UNIVERSITATEA ROMÂNO-AMERICANĂ

RAUCODER 2023

INDICAȚII DE REZOLVARE

6 mai 2023

Problema 1. Conflicte

Propunător: Daniela Alexandra Crișan

Cuvinte cheie: vector de frecvență, operații pe biți, reprezentarea submulțimilor ca numere întregi

Indicații

Fiecărui nume de țară i se asociază un număr natural astfel: bitul *i* este setat (are valoarea *1*) dacă cifra *i* face parte din nume. Pentru aceasta se poate folosi operatorul de disjuncție pe biți: | .

Astfel, țările care fac parte din aceeași alianță (au în compunerea numelui lor aceleași cifre, fără să conteze numărul de apariții), vor avea același număr asociat.

Vom folosi acest număr pentru a reține într-un vector de frecvență numărul de țări care fac parte din aceeași alianță. Tările care au frecvența 1 vor fi neutre din punct de vedere politic (întrebarea de tipul 1).

Pentru întrebările de tipul *2*, frecvența corespunzătoare numărului asociat unei țări ne arată câte țări fac parte din aceeași „frăție". Numărul țărilor în conflict se află însumând frecvențele numerelor asociate care nu au cifre în comun cu țara de interes. Pentru aceasta se poate folosi operatorul de conjuncție pe biți *&* .

Complexitate spațială: O(2^10) – pt vectorul de frecvență + O(QMAX) – pt orașele preferate, unde QMAX e numărul de înterogări
Complexitate temporală:

- Teste de tip 1: O(N * log10(Nr) + 2^10), unde Nr e lungimea maximă a numelui unei țări
- Teste de tip 2:
    - Numărul de aliați: O((N+Q) * log10(Nr) + Q)
    - Numărul de țări în conflict: O((N+Q) * log10(Nr) + Q * (2^10))

Soluție

```
#include <fstream>
using namespace std;

ifstream in("conflicte.in");
ofstream out("conflicte.out");

int alianta(int x)
{
    int a = 0;
    while (x)
    {
        a |= (1 << x % 10);
        x /= 10;
    }
    return a;
}

int main()
{

    int T, Q, f[(1 << 10)] = { 0 }, q[1000], x;
    in >> T;
    if (T == 2)
    {
        in >> Q;
        for (int i = 0; i < Q; i++)
        {
            in >> x;
            q[i]=alianta(x);
        }
    }
    while (in >> x) f[alianta(x)]++;

    if (T == 1)
    {
        int cnt = 0;
        for (int i = 1; i < 1 << 10; i++)
        {
            if (f[i] == 1)
                cnt++;
        }
        out << cnt;
    }
    else
    {
         for (int i = 0; i < Q; i++)
        {
            int sum = 0;
            for (int b = 1; b < 1 << 10; b++)
                if ((b & q[i]) == 0)
                    sum += f[b];
            out << f[q[i]] - 1 << ' ' << sum << '\n';
        }
    }
    return 0;}
```

Problema 2. Limbaj formal

Propunător: Daniela Alexandra Crișan

Cuvinte cheie: programare dinamică, exponențiere rapidă de matrici, aritmetică modulară

Indicații

Considerând un alfabet cu *N* simboluri ordonate crescător, orice cuvânt peste acel alfabet este o secvență de *N* biți, unde bitul *i* este setat (are valoarea *1*) dacă cel de-al *i*-lea simbol din alfabet este prezent în cuvânt. Problema se reduce astfel la a număra secvențele de *N* biți în care nu există doi biți consecutivi setați. Pentru că un cuvânt trebuie să aibă cel puțin o literă, nu vom număra și secvența formată numai din zerouri.

Pentru a afla numărul de secvențe de *N* biți în care nu există doi biți consecutivi setați vom folosi programarea dinamică:

Fie *F(n)* = numărul de secvențe de *n* biți care nu au doi biți consecutivi setați și
fie *F0(n)* = numărul de secvențe de *n* biți care nu au doi biți consecutivi setați și se termină cu *0*, respectiv
F1(n) = numărul de secvențe de *n* biți care nu au doi biți consecutivi setați și se termină cu *1*.
Deci, *F(n) = F0(n) + F1(n)* [relația 1].

Dar, *F1(n) = F0(n-1)* [relația 2], pentru că dacă secvența de n biți se termină cu un *1*, înseamnă că înaintea lui era un *0*.
La fel, *F0(n) = F0(n-1)+F1(n-1)*, pentru că dacă secvența de *n* biți se termină cu un *0*, atunci secvența de *n-1* biți se putea termina fie cu *0*, fie cu *1*. Avem așadar că: *F0(n) = F0(n-1)+F1(n-1) = F(n-1)* [relația 3].

Din relațiile 1, 2 și 3 rezultă că *F(n) = F0(n) + F1(n) = F(n-1) + F0(n-1) = F(n-1) +F(n-2)*.

Avem că:
F(n)=F(n-1)+F(n-2), n>=3
F(1)=2 (secvențe de lungime 1: 0, 1)
F(2)=3 (secvențe de lungime 2: 00, 01, 10)
rezultă că răspunsul la problemă este *F(N)-1*, unde *N* este ordinul de mărime al alfabetului, iar *F(N)* este termenul al *N*-lea termen din șirul Fibonacci: 2, 3, 5, 8, 13....

Pentru a determina valoarea acestuia în timp logaritmic vom folosi o matrice Fibonacci și exponețierea rapidă de matrici (https://www.pbinfo.ro/articole/19411/matrice-fibonacci).

Complexitate spațială: O(1)
Complexitate temporală: O(log2(N))

O soluție liniară ar obține maxim 40 de puncte.

Soluție

```
#include <fstream>
using namespace std;

ifstream in("limbajformal.in");
ofstream out("limbajformal.out");

const int MOD = 1000000009;

void copiere(int A[][2], int B[][2])
{
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            A[i][j] = B[i][j];
}

void inmultire(int a[][2], int b[][2])
{
    int c[2][2];
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
        {
            c[i][j] = 0;
            for (int k = 0; k < 2; ++k)
                c[i][j] = (c[i][j] + (long long)a[i][k] * b[k][j] % MOD) % MOD;
        }
    copiere (a, c);
}

void exp(int x[][2], int n)
{
    int rez[][2] = { {1,1},{1,0}};
    copiere(x, rez);
    for (int p = 1; p <= n; p <<= 1)
    {
        if (n & p)
            inmultire(rez, x);
        inmultire(x, x);
    }
    copiere(x, rez);
}

int main()
{
    int Q, x[2][2];
    in >> Q;

    exp(x, Q);

    if (x[0][0] == 0)
        out << MOD - 1;
    else
        out << x[0][0] - 1;

    return 0;
}
```

Problema 3. Meeting

Propunător: Alexandru Lorintz

Cuvinte cheie: Căutare binară, Arbori indexați binar, Șmenul lui Batog (Square Root Decomposition), Arbori de intervale, Aflarea medianului

Indicații:

În contextul problemei, având la dipoziție posibilitățile date pentru mutarea dintr-un punct în altul, putem defini distanța minimă dintre două puncte de coordonate x_1, y_1, respectiv x_2, y_2 drept $|x_2 - x_1| + |y_2 - y_1|$, adică exact Distanța Manhattan. Astfel, cerința este să se găsească **valoarea minimă a sumei distanțelor Manhattan** de la un interval de puncte din cele date până la un anumit punct care trebuie fixat optim, astfel încât să se obțină această sumă minimă.

Având un punct fixat, tot ce rămâne de făcut este calcularea sumei distanțelor de la restul punctelor la acesta pe baza formulei menționată anterior. Astfel, pentru fiecare interogare se pot încerca toate punctele cu coordonate în intervalul **[1, N]** (este evident că doar acestea sunt relevante din restricțiile problemei) și se poate calcula suma distanțelor, aflând suma minimă. Complexitatea acestei solutii este $O(Q \cdot N^3)$ și obține doar primele **5** puncte, fiind foarte ineficientă.

Soluția anterioară poate fi îmbunătățită dacă se găsește o modalitate mai eficientă pentru calcularea sumei distanțelor de la intervalul de puncte dat până la un punct fixat. Pentru aceasta trebuie să introducem întâi una dintre observațiile importante pentru rezolvarea problemei în continuare și anume faptul că suma se poate afla independent pentru cele două coordonate, deoarece acestea nu se influențează una pe cealaltă. Astfel, am redus problema de la una cu două dimensiuni la o singură dimensiune. Putem deci fixa independent coordonatele **x** și **y** și calcula diferențele în modul corespunzătoare. Această abordare are complexitatea $O(Q \cdot N^2)$ și obține **15** puncte.

Pentru a îmbunătăți în continuare soluția, se poate eficientiza și mai mult modalitatea de calculare a sumei pentru **x** sau **y** fixate independent. Observăm că în cazul problemei unidimensionale la care am ajuns, dacă avem un număr **i** fixat, toate valorile **j ≤ i** contribuie la sumă cu **i - j**, iar valorile **j > i** contribuie, în mod simetric, cu **j - i** (lucru ce reiese evident din explicitarea modulului diferenței a două numere). Astfel, dacă avem valorile sortate, putem calcula eficient suma acestor valori folosind sume parțiale sau chiar mai ușor cu doar două variabile pentru sumele numerelor mai mici sau egale, respectiv mai mari dacă fixăm valorile în ordine crescătoare. Această abordare poate avea complexitatea $O(Q \cdot N)$ sau $O(Q \cdot N \cdot log(N))$ în funcție de metoda de sortare aleasă pentru valorile din intervalul de interogare (numerele fiind mici, se poate folosi un algoritm de sortare liniar, de exemplu **Counting Sort**). Astfel, abordarea care folosește sortarea liniară obține cu siguranță **40** de puncte, iar cea care folosește o metodă de sortare de complexitate $O(N \cdot log(N))$ poate, în funcție de implementare, să obțină între **25** și **40** de puncte.

Pentru a ne apropia de soluția completă trebuie făcută observația crucială în vederea rezolvării problemei: dacă considerăm problema unidimensională la care am ajuns, numerele pentru care se obține suma minimă se află în intervalul determinat de cele două numere aflate "la mijlocul" șirul dacă acesta este ordonat crescător sau există un unic număr dacă șirul are număr impar de elemente, având astfel un unic "mijloc" sau **element median**. Demonstrația acestui fapt nu este complicată, considerând ce se întâmplă dacă "mutăm" numărul din mijloc cu o unitate "la stânga" sau "la dreapta". Se observă că dacă se iese din **intervalul median**, suma crește (intuitiv, numărul "se îndepărtează" de un număr strict mai mare de puncte decât de numărul de numere de care se apropie). Astfel, am redus problema completă la a afla **medianul / medianele** mai multor intervale de

numere date dintr-un șir inițial dat (fără a suferi modificări) și la aflarea sumei respective. Ne vom concentra inițial pe aflarea **elementelor mediane**, calculând mai târziu suma cerută, nefiind un lucru atât de complicat. Desigur că o abordare de aflare a **medianului** este folosind sortarea sau algoritmul **Quickselect** în timp liniar (care are ca alternativă funcția nth_element din header-ul **<algorithm>**), dar aceasta se suprapune din punct de vedere al complexității cu metodele care rezolvă subtask-urile anterioare. Un nou subtask pe care îl putem totuși rezolva având această noua observație la îndemână este al cincilea ($\mathbf{1} \leq \boldsymbol{N} < \mathbf{2^{11}}$ și $\mathbf{1} \leq \boldsymbol{Q} < \mathbf{2^{17}}$). Acest subtask permite precalcularea rezultatului de la fiecare cerință pentru fiecare interval al șirului inițial, răspunzând apoi la interogări în timp constant. Un fapt care va fi de folos mai departe este că aflarea **medianului** dintr-un șir sortat este un caz particular al problemei determinării celui de-al K-lea cel mai mic element din același sir. Astfel, acest subtask se poate rezolva în $\boldsymbol{O(N^2 \cdot \log(N) + Q)}$ folosind o structură de date care să permită operații de update (pentru când se adaugă un nou element la interval) și query pentru aflarea celui de-al K-lea element, cea mai simplă fiind un **arbore indexat binar** pe valorile numerelor (acestea sunt deja "mici", nefiind necesară normalizarea lor), în care se va **căuta binar medianul** (căutarea se face **direct în arbore**). Pentru aflarea sumei cerute se poate "sparge" fiecare interval într-o diferență de prefixe (similar cu sumele parțiale) și afla costul la fiecare prefix în $\boldsymbol{O(N)}$ (la fiecare prefix vom obține o problemă similară pentru calcularea sumei cu cea de la subtask-ul rezolvat anterior, unde am fixat valorile de interogare în ordine crescătoare, folosind o metodă de sortare a acestora eficientă, ceea ce trebuie făcut și acum), complexitatea doar pentru aflarea sumei fiind astfel $\boldsymbol{O(N^2)}$. Posibil să existe și o soluție de complexitate $\boldsymbol{O(N^2 + Q)}$ pentru întregul subtask, dar autorul problemei nu a reușit *încă* să o găsească ☺.

Am ajuns (în sfârșit) la subtask-urile care necesită o rezolvare mai generală a problemei. Trebuie găsită o modalitate eficientă pentru aflarea celui de-al K-lea cel mai mic element dintr-un interval dat într-un șir. O abordare ar putea fi folosind o **căutare binară** care să se ajute de un **Merge Sort Tree**, prin care să se poată returna câte elemente sunt mai mici sau egale decât un element dat (de căutarea binară). Un **Merge Sort Tree** este un simplu arbore de intervale, dar care ține în fiecare nod un vector ce conține toate valorile intervalului corespunzător nodului respectiv sortate. Construcția sa se poate realiza aproape identic cu cea a unui arbore de intervale simplu, pornind de la frunze și pentru fiecare nod **interclasând** vectorii corespunzători celor doi fii ai săi (de aici cuvântul **Merge** din denumirea sa, care se referă la etapa de interclasare). Această abordare are complexitatea $\boldsymbol{O(N \cdot \log(N) + Q \cdot log^3(N))}$ (Complexitatea pentru construcția arborelui este $\boldsymbol{O(N \cdot \log(N))}$, iar la interogări un factor de **log** vine de la căutarea binară și restul de la operația de query necesară pe arborele construit, care sparge fiecare interval de query în $\boldsymbol{O(\log)}$ intervale din arbore, iar în fiecare din acestea face la rândul ei o căutare binară în vectorul corespunzător, pentru a afla numărul de elemente cerut din acel interval). De asemenea, este de menționat că această soluție are complexitatea spațială $\boldsymbol{O(N \cdot \log(N))}$. Pentru aflarea sumei vom discuta o metodă la subtask-urile următoare, această fiind o problemă diferită și puțin mai simplă față de cea a aflării medianului.

O altă abordare care obține un punctaj bun este folosirea Algoritmului lui MO împreună cu un **arbore indexat binar** ca în soluțiile anterioare pentru a putea insera și elimina elemente din intervalul curent la fiecare pas al algoritmului și pentru a putea afla elementul dorit, printr-o **căutare binară**, care poate fi realizată **direct pe arbore**. Această abordare are complexitatea $\boldsymbol{O((N+Q) \cdot \sqrt{N} \cdot \log(N))}$. O optimizare foarte interesantă a acestei abordări care coboară complexitatea la $\boldsymbol{O((N+Q) \cdot \sqrt{N})}$ se poate observa dacă se realizează o comparație între numărul de update-uri necesare la **Algoritmul lui MO**, respectiv numărul de query-uri din problemă. Se observă că ordinul numărului de update-uri este $\boldsymbol{O((N+Q) \cdot \sqrt{N})}$, iar query-uri sunt exact **Q**, adică $\boldsymbol{O(N)}$. Astfel, ne permitem să facem operația query puțin mai ineficientă decât cea de update. Folosind, de exemplu, **Square Root Decomposition** pentru a realiza aceleași operații dorite și înainte când foloseam o structură de date arborescentă, se obține $\boldsymbol{O(1)}$ complexitatea pentru o operație de update și $\boldsymbol{O(\sqrt{N})}$ pentru o operație de query, ajungând astfel la complexitatea dorită, $\boldsymbol{O((N+Q) \cdot \sqrt{N})}$, care ar trebui să obțină majoritatea punctelor acordate problemei, poate chiar toate, în funcție de implementare. Optimizarea găsită

este descrisă mai pe larg în articolul acesta. Pentru a afla și suma cerută, este nevoie de o structură de date care să suporte update-uri de inserare/eliminare a unui element (pentru a putea fi modificată corespunzător pe parcursul executării **Algoritmului lui MO**) și query de sumă a elementelor din structură mai mici sau egale decât o valoare dată (în cazul nostru, cea a medianului). Astfel, se poate folosi un **arbore indexat binar** pentru o soluție de complexitate $\boldsymbol{O((N+Q)\cdot\sqrt{N}\cdot\log(N))}$, respectiv **Square Root Decomposition** pentru a obține cea mai bună complexitate, $\boldsymbol{O((N+Q)\cdot\sqrt{N})}$.

O abordare care obține sigur punctajul maxim folosește puțină intuiție de la soluțiile anterioare unde am observat că am putea **căuta binar medianul**. Totuși, observăm că nu ne permitem să căutăm binar medianul la fiecare interogare, pentru că verificarea va fi apoi ineficientă. Astfel, de aici vine ideea Căutării binare în paralel a rezultatelor pentru interogări. Partea de verificare a rezultatului în căutarea binară se va realiza folosind un **arbore indexat binar**, similar cu soluțiile anterioare, dar de data aceasta se vor interclasa query-urile și update-urile create print sortarea șirului cu care se lucrează. Astfel, se obține complexitatea $\boldsymbol{O((N+Q)\cdot log^2(N))}$, cu care se poate obține punctajul maxim. Pentru calcularea sumei, se poate proceda în diverse moduri, dar cel mai simplu este "spargerea" acesteia într-o diferență de prefixe odată ce se știe medianul (similar cu sumele parțiale) și aflarea costului la fiecare prefix folosind un **arbore indexat binar**, similar din punct de vedere al operațiilor suportate cu structurile folosite la aflarea sumei cerute în soluțiile care folosesc **Algoritmul lui MO**.

Soluție alternativă: Pentru a afla elementele mediane din interogări, soluția de complexitate $\boldsymbol{O(N\cdot\log(N)+Q\cdot log^3(N))}$ care folosește un **Merge Sort Tree** poate fi optimizată la $\boldsymbol{O(N\cdot\log(N)+Q\cdot log^2(N))}$ folosind Fractional Cascading.

Câteva note de final:

- Problema are o soluție de complexitate și mai bună decât cea descrisă, fiind o aplicație directă a unei structuri de date de curând descoperită, numită Wavelet Tree, cu care se obține complexitatea $\boldsymbol{O((N+Q)\cdot log(N))}$, dar desigur că aceasta nu era necesară pentru obținerea punctajului maxim la problemă. Singurul dezavantaj minor al acestei structuri de date față în comparație cu soluțiile propuse este că are și complexitatea spațială $\boldsymbol{O(N\cdot log(N))}$, față de $\boldsymbol{O(N)}$ cum are soluția oficială, dar acest lucru este nesemnificativ și nu conta oricum pentru că limita de memorie este destul de îngăduitoare și pentru această eventuală soluție.
- Este de remarcat faptul că soluția oficială nu este foarte complexă din punct de vedere teoretic la final, folosindu-se doar de un **arbore indexat binar**, una dintre cele mai simple structuri de date arborescente, mai ales din perspectiva codului scris.
- De asemenea, se remarcă cât de utilă poate fi **tehnica căutării binare în paralel** la anumite probleme, o tehnică de reținut pe mai departe din această problemă.
- **Bonus**: Puteți afla și pentru câte puncte se obține suma minimă cerută?

Soluție

```
#include <bits/stdc++.h>

using namespace std;

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

const int kN = 1 << 17;
int aib[kN];
int64_t aibSum[kN];
vector<pair<int, int>> queries[kN];

void updateOne(int x, int n) {
  for (int i = x; i <= n; i += i & -i) {
    aib[i] += 1;
  }
}

int queryOne(int x) {
  int res = 0;

  for (int i = x; i > 0; i = i & (i - 1)) {
    res += aib[i];
  }

  return res;
}

void updateBoth(int x, int n) {
  for (int i = x; i <= n; i += i & -i) {
    aib[i] += 1;
    aibSum[i] += x;
  }
}

pair<int, int64_t> queryBoth(int x) {
  int res = 0;
  int64_t sum = 0;

  for (int i = x; i > 0; i = i & (i - 1)) {
    res += aib[i];
    sum += aibSum[i];
  }

  return make_pair(res, sum);
}

int main() {
  int n;
  fin >> n;

  vector<int> xs(n), ys(n);

  for (int i = 0; i < n; ++i) {
    fin >> xs[i] >> ys[i];
  }

  int q;
```

```
  fin >> q;

  vector<pair<int, int>> intervals(q);

  for (auto &it : intervals) {
    fin >> it.first >> it.second;
  }

  vector<pair<int, int>> a(n), sol(q);
  vector<int64_t> fullSolution(q);

  int step = 0;

  while ((1 << (step + 1)) <= n) {
    step += 1;
  }

  for (const auto &vec : {xs, ys}) {
    for (int i = 0; i < n; ++i) {
      a[i] = make_pair(vec[i], i + 1);
    }

    sort(a.begin(), a.end());

    for (int i = 0; i < q; ++i) {
      sol[i] = make_pair(0, i + 1);
    }

    for (int i = 1; i <= n; ++i) {
      aib[i] = 0;
    }

    for (int i = step; i >= 0; --i) {
      for (int j = 1; j <= n; ++j) {
        aib[j] = 0;
      }

      int ptr = 0;

      for (auto &it : sol) {
        while (ptr < n && a[ptr].first <= (it.first | (1 << i))) {
          updateOne(a[ptr].second, n);
          ptr += 1;
        }

        int l, r;
        tie(l, r) = intervals[it.second - 1];

        int med = (r - l + 2) / 2;

        if (queryOne(r) - queryOne(l - 1) < med) {
          it.first |= 1 << i;
        }
      }

      sort(sol.begin(), sol.end());
    }
    for (int i = 1; i <= n; ++i) {
      queries[i].clear();
```

```
    }

    for (auto &it : sol) {
      it.first += 1;
      queries[intervals[it.second - 1].first - 1].emplace_back(it.first, -it.second);
      queries[intervals[it.second - 1].second].emplace_back(it.first, it.second);
    }

    int prefLen = 0;
    int64_t prefSum = 0;

    for (int i = 1; i <= n; ++i) {
      aib[i] = 0;
      aibSum[i] = 0;
    }

    auto solve = [&](const int &x) -> int64_t {
      int low;
      int64_t lowSum;
      tie(low, lowSum) = queryBoth(x);

      int high = prefLen - low;
      int64_t highSum = prefSum - lowSum;

      return (int64_t)low * x - lowSum + highSum - (int64_t)high * x;
    };

    for (int i = 1; i <= n; ++i) {
      prefLen += 1;
      prefSum += vec[i - 1];

      updateBoth(vec[i - 1], n);

      for (const auto &it : queries[i]) {
        int value, index;
        tie(value, index) = it;

        int sgn = 1;

        if (index < 0) {
          sgn = -1;
          index *= -1;
        }

        fullSolution[index - 1] += solve(value) * sgn;
      }
    }
  }

  for (const int64_t &it : fullSolution) {
    fout << it << '\n';
  }

  fin.close();  fout.close();
  return 0;
}
```