dimensiune $n \times n \times \log_2 n$.

Așa cum la cazul unidimensional pentru fiecare poziție de i determinam minimele din secvențe care încep la poziția i și au lungime putere de 2, la cazul bidimensional pentru

fiecare poziție (i, j) din matrice determinăm maximele din toate submatricele pătratice care au colțul stânga-sus (i, j) și au laturi cu lungimea putere de 2.

Pentru o mai bună înțelegere, să exemplificăm. Fie matricea A:

3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3
3	3	4	3	3	3	3	3	3
3	5	3	4	6	3	3	3	3
3	3	3	3	3	3	5	3	3
7	3	7	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	8	3
2	3	3	3	8	3	3	3	3
3	3	3	3	3	3	3	3	3
1	3	3	1	3	3	3	2	3

$r[0][3][2] = 3$
$r[1][3][2] = 5$
$r[2][3][2] = 7$
$r[3][3][2] = 8$

Tabela 4.3: Matricea r corespunzătoare matricei A.

În partea dreaptă avem toate valorile calculate pentru submatrice cu colțul stânga-sus în poziția (3,2).

Pentru a realiza precalcularea ne folosim de aceeași observație ca și la vectori: o putere de 2 se poate compune din exact două puteri de 2 cu exponent cu 1 mai mic. Aici un pătrat cu latură putere 2^p îl vom descompune în cele 4 „sferturi" care sunt fiecare pătrate de latură 2^{p-1}.

Astfel,

$$r[p][i][j] = \max \begin{cases} r[p-1][i][j], \\ r[p-1][i+2^{p-1}][j], \\ r[p-1][i][j+2^{p-1}], \\ r[p-1][i+2^{p-1}][j+2^{p-1}] \end{cases}$$

Extinzând de la vectori, ne imaginăm că structura este formată din $\log_2 n$ matrice (fiecare de $n \times m$) una peste alta, și un element al unei matrice (aflat pe poziția p în șirul matricelor) se calculează în funcție de 4 elemente ale matricei anterioare (cea de pe poziția $p-1$).

Matricea 0 este chiar matricea dată (valorile corespund unor submatrice pătratice de latură $1 = 2^0$).

Pentru a le calcula pe celelalte una din cealaltă avem următorul cod în care implementăm cele spuse mai sus.

```
for (int p=1,lat=2; lat<=n; p++,lat*=2){
    for (int i1=1; i1<=n-lat+1; i1++){
        for (int j1=1; j1<=n-lat+1; j1++){
            i2=i1+(lat>>1);
            j2=j1+(lat>>1);
            r[p][i1][j1] = maxim(
```

```
                            r[p-1][i1][j1],
                            r[p-1][i2][j1],
                            r[p-1][i1][j2],
                            r[p-1][i2][j2]));
        }
    }
}
```

Ca detaliu de implementare, am ales ca în primul `for` să calculăm simultan puterea de 2 și exponentul.

Odată realizată precalcularea vom folosi un raționament similar pentru a răspunde la interogare. Reluăm contextul: avem precalculat tabloul tridimensional r și avem o interogare de forma (i, j, L) care specifică o submatrice pătratică de latură L și care are colțul stânga sus (i, j) pentru care dorim să aflăm minimul.

Vom considera len = cea mai mare putere de 2 mai mică sau egală cu L (notăm cu e exponentul acestei puteri). Vom determina minimul cerut folosind 4 valori din r care corespund la 4 submatrice pătratice de latură 2^{e-1} și care se suprapun peste colțuri ale submatricei pătratice de latură L pe care facem interogarea. Evident că aceste 4 submatrice se suprapun eventual în mijloc dar acest lucru nu ne încurcă, este important că ele asigură minimul din toată submatricea de dimensiune L și că nu iau în calcul elemente din afara ei.

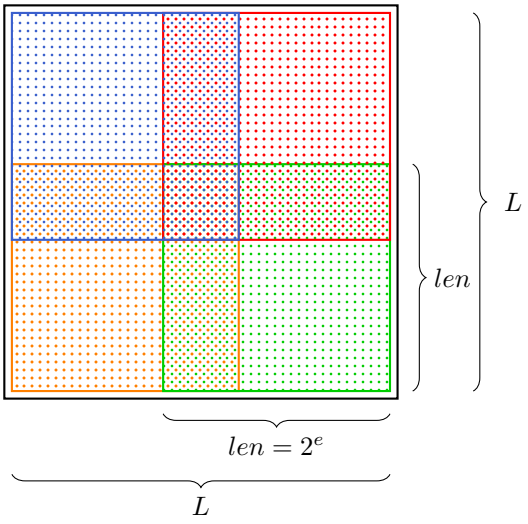

Figura 4.2: Acoperirea cu submatrice pătrate de dimensiune putere a lui 2.

Implementare

```
#include <fstream>
#define DIM 502

using namespace std;

int n,m,i1,i2,j1,j2,lat,L,len,k;
int r[10][DIM][DIM];
```

```cpp
int E[DIM];

inline int maxim(int a, int b){
    return (a>b ? a : b);
}

int main(){
    ifstream fin("plantatie.in");
    ofstream fout("plantatie.out");
    fin>>n>>m;
    for (int i=1;i<=n;i++){
        for (int j=1;j<=n;j++){
            fin>>r[0][i][j];
        }
    }
    for (int p=1,lat=2; lat<=n; p++,lat*=2){
        for (int i1=1;i1<=n-lat+1;i1++){
            for (int j1=1;j1<=n-lat+1;j1++){
                i2=i1+(lat>>1);
                j2=j1+(lat>>1);
                r[p][i1][j1] = max(
                                max(r[p-1][i1][j1],r[p-1][i2][j1]),
                                max(r[p-1][i1][j2],r[p-1][i2][j2])
                            );
            }
        }
    }
    E[1] = 0;
    for (int i=2;i<=n;i++)
        E[i] = 1 + E[i/2];

    for (;m;m--){
        fin>>i1>>j1>>L;
        k = E[L];
        len = (1<<k); /// cea mai mare putere de 2 <= latura
                      /// patratului de la interogare
        i2 = i1+L-len;
        j2 = j1+L-len;
        fout<<max(
                  max(r[k][i1][j1],r[k][i1][j2]),
                  max(r[k][i2][j1],r[k][i2][j2])
                )<<"\n";
    }
    return 0;
}
```

Complexitate

În concluzie avem o complexitate timp de $\mathcal{O}(n^2 \log_2 n)$ în etapa de preprocesare (+memorie de același ordin) dar obținem timp constant de răspuns la interogări.

4.3 RMQ 2D cu interogări pe dreptunghiuri

Așa cum este scris în titlu, diferența față de problema anterioară este că avem interogare pe un dreptunghi și nu pe un pătrat.

Vom face analiza folosind problema Euclid de pe infoarena.

Se dă o matrice cu m linii și n coloane (cu elemente naturale) și trebuie să determinăm un dreptunghi cu h linii și w coloane în care cmmdc al tuturor elementelor sale să fie cât mai mare posibil.

Este acum ocazia să subliniem că trucul de RMQ funcționează nu numai la maxime și minime pe scevențe ci și la cmmdc (precalculăm cmmdc pentru toate secvențele de lungime putere de 2).

Revenind la problema pe care o analizăm în acest paragraf, vă invităm să studiați în primul rând descrierea oficială a soluției sale, foarte frumoasă, care se găsește pe infoarena.

Venim și noi aici, în continuare, cu câteva explicații. O primă abordare este să construim un tablou cu 4 dimensiuni, în care un element $D[i][j][e_i][e_j]$ reprezintă informația dorită (cmmdc la noi) pentru un dreptunghi ce are colțul stânga-sus la linia i și coloana j și care are 2^{e_i} linii și 2^{e_j} coloane. Astfel, dacă avem o interogare pe o submatrice cu h linii și w coloane, ne interesează cea mai mare putere de 2 mai mică sau egală cu h (să notăm e_h exponentul ei) și cea mai mare putere de 2 mai mică sau egală w (notăm e_w exponentul ei). Pentru a afla cmmdc pentru un dreptunghi de dimensiuni (h, w) ne interesează valorile precalculate din cele 4 dreptunghiuri de dimensiuni $(2^{e_h}, 2^{e_w})$ suprapuse peste colțurile dreptunghiului de la interogare.

La această soluție avem nevoie de memorie de $\mathcal{O}(m \times n \times \log_2 m \times \log_2 n)$. Timpul de calcul este de același ordin, în etapa de preprocesare.

A doua abordare reduce cu un log factorul de timp și de memorie.

În prima etapă vom calcula informații pentru secvențe de lungime putere de 2 pe linii și apoi determinăm cmmdc din toate secvențele de lungime w, pe linii. Deci nu mai gândim cmmdc pe submatrice ci doar pe vectori (tratam fiecare linie ca pe un vector). Așadar putem obține o structură L în care $L[i][j]$ = cmmdc al elementelor din secvența de lungime w care începe pe linia i la coloana j. Deci fiecare element din L ține acum informația dintr-o secvență de lungime w din matricea dată. Rămâne acum să aplicăm asupra matricei L același raționament, dar pe coloane și ne interesează cmmdc pe scevențe de lungime putere de 2 pe coloane din L și în final obținem cmmdc pe secvențe de lungime h pe coloane din L și evident soluția este maximul dintre cmmdc din aceste secvențe.

Implementare

```
#include <cstdio>
#define DIM 401
using namespace std;
int D[9][DIM][DIM];
int L[DIM][DIM];
int E[DIM];
int m, n, h, w, i, j, t, e, sol, T;
```

```
int cmmdc(int a, int b)
{
    int r;
    while (b)
    {
        r = a%b;
        a = b;
        b = r;
    }
    return a;
}

int maxim(int a, int b)
{
    return a > b ? a : b;
}

int main ()
{
    FILE *fin  = fopen("euclid.in","r");
    FILE *fout = fopen("euclid.out","w");

    /// precalculare: E[i] = exponentul celei mai mari
    /// puteri de 2 mai mică sau egală cu i
    E[1] = 0;
    for (i=2; i<=400; i++)
        E[i] = 1 + E[i/2];
    fscanf(fin,"%d",&T);
    for (t = 1; t<=T; t++)
    {
        sol = 0;
        fscanf(fin,"%d%d%d%d",&m,&n,&h,&w);
        for (i=1; i<=m; i++)
            for (j=1; j<=n; j++)
            {
                fscanf(fin,"%d",&D[0][i][j]);
            }
/// calculăm în D cmmdc pentru secvențe
/// de lungime putere de 2, pe linii
        for (e=1; (1<<e) <= n; e++)
            for (i=1; i<=m; i++)
                for (j=1; j<=n; j++)
                {
                    D[e][i][j] = D[e-1][i][j];
                    if (j+(1<<(e-1)) <= n)
                        D[e][i][j] = cmmdc(D[e-1][i][j],
                                           D[e-1][i][j+(1<<(e-1))]);
                }
/// interogăm D și obținem în L cmmdc pe secvențe
/// de lungime w, pe linii
        for (i=1; i<=m; i++)
            for (j=1; j+w-1<=n; j++)
            {
                e = E[w];
                L[i][j] = cmmdc(D[e][i][j], D[e][i][j+w - (1<<e)]);
            }
```

```
/// aplicăm asupra lui L (care practic a contractat
/// rezultatul de pe w coloane în una singură) același
/// raționament dar pe coloane
/// obținem în D cmmdc pe secvențe din L
/// de lungime putere de 2, pe coloane
        for (i=1; i<=m; i++)
            for (j=1; j+w-1<=n; j++)
                D[0][i][j] = L[i][j];
/// interogând noul D pe secvențe de lungime h
/// și luând maximul dintre rezultate,
/// obținem rezultatul
        for (e=1; (1<<e) <= m; e++)
            for (j=1; j+w-1<=n; j++)
            {
                for (i=1; i<=m; i++)
                {
                    D[e][i][j] = D[e-1][i][j];
                    if (i+(1<<(e-1)) <= m)
                        D[e][i][j] = cmmdc(D[e][i][j],
                                           D[e-1][i+(1<<(e-1))][j]);
                }
            }

        for (j=1; j+w-1<=n; j++)
        {
            for (i=1; i+h-1 <= m; i++)
            {
                e = E[h];
                sol = maxim (sol, cmmdc(D[e][i][j],
                                        D[e][i+h- (1<<e)][j]));
            }
        }
        fprintf(fout,"Case #%d: %d\n",t,sol);
    }

    return 0;
}
```

4.4 Probleme propuse

- Problema Divquery
- Problema Minisecvențe
- Problema Consecutive1
- Problema Excursie1
- Problema Cârtița
- Problema Pian

4.5 Bibliografie

[1] *Range Minimum Query (Square Root Decomposition and Sparse Table)*, URL: https://www.geeksforgeeks.org/range-minimum-query-for-static-array/.
[2] *Range Minimum Query*, URL: https://www.infoarena.ro/problema/rmq.

Partea a II-a

Tehnici de programare

Capitolul 5

Tehnica Two Pointers

instr. Ștefan Dăscălescu
stefdasca.ro
Centrul Județean de Excelență Neamț

5.1 Introducere

Tehnica **Two pointers** este o tehnică pe care o putem folosi în foarte multe tipuri de probleme în care avem de căutat subsecvențe cu diverse proprietăți, condiția principală fiind aceea că vrem să găsim o pereche de valori sau de indici ce respectă anumite reguli, fără să depășim o anumită valoare sau o anumită condiție. Această tehnică apare în foarte multe tipuri de probleme ce se dau la concursurile de informatică, de foarte multe ori reprezentând o optimizare la posibile soluții cu căutare binară sau alte structuri de date ce ar adăuga un factor de complexitate în plus la soluție.

În articol voi începe prin a explica tipurile de probleme unde putem folosi **Two pointers**, urmând ca apoi să prezint câteva probleme de diverse dificultăți, explicând principalele strategii de abordare a acestor tipuri de probleme și punând accentul și pe implementări clare, care au drept scop evitarea greșelilor tipice când vine vorba de implementarea acestei metode.

Pentru a folosi această metodă, e nevoie să stăpânim lucrul cu secvențe și ideal și căutarea binară, deoarece pentru multe dintre exemplele ce vor fi menționate, există soluții și folosind acest algoritm. Nu în ultimul rând, pentru anumite probleme e posibil să fie nevoie de structuri de date adiționale, cum ar fi `map` sau `set`.

În ceea ce privește modul în care începem implementările, avem două tipuri majore de implementări, în funcție de algoritm. Merită menționat faptul că acești pointeri sunt de fapt variabile corespunzătoare indicilor din vector la care ne aflăm la momentul respectiv.

În primul rând, vorbim de problemele în care vrem să plecăm de la prima poziție și să procesăm secvențele care respectă o anumită proprietate monotonă (crescătoare sau descrescătoare). În acest caz, vom avea ambii pointeri cu indexul de început de la 1 și vom avansa cu pointerul din dreapta atâta timp cât încă se mai respectă condiția cerută din enunț.

De asemenea, mai există probleme în care plecăm cu pointerul stâng de la prima poziție și cu pointerul drept de la ultima poziție și vrem să mergem cu acești pointeri în direcții opuse, deoarece căutăm o proprietate ce are o variație descrescătoare față de scopul problemei.

5.2 Aplicații

5.2.1 Problema Subarray Sums

Dat fiind un vector cu n elemente numere naturale, determinați numărul de subsecvențe din vector pentru care suma elementelor este egală cu x.

Date de intrare

De pe prima linie se citesc valorile n și x. Pe cea de a doua linie se află n numere naturale, separate prin spațiu, $a_1, a_2, \ldots, a_n$ reprezentând elementele vectorului.

Date de ieșire

Afișați numărul de subsecvențe cu suma elementelor egală cu x.

Restricții și precizări

- $1 \leq n \leq 200\,000$
- $1 \leq x, a_i \leq 10^9$ pentru $1 \leq i \leq n$

Exemple

Intrare	Ieșire
5 7 2 4 1 2 7	3

Soluție

Pentru a rezolva această problemă, trebuie să ne folosim de faptul că toate numerele din șirul dat sunt pozitive. Astfel, vom putea afla pentru fiecare poziție de început a șirului, care este poziția cea mai din dreapta pentru care suma valorilor din acel interval să fie mai mică sau egală cu x. Dacă acea sumă este egală cu x, vom incrementa răspunsul. Vom avea grijă la fiecare pas, înainte de a incrementa variabila st, să scădem valoarea a_st din suma curentă.

Implementare

```
#include <iostream>
using namespace std;
int n, v[200002], x, st = 1, dr = 1, sum, ans;
int main()
{
  cin >> n >> x;
```

```
  for (int i = 1; i <= n; i++)
    cin >> v[i];
  while (st <= n)
    {
      while (dr <= n && sum < x)
        {
          sum += v[dr];
          dr++;
        }
      if (sum == x)
        ans++;
      sum -= v[st];
      st++;
    }
  cout << ans;
  return 0;
}
```

5.2.2 Problema Sum of Two Values

Se dă un vector cu n valori pozitive și o valoare x. Scrieți un program care să determine două valori aflate pe poziții distincte care adunate să dea suma x.

Date de intrare

De la intrare se citesc valorile n și x, apoi o succesiune de n numere reprezentând elementele vectorului.

Date de ieșire

Afișați pe ecran două numere separate prin spațiu, reprezentând pozițiile celor două valori determinate. Dacă există mai multe soluții, poate fi afișată oricare dintre acestea. Dacă nu există nicio soluție, afișați mesajul `IMPOSSIBLE`.

Restricții și precizări

- $1 \leq n \leq 2 \cdot 10^5$
- $1 \leq x, a_i \leq 10^9$ pentru $1 \leq i \leq n$

Exemple

Intrare	Ieșire
4 8 2 7 5 1	2 4

Soluție

Pentru a rezolva această problemă, trebuie să folosim o variantă diferită a tehnicii celor doi pointeri. Astfel, de data asta vom începe cu pointerul p_1 de la poziția 1 și cu pointerul p_2 la poziția n. Pe parcurs, vom avea trei cazuri în funcție de suma $a_{p_1} + a_{p_2}$. Dacă găsim

două poziții cu suma valorilor egală cu x, afișăm pozițiile respective, altfel modificăm p_1 sau p_2 după caz. Dacă nu găsim nicio soluție, afișăm `IMPOSSIBLE`.

Implementare

```
#include <iostream>
#include <algorithm>
using namespace std;
int n, x, a[200002], b[200002];
int main()
{int i;
  cin >> n >> x;
  for (i = 1; i <= n; ++i)
    {
      cin >> a[i];
      b[i] = a[i]; // avem nevoie de un alt vector pentru a afla pozițiile inițiale
    }

  sort(a+1, a+n+1); // sortăm șirul

  int p1 = 1, p2 = n, valA, valB;

  while (p1 < p2)
    {
      if (a[p1] + a[p2] == x)  //dacă am găsit suma, aflăm pozițiile valorilor
        {
          valA = a[p1];
          valB = a[p2];
          for (i = 1; i <= n; ++i)
            {
              if (b[i] == valA)
                {
                  cout << i << " ";
                  valA = 0;
                }
              else
                {
                  if (b[i] == valB)
                    {
                      cout << i << " ";
                      valB = 0;
                    }
                }
            }
          // ieșim din program ca să evităm afișarea mai multor soluții
          return 0;
        }
      else
        {
          // dacă suma e mai mare decât x, scădem p2, altfel creștem p1
          if (a[p1] + a[p2] > x)
            --p2;
          else
            ++p1;
        }
    }
```

```
  cout << "IMPOSSIBLE";
  return 0;
}
```

5.2.3 Problema Nane

Nane de pe Jiu, mare algoritmician fiind, vă provoacă să rezolvați o problemă prea ușoară pentru el. Nane vă dă N numere naturale și un număr K. Numim *subsecvență specială* o subsecvență pentru care efectuând operația OR pe biți pentru elementele din subsecvență (să numim această operație sumă OR) obținem un rezultat care are, în reprezentare binară, cel mult K biți de 1. Două subsecvențe sunt diferite dacă cel puțin o poziție din una nu se regăsește în cealaltă. Scrieți un program care să determine numărul de subsecvențe speciale.

Date de intrare

Fișierul de intrare `nane.in` va conține pe prima linie numerele N și K, iar pe a doua linie N numere naturale.

Date de ieșire

În fișierul de ieșire `nane.out` se va afla pe prima linie un singur număr Nr, reprezentând numărul de subsecvențe speciale.

Restricții și precizări

- $1 \leq N \leq 10^5$
- $1 \leq K \leq 30$
- Toate cele N numere vor fi naturale și se pot reprezenta pe 30 de biți.

Exemple

nane.in	nane.out
5 2 2 14 3 2 10	6

Soluție

Pentru a rezolva această problemă, vom folosi metoda celor doi pointeri pentru a afla numărul de secvențe care au suma OR cu cel mult K de 1, actualizările fiind foarte similare cu cele de la celelalte probleme de acest tip. De asemenea, deoarece vorbim de suma OR, trebuie să folosim câte un vector de frecvență pentru fiecare bit pentru a evita calculele adiționale.

Implementare

```
#include<bits/stdc++.h>
#define ll long long
using namespace std;

int n, k, v[100002], fr[32], st = 1, dr = 1;
ll ans;

bool ok()
{int i, cnt = 0;
 for (i = 0; i <= 30; ++i)
      if (fr[i])
          ++cnt;
 if (cnt <= k)
    return 1;
 return 0;
}

void add(int poz, int val)
{
 for (int i = 0; i <= 30; ++i)
     if ((v[poz] & (1<<i)))
         fr[i] += val;
}

int main()
{
 ifstream cin("nane.in");
 ofstream cout("nane.out");
 cin >> n >> k;
 for (int i = 1; i <= n; ++i)
      cin >> v[i];
 while (st <= n)
       {
        while (st <= n && (dr > n || !ok()))
              {
               if (ok())
                  {
                   ans += dr - st;
                   add(st, -1);
                   ++st;
                  }
                  else
                  {
                   add(st, -1);
                   ++st;
                   ans += dr - st;
                  }
               }
        while (dr <= n && ok())
              add(dr, 1), ++dr;
        }
 cout << ans << '\n';
 return 0;
}
```

5.2.4 Problema JJOOII

Se consideră un șir format din N caractere din mulțimea $\{J, O, I\}$. Se numește *JOI-șir* de nivel K un șir format din K litere J, K litere O și K litere I (în această ordine). De exemplu, $JJOOII$ este un *JOI-șir* de nivel 2. Bitaro dorește să transforme șirul S într-un *JOI-șir* de nivel K, utilizând următoarele 3 operații, de oricâte ori și în orice ordine:

- Operația 1: Bitaro șterge primul caracter din S;
- Operația 2: Bitaro șterge ultimul caracter din S;
- Operația 3: Bitaro șterge un caracter din interiorul lui S (care nu este nici primul nici ultimul).

Deoarece operațiile de tip 3 necesită mult timp, Bitaro dorește să transforme șirul S într-un *JOI-șir* de nivel K folosind un număr minim de operații de tip 3.

Scrieți un program care, cunoscând N, S și K, determină numărul minim de operații de tip 3 necesare pentru a transforma șirul S într-un *JOI-șir* de nivel K. Dacă acest lucru nu este posibil, programul va afișa valoarea `-1`.

Date de intrare

De la intrare se vor citi de pe prima linie numerele naturale N și K, iar de pe a doua linie șirul S.

Date de ieșire

La ieșire veți afișa o singură linie, pe care va fi scris rezultatul la cerința din enunț.

Restricții și precizări

- $3 \leq N \leq 200\,000$
- $1 \leq K \leq N/3$

Exemple

Intrare	Ieșire	Explicații
`10 2` `OJIJOIOIIJ`	`2`	Se poate obține un *JOI-șir* de nivel 2 executând următoarele operații: • Operația 1: șirul S devine $JIJOIOIIJ$. • Operația 2: șirul S devine $JIJOIOII$. • Operația 3: se elimină al doilea caracter și șirul S devine $JJOIOII$. • Operația 3: se elimină al patrulea caracter și șirul S devine $JJOOII$. Este imposibil să obținem un *JOI-șir* de nivel 2 utilizând mai puțin de două operații de tip 3.

Soluție

Pentru a rezolva această problemă, va trebui mai întâi să aflăm unde sunt situate literele J, O și I în șirul S. Ulterior, pe măsură ce fixăm secvențele de K de J, avem pointeri care duc la secvențele corespunzătoare de O și I din celelalte două șiruri, calculele ulterioare devenind destul de ușoare.

Implementare

```
#include <bits/stdc++.h>
using namespace std;
int vj[200001], vo[200001], vi[200001];
int lj, lo, li;
int main()
{int n, k, i;
  cin >> n >> k;
  string s;
  cin >> s;
  for (i = 0; i < n; i++)
    {
      if (s[i] == 'J')
        vj[++lj] = i;
      if (s[i] == 'O')
        vo[++lo] = i;
      if (s[i] == 'I')
        vi[++li] = i;
    }
  int pj = 1, po = 1, pi = 1, ans = n+1;
  while (pj + k - 1 <= lj && po + k - 1 <= lo && pi + k - 1 <= li)
    {
      while (po + k - 1 <= lo && vo[po] <= vj[pj + k - 1])
        po++;
      if (po + k - 1 <= lo)
        while (pi + k - 1 <= li && vi[pi] <= vo[po + k - 1])
          pi++;
      if (pj + k - 1 <= lj && po + k - 1 <= lo && pi + k - 1 <= li)
        {
          int fi = vj[pj];
          int lst = vi[pi + k - 1];
          ans = min(ans, (lst - fi + 1) - 3 * k);
        }
      pj++;
    }
  if (ans == n+1)
    cout << -1;
  else
    cout << ans;
  return 0;
}
```

5.3 Probleme propuse

- Categoria Two pointers pe Kilonova

- Categoria Two pointers pe Codeforces
- Cursul Two pointers pe Codeforces (necesită înregistrare; include 19 probleme)
- Problema 3secv
- Problema JJOOII 2
- Problema Global Warming
- Problema Șirbun
- Problema Nane
- Problema Social Distancing
- Problema MooTube
- Problema Wormhole Sort
- Problema Sprinklers
- Problema Cow Dating

5.4 Bibliografie

[1] Darren Yao, Qi Wang și Ryan Chou, *Two Pointers*, URL: https://usaco.guide/silver/two-pointers?lang=cpp.

[2] Antti Laaksonen, *Competitive Programmer's Handbook*, 2018, cap. I.8.1, URL: https://cses.fi/book/book.pdf.

[3] *The Two Pointer Technique*, URL: https://algodaily.com/lessons/using-the-two-pointer-technique.

Capitolul 6

Tehnica „Meet in the middle”

prof. Eugen Nodea
Colegiul Național „Tudor Vladimirescu” Târgu-Jiu
Centrul Județean de Excelență Gorj

6.1 Introducere

Meet in the middle este o tehnică de programare folosită atunci când datele de intrare sunt relativ mici, dar nu suficient de mici ca să poți folosi brute-force. Asemănător metodei **Divide et impera**, metoda **Meet in the middle** împarte problema în două subprobleme (nu obligatoriu de același tip) pe care le rezolvă individual pentru a combina apoi rezultatele.

6.2 Aplicații

6.2.1 Problema MITM

Se consideră un șir S de n numere întregi și un număr natural SUM. Să se determine suma maximă posibilă, mai mică sau egală cu SUM, care se poate obține ca sumă a elementelor dintr-un subșir al șirului S.

Date de intrare

Programul citește de la tastatură numărul n și SUM, apoi n numere naturale, separate prin spații, reprezentând elementele șirului.

Date de ieșire

Programul va afișa pe ecran numărul $smax$, reprezentând suma maximă posibilă, mai mică sau egală cu SUM ce se poate obține ca sumă a elementelor unui subșir al șirului dat.

Restricții și precizări

- $3 \leq n \leq 40$
- $0 \leq SUM \leq 10^{18}$
- $-10^{12} \leq S_i \leq 10^{12}$, pentru $1 \leq i \leq n$

Exemple

Intrare	Ieșire	Explicații
5 20 5 10 6 8 3	19	Suma maximă mai mică sau egală decât 20 este 19 și se formează din 10, 6 și 3.

Soluție

O abordare brute-force ar presupune generarea tuturor sumelor subșirurilor de elemente din S și determinarea subșirului de sumă maximă, mai mică sau egală cu SUM.

Complexitatea timp a unei astfel de abordări ar fi de $\mathcal{O}(2^n)$.

Ce se întâmplă dacă împărțim șirul S în două șiruri A și B cu câte $n/2$ elemente fiecare? Problema se reduce la determinarea tuturor sumelor subșirurilor de elemente ale șirului A și reținerea lor într-un vector, fie el X, vector ce conține $2^{n/2}$ elemente. În mod similar procedăm și cu șirul B, reținând sumele în vectorul Y. Complexitatea timp a fost astfel redusă la $\mathcal{O}(2^{n/2})$.

Vom combina rezultatele reținute în cei doi vectori pentru a găsi suma maximă mai mică sau egală cu SUM. Pentru aceasta, sortăm crescător vectorul Y care reține sumele generate pentru subșirul B. Vom lua fiecare element x din vectorul X și căutăm binar în vectorul Y elementul y maxim astfel încât $x + y \leq SUM$.

Căutarea binară reduce astfel complexitatea timp de la $\mathcal{O}(2^n)$ la $\mathcal{O}(2^{n/2} \log 2^{n/2})$, care de fapt este $\mathcal{O}(n \cdot 2^{n/2})$.

Implementare

```
#include <bits/stdc++.h>
#define nmax 1050000
using namespace std;

long long X[nmax], Y[nmax], A[21], B[21], SUM;

/// generare submultimi
void genset(long long v[], long long x[], int n)
{
  for (int i = 0; i < (1<<n); i++)
    {
      long long s = 0;
      for (int j = 0; j < n; j++)
        if (i & (1 << j))
          s += v[j];
      x[i] = s;
    }
```

```
}
int main()
{
  int n, i, n1, n2;

  cin >> n >> SUM;

  // împărțim setul în două subseturi A și B
  n1 = n / 2;
  n2 = n - n / 2;
  for(i = 0; i < n1; i++)
    cin >> A[i];
  for(i = 0; i < n2; i++)
    cin >> B[i];

  genset(A, X, n1);
  genset(B, Y, n2);

  n1 = 1 << (n1);
  n2 = 1 << (n2);

  sort(Y, Y+n2);

  long long smax = 0;
  for (int i = 0; i < n1; i++)
    {
      if (X[i] <= SUM)
        {
          int st = 0, dr = n2 - 1, mid;
          while (st < dr)
            {
              mid = (st + dr + 1) / 2;
              if (Y[mid] + X[i] <= SUM)
                st = mid;
              else
                dr = mid - 1;
            }
          if (X[i] + Y[st] <= SUM)
            smax = max(smax, X[i] + Y[st]);
        }
    }

  cout << smax;
  return 0;
}
```

6.2.2 Problema Triplete

Se consideră un șir A de n numere întregi. Să se determine dacă există un triplet (A_i, A_j, A_k) care să verifice teorema lui Pitagora.

Restricții și precizări

- $1 \leq n \leq 10^5$

- $1 \leq A_i \leq 10^9$, pentru $1 \leq i \leq n$

Exemple

Intrare	Ieșire	Explicații
6 3 10 4 6 13 8	DA	Alegem tripletul 10, 6 și 8.

Soluție

Abordarea brute-force (naivă): Calculăm toate combinațiile tuturor tripletelor ce se pot forma cu elementele din tablou. Complexitate $\mathcal{O}(n^3)$.

Abordarea eficientă: Vom optimiza abordarea naivă folosind tehnica **Meet in the middle** pentru a obține o complexitate de calcul mai bună.

- Construim un alt vector cu pătratele elementelor $B_i = A_i * A_i$, pentru $1 \leq i \leq n$.
- Sortăm crescător vectorul B.
- Folosim trei indici i, j, k (three pointers).
- Fixăm valoarea cea mai mare a formulei lui Pitagora prin indicele i, fie acesta B_i.
- Vom încerca să găsim elementele $B_j + B_k = B_i$ plecând cu indicii $j = 0$, $k = i - 1$.

Complexitate: $\mathcal{O}(n^2)$.

Implementare

```
#include <algorithm>
#define NMAX 100002
int A[NMAX], B[NMAX];
int n;
bool triplet()
{
  int i, j, k;
  for (i = 0; i < n; i++)
    B[i] = A[i] * A[i];

  std::sort(B, B+n);

  for (i = n - 1; i >= 2; i--)
    {
      j = 0, k = i-1;
      while (j < k)
        {
          if (B[j] + B[k] == B[i])
            return true;
          if (B[j] + B[k] < B[i])
            j++;
          else
            k--;
        }
    }
```

```
  return false;
}
```

6.2.3 Problema Maximum Subsequence

Se consideră un șir A de n numere întregi și o valoare întreagă m. Să se determine un subșir al șirului A, având indicii $b_1, b_2, \ldots, b_k$ ($1 \leq b_1 < b_2 < \ldots < b_k \leq n$) astfel încât suma valorilor din subșir modulo m să fie maximă. Subșirul determinat poate fi vid.

Date de intrare

Programul citește de la tastatură numerele n și m, apoi n numere naturale, separate prin spații, reprezentând elementele șirului.

Date de ieșire

Programul va afișa pe ecran numărul $smax$, reprezentând suma modulo m a valorilor din subșirul determinat (maximă posibil).

Restricții și precizări

- $1 \leq n \leq 35$
- $1 \leq m \leq 10^9$
- $1 \leq A_i \leq 10^9$, pentru $1 \leq i \leq n$

Exemple

Intrare	Ieșire	Explicații
4 4 5 2 4 1	3	Alegem subșirul cu indicii 1 și 2 având suma elementelor egală cu 7; 7 mod 4 este 3, maxim posibil.

Soluție

O abordare brute-force ar presupune să generăm toate subșirurile șirului A, să calculăm sumele modulo m și să alegem maximul. O astfel de abordare are complexitatea $\mathcal{O}(n \cdot 2^n)$.

O abordare care utilizează tehnica **Meet in the middle** ar presupune să împărțim șirul A în două subșiruri cu $n/2$ elemente, respectiv $n - n/2$ elemente și să generăm pentru fiecare subșir toate sumele modulo m ale subșirurilor lor, sume pe care le vom reține în doi vectori x și y. Sortăm vectorii x și y.

Deoarece suma a două numere, unul aflat în vectorul x, celălalt în vectorul y, nu poate depăși $2m$, pentru determinarea soluției vom fixa un element din x, fie acesta x_{st} și vom căuta în y cel mai mare element care este mai mic decât $m - x_{st}$.

Complexitatea timp a fost astfel redusă la $\mathcal{O}(n \cdot 2^{n/2})$.

Implementare

```
#include <bits/stdc++.h>
using namespace std;

int a[40], n, m, ans;
int x[1 << 20];
int y[1 << 20];

void gen(int a[], int n, int b[])
{int i;
  for (i = 0; i < n; i++) b[1 << i] = a[i];

  for (i = 0; i < (1 << n); i++)
    b[i] = (b[i - (i & -i)] + b[i & -i]) % m;
}
int main()
{int i;
  cin >> n >> m;
  for (i = 0; i < n; i++)
    {
      cin >> a[i];
      a[i] %= m;
    }
  int n1 = n / 2;
  int n2 = n-n1;
  gen(a, n1, x);
  gen(a + n1, n2, y);

  sort(x, x + (1 << n1));
  sort(y, y + (1 << n2));

  int st, dr = (1 << n2) - 1;
  for (st = 0; st < (1 << n1); st++)
    {
      while (x[st] + y[dr] >= m)
        {
          dr--;
        }
      ans = max(ans, x[st] + y[dr]);
    }

  cout << ans;
  return 0;
}
```

6.2.4 Problema Prime Gift

Se consideră un șir de n numere prime distincte ordonate crescător $P = (P_1, P_2, .., P_n)$ și un număr natural nenul k.

Cerință

Să se determine al k-lea cel mai mic număr natural astfel încât toți factorii primii ai săi să se găsească în șirul P.

Date de intrare

Pe prima linie se află numărul natural n, pe cea de a doua linie se află cele n numere prime din șirul P, iar pe ultima linie se află valoarea k.

Date de ieșire

Afișați pe ecran al k-lea cel mai mic număr astfel încât toți factorii primii ai numărului să se găsească în șirul P.

Restricții și precizări

- $1 \leq n \leq 16$
- $1 \leq P_i \leq 100$, pentru $1 \leq i \leq n$
- Se garantează că rezultatul nu depășește 10^{18}.
- Numerotarea se face începând cu 1.

Exemple

Intrare	Ieșire	Explicații
3 2 3 5 7	8	Lista primelor 7 numere care au toți factorii primi în șirul 2,3,5 este 1,2,3,4,5,6,8

Soluție

Este ușor de observat că toate aceste numere pot fi reprezentate ca $P_1^{e_1} \cdot P_2^{e_2} \cdot \cdots \cdot P_n^{e_n}$.

Notăm cu $D(P)$ mulțimea numerelor care nu depășesc 10^{18}, numere pentru care toți factorii primi se află în mulțimea P.

Vom împărți mulțimea P în două submulțimi disjuncte A și B. Deoarece fiecare element al mulțimii $D(P)$ poate fi reprezentat ca produs dintre un element din $D(A)$ și un element din $D(B)$, generăm mulțimile $D(A)$ și $D(B)$, apoi sortăm cele două mulțimi. Pentru a determina al k-lea cel mai mic element care are toți factorii primi în P, putem face o căutare binară pe rezultat, folosind funcția `get()` care determină numărul de produse dintre un element din $D(A)$ și un element din $D(B)$ care nu depășesc o valoare specificată x.

Cum găsim o partiționare optimă a mulțimii P? Prima idee este de a plasa primele $n/2$ elemente din P în mulțimea A și restul în mulțimea B. Cu toate acestea, acest lucru nu este eficient pentru că dimensiunea aproximativă a mulțimii $D(A)$ ar putea ajunge la $7 \cdot 10^6$, deci este prea mare.

Pentru a reduce dimensiunea mulțimii $D(A)$, putem selecta în A elementele cu indici pari din P, iar în mulțimea B elementele cu indici impari. Astfel dimensiunile mulțimilor $D(A)$ și $D(B)$ nu depășesc 10^6.

Complexitate $\mathcal{O}(\log 10^{18} \cdot (|D(A)| + |D(B)|))$.

Implementare

```
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long int ull;
const int MaxN = 20;
int n;
int p[MaxN];
ull k, ans;
vector <ull> a, b, A, B;

ull get(vector <ull> & a, vector <ull> & b, ull x)
{
  int p = a.size() - 1;
  ull res=0;
  for (int i = 0; i < b.size(); i++)
    {
      while (p >= 0 && b[i] * a[p] > x)
        --p;
      res += p + 1;
    }
  return res;
}

void precalc(vector <ull> & A, vector <ull> & a, ull crt, int pos)
{
  if (pos == a.size())
    {
      A.push_back(crt);
      return;
    }
  do
    {
      precalc(A, a, crt, pos + 1);
      long double val = 1.0 * crt * a[pos];
      if(val <= 1e18)
        crt *= a[pos];
      else
        break;
    }
  while (1);
}

int main()
{
  cin >> n;
  for (int i = 1; i <= n; i++)
    {
      cin >> p[i];
      if(i & 1)
        a.push_back(p[i]);
      else
        b.push_back(p[i]);
    }

  precalc(A, a, 1, 0);
  precalc(B, b, 1, 0);
```

```
  sort(A.begin(), A.end());
  sort(B.begin(), B.end());

  cin >> k;
  ull st = 1, dr = 1e18;
  while(st <= dr)
    {
      ull mid = (st + dr) / 2;
      if( get(A, B, mid) >= k )
        {
          ans = mid;
          dr = mid - 1;
        }
      else
        st = mid + 1;
    }

  cout << ans;
  return 0;
}
```

6.3 Probleme propuse

- Problema Piese
- Problema Colecție
- Problema Ecuații
- Problema Subset Sums
- Problema CWC 2015
- Problema Robot instructions
- Problema Meet in the middle
- Problema Shortest Path using Meet In The Middle
- Problema SplitAndMergeGame
- Problema Anagram division

6.4 Bibliografie

[1] Cosmin Negruşeri, *Coding contest trick: Meet in the middle*, URL: https://infoarena.ro/blog/meet-in-the-middle.

[2] *Meet in the middle*, URL: https://www.geeksforgeeks.org/meet-in-the-middle/.

[3] *Baby-step giant-step*, URL: https://en.wikipedia.org/wiki/Baby-step_giant-step.

Capitolul 7

Square root decomposition

prof. Emanuela Cerchez
Colegiul Național „Emil Racoviță” Iași
Centrul Județean de Excelență Iași

stud. Andrei Onuț
Yale University
Centrul Județean de Excelență Prahova

7.1 Introducere

Square root decomposition este o tehnică de programare care constă în divizarea unei structuri de date de dimensiune $\mathcal{O}(N)$ în blocuri de dimensiune $\mathcal{O}(\sqrt{N})$ și prelucrarea fiecărui bloc în scopul determinării unei soluții pentru întreaga structură.

7.2 Aplicații

7.2.1 Problema Sume

Fie A un tablou cu N elemente. Asupra acestui tablou se vor executa două tipuri de operații:

- Actualizare (*update*): *U poz val*, cu semnificația „valoarea de pe poziția *poz* devine *val*”;
- Interogare (*query*): *Q st dr*, cu semnificația „să se determine suma elementelor din segmentul $[st, dr]$”.

Segmentul $[st, dr]$ este format din elementele $A_{st}, A_{st+1}, \ldots A_{dr}$ (st este extremitatea inițială, iar dr cea finală); poziția i aparține segmentului $[st, dr]$ dacă $st \leq i \leq dr$.

Se citește o succesiune de nr operații și se cere afișarea rezultatelor obținute la interogări (pe linii diferite).

Soluție

Evident, această problemă poate fi rezolvată cu arbori indexați binar sau cu arbori de intervale dar, fiind cea mai simplă posibilă, o utilizăm pentru exemplificarea acestei tehnici.

Vom diviza tabloul A în blocuri de dimensiune $BLOCK\_SIZE = sqrt(N)$, unde $sqrt(x)$ reprezintă cel mai mare număr întreg mai mic sau egal cu $\sqrt{x}$. Pentru simplitatea calculelor, vom numerota atât pozițiile în vectorul A, cât și blocurile obținute începând de la 0. Fie K indicele ultimului bloc. Fiind numerotate de la 0, vor exista $K + 1$ blocuri. Este posibil ca ultimul bloc să aibă mai puțin de $BLOCK\_SIZE$ elemente.

Blocul r $(0 \leq r \leq K)$ este segmentul

$$[r \times BLOCK\_SIZE, \min(N - 1, (r + 1) \times BLOCK\_SIZE - 1)]$$

Demonstrația se poate face ușor prin inducție). Două poziții i și j aparțin aceluiași bloc r dacă și numai dacă $i/BLOCK\_SIZE = j/BLOCK\_SIZE = r$.

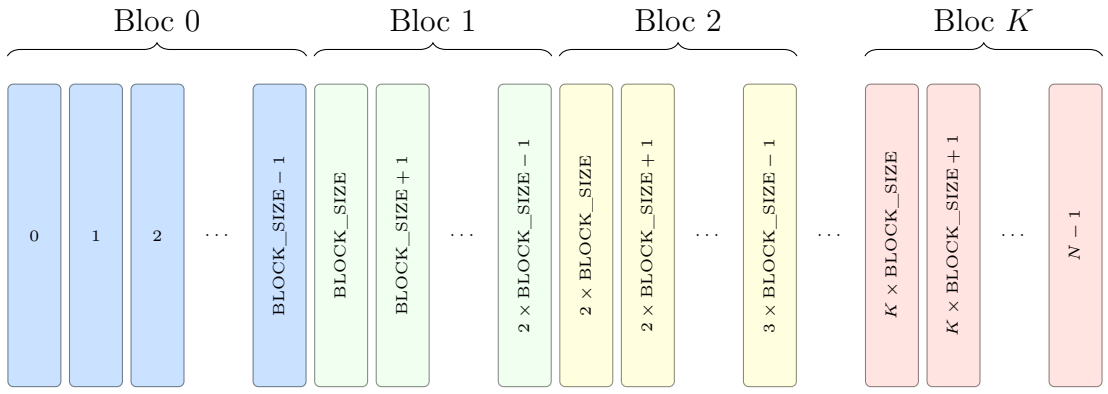

Figura 7.1: Un vector de N elemente descompus în blocuri de lungime BLOCK_SIZE.

Când utilizăm tehnica **Square root decomposition**, pentru fiecare bloc reținem informații suplimentare, utile pentru determinarea rezultatului. În acest caz reținem suma elementelor din blocul respectiv.

Atunci când efectuăm o operație de actualizare, trebuie să modificăm informațiile corespunzătoare blocului în care se află valoarea modificată. În acest caz, suma blocului în care se află poziția *poz* va scădea cu valoarea eliminată și va crește cu valoarea adăugată, complexitatea fiind $\mathcal{O}(1)$. Atunci când efectuăm o operație de interogare, cumulăm sumele blocurilor aflate integral în intervalul $[st, dr]$ (cel mult $\sqrt{N}$ sume), precum și eventualele elementele din blocurile incomplete aflate la începutul, respectiv la sfârșitul intervalului. Deoarece într-un bloc incomplet pot fi cel mult $\sqrt{N}$ elemente, complexitatea totală a unei operații de interogare va fi $\mathcal{O}(\sqrt{N})$.

Implementare

```
#include <bits/stdc++.h>
#define NMAX 100002
```

```cpp
#define SQMAX 1002
using namespace std;
ifstream fin("date.in");
ofstream fout("date.out");

int n, nrb;
int A[NMAX];      // vectorul dat
int sumb[SQMAX];  // tablou cu sumele calculate pe blocuri
int BLOCK_SIZE;   // dimensiunea blocurilor

// Complexitate : O(1)
void update(int poz, int val)
{
    int blockNumber = poz / BLOCK_SIZE;
    sumb[blockNumber] += val - A[poz];
    A[poz] = val;
}

// Complexitate: O(sqrt(n))
int query(int st, int dr)
{
    int sum = 0;
//parcurgem elementele până la începutul primului bloc
    while (st <=dr && st%BLOCK_SIZE != 0)
        sum += A[st++];
//parcurgem blocurile intregi
    while (st +  BLOCK_SIZE- 1 <= dr)
    {
        sum += sumb[st /BLOCK_SIZE];
        st += BLOCK_SIZE;
    }
//parcurgem restul elementelor
    while (st <= dr) sum += A[st++];
    return sum;
}

int main()
{
    int i, st, dr, val, poz, nr;
    char ch;
    fin>>n;
    for (i=0; i<n; i++) fin>>A[i];
    BLOCK_SIZE=(int) sqrt(n);
//calculăm sumele din fiecare bloc
    nrb=-1;
    for (i=0; i<n; i++)
    {
        if (i%BLOCK_SIZE==0) nrb++;
        sumb[nrb]+=A[i];
    }
//operații
    fin>>nr;
    for (i=0; i<nr; i++)
    {
        fin>>ch;
        if (ch=='U')
        {
```

```
            fin>>poz>>val;
            poz--;
            update(poz,val);
        }
        else
        {
            fin>>st>>dr;
            st--;
            dr--;
            fout<<query(st,dr)<<'\n';
        }
    }
    return 0;
}
```

7.2.2 Problema aib

Aveți la dispoziție un număr natural nenul n și o permutare $a = (a_1, a_2, \ldots a_n)$ a mulțimii $1, 2, \ldots n$.

Cerință

Pentru fiecare număr a_i trebuie să determinați câte numere mai mici decât a_i se află la stânga sa, adică în secvența $a_1, a_2, \ldots a_{i-1}$.

Date de intrare

Programul citește de la tastatură numărul n, iar apoi n numere naturale separate prin spații reprezentând permutarea.

Date de ieșire

Programul va afișa pe ecran pentru fiecare i, $1 \leq i \leq n$, câte numere mai mici decât a_i se află la stânga sa.

Restricții și precizări

- $3 \leq n \leq 100\,000$

Exemple

Intrare	Ieșire
7 3 1 6 5 2 7 4	0 0 2 2 1 5 3

Explicație

Sunt 0 numere mai mici decât 3 și aflate la stânga lui 3. Sunt 0 numere mai mici decât 1 și aflate la stânga lui 1 Sunt 2 numere mai mici decât 6 și aflate la stânga lui 6 (acestea sunt primele două numere din șir, adică 3 și 1).

Soluție

Numele problemei sugerează o abordare cu arbori indexați binar, dar, pentru a exemplifica tehnica denumită **Square root decomposition** vom aborda problema în acest mod.

Citim succesiv valorile din permutare și construim un vector caracteristic v ($v[x] = 1$ dacă valoarea x a fost deja citită). Când citim valoarea x suma elementelor de pe poziții $< x$ în vectorul v reprezintă numărul de elemente mai mici decât x situate înaintea sa. Pentru determinarea acestei sume vom proceda într-un mod similar cu problema precedentă.

Implementare

```
#include <bits/stdc++.h>
#define NMAX 100002
#define SQMAX 1002
using namespace std;
int n, nrb;
int A[NMAX];       // vectorul dat
int sumb[SQMAX];  // tablou cu sumele calculate pe blocuri
int v[NMAX];       // vectorul caracteristic
int BLOCK_SIZE;   // dimensiunea blocurilor

// Complexitate: O(1)
void update(int poz, int val)
{
    int blockNumber = poz / BLOCK_SIZE;
    sumb[blockNumber] += val - v[poz];
    v[poz] = val;
}

// Complexitate: O(sqrt(n))
int query(int st, int dr)
{
    int sum = 0;
//parcurgem elementele până la începutul primului bloc
    while (st <=dr && st%BLOCK_SIZE != 0)
        sum += v[st++];
//parcurgem blocurile întregi
    while (st +  BLOCK_SIZE- 1 <= dr)
    {
        sum += sumb[st /BLOCK_SIZE];
        st += BLOCK_SIZE;
    }
//parcurgem restul elementelor
    while (st <= dr) sum += v[st++];
    return sum;
}
int main()
{
    int i, x;
    cin>>n;
    for (i=0; i<n; i++) cin>>A[i];

    BLOCK_SIZE=(int) sqrt(n);
//calculăm sumele din fiecare bloc
    for (i=0; i<n; i++)
```

```
    {
        x=A[i];
        cout<<query(1,x-1)<<' ';
        update(x,1);
    }
    return 0;
}
```

7.2.3 Problema kth

Se dă un șir A ce conține N numere întregi numerotate începând de la 1, $A_1, A_2, \ldots A_N$ și două numere naturale nenule K și L, cu proprietatea că: $1 \leq K \leq L \leq N$. Mihai studiază doar secvențele de lungime L, adică secvențele formate din exact L elemente situate pe poziții alăturate în acest șir V. El își poate pune următoarea întrebare: „Dacă aș rearanja, în ordine crescătoare, elementele secvenței de lungime L care începe la poziția poz în șirul A, ce valoare s-ar afla pe a K-a poziție în cadrul secvenței rezultate?”. Pentru secvența din șir care începe la poziția poz și are L elemente, adică $A_{poz}, A_{poz+1}, \ldots A_{poz+L-1}$, valoarea elementului de pe a K-a poziție în cadrul secvenței este $A_{poz+K-1}$.

Cerință

Ajutați-l pe Mihai să afle care este răspunsul corect pentru Q întrebări de tipul descris mai sus!

Date de intrare

Pe prima linie a fișierului de intrare `kth.in` se află trei numere naturale nenule N, K și L, separate între ele prin câte un spațiu, cu semnificațiile de mai sus. Pe următoarea linie se află, separate între ele prin câte un spațiu, N numere întregi, reprezentând, în ordine, elementele șirului A. Pe următoarea linie se află numărul natural nenul Q, reprezentând numărul de întrebări formulate de către Mihai. Pe fiecare dintre următoarele Q linii se află câte un număr natural nenul poz, reprezentând poziția de început a secvenței de L elemente pentru care se pune întrebarea respectivă.

Date de ieșire

Fișierul de ieșire `kth.out` va conține Q linii. Pe linia i se va afla un număr întreg ce reprezintă răspunsul la întrebarea i, în ordinea dată în fișierul de intrare, pentru fiecare i: $1 \leq i \leq Q$.

Restricții și precizări

- $2 \leq N \leq 300\,000$
- $1 \leq Q \leq 300\,000$
- $-50\,000 \leq A_i \leq 50\,000$ pentru $1 \leq i \leq N$
- $1 \leq poz \leq N - L + 1$ pentru fiecare dintre cele Q întrebări.

- Valorile *poz* din cadrul celor Q întrebări nu sunt neapărat distincte între ele oricare două.

Exemple

kth.in	kth.out
5 2 3 4 -5 2 1 4 2 2 1	1 2

Explicație

Sunt $N = 5$ elemente în șirul $A = (4, -5, 2, 1, 4)$. Pentru prima întrebare (pentru care $poz = 2$), dacă secvența formată din $L = 3$ elemente: (A_2, A_3, A_4) ar fi ordonată crescător, aceasta ar deveni $(-5, 1, 2)$, ceea ce înseamnă că pe cea de a doua ($K = 2$) poziție în cadrul ei s-ar afla valoarea 1.

Soluție

O primă observație ar fi că valorile din șirul A sunt relativ mici, ca urmare putem construi un vector de frecvență $v[x]$=numărul de apariții ale valorii $x - 50000$ (am translat intervalul de valori [-50000,50000], pe intervalul de indici [0,100000]). Vom parcurge secvențele de lungime L în ordinea crescătoare a pozițiilor de început. Pentru prima secvență de lungime L determinăm vectorul de frecvență v. Atunci când trecem de la o secvență la următoarea, practic eliminăm din secvență un număr și adăugăm altul:

Secvența care începe la poziția poz	V_{poz}	V_{poz+1}	$\dots$	$V_{poz+L-1}$	
Secvența care începe la poziția $poz+1$		V_{poz+1}	$\dots$	$V_{poz+L-1}$	V_{poz+L}

Pentru a determina care este cea de a K-a valoare din secvența curentă dacă aceasta ar fi sortată vom însuma valorile din vectorul de frecvență până când identificăm valoarea pentru care suma a devenit $\geq K$. Pentru a face suma în timp $\mathcal{O}(\sqrt{N})$, putem utiliza tehnica **Square root decomposition**.

Implementare

```
#include <bits/stdc++.h>
#define DMAX 300002
#define SQMAX 402
#define OFFSET 50000
#define VMAX 50002
using namespace std;
ifstream fin("kth.in");
ofstream fout("kth.out");

int n, K, L;
int A[DMAX];          // vectorul dat
int sumb[SQMAX];      // tablou cu sumele calculate pe blocuri
```

```
int v[VMAX+OFFSET];  // vectorul de frecvență
int BLOCK_SIZE;      // dimensiunea blocurilor
int rez[DMAX];

// Complexitate: O(1)
void update(int poz, int val)
//valoarea poate fi +1 sau -1 - incrementăm sau decrementăm
{
    int blockNumber = poz / BLOCK_SIZE;
    sumb[blockNumber] += val;
    v[poz]+=val;
}

int kth()
//returnează al k-lea element din secvența curentă de lungime L
{
    int i=0, j, sum=0;
    for (i=0; sum+sumb[i]<K; i++) sum+=sumb[i];
//în acest moment sum+sumb[i]>=K
//al K-lea element se află în blocul i, format din elementele de pe pozițiile
//i*BLOCK_SIZE, min(N-1, (i+1)*BLOCK_SIZE-1)]
    for (j=i*BLOCK_SIZE; ; j++)
    {
        sum+=v[j];
        if (sum>=K) return j-OFFSET;
    }
    return 0;
}

int main()
{
    int i, poz, q, maxim=0;
    fin>>n>>K>>L;
    for (i=0; i<n; i++)
    {
        fin>>A[i];
        A[i]+=OFFSET;
        if (A[i]>maxim) maxim=A[i];
    }
    BLOCK_SIZE=(int) sqrt(maxim+1);
//construim vectorul de frecvență pentru prima secvență de lungime L
    for (i=0; i<L; i++) update(A[i],1);

    for (i=0; i<=n-L; i++)
    {
        //aflu rezultatul pentru secvența care începe la poziția i
        rez[i]=kth();
        //trec la secvența următoare
        update(A[i],-1);
        update(A[i+L], 1);
    }
//citim interogările și afișăm rezultatele
    fin>>q;
    for (i=0; i<q; i++)
    {
        fin>>poz;
        poz--;
```

```
        fout<<rez[poz]<<'\n';
    }
    return 0;
}
```

Soluție alternativă

O abordare simplă pentru această problemă este de a utiliza două structuri de tip `multiset`:

- *mink* – aici reținem cele mai mici K valori din segmentul curent de lungime L
- *maxLk* – aici reținem cele mai mari $L - K$ valori din segmentul curent de lungime L (restul).

Cel de al K-lea element din segmentul curent de lungime L sortat este ultimul din *mink*.

La trecerea de la un segment la următorul:

- eliminăm A_i (căutăm A_i în *mink* și dacă îl găsim, îl ștergem; dacă nu, în căutăm în *maxLk* și îl ștergem de acolo).
- adăugăm A_{i+L} în *mink* dacă A_{i+L} este $\leq$ decât cel mai mare element din *mink* (ultimul); în caz contrar îl adăugăm în *maxLk*.
- reechilibrăm numeric cele două multiseturi (*mink* trebuie să conțină exact K elemente).

Implementare

```
#include <bits/stdc++.h>
#define DMAX 300002
#define SQMAX 402
#define VMIN -50000
#define VMAX 50002
using namespace std;
ifstream fin("kth.in");
ofstream fout("kth.out");

int n, K, L;
int A[DMAX];      // vectorul dat
int rez[DMAX];
multiset<int> mink;
//reținem cele mai mici k valori din secvența curentă
multiset<int> maxLk;
//reținem cele mai mari L-k elemente din secvența curentă (restul)

int main()
{
    int i, poz, q;
    multiset<int>:: iterator it;
    multiset<int>:: reverse_iterator rit;
    fin>>n>>K>>L;
    for (i=0; i<n; i++) fin>>A[i];
    for (i=0; i<L; i++) maxLk.insert(A[i]);
```

```
    for (i=0; i<K; i++) //mut cele mai mici K elemente în mink
    {
        it=maxLk.begin();
        mink.insert(*it);
        maxLk.erase(it);
    }

    for (i=0; i<=n-L; i++)
    {
        //aflu rezultatul pentru secvența care începe la poziția i
        rit=mink.rbegin();
        rez[i]=*rit;
        //trec la secvența următoare
        //șterg A[i];
        it=mink.find(A[i]);
        if (it!=mink.end())
            mink.erase(it);
        else
        {
            it=maxLk.find(A[i]);
            maxLk.erase(it);
        }
        //adaug A[i+L];
        if (*rit>A[i+L]) mink.insert(A[i+L]);
        else
            maxLk.insert(A[i+L]);
        //echilibrăm
        while (mink.size()<K)
        {
            it=maxLk.begin();
            mink.insert(*it);
            maxLk.erase(it);
        }
        while (mink.size()>K)
        {
            rit=mink.rbegin();
            maxLk.insert(*rit);
            it=mink.find(*rit);
            mink.erase(it);
        }
    }
//citim interogările și afișăm rezultatele
    fin>>q;
    for (i=0; i<q; i++)
    {
        fin>>poz;
        poz--;
        fout<<rez[poz]<<'\n';
    }
    return 0;
}
```

7.2.4 Problema Toorcmmdc

Se dă o listă, inițial vidă, cu numere. Toor dorește să completeze lista aceasta, adăugând numere pe care le consideră importante pentru determinarea celui mai mare divizor comun (cmmdc). El efectuează asupra listei o succesiune de operații și, după fiecare operație, dorește să afle cmmdc-ul tuturor numerelor care sunt în listă la momentul respectiv. Operațiile sunt de următoarele două tipuri:

- $+X$ se adaugă numărul X în listă
- $-X$ se elimină numărul X din listă

Cerință

Cunoscând succesiunea operațiilor, să se determine, după efectuarea fiecărei operații, cmmdc-ul numerelor aflate în listă.

Date de intrare

Fișierul de intrare `toorcmmdc.in` conține pe prima un număr natural N. Pe următoarele N linii, se găsesc operațiile în forma descrisă în enunț.

Date de ieșire

Fișierul de ieșire `toorcmmdc.out` va conține N linii, pe care va fi scris cmmdc-ul determinat după efectuarea fiecăreia dintre cele N operații.

Restricții și precizări

- $1 \leq N \leq 100\,000$
- În fiecare operație $X \leq 10^9$
- Dacă un număr nu se află în listă, va fi ignorată ștergerea. Dacă se află de mai multe ori, va fi scos doar o singură dată.
- Prin convenție, cmmdc-ul unei mulțimi cu 0 elemente este 1.

Exemple

toorcmmdc.in	toorcmmdc.out
5 + 2 + 6 + 12 - 2 + 9	2 2 2 6 3

Explicație

După prima operație, lista are doar elementul 2. După a patra operație, lista are elementele 6, 12.

Soluție

Vom citi operațiile și vom reține într-un vector V valorile distincte care apar în aceste operații. În acest mod putem identifica fiecare valoare prin poziția sa în V (care este un număr natural cel mult egal cu 10^5).

Asociind fiecărei valori poziția sa, vom putea construi un vector de frecvență fr, unde $fr[i]$=numărul de apariții ale valorii $V[i]$ în vector.

Pentru a determina cmmdc-ul valorilor existente în listă la un moment dat, vom partiționa vectorul V în blocuri de dimensiune $DIM = \sqrt{N}$. Blocul 1 conține pozițiile $1, 2, \ldots DIM$; blocul al doilea conține pozițiile $DIM + 1, \ldots 2 \cdot DIM$; ...al i-lea bloc va conține pozițiile $(i-1) \cdot DIM + 1, \ldots i \cdot DIM$. Ultimul bloc poate fi incomplet.

Vom efectua operațiile în ordine. Dacă operația este de adăugare, verificăm dacă valoarea X este la prima apariție în listă și dacă da, determinăm blocul din care face parte această valoare și actualizăm cmmdc pentru acest bloc (acesta va fi cmmdc dintre X și cmmdc-ul curent al blocului).

Dacă operația este de eliminare, verificăm dacă există valoarea X în lista curentă și dacă se elimină singura ei apariție. În acest caz cmmdc-ul blocului din care face parte X trebuie recalculat integral (parcurgem blocul și calculăm cmmdc luând în considerare fiecare valoare care apare în listă). Prin urmare la eliminare complexitatea este de $\mathcal{O}(\sqrt{N})$.

Determinarea cmmdc-ului după o operație constă în determinarea cmmdc pentru toate cmmdc-urile blocurilor (din nou $\mathcal{O}(\sqrt{N})$).

Deoarece determinarea cmmdc se realizează după fiecare operație, complexitatea efectuării tuturor operațiilor va fi $\mathcal{O}(N \times \sqrt{N})$.

Implementare

```
#include <fstream>
#include <algorithm>
#define DIM 317
#define QMAX 100008
using namespace std;

ifstream fin("toorcmmdc.in");
ofstream fout("toorcmmdc.out");

int Q;
pair<int, int> Op[QMAX]; //operațiile

int B[QMAX]; //valorile care intervin în operații
int V[QMAX]; //valorile distincte care intervin în operații
int lgV; //numărul de elemente din V

int fr[QMAX];
//vector de frecvență;
//în fr[i] reținn numărul de apariții ale valorii V[i]

int cmmdc_bloc[DIM + 8];
//în vectorul cmmdc_bloc reținem cmmdc-ul valorilor din fiecare bloc
```

```
void Citire();
int determina_bloc(int i);
void Normalizare();
int Caut(int Val);
int cmmdc(int a, int b);
void Rezolvare();

int main()
{
    Citire();
    Normalizare();
    Rezolvare();
    return 0;
}

void Citire()
{
    int i, X;
    char tip;
    fin >> Q;
    for (i = 1; i <= Q; ++i)
    {
        fin >> tip >> X;
        if (tip == '+')
            Op[i] = {X, +1};
        else
            Op[i] = {X, -1};
        B[i] = X; // B - vector auxiliar;
    }
    return;
}

int cmmdc(int a, int b)
{
    int r = 0;
    while (b)
    {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

void Normalizare()
{
    int i;
    sort(B + 1, B + Q + 1);
    V[++lgV] = B[1];
    for (i = 2; i <= Q; ++i)
        if (B[i] != B[i - 1])
            V[++lgV] = B[i];
}

int Caut(int Val)
//caut binar valoarea Val în vectorul V și returnez poziția pe care se află în vector
// respectiv 0 dacă Val nu se află în vectorul V
```

```
{
    int st = 0, dr= lgV+1, mij;
    while (dr-st>1)
    {
        mij = (st + dr)/2;
        //invariant V[st]<Val<=V[dr]
        if (V[mij] < Val)
            st=mij;
        else
            dr=mij;
    }
    if (dr <= lgV && V[dr] == Val) return dr;
    return 0;
}

int determina_bloc(int i)
//returnez indicele blocului în care se află poziția i
{
    if (i % DIM == 0)
        return (i / DIM);
    return (i / DIM + 1);
}

void Rezolvare()
{
    int i, Val, poz, bloc_crt, x, rez, j;
    for (i = 1; i <= Q; ++i)
    {
        Val = Op[i].first;
        poz = Caut(Val);
        if (Op[i].second == 1) /// Adaug element la listă
        {
            ++fr[poz];
            if (fr[poz] == 1) //este prima apariție a acestei valori
            {
                bloc_crt = determina_bloc(poz);
                /// Modific gcd-ul in bloc_crt;
                if (cmmdc_bloc[bloc_crt] == 0)
                    cmmdc_bloc[bloc_crt] = Val;
                else
                    cmmdc_bloc[bloc_crt] = cmmdc(cmmdc_bloc[bloc_crt], Val);
            }
        }
        else if (fr[poz] > 0) //Val există
        {
            --fr[poz];
            if (fr[poz] == 0) //am eliminat singura apariție
            {
                bloc_crt = determina_bloc(poz);
                //recalculez cmmdc pentru blocul curent
                cmmdc_bloc[bloc_crt] = 0;
                for (x = (bloc_crt - 1) * DIM + 1; x <= bloc_crt * DIM; ++x)
                    if (fr[x] > 0)
                        if (cmmdc_bloc[bloc_crt] == 0)
                            cmmdc_bloc[bloc_crt] = V[x];
                        else
                            cmmdc_bloc[bloc_crt] = cmmdc(cmmdc_bloc[bloc_crt], V[x]);
```

```
            }
        }
        rez = 0; //rezultatul va fi cmmdc dintre cmmdc pentru toate blocurile
        for (j = 1; j <= determina_bloc(lgV); ++j)
            if (cmmdc_bloc[j] != 0)
            {
                if (rez == 0)
                    rez = cmmdc_bloc[j];
                else
                    rez = cmmdc(rez, cmmdc_bloc[j]);
            }
        if (rez == 0) // cmmdc-ul unei mulțimi vide e 1 (convenție);
            rez = 1;
        fout << rez << '\n';
    }
}
```

7.2.5 Problema Mayonaka

Eudanip nu și-a rezolvat problemele de implementare care-l bântuie de atâția ani. Pentru a se ascunde de rușinea publică cauzată de acest fapt, el s-a stabilit în Australia, unde lucrează la un centru de studiu al cangurilor. Specialiștii în canguri încearcă să studieze cum se schimbă greutatea cangurilor în timp. În acest scop, ei au împărțit o anumită cărare pe care călătoresc deseori canguri în N celule. Fiecare dintre cele N celule conține un cântar, care va însuma masa tuturor cangurilor care pășesc pe celula respectivă. Un cangur poate fi descris succint printr-un tuplu $(x, y, salt, greutate)$. El va începe călătoria din celula x și va păși pe celulele $x + k \times salt$ pentru toți $k \geq 0$ pentru care $x + k * salt \leq y$.

Cerință

Dându-se un N și o listă de M canguri, Eudanip trebuie să afle suma înregistrată pe fiecare cântar după încheierea tuturor călătoriilor. El știe să facă asta, dar a decis să vă propună vouă problema în concurs, poate poate ia o sursă de 100 de puncte de la voi.

Date de intrare

Fișierul de intrare `mayonaka.in` va conține pe prima linie numerele N și M, având semnificația din enunț. Următoarele M linii vor conține câte 4 numere x, y, $salt$, $greutate$ având semnificația din enunț.

Date de ieșire

Fișierul de ieșire `mayonaka.out` va conține N numere, reprezentând sumele înregistrate de cântarul din fiecare dintre cele N celule, în ordine de la 1 la N.

Restricții și precizări

- $1 \leq N, M \leq 100\,000$
- Pentru fiecare cangur, $1 \leq x \leq y \leq N$, $1 \leq salt \leq N - 1$ și $1 \leq greutate \leq 10\,000$

Exemple

mayonaka.in	mayonaka.out
10 5 1 10 1 1 1 10 2 5 2 9 3 7 2 6 2 10 1 10 5 10	16 18 6 11 13 21 6 8 6 1

Soluție

Pentru a reține rezultatele, vom utiliza un vector rez cu N elemente, unde $rez[i]$=suma greutăților cangurilor care pășesc în celula i ($1 \leq i \leq N$).

Un cangur poate trece prin $(y - x + 1)/salt$ celule. Acest număr poate fi prea mare în cazul în care lungimea saltului este mică. Pentru a optimiza calculele, vom aplica tehnica **Square root decomposition**. Să notăm cu $Root = sqrt(N)$.

Vom împărți cangurii în două categorii.

Categoria 1. Cangurii care au lungimea saltului $\geq Root$.

Pentru acești canguri numărul de salturi efectuate este $\leq \sqrt{N}$, ca urmare vom efectua salturile, adăugând greutatea acestor canguri la $rez[i]$ pentru toate celulele i prin care aceștia pășesc.

Categoria 2. Cangurii care au lungimea saltului $< Root$.

Vom construi un vector V cu $Root$ elemente, unde $V[i]$=lista cangurilor care au saltul egal cu i. Vom procesa cangurii în ordinea lungimii salturilor efectuate. Mai exact la pasul i vom parcurge lista cangurilor care au saltul egal cu i (adică lista $V[i]$).

Pentru a determina sumele greutăților pentru celulele pe care pășesc acești canguri vom adapta șmenul lui Mars (adică vom utiliza un tablou de diferențe). Pentru aceasta vom utiliza tabloul $Mars$ (care va avea dimensiunea N). Inițial pentru fiecare cangur din $V[i]$ adunăm greutatea sa la $Mars[Start]$ unde $Start$ punctul din care pleacă acest cangur și scădem greutatea sa din $Mars[Final]$, unde $Final$ este prima celulă pe care cangurul nu mai ajunge să sară (fiind mai mare decât y).

Pentru a determina sumele greutăților cangurilor cu saltul de lungime i care pășesc în celula j ($i + 1 \leq j \leq N$) adunăm la $Mars[j]$ valoarea $Mars[j - i]$.

Apoi adunăm la vectorul rez sumele obținute, însumând astfel greutățile tuturor cangurilor care au saltul egal cu i.

Implementare

```
#include <fstream>
#include <cmath>
#include <vector>
#define NMAX 100002
#define DIM 317
```

```cpp
using namespace std;

ifstream fin("mayonaka.in");
ofstream fout("mayonaka.out");

int N, M, Root;

struct Cangur
{
    int X, Y, S, G;
};
Cangur C[NMAX];

vector<Cangur> V[DIM];

int Mars[NMAX], rez[NMAX];

void Citire();
void Rezolvare();
void Afisare();

int main()
{
    Citire();
    Rezolvare();
    Afisare();
    return 0;
}

void Citire()
{
    int i, j;
    fin.tie(nullptr);
    fin >> N >> M;
    Root = (int)sqrt(N);
    for (i = 1; i <= M; ++i)
    {
        fin >> C[i].X >> C[i].Y >> C[i].S >> C[i].G;
        if (C[i].S >= Root) //efectuez salturile
            for (j = C[i].X; j <= C[i].Y; j += C[i].S)
                rez[j] += C[i].G;
        else //retin acest cangur in lista cangurilor cu acest salt
            V[C[i].S].push_back(C[i]);
    }
}

void Rezolvare()
{
    int i, j, Start, Final;
    for (i = 1; i < Root; ++i)
        if (!V[i].empty())
        {
            //inițializare
            for (j = 1; j <= N + 1; ++j) Mars[j] = 0;
            for (j = 0; j < V[i].size(); ++j)
            {
                Start = V[i][j].X;
```

```
                Final = Start + ((V[i][j].Y - Start)/i) *i + i;
                 Mars[Start] += V[i][j].G;

                if (Final <= N)
                    Mars[Final] -= V[i][j].G;
            }

            for (j = i + 1; j <= N; ++j)
                Mars[j] += Mars[j - i];

            for (j = 1; j <= N; ++j)
                rez[j] += Mars[j];
        }
}

void Afisare()
{
    int i;
    for (i = 1; i <= N; ++i)
        fout << rez[i]<<' ';
    fout << '\n';
}
```

7.3 Probleme propuse

- Problema Array Queries
- Problema Aparate
- Problema Xp

7.4 Bibliografie

[1] Dan Pracsiu, *Square Root Decomposition (Șmenul lui Batog)*, URL: https://www.youtube.com/watch?v=tLDEZkytmXQ.

[2] Cosmin Negruseri, *Coding Contest Byte: The Square Root Trick*, URL: https://www.infoarena.ro/blog/square-root-trick.

[3] *Square root decomposition and applications*, URL: https://codeforces.com/blog/entry/83248.

Capitolul 8

Algoritmul lui Mo

prof. Emanuela Cerchez
Colegiul Național „Emil Racoviță” Iași
Centrul Județean de Excelență Iași

8.1 Introducere

Algoritmul lui Mo este un algoritm generic, care se poate aplica problemelor de tipul următor:
Se consideră un vector A cu N elemente și Q interogări de tipul „să se determine rezultatul unei funcții `Fct()` , pentru intervalul $[st, dr]$”.
Să notăm cu `Fct(st,dr)` rezultatul pentru o interogare pentru intervalul $[st, dr]$.

Algoritmul poate fi aplicat doar dacă sunt îndeplinite următoarele condiții:

- interogările nu modifică tabloul;
- toate interogările sunt cunoscute de la început (un algoritm care necesită o astfel de condiție se numește *offline*);
- dacă știm `Fct(st,dr)` , putem determina `Fct(st+1,dr)` , `Fct(st-1,dr)` , `Fct(st,dr+1)` , `Fct(st,dr-1)` într-o complexitate timp $\mathcal{O}(F)$.

Complexitate

Algoritmul lui Mo oferă o soluție în $\mathcal{O}((N+Q)\sqrt{N} \times F)$ timp cu cel puțin $\mathcal{O}(Q)$ memorie suplimentară.

Observație

Condițiile 1 și 2 sunt foarte restrictive. Dar condiția 3 face ca acest algoritm să ofere o soluție pentru probleme care nu pot fi rezolvate într-un alt mod.

Dacă este prea abstract, iată un exemplu:
Fie A un vector cu N elemente și Q interogări. Pentru interogarea i trebuie să determinăm suma elementelor din intervalul $[st_i, dr_i]$ $(1 \leq i \leq Q)$.

Aceasta nu este, evident, o problemă dificilă și există soluții bazate pe sume parțiale, arbori indexați binar sau arbori de intervale.

Notații

1. Segmentul $[st, dr]$ este format din elementele $A_{st}, A_{st+1}, \ldots, A_{dr}$ (st este extremitatea inițială, iar dr cea finală). Evident, poziția i aparține segmentului $[st, dr]$ dacă $st \leq i \leq dr$.
2. x/y reprezintă câtul împărțirii întregi a lui x la y;
3. `sqrt(x)` reprezintă cel mai mare număr întreg mai mic sau egal cu $\sqrt{x}$;
4. pentru simplitatea calculelor, considerăm pozițiile în vector numerotate începând cu 0.

8.2 Algoritm

- Determinăm `BLOCK_SIZE=sqrt(N)` ;
- Reordonăm toate interogările într-o ordine pe care o vom denumi **ordinea lui Mo**, definită astfel:
  segmentul $[st1, dr1]$ va fi înaintea segmentului $[st2, dr2]$ dacă și numai dacă
  `st1/BLOCK_SIZE < st2/BLOCK_SIZE`
  sau
  `st1/BLOCK_SIZE==st2/BLOCK_SIZE && dr1 < dr2`
- Parcurgem interogările în ordinea lui Mo. Să presupunem că interogarea curentă este $[st, dr]$. Executăm următoarele operații:
  - cât timp $mo\_dr < dr$, extindem segmentul Mo curent la $[mo\_st, mo\_dr + 1]$;
  - cât timp $mo\_dr > dr$, reducem segmentul Mo curent la $[mo\_st, mo\_dr - 1]$;
  - cât timp $mo\_st > st$, extindem segmentul Mo curent la $[mo\_st - 1, mo\_dr]$;
  - cât timp $mo\_st < st$, reducem segmentul Mo curent la $[mo\_st + 1, mo\_dr]$.

Observații

1. Pasul 4 necesită $\mathcal{O}((|mo\_st - st| + |mo\_dr - dr|) \times F)$ timp, deoarece fiecare extindere/reducere este executată în $\mathcal{O}(F)$ timp.
2. La acest pas, după toate tranzițiile, $mo\_st = st$ și $mo\_dr = dr$, prin urmare am calculat `Fct(st,dr)` .

Complexitate timp

Considerăm că vectorul A este o succesiune de segmente disjuncte de dimensiune $BLOCK\_SIZE$, pe care le vom denumi blocuri, ca în Figura 8.1.

Observația 1. Fie K indicele ultimului bloc. Fiind numerotate de la 0, vor exista $K+1$ blocuri. Este posibil ca ultimul bloc să aibă mai puțin de $BLOCK\_SIZE$ elemente.

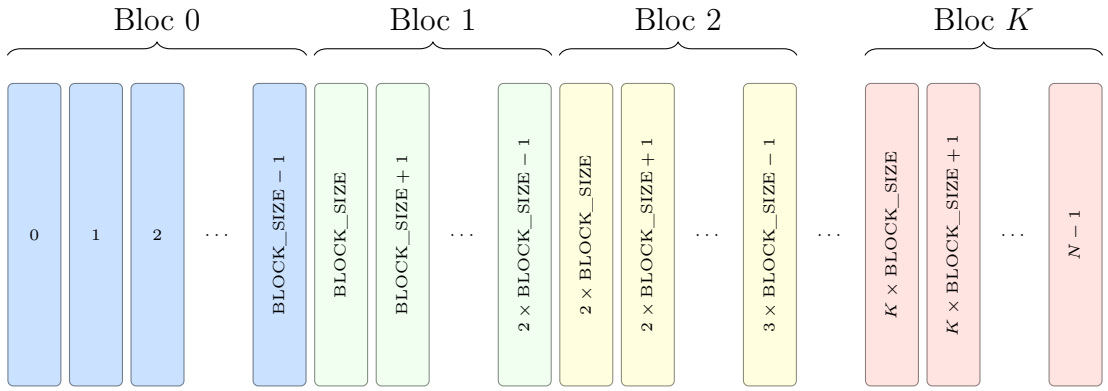

Figura 8.1: Un vector de N elemente descompus în blocuri de lungime BLOCK_SIZE.

Observația 2. Blocul r $(0 \leq r \leq K)$ este segmentul

$$[r \times BLOCK\_SIZE, \min(N-1, (r+1) \times BLOCK\_SIZE - 1)]$$

Demonstrația se poate face ușor prin inducție.

Observația 3. Două poziții i și j aparțin aceluiași bloc r dacă și numai dacă

$$i/BLOCK\_SIZE = j/BLOCK\_SIZE = r$$

Observația 4. Notăm Q_r=mulțimea interogărilor $[st, dr]$ pentru care $st/BLOCK\_SIZE = r$ (deci interogările pentru care extremitatea inițială aparține blocului r). Evident, Q_r poate fi vidă. În ordinea lui Mo, interogările vor fi plasate în ordinea crescătoare a blocurilor și pentru orice bloc r, interogările din Q_r vor fi consecutive și vor fi în ordinea crescătoare a extremităților finale. Deci, când vom procesa interogările în ordinea lui Mo, mai întâi procesăm interogările din Q_0, apoi toate interogările din Q_1 ș.a.m.d până la Q_K.

Observația 5. La pasul 4 în algoritmul lui Mo, complexitatea timp a schimbării valorii mo_dr este $\mathcal{O}(N\sqrt{N})$.

Demonstraţie. Să presupunem că am început să prelucrăm interogările din Q_r și deja am prelucrat prima interogare din Q_r. Să notăm această interogare $[st, dr_0]$. Prin urmare, în acest moment $mo\_dr = dr_0$.

Interogările din Q_r sunt ordonate crescător după extremitatea finală, deci $dr_0 \leq dr_1 \leq \cdots \leq dr_{|Q_r|-1}$, unde $dr_0, dr_1, \ldots, dr_{|Q_r|-1}$ sunt capetele din dreapta ale interogărilor din Q_r.

Din definiția lui Q_r știm că $r \times BLOCK\_SIZE \leq st$ și deoarece $st \leq dr$, deducem că $r \times BLOCK\_SIZE \leq dr_0 \leq dr_1 \leq \cdots \leq dr_{|Q_r|-1} \leq N-1$.

Deci mo_dr se modifică de $dr_{|Q_r|-1} - dr_0 \leq N - 1 - r \times BLOCK\_SIZE = \mathcal{O}(N)$ la procesarea interogărilor Q_r (am înlocuit $dr_{|Q_r|-1}$ cu cea mai mare valoare posibilă și dr_0 cu cea mai mică valoare posibilă pentru a maximiza rezultatul).

Există $\mathcal{O}(\sqrt{N})$ valori posibile pentru r deci în total $\mathcal{O}(N\sqrt{N})$ modificări, considerând că începem de la prima interogare din fiecare Q_r.

Să presupunem acum că am terminat de prelucrat interogările din Q_r și trebuie să prelucrăm prima interogare din $Q_{r'}$ (următoarea mulțime de interogări care nu este vidă).

La momentul curent, mo_dr îndeplinește condiția: $r \times BLOCK\_SIZE \leq mo\_dr \leq N-1$.

Să considerăm că prima interogare din $Q_{r'}$ este $[st', dr']$. Știm că $(r+1)\times BLOCK\_SIZE \leq dr' \leq N-1$ pentru că $(r+1) \times BLOCK\_SIZE \leq st'$ interogarea fiind din următoarea mulțime și $st' \leq dr'$.

Deci, mo_dr trebuie să fie schimbat de cel mult $\max(|r \times BLOCK\_SIZE-(N-1)|, N-1-(r+1) \times BLOCK\_SIZE)$ ori (am luat cea mai mică valoare pentru mo_dr cu cea mai mare valoare pentru dr'), deci $\mathcal{O}(N)$.

Există $\mathcal{O}(\sqrt{N})$ treceri de la o mulțime de interogări la următoarea deci în total $\mathcal{O}(N\sqrt{N})$ modificări pentru mo_dr.

Deci toate modificările pentru mo_dr au complexitatea timp $\mathcal{O}(N \times \sqrt{N} \times F)$. □

Observația 6. La pasul 4 în algoritmul lui Mo, mo_st se modifică de $\mathcal{O}(Q\sqrt{N})$ ori.

Demonstraţie. În mod similar cu demonstrația precedentă, vom presupune că am început să prelucrăm interogările din mulțimea Q_r $(0 \leq r \leq K)$ și că la momentul curent $mo\_st = st$, $mo\_dr = dr_0$, unde $[st, dr_0]$ este prima interogare din Q_r.

Pentru fiecare interogare $[st', dr']$ din Q_r avem: $r \times BLOCK\_SIZE \leq st' \leq (r+1) \times BLOCK\_SIZE - 1$.

Deci, mo_st se va modifica, pentru fiecare interogare din Q_r de cel mult $(r+1) \times BLOCK\_SIZE - 1 - r \times BLOCK\_SIZE = BLOCK\_SIZE - 1 = \mathcal{O}(\sqrt{N})$ ori.

Există $|Q_r|$ interogări în Q_r. Deci, pentru un r fixat complexitatea este $\mathcal{O}(|Qr| \times \sqrt{N})$. Dacă facem suma pentru orice r, $\mathcal{O}(\sqrt{N} \times (|Q_0| + |Q_1| + ... + |Q_K|)) = \mathcal{O}(\sqrt{N} \times Q)$

Acum să presupunem că am terminat de prelucrat toate interogările din Q_r și trebuie să trecem la prima interogare din următoarea mulțime nevidă Q_{r+k} $(k > 0)$. Orice interogare $[st', dr']$ din Q_{r+k} respectă condiția $(r+k) \times BLOCK\_SIZE \leq st' \leq (r+k+1) \times BLOCK\_SIZE - 1$.

Similar, orice interogare $[st, dr]$ din Q_r respectă condiția $r \times BLOCK\_SIZE \leq st \leq (r+1) \times BLOCK\_SIZE - 1$.

Deci numărul maxim de modificări pentru tranziție este $(r+k+1) \times BLOCK\_SIZE - 1 - r \times BLOCK\_SIZE = k \times BLOCK\_SIZE - 1$. Însumând pentru toate mulțimile de interogări, rezultă $k \times BLOCK\_SIZE = \mathcal{O}(\sqrt{N}) \times \mathcal{O}(\sqrt{N}) = \mathcal{O}(N)$ modificări ale capetelor st.

În total, $\mathcal{O}(\sqrt{N} \times Q)+\mathcal{O}(N) = \mathcal{O}(\sqrt{N} \times Q)$ modificări pentru mo_st.

Deci, toate modificările pentru mo_st au o complexitate timp $\mathcal{O}(Q \times \sqrt{N} \times F)$. □

Rezultă că algoritmul lui Mo are complexitatea timp $\mathcal{O}((N+Q) \times \sqrt{N} \times F)$.

8.3 Aplicații

8.3.1 Problema Sumcnt

Fie A un vector cu N elemente ($N \leq 10^5$), numere naturale mai mici decât 100. Se cere să răspundeți la Q interogări ($Q \leq 10^5$) de forma: să se determine

$$V(st, dr) = \sum_{i=0}^{99} i \times cnt^2(i)$$

unde $cnt(i)$ este numărul de apariții ale valorii i în segmentul $[st, dr]$.

Soluție

Putem aplica algoritmul lui Mo, deoarece interogările sunt cunoscute și nu modifică vectorul. În plus, dacă știm deja $V(st, dr)$ putem determina $V(st, dr + 1)$, $V(st, dr - 1)$, $V(st - 1, dr)$ și $V(st + 1, dr)$ în O(1) după cum urmează.

Vom utiliza un vector de frecvență cnt cu 100 de elemente, unde $cnt[i]$= numărul de apariții ale valorii i în segmentul Mo curent [mo_st, mo_dr].

Pentru a determina $V(mo\_st, mo\_dr + 1)$ trebuie să luăm în considerare și valoarea $x = A[mo\_dr + 1]$. Pentru aceasta incrementăm $cnt[x]$ și actualizăm corespunzător V (scădem $x * cnt^2[x]$, actualizăm $cnt[x]$ apoi adăugăm $x * cnt^2[x]$).

Să analizăm un exemplu:

- $N = 18$
- $A = [0, 1, 1, 0, 2, 3, 4, 1, 3, 5, 1, 5, 3, 5, 4, 0, 2, 2]$
- Interogările sunt $[0, 8], [2, 5], [2, 11], [16, 17], [13, 14], [1, 17], [17, 17]$

$BLOCK\_SIZE = \lfloor\sqrt{18}\rfloor = 4$. Vom avea 5 blocuri: $[0, 3], [4, 7], [8, 11], [12, 15], [16, 17]$.

Inițializăm segmentul Mo cu segmentul vid ($mo\_st = 0, mo\_dr = -1$), $rez\_crt = 0$ și $cnt = [0, 0, 0, 0, 0, 0]$.

Ordonăm interogările:

Blocul	Q_0	Q_3	Q_4
Interogările	[2,5], [0, 8], [2, 11], [1, 17]	[13, 14]	[16, 17], [17, 17]

Tabela 8.1: Interogările în ordinea Mo.

Procesăm interogările în ordinea Mo.

Segmentul Mo curent	Interogarea curentă	Operații	V
[0,-1]	[2,5]	$mo\_dr = 0, rez\_crt = 0, cnt = [1,0,0,0,0,0]$ $mo\_dr = 1, rez\_crt = 1, cnt = [1,1,0,0,0,0]$ $mo\_dr = 2, rez\_crt = 4, cnt = [1,2,0,0,0,0]$ $mo\_dr = 3, rez\_crt = 4, cnt = [2,2,0,0,0,0]$ $mo\_dr = 4, rez\_crt = 6, cnt = [2,2,1,0,0,0]$ $mo\_dr = 5, rez\_crt = 9, cnt = [2,2,1,1,0,0]$ $mo\_st = 1, rez\_crt = 9, cnt = [1,2,1,1,0,0]$ $mo\_st = 2, rez\_crt = 6, cnt = [1,1,1,1,0,0]$	6
[2,5]	[0,8]	$mo\_dr = 6, rez\_crt = 10, cnt = [1,1,1,1,1,0]$ $mo\_dr = 7, rez\_crt = 13, cnt = [1,2,1,1,1,0]$ $mo\_dr = 8, rez\_crt = 22, cnt = [1,2,1,2,1,0]$ $mo\_st = 1, rez\_crt = 27, cnt = [1,3,1,2,1,0]$ $mo\_st = 0, rez\_crt = 27, cnt = [2,3,1,2,1,0]$	27
[0,8]	[2,11]	$mo\_dr = 9, rez\_crt = 32, cnt = [2,3,1,2,1,1]$ $mo\_dr = 10, rez\_crt = 39, cnt = [2,4,1,2,1,1]$ $mo\_dr = 11, rez\_crt = 54, cnt = [2,4,1,2,1,2]$ $mo\_st = 1, rez\_crt = 54, cnt = [1,4,1,2,1,2]$ $mo\_st = 2, rez\_crt = 47, cnt = [1,3,1,2,1,2]$	47
[2,11]	[1,17]	$mo\_dr = 12, rez\_crt = 62, cnt = [1,3,1,3,1,2]$ $mo\_dr = 13, rez\_crt = 87, cnt = [1,3,1,3,1,3]$ $mo\_dr = 14, rez\_crt = 99, cnt = [1,3,1,3,2,3]$ $mo\_dr = 15, rez\_crt = 99, cnt = [2,3,1,3,2,3]$ $mo\_dr = 16, rez\_crt = 105, cnt = [2,3,2,3,2,3]$ $mo\_dr = 17, rez\_crt = 115, cnt = [2,3,3,3,2,3]$ $mo\_st = 1, rez\_crt = 122, cnt = [2,4,3,3,2,3]$	122
[1,17]	[13,14]	$mo\_dr = 16, rez\_crt = 112, cnt = [2,4,2,3,2,3]$ $mo\_dr = 15, rez\_crt = 106, cnt = [2,4,1,3,2,3]$ $mo\_dr = 14, rez\_crt = 106, cnt = [1,4,1,3,2,3]$ $mo\_st = 2, rez\_crt = 99, cnt = [1,3,1,3,2,3]$ $mo\_st = 3, rez\_crt = 94, cnt = [1,2,1,3,2,3]$ $mo\_st = 4, rez\_crt = 94, cnt = [0,2,1,3,2,3]$ $mo\_st = 5, rez\_crt = 92, cnt = [0,2,0,3,2,3]$ $mo\_st = 6, rez\_crt = 77, cnt = [0,2,0,2,2,3]$ $mo\_st = 7, rez\_crt = 65, cnt = [0,2,0,2,1,3]$ $mo\_st = 8, rez\_crt = 62, cnt = [0,1,0,2,1,3]$ $mo\_st = 9, rez\_crt = 53, cnt = [0,1,0,1,1,3]$ $mo\_st = 10, rez\_crt = 28, cnt = [0,1,0,1,1,2]$ $mo\_st = 11, rez\_crt = 27, cnt = [0,0,0,1,1,2]$ $mo\_st = 12, rez\_crt = 12, cnt = [0,0,0,1,1,1]$ $mo\_st = 13, rez\_crt = 9, cnt = [0,0,0,0,1,1]$	9
[13,14]	[16,17]	$mo\_dr = 15, rez\_crt = 9, cnt = [1,0,0,0,1,1]$ $mo\_dr = 16, rez\_crt = 11, cnt = [1,0,1,0,1,1]$ $mo\_dr = 17, rez\_crt = 17, cnt = [1,0,2,0,1,1]$ $mo\_st = 14, rez\_crt = 12, cnt = [1,0,2,0,1,0]$ $mo\_st = 15, rez\_crt = 8, cnt = [1,0,2,0,0,0]$ $mo\_st = 16, rez\_crt = 8, cnt = [0,0,2,0,0,0]$	8
[16,17]	[17,17]	$mo\_st = 17, rez\_crt = 2, cnt = [0,0,1,0,0,0]$	2

Tabela 8.2: Aplicarea algoritmului lui Mo pe exemplu.

Implementare

```
#include <bits/stdc++.h>
#define VMAX 100
#define QMAX 100500
#define NMAX 100500
using namespace std;

int N, Q;
long long int rez_crt;
long long int cnt[VMAX];

// în tabloul rez vom reține rezultatele interogărilor, deoarece le
// obținem în altă ordine decât trebuie să fie afișate
long long int rez[QMAX];
```

```
int BLOCK_SIZE;
int A[NMAX];

//reprezentăm o interogare prin 3 numere: st, dr, nr - numărul de ordine
struct interogare {int st, dr, nr;};
interogare queries[QMAX];

//funcția de comparare a interogărilor
inline bool mo_cmp(interogare x, interogare y)
{   int block_x=x.st / BLOCK_SIZE;
    int block_y=y.st / BLOCK_SIZE;
    if(block_x != block_y)
        return block_x < block_y;
    return x.dr < y.dr;
}

inline void add(int x)
{
    rez_crt -= cnt[x]*cnt[x]*x;
    cnt[x]++;
    rez_crt += cnt[x]*cnt[x]*x;
}

inline void remove(int x)
{
    rez_crt -= cnt[x]*cnt[x]*x;
    cnt[x]--;
    rez_crt += cnt[x]*cnt[x]*x;
}

int main()
{int i, st, dr;
 cin.sync_with_stdio(false);
 cin >> N >> Q;
 BLOCK_SIZE=(int)sqrt(N);
 for (i=0; i < N; i++) cin >> A[i];
 for (i=0; i < Q; i++)
     {
      cin >> queries[i].st >> queries[i].dr;
      queries[i].nr=i;
     }

 //sortarea interogărilor
 sort(queries, queries+Q, mo_cmp);

// Inițializarea segmentului Mo curent [mo_st, mo_dr].
 int mo_st=0,mo_dr=-1;

 //prelucrarea interogărilor în ordinea Mo
 for (i=0; i < Q; i++)
     {
      st=queries[i].st; dr=queries[i].dr;
      while (mo_dr < dr) { mo_dr++;  add(A[mo_dr]);  }
      while (mo_dr > dr) { remove(A[mo_dr]); mo_dr--; }
      while (mo_st < st) { remove(A[mo_st]); mo_st++; }
      while (mo_st > st) { mo_st--; add(A[mo_st]);    }
      rez[queries[i].nr]=rez_crt;
```

```
    }
 for (i=0; i < Q; i++) cout << rez[i] << "\n";
 return 0;
}
```

8.3.2 Problema D-Query

Se consideră o secvență de n numere naturale a_1, a_2, ..., a_n și o succesiune de interogări denumite **d-query**. Un **d-query** constă în a determina pentru o pereche (i, j) $(1 \leq i \leq j \leq n)$ numărul de valori distincte din secvența $a_i, a_{i+1}, \ldots, a_j$.

Date de intrare

Pe prima linie se află numărul natural n $(1 \leq n \leq 30\,000)$. Pe cea de a doua linie se află numerele $a_1, a_2, \ldots, a_n$ $(1 \leq a_i \leq 10^6$, pentru $1 \leq i \leq n)$, separate prin câte un spațiu. Pe linia a treia se află numărul natural q, reprezentând numărul de interogări $(1 \leq q \leq 200\,000)$. Următoarele q linii conțin fiecare câte două numere ij, separate prin spațiu, reprezentând câte o interogare $(1 \leq i \leq j \leq n)$.

Date de ieșire

Pentru fiecare interogare ij, afișați numărul de valori distincte din secvența $a_i, a_{i+1}, \ldots, a_j$, câte un rezultat pe o linie, în ordinea în care au fost citite interogările.

Exemple

Intrare	Ieșire
5 1 1 2 1 3 3 1 5 2 4 3 5	3 2 3

Soluție

Vom utiliza un vector de frecvență cnt, unde $cnt[x]$ = numărul de apariții ale valorii x în segmentul Mo curent. Când adăugăm un element x în segmentul Mo verificăm dacă aceasta este prima sa apariție $(cnt[x] == 0)$ și în acest caz îl adăugăm la rezultatul pentru segmentul Mo curent (o nouă valoare distinctă). Când eliminăm o valoare x din segmentul Mo curent verificăm dacă aceasta este singura sa apariție $(cnt[x] == 1)$, caz în care o eliminăm din rezultatul pentru segmentul Mo curent.

Implementare

```
#include <bits/stdc++.h>
#define QMAX 200200
#define NMAX 30300
```

```
#define AMAX 1000100
using namespace std;
int rezultat;
struct interogare { int st, dr, nr; };
interogare Q[QMAX];
int rez[QMAX];
int a[NMAX];
int cnt[AMAX];

void add(int index)
{   cnt[a[index]]++;
    if(cnt[a[index]]==1) rezultat++;
}

void remove(int index)
{
    cnt[a[index]]--;
    if(cnt[a[index]]==0) rezultat--;
}

int block_size;
bool mo_cmp(interogare x, interogare y)
{int block_x = x.st/block_size;
 int block_y = y.st/block_size;
 if (block_x == block_y)
    return x.dr < y.dr;
 return block_x < block_y;
}

int main()
{ios_base::sync_with_stdio(false);
 int n, q, i;
 scanf("%d",&n);
 for (i=0; i<n; i++) scanf("%d",&a[i]);
 scanf("%d",&q);
 for (i=0; i<q; i++)
     {
        scanf("%d%d", &Q[i].st, &Q[i].dr);
        Q[i].nr=i;  Q[i].st--;  Q[i].dr--;
      }
  block_size=(int)sqrt(n);
  sort(Q, Q+q, mo_cmp);

  int mo_st=0,mo_dr=-1;
  for (i=0; i<q; i++)
      {
        int left=Q[i].st,right=Q[i].dr;
        while(mo_st<left)  remove(mo_st++);
        while(mo_st>left)  add(--mo_st);
        while (mo_dr<right)add(++mo_dr);
        while(mo_dr>right) remove(mo_dr--);
        rez[Q[i].nr]=rezultat;
    }
  for (i=0; i<q; i++) printf("%d\n",rez[i]);
  return 0;
}
```

8.3.3 Problema Kriti and her birthday gift

Astăzi este aniversarea lui Kriti. Fiindcă îi place foarte mult programarea, îi voi face cadou următoarea problemă. Se consideră N șiruri numerotate de la 1 la N și o succesiune formată din Q interogări. Fiecare interogare este de forma *st dr S*, rezultatul unei astfel de interogări fiind numărul de apariții ale șirului S în secvența de șiruri cu numere de ordine cuprinse în intervalul $[st, dr]$.

Date de intrare

Pe prima linie se află numărul natural N, iar pe următoarele N linii se află cele N șiruri, câte un șir pe o linie. Următoarea linie conține numărul natural Q, reprezentând numărul de interogări, iar următoarele Q linii conțin interogările, în forma descrisă în enunț, câte o interogare pe o linie.

Date de ieșire

Se vor afișa rezultatele pentru cele Q interogări, câte un rezultat pe o linie, în ordinea în care au fost citite interogările.

Restricții și precizări

- $100 \leq N \leq 100\,000$
- $100 \leq Q \leq 100\,000$
- $1 \leq st \leq dr \leq N$
- Șirurile conțin cel puțin 5 și cel mult 10 litere mici din alfabetul englez.

Exemple

Intrare	Ieșire	Explicații
3 abc def abc 3 1 2 abc 1 3 abc 1 2 hgj	1 2 0	Pentru prima interogare, șirul `abc` apare o singură dată în intervalul de șiruri [1,2]. Pentru a doua interogare șirul `abc` apare de două ori în intervalul de șiruri [1,3]. Pentru a treia interogare, șirul `hgj` nu are nicio apariție în intervalul de șiruri [1,2].

Soluție

Vom asocia fiecărui șir o valoare *hash* obținută prin metoda Zobrist.

Pentru aceasta vom construi un tablou cu $LGMAX$ linii (unde $LGMAX$ este lungimea maximă a unui șir) și 26 de coloane (numărul de litere mici din alfabetul englez) și asociem astfel fiecărei litere plasată pe o anumită poziție în șir o valoare generată aleator.

Valoarea *hash* asociată unui șir de caractere se obține realizând o disjuncție exclusivă (XOR) între valorile literelor de pe fiecare poziție din șir.

Prin urmare, vom reprezenta o interogare prin 4 valori st, dr, h – valoarea *hash* asociată șirului din interogare, respectiv nr – numărul de ordine al interogării în datele de intrare.

Deoarece valorile *hash* vor fi numere foarte mari, nu putem utiliza un vector de frecvență ca în problema precedentă, ca urmare, pentru a contoriza aparițiile șirurilor care apar într-un segment vom utiliza un `unordered_map` denumit $nrap$ ($nrap[valh]$ = numărul de apariții din segmentul curent al șirului cu valoarea *hash* $valh$).

Implementare

```
#include <bits/stdc++.h>
#include <random>
#define LGMAX 11
#define LETTERS 26
#define NMAX 100500
#define QMAX 100500
using namespace std;
// Zobrist hashing
unsigned long long int zob[LGMAX][LETTERS];
int block_size;

void init_hash()
{mt19937_64 generator;
 generator.seed(123);
 int i, j;
 for (i = 0; i < LGMAX; i++)
      for (j = 0; j < LETTERS; j++)
          zob[i][j] = generator();
}

unsigned long long int get_hash(const string &s)
{unsigned long long int rez = 0;
 for (size_t i = 0; i < s.size(); i++)
      rez ^= zob[i][s[i]-'a'];
 return rez;
}

struct interogare {int st, dr, nr; unsigned long long int h;};
interogare qo[QMAX];

bool mo_cmp(interogare x, interogare y)
{
 int block_x = x.st / block_size;
 int block_y = y.st / block_size;
 if (block_x == block_y)
     return x.dr < y.dr;
 return block_x < block_y;
}

unordered_map<unsigned long long, int> nrap;

unsigned long long hashes[NMAX];
int rez[QMAX];

void add(unsigned long long val)
{nrap[val]++; }
```

```
void remove(unsigned long long val)
{  nrap[val]--;    }

int main()
{
 init_hash();
 cin.sync_with_stdio(false);
 int n, q, i, st, dr, nr;
 unsigned long long valh;
 string sir;
 interogare x;
 cin >> n;
 block_size = (int)(sqrt(n)) + 1;
 for (i = 0; i < n; i++)
     {
      cin >> sir;
      hashes[i] = get_hash(sir);
     }
 cin >> q;
 for (i = 0; i < q; i++)
     {cin >> x.st >> x.dr >> sir;
      x.st--; x.dr--;  x.h=get_hash(sir); x.nr=i;
      qo[i]=x;
     }
sort(qo, qo+q, mo_cmp);
//initializare
nrap.clear();
int prev_block = -1, mo_st = 0, mo_dr = -1;
for (i = 0; i < q; i++)
    {
      st=qo[i].st;   dr=qo[i].dr;   nr=qo[i].nr;   valh=qo[i].h;
      if (st / block_size != prev_block)
         {nrap.clear();  mo_st = st;  mo_dr = st - 1;  }
      prev_block = st / block_size;
      while (mo_dr < dr)  add(hashes[++mo_dr]);
      while (mo_st < st)  remove(hashes[mo_st++]);
      while (mo_st > st)  add(hashes[--mo_st]);
      rez[nr] = nrap[valh];
        }
for (i = 0; i < q; i++)  cout << rez[i] << "\n";
return 0;
}
```

8.3.4 Problema Substrings count

Se consideră n șiruri (numerotate de la 1 la n) și Q interogări referitoare la aceste șiruri. Fiecare dintre aceste interogări are forma st dr S, rezultatul unei astfel de interogări fiind numărul de șiruri cu numere de ordine din intervalul $[st, dr]$ conțin pe S ca subsecvență.

Date de intrare

Pe prima linie se află numărul natural n, iar pe următoarele n linii se află cele n șiruri, câte un șir pe o linie. Următoarea linie conține numărul natural Q, reprezentând numărul

de interogări, iar următoarele Q linii conțin interogările, în forma descrisă în enunț, câte o interogare pe o linie.

Date de ieșire

Se vor afișa rezultatele pentru cele Q interogări, câte un rezultat pe o linie, în ordinea în care au fost citite interogările.

Restricții și precizări

- $1 \leq n \leq 10\,000$
- $1 \leq Q \leq 500\,000$
- $1 \leq st \leq dr \leq N$
- Șirurile conțin cel mult 10 litere mici din alfabetul englez.

Exemple

Intrare	Ieșire	Explicații
3 code coder coding 2 1 3 code 1 3 co	2 3	Pentru prima interogare, trebuie să afișăm câte dintre șirurile cu numere de ordine din intervalul [1,3] conțin șirul *code* ca subsecvență. Rezultatul este 2 (doar șirurile 1 și 2 conțin *code* ca subsecvență). Pentru a doua interogare, trebuie să afișăm câte dintre șirurile cu numere de ordine din intervalul [1,3] conțin șirul *co* ca subsecvență. Rezultatul este 3, deoarece toate cele 3 șiruri conțin *co* ca subsecvență.

Soluție

Vom proceda într-un mod similar cu problema precedentă, cu o singură modificare: vom reține pentru fiecare dintre șirurile date într-un `unordered_map` valorile *hash* pentru toate subsecvențele șirurilor respective.

Implementare

```
#include <bits/stdc++.h>
#define LGMAX 11
#define LETTERS 26
#define QMAX 500500
#define NMAX 10500
using namespace std;

//Zobrist hashing
unsigned long long int zob[LGMAX][LETTERS];
void init_hash()
{int i, j;
 mt19937_64 generator;
```

```
 generator.seed(123);
 for (i=0; i < LGMAX; i++)
      for (j=0; j < LETTERS; j++)
          zob[i][j]=generator();
}

unsigned long long get_hash(const string &s)
{unsigned long long rez=0;
 for (int i=0; i < s.size(); i++)
      rez ^= zob[i][s[i]-'a'];
 return rez;
}

int block_size;
unordered_map<unsigned long long, int> hashes[NMAX];
//hashes[i] retine hashurile pentru toate subsecvențele din al i-lea șir

struct interogare {int st, dr, nr; unsigned long long int h;};
interogare qo[QMAX];
bool mo_cmp(interogare x, interogare y)
{int block_x=x.st/block_size;
 int block_y=y.st/block_size;
 if (block_x == block_y)
     return x.dr < y.dr;
 return block_x < block_y;
}
int rez[QMAX];
unordered_map<unsigned long long, int> nrap;

void add(const unordered_map<unsigned long long, int> &v)
{  for (const auto &u: v) {nrap[u.first] += u.second;}  }

void remove(const unordered_map<unsigned long long, int> &v)
{  for (const auto &u: v) {nrap[u.first] -= u.second;}  }

int main()
{init_hash();
 cin.sync_with_stdio(false);
 int n, q, i, j, k, st, dr, nr;
 string sir;
 unsigned long long hash_val, valh;
 interogare x;
 cin >> n;
 block_size=(int)(sqrt(n))+1;
 for (i=0; i < n; i++)
     {cin >> sir;
     //vom reține hash-ul pentru fiecare subsecvență a lui sir
      for (j=0; j < sir.size(); j++)
          {hash_val=0;
           for (k=j; k < sir.size(); k++)
               {
                hash_val ^= zob[k-j][sir[k]-'a'];
                hashes[i][hash_val]=1;
               }
          }
     }
 cin >> q;
```

```
 for (i=0; i < q; i++)
     { cin >> x.st >> x.dr >> sir;
       x.h=get_hash(sir); x.st--; x.dr--; x.nr=i;
       qo[i]=x;
     }
sort(qo, qo+q, mo_cmp);
//initializare
nrap.clear();
int prev_block=-1;
int mo_st=0, mo_dr=0;
for (i=0; i < q; i++)
    { st=qo[i].st; dr=qo[i].dr; nr=qo[i].nr; valh=qo[i].h;
      if (st / block_size != prev_block)
         { nrap.clear(); mo_st=st; mo_dr=st-1; }
      prev_block=st / block_size;
      while (mo_dr < dr) add(hashes[++mo_dr]);
      while (mo_st < st) remove(hashes[mo_st++]);
      while (mo_st > st)  add(hashes[--mo_st]);
      rez[nr]=nrap[valh];
  }
 for (i=0; i < q; i++)  cout << rez[i] << "\n";
 return 0;
}
```

8.3.5 Problema Sherlock and inversions

Watson îi dă lui Sherlock un vector format din N numere întregi, notat $A_1, A_2 \dots A_N$, precum și o succesiune de Q interogări. Fiecare interogare are forma st dr, rezultatul unei astfel de interogări fiind numărul de inversiuni din subsecvența $A_{st}, A_{st+1} \dots A_{dr}$. O inversiune în subsecvența $A_{st}, A_{st+1} \dots A_{dr}$ este o pereche de indici (i,j) astfel încât $st \leq i < j \leq dr$ și $A_i > A_j$.

Date de intrare

Prima linie conține numerele naturale N Q, separate prin spațiu. A doua linie conține N numere naturale separate prin spațiu, reprezentând elementele vectorului. Fiecare dintre următoarele Q linii conține câte o interogare, în forma descrisă în enunț.

Date de ieșire

Se vor afișa în ordinea citirii rezultatele pentru cele Q interogări, câte un rezultat pe o linie.

Restricții și precizări

- $1 \leq N, Q \leq 100\,000$
- $1 \leq A_i \leq 10^9$
- $1 \leq st \leq dr \leq N$

Exemple

Intrare	Ieșire	Explicații
5 3 1 4 2 3 1 1 2 3 5 1 5	0 2 5	Pentru prima interogare, trebuie să afișăm numărul de inversiuni din subsecvența 1, 4 (acesta este 0). Pentru a doua interogare, trebuie să afișăm numărul de inversiuni din subsecvența 2, 3, 1 (acesta este 2, inversiunile fiind 2 1 și 3 1). Pentru a treia interogare, trebuie să afișăm numărul de inversiuni din tot vectorul (acesta este 5: 4 2, 4 3, 4 1, 2 1, 3 1).

Soluție

Să notăm cu nrd = numărul de valori distincte din vectorul citit. Transformăm numerele din vectorul citit în numere cuprinse între 1 și nrd, asociind fiecărui număr citit numărul său de ordine în șirul valorilor distincte ordonate strict crescător. Aceste valori vor fi memorate în vectorul nr.

Să presupunem că segmentul Mo curent este $[mo\_st, mo\_dr]$.

Atunci când extindem segmentul la dreapta (deci introducem valoarea mo_dr+1), noua valoare va genera inversiuni cu toate valorile strict mai mari decât ea aflate în segmentul Mo curent. Deci ar trebui să știm câte valori mai mari decât noua valoare există în segmentul curent.

Atunci când extindem segmentul curent la stânga (deci introducem valoarea mo_st-1), noua valoare va genera inversiuni cu toate valorile strict mai mici decât ea aflate în segmentul Mo curent. Deci ar trebui să știm câte valori mai mici decât noua valoare există în segmentul Mo curent.

Atunci când reducem segmentul Mo curent (acest lucru va fi realizat doar la stânga, deci mo_st dispare din segmentul curent), trebuie să eliminăm din rezultat și numărul de inversiuni generate de valoarea eliminată (adică să scădem din rezultat numărul de valori strict mai mici decât valoarea eliminată aflate în segmentul Mo curent).

Pentru a determina în timp logaritmic valorile necesare celor 3 tipuri de modificări posibile, vom utiliza un arbore indexat binar (*Fenwick tree*) denumit *counter*, cu nrd elemente.

Implementare

```
#include <bits/stdc++.h>
#define DMAX 100100
using namespace std;
class FenwickTree
{vector<int> tree;
 int n;
    void add(int poz, int val)
    {
      for ( ; poz < n; poz = (poz | (poz + 1)))
           tree[poz] += val;
```

```
    }
    int get(int poz)
    {int rez = 0;
     for ( ; poz >= 0; poz = (poz & (poz + 1)) - 1)
          rez += tree[poz];
     return rez;
    }
public:
    FenwickTree() = default;
    FenwickTree(int n)
    {   this->n = n;   tree = vector<int>(n, 0);  }

    void inc(int poz)
    {        add(poz, 1);    }

    void dec(int poz)
    {        add(poz, -1);    }

    int get_sum(int st, int dr)
    { return get(dr) - get(st - 1); }

};

int nr[DMAX], snr[DMAX];
long long int rez[DMAX];
map<int, int> nr_transf;
FenwickTree counter;
int nrd;
long long int raspuns;

// adăugarea se poate face la dreapta sau la stânga
void add(int val, bool dreapta)
{int nr_inversiuni = 0;
 if (dreapta)
    {nr_inversiuni = counter.get_sum(val+1, nrd-1); }
     else
     nr_inversiuni = counter.get_sum(0, val-1);
 raspuns += nr_inversiuni;
 counter.inc(val);
}
// eliminăm din stânga
void remove(int val)
{int inv_crt = counter.get_sum(0, val - 1);
 raspuns -= inv_crt;
 counter.dec(val);
}

int block_size;
struct interogare {int st, dr, nr;};
interogare qo[DMAX];
bool mo_cmp(interogare x, interogare y)
{int block_x = x.st/block_size;
 int block_y = y.st/block_size;
 if (block_x == block_y) return x.dr < y.dr;
 return block_x < block_y;
}
```

```
int main()
{cin.sync_with_stdio(false);
 int n, q, i;
 cin >> n >> q;
 for (i = 0; i < n; i++) {cin >> nr[i]; snr[i]=nr[i];}
 //transformăm numerele
 sort(snr, snr+n);
 for (i = 0; i < n; i++)
     {if (nr_transf.count(snr[i]) == 0)
          nr_transf[snr[i]] = nr_transf.size();
     }
 for (i = 0; i < n; i++) nr[i] = nr_transf[nr[i]];

 nrd=nr_transf.size()+1;
 counter = FenwickTree(nrd);

 //sortăm interogările
 block_size = (int)(sqrt(n)) + 1;
 for (i = 0; i < q; i++)
     {cin >> qo[i].st >> qo[i].dr;
      qo[i].st--; qo[i].dr--; qo[i].nr=i; }
 sort(qo, qo+q, mo_cmp);

 int prev_block = -1, mo_st = 0, mo_dr = 0;
 for (i = 0; i < q; i++)
      {
       if (qo[i].st / block_size != prev_block)
            {raspuns=0;
             counter=FenwickTree(nrd);
             mo_st = qo[i].st; mo_dr = qo[i].st - 1;
            }

        prev_block = qo[i].st / block_size;
        while (mo_dr < qo[i].dr)
              add(nr[++mo_dr], true);
        while (mo_st > qo[i].st)
              add(nr[--mo_st], false);
        while (mo_st < qo[i].st)
              remove(nr[mo_st++]);
        rez[qo[i].nr] = raspuns;
    }
    for (i = 0; i < q; i++) cout << rez[i] << "\n";
    return 0;
}
```

8.3.6 Problema Calafat

Se dă un șir format din N numere naturale. Pentru fiecare valoare distinctă dintr-o subsecvență cuprinsă între 2 indici st și dr considerăm distanța dintre indicii primei și ultimei apariții ale acesteia în cadrul subsecvenței. Dându-se M subsecvențe de forma $[st, dr]$, se cere să se calculeze suma distanțelor corespunzătoare tuturor valorilor distincte din subsecvență.

Date de intrare

Fișierul de intrare `calafat.in` conține pe prima linie două numere naturale N și M. Pe a doua linie se vor afla cele N numere din șirul dat. Pe următoarele M linii se vor afla câte două numere st și dr, cu semnificația că vrem să calculăm suma menționată mai sus pentru subsecvența $[st, dr]$.

Date de ieșire

Fișierul de ieșire `calafat.out` va conține M numere naturale, câte unul pe fiecare linie, reprezentând cele M sume cerute.

Restricții și precizări

- $1 \leq N, M \leq 200\,000$
- $1 \leq st \leq dr \leq N$
- Valorile din șir se vor afla în intervalul $[1, N]$

Exemple

`calafat.in`	`calafat.out`	Explicații
7 3 1 3 1 2 2 1 3 2 4 2 7 3 6	0 9 4	În prima subsecvență fiecare valoare apare o singură dată, deci suma diferențelor este 0. În a doua subsecvență: Valoarea 3 apare la indicii 2 și 7 rezultând diferența $7 - 2 = 5$ Valoarea 1 apare la indicii 3 și 6, deci diferența $6 - 3 = 3$ Valoarea 2 apare la indicii 4 și 5, deci diferența $5 - 4 = 1$ Suma diferențelor este 9. În a treia subsecvență: Valoarea 1 apare la indicii 3 și 6, deci diferența $6 - 3 = 3$ Valoarea 2 apare la indicii 4 și 5, deci diferența $5 - 4 = 1$ Suma diferențelor este 4.

Soluție de 70 de puncte

O soluție de 70 de puncte se poate implementa cu algoritmul lui Mo.

Implementare

```
#include <bits/stdc++.h>
#define DMAX 200200
#define SQMAX 500
using namespace std;
```

```
ifstream fin("calafat.in");
ofstream fout("calafat.out");
long long int rezultat;
struct interogare { int st, dr, nr; };
interogare Q[DMAX];
long long int rez[DMAX];
int prim[DMAX]; //prim[i]=poziția în ap[i] a primei apariții a valorii i
int ultim[DMAX];//ultim[i]=poziția în ap[i] a ultimei apariții a lui i
int a[DMAX];
vector<int> ap[DMAX]; //ap[i] conține toate aparițiile valorii i

void add(int index, bool unde)
//dacă unde este 1 adăugarea se face la dreapta
{int x=a[index];
 if (unde) //adăugăm la dreapta valoarea x
    {if (prim[x]==-1)
        {prim[x]=ultim[x]=(lower_bound(ap[x].begin(),ap[x].end(),index)-
                           ap[x].begin());}
         else
         {rezultat=rezultat-(ap[x][ultim[x]]-ap[x][prim[x]]);
          ultim[x]++;
          rezultat=rezultat+(ap[x][ultim[x]]-ap[x][prim[x]]);
         }
    }
    else //la stânga
    if (prim[x]==-1)
       {prim[x]=ultim[x]=(lower_bound(ap[x].begin(),ap[x].end(),index)-
                          ap[x].begin());}
       else
       {rezultat=rezultat-(ap[x][ultim[x]]-ap[x][prim[x]]);
        prim[x]--;
        rezultat=rezultat+(ap[x][ultim[x]]-ap[x][prim[x]]);
       }
}

void remove(int index, bool unde)
{int x=a[index];
 if (unde) //eliminăm din dreapta
    {if (prim[x]==ultim[x])
        prim[x]=ultim[x]=-1;
        else
        {rezultat=rezultat-(ap[x][ultim[x]]-ap[x][prim[x]]);
         ultim[x]--;
         rezultat=rezultat+(ap[x][ultim[x]]-ap[x][prim[x]]);
        }
    }
    else //eliminăm din stânga
    {if (prim[x]==ultim[x])
        prim[x]=ultim[x]=-1;
        else
        {rezultat=rezultat-(ap[x][ultim[x]]-ap[x][prim[x]]);
         prim[x]++;
         rezultat=rezultat+(ap[x][ultim[x]]-ap[x][prim[x]]);
        }
    }
}
int block_size;
```

```
bool mo_cmp(interogare x, interogare y)
{int block_x = x.st/block_size;
 int block_y = y.st/block_size;
 if (block_x == block_y)
    return x.dr < y.dr;
 return block_x < block_y;
}
int main()
{ios_base::sync_with_stdio(false);
 int n, q, i, st, dr;
 fin>>n>>q;
 for (i=0; i<n; i++)
     {fin>>a[i]; ap[a[i]].push_back(i);}
 for (i=1; i<=n; i++) prim[i]=ultim[i]=-1;
 for (i=0; i<q; i++)
     {fin>>Q[i].st>>Q[i].dr;  Q[i].nr=i;  Q[i].st--;  Q[i].dr--;   }
  block_size=(int)(sqrt(n))+1;
  sort(Q, Q+q, mo_cmp);
  int mo_st=0, mo_dr=-1;
  for (i=0; i<q; i++)
      {st=Q[i].st,dr=Q[i].dr;
       if (mo_dr==-1) mo_dr=mo_st-1;
       while (mo_dr<dr)add(++mo_dr, 1);
       while(mo_st>st)  add(--mo_st, 0);
       while(mo_st<st)  remove(mo_st++,0);
       while(mo_dr>dr) remove(mo_dr--,1);
       rez[Q[i].nr]=rezultat;
    }
  for (i=0; i<q; i++) fout<<rez[i]<<'\n';
  return 0;
}
```

8.4 Probleme propuse

- Problema Fsecv
- Problema Chef and Graph Queries
- Problema Powerful array
- Problema Snow White and the N dwarfs

8.5 Bibliografie

[1] Mike Koltsov, *Mo's Algorithm*, URL: https://www.hackerearth.com/practice/notes/mos-algorithm/.

[2] Antti Laaksonen, *Competitive Programmer's Handbook*, 2018, cap. III.27.3, URL: https://cses.fi/book/book.pdf.

[3] *Square root decomposition and applications*, URL: https://codeforces.com/blog/entry/83248.

[4] *Mo's Algorithm*, URL: https://www.hackerrank.com/topics/mos-algorithm.

Capitolul 9

Recurențe liniare și exponențierea matricilor

prof. Szabó Zoltan
Inspectoratul Școlar Județean Mureș

9.1 Aplicații

9.1.1 Problema Fibonacci

Se consideră șirul Fibonacci, definit astfel:

$$f_1 = f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{ dacă } n > 2$$

Se cere valoarea termenului Fibonacci cu indicele n, modulo $1\,000\,000\,007$.

Restricții și precizări

- $1 \leq n \leq 1\,000\,000\,000$

Soluție

Soluția banală, de complexitate $\mathcal{O}(n)$, arată în felul următor:

```
int n, i, a, b, c;
const int MOD=1000000007;
cin>>n;
a=1; b=0;
for (i=1; i<=n; i++)
  {c=(a+b)%MOD;
    a=b;
    b=c;
  }
cout<<c;
}
```

Această soluție nu se va încadra în timp rezonabil pentru valori foarte mari ale lui n. În continuare vom căuta o soluție de complexitate $\mathcal{O}(\log n)$ folosindu-ne de exponențierea rapidă a matricilor.

Pentru orice $n > 2$, termenul al n-lea se calculează ca sumă a celor doi termeni anteriori cunoscuți. Pentru modelul matematic vom defini o matrice pătratică A constantă, (A se va numi **matrice generatoare**) și un șir de matrici coloană F_i cu elemente consecutive ale șirului lui Fibonacci, astfel încât după înmulțire să obținem o nouă matrice coloană cu următoarele elemente ale șirului lui Fibonacci. Matricile coloană F vor forma un șir indexat cu numerele $0, 1, 2, \ldots, n$.

$$F_0 = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}, F_1 = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, F_2 = \begin{bmatrix} f_2 \\ f_3 \end{bmatrix}, \ldots F_{n-1} = \begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix}, F_n = \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix}$$

Vom construi matricea generatoare A astfel încât pentru orice număr natural i să avem: $A \times F_i = F_{i+1}$

Deducem relativ ușor că $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$

Într-adevăr $A \times F_1 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 0 + f_2 \\ f_1 + f_2 \end{bmatrix} = \begin{bmatrix} f_2 \\ f_3 \end{bmatrix} = F_2$ Astfel avem următoarele relații:

- $A \times F_0 = F_1$
- $A \times F_1 = F_2$
- ...
- $A \times F_{n-1} = F_n$

De aici deducem că: $F_n = A \times F_{n-1} = A \times A \times F_{n-2} = A \times A \times A \times F_{n-3} = A^3 \times F_{n-3} = A^4 \times F_{n-4} = \cdots = A^n \times F_{n-n} = A^n \times F_0$

Exponențierea matricilor se poate realiza în timp logaritmic, folosind următoarea relație de recurență:

$$A^n = \begin{cases} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, & \text{dacă } n = 0 \\ A, & \text{dacă } n = 1 \\ A^{2^{\frac{n}{2}}}, & \text{dacă } n \text{ par}, n > 1 \\ A \times A^{2^{\frac{(n-1)}{2}}}, & \text{dacă } n \text{ impar}, n > 1 \end{cases} \tag{9.1}$$

Implementare

```
#include <iostream>
#define DIM 3
using namespace std;

int n, d;
```

```
long long int F[DIM], a[DIM][DIM], b[DIM][DIM], t;
const long long int MOD = 1000000007;

void unitate(long long int d,long long int b[][DIM])
//construiește matricea unitate
{int i, j;
 for (i=1; i<=d; i++)
     {for (j=1; j<=d; j++)
           b[i][j]=0;
      b[i][i]=1;
     }
}

void atrib(int d, long long int a[][DIM], long long int b[][DIM])
//atribuie matricii a matricea b
{int i, j;
 for (i=1; i<=d; i++)
     for (j=1; j<=d; j++)
          a[i][j]=b[i][j];
}

void inmult(int d, long long int a[][DIM], long long int b[][DIM], long long int c[][DIM])
//construiește matricea c ca rezultat al înmulțirii matricilor a și b
{int i, j, k;
 for (i=1; i<=d; i++)
     for (j=1; j<=d; j++)
         {
          c[i][j]=0;
          for (k=1; k<=d; k++)
               c[i][j]=(c[i][j]+(a[i][k]*b[k][j])% MOD) % MOD;
         }
}

void putere (int n, int d, long long int a[][DIM], long long int b[][DIM])
//construiește în matricea b pe a la puterea n
{long long int c[DIM][DIM];
 unitate(2,b);
 while (n)
       {
        if (n%2==1)
            {
            inmult(d,a,b,c);
            atrib(d,b,c);
            }
        inmult(d,a,a,c);
        atrib(d,a,c);
        n=n/2;
       }
}

int main()
{
    cin>>n;
    F[0]=0;F[1]=1;
    a[1][1]=0;a[1][2]=1;
    a[2][1]=1;a[2][2]=1;
    putere(n,2,a,b);
```

```
    t=((b[1][1]*F[0])% MOD + (b[1][2]*F[1])% MOD) % MOD;
    cout<<t<<"\n";
    return 0;
}
```

9.1.2 Problema Șir recurent

Se consideră șirul recurent s, definit astfel:

$$s_n = a \cdot s_{n-1} + b \cdot s_{n-2} + c \cdot s_{n-3}, \text{ dacă } n > 3$$

Cunoscând primii 3 termeni ai șirului s_1, s_2, s_3, coeficienții a, b, c, precum și numerele n și M, se cere să se determine al n-lea termen al șirului modulo M.

Restricții și precizări

- $1 \leq n \leq 1\,000\,000\,000$

Soluție

Vom construi matricea generatoare A și un șir de matrici coloană S_i cu elemente consecutive ale șirului lui s, astfel încât după înmulțire să obținem o nouă matrice coloană cu următoarele elemente ale șirului s. Deoarece recurența este de ordin 3, matricile coloană vor avea 3 elemente, iar matricea A va avea 3 linii și 3 coloane.

Construcția acestor matrici se realizează astfel:

1. $S_i = \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \end{bmatrix}$
2. Considerăm matricea $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ c & b & a \end{bmatrix}$
3. $A \times S_i = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ c & b & a \end{bmatrix} \times \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \end{bmatrix} = \begin{bmatrix} s_{i+1} \\ s_{i+2} \\ c \cdot s_i + b \cdot s_{i+1} + a \cdot s_{i+2} \end{bmatrix} = S_{i+1}$, pentru orice i
4. Deoarece $A \times S_i = S_{i+1}$, deducem în mod similar cu problema precedentă că $S_n = A^{n-1} \times S_1$

9.1.3 Problema Recurență cu constantă

Se consideră șirul recurent s, definit astfel:

$$s_n = a \cdot s_{n-1} + b \cdot s_{n-3} + c \cdot s_{n-5} + d, \text{ dacă } n > 4$$

Cunoscând primii 5 termeni ai șirului s_0, s_1, s_2, s_3, s_4, coeficienții a, b, c, d și valorile n și M, să se determine termenul șirului cu indicele n modulo M.

Restricții și precizări

- $1 \leq n \leq 1\,000\,000\,000$

Soluție

Observăm că în formula șirului recursiv apare un termen constant. Putem construi un model matriceal în două moduri diferite.

Varianta 1. Vom construi matricea generatoare A și un șir de matrici coloană S_i cu elemente consecutive ale șirului lui s, astfel încât după înmulțire să obținem o nouă matrice coloană cu următoarele elemente ale șirului s. Deoarece recurența este de ordin 5, matricile coloană vor avea 5 elemente, la care adăugăm și constanta d, iar matricea A va avea 6 linii și 6 coloane.

Construcția acestor matrici se realizează astfel:

1. $S_i = \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \\ s_{i+3} \\ s_{i+4} \\ d \end{bmatrix}$
2. Considerăm matricea $A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ c & 0 & b & 0 & a & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
3. $A \times S_i = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ c & 0 & b & 0 & a & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \\ s_{i+3} \\ s_{i+4} \\ d \end{bmatrix} = \begin{bmatrix} s_{i+1} \\ s_{i+2} \\ s_{i+3} \\ s_{i+4} \\ c * s_i + b * s_{i+2} + a * s_{i+4} + d \\ d \end{bmatrix} = S_{i+1},$ pentru orice i
4. Deoarece $A \times S_i = S_{i+1}$, deducem în mod similar cu problema precedentă că $S_n = A^n \times S_0$

Varianta 2. Eliminăm constanta din formulă, scriind formula a doi termeni consecutivi și scăzând termenii din cele două părți în cele două ecuații.

1. Scriem ecuațiile pentru doi termeni consecutivi:

$$s_n = a \cdot s_{n-1} + b \cdot s_{n-3} + c \cdot s_{n-5} + d$$
$$s_{n-1} = a \cdot s_{n-2} + b \cdot s_{n-4} + c \cdot s_{n-6} + d$$

2. Scăzând a doua ecuație din prima, obținem:

$$s_n = (a+1) \cdot s_{n-1} - a \cdot s_{n-2} + b \cdot s_{n-3} - b \cdot s_{n-4} + c \cdot s_{n-5} - c \cdot s_{n-6}$$

3. Construim corespunzător matricea generatoare A cu 6 linii și 6 coloane, precum și matricile coloana S_i având 6 termeni consecutivi din șirul s.

4. $S_i = \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \\ s_{i+3} \\ s_{i+4} \\ s_{i+5} \end{bmatrix}$

5. Considerăm matricea $A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -c & c & -b & b & -a & a+1 \end{bmatrix}$

6. Deoarece $A \times S_i = S_{i+1}$, deducem în mod similar cu problema precedentă că $S_n = A^n \times S_0$

9.1.4 Problema Recurență indirectă pe șiruri liniare

Se consideră șirurile recurente a, b și c definite astfel:

$$\begin{aligned} a_n &= a_{n-1} + b_{n-1} + c_{n-2} \\ b_n &= a_{n-1} + b_{n-2} + c_{n-3} \\ c_n &= a_{n-2} + 2 \cdot b_{n-3} + c_{n-1} \end{aligned}$$

Cunoscând termenii a_0, a_1, b_0, b_1, b_2, c_0, c_1, c_2, precum și valorile n și M, să se determine termenii cu indicele n din șirurile a, b și c modulo M.

Restricții și precizări

- $1 \leq n \leq 1\,000\,000\,000$

Soluție

Valoarea lui a_2 se calculează cu ajutorul relației de recurență. În pasul următor construim relația matriceală care va permite calculul termenilor de indice n din toate cele 3 șiruri:

1. $S_i = \begin{bmatrix} a_i \\ a_{i+1} \\ a_{i+2} \\ b_i \\ b_{i+1} \\ b_{i+2} \\ c_i \\ c_{i+1} \\ c_{i+2} \end{bmatrix}$

2. Considerăm matricea $A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

3. Deoarece $A \times S_i = S_{i+1}$, deducem în mod similar cu problema precedentă că $S_n = A^n \times S_0$

9.2 Probleme propuse

- Problema Iepuri
- Problema 2sah
- Problema Ikebana

Partea a III-a

Algoritmi pe grafuri

Capitolul 10

Puncte de articulație, punți și componente biconexe

prof. Emanuela Cerchez
Colegiul Național „Emil Racoviță” Iași
Centrul Județean de Excelență Iași

10.1 Definiții

Definiția 10.1. Fie $G = (V, E)$ un graf neorientat conex. Vârful $v \in V$ se numește **punct de articulație** dacă subgraful obținut prin eliminarea vârfului v și a muchiilor incidente cu acesta nu mai este conex.

De exemplu, pentru graful din Figura 10.1 punctele de articulație sunt 1, 3, 5, 7.

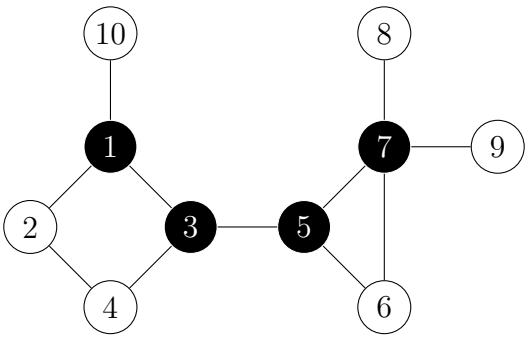

Figura 10.1: Un graf cu punctele de articulație marcate cu negru.

Definiția 10.2. Un graf se numește **biconex** dacă nu are puncte de articulație.

În multe aplicații practice, ce se pot modela cu ajutorul grafurilor, nu sunt de dorit punctele de articulație. De exemplu, într-o rețea de telecomunicații, dacă o centrală dintr-un punct de articulație se defectează rezultatul este nu doar întreruperea comunicării cu centrala respectivă ci și cu alte centrale.

Definiția 10.3. O **componentă biconexă** a unui graf este un subgraf biconex maximal cu această proprietate.

Definiția 10.4. Se numește **punte** o componentă biconexă formată dintr-o singură muchie.

De exemplu, pentru graful din Figura 10.1 componentele biconexe sunt:

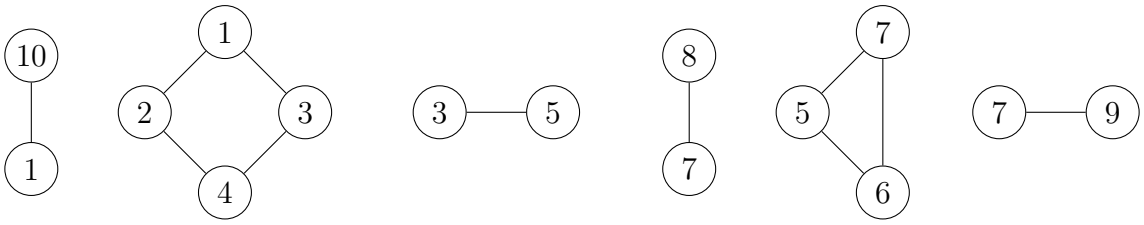

Figura 10.2: Componentele biconexe ale grafului din Figura 10.1.

Observații

1. Componentele biconexe ale unui graf reprezintă o partiție a mulțimii muchiilor grafului.
2. Punctele de articulație aparțin la cel puțin două componente biconexe.

10.2 Descompunerea unui graf în componente biconexe

Pentru a descompune graful în componente biconexe vom utiliza proprietățile parcurgerii DFS. Parcurgând graful DFS putem clasifica muchiile grafului în:

- muchii care aparțin arborelui parțial DFS (*tree edges*);
- muchii $[u, v]$ care nu aparțin arborelui și care unesc vârful u cu un strămoș al său v în arborele parțial DFS numite muchii de întoarcere (*back edges*). Acestea sunt marcate în exemplu punctat.

De exemplu graful precedent poate fi redesenat, clasificând muchiile ținând cont de arborele parțial DFS cu rădăcina 3:

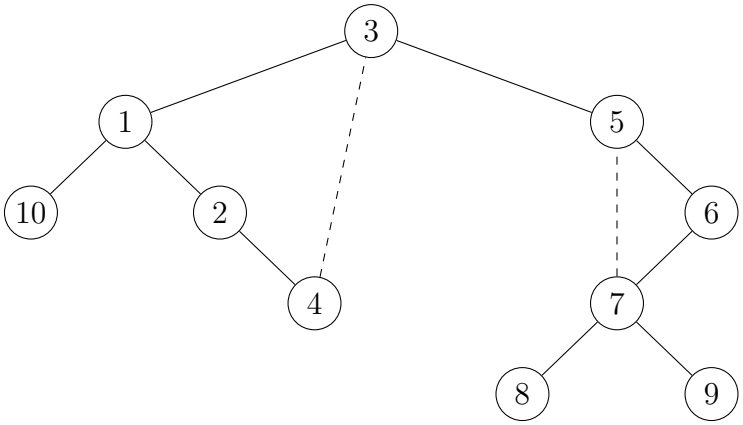

Figura 10.3: Arborele DFS al grafului din Figura 10.1.

Observăm că rădăcina arborelui parțial DFS este punct de articulație dacă și numai dacă are cel puțin doi descendenți, între vârfuri din subarbori diferiți ai rădăcinii neexistând muchii. Mai mult, un vârf x oarecare nu este punct de articulație dacă și numai dacă din orice fiu y al lui x poate fi atins un strămoș al lui x pe un lanț format din descendenți ai lui x și o muchie de întoarcere (un drum „de siguranță" între x și y).

Pentru fiecare vârf x al grafului definim $dfn(x)$ = numărul de ordine al vârfului x în parcurgerea DFS a grafului (*depth-first-number*).

De exemplu

x	1	2	3	4	5	6	7	8	9	10
$dfn(x)$	2	4	1	5	6	7	8	9	10	3

Tabela 10.1: Valorile $dfn(x)$ pentru arborele DFS din figura 10.3.

Observăm că dacă x este un strămoș al lui y în arborele parțial DFS atunci $dfn(x) < dfn(y)$.

Pentru fiecare vârf x din graf definim $low(x)$ = numărul de ordine al primului vârf din parcurgerea DFS ce poate fi atins din x pe un alt lanț decât lanțul unic din arborele parțial DFS.

$$low(x) = \min \begin{cases} dfn(x), \\ \min\{low(y) \mid y \text{ fiu al lui } x\}, \\ \min\{dfn(y) \mid [x,y] \text{ muchie de întoarcere}\} \end{cases}$$

De exemplu

x	1	2	3	4	5	6	7	8	9	10
$dfn(x)$	2	4	1	5	6	7	8	9	10	3
$low(x)$	1	1	1	1	6	6	6	9	10	3

Tabela 10.2: Valorile $dfn(x)$ și $low(x)$ pentru arborele DFS din figura 10.3.

Caracterizăm punctele de articulație dintr-un graf astfel: x este punct de articulație dacă și numai dacă este rădăcina unui arbore parțial DFS cu cel puțin doi descendenți sau, dacă nu este rădăcină, are un fiu y astfel încât $low(y) \geq dfn(x)$.

Pentru exemplul din figură nodul 3 este punct de articulație deoarece este rădăcina arborelui parțial DFS și are doi descendenți, nodul 7 este punct de articulație deoarece $low(8) = 9 \geq dfn(7) = 8$, nodul 5 este punct de articulație deoarece $low(6) = 6 \geq dfn(5) = 6$, iar nodul 1 este punct de articulație deoarece $low(10) = 3 \geq dfn(1) = 2$.

10.2.1 Reprezentarea informațiilor

Vom reprezenta graful prin liste de adiacență alocate dinamic utilizând clasa `vector` din STL.

Vectorii `dfn[]` și `low[]` conțin valorile calculate în timpul parcurgerii DFS.

Variabila globală `num` este utilizată pentru a calcula numărul de ordine al vârfului curent în parcurgerea în adâncime.

Variabila globală `nrfii` reține numărul de fii ai vârfului de start în arborele parțial DFS.

Vom utiliza o stivă `S` în care vom reține muchiile din graf (atât cele care aparțin arborelui cât și cele de întoarcere) în ordinea în care sunt întâlnite în timpul parcurgerii. Atunci când identificăm o componentă biconexă, mai exact când identificăm un nod u care are un fiu x astfel încât $low(x) \geq dfn(u)$, eliminăm din stivă toate muchiile din componenta biconexă respectivă. Funcția `Initializare()` inițializează stiva cu o muchie fictivă (cu extremitatea finală egală cu vârful de start și extremitatea inițială -1, un nod inexistent).

Vom reprezenta mulțimea punctelor de articulație identificate în timpul parcurgerii în adâncime prin vectorul caracteristic `Art` (`Art[i] = 1` , dacă `i` este punct de articulație și `0` în caz contrar).

În funcția `main()` , după citire și inițializare, vom apela funcția `Biconex(start, -1)` care realizează descompunerea grafului în componente biconexe, apoi afișăm punctele de articulație.

```
#include <bits/stdc++.h>
#define NMAX 100100
using namespace std;
ifstream fin("biconex.in");
ofstream fout("biconex.out");
int n;       //numărul de vârfuri din graf
int start=1; //vârful de start în parcurgerea DFS
int num;     //numărul de ordine al vârfului curent în parcurgerea DFS
int nrfii;   //numărul de fii ai vârfului de start
int nr;      //numărul de componente biconexe
vector<int> G[NMAX]; //reprezentarea grafului prin liste de adiacență
int dfn[NMAX], low[NMAX];
bool Art[NMAX];
//vectorul caracteristic al mulțimii punctelor de articulație
struct Muchie {int f,  t;};
stack<Muchie> S;
void Citire();
void Biconex(int ,int );
void Initializare();
void Afisare_Comp_Biconexa(int, int);

int main()
{int i;
```

```
  Citire();
  Initializare();
  Biconex(start,-1);
  //Afișez punctele de articulatie
  if (nrfii>1) //start este punct de articulație
    Art[start]=1;
  fout<<"Punctele de articulatie sunt: ";
  for (i=1; i<=n; i++)
    if (Art[i]) fout<<i<<' ';
  return 0;
}
void Citire()
{int x, y, m, i;
  fin>>n>>m;
  for (i=0; i<m; i++)
    {fin>>x>>y;
      G[x].push_back(y); G[y].push_back(x);
    }
}

void Initializare()
{int i;
  for (i=1; i<=n; i++) dfn[i]=low[i]=-1;
  Muchie fictiv={start, -1};
  S.push(fictiv);
}

void Biconex(int u, int tu)
//u este nodul curent; tu este nodul părinte al lui u
{int x, p;
  Muchie m;
  dfn[u]=low[u]=++num;
  //parcurg lista de adiacență a nodului u
  for (p=0; p<G[u].size(); p++)
    {x=G[u][p];  //x este un nod adiacent cu u
      if (x!=tu && dfn[x]<dfn[u])
        //inserează în stiva S muchia [u,x]
        {m.f=x; m.t=u; S.push(m); }
      if (dfn[x]==-1) //x nu a mai fost vizitat
        {if (u==start) //am găsit un fiu al vârfului start
            nrfii++;
          Biconex(x,u);
          low[u]=min(low[u],low[x]);
          if (low[x]>=dfn[u]) //u este punct de articulație
            //am identificat o componentă biconexă,
            //formată din muchiile din stiva S până la
            //întâlnirea muchiei [u,x]
            { if (u!=start) Art[u]=1;
              Afisare_Comp_Biconexa(x, u); }
        }
      else //x a mai fost vizitat
        if (x!=tu)
          //x nu este tatăl lui u,
          //deci [u,x] e muchie de întoarcere de la u la x
          low[u]=min(low[u],dfn[x]);
    }
}
```

```
void Afisare_Comp_Biconexa(int x, int u)
//afișează componenta biconexă a muchiei [x,u]
{Muchie p;
  nr++;      //incrementez numărul de componente biconexe
  fout<<"Componenta biconexă "<<nr<<" conține muchiile:\n";
  do
    {p=S.top(); S.pop(); //extrag o muchie din stivă
      fout<<p.t<<' '<<p.f<<'\n';  }
  while (p.t!=u || p.f!=x);
}
```

Teorema 1. Apelul funcției `Biconex(start, -1)` generează componentele biconexe ale grafului conex G.

Demonstraţie. Vom lua în considerare cazul în care n, numărul de vârfuri din G, este cel puțin 2 (dacă $n = 1$, G are o singură componentă biconexă, formată dintr-un singur vârf).

Vom face demonstrația prin inducție după B, numărul de componente biconexe.

P(1): $B = 1 \equiv$ graful este biconex: rădăcina start a arborelui parțial DFS va avea un singur fiu, să-l notăm x. Apelul `Biconex(x, start)` a explorat toate muchiile grafului și le-a introdus în stiva `S`. Cum x este singurul vârf pentru care $low[x] \geq dfn[start]$, muchiile grafului vor fi afișate într-o singură componentă biconexă.

Pasul inductiv:

P(B): Să presupunem acum că algoritmul funcționează corect pentru toate grafurile conexe cu cel mult B componente biconexe.

P($B+1$): Vom demonstra că algoritmul funcționează pentru toate grafurile conexe cu cel mult $B + 1$ componente biconexe.

Fie G un graf cu $B+1$ componente biconexe și fie x primul vârf pentru care este îndeplinită condiția $low[x] \geq dfn[u]$. Până în acest moment nu a fost identificată nici o componentă biconexă. În urma apelului `Biconex(x, u)`, toate muchiile incidente cu descendenți ai lui x se află în stiva `S`, deasupra muchiei $[u, x]$. Cum x este primul vârf care îndeplinește condiția $low[x] \geq dfn[u]$, deducem că descendenții lui x nu sunt puncte de articulație. Cum u este punct de articulație, rezultă că muchiile situate în `S`, deasupra muchiei $[u, x]$, inclusiv, formează o componentă biconexă. Muchiile acestei componente biconexe sunt extrase din stivă și afișate, deci, de la acest pas algoritmul revine la aflarea componentelor biconexe ale unui graf G, obținut din G prin eliminarea unei componente biconexe. Cum G are B componente biconexe, conform ipotezei inductive, algoritmul determină corect componentele biconexe ale lui G. □

Complexitate

Complexitatea algoritmului de descompunere în componente biconexe a unui graf reprezentat prin listele de adiacență este dată de parcurgerea DFS, deci este $\mathcal{O}(n + m)$, unde n este numărul de vârfuri, iar m numărul de muchii din graf.

10.3 Exerciții propuse

1. Care sunt punctele de articulație ale grafului din figura următoare? Descompuneți graful următor în componente biconexe.

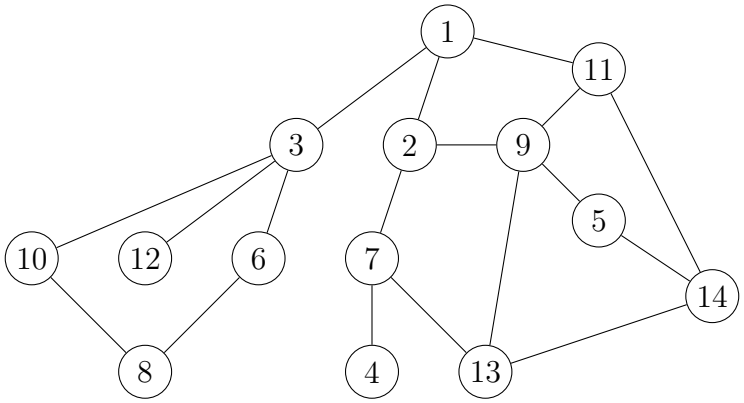

Figura 10.4

2. Adăugați un număr minim de muchii în graful următor astfel încât să obțineți un graf biconex.

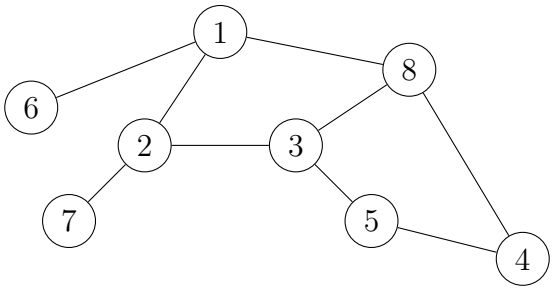

Figura 10.5

3. Scrieți un program care să identifice punțile unui graf.

10.4 Aplicații

10.4.1 Problema Pământ (ONI 2011, clasele XI-XII)

Pe pământul din apropierea localității Gheorgheni s-au întâlnit toți copiii și doresc organizarea unui joc mai deosebit. Copiii au fost numerotați de la 1 la N și știm care sunt prietenii fiecărui copil.

O echipă este un grup maximal de copii cu proprietatea că oricare ar fi copiii P și Q din echipă, există un șir de copii $C_1, \ldots, C_k$ astfel încât $P = C_1$, $Q = C_k$, și oricare ar fi $1 \leq i < k$, C_i este prieten cu C_{i+1}.

Fiecare echipă va primi un cod, egal cu cel mai mic număr de ordine al unui copil din echipa respectivă.

Dorim să aflăm care sunt copiii vulnerabili, adică acei copii care dacă ar fi eliminați ar duce la spargerea echipei sale în două sau mai multe echipe.

Cerință

Scrieți un program care să identifice toate echipele formate conform regulilor de mai sus, precum și care sunt copiii vulnerabili.

Date de intrare

Fișierul de intrare `pamant.in` conține pe prima linie două numere naturale N și M reprezentând numărul de copii și respectiv numărul relațiilor de prietenie. Următoarele M linii conțin câte două numere naturale distincte x și y, cu semnificația că x și y sunt numerele de ordine a doi copii în relație de prietenie.

Date de ieșire

Prima linie a fișierului de ieșire `pamant.out` conține o singură valoare naturală A, reprezentând numărul de echipe. A doua linie conține A numere naturale separate prin câte un spațiu reprezentând codurile echipelor, în ordine crescătoare. A treia linie conține o valoare naturală B reprezentând numărul de copii vulnerabili. A patra linie conține B valori naturale, separate prin câte un spațiu, reprezentând numerele de ordine, scrise în ordine crescătoare, ale copiilor vulnerabili.

Restricții și precizări

- $1 \leq N \leq 100\,000$
- $1 \leq M \leq 200\,000$
- Relațiile de prietenie sunt reciproce: dacă x este prieten cu y, atunci și y este prieten cu x.
- Dacă x este prieten cu y și y este prieten cu z nu înseamnă că x este prieten cu z.

Exemple

`pamant.in`	`pamant.out`	Explicații

`10 7`
`1 2`
`2 8`
`4 6`
`3 5`
`3 10`
`5 10`
`5 7`
	`4`
	`1 3 4 9`
	`2`
	`2 5`
		Există 4 echipe și anume:
		prima echipă: 1 2 8
		a doua echipă: 3 5 7 10
		a treia echipă: 4 6
		a patra echipă: 9
		Există doi copii speciali și anume 2 și 5.

Soluție

Problema presupune descompunerea grafului în componente conexe, iar pentru fiecare componentă conexă identificarea punctelor de articulație.

Implementare

```
#include <bits/stdc++.h>
#define NMAX 100100
#define MMAX 200100
using namespace std;
ifstream fin("pamant.in");
ofstream fout("pamant.out");

int n;       //numărul de vârfuri din graf
int start;   //vârful de start în parcurgerea DFS
int num;     //numărul de ordine al vârfului curent în parcurgerea DFS
int nrfii;   //numărul de fii ai vârfului de start
int nrc;     //numărul de componente conexe
int nra;     //numărul de puncte de articulație

vector<int> G[NMAX]; //reprezentarea grafului prin liste de adiacență
int dfn[NMAX], low[NMAX];
bool Art[NMAX];
//vectorul caracteristic al mulțimii punctelor de articulație
bool cc[NMAX];
//vectorul caracteristic al reprezentanților componentelor conexe

void Citire();
void Biconex(int ,int );
void Afisare();

int main()
{Citire();
  for (start=1; start<=n; start++)
    if (dfn[start]==-1)
      {nrc++; nrfii=0;
        cc[start]=1;
        Biconex(start,-1);
        if (nrfii>1) //start este punct de articulație
          Art[start]=1;
      }
  Afisare();
  return 0;
}

void Citire()
{int x, y, m, i;
  fin>>n>>m;
  for (i=0; i<m; i++)
    {fin>>x>>y;
      G[x].push_back(y);
      G[y].push_back(x);
    }
  for (i=1; i<=n; i++) dfn[i]=low[i]=-1;
}

void Biconex(int u, int tu)
//u este nodul curent; tu este nodul părinte al lui u
{int x, p;
  dfn[u]=low[u]=++num;
  //parcurg lista de adiacență a nodului u
```

```
  for (p=0; p<G[u].size(); p++)
    {x=G[u][p];  //x este un nod adiacent cu u
      if (dfn[x]==-1) //x nu a mai fost vizitat
        {if (u==start) //am gasit un fiu al vârfului start
            nrfii++;
          Biconex(x,u);
          low[u]=min(low[u],low[x]);
          if (low[x]>=dfn[u]) //u este punct de articulație
            { if (u!=start) Art[u]=1; }
        }
      else //x a mai fost vizitat
        if (x!=tu)
          //x nu este tatăl lui u,
          //deci [u,x] e muchie de întoarcere de la u la x
          low[u]=min(low[u],dfn[x]);
    }
}

void Afisare()
{int i;
  fout<<nrc<<'\n';
  for (i=1; i<=n; i++)
    if (cc[i]) fout<<i<<' ';
  fout<<'\n';
  for (i=1; i<=n; i++)
    if (Art[i]) nra++;
  fout<<nra<<'\n';
  for (i=1; i<=n; i++)
    if (Art[i]) fout<<i<<' ';
  fout<<'\n';
}
```

10.4.2 Problema Police Query (Croatian Olympiad in Informatics 2006)

Pentru a prinde infractorii, poliția vrea să introducă un nou sistem computerizat. Zona pe care acest sistem o va acoperi este constituită din N orașe (numerotate de la 1 la N), conectate prin E drumuri bidirecționale. Pentru a prinde infractorii, deseori poliția trebuie să se deplaseze dintr-un oraș în altul. În plus, polițiștii, analizând harta, trebuie să decidă unde să instaleze baricade și să blocheze drumuri. Prin urmare, noul sistem trebuie să răspundă la următoarele două tipuri de interogări:

Interogare	Semnificație	Restricții
`1 A B X Y`	Pot infractorii să ajungă din orașul A în orașul B dacă poliția blochează drumul dintre $X i Y$ și infractorii nu pot trece pe acest drum?	$A \neq B$ există drumul $[X, Y]$
`2 A B C`	Pot infractorii să ajungă din orașul A în orașul B dacă poliția izolează orașul C și infractorii nu pot trece prin acesta?	$A \neq B, A \neq C, C \neq B$

Scrieți un program care să implementeze un astfel de sistem.

Date de intrare

Prima linie conține numerele naturale N și E, separate prin spațiu, reprezentând numărul de orașe și numărul de drumuri. Fiecare dintre următoarele E linii conține două numere naturale distincte cuprinse între 1 și N, separate prin spațiu, reprezentând drumurile. Următoarea linie conține numărul natural Q reprezentând numărul de interogări. Fiecare dintre următoarele Q linii conține câte o interogare, în forma descrisă în enunț.

Date de ieșire

Fișierul de ieșire va conține Q linii, pe fiecare linie fiind scris răspunsul pentru o interogare, în ordinea din fișierul de intrare. Răspunsul poate fi `yes` sau `no`.

Restricții și precizări

- $2 \leq N \leq 10^5$
- $1 \leq E \leq 5 \cdot 10^5$
- $1 \leq Q \leq 10^5$
- Se garantează că inițial este posibil să ajungi din orice oraș în orice alt oraș utilizând drumurile existente.

Exemple

Intrare	Ieșire
13 15 1 2 2 3 3 5 2 4 4 6 2 6 1 4 1 7 7 8 7 9 7 10 8 11 8 12 9 12 12 13 5 1 5 13 1 2 1 6 2 1 4 1 13 6 7 8 2 13 6 7 2 13 6 8	yes yes yes no yes

Soluție

Evident, vom asocia problemei un graf neorientat, în care orașele corespund vârfurilor, iar drumurile corespund muchiilor. Se garantează că graful este conex.

Pentru interogările de tip 1 răspunsul va fi `no` dacă și numai dacă muchia $[X, Y]$ este o puncte și orice drum de la A la B utilizează această muchie. Mai mult decât atât, se poate demonstra ușor prin reducere la absurd că dacă $[X, Y]$ este punte, dacă un drum de la A la B folosește această muchie, atunci orice drum de la A la B va trece prin muchia $[X, Y]$; deci este suficient să găsim un drum de la A la B și să verificăm dacă muchia $[X, Y]$ aparține drumului.

Pentru interogările de tip 2 răspunsul va fi `no` dacă și numai dacă orașul C este un punct de articulație, A și B sunt în componente biconexe diferite și toate drumurile de la A la B trec prin orașul C.

Pe parcursul parcurgerii DFS, similar cu descompunerea unui graf în componente biconexe:

1. calculăm dfn și low;
2. identificăm punctele de articulație;
3. identificăm punțile;
4. reținem arborele parțial DFS, reprezentat prin liste de adiacență; observați că în lista de adiacență a unui vârf, nodurile sunt ordonate crescător după dfn (vor fi plasate în ordinea în care au fost vizitate);
5. determinăm pentru fiecare vârf x din graf valoarea $ultim[x]$ = pasul la care a fost „atins" pentru ultima dată vârful x în timpul parcurgerii DFS (moment la care toate vârfurile din subarborele cu rădăcina x în arborele parțial DFS au fost deja vizitate).

Pentru a verifica dacă un vârf a se află în subarborele cu rădăcina b în arborele parțial DFS vom utiliza funcția `este_descendent` care verifică dacă `dfn[b] <= dfn[a] && ultim[a] <= ultim[b]`.

Pentru interogările de tip `1 a b x y`:

1. dacă muchia $[x, y]$ nu este punte, răspunsul este `yes`;
2. dacă muchia $[x, y]$ este punte (considerăm că $dfn[x] < dfn[y]$) atunci dacă a și b sunt ambii în subarborele cu rădăcina y răspunsul este `yes`; de asemenea dacă nici a, nici b nu sunt în subarborele cu rădăcina y atunci răspunsul este `yes` (pentru că drumul de la a la b nu utilizează muchia $[x, y]$).

Pentru interogările de tip 2 a b c:

1. dacă c nu este punct de articulație, răspunsul este `yes`;
2. dacă c este punct de articulație, analizăm următoarele 3 cazuri:
   (a) dacă a și b nu sunt descendenți ai lui c (nu fac parte din subarborele cu rădăcina c) atunci răspunsul este `yes`;
   (b) dacă a și b sunt ambii descendenți ai lui c, apelăm funcția fiu care determină acel fiu al lui c (nod din lista sa de adiacență în arborele parțial DFS) care

conține în subarborele său vârful a (să-l notăm f_a), respectiv acel fiu al lui c care conține în subarborele său pe b (să-l notăm f_b); dacă $f_a = f_b$, atunci răspunsul este `yes`; de asemenea răspunsul este `yes` în cazul în care $low(f_a) < dfn(c)$ și $low(f_b) < dfn(c)$ (pentru că astfel există rute alternative de la a la b, care nu trec prin c);

(c) să considerăm că a nu este descendent al lui c și b este descendent al lui c (în caz contrar inversăm a cu b); în acest ultim caz apelăm funcția fiu pentru a determina f_b (fiul lui c care îl are pe b ca descendent); dacă $low(f_b) < dfn(c)$ atunci răspunsul este `yes` (deoarece există o rută alternativă, care nu trece prin c)

Implementare

```
#include <bits/stdc++.h>
#define NMAX 100008
#define MMAX 500008
using namespace std;

typedef pair<int,int> muchie;
map<muchie, int> punte; //reținem muchiile care sunt punte
vector<int> ADFS[NMAX]; //arborele parțial DFS

int n;       //numărul de vârfuri din graf
int num;     //numărul de ordine al vârfului curent în parcurgerea DFS
int nrfii;   //numărul de fii ai vârfului de start

vector<int> G[NMAX]; //reprezentarea grafului prin liste de adiacență
int dfn[NMAX], low[NMAX], ultim[NMAX];
bool Art[NMAX]; //vectorul caracteristic al mulțimii punctelor de articulație

void Citire();
void Biconex(int u, int tu);
int este_descendent(int a, int b);
int fiu (int x, int a);

int main()
{int i, tip, a, b, c, x, y, fa, fb, q;
  cin.sync_with_stdio(false);
  Citire();
  Biconex(1,-1);
  if (nrfii>1) Art[1]=1;
  cin>>q;
  for (i=0; i<q; i++)
    {cin>>tip>>a>>b;
      if (tip==1)
        {cin>>x>>y;
          if(dfn[x] > dfn[y]) swap(x,y);
          if (!punte.count(muchie(x,y))) //muchia [x,y] nu este punte
            cout<<"yes\n";
          else
            if (este_descendent(a,y) == este_descendent(b,y))
              cout<<"yes\n";
            else
              cout<<"no\n";
```

```
        }
      else
        {
          cin>>c;
          if(!Art[c]) //c nu este punct de articulație
            cout<<"yes\n";
          else
            {
              if (!este_descendent(a,c) && !este_descendent(b,c))
                cout<<"yes\n";
              else
                if (este_descendent(a,c) && este_descendent(b,c))
                  {fa = fiu(c,a); fb = fiu(c,b);
                    if (fa == fb)
                      cout<<"yes\n";
                    else
                      if (low[fa]<dfn[c] && low[fb] < dfn[c])
                        cout<<"yes\n";
                      else
                        cout<<"no\n";
                  }
                else
                  {if (este_descendent(a,c)) swap(a,b);
                    fb = fiu(c,b);
                    if (low[fb] < dfn[c]) cout<< "yes\n";
                    else
                      cout<<"no\n";
                  }
            }
        }
    }
  return 0;
}

void Citire()
{int x, y, m, i;
  cin>>n>>m;
  for (i=0; i<m; i++)
    {cin>>x>>y;
      G[x].push_back(y);
      G[y].push_back(x);
    }
  for (i=1; i<=n; i++) dfn[i]=low[i]=-1;
}

void Biconex(int u, int tu)
//u este nodul curent; tu este nodul părinte al lui u
{int x, p;
  dfn[u]=low[u]=++num;
  //parcurg lista de adiacență a nodului u
  for (p=0; p<G[u].size(); p++)
    {x=G[u][p];  //x este un nod adiacent cu u
      if (dfn[x]==-1) //x nu a mai fost vizitat
        {ADFS[u].push_back(x); //muchie în arborele parțial DFS
          if (u==1) //am găsit un fiu al vârfului start
            nrfii++;
          Biconex(x,u);
```

```
          low[u]=min(low[u],low[x]);
          if (low[x]>=dfn[u]) //u este punct de articulație
            { if (u!=1) Art[u]=1;
              if(low[x] > dfn[u]) punte[muchie(u,x)] = 1; }
        }
      else //x a mai fost vizitat
        if (x!=tu)
          //x nu este tatăl lui u,
          //deci [u,x] e muchie de întoarcere de la u la x
          low[u]=min(low[u],dfn[x]);
    }
  ultim[u]=++num;
}

int este_descendent(int a, int b)
//returnează 1 dacă și numai dacă a este în subarborele cu rădăcina b în
//arborele parțial DFS
{
  return dfn[b] <= dfn[a] && ultim[a] <= ultim[b];
}

int fiu (int x, int a)
//căutăm binar fiul lui x care conține în subarborele său pe a
{int st = -1, dr = ADFS[x].size(), mij;
  while (dr-st>1)
    {mij=(st+dr)/2;
      if (dfn[a] > ultim[ ADFS[x][mij]]) st = mij;
      else dr=mij;
    }
  return ADFS[x][dr];
}
```

10.5 Probleme propuse

- Problema Componente biconexe
- Problema Santa
- Problema Two papers II
- Problema Submerging Islands

10.6 Bibliografie

[1] Cătălina Cangea, *Componente biconexe*, URL: https://sites.google.com/site/centrulinfo1/materiale-video/algoritmi-video/componente-biconexe.

[2] Silviu Candale, *Biconexitate*, URL: https://www.pbinfo.ro/articole/5983/biconexitate.

[3] *Biconnected components*, URL: https://www.hackerearth.com/practice/algorithms/graphs/biconnected-components/tutorial/.

Capitolul 11

Algoritmul lui Dial

prof. Dan Pracsiu
Liceul Teoretic „Emil Racoviță” Vaslui
Centrul Județean de Excelență Vaslui

Algoritmul lui Dial determină drumul de cost minim de la un nod al unui graf ponderat la toate celelalte noduri accesibile. Însă, spre deosebire de alți algoritmi de drum minim, precum algoritmul lui Dijkstra sau algoritmul Bellman-Ford, algoritmul lui Dial se aplică asupra grafurilor în care ponderile pe muchii sunt suficient de mici.

11.1 Algoritmul lui Dijkstra

Considerăm un graf ponderat $G = (V, E)$, unde V este mulțimea nodurilor (numerotate de la 1 la n) și E este mulțimea muchiilor, graful fiind reprezentat prin liste de adiacență, iar costurile pe muchii fiind strict pozitive. Să ne reamintim algoritmul lui Dijkstra de complexitate $\mathcal{O}(m + n \log n)$ care determină distanțele minime de la un nod de start s la toate celelalte noduri accesibile. Se construiesc vectorii:

- d, de lungime n, în care $d[i]$ memorează distanța minimă de la nodul de start s la nodul i;
- viz, de lungime n, în care $viz[i] = 0$, dacă nu s-a găsit încă distanța minimă de la s la i, sau $viz[i] = 1$, dacă distanța minimă de la s la i a fost deja determinată.

Prin $cost(i, j)$ se înțelege costul muchiei $[i, j]$.

Algoritmul de mai sus are complexitatea $\mathcal{O}(n^2)$ dacă la fiecare pas determinarea nodului p cu $d[p]$ minim se face liniar, sau complexitatea $\mathcal{O}(m + n \log n)$ dacă se utilizează în loc de d un min-heap (coadă de priorități) pentru extragerea minimului.

Algoritmul 1 Algoritmul lui Dijkstra

```
1: d[s] ← 0
2: d[i] ← ∞, pentru orice i ≠ s
3: viz[i] ← 0, pentru orice i = 1...n
4: pentru i = 1, ..., n − 1 execută
5:     determină nodul p cu d[p] minim și viz[p] = 0
6:     dacă d[p] = ∞ atunci
7:         STOP
8:     altfel
9:         viz[p] ← 1;
10:        pentru fiecare vârf adiacent i al lui p execută
11:            dacă d[i] > d[p] + cost(p, i) atunci
12:                d[i] ← d[p] + cost(p, i)
```

11.2 Algoritmul lui Dial de determinare a distanțelor minime

Pornind de la algoritmul lui Dijkstra, în cazul în care distanțele de la nodul s la toate celelalte noduri sunt suficient de mici (valori între 0 și W), putem obține o soluție de complexitate liniară cu algoritmul lui Dial. Ideea este de a memora pentru fiecare distanță $x = 0 \dots W$ câte o listă de noduri. Mai precis, la fiecare pas al algoritmului, $L[x]$ reține lista nodurilor care se găsesc la distanță x de nodul de start s. Ca și la algoritmul lui Dijkstra, utilizăm vectorii:

1. d, de lungime n, în care $d[i]$ memorează distanța minimă de la nodul s la nodul i;
2. viz, de lungime n, în care $viz[i] = 0$, dacă nu s-a găsit încă distanța minimă de la s la i, sau $viz[i] = 1$, dacă distanța minimă de la s la i a fost deja determinată.

Se parcurg listele de la $L[0]$ la $L[W]$. Pentru fiecare listă $L[x]$, parcurgem nodurile listei și pentru fiecare astfel de nod i din $L[x]$:

1. Marcăm $viz[i] = 1$ (am găsit distanța minimă).
2. Pentru fiecare nod adiacent nevizitat j al lui i, dacă $d[j]$ este mai mare decât $d[i]$ la care se adaugă costul muchiei (i, j):
   (a) scoatem nodul j din lista $L[d[j]]$;
   (b) actualizăm valoarea lui $d[j]$;
   (c) depunem nodul j în lista $L[d[j]]$.

După parcurgerea tuturor listelor de la 0 la W, vectorul d de distanțe minime a fost construit.

La pasul (b) atunci când actualizăm valoarea $d[j]$ putem renunța la extragerea nodului j din vechea listă $L[d[j]]$ în cazul în care avem suficientă memorie. Nodul j va apărea astfel în mai multe liste, dar când îl întâlnim prima oară și vizităm pe j ($viz[j] = 1$), atunci celelalte apariții ale lui j în alte liste se ignoră.

11.3 Aplicație: Problema TollRoads

N orașe sunt conectate între ele prin M autostrăzi bidirecționale, fiecare autostradă (a, b) având un cost de tranzit c atașat. Se dorește revizuirea sistemului de taxare, însă sunt câteva aspecte ce trebuie luate în calcul și necesită investigație, deoarece o parte dintre cele N orașe sunt centre comerciale sau turistice importante. Se dorește să se afle răspunsul la o serie de Q întrebări de forma: (X, T) — câte dintre celelalte $N - 1$ orașe au acces către orașul X, cu o taxă totală de cel mult T către fiecare oraș? $(T \leq 100\,000)$

Soluție

Algoritmul lui Dial face în acest caz T pași, deoarece valoarea T nu poate fi depășită, deci se pornește cu $L[0]$ care memorează nodul de start k. Apoi se parcurg listele de la $L[1]$ la $L[T]$ pentru actualizări, exact precum s-a descris în algoritmul de mai sus.

```
#include <bits/stdc++.h>
using namespace std;
ifstream fin("tollroads.in");
ofstream fout("tollroads.out");
int n, d[100002];
vector< pair<int, int> > h[100002];
bitset<100002> viz;

int Test(int k, int T) {
  int i, x, cnt = -1;
  vector<int> L[100002];
  // L[i] reține nodurile aflate la distanța i de nodul de start k
  viz.reset();
  for (i = 1; i <= n; i++)
    d[i] = 1e9;
  d[k] = 0;
  L[0].push_back(k);

  for (i = 0; i <= T; i++)
    if (L[i].size() > 0) {
      for (int nod : L[i])
        if (viz[nod] == 0) {
          viz[nod] = 1;
          cnt++;
          for (auto e : h[nod])
            if (viz[e.first] == 0) {
              x = d[nod] + e.second;
              if (x <= T && d[e.first] > x) {
                d[e.first] = x;
                L[x].push_back(e.first);
              }
            }
        }
    }
  return cnt;
}

int main() {
  int i, j, k, c, T, m, Q;
```

```
  fin >> n >> m >> Q;
  for (k = 1; k <= m; k++) {
    fin >> i >> j >> c;
    h[i].push_back({j, c});
    h[j].push_back({i, c});
  }

  while (Q--) {
    fin >> k >> T;
    fout << Test(k, T) << "\n";
  }

  fout.close();
  return 0;
}
```

Partea a IV-a

Algoritmi pe șiruri de caractere

Capitolul 12

Căutări în șiruri de caractere. Algoritmul Rabin-Karp

Cătălin Frâncu
Nerdvana București

Căutarea unui subșir într-un șir înseamnă găsirea tuturor pozițiilor la care subșirul apare în șir. Să introducem câteva definiții și notații pentru restul articolului.

12.1 Terminologie

Definiția 12.1. Un **șir de caractere** este un vector cu elemente de tip caracter.

Notație. Vom nota cu Σ alfabetul din care fac parte elementele șirului.

În multe probleme de informatică, $\Sigma = \{a, b, \ldots, z\}$, dar ocazional alfabetul poate consta din 0 și 1, din litere mari etc.

Notație. Vom nota cu $|S|$ lungimea șirului S.

Vom opera pe șiruri C (`char*`), astfel că un șir de n caractere își stochează informația pe pozițiile $0 \ldots n-1$ și conține terminatorul `'\0'` pe poziția n. Codul poate fi ușor adaptat și pentru clasa C++ `string`.

Definiția 12.2. Spunem că un subșir P **apare la poziția** k într-un șir S dacă $|P|+k \leq |S|$ și $P[i] = S[i+k]$ pentru $0 \leq i < |P|$. Valoarea k se numește **deplasare validă**. Pozițiile la care P nu apare în S se numesc **deplasări invalide**.

Cu aceste definiții și notații, putem reformula căutarea subșirului P în șirul S ca găsirea tuturor deplasărilor valide ale lui P în S. Subșirul de căutat se mai numește și **șablon** (engl. *pattern*). Vom nota $|P| = m$ și $|S| = n$.

12.2 Algoritmul naiv

Următorul algoritm evaluează naiv fiecare deplasare și le tipărește pe cele valide. Pentru o deplasare `dep` fixată, algoritmul caută liniar prima nepotrivire.

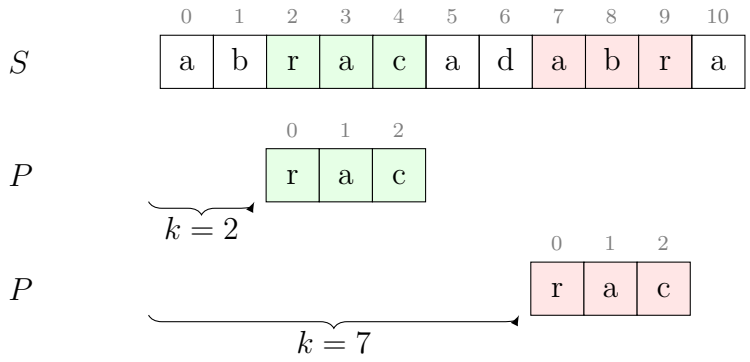

Figura 12.1: Pentru șablonul $P =$ `rac` și șirul $S =$ `abracadabra`, $k = 2$ este o deplasare validă, iar $k = 7$ este o deplasare invalidă.

```
int n = strlen(s);
int m = strlen(p);

for (int dep = 0; dep <= n - m; dep++) {
  int i = 0;
  while ((i < m) && (p[i] == s[i + dep])) {
    i++;
  }
  if (i == m) {
    printf("Deplasare validă: %d\n", dep);
  }
}
```

Algoritmul este ineficient, având complexitate $\mathcal{O}(m(n-m))$. Putem expune această ineficiență cu valori ca $S =$ `aaaaaaaaaa` și $P =$ `aaaaab`. Pentru completitudine, menționăm totuși că există situații în care algoritmul naiv se comportă optim. Iată trei dintre ele.

12.2.1 Toate caracterele din șablon sunt diferite

Să presupunem că pentru o deplasare k găsim o primă nepotrivire între $P[i]$ și $S[i+k]$. În particular, aceasta înseamnă că $P[1 \ldots i-1] = S[1+k \ldots i-1+k]$. Așadar, în toată acea regiune din S apar caractere din P, dar **nu** $P[0]$. Evaluarea acelor deplasări va eșua încă de la prima comparație. Următoarea deplasare care are o șansă să fie validă va fi $k+i$.

Din acest motiv, complexitatea este $\mathcal{O}(n)$.

12.2.2 Unul dintre șirurile S și P este generat aleatoriu

În această situație, fiecare comparație între un caracter din S și unul din P va eșua cu probabilitatea

$$\frac{|\Sigma| - 1}{|\Sigma|}$$

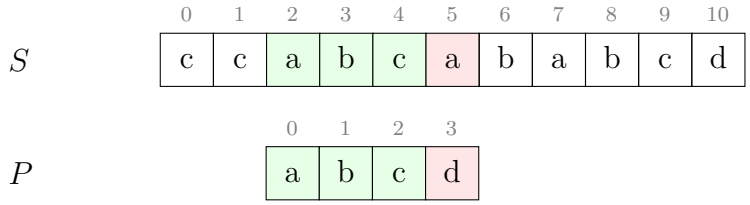

Figura 12.2: Pentru deplasarea $k = 2$, prima nepotrivire apare la poziția $i = 3$. Caracterul `a` nu apare pe pozițiile 3 și 4. Următoarea deplasare care merită verificată este $k = 5$.

, de unde cu puțină teorie a probabilităților aflăm că numărul așteptat de comparații pentru o deplasare dată va fi

$$\frac{|\Sigma|}{|\Sigma| - 1}$$

Această valoare este 2 pentru alfabete binare și tinde la 1 pe măsură ce mărimea alfabetului crește.

12.2.3 m are valoare apropiată de n

Reamintim că complexitatea nu este $\mathcal{O}(mn)$, ci $\mathcal{O}(m(n-m))$.

12.3 Funcții hash

Dacă ați descărcat vreodată un fișier mare (de exemplu, o imagine de stick bootabil), poate ați observat că alături de el găsim o **sumă de control** (engl. *checksum*), de regulă sub forma unui număr hexazecimal lung calculat cu un algoritm ca SHA256. Care este rostul acestui *checksum*? În esență, este un mod de a ne asigura că am descărcat fișierul fără erori. Rulăm local același algoritm și ne asigurăm că obținem același rezultat ca pe site.

Un algoritm de *checksum* este un tip particular de **funcție hash** (diferențele depășesc scopul prezentului articol). O funcție hash transformă niște date de o lungime arbitrară în alte date (numite **valori hash**) de lungime fixă. Indiferent cât de mare este fișierul descărcat sau lungimea șirului citit, funcția hash returnează valori pe un număr fix de biți, de exemplu 256.

De la o funcție hash bună dorim următoarele proprietăți:

1. Să fie ușor de calculat.
2. Să traducă valori de intrare aproape identice în valori hash substanțial diferite.
3. Să genereze toate valorile hash posibile cu aceeași probabilitate pentru valori de intrare aleatorii; cu alte cuvinte, să distribuie bine valorile de intrare.

Este important să observăm că funcțiile hash **nu sunt injective**. Nici nu ar avea cum, căci spațiul valorilor de intrare este infinit, pe când spațiul valorilor hash este finit. Ce

implică aceasta? Dacă două valori de intrare X și Y corespund la valori hash diferite, atunci știm sigur că $X \neq Y$. În schimb, dacă valorile hash sunt egale, nu avem nicio garanție că $X = Y$. De exemplu, pentru funcția hash $H(x) = x \bmod 10$, valorile de intrare 27 și 57 corespund aceleiași valori hash, 7. Această situație se numește **coliziune**.

12.4 Căutarea cu funcții hash

Din secțiunea anterioară decurge o metodă de căutare cu funcții hash:

```
int n = strlen(s);
int m = strlen(p);
int hp = hash(p, m);

for (int dep = 0; dep <= n - m; dep++) {
  if ((hash(s + dep, m) == hp) &&
      compar_naiv(p, s + dep, m)) {
    printf("Deplasare validă: %d\n", dep);
  }
}
```

Pentru concizie, am exclus codul funcțiilor `hash` și `compar_naiv`. Observăm că acest algoritm are tot complexitate $\mathcal{O}(m(n-m))$, din două motive:

1. Funcția hash, dacă îi permitem să examineze complet subșirul curent, are complexitate $\mathcal{O}(m)$. Vom arăta în secțiunea următoare cum putem reduce complexitatea la $\mathcal{O}(1)$.
2. Pentru date de intrare ca $S =$ `aaaaaaaaaa` și $P =$ `aaaaaa`, toate valorile hash vor fi egale și algoritmul va degenera în $n - m$ comparări naive.

12.5 Funcții *rolling hash*

Să considerăm alfabetul $\Sigma = \{0, 1, \ldots, 9\}$. Să considerăm, pentru simplitate, că $m \leq 9$. Atunci putem defini funcția hash a unui subșir de lungime m ca fiind valoarea numerică a acelui șir. De exemplu, $H($`"2943"`$) = 2\,943$.

Acum, fie $P =$ `9435` și $S =$ `1112943511`. Atunci vom calcula $H(P) = 9\,435$ și $H(S[3...6]) = 2\,943$. Când trecem de la deplasarea 3 la deplasarea 4, putem calcula în doar $\mathcal{O}(1)$ noua valoare hash, $H(S[4...7])$. Procedăm astfel:

- Din 2 943 scădem prima cifră (adică 2) înmulțită cu 10^{m-1} (adică 1 000). Diferența este 943. În termeni de șiruri, am eliminat primul caracter din stânga.
- Pe 943 îl înmulțim cu 10. Obținem 9 430.
- Adăugăm următoarea cifră din S (5). Obținem 9 435. În termeni de șiruri, am adăugat la dreapta următorul caracter.

Avantajul este că am redus la $\mathcal{O}(n)$ efortul total pentru a calcula valorile hash ale tuturor subșirurilor lui S de lungime m. Acest tip de funcție se numește *rolling hash* pentru că *se rostogolește* de la o poziție la următoarea.

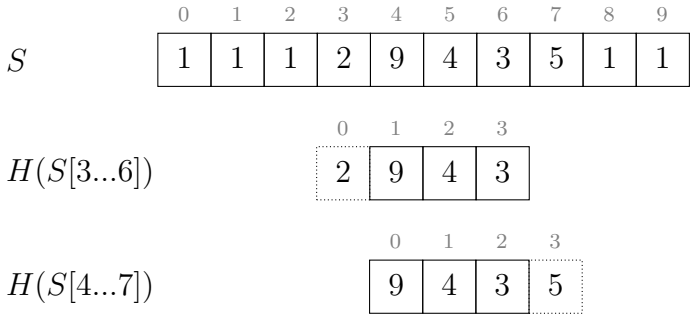

Figura 12.3: Pentru deplasarea $k = 3$, valoarea hash este 2 943.
Pentru $k = 4$, valoarea hash este $(2\,943 - 2\,000) \times 10 + 5 = 9\,435$.

În cazul general, valorile hash vor avea mărimi de ordinul $|\Sigma|^m$ și vor fi prea mari pentru a fi reprezentate în tipuri de date simple. Soluția este să operăm modulo o valoare q. Astfel, formula generală pentru a trece de la o valoare hash la următoarea este:

$$H(S[k+1...k+m]) = \left\{ \left[H(S[k...k+m-1]) - S[k] \cdot |\Sigma|^{m-1} \right] \cdot |\Sigma| + S[k+m] \right\} \bmod q$$

În concluzie, funcția traduce $|\Sigma|^m$ șiruri posibile în doar q valori hash. Așadar, pot apărea coliziuni. De aceea, ori de câte ori găsim în S valoarea $H(P)$ este necesar să facem și o comparare naivă. Complexitatea teoretică a algoritmului rămâne $\mathcal{O}(m(n-m))$.

12.6 Algoritmul Rabin-Karp

Punând cap la cap toate elementele de mai sus, rezultă codul:

```
#include <stdio.h>
#include <string.h>

const int MAX_N = 1'000'000;
const int SIGMA = 26;
const int MOD = 1'000'000'007;

char s[MAX_N + 1], p[MAX_N + 1];
long long q; // sigma^{m-1}

void precompute_q(int exp) {
  q = 1;
  while (exp--) {
    q = q * SIGMA % MOD;
  }
}

long long init_hash(char* s, int len) {
  long long result = 0;
  for (int i = 0; i < len; i++) {
    result = (result * SIGMA + (s[i] - 'a')) % MOD;
```

```
  }
  return result;
}

long long avans_hash(long long old, char* s, int len) {
  old += MOD - (q * (s[0] - 'a')) % MOD;
  return (old * SIGMA + (s[len] - 'a')) % MOD;
}

bool compar_naiv(char* a, char* b, int len) {
  int i = 0;
  while ((i < len) && (a[i] == b[i])) {
    i++;
  }

  return (i == len);
}

int main() {
  scanf("%s %s", s, p);
  int n = strlen(s);
  int m = strlen(p);
  precompute_q(m - 1);

  long long hp = init_hash(p, m);
  long long hs = init_hash(s, m);

  for (int dep = 0; dep <= n - m; dep++) {
    if ((hs == hp) &&
        compar_naiv(p, s + dep, m)) {
      printf("Deplasare validă: %d\n", dep);
    }
    hs = avans_hash(hs, s + dep, m);
  }

  return 0;
}
```

Notăm următoarele considerente practice.

- Valoarea $|\Sigma|^{m-1}$ trebuie precalculată.
- Pentru a evita depășirile, este necesar ca $|\Sigma| \cdot q$ să încapă în tipul de date ales (de exemplu `int` sau `long long`).

Ca încercare de optimizare, putem folosi un modul putere a lui doi pentru viteză mai mare. Putem chiar să operăm modulo 2^{32} sau modulo 2^{64} și să renunțăm complet la operațiile de împărțire (depășirile vor face automat acest lucru pentru noi). Totuși, este posibil ca datele de test să fie alese adversarial și să genereze multe coliziuni pentru aceste module arhicunoscute. Din acest punct de vedere este preferabil un modul mai obscur.

Putem renunța la comparările naive dacă avem speranțe reale că nu vor exista coliziuni. Vom considera orice apariție a lui $H(P)$ ca fiind o deplasare validă. Atunci algoritmul devine $\mathcal{O}(m+n)$, dar subliniem că această abordare este teoretic incorectă.

Pentru a reduce numărul de coliziuni, putem calcula două funcții hash modulo două valori

diferite, $H_1(P)$ și $H_2(P)$. Șansa să întâlnim o coliziune modulo ambele valori simultan scade considerabil.

12.7 Bibliografie

[1] Thomas H. Cormen et al., *Introduction to Algorithms*, The MIT Press, 2022, cap. 32.2.

[2] *String Hashing*, URL: https://cp-algorithms.com/string/string-hashing.html.

[3] *Zobrist hashing*, URL: https://en.wikipedia.org/wiki/Zobrist_hashing, Un alt exemplu de funcție hash recalculabilă incremental.

Capitolul 13

Căutări cu automate finite. Algoritmul Knuth-Morris-Pratt

Cătălin Frâncu
Nerdvana București

Algoritmul KMP este denumit astfel după inițialele inventatorilor săi, Donald Knuth, James Morris și Vaughan Pratt. Este un algoritm remarcabil de scurt și de elegant, dar, fără înțelegerea bazelor pe care este construit, el poate părea mistic, revelația unui profet pe un munte. Pentru a destrăma acest mister, vom face un periplu prin lumea automatelor finite.

13.1 Terminologie

Introducem încă câteva notații suplimentare față de Secțiunea 12.1.

Notație. Vom nota cu S_k prefixul de lungime k al șirului S.

Notație. Vom nota cu $A \sqsubseteq B$ faptul că A este prefix al lui B, așadar că $|A| \leq |B|$ și că $A[i] = B[i]$ pentru $0 \leq i < |A|$. Dacă în plus $|A| < |B|$, vom folosi notația strictă $A \sqsubset B$.

Notație. Vom nota cu $A \sqsupseteq B$ faptul că A este sufix al lui B, așadar că $|A| \leq |B|$ și că $A[i] = B[|B| - |A| + i]$ pentru $0 \leq i < |A|$. Dacă în plus $|A| < |B|$, vom folosi notația strictă $A \sqsupset B$.

13.2 Automate finite

Colocvial, un **automat finit determinist** (AFD) este un sistem teoretic care procesează caracter cu caracter un șir dat la intrare, de lungime arbitrară, dar finită. Automatul acceptă unele șiruri și le respinge pe restul. Mulțimea șirurilor pe care automatul la acceptă se numește **limbajul recunoscut** de automat.

Să studiem următorul exemplu.

Distingem următoarele elemente:

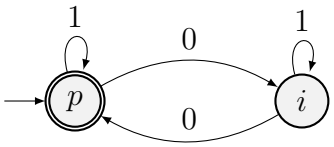

Figura 13.1: Un exemplu de automat finit determinist.

- Automatul are două **stări** numite p și i. După fiecare caracter citit, automatul se va afla într-una din aceste stări.
- Starea p are o săgeată venind din exterior. Ea se numește **stare inițială**. În această stare se află automatul înainte de citirea primului caracter.
- Tot starea p are contur dublu. Aceasta o desemnează drept **stare de acceptare**. Pot exista oricâte stări de acceptare, inclusiv zero.
- Din fiecare stare pornește exact o săgeată pentru fiecare simbol din alfabet, în acest caz $\{0, 1\}$. Săgețile arată în ce stare trece automatul la citirea respectivului simbol. Aceste stări pot fi rezumate în **funcția** (sau **matricea**) **de tranziție**:

simbolul / starea	`0`	`1`
p	i	p
i	p	i

Tabela 13.1: Matricea de tranziție pentru automatul din Figura 13.1.

Pentru șirul de intrare `11010111`, automatul va porni din starea p și va trece succesiv prin stările p, p, i, i, p, p, p, p. Întrucât starea finală este una de acceptare, automatul va accepta șirul `11010111`.

Întrebarea este: Ce face acest automat în cazul general? Observația esențială este că el rămâne în aceeași stare pe caractere `1` și alternează între cele două stări pe caractere `0`. Rezultă că automatul se va afla în starea de acceptare (p) ori de câte ori va fi văzut un număr par de caractere `0`. Formal, spunem că automatul recunoaște limbajul

$$L = \{s \mid s \in \{0,1\}^* \text{ și } s \text{ conține un număr par de zerouri}\}$$

De regulă avem de rezolvat problema inversă: dându-se un limbaj, dorim să construim un AFD care să-l recunoască. În acest caz, strategia este să înțelegem ce anume este reprezentativ pentru porțiunea din șir văzută până la caracterul curent și să definim stările automatului ca să captureze acea informație. Pentru automatul din Figura 13.1, informația reprezentativă este paritatea numărului de zerouri. De aceea am și denumit stările p (automatul a citit un număr par de zerouri, inițial 0) și i (automatul a citit un număr impar de zerouri).

13.2.1 Alte exemple de automate

Ca să ne familiarizăm cu procesul de gândire prin care concepem un AFD pentru un limbaj dat, vom da câteva exemple. Încercați să descoperiți voi înșivă ce limbaje recunosc aceste automate, înainte de a citi descrierile!

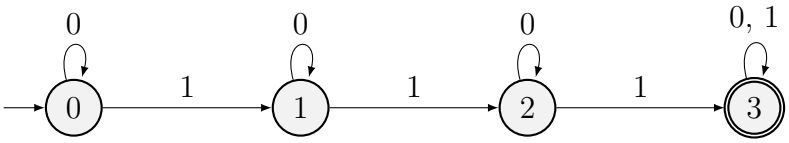

Figura 13.2

Automatul din Figura 13.2 ține evidența numărului de caractere `1` întîlnite: 0, 1, 2 sau 3 și peste. Într-adevăr, stările 0, 1, 2 și 3 au tocmai acest scop. Automatul acceptă șirurile care conțin cel puțin 3 caractere `1`.

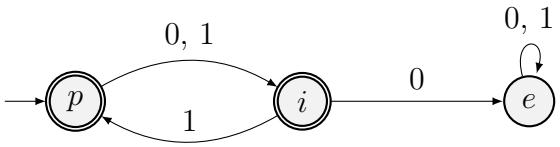

Figura 13.3

Automatul din Figura 13.3 acceptă șirurile care conțin `1` pe toate pozițiile pare. Pentru acest limbaj trebuie să ținem evidența a două informații: paritatea numărului de caractere citite (indicată de stările p și i) și dacă am văzut, pe orice poziție pară anterioară, un caracter `0`. În situația din urmă, automatul trece într-o stare „de eroare", e, în care el rămâne tot restul șirului și din care nu mai revine niciodată într-o stare de acceptare. Astfel, automatul va respinge toate șirurile în care vede un `0` pe o poziție pară și le va accepta pe restul.

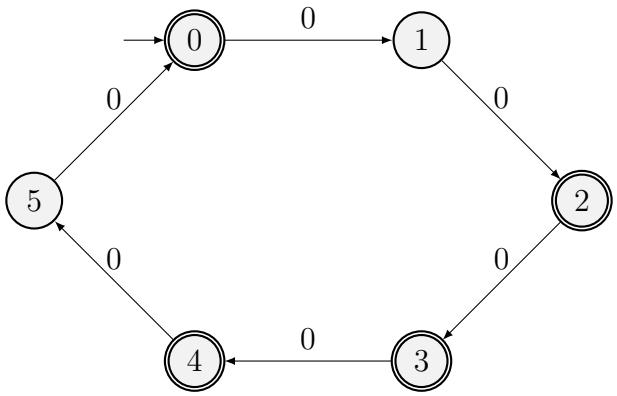

Figura 13.4

Automatul din Figura 13.4 operează pe un alfabet unar: șirurile conțin doar caracterul `0`. Observăm că el acceptă doar șiruri care constau din n simboluri, unde $n = 6k$ sau $n = 6k + 2$ sau $n = 6k + 3$ sau $n = 6k + 4$. Cu alte cuvinte, șiruri a căror lungime este multiplu de 2 sau multiplu de 3.

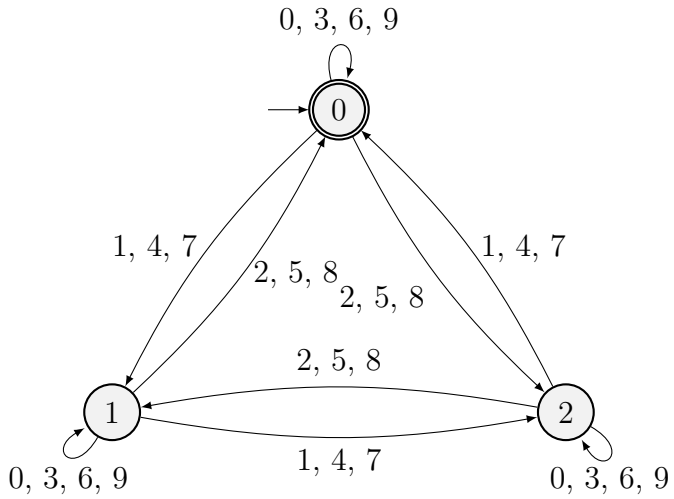

Figura 13.5

Automatul din Figura 13.5 operează pe alfabetul cifrelor zecimale. Dacă studiem cum sunt grupate cifrele, vom observa că automatul își schimbă starea în funcție de restul modulo trei al ultimei cifre citite. De exemplu, o cifră 1, 4 sau 7 va cauza trecerea în următoarea stare (ciclic). Astfel observăm că starea k arată că suma cifrelor citite până în prezent este congruentă cu k modulo 3. În fapt, automatul acceptă numere naturale divizibile cu 3.

13.3 Automate pentru recunoașterea de subșiruri

Să revenim la problema căutării unui subșir într-un șir. De exemplu, pe alfabetul binar `{a, b}`, să scriem un automat care ne ajută să găsim, în șirul dat la intrare, toate aparițiile șablonului `aab`. Metoda este să scriem un automat care recunoaște toate șirurile **terminate** în `aab`. Atunci automatul se va afla într-o stare de acceptare imediat după fiecare apariție a șablonului `aab`.

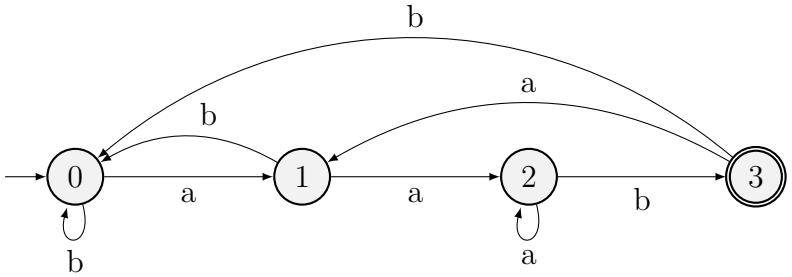

Figura 13.6: Un automat care recunoaște aparițiile subșirului `aab`.

Observăm că spre dreapta avem exact trei săgeți corespunzătoare caracterelor șablonului: `a`, `a`, `b`. Restul săgeților rămân în aceeași stare sau merg spre stânga. Intuitiv, cele patru stări semnifică:

- 0: Ultimele caractere văzute nu sunt interesante.
- 1: Ultimul caracter văzut este `a`.

- 2: Ultimele două caractere văzute sînt `aa`.
- 3: Ultimele trei caractere văzute sînt `aab`.

În cazul general, starea k înseamnă că ultimele k simboluri citite de la intrare coincid cu prefixul de lungime k al șablonului. De aceea, este relativ clar că avem nevoie de trei săgeți care consumă caracterele `a`, `a`, `b` și merg exact cu o stare spre dreapta fiecare. Mai puțin evident este unde trebuie să ducă săgețile care merg spre stânga.

Un exemplu relevat este săgeata care ciclează în starea 2 pentru simbolul `a`. Într-adevăr, dacă ultimele două caractere văzute sunt `aa` și mai vedem încă unul, rezultă că ultimele trei caractere văzute sunt `aaa`. Dintre acestea, ultimele două încă se potrivesc cu definiția stării 2, deci automatul rămâne în acea stare. De exemplu, pentru șirul de intrare `aaaab`, automatul va trece succesiv prin stările $0, 1, 2, 2, 2, 3$ și va accepta șirul. Pentru același motiv, automatul trece din starea 3 în starea 1 pe simbolul `a`: putem folosi acel simbol ca eventual început al unei noi apariții a lui `aab`.

Să încheiem această secțiune cu un exemplu mai complex: un automat care recunoaște subșirul `abacaba` în alfabetul literelor mici.

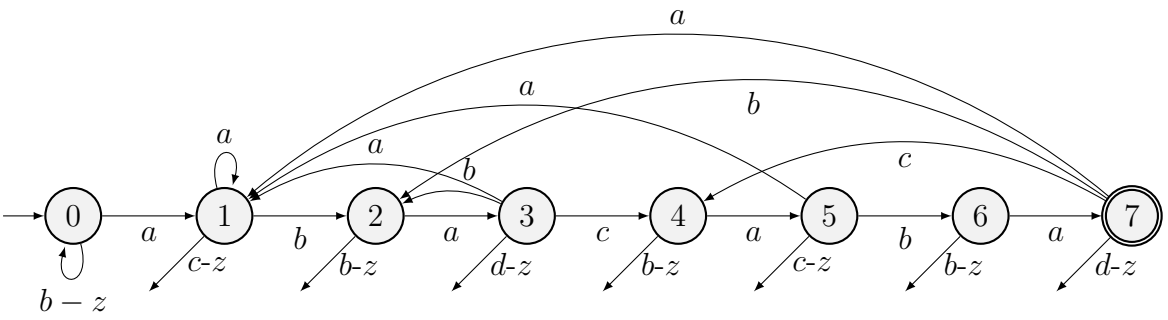

Figura 13.7: Un automat care recunoaște aparițiile subșirului `abacaba`. Săgețile de sub fiecare nod duc înapoi în starea 0.

Și în Figura 13.7 recunoaștem săgețile spre dreapta care consumă câte un caracter din șablon. Detaliem și traiectoriile a două săgeți spre stânga:

- În starea 3, ultimele caractere văzute sunt `aba`. Săgeata din starea 3 în starea 2 pe simbolul `b` înseamnă că ultimele patru caractere văzute sînt `abab`, dintre care putem refolosi sufixul `ab`, cum ar fi de exemplu pentru șirul de intrare `ababacaba`.
- În starea 7, ultimele caractere văzute sunt `abacaba`. Săgeata din starea 7 în starea 4 pe simbolul `c` înseamnă că ultimele opt caractere văzute sînt `abacabac`, dintre care putem refolosi sufixul `abac`, cum ar fi de exemplu pentru șirul de intrare `abacabacaba`.

13.4 Algoritmul de căutare cu automate finite

Acum, să generalizăm exemplele din secțiunea anterioară. Dorim să construim un automat care să recunoască un șablon arbitrar P primit la intrare. Putem deja spune destul de multe lucruri despre acest automat:

- El va avea $m + 1$ stări numerotate de la 0 la m (reamintim că $m \overset{\text{not.}}{=} |P|$).

- Starea inițială este 0.
- Unica stare de acceptare este m.
- Rămâne să definim matricea de tranziție, notată cu $\delta(q, c)$, pentru fiecare stare q și fiecare caracter c.

Dacă reușim să construim matricea de tranziție, algoritmul va rula în mod banal în $\mathcal{O}(n)$: tot ce are de făcut este să consume un caracter și să treacă în noua stare! Ori de câte ori ajunge în starea m, algoritmul raportează o apariție a lui P.

Așadar, cum definim $\delta(q, c)$? Din Figurile 13.6 și 13.7 deducem un principiu general, a cărui demonstrație formală o omitem. Dacă automatul se află în starea q, înseamnă că ultimele q caractere citite coincid cu prefixul P_q. Dacă presupunem că următorul caracter întâlnit este c, atunci ultimele $q + 1$ caractere citite sunt $P_q c$. Dintre acestea, încercăm să refolosim cît mai multe dintre ultimele, ca să rămânem într-o stare k cît mai mare.

Formal, când căutăm un șablon P într-un șir S, matricea de tranziție este:

$$\delta(q, c) = \max\{k \mid P_k \sqsupseteq P_q c\}$$

De aici decurge programul de mai jos, care construiește matricea în mod naiv, în $\mathcal{O}(|\Sigma| \cdot m^3)$. Complexitatea totală este $\mathcal{O}(|\Sigma| \cdot m^3 + n)$.

```
#include <stdio.h>
#include <string.h>

const int MAX_N = 1'000'000;
const int SIGMA = 26;

char s[MAX_N + 1], p[MAX_N + 1];
int d[MAX_N + 1][SIGMA];
int n, m;

int min(int x, int y) {
  return (x < y) ? x : y;
}

bool compar_naiv(char* a, char* b, int len) {
  int i = 0;
  while ((i < len) && (a[i] == b[i])) {
    i++;
  }

  return (i == len);
}

// Returnează true dacă și numai dacă P_k este sufix al lui P_{q}c.
bool este_sufix(int k, int q, char c) {
  return
    (k == 0) ||
    ((p[k - 1] == c) && compar_naiv(p, p + (q - k + 1), k - 1));
}

void constr_delta() {
  for (int q = 0; q <= m; q++) {
```

```
    for (int c = 0; c < SIGMA; c++) {
      int k = min(q + 1, m); // Nu putem depăși starea m.
      while (!este_sufix(k, q, c + 'a')) {
        k--;
      }
      d[q][c] = k;
    }
  }
}

int main() {
  scanf("%s %s", s, p);
  n = strlen(s);
  m = strlen(p);

  constr_delta();

  int q = 0;
  for (int i = 0; i < n; i++) {
    q = d[q][s[i] - 'a'];
    if (q == m) {
      printf("Deplasare validă: %d\n", i - m + 1);
    }
  }

  return 0;
}
```

13.5 Algoritmul Knuth-Morris-Pratt

Înarmați cu aceste cunoștințe, putem analiza algoritmul KMP. El îmbunătățește major două aspecte ale algoritmului de căutare cu automate:

1. Reduce memoria necesară de la $\mathcal{O}(|\Sigma| \cdot m)$ la $\mathcal{O}(m)$.
2. Reduce timpul de calcul al informațiilor necesare de la $\mathcal{O}(|\Sigma| \cdot m^3)$ la $\mathcal{O}(m)$.

Automatul calculează o **funcție de prefix** π, un vector cu m elemente, dependent doar de P, dar nu de S nici de mărimea alfabetului. Funcția de tranziție δ răspundea la întrebarea (în limbaj natural): „Sunt în starea q și văd la intrare caracterul c. În ce stare să trec?” Funcția π răspunde la o întrebare similară: „Sunt în starea q. Presupunând că următorul caracter de la intrare nu se potrivește, în ce stare să trec?”

Pentru a înțelege de ce această informație este semnificativă, să studiem comportamentul algoritmului cu automate finite pentru datele din figura 13.8.

Automatul va găsi o deplasare validă și va trece în starea $|P| = 7$ după ce consumă caracterele $S[0 \ldots 9]$. Apoi va consuma următorul caracter (`b`) și va trece din starea 7 în starea $\delta(7, \texttt{b}) = 2$, deoarece poate reutiliza ultimele două caractere, `ab`. În fapt, automatul precalculează cel mai lung sufix al lui P_7 care este și sufix al lui P_7 și care poate fi extins cu caracterul `b`.

Algoritmul KMP are o abordare ușor diferită: el evaluează **toate** prefixele lui P_7 care

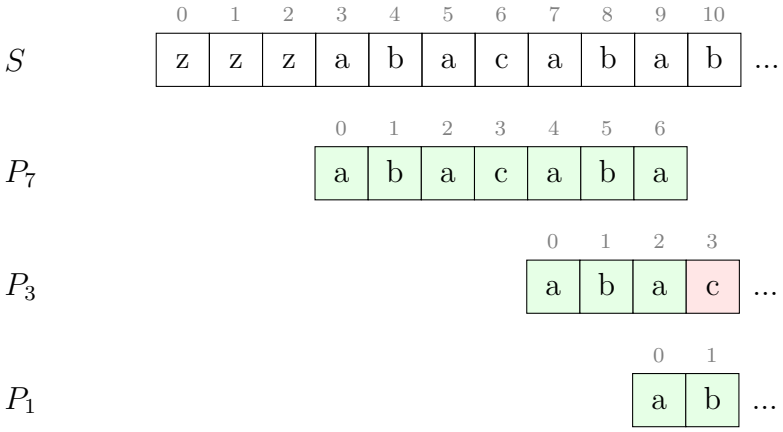

Figura 13.8: La poziția 10, nu putem extinde prefixul P_3 cu caracterul `b`, ci numai prefixul P_1.

sunt și sufixe ale lui P_7, respectiv $P_3 =$ `aba` și $P_1 =$ `a`. Algoritmul încearcă să treacă în starea 3, dar constată că P_3 nu poate fi extins cu caracterul `b` (căci `b` $\neq P[3] =$ `c`). Apoi, algoritmul încearcă să treacă în starea 1 și constată că P_1 poate fi extins. Așadar, algoritmul trece în starea 1, apoi consumă caracterul de la intrare (`b`) și urcă în starea 2.

Rezultă că funcția de prefix π aferentă unui șablon P trebuie să calculeze, pentru fiecare stare q, toate prefixele lui P_q care sînt și sufixe al lui P_q. În fapt, lucrurile sînt mai simple. Este suficient ca $\pi[q]$ să stocheze doar cel mai lung dintre aceste prefixe, din care apoi le putem deduce pe următoarele. De exemplu, prefixele lui $P_7 =$ `abacaba` care sunt și sufixe sunt $P_3 =$ `aba` și $P_1 =$ `a`, dar observăm că, la rândul său, P_1 este prefix și sufix al lui P_3. De aceea, lista completă a prefixelor lui P_q care sunt și sufixe va fi $\pi[q], \pi[\pi[q]], \pi[\pi[\pi[q]]], \ldots$

Formal, definiția lui π este:

$$\pi(q) = \max\{k \mid k < q \text{ și } P_k \sqsupset P_q\}$$

Cum construim vectorul π? În mod foarte elegant, P trebuie să răspundă la aceeași întrebare despre el însuși ca și despre vectorul S: dacă mă aflu în starea q, care este cel mai lung prefix al lui P care este și sufix al lui P_q?

Redăm implementarea algoritmului. Precizăm că indicii în `pi[]` sunt decalați cu 1, deoarece vectorul este indexat de la zero. De exemplu, informația „pentru șirul `abacaba`, cel mai lung prefix care este și sufix este `aba`” se notează la poziția 6, așadar `pi[6] = 3`.

```
#include <stdio.h>
#include <string.h>

const int MAX_N = 1'000'000;

char s[MAX_N + 1], p[MAX_N + 1];
int pi[MAX_N + 1];
int n, m;

void constr_functie_prefix() {
```

```
  pi[0] = 0;
  int k = 0;
  for (int q = 1; q < m; q++) {
    while ((k > 0) && (p[k] != p[q])) {
      k = pi[k - 1];
    }
    if (p[k] == p[q]) {
      k++;
    }
    pi[q] = k;
  }
}

void kmp() {
  int q = 0; // starea curentă
  for (int i = 0; i < n; i++) {
    while ((q > 0) && (p[q] != s[i])) {
      q = pi[q - 1];
    }
    if (p[q] == s[i]) {
      q++;
    }
    if (q == m) {
      printf("Deplasare validă: %d\n", i - m + 1);
    }
  }
}

int main() {
  scanf("%s %s", s, p);
  n = strlen(s);
  m = strlen(p);

  constr_functie_prefix();
  kmp();

  return 0;
}
```

13.5.1 Analiza complexității

Aparent, construcția lui π poate dura $\mathcal{O}(m^2)$, întrucât fiecare buclă `while` poate face m iterații. Totuși, algoritmul KMP, ca și automatul finit, avansează spre dreapta cel mult cu o stare la fiecare caracter citit. De aceea, numărul de incrementări ale lui k în funcția `constr_functie_prefix()` este de cel mult $m - 1$, iar numărul de scăderi nu poate fi mai mare. Funcția are complexitatea amortizată $\mathcal{O}(m)$. Prin același raționament, funcția `kmp()` are complexitatea amortizată $\mathcal{O}(n)$, iar complexitatea totală a algoritmului KMP este $\mathcal{O}(m + n)$.

13.6 Bibliografie

[1] Michael Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 2012, Pentru mai multe detalii despre automate finite, limbaje regulate și limitele lor.

[2] *Prefix function. Knuth-Morris-Pratt algorithm*, URL: https://cp-algorithms.com/string/prefix-function.html.

[3] *Z-function and its calculation*, URL: https://cp-algorithms.com/string/z-function.html, O funcție similară funcției de prefix, care oferă diverse statistici pe șiruri.

Partea a V-a

Geometrie

Capitolul 14

Elemente de geometrie computațională

prof. Marius Nicoli
Colegiul Național „Frații Buzești” Craiova
Centrul de Pregătire pentru Performanță în Informatică Craiova

14.1 Introducere

Materialul următor își propune prezentarea modului de rezolvare pentru câteva probleme de geometrie computațională reducând cât mai mult posibil utilizarea funcțiilor de bibliotecă (mai ales a celor ce implică lucru cu numere reale).

Prezentăm, mai întâi, câteva rezultate utile:

14.1.1 Determinarea distanței dintre două puncte din plan

Fie două puncte de coordonate (x_1, y_1) și (x_2, y_2). Distanța în plan dintre ele se calculează folosind teorema lui Pitagora. Se formează un triunghi dreptunghic, în care catetele au lungimi $|x_1 - x_2|$ respectiv $|y_1 - y_2|$. Așadar, distanța este lungimea ipotenuzei acestui triunghi, adică:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Formula este valabilă și pentru cazurile particulare de puncte cu același x sau cu același y.

14.1.2 Verificarea coliniarității a trei puncte. Aria unui triunghi determinat de trei puncte în plan

Fie punctele de coordonate: (x_1, y_1), (x_2, y_2) și (x_3, y_3). Dacă acestea sunt coliniare ne putem imagina Figura 14.1:

Cele două triunghiuri sînt asemenea, deci avem relația

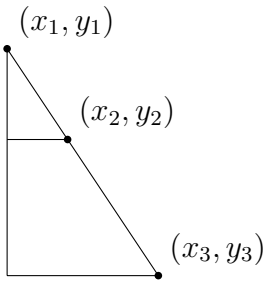

Figura 14.1: Trei puncte coliniare determină triunghiuri asemenea.

$$\frac{x_2 - x_1}{x_3 - x_1} = \frac{y_2 - y_1}{y_3 - y_1}$$

care poate fi scrisă

$$(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1) = 0$$

Aceasta este relația pe care trebuie să o verifice punctele pentru ca ele să fie coliniare.

Un rezultat interesant este următorul: dacă notăm cu D expresia din partea stângă a semnului „=", atunci $\frac{|D|}{2}$ reprezintă aria triunghiului format cu punctele date (am notat cu $|D|$ valoarea absolută a numărului D). Iată o modalitate simplă de justificare. Înscriem triunghiul format de cele 3 puncte într-un dreptunghi, cu laturile paralele cu axele, ca în figură (consider laturi ale dreptunghiului cele cu x minim, x maxim, y minim, y maxim).

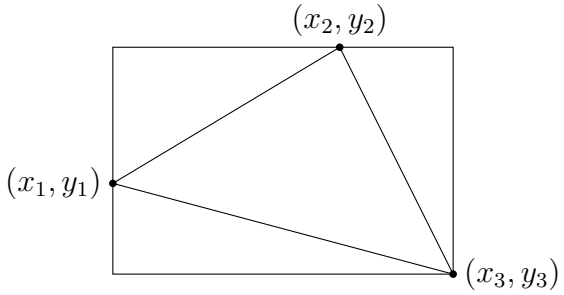

Figura 14.2: Trei puncte necoliniare.

Putem calcula aria triunghiului în următorul mod: Scădem din aria dreptunghiului ariile a trei triunghiuri dreptunghice. Pe exemplul prezentat obținem:

$$(x_3 - x_1)(y_2 - y_3) - \frac{(x_2 - x_1)(y_2 - y_1)}{2} - \frac{(x_3 - x_2)(y_2 - y_3)}{2} - \frac{(x_3 - x_1)(y_1 - y_3)}{2}$$

Desfăcând parantezele și reducând termenii asemenea, vom obține același lucru (eventual cu semne schimbate). Exemplul este unul particular, dar indiferent de poziționarea vârfurilor triunghiului în raport cu dreptunghiul, relația este valabilă (de exemplu, putem avea două vârfuri ale triunghiului suprapuse cu două ale dreptunghiului, vecine sau diagonal opuse).

14.1.3 Ecuația dreptei în plan

Pentru 3 valori date a, b și c, toate punctele din plan care verifică relația $a \cdot x + b \cdot y + c = 0$ se găsesc pe aceeași dreaptă. Dându-se două puncte (x_1, y_1) și (x_2, y_2), pentru a scrie ecuația dreptei determinată de ele facem următorul raționament. Fie un alt punct aflat pe dreapta determinată de cele două puncte date. Din cele de mai sus, înseamnă că el verifică relația: $(x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1) = 0$. Efectuând operațiile și cuplând altfel termenii obținem: $(y_1 - y_2)x + (x_2 - x_1)y + x_1(y_2 - y_1) - y_1(x_2 - x_1) = 0$, deci:

$$\begin{aligned} a &= y_1 - y_2 \\ b &= x_2 - x_1 \\ c &= x_1(y_2 - y_1) - y_1(x_2 - x_1) \end{aligned}$$

Odată obținute a, b, c ca mai sus, așa cum am spus, pentru toate punctele (x, y) de pe dreaptă, expresia $a \cdot x + b \cdot y + c$ are valoarea 0. Mai mult, este de o mare valoare următorul rezultat: toate punctele din același semiplan, introduse în ecuația dreptei, fac ca expresia să dea o valoare nenulă și cu același semn. Adică pentru toate punctele (x, y) dintr-un semiplan avem: $a \cdot x + b \cdot y + c > 0$ și pentru toate punctele (x, y) din celălalt semiplan avem $a \cdot x + b \cdot y + c < 0$.

14.2 Aplicații rezolvate

14.2.1 Verificare dacă un punct se găsește pe un segment

Enunț

Se dau un punct și un segment în plan. Să se verifice dacă punctul se găsește pe segment.

Date de intrare

Fișierul `punctsegment.in` conține pe prima linie 6 numere naturale separate prin spații, respectiv: $X_1, Y_1, X_2, Y_2, X_3, Y_3$. Segmentul are capetele (X_2, Y_2) și (X_3, Y_3).

Date de ieșire

Fișierul `punctsegment.out` conține pe primul rând `DA` (dacă punctul de coordonate (X_1, Y_1) se găsește pe segment) sau `NU` (în caz contrar).

Restricții și precizări

- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Segmentul are lungimea nenulă.

Exemple

punctsegment.in	punctsegment.out
2 2 1 1 3 3	DA

Descrierea soluției

Verificăm mai întâi dacă punctul (X_1, Y_1) se găsește pe dreapta determinată de punctele (X_2, Y_2) și (X_3, Y_3). Acest lucru este necesar, dar nu suficient. Trebuie în plus ca X_1 să aparțină intervalului $[\min(X_2, X_3), \max(X_2, X_3)]$ și Y_1 să aparțină intervalului $[\min(Y_2, Y_3), \max(Y_2, Y_3)]$. Dacă sunt puse ambele condiții se rezolvă și cazurile particulare când dreapta suport a segmentului este paralelă cu una dintre axele sistemului de coordonate.

Implementare

```
#include <fstream>
#include <stdlib.h>
using namespace std;

ifstream fin ("punctsegment.in");
ofstream fout("punctsegment.out");
int X1, Y1, X2, Y2, X3, Y3;

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
  int d = det(x1, y1, x2, y2, x3, y3);
  if (d!=0)
    return 0;
  if (x1 == x3 && y1 == y3)
    return 1;
  if (x2 == x3 && y2 == y3)
    return 1;
  if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
    return 1;
  else
    return 0;
}

void msgAndOut(const char *msg) {
  fout<<msg;
  exit(0);
}

int main() {
  fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
  if (punctPeSegment(X3, Y3, X2, Y2, X1, Y1))
    msgAndOut("DA\n");
  else
    msgAndOut("NU\n");
  return 0;
}
```

14.2.2 Verificarea intersecției a două segmente

Enunț

Se dau două segmente în plan, specificate prin coordonatele capetelor. Să se verifice dacă au cel puțin un punct comun.

Date de intrare

Fișierul `intersectiesegmente.in` conține pe prima linie 8 numere naturale separate prin spații, respectiv: $X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4, Y_4$. Primul segment are capetele (X_1, Y_1) și (X_2, Y_2).

Date de ieșire

Fișierul `intersectiesegmente.out` conține pe primul rând `DA` (când segmentele se intersectează) sau `NU` (în caz contrar).

Restricții și precizări

- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Ambele segmente au lungimea nenulă.

Exemple

intersectiesegmente.in	intersectiesegmente.out
-1 -1 1 1 -1 1 1 -1	DA

Descrierea soluției

Pentru a elimina diverse cazuri particulare, putem mai întâi testa următoarele:

- dacă un punct al unui segment coincide cu un punct al celuilalt segment;
- dacă un punct al unui segment se află pe celălalt segment;

Rămâne de tratat cazul general. Introducând coordonatele punctelor unui segment în ecuația dreptei determinate de celălalt segment trebuie să obținem semne contrare. Nu este suficient să facem testul doar pentru unul dintre segmente, ci pentru ambele (vezi Figura 14.3).

Implementare

```
#include <fstream>
#include <stdlib.h>
using namespace std;

ifstream fin ("intersectiesegmente.in");
ofstream fout("intersectiesegmente.out");
int X1, Y1, X2, Y2, X3, Y3, X4, Y4;
int d1, d2, d3, d4;
```

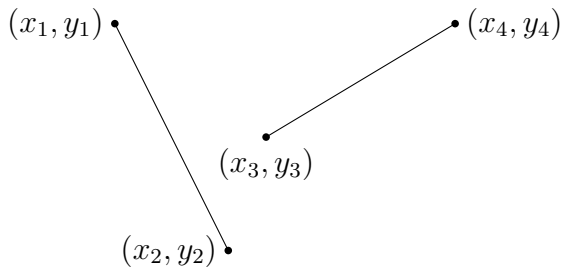

Figura 14.3: Deși punctele (X_1, Y_1) și (X_2, Y_2) introduse în ecuația dreptei determinate de celelalte două puncte dau semne contrare, segmentele nu se intersectează, fiind necesar ca și punctele (X_3, Y_3) respectiv (X_4, Y_4) introduse în ecuația dreptei determinate de (X_1, Y_1) și (X_2, Y_2) să dea semne contrare.

```
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
  int d = det(x1, y1, x2, y2, x3, y3);
  if (d!=0)
    return 0;
  if (x1 == x3 && y1 == y3)
    return 1;
  if (x2 == x3 && y2 == y3)
    return 1;
  if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
    return 1;
  else
    return 0;
}

void msgAndOut(const char *msg) {
  fout<<msg;
  exit(0);
}

int main() {
  fin>>X1>>Y1>>X2>>Y2>>X3>>Y3>>X4>>Y4;
  if (punctPeSegment(X1, Y1, X2, Y2, X3, Y3)) {
    msgAndOut("DA\n");
  }
  if (punctPeSegment(X1, Y1, X2, Y2, X4, Y4)) {
    msgAndOut("DA\n");
  }
  if (punctPeSegment(X3, Y3, X4, Y4, X1, Y1)) {
    msgAndOut("DA\n");
  }
  if (punctPeSegment(X3, Y3, X4, Y4, X2, Y2)) {
    msgAndOut("DA\n");
  }
  d1 = det(X3, Y3, X4, Y4, X1, Y1);
  d2 = det(X3, Y3, X4, Y4, X2, Y2);
  d3 = det(X1, Y1, X2, Y2, X3, Y3);
  d4 = det(X1, Y1, X2, Y2, X4, Y4);
```

```
  if (d1 * d2 < 0 && d3 * d4 < 0)
    msgAndOut("DA\n");
  else
    msgAndOut("NU\n");
  return 0;
}
```

În continuare, s-ar putea formula și cerința: care sunt coordonatele punctului de intersecție al dreptelor suport pentru cele două segmente? Pentru rezolvarea acestei cerințe scriem mai întâi ecuațiile celor două drepte și vom obține:

$$a_1x + b_1y + c_1 = 0, \text{ pentru dreapta determinată de punctele } (X_1, Y_1) \text{ și } (X_2, Y_2)$$
$$a_2x + b_2y + c_2 = 0, \text{ pentru dreapta determinată de punctele } (X_3, Y_3) \text{ și } (X_4, Y_4)$$

Dacă o ecuație este combinație liniară a celeilalte (adică există o valoare nenulă k așa încât $a_1 = ka_2$, $b_1 = kb_2$ și $c_1 = kc_2$), atunci cele două drepte coincid și putem spune că au o infinitate de puncte în comun. Exemplu: $2x+3y+4 = 0$ respectiv $6x+9y+12 = 0$. Dacă există k nenul încât $a_1 = ka_2$, $b_1 = kb_2$ și $c_1 \neq kc_2$ atunci cele două drepte nu coincid dar sunt paralele. Exemplu: $2x + 3y + 4 = 0$ respectiv $6x + 9y + 11 = 0$. În celelalte cazuri cele două drepte au un singur punct de intersecție pe care îl determinăm ca fiind soluția unică a sistemului de ecuații:

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases}$$

Un mod de rezolvare este prezentat în continuare. Presupunem că înmulțim prima ecuație cu a_2 și pe a doua cu $-a_1$.

$$\begin{cases} +a_2a_1x + a_2b_1y + a_2c_1 = 0 \\ -a_1a_2x - a_1b_2y - a_1c_2 = 0 \end{cases}$$

Adunând cele două relații observăm că se reduc termenii ce îl conțin pe x și îl putem exprima pe y:

$$y = \frac{a_1c_2 - a_2c_1}{a_2b_1 - a_1b_2}$$

De fapt doar această formulă ne este necesară în calcule. Observăm că numitorul nu poate fi 0 (din relațiile de calcul pentru coeficienții a și b rezultă că numitorul ar fi 0 doar dacă cele două drepte sunt paralele sau coincid). Pe x îl putem calcula cu relația $x = (-b_1y - c_1)/a_1$. Cazul în care a_1 este 0 corespunde cazului particular că prima dreaptă este paralelă cu axa Ox. În acest caz îl extragem pe x din a doua ecuație (nu pot fi simultan 0 a_1 și a_2 căci ar însemna că dreptele sunt paralele sau coincid).

14.2.3 Determinarea distanței de la un punct la o dreaptă

Enunț

Se dau în plan, un punct și o dreaptă. Să se determine distanța de la punct la dreaptă.

Date de intrare

Fișierul `distantapunctdreapta.in` conține pe prima linie 6 numere naturale separate prin spații, respectiv: $X_1, Y_1, X_2, Y_2, X_3, Y_3$. Se cere determinarea distanței de la punctul de coordonate (X_1, Y_1) la dreapta care trece prin punctele de coordonate (X_2, Y_2) și (X_3, Y_3).

Date de ieșire

Fișierul `distantapunctdreapta.out` conține pe primul rând un număr real cu exact două zecimale exacte (fără rotunjire).

Restricții și precizări

- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Punctele care determină dreapta sunt distincte.

Exemple

distantapunctdreapta.in	distantapunctdreapta.out
0 1 0 0 1 0	1.00

Descrierea soluției

Vom exprima în două moduri aria triunghiului determinat de cele trei puncte.

- prin formula dedusă la începutul prezentării: $\frac{|(x_2-x_1)(y_3-y_1)-(x_3-x_1)(y_2-y_1)|}{2}$;
- prin arhicunoscuta formulă baza $\times$ înălțimea / 2.

În cazul nostru, baza este chiar distanța dintre punctele (X_2, Y_2) și (X_3, Y_3) iar înălțimea este chiar valoarea care ni se cere. Așadar obținem rezultatul:

$$\frac{|(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)|}{\sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2}}$$

Se păstrează valoarea absolută a rezultatului.

Dacă avem deja scrisă ecuația dreptei la care avem de calculat distanța ca fiind: $ax + by + c = 0$, atunci formula se mai poate scrie:

$$\frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}$$

Implementare

```
#include <fstream>
#include <cmath>
#include <iomanip>
using namespace std;

ifstream fin ("distantapunctdreapta.in");
ofstream fout("distantapunctdreapta.out");
int X1, Y1, X2, Y2, X3, Y3;

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int distantaPuncte(int X1, int Y1, int X2, int Y2) {
  return (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2);
}

int main() {
  fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
  int a = det(X1, Y1, X2, Y2, X3, Y3);
  if (a < 0)
    a = -a;
  int d = distantaPuncte(X2, Y2, X3, Y3);
  double h = a/sqrt(d);
  fout<<setprecision(2)<<fixed<< (int)(h*100)/100.0;
  return 0;
}
```

14.2.4 Determinarea distanței de la un punct la un segment

Enunț

Se dau în plan, un punct și un segment. Să se determine distanța minimă de la punctul dat la un punct aparținând segmentului.

Date de intrare

Fișierul `distantapunctsegment.in` conține pe prima linie 6 numere naturale separate prin spații, respectiv: $X_1, Y_1, X_2, Y_2, X_3, Y_3$. Se cere determinarea distanței minime de la punctul de coordonate (X_1, Y_1) la un punct aparținând segmentului cu capetele în punctele (X_2, Y_2) și (X_3, Y_3).

Date de ieșire

Fișierul `distantapunctsegment.out` conține pe primul rând un număr real cu exact două zecimale (fără rotunjire).

Restricții și precizări

- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.

- Punctele care determină segmentul sunt distincte.

Exemple

distantapunctsegment.in	distantapunctsegment.out
0 1 0 0 1 0	1.00

Descrierea soluției

Dacă piciorul perpendicularei dusă din punct pe dreapta suport a segmentului cade chiar pe segment, problema se reduce la cea anterioară. Altfel, distanța de la punct la segment este egală cu distanța de la punct la unul dintre capetele segmentului.

Testăm mai întâi dacă punctul este chiar pe segment, caz în care rezultatul este 0. Apoi pentru a decide dacă piciorul perpendicularei cade chiar pe segment facem observația: considerând triunghiul format de cele trei puncte, piciorul perpendicularei dusă din (X_1, Y_1) pe dreapta determinată de punctele (X_2, Y_2) și (X_3, Y_3) este pe segment dacă atât unghiul din (X_2, Y_2) cât și cel din (X_3, Y_3) sunt mai mici sau egale cu 90 de grade (vezi Figura 14.4).

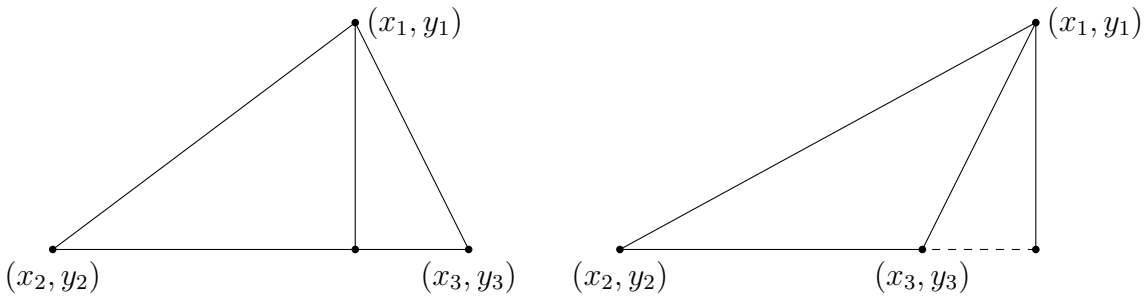

Figura 14.4: Piciorul perpendicularei dintr-un punct pe un segment.

Pentru a verifica dacă un unghi este ascuțit sau obtuz, ne putem folosi de Teorema lui Pitagora, așa cum se observă din Figura 14.5.

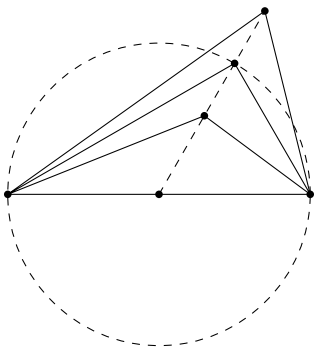

Figura 14.5: Diferențierea între unghiuri ascuțite, drepte și obtuze cu ajutorul Teoremei lui Pitagora.

Considerăm cele trei triunghiuri cu o latură ca fiind diametrul desenat și cu cel de-al treilea vârf așa cum se vede în figură. Triunghiul care are al treilea vârf pe cerc are acel unghi de 90°. Suma pătratelor laturilor sale (altele decât cea comună cu diametrul) este

egală cu pătratul diametrului. Triunghiul care are al treilea vârf în interiorul cercului (și acel unghi este mai mare decât 90°) are suma pătratelor laturilor (altele decât cea comună cu diametrul) mai mică decât pătratul diametrului iar pentru triunghiul cu vârful în afara cercului (deci cu acel unghi ascuțit), suma pătratelor laturilor (altele decât cea comună cu diametrul) este mai mare decât pătratul diametrului.

Așadar, pentru a testa dacă unghiul este obtuz putem verifica inegalitatea (din al treilea caz) din teorema lui Pitagora.

Implementare

```
#include <fstream>
#include <cmath>
#include <iomanip>
using namespace std;

ifstream fin ("distantapunctsegment.in");
ofstream fout("distantapunctsegment.out");
int X1, Y1, X2, Y2, X3, Y3;

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int distantaPuncte(int X1, int Y1, int X2, int Y2) {
  return (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2);
}

int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
  int d = det(x1, y1, x2, y2, x3, y3);
  if (d!=0)
    return 0;
  if (x1 == x3 && y1 == y3)
    return 1;
  if (x2 == x3 && y2 == y3)
    return 1;
  if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
    return 1;
  else
    return 0;
}

int main() {
  fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
  int a = det(X1, Y1, X2, Y2, X3, Y3);
  if (a == 0) {
    if (punctPeSegment(X3, Y3, X2, Y2, X1, Y1)) {
      fout<<setprecision(2)<<fixed<< 0.0;
    } else {
      int d1 = distantaPuncte(X1, Y1, X2, Y2);
      int d2 = distantaPuncte(X1, Y1, X3, Y3);
      if (d1 < d2)
        fout<<setprecision(2)<<fixed<< (int)(sqrt(d1)*100)/100.0;
      else
        fout<<setprecision(2)<<fixed<< (int)(sqrt(d2)*100)/100.0;
    }
```

```
    return 0;
  }
  if (a < 0)
    a = -a;
  int d1 = distantaPuncte(X2, Y2, X3, Y3);
  int d2 = distantaPuncte(X1, Y1, X3, Y3);
  int d3 = distantaPuncte(X2, Y2, X1, Y1);
  if (d2 < d1 + d3 && d3 < d1 + d2) {
    double h = a/sqrt(d1);
    fout<<setprecision(2)<<fixed<< (int)(h*100)/100.0;
  } else {
    int d1 = distantaPuncte(X1, Y1, X2, Y2);
    int d2 = distantaPuncte(X1, Y1, X3, Y3);
    if (d1 < d2)
      fout<<setprecision(2)<<fixed<<(int)(sqrt(d1)*100)/100.0;
    else
      fout<<setprecision(2)<<fixed<<(int)(sqrt(d2)*100)/100.0;
  }
  return 0;
}
```

14.2.5 Determinarea ariei unui poligon simplu (ale cărui laturi nu se autointersectează)

Enunț

Se dau coordonatele în plan pentru n puncte. Să se afișeze valoarea ariei poligonului pe care acestea îl formează.

Date de intrare

Fișierul `ariapoligonsimplu.in` conține pe prima linie numărul de vârfuri ale poligonului, notat cu n. Pe următoarele n linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui vârf. Acestea sunt date într-un sens de parcurgere a laturilor poligonului.

Date de ieșire

Fișierul `ariapoligonsimplu.out` conține pe primul rând un numpăr natural, cu exact o zecimală, reprezentând valoarea cerută.

Restricții și precizări

- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Poligonul nu este neapărat convex, dar nu se autointersectează.

Exemple

ariapoligonsimplu.in	ariapoligonsimplu.out
4 0 0 1 0 1 1 0 1	1.0

Descrierea soluției

Să analizăm mai întâi cazul poligonului convex. Putem alege un punct în interiorul său (care poate fi și unul dintre vârfuri) și însumăm ariile triunghiurilor formate cu acel punct și oricare două vârfuri vecine ale poligonului (vezi Figura 14.6).

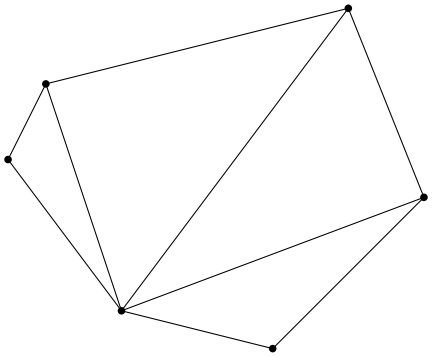

Figura 14.6: Aria unui poligon convex.

Pentru cazul general pare că acest lucru nu mai este valabil. Însă vom vedea că lucrurile sunt aproape la fel de simple. Să analizăm mai întâi Figura 14.7:

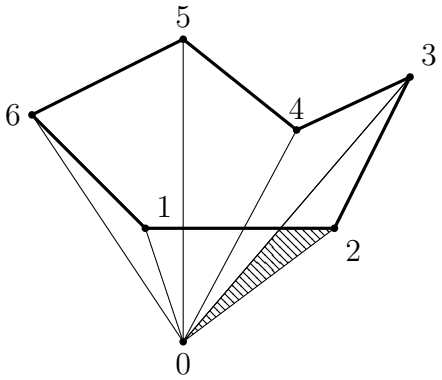

Figura 14.7: Aria unui poligon convex.

Vom calcula valoarea D (cu semn) pentru următoarele triplete de puncte (0,1,2), (0,2,3), (0,3,4), (0,4,5), (0,5,6), (0,6,1). Adică vom considera triunghiurile formate de punctul 0 cu fiecare latură a poligonului, dar vom păstra un sens de parcurgere (coordonatele punctelor unui triplet se vor scrie în formulă în ordinea dată). Pentru triunghiul hașurat observăm că aria sa se adună cu un semn în cazul scrierii lui D pentru triunghiul (0, 1, 2) și cu celălalt semn în cazul scrierii lui D pentru triunghiul (0, 2, 3). Aceasta pentru

că punctul 0 este de o parte a dreptei (1, 2) în sensul de orientare a sa de la 1 la 2, dar este de cealaltă parte a dreptei (2, 3) dacă o privim orientată de la 2 la 3. Raționamentul este valabil pentru toate zonele exterioare poligonului, deci valoarea ariei lor se va anula. Pentru zonele interioare valoarea se va aduna o singură dată și întotdeauna cu același semn. Așadar, rezultatul este:

$$\frac{|D_{(0,1,2)} + D_{(0,2,3)} + \cdots + D_{(0,n-1,n)} + D_{(0,n,1)}|}{2}$$

Implementare

```
#include <fstream>
#include <iomanip>
#define DIM 100010
using namespace std;

ifstream fin ("ariapoligonsimplu.in");
ofstream fout("ariapoligonsimplu.out");
pair<int, int> v[DIM];
int sol;
int n, i;

int aria(pair<int, int> a, pair<int, int> b, pair<int, int> c) {
  return  (b.first-a.first) * (c.second-a.second) -
    (c.first-a.first) * (b.second-a.second);
}

int main() {
  fin>>n;
  for (i=1;i<=n;i++) {
    fin>>v[i].first>>v[i].second;
  }
  v[0] = v[n];
  for (i=0;i<n;i++) {
    sol += aria(v[0], v[i], v[i+1]);
  }
  fout<<setprecision(1)<<fixed<<sol/2.0;
  return 0;
}
```

14.2.6 Ordonarea unor puncte date în plan după unghiul pe care segmentul ce le unește cu originea îl face cu axa Ox

Enunț

Se dau puncte distincte în plan. Asociem fiecărui punct semidreapta care pornește din originea sistemului de coordonate și trece prin acel punct. Să se afișeze punctele în ordine crescătoare a unghiului pe care semidreapta asociată îl face cu semidreapta dusă către plus infinit, a axei Ox. Dacă două unghiuri sunt egale se va afișa punctul cel mai apropiat de origine.

Date de intrare

Fișierul `sortareunghi.in` conține pe prima linie un număr n, reprezentând numărul de puncte. Pe următoarele n linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui punct.

Date de ieșire

Fișierul `sortareunghi.out` conține n linii cu câte două numere separate prin spațiu fiecare, reprezentând abscisa respectiv ordonata câte unui punct, în ordinea cerută.

Restricții și precizări

- $1 \leq n \leq 100$
- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Unghiurile sunt în intervalul $[0°, 360°)$.
- Punctul $(0, 0)$ nu se găsește în fișierele de intrare.

Exemple

sortareunghi.in	sortareunghi.out
3 1 1 -1 -1 -1 1	1 1 -1 1 -1 -1

Descrierea soluției

Să analizăm Figura 14.8, în care am numerotat punctele chiar cu poziția pe care o ocupă în șirul final.

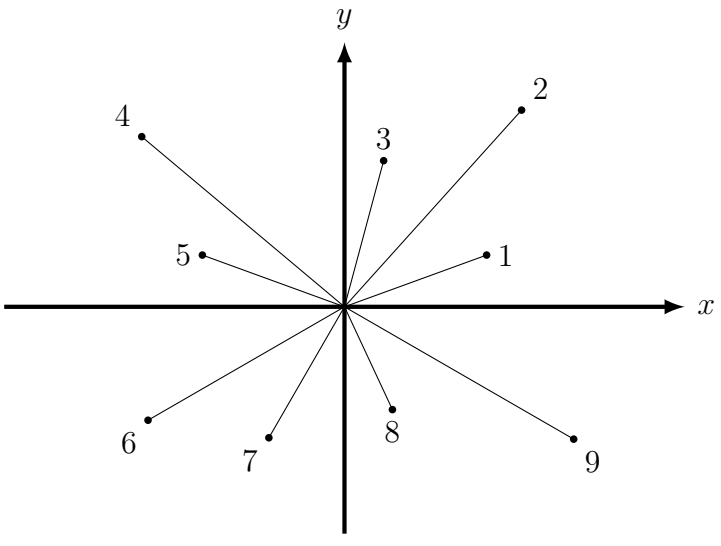

Figura 14.8: Sortarea unui set de puncte după unghi.

Observăm că dacă am scrie $D_{(0,1,2)}$ și $D_{(0,2,1)}$ am obține semne diferite. Dacă șirul ar fi sortat vom avea deci mereu același semn pentru $D_{(0,i,i+1)}$. Semnul lui D este deci criteriul pe care îl considerăm la compararea a 2 puncte în funcția de sortare. Apar însă probleme dacă punctele sunt coliniare. În acest caz valoarea D este 0 și când punctele sunt în același cadran, și când sunt în cadrane diferite, deci criteriul semnului lui D nu se mai poate aplica. Această problemă se rezolvă considerând ca prim criteriu de sortare cadranul în care se află cele două puncte de comparat. Astfel, dacă $X > 0$ și $Y \geq 0$ considerăm cadranul 1, pentru $X \leq 0$ și $Y > 0$ considerăm cadranul 2, pentru $X < 0$ și $Y \leq 0$ considerăm cadranul 3 iar pentru $X \geq 0$ și $Y < 0$ cadranul 4. Evident, al treilea criteriu de sortare, ca prioritate este distanța față de origine, conform cerinței.

O altă observație utilă pentru reducerea calculelor este că dacă unul dintre cele 3 puncte pentru care calculăm valoarea D este originea, formula D devine: $x_2y_1 - x_1y_2$. În practică sunt multe cazuri în care toate punctele de sortat sunt în același cadran sau toate deasupra axei Ox. Este suficient ca și criteriu de comparare semnul expresiei scrise anterior.

Implementare

```
#include <fstream>
#include <algorithm>
using namespace std;

ifstream fin ("sortareunghi.in");
ofstream fout("sortareunghi.out");
pair<int, int> v[103];
int n, i;

int cadran(int x, int y) {
  if (x > 0 && y >= 0)
    return 1;
  if (x >= 0 && y<0)
    return 4;
  if (y > 0 && x <= 0)
    return 2;
  return 3;
}

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int cmp(const pair<int, int> &a, const pair<int, int> &b) {
  int c1 = cadran(a.first, a.second);
  int c2 = cadran(b.first, b.second);
  if (c1 != c2)
    return c1 < c2;
  else {
    int d = det(0, 0, a.first, a.second, b.first, b.second);
    if (d != 0)
      return d > 0;
    else
      return a.first*a.first+a.second*a.second<b.first*b.first+b.second*b.second;
  }
}
```

```
int main() {
  fin>>n;
  for (i=1;i<=n;i++)
    fin>>v[i].first>>v[i].second;
  sort(v+1, v+n+1, cmp);
  for (i=1;i<=n;i++)
    fout<<v[i].first<<" "<<v[i].second<<"\n";
  return 0;
}
```

14.2.7 Înfășurătoarea convexă (cu număr maxim posibil de puncte pe margine)

Enunț

Se dau n puncte distincte în plan. Să se determine un poligon de arie maximă care are vârfuri dintre punctele date.

Date de intrare

Fișierul `infasuratoareconvexa.in` conține pe prima linie un număr n, reprezentând numărul de puncte. Pe următoarele n linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui punct.

Date de ieșire

Fișierul `infasuratoareconvexa.out` conține pe prima linie un număr k, reprezentând numărul de vârfuri ale poligonului determinat. Pe următoarele k linii se găsesc câte două numere separate printr-un spațiu, reprezentând respectiv abscisa și ordonata câte unui punct.

Restricții și precizări

- $1 \leq n \leq 100$
- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Primul punct care se va afișa va fi cel cu ordonata minimă, iar în caz de egalitate cel cu abscisa minimă.
- Punctele se vor afișa în sens trigonometric al parcurgerii lor pe înfășurătoare.

Exemple

infasuratoareconvexa.in	infasuratoareconvexa.out
5 1 1 0 0 0 2 2 2 2 0	4 0 0 2 0 2 2 0 2

Descrierea soluției

În prima etapă determinăm punctul cu valoarea Y minimă, iar dintre acestea pe acela cu valoarea X minimă. Scăzând coordonatele sale din coordonatele celorlalte puncte translatăm practic originea sistemului de axe în acel punct și toate celelalte se vor afla în cadranele 1 și 2. Ordonăm apoi celelalte punctele ca în problema anterioară. Originea precum și primul și ultimul punct din șirul ordonat vor face parte sigur din înfășurătoare. Pentru început inițializăm șirul punctelor de pe înfășurătoare cu primele două (originea și primul punct din șirul sortat). Parcurgem restul de puncte în ordinea sortării.

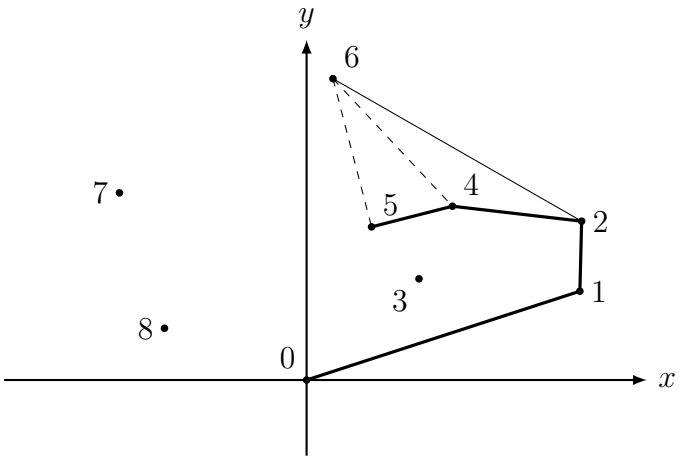

Figura 14.9: Calculul înfășurătorii convexe a unei mulțimi de puncte.

Cu punctele parcurse deja avem construită înfășurătoarea. În Figura 14.9 este reprezentată starea de înaintea analizării punctului 6. Cu punctele de la 1 la 5 este marcată îngroșat înfășurătoarea. Deci șirul punctelor de pe înfășurătoare este 0, 1, 2, 4, 5 (punctul 3 este eliminat). La analizarea punctului 6 observăm că nu putem extinde înfășurătoarea curentă întrucât unghiul care se formează între ultimele 2 puncte din șir (4 și 5) și punctul 6 este unul reflex (identificăm asta prin semnul lui D pentru cele 3 puncte). Așadar, punctul 6 face să dispară din șir punctul 5. Pe același principiu, punctul 6 face să dispară și punctul 4 (unghiul format cu ultimele două puncte, acum 2 și 4 și punctul 6 este tot reflex). Însă, unghiul format din 1,2 și 6 nu este reflex, așadar 6 nu mai elimină puncte din șir iar la final se adaugă el în șir (ultimul punct ca unghi este mereu pe înfășurătoare). O altă observație: cum punctele 0 și 1 fac sigur parte de pe înfășurătoare, mereu vor fi în șir cel puțin 2 puncte.

Șirul punctelor aflate pe parcurs pe înfășurătoare se comportă așadar ca o stivă, fiecare punct dat ajungând acolo o dată (după ce eventual a eliminat dintre punctele puse în stivă anterior). Timpul de rulare a codului descris mai sus este așadar de ordinul numărului de puncte care se dau.

Pentru a obține număr maxim de puncte de pe înfășurătoare nu vom scoate din stivă punctul din vârf atunci când D este 0 (la testarea unghiului).

Ultimul lucru de care trebuie ținut cont este următorul: În cazul în care două puncte sunt coliniare cu originea în ce ordine le sortăm? Observăm că ultimele puncte trebuie parcurse în ordine descrescătoare a distanței față de origine (pentru a rămâne pe înfășurătoare număr maxim de puncte). Singurele puncte care trebuie parcurse în ordine crescătoare ale distanței față de origine sunt cele care fac unghiul minim cu axa Ox (cele aflate

pe dreapta determinată de punctele 0 și 1). Deci, ca al doilea criteriu de sortare vom considera descrescător după distanța față de origine, iar secvența de puncte de la început de pe dreapta determinată de 0 și 1 o vom simetriza.

Implementare

```
#include <fstream>
#include <algorithm>
#include <cmath>
#define INF 1002
using namespace std;

ifstream fin ("infasuratoareconvexa.in");
ofstream fout("infasuratoareconvexa.out");

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int cmp(const pair<int, int> &a, const pair<int, int> &b) {
  int d = det(0, 0, a.first, a.second, b.first, b.second);
  if (d != 0)
    return d > 0;
  else
    return a.first*a.first + a.second*a.second > b.first*b.first + b.second*b.second;
}

pair<int, int> v[103], s[103], aux;
int n, i, j, k, pminim;

int main() {
  fin>>n;
  pminim = 0;
  v[0].first = v[0].second = INF;
  for (i=1;i<=n;i++) {
    fin>>v[i].first>>v[i].second;
    if (v[i].second < v[pminim].second || ( (v[i].second == v[pminim].second)&&(v[i].first < v[pmini
      pminim = i;
  }

  v[0] = v[pminim];
  v[pminim] = v[1];
  v[1] = v[0];
  for (i=1;i<=n;i++) {
    v[i].first -= v[0].first;
    v[i].second -= v[0].second;
  }

  sort(v+2, v+n+1, cmp);

  for (j=3;j<=n;j++)
    if (det(v[1].first, v[1].second, v[2].first, v[2].second, v[j].first, v[j].second)) {
      break;
    }

  i=2;
```

```
  j--;

  while (i<j) {
    aux = v[i];
    v[i] = v[j];
    v[j] = aux;
    i++;
    j--;
  }

  s[1] = v[1];
  s[2] = v[2];
  k = 2;
  for (i=3;i<=n;i++) {
    while (k>=2&&det(s[k-1].first,s[k-1].second,s[k].first,s[k].second,v[i].first, v[i].second)<0) {
      k--;
    }
    s[++k] = v[i];
  }

  fout<<k<<"\n";
  for (i=1;i<=k;i++)
    fout<<s[i].first + v[0].first<<" "<<s[i].second + v[0].second<<"\n";

  return 0;
}
```

14.2.8 Verificarea dacă un punct este în interiorul sau pe perimetrul unui poligon

Enunț

Se dau coordonatele în plan pentru n puncte care determină un poligon. Se mai dau coordonatele altor m puncte. Să se verifice, pentru fiecare dintre cele m puncte, dacă se găsește sau nu în interiorul (sau pe marginea) poligonului.

Date de intrare

Fișierul `punctinpoligonsimplu.in` conține pe prima linie două numere separate prin spațiu: n și m, reprezentând respectiv, numărul de vârfuri ale poligonului și numărul de puncte de testat. Pe următoarele n linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui vârf al poligonului. Acestea sunt date într-un sens de parcurgere a laturilor poligonului. Pe următoarele m linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui punct care trebuie testat.

Date de ieșire

Fișierul `punctinpoligonsimplu.out` conține m linii și pe fiecare dintre ele se află mesajul `DA` sau `NU` după cum punctul corespunzător este sau nu în interiorul poligonului.

Restricții și precizări

- $1 \leq n, m \leq 100$
- Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001.
- Poligonul nu este neapărat convex, dar nu se autointersectează.
- În primele două teste poligonul este convex.

Exemple

punctinpoligonsimplu.in	punctinpoligonsimplu.out
4 2 0 0 2 0 2 2 0 2 1 1 10 -2	DA NU

Descrierea soluției

Dacă poligonul este convex testul pentru un punct poate fi făcut astfel: Calculăm semnul valorilor D pentru toate tripletele formate din punctul de testat și oricare două puncte vecine pe poligon (respectând mereu un sens de parcurgere). Dacă se obține întotdeauna același semn atunci punctul se află în poligon.

Dacă poligonul nu este convex, observația cheie este următoarea: punctul este în poligon dacă și numai dacă există o semidreaptă (de fapt orice semidreaptă) cu originea în punctul de testat care va începa poligonul de un număr impar de ori. Astfel, testul ar avea drept cărămidă testarea intersecției a două segmente. În funcție de modul de alegere a semidreptei, pot apărea cazuri particulare (când semidreapta trece printr-un vârf al poligonului).

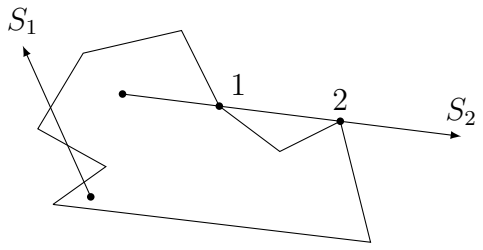

Figura 14.10: Intersecția unei semidrepte cu un poligon.

Observăm că o semidreaptă aleasă precum S_1 este una potrivită. O semidreaptă aleasă precum S_2 poate face să apară cazuri particulare mai dificil de tratat. De exemplu, dacă pe ea se află un punct ca 1, se observă că la întâlnirea lui trece din interiorul în exteriorul poligonului, iar dacă pe ea se află un punct ca 2, va rămâne în exterior (sau, după caz, în interior).

Pentru problema enunțată, datele fiind mici (atât numarul de puncte cât și coordonatele), este probabilitate foarte mare ca alegând aleator un punct la coordonate mai mari,

dreapta ce se obține unindu-l cu punctul de testat să evite toate vârfurile poligonului. Așadar, în acest mod alegem semidreapta (generăm aleator celălalt punct de pe dreapta suport până când această dreaptă nu trece prin niciun vârf al poligonului dat).

Implementare

```
#include <fstream>
#include <cmath>
#include <iomanip>
#include <stdlib.h>
#include <time.h>
using namespace std;

ifstream fin ("punctinpoligonsimplu.in");
ofstream fout("punctinpoligonsimplu.out");

int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}

int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
  int d = det(x1, y1, x2, y2, x3, y3);
  if (d!=0)
    return 0;
  if (x1 == x3 && y1 == y3)
    return 1;
  if (x2 == x3 && y2 == y3)
    return 1;
  if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
    return 1;
  else
    return 0;
}

int n, m, i, j, X1, Y1, intersect, testcurent, X3, Y3, X4, Y4, d1, d2, d3, d4;
pair<int, int> p[102];

int main() {
  fin>>n>>m;
  for (i=1;i<=n;i++)
    fin>>p[i].first>>p[i].second;
  p[0] = p[n];
  int X2 = 1002;
  int Y2 = 0;
  srand(time(0));

  for (j=1;j<=m;j++) {
    fin>>X1>>Y1;
    int ok = 0;
    for (i=0;i<n;i++) {
      if (punctPeSegment(p[i].first, p[i].second, p[i+1].first, p[i+1].second,
                         X1, Y1)) {
        ok = 1;
        break;
      }
    }
```

```
    if (ok) {
      fout<<"DA\n";
      continue;
    }

    do {
      X2 = 1002 + rand()%1000;
      Y2 = 1002 + rand()%1000;
      intersect = 0;
      testcurent = 1;
      for (i=0;i<n;i++) {
        X3 = p[i].first;
        Y3 = p[i].second;
        X4 = p[i+1].first;
        Y4 = p[i+1].second;

        d1 = det(X3, Y3, X4, Y4, X1, Y1);
        d2 = det(X3, Y3, X4, Y4, X2, Y2);
        d3 = det(X1, Y1, X2, Y2, X3, Y3);
        d4 = det(X1, Y1, X2, Y2, X4, Y4);
        if (d1 == 0 && d2 == 0) {
          testcurent = 0;
          break;
        }
        if (punctPeSegment(X1, Y1, X2, Y2, X3, Y3) ||
            punctPeSegment(X1, Y1, X2, Y2, X4, Y4)) {
          testcurent = 0;
          break;
        }
        if (d1 * d2 < 0 && d3 * d4 < 0)
          intersect++;
      }

      if (testcurent) {
        if (intersect % 2 == 1) {
          fout<<"DA\n";
        } else {
          fout<<"NU\n";
        }
        break;
      }
    } while (1);
  }
  return 0;
}
```

14.2.9 Problema Popândăi 2 (Infoarena)

Enunț

Popândăii de pe „tarlaua veselă” au scăpat de atacul vulturilor și acum trebuie să se adăpostească de lupi în vizuinile lor. Aceste vizuini se pot identifica prin puncte având coordonate întregi în plan și sunt dispuse în colțurile unui poligon convex. Pentru a fi protejați de atacul lupilor, popândăii vor să stabilească un perimetru de siguranță cât

mai mare posibil, unde se pot mișca în voie. Acest perimetru va fi în formă de patrulater și va avea vârfurile situate în patru din cele N puncte care reprezintă vizuinile.

Ajutați popândăii să determine zona de arie maximă care satisface condițiile de mai sus!

Date de intrare

Pe prima linie a fișierului de intrare `popandai2.in` se află un număr întreg N, reprezentând numărul vizuinilor. Următoarele N linii conțin fiecare câte două numere întregi (X_i, Y_i) (separate printr-un spațiu) reprezentând coordonatele celei de-a i-a vizuini. Aceste coordonate vor fi date în ordine trigonometrică.

Date de ieșire

Pe prima linie a fișierului de ieșire `popandai2.out` va fi afișat un singur număr real cu o zecimală exactă reprezentând aria maximă a patrulaterului căutat.

Restricții și precizări

- $4 \leq Nleq1000$
- $1 \leq X_i, Y_i \leq 30000$
- Pentru 60% din punctaj rezolvați problema pe cazul $3 \leq Nleq300$.

Exemple

popandai2.in	popandai2.out
7 3 2 6 1 9 3 8 7 6 9 3 8 2 4	28.5

Descrierea soluției

Pe scurt, problema spune așa: dat fiind un poligon convex, se cere să determinăm un patrulater convex cu vârfurile dintre cele ale poligonului și de arie maximă.

Soluția 1. Vom folosi 4 foruri, unul în altul, fixând în toate modurile cele 4 vârfuri ale patrulaterului. Timpul de calcul este $\mathcal{O}(n^4)$.

Soluția 2. Observăm că dacă fixăm o diagonală a poligonului, de o parte și de alta a ei se formează două triunghiuri. Ne interesează să maximizăm aria fiecăruia dintre ele. Întrucât ele au baza stabilită (diagonala fixată), rămâne să alegem, pentru fiecare, celălalt vârf încât înălțimea să fie cât mai mare. Practic, avem un `for` de o parte a diagonalei fixate, pentru a alege punctul care împreună cu cele de pe diagonală maximizează aria triunghiului și încă un `for`, separat, pentru a face același raționament de cealaltă parte

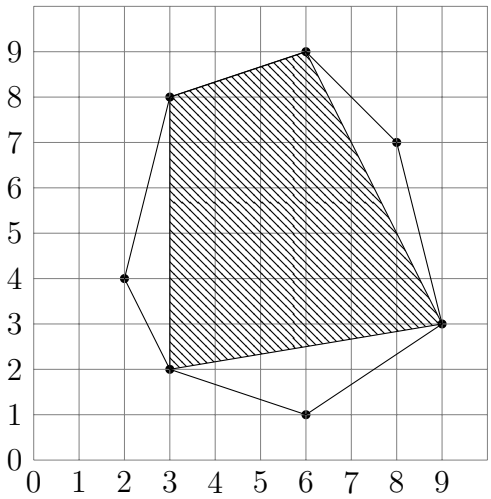

Figura 14.11: Imagine pentru datele din exemplu.

a diagonalei. Timpul total de calcul ajunge la $\mathcal{O}(n^3)$ (avem nevoie de n^2 pentru a fixa diagonala).

Soluția 3. Vom folosi ideea de la soluția anterioară, fixând o diagonală. Cu un `for` stabilim un punct al poligonului și cu al doilea `for` punctul diagonal opus. Să ne imaginăm că suntem la o anume diagonală și, folosind raționamentul de la soluția anterioară, am determinat și cele două puncte care maximizează aria poligonului (adică aria fiecăruia dintre triunghiurile aflate de o parte și de alta a diagonalei).

Ne gândim acum că trecem la următoarea iterare a celui de-al doilea `for` (adică, alegem altă diagonală, păstrând pentru ea același punct de „început" ca și la anterioara, dar ca punct de final trecem la următorul de pe poligon). Acum este observația cheie: punctele care maximizează ariile celor două triunghiuri (pentru noua diagonală aleasă) sunt „după" (în sensul de parcurgere - să zicem trigonometric) față de cele de la poziția anterioară a diagonalei. Să analizăm figura 14.12.

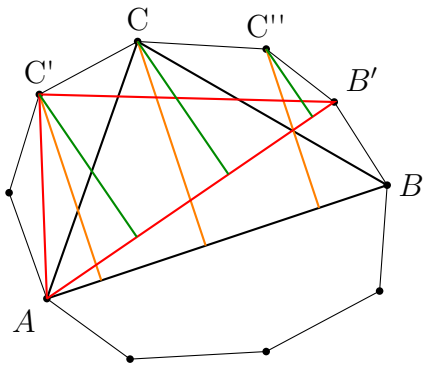

Figura 14.12: Recalcularea ariei maxime la avansul diagonalei.

Considerăm diagonala AB. De o parte a ei, maximizăm aria alegând punctul C. Nu am mai figurat ce se întâmplă de cealaltă parte pentru că se face un raționament similar.

Dacă trecem la altă poziție a diagonalei (păstrăm pe A și ne mutăm cu celălalt capăt în B'), celălalt vârf al triunghiului fie rămâne C, fie va fi un punct de după el (de exemplu C'), dar în niciun caz un punct dinaintea lui (precum C'').

Iată o justificare rapidă: Întrucât C a fost cel mai depărtat punct de AB, segmentul portocaliu care îl proiectează pe C pe AB este mai lung decât cel portocaliu care îl proiectează pe C' pe AB. Mutându-ne cu diagonala către C' (adică A rămâne nemodificat și B „merge" în B'), proiecția verde a lui C pe AB' va fi și ea mai mare decât cea verde din C'' pe AB' (pentru o justificare mai riguroasă, observăm că se formează trapeze dreptunghice iar proiecția din C este tot timpul latura „paralelă" mai mare).

Așadar, mutând într-un sens extremitatea a doua a diagonalei, pentru determinarea vârfurilor pentru care se maximizează ariile în cele două triunghiuri, continuăm în același sens (sau rămânem pe loc!) pornind din vârfurile determinate la diagonala anterioară.

Astfel, avem nevoie de un `for` prin care fixăm extremitatea inițială a diagonalei, iar pentru el facem încă o parcurgere pentru a fixa cealaltă extremitate a diagonalei, și odată cu ea, și celelalte două parcurgeri pentru a alege optim celelalte vârfuri ale celor două triunghiuri (folosind, cum am justificat, faptul că se continuă în același sens de parcurgere din pozițiile de la pasul anterior). Timpul de executare este $\mathcal{O}(n^2)$.

Mai jos este o sursă pe această idee.

Implementare

```
#include <fstream>
#define DIM 30010
#define x first
#define y second

using namespace std;

int n, i, j, nexti, nextj, S, maxim;

int Next(int i) {
  if (i < n)
    return i+1;
  else
    return 1;
}

int aria(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
  int r = (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
  if (r > 0)
    return r;
  else
    return -r;
}

pair<int, int> v[DIM];

int main () {
  ifstream fin ("popandai2.in");
  ofstream fout("popandai2.out");
  fin>>n;
  for (i=1;i<=n;i++)
    fin>>v[i].x>>v[i].y;

  for (int i=1;i<n;i++) {
```

```
    nexti = i;
    nextj = i+1;
    for (int j=i+1;j<=n;j++) {
      while (nexti != j &&
             aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nexti].x, v[nexti].y)<=
             aria(v[i].x, v[i].y, v[j].x, v[j].y, v[Next(nexti)].x, v[Next(nexti)].y)){
        nexti = Next(nexti);
      }
      while (nextj != i &&
             aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nextj].x, v[nextj].y)<=
             aria(v[i].x, v[i].y, v[j].x, v[j].y, v[Next(nextj)].x, v[Next(nextj)].y)){
        nextj = Next(nextj);
      }

      S = aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nexti].x, v[nexti].y) +
        aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nextj].x, v[nextj].y);

      if (S > maxim)
        maxim = S;
    }
  }

  fout<<maxim/2;
  if (maxim%2 == 0)
    fout<<".0\n";
  else
    fout<<".5\n";
  return 0;
}
```

Ca și detalii de implementare:

- Remarcăm folosirea funcției `Next` care ne ușurează trecerea la vârful următor, având nevoie să parcurgem circular și uneori fiind nevoie să trecem de la vârful n la vârful 1.
- Întrucât aria este fie număr întreg fie jumătate de întreg, lucrăm pe parcurs cu dublul valorilor (evitând deci numerele reale), iar la final afișăm `.0` sau `.5` în funcție de paritatea rezultatului.

14.2.10 Baleiere

În foarte multe cazuri prin baleiere, în contextul problemelor de geometrie, înțelegem faptul că avem o dreaptă imaginară, de exemplu verticală, care traversează planul de la $-\infty$ la $+\infty$, iar la întâlnirea diverselor elemente considerăm că apar evenimente pe care le tratăm.

Considerăm următoarea problemă:

Intersecție segmente

Se dau N segmente în plan, fiecare fiind paralel cu una dintre axele de coordonate. Determinați numărul total de puncte de intersecție între două segmente.

Date de intrare

Prima linie a fișierului `is.in` conține un număr N, ce reprezintă numărul de segmente. Fiecare din următoarele N linii conțin câte 4 numere, separate prin câte un spațiu: X_1 Y_1 X_2 Y_2. X_1 și Y_1 reprezintă abscisa respectiv ordonata unui capăt al segmentului, iar X_2 și Y_2 abscisa respectiv ordonata celuilalt capăt.

Date de ieșire

Fișierul `is.out` conține un singur număr, pe prima linie, reprezentând valoarea cerută.

Restricții și precizări

- $1 \leq N \leq 100\,000$
- $0 \leq X_1, Y_1, X_2, Y_2 \leq 300\,000$
- Oricare două puncte date sunt distincte.
- Nu există capete ale vreunui segment aflate pe alt segment.

Exemple

is.in	is.out
5 1 1 10 1 1 4 5 4 2 0 2 20 3 0 3 3 15 15 20 15	3

Descrierea soluției

O primă variantă de rezolvare este să analizăm segmentele „fiecare cu fiecare" și, atunci când întâlnim un segment orizontal testat cu unul vertical, verificăm dacă se intersectează (cu timp de calcul de ordin constant). Această soluție are însă timp de calcul $\mathcal{O}(n^2)$ și nu se va încadra în timp pentru seturile de date mari.

Să vedem acum o soluție mai bună.

Pentru fiecare punct care apare ca și capăt al vreunui segment reținem segmentul pe care apare și tipul acestui segment (orizontal sau vertical). Sortăm aceste puncte crescător după x (pe implementarea de mai jos, am introdus punctele într-o structură `set`, care le clasifică după x).

Analizăm punctele în ordine crescătoare a abscisei lor.

Când întâlnim un punct care se află pe un segment vertical, ne imaginăm lucrurile astfel:

- segmentul corespunzător punctului este dreapta verticală de baleiere (care se mișcă de la stânga la dreapta, întrucât punctele sunt sortate crescător după x);
- în acest moment sunt unele segmente orizontale la care li s-a întâlnit anterior doar capătul stâng;

- trebuie să vedem care dintre aceste segmente orizontale sunt intersectate de segmentul nostru vertical;

Avem trei tipuri de evenimente:

1. Întâlnim capătul stâng al unui segment orizontal. În acest moment, într-un vector de frecvență F, mărim cu 1 pe poziția valorii y a segmentului orizontal (semnificație: începe un segment orizontal, cele verticale întâlnite până la capătul său drept se poate să îl intersecteze).
2. Întâlnim capătul drept al unui segment orizontal. În acest moment, într-un vector de frecvență F, scădem cu 1 pe poziția valorii y a segmentului orizontal (semnificație: se termină un segment orizontal, deci cele verticale ce vor mai apărea nu îl mai pot intersecta).
3. Întâlnim un segment vertical. Practic, acum trebuie să vedem care este suma valorilor din F aflate între indicii dați de valorile y ale celor două capete ale segmentului vertical.

Următorul pas este să observăm că operațiile pe vectorul F le putem simula pe un arbore de intervale (sau chiar arbore indexat binar).

Astfel, dacă întâlnim un punct (x, y) capăt stâng al unui segment orizontal (evenimentul 1 de mai sus), facem update în AINT la poziția y (mărim cu 1).

Dacă întâlnim un punct (x, y) capăt drept al unui segment orizontal (evenimentul 2 de mai sus), facem update în AINT la poziția y (scădem cu 1).

Dacă întâlnim un punct (x_1, y_1) care se află pe un segment vertical ce are al doilea punct în (x_1, y_2), este suficient să facem query de sumă în AINT pentru intervalul (y_1, y_2). Așa cum spuneam, asta contorizează segmentele „începute" și „neterminate" (deci cu x de început în stânga x-ului curent și cu x de final în dreapta x-ului curent - întrucât, noi știm că analizăm punctele crescător după x), și care au y cuprins între valorile y ale capetelor segmentului vertical. Valoarea obținută este chiar numărul de segmente intersectate de cel curent.

În sursa de mai jos remarcăm și un detaliu de implementare care ușurează lucrurile: pentru segmentele verticale luăm în calcul doar unul dintre puncte ca generator de evenimente.

Am putut folosi arborele de intervale întrucât valorile y sunt relativ mici (cel mult 300 000, deci pot reprezenta indicii elementelor unui vector). Dacă valorile erau mai mari, întrucât numărul lor este relativ mic (cel mult 100 000), am fi putut folosi arborele de intervale după ce procedam în prealabil la o normalizare.

Timpul de calcul este de ordin $\mathcal{O}(n \log n)$ (avem pe de o parte sortarea cu această complexitate, apoi, la fiecare punct întâlnit facem o operație în arborele de intervale, având și ea timp de calcul de ordin logaritmic).

Implementare

```
#include <fstream>
#include <set>
```

```
#define DIM 100010

using namespace std;

ifstream fin ("is.in");
ofstream fout("is.out");

struct segment {
  int x1;
  int y1;
  int x2;
  int y2;

  int getType() {
    if (y1 == y2)
      return 1;
    else
      return 2;
  }
};

set <pair <int, int> > s;

segment v[DIM];
int A[DIM*3*4*2];
int n, N, sol;

void update(int nod, int st, int dr, int poz, int val) {
  if (st == dr) {
    A[nod]+=val;
  } else {
    int mid = (st + dr)/2;
    if (poz <= mid)
      update(2*nod, st, mid, poz, val);
    if (poz > mid)
      update(2*nod+1, mid+1, dr, poz, val);
    A[nod] = A[2*nod] + A[2*nod+1];
  }
}

void query(int nod, int st, int dr, int a, int b) {
  if (a <= st && dr <= b) {
    sol += A[nod];
  } else {
    int mid = (st + dr)/2;
    if (a <= mid)
      query(2*nod, st, mid, a, b);
    if (b > mid)
      query(2*nod + 1, mid+1, dr, a, b);

  }
}

int main () {
  fin>>n;
  for (int i=1;i<=n;i++) {
    fin>>v[i].x1>>v[i].y1>>v[i].x2>>v[i].y2;
```

```
    if (v[i].x1 == v[i].x2) {
      if (v[i].y1 > v[i].y2)
        swap(v[i].y1, v[i].y2);
      s.insert(make_pair(v[i].x1, i));
      N = max(N, v[i].y1);
      N = max(N, v[i].y2);
    } else {
      if (v[i].x1 > v[i].x2)
        swap(v[i].x1, v[i].x2);
      s.insert(make_pair(v[i].x1, i));
      s.insert(make_pair(v[i].x2, i));
      N = max(N, v[i].y1);
      N = max(N, v[i].y2);
    }
  }

  for (set<pair<int, int> >::iterator it = s.begin(); it != s.end(); it++) {
    int x = it->first;
    int i = it->second;

    if (v[i].getType() == 2) {
      query(1, 0, N, v[i].y1, v[i].y2);
    } else {
      if (v[i].x1 == x)
        update(1, 0, N, v[i].y1, 1);
      else
        update(1, 0, N, v[i].y1, -1);
    }
  }
  fout<<sol;
  return 0;
}
```

14.3 Bibliografie

[1] *Aria*, URL: https://www.infoarena.ro/problema/aria.

[2] *Graham scan*, URL: https://en.wikipedia.org/wiki/Graham_scan.

[3] *Computational Geometry - Algorithms for Geometry*, URL: https://www.geeksforgeeks.org/what-is-computational-geometry-and-how-is-it-applied-in-solving-geometric-problems/.

ISBN 978-630-6559-05-3