Atena, septembrie 2004

Olimpiada INTERNAȚIONALĂ de Informatică 2004

Vă prezentăm în continuare soluțiile oficiale ale celor șase probleme propuse spre rezolvare la ediția din acest an a Olimpiadei Internaționale de Informatică.

P040601: Artemis

Vom nota prin $f(x, y)$ numărul copacilor aflați la stânga și sub punctul de coordonate (x, y).

Considerăm două puncte de coordonate (x_1, y_1) și (x_2, y_2). În cazul în care unul dintre puncte se află la stânga și sub celălalt punct, atunci numărul copacilor aflat în dreptunghiul determinat de cele două puncte este dat de formula:

$$f(x_1, y_1) + f(x_2, y_2) - f(x_1, y_2) - f(y_1, x_2) + 1.$$

În cealalată situație (nici unul dintre puncte nu se află la stânga și sub celălalt punct), formula este similară.

Algoritmul #1

Algoritmul trivial de rezolvare a problemei constă în considerarea tuturor dreptunghiurilor și numărarea copacilor aflați în interiorul acestora.

Ordinul de complexitate al acestui algoritm este $O(N^3)$.

Algoritmul #2

Un al doilea algoritm constă în utilizarea metodei *programării dinamice* pentru a calcula toate valorile $f(x_i, y_j)$ posibile. După determinarea tuturor acestor valori, pe baza formulelor se determină numărul copacilor din fiecare dreptunghi.

Ordinul de complexitate al acestui algoritm este $O(N^2)$, dar și spațiul de memorie utilizat are tot ordinul de complexitate $O(N^2)$.

Algoritmul #3

Cel de-al treilea algoritm pe care îl prezentăm constă în parcurgerea tuturor copacilor și determinarea, pentru fiecare copac, a dreptunghiurilor care au colțul din stânga-sus în poziția respectivă.

Dacă acest copac se află în poziția (x, y), atunci avem nevoie numai de valorile de forma $f(x, \cdot)$ și $f(\cdot, y)$.

Acestea pot fi determina la fel ca și în cazul algoritmului anterior, folosind metoda *programării dinamice*.

Principalul avantaj este faptul că ordinul de complexitate al spațiului de memorie este $O(N)$, chiar dacă ordinul de complexitate al algoritmului este tot $O(N^2)$.

Algoritmul #4

Cel de-al patrulea algoritm constă în sortarea copacilor în funcție de coordonata orizontală.

Copacii sunt parcurși în această ordine și fiecare copac este inserat în lista copacilor curenți care este sortată în funcție de coordonata verticală.

Dacă punctul în care se află copacul considerat are coordonatele (x_0, y_0), atunci pot fi calculate în timp liniar toate valorile de forma $f(x, y_0)$ și $f(x_0, y)$, unde (x, y) reprezintă coordonatele unui punct în care se află un copac situat în stânga copacului considerat.

Așadar, putem determina numărul copacilor din fiecare dreptunghi care au un colț în dreptul copacului considerat.

Până la urmă, se dovedește că algoritmul este foarte asemănător cu cel anterior, iar ordinul său de complexitate este tot $O(N^2)$.

Algoritmul #5

Ultimul algoritm pe care îl prezentăm este o versiune optimizată a primului algoritm.

În prima fază a algoritmului (preprocesare) vom atașa fiecărui copac o listă de biți (aceasta va conține N biți) în care vom marca toți copacii care se află la stânga și sub copacul considerat.

Copacii aflați în interiorul unui dreptunghi pot fi determinați folosindu-se o conjuncție logică între listele de biți corespunzătoare celor doi copaci care determină triunghiul.

Pentru a mări viteza vom folosi o tabelă care va conține, pentru toate numerele care pot fi reprezentate pe 16 biți, numărul biților cu valoarea 1.

Algoritmul prezentat are ordinul de complexitate $O(N^3)$, iar spațiul de memorie necesar este $O(N^2)$, însă factorii constanți care se ascund în spatele acestor notații sunt mici.

P040602: Hermes

Notăm cu (x_i, y_i) punctele în care se află zeii care trebuie să primească mesajul (i variază între 1 și N). Vom avea $(x_0, y_0) = (0, 0)$ pentru poziția inițială a lui *Hermes*.

Vom calcula valorile A_{ij} și B_{ij}, unde A_{ij} reprezintă distanța pe care trebuie să o parcurgă *Hermes* pentru a trimite mesaje primilor i zei și a ajunge în poziția (x_i, y_j), iar B_{ij} reprezintă distanța pe care trebuie să o parcurgă *Hermes* pentru a trimite mesaje primilor i zei și a ajunge în poziția (x_j, y_i). Formulele care stau la baza calculării acestor valori sunt următoarele:

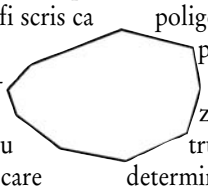
$$A_{i+1,j} = min(A_{i,j} + |x_i - x_{i+1}|, B_{i,i+1} + |y_i - y_j|)$$
$$B_{i+1,j} = min(B_{i,j} + |y_i - y_{i+1}|, A_{i,i+1} + |x_i - x_j|)$$

Rezultatul final va fi ales ca fiind minimul valorilor de forma A_{Nj} sau B_{jN}, unde j variază între 0 și N.

Ordinul de complexitate al acestui algoritm este $O(N^2)$.

P040603: Poligon

Chiar dacă determinarea sumei *Minkovski* a două poligoane este o operație relativ simplă ([2]), deciderea faptului dacă un poligon poate fi scris ca suma *Minkovski* a două poligoane este o problemă *NP-completă*.



Există un algoritm pentru rezolvarea acestei probleme care rulează în timp pseudopolinomial și care a fost prezentat în [1]; acest algoritm va fi descris pe scurt în cele ce urmează.

Dacă intrarea constă într-o succesiune de N laturi, atunci dimensiunea intrării este considerată a fi $O(N \cdot (\log m + \log E)$, unde m reprezintă numărul maxim de puncte cu coordonate întregi de pe o muchie, iar E reprezintă valoarea maximă absolută a unei coordonate care definește un vector primar (definiția vectorului primar va fi prezentată mai jos).

Algoritmul prezentat în [1] are ordinul de complexitate $O(t \cdot N \cdot m)$, unde numărul punctelor poligonului care au coordonate întregi.

Vom nota poligonul dat prin P și vom considera că el conține N vârfuri care au toate coordonate întregi nenegative de forma $v_i = (x_i, y_i)$, unde i variază între 0 și N - 1. Vârfurile poligonului sunt date în ordine trigonometrică.

Definiție

Vom numi un vector $v = (a, b)$ vector ***primar*** dacă cel mai mare divizor comun al valorilor a și b este 1, unde a și b sunt numere întregi nenegative.

Cu alte cuvinte, spunem că un vector v este vector primar dacă și numai dacă el nu conține în interior nici un punct care are ambele coordonate întregi.

Putem spune că laturile poligonului P sunt date de vectori de forma $E_i = v_i - v_{i-1} = (a_i, b_i)$, unde i variază între 1 și N, indicii sunt considerați modulo N, iar valorile a_i și b_i sunt numere întregi.

Pentru o latură (a_i, b_i), dacă cel mai mare divizor comun n_i al valorilor a_i și b_i nu este egal cu 1, atunci vom considera latura primară corespunzătoare $e_i = (a_i / n_i, b_i / n_i)$.

Vom numi secvența de vectori de forma $E_i = e_i \cdot n_i$ ***secvența de laturi*** a poligonului, unde e_i este un vector primar.

Următoarea lemă reprezintă o observație crucială pentru găsirea unei modalități de determinare a două poligoane A și B, astfel încât suma *Minkovski* a acestora să fie un poligon dat P.

Lemă

Fiecare latură primară a poligonului P trebuie să apară ca latură fie în A, fie în B. În cazul în care latura nu este primară, atunci există o a treia posibilitate și anume ca latura să reprezinte suma *Minkovski* a unei laturi din A și a unei laturi din B care sunt paralele.

Pe baza acestei leme putem deduce că un poligon poate fi un termen a unei sume *Minkovski* care duce la obținerea poligonului P dacă și numai dacă secvența sa de laturi este formată din vectori de forma $k_j \cdot e_j$, unde valorile j sunt cuprinse între 1 și N (evident, unele valori pot lipsi), valorile k_j sunt numere întregi cuprinse între 0 și n_j, iar suma vectorială a vectorilor care reprezintă laturile poligonului este 0.

În cele ce urmează vom descrie algoritmii care pot fi utilizați pentru determinarea termenilor de forma celor ceruți în enunțul problemei (segmente, triunghiuri și patrulatere).

Poligoanele obținute pot avea vârfuri cu coordonate negative. În acest caz va trebui să translatăm ambii termeni obținuți astfel încât să nu mai existe coordonate negative. Această translatare este permisă deoarece nu duce la modificarea formei poligoanelor.

Segmente

Putem spune că suma *Minkovski* care duce la obținerea poligonului P poate conține un termen care este segment dacă și numai dacă cel puțin o pereche de (sub)vectori din secvența de laturi a poligonului au suma egală cu 0.

Pe baza acestei observații putem construi algritmii care sunt prezentați în cele de urmează.

Fiecare algoritm începe cu determinarea vectorilor primari pentru fiecare latură folosind în acest scop un algoritm de determinare a celui mai mare divizor comun.

Evident, acest pas nu este necesar pentru poligoanele care conțin doar laturi primare.

Algoritmul banal

Cel mai simplu dintre algoritmi (și cel mai lent) constă în determinarea muchiilor poligonului ca diferențe a câte doi vectori și apoi determinarea vectorilor primari folosind un algoritm de determinare a celui mai mare divizor comun.

Ordinul de complexitate al acestui pas este $O(N \cdot m)$, unde m este cea mai mare valoare obținută pentru un cel mai mare divizor comun calculat.

Pentru fiecare pereche de vectori din secvența de muchii vom calcula suma acestora, operație al cărei ordin de complexitate este $O(N^2 \cdot m^2)$.

Un algoritm mai performant

Pentru cel de-al doilea algoritm, vom începe tot cu determinarea muchiilor poligonului, folosind același procedeu al cărui ordin de complexitate este $O(N \cdot m)$.

În continuare sortăm muchiile în funcție de componenta orizontală a vectorilor, operație al cărui ordin de complexitate este $O(N \cdot m \cdot \log(N \cdot m))$.

Acum, pentru fiecare vector, verificăm dacă există un vector a cărui componentă orizontală este inversul componentei orizontale a vectorului considerat și dacă rezultatul însumării celor doi vectori este 0.

Căutarea se va realiza în timp logaritmic ($O(\log(N \cdot m))$ folosind metoda căutării binare, deoarece laturile sunt deja sortate.

Întregul algoritm are ordinul de complexitate $O(N \cdot m \cdot \log(N \cdot m))$.

Algoritmul eficient

Cel de-al treilea algoritm începe tot cu determinarea muchiilor, operație al cărei ordin de complexitate este $O(N \cdot m)$.

Laturile sunt apoi inserate într-o tabelă de dispersie; componentele orizontale ale vectorilor au rolul cheilor în tabela de dispersie. Inserarea se realizează în timp constant.

La fel ca și în cazul algoritmului anterior, pentru fiecare vector verificăm dacă există un vector a cărui componentă orizontală este inversul componentei orizontale a vectorului considerat și dacă rezultatul însumării celor doi vectori este 0.

Această căutare se realizează în timp constant, motiv pentru care algoritmul descris are ordinul de complexitate $O(N \cdot m)$.

Triunghiuri

Similar situației pe care am întâlnit-o în cazul segmentelor, putem spune că suma *Minkovski* care duce la obținerea poligonului P poate conține un termen care este triunghi dacă și numai dacă pot fi aleși trei vectori din secvența de laturi a poligonului care au suma egală cu 0.

La fel ca și în cazul segmentelor, vom prezenta trei algoritmi care determină astfel de triplete.

Algoritmul banal

Primul dintre algoritmi constă în determinarea laturilor poligonului (la fel ca în cazul algoritmilor pentru segmente) și calcularea, pentru fiecare triplet de vectori din secvența de muchii, a sumei acestora.

Acest algoritm are ordinul de complexitate $O(N^3 \cdot m^3)$.

Un algoritm mai performant

Cel de-al doilea algoritm începe tot cu determinarea laturilor poligonului.

Muchiile sunt apoi sortate în funcție de componentele orizontale ale vectorilor corespunzători.

Pentru fiecare pereche de vectori vom calcula suma și apoi, cu ajutorul căutării binare, vom verifica dacă există un al treilea vector astfel încât suma celor trei vectori să fie 0.

Ordinul de complexitate al acestui algoritm este $O(N^2 \cdot m^2 \cdot \log(N \cdot m))$.

Algoritmul eficient

Pentru cel de-al treilea algoritm vom determina laturile poligonului și le vom sorta în funcție de componenta orizontală.

Pentru fiecare muchie k a secvenței, vom parcurge secvența dinspre stânga pentru a determina o valoare i și dinspre dreapta pentru a determina a valoare j, astfel încât să obținem $E_i + E_k + E_j = 0$.

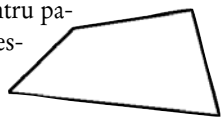
Datorită faptului că laturile au fost sortate, la fiecare pas vom modifica fie valoarea i, fie valoarea j, în funcție de semnul rezultatului.

Ordinul de complexitate al algoritmului este $O(N^2 \cdot m^2)$.

O altă posibilitate o constituie inserarea vectorilor într-o tabelă de dispersie și căutarea, pentru fiecare pereche de vectori, a unui al treilea vector care să fie inversul sumei primilor doi.

Datorită faptului că o astfel de căutare se realizează în timp constant, obținem tot ordinul de complexitate $O(N^2 \cdot m^2)$.

Patrulatere



La fel ca și în cazul segmentelor și triunghiurilor, pentru patrulatere vom descrie tot trei algoritmi.

Evident, putem spune și de această dată că suma *Minkovski* care duce la obținerea poligonului P poate conține un termen care este patrulater dacă și numai dacă pot fi aleși patru vectori din secvența de laturi a poligonului care au suma egală cu 0.

Algoritmul banal

Primul algoritm este foarte simplu și constă în determinarea laturilor poligonului și calcularea sumelor pentru oricare patru vectori, operație al cărei ordin de complexitate este $O(N^4 \cdot m^4)$.

Un algoritm mai performant

Pentru cel de-al doilea algoritm vom calcula, după ce am determinat laturile, sumele tuturor tripletelor de vectori posibile și apoi, vom căuta un al patrulea vector astfel încât suma celor patru vectori să fie 0.

Putem fie să ordonăm secvența de laturi în funcție de componenta orizontală, fie să folosim o tabelă de dispersie.

În primul caz ordinul de complexitate este $O(N^3 \cdot m^3 \cdot \log(N \cdot m))$, iar în al doilea caz acesta este $O(N^3 \cdot m^3)$.

Algoritmul eficient

Cel de-al treilea algoritm constă în determinare laturilor și inserarea într-o tabelă de dispersie a tuturor sumelor posibile a doi vectori.

Cheia pentru tabela de dispersie este dată de componenta orizontală a sumei. Operația de inserare a tuturor acestor sume de vectori are ordinul de complexitate $O(N^2 \cdot m^2)$.

Pentru fiecare sumă vom căuta în tabela de dispersie o altă sumă astfel încât suma celor două sume (deci suma celor patru vectori corespunzători) să fie 0.

Datorită faptului că o căutare are nevoie de un timp constant, ordinul de complexitate al întregului algoritm este $O(N^2 \cdot m^2)$.

Cel mai mare divizor comun

Un element important în rezolvarea acestei probleme o constituie determinarea celui mai mic divizor comun a două numere întregi.

Vom prezenta doi algoritmi care realizează această operație; primul dintre aceștia este algoritmul clasic al lui *Euclid*:

```
algoritm cmmdc_Euclid(a, b)
  dacă b = 0 atunci
    retunează a
  altfel
    retunează cmmdc_Euclid(b,
                  rest[a / b])
  sfârșit dacă
sfârșit algoritm
```

Cel de-al doilea algoritm este preluat din [3] și este cunoscut ca algoritmul binar de determinare a celui mai mare divizor comun:

```
algoritm cmmdc_Binar(a, b)
  g ← 1
  cât timp a este par și
              b este par execută
    a ← a / 2 // deplasare spre
              // dreapta
    b ← b / 2
    g ← g * 2 // deplasare spre
              // stânga
  sfârșit cât timp
  t ← |a - b| / 2
  cât timp t > 0 execută
    dacă a este par atunci
      a ← a / 2
    altfel
      dacă b este par atunci
        b ← b / 2
      altfel
        t ← |a - b| / 2
      sfârșit dacă
    sfârșit dacă
    dacă a < b atunci
      b ← t
    altfel
      a ← t
    sfârșit dacă
  sfârșit cât timp
  returnează g * b
sfârșit algoritm
```

Bibliografie

[1] S. Gao, A. Lauder, *Decomposition of polytopes and Polynomials*, Discrete and Computational Geometry 26 (2001), pp. 501-520

[2] L.J. Guibas, R. Seidel, *Computing Convolutions by Reciprocal Search*, Symposium of Computational Geometry, 2001

[3] D. Knuth, *The Art of Computer Programming*, vol. II, Addison-Wesley, ediția a III-a, 1998

P040604: Empodii

Potrivit [1], permutările sunt importante în studierea secvențelor *ADN*. Următoarele două propoziții sunt adevărate pentru oricare interval mărginit:

Propoziția #1

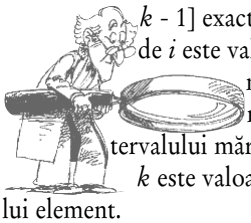
Între punctul de început (F_s) și punctul de sfârșit (F_f) ale unui interval mărginit F există următoarea legătură:

$val(F_s) = val(F_s) + lungime(F) - 1$,

unde prin $val(i)$ am notat valoarea celui de-al i-lea al secvenței biologice, iar prin $lungime(I)$ am notat numărul de elemente din care este format intervalul I.

Propoziția #2

Toate valorile din interiorul unui interval mărginit sunt distincte și acoperă întregul interval $[i + 1, i + k - 1]$ exact o dată, unde i este valoarea primului element al intervalului mărginit, iar $i + k$ este valoarea ultimului element.



Un algoritm simplu

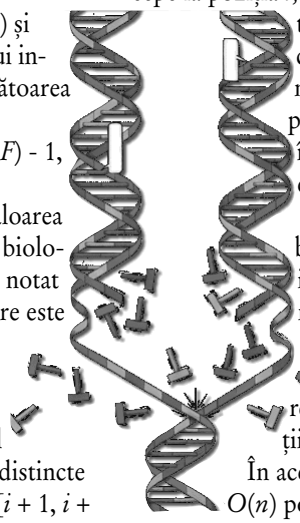
Pentru fiecare element i al secvenței bilogice σ vom căuta intervale mărginite care încep la poziția i și care au lungimea cuprinsă între 2 și $|\sigma| - i + 1$, unde prin $|\sigma|$ am notat lungimea secvenței biologice.

Dacă notăm lungimea considerată cu l, atunci va trebui să verificăm propoziția #1 pentru perechea (i, j), unde $j = i + l - 1$.

Dacă propoziția este respectată, atunci trebuie verificată și propoziția #2. Așadar, va trebui să verificăm dacă toate elementele cuprinse între pozițiile i și j au valori cuprinse între $val(i)$ și $val(j)$.

Avem $O(n^2)$ perechi (i, j) și, pentru fiecare interval pentru care propoziția #1 este adevărată, avem nevoie de un timp de ordinul $O(n)$ pentru a verifica dacă intervalul este mărginit (propoziția #2 este adevărată). Ca urmare, pentru identificarea intervalelor mărginite avem nevoie de un timp de ordinul $O(n^3)$.

Pe parcursul efectuării verificărilor vom păstra posibilele empodii care încep la poziția i. Bineînțeles, nu va trebuie să păstrăm toate cele $O(n)$ intervale mărginite care ar putea începe la poziția i, deoarece toate intervalele mărginite care nu au lungimea minimă nu pot fi empodii (ele vor conține în interior intervalul de lungime minimă).



Mai mult, va trebui să verificăm dacă intervalele mărginte nu conțin în interiorul lor alte intervale mărginite care încep la alte poziții.

În acest moment avem $O(n)$ posibile empodii.

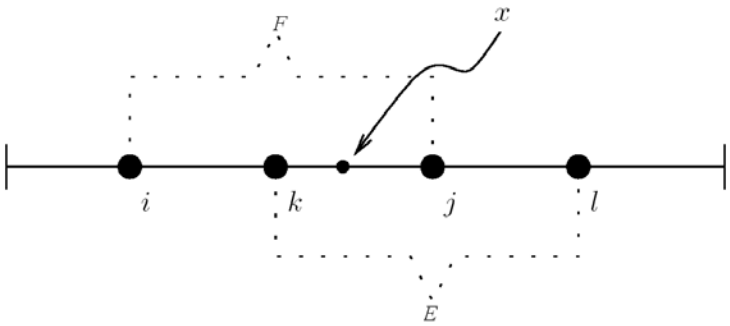


Figura 1

Pentru fiecare dintre acestea putem verifica într-un timp de ordinul $O(n)$ dacă intervalul conține în interiorul său un alt interval mărginit și dacă acest interval mărginit este sau nu un empodiu.

Acest pas al algoritmului are ordinul de complexitate $O(n^2)$, deci întregul algoritm are ordinul de complexitate $O(n^3)$.

Un algoritm performant

Pentru a prezenta acest algoritm vom începe cu enunțarea a două propoziții care sunt adevărate întotdeauna:

Propoziția #3

Într-un empodiu, elementul care are cea mai mică valoare se află întotdeauna pe prima poziție a empodiului considerat.

Propoziția #4

Într-un empodiu, elementul care are cea mai mare valoare se află întotdeauna pe ultima poziție a empodiului considerat.

Prezentăm acum o informație esențială pentru algoritmul pe care îl vom descrie:

Lemă

Nici una dintre extremitățile unui empodiu nu se poate afla în interiorul unui alt empodiu.

Demonstrație

În continuare vom demonstra această lemă folosind metoda reducerii la absurd.

În cazul în care ambele extremități ale unui empodiu E s-ar afla în interiorul unui empodiu F, atunci empodiul F ar conține în interiorul său empodiul E, ceea ce contravine definiției.

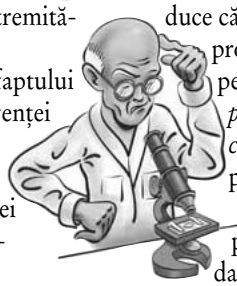
În cazul în care doar una dintre extremitățile unui empodiu E s-ar afla în interiorul unui empodiu F, am avea o situație asemănătoare cu cea din figura 1.

Presupunem, fără a restrânge generalitatea, că extremitatea stângă a empodiului E se află în interiorul empodiului F.

Pe baza propozițiilor #3 și #4 obținem:

$val(i) < val(k) < val(j) < val(l)$,

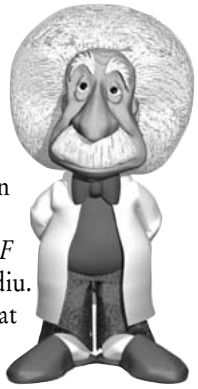
unde i și j sunt extremitățile empodiului F, iar k și l sunt extremitățile empodiului E.



Aceasta se datorează faptului că al k-lea element al secvenței biologice se află în interiorul empodiului F, iar al j-lea element al secvenței se află în interiorul empodiului E.

Alegem la întâmplare un element aflat pe o poziție x situată între pozițiile k și j. Este evident că acest element face parte din ambele empodii considerate.

Ca urmare, avem $val(x) < val(j)$ și $val(k) < val(x)$. Datorită faptului că poziția x a fost aleasă la întâmplare, putem concluziona ca toate valorile situate pe poziții cuprinse între k și j (exclusiv) sunt cuprinse între valorile $val(k)$ și $val(j)$ (exclusiv).



Putem concluziona că intervalul cuprins între pozițiile k și j este un interval mărginit, deci nici E și nici F nu poate fi empodiu.

Am demonstrat astfel că ipoteza este falsă, deci nici o extremitate a unui empodiu nu se poate afla în interiorul unui alt empodiu.

Notă

Se observă imediat că, datorită faptului că elementele aflate între pozițiile k și j (exclusiv) sunt cuprinse între valorile $val(k)$ și $val(j)$ (exclusiv), atunci toate elemente aflate între pozițiile i și k (exclusiv) sunt cuprinse între valorile $val(i)$ și $val(k)$ (exclusiv). Așadar, intervalul cuprins între pozițiile i și k este și el mărginit.

Similar, putem deduce că toate elemente aflate între pozițiile j și l (exclusiv) sunt cuprinse între valorile $val(j)$ și $val(l)$ (exclusiv). Așadar, și intervalul cuprins între pozițiile i și k este mărginit.

Determinarea candidaților

Pe baza notei de mai sus putem deduce că o modalitate de a aborda problema este de a determina, pentru fiecare poziție i, cel mai *potrivit* interval mărginit *candidat* care începe la poziția i.

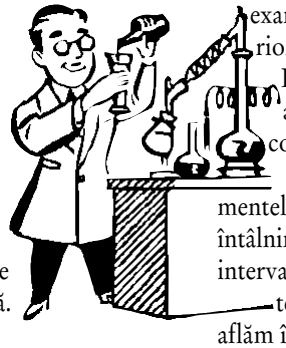
Pentru aceasta vom parcurge elementele secvenței date de la stâng spre dreapta și vom păstra valoarea maximă (max) în fiecare moment.

Dacă la un moment dat întâlnim o valoare egală cu $max + 1$ și, în același timp, propoziția #1 este satisfăcută, atunci putem deduce că am ajuns în dreptul extremității drepte a unui interval mărginit.

Datorită faptului că vom verifica $O(n)$ elemente și, pentru fiecare dintre acestea, vom parcurge $O(n)$ elemente, ordinul de complexitate al operației de determinare a celor mai potrivite intervale mărginite candidate este $O(n^2)$.

În cele ce urmează vom prezenta versiunea în pseudocod a algoritmului descris.

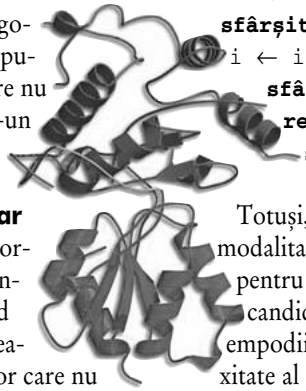
Algoritmul are ca parametru secvența biologică pe care o notăm cu *secv_bio* și considerăm că ea conține $N + 2$ elemente identificate prin numere întregi cuprinse între 0 și $N + 1$.



Rezultatul va fi un vector pe care îl vom nota prin *int_mărg* și care va conține, pentru fiecare poziție *i*, lungimea unui candidat care începe la poziția respectivă.

```
algoritm Candidați(secv_bio)
  pentru i ← 0, N + 1
    execută int_mărg_i ← nil
  sfârșit pentru
  pentru i ← 0, N execută
    max ← secv_bio_i
    pentru k ← 1, N + 1 - i
                          execută
      dacă secv_bio_i+k =
            secv_bio_i + k și
        secv_bio_i+k = max + 1
                          atunci
        int_mărg_i ← k + 1
      altfel
        dacă secv_bio_i+k <
              secv_bio_i atunci
          întrerupe ciclul pentru
        altfel
          dacă secv_bio_i+k >
                  max atunci
            max ← secv_bio_i+k
          sfârșit dacă
        sfârșit dacă
      sfârșit dacă
    sfârșit pentru
  sfârșit pentru
  returnează int_mărg
sfârșit algoritm
```

La fel ca și în cazul algoritmului descris anterior, putem elimina candidații care nu sunt de fapt empodii într-un timp de ordinul $O(n^2)$.



Eliminare în timp liniar

Deși nu se va îmbunătăți ordinul de complexitate al întregului algoritm, folosind lema prezentată, putem realiza eliminarea candidaților care nu sunt empodii în timp liniar.

În vectorul *int_mărg* avem lungimile intervalelor mărginite candidate. Spunem că un interval este *activ* dacă examinăm elemente aflate în interiorul intervalului respectiv.

Ideea care stă la baza algoritmului este prezentată în continuare.

Vom parcurge secvențial elementele vectorului *int_mărg*. Dacă întâlnim extremitatea stângă a unui interval candidat *F* în timp ce un interval *E* este activ, atunci ne aflăm în situația prezentată pentru primul caz al demonstrației lemei. Aceasta se datorează faptului că *E* este intervalul mărginit de lungime minimă care începe la poziția corespunzătoare.

Acest procedeu constă în cel mult $N + 1$ interogări ale valorilor vectorului *int_mărg*. Așadar, ordinul de complexitate al operației este $O(n)$.

Prezentăm în continuare versiunea în pseudocod a acestui algoritm.

```
algoritm Eliminare(int_mărg)
  i ← 0
  cât timp i < N + 1 execută
    dacă int_mărg_i ≠ nil
                          atunci
      k ← i + 1
      cât timp k < int_mărg_i
              + i - 1 execută
        dacă int_mărg_i ≠ nil
                          atunci
          int_mărg_i ← nil
          întrerupe ciclul
                          cât timp
        sfârșit dacă
        k ← k + 1
      sfârșit cât timp
      i ← k - 1
    sfârșit dacă
    i ← i + 1
  sfârșit cât timp
  returnează int_mărg
sfârșit algoritm
```

Totuși, indiferent care este modalitatea pe care o utilizăm pentru a elimina intervalele candidate care nu sunt empodii, ordinul de complexitate al acestui algoritm este $O(n^2)$.

Algoritmul eficient

Această problemă poate fi rezolvată în timp supraliniar $O(n \cdot \alpha(n))$ așa cum se descrie în [2], sau liniar $O(n)$ așa cum se arată în [3]. Aceste rezolvări implică utilizarea unor tehnici ale teoriei grafurilor.

Aceste rezultate provin dintr-unul dintre cele mai interesante domenii al informaticii în ultimul deceniu, ai cărui pionieri au fost *Hannenhalli* și *Pevzner* [4], iar implementările sunt relativ simple.

O prezentare detaliată a acestui domeniu din care a fost inspirată această problemă poate fi găsită în [5].

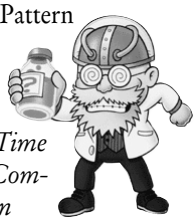
Nu vom reda aici aceste detalii deoarece, pentru problema de la *IOI* 2004, un algoritm cu timp de execuție pătratic era suficient pentru a obține cel puțin 90% din punctaj.

Bibliografie

[1] A. Bergeron, *A Very Elementary Presentation of the Hannenhalli-Pevzner Theory*, CPM 2001 - note de curs, vol. 2089, pp. 106-117, 19 aprilie 2001.

[2] P. Berman, S. Hannenhalli, *Fast Sorting by Reversal*, Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, pp. 168-185, 1996.

[3] Bader, Moret, Yan, *A Linear-Time Algorithm for Computing Inversion*



Distance between Signed Permutations with an Experimental Study - note de curs, WADS: 7th Workshop on Algorithms and Data Structures, vol. 2125, pp. 365-376, 2001.

[4] **S. Hannenhalli, Pavel Pevzner**, *Transforming Cabbage into Turnip*, Proceedings of the 27th Annual Symposium on Theory of Computing (STOC95), pp. 178-189, 1995.

[5] **A. C. Siepel**, *E xact Algorithms for the Reversal Median Problem* - teză de masterat, Universitatea din New Mexico, decembrie 2001.

[6] **A. C. Siepel**, *An Algorithm to Enumerate All Sorting Reversals*, RECOMB, pp. 281-290, aprilie 2002.

P040605: Fermierul

Fie g_i un câmp sau o fâșie, iar n_i numărul chiparoșilor dintr-un câmp sau o fâșie. Dacă notăm prin e_i numărul măslinilor din g, atunci avem $e_i = n_i$ dacă g este un câmp și $e_i = n_i - 1$ dacă g este o fâșie.

În cele ce urmează vom considera următoarea problemă a rucsacului:

determinați $\max \sum_{i=1}^{n+m} e_i \cdot x_i$ astfel încât

$\sum_{i=1}^{n+m} n_i \cdot x_i \le Q$ și x_i poate fi 0 sau 1,

unde n reprezintă numărul grupurilor, iar m reprezintă numărul fâșiilor.

Soluția este dată de o submulțime a elementelor g astfel încât numărul total al chiparoșilor din această submulțime este cel mult egal cu Q, iar numărul măslinilor este cât mai mare posibil.

În general, vom avea

$$Q' = \sum_{i=1}^{n+m} n_i \cdot x_i \le Q .$$

În această situația va exista un element g pentru care avem $x_i = 0$ și $n_i > Q - Q'$ deoarece altfel soluția ar fi fost îmbunătățită prin includerea elementului g în submulțimea aleasă, ceea ce contravine ipotezei potrivit căreia soluția aleasă este cea mai bună.

Ca urmare, prin adăugarea unui șir de $Q - Q'$ chiparoși (împreună cu cei $Q - Q' - 1$ măslini dintre ei) va duce la obținerea soluției optime.

Problema rucsacului poate fi rezolvată eficient folosind metoda programării dinamice. Ordinul de complexitate al algoritmului este $O((m + n) \cdot Q)$.

P040606: Phidias

Pentru rezolvarea acestei probleme vom folosi metoda programării dinamice.

Vom nota prin a_{xy} suprafața pierdută pentru un dreptunghi de lungime x și lățime y, unde x variază între 1 și lungimea dalei, iar y variază între 1 și lățimea dalei.

Inițial, pentru toate plăcile date vom considera valorile corespunzătoare $a_{xy} = 0$ deoarece știm cu siguranță că putem acoperi un astfel de dreptunghi fără pierderi, folosind o placă dată. Pentru toate celelalte valori vom considera $a_{xy} = x \cdot y$.

Pentru fiecare dreptunghi, vom lua în considerare toate posibilitățile de a îl tăia. Vor fi $x - 1$ tăieturi verticale posibili și $y - 1$ tăieturi orizontale.

Pentru o tăietură orizontală c, vom obține două dreptunghiuri de lungime x și lățimi c și $y - c$. În acest caz spațiul irosit ar fi $a_{xc} + a_{x,y-c}$.

Pentru o tăietură verticală c, vom obține două dreptunghiuri de lățime y și lungimi c și $x - c$. Spațiul pierdut în această situație ar fi $a_{cy} + a_{x-c,y}$.

Pentru fiecare dimensiunea vom alege valoarea minimă, iar în final vom afișa valoarea corespunzătoare întregii dale.

Notă

Acestea sunt soluțiile oficiale ale problemelor propuse spre rezolvare la ediția 2004 a Olimpiadei Internaționale de Informatică. Ele au fost preluate de pe site-ul oficial al competiției și au fost traduse de către membrii redacției GInfo.

ATHENS 2004
INTERNATIONAL OLYMPIAD IN INFORMATICS
SEPTEMBER 11-18