ACM

Mihai Stroe

Vă prezentăm în continuare soluțiile problemelor propuse spre rezolvare la concursul internațional de programare ACM, ediția 2003/2004 care a avut loc la București. Aceste soluții au fost realizate de redacția GInfo pe baza soluțiilor oficiale prezentate de către autorii problemelor.

P070301: Codificare delta

Aceasta este o problemă "muncitorească". Dificultatea ei constă în implementarea rapidă, în timpul concursului, a algoritmului (specificat în enunț) și respectarea restricțiilor impuse asupra datelor de intrare și ieșire.

Algoritmul de decriptare se deduce în mod evident din algoritmul de criptare, folosind aritmetica modulo 26. Testarea validității cifrului este foarte simplă.

Analiza complexității

Toate operațiile se pot efectua în timp proporțional cu lungimea datelor asupra cărora se execută.

Ordinul de complexitate a algoritmului este $O(L)$, unde L este lungimea datelor de intrare.

P070302: Parola ascunsă

În continuare vom descrie algoritmul de rezolvare a acestei probleme pentru un test, urmând ca acesta să se aplice de câte ori este nevoie.

Mai întâi se dublează șirul inițial prin concatenare cu el însuși.

Soluția problemei este dată de către un indice X cuprins între 1 și L inclusiv, pentru care subșirul de L caractere al șirului dublat, care începe din poziția X, este primul în ordine lexicografică.

Vom nota cu S_i subșirul de N caractere care începe din poziția i.

Pentru testele date, folosirea funcției `"strncmp"` ar fi dus la o rezolvare eficientă. Aceasta poate compara două subșiruri de L caractere, terminându-se în momentul în care se întâlnește o diferență.

Ordinul de complexitate al acestui algoritm este $O(L^2)$, deci teoretic această variantă nu s-ar fi încadrat în timp (concurenții nu știau numărul de teste etc.). Practic, pe testele date în concurs și eventual folosind câteva euristici suplimentare, această rezolvare ar fi fost acceptată.

Concurentii mai experimentați au preferat rezolvări mai eficiente. Astfel, folosind structuri de date avansate (arbori de sufixe, vectori de sufixe), se pot obține algoritmi de complexitate $O(L \cdot \log^2 L)$ sau $O(L \cdot \log L)$.

Să examinăm o astfel de rezolvare.

Se vor executa $O(\log L)$ pași. După fiecare pas P se va obține o listă de grupe de indici, ordonată în funcție de ordinea șirurilor de lungime 2^P care încep din pozițiile determinate de indicii respectivi. Dacă avem subșiruri identice de lungime 2^P, acestea vor face parte din aceeași grupă. P crește până când o nouă creștere ar duce la depășirea lui L.

Presupunând că putem obține această structură după P pași, să vedem cum o construim pe cea de la pasul $P + 1$.

Pentru a construi structura de la pasul $P + 1$ trebuie să putem compara șiruri care încep cu indici i și j aflați în aceeași grupă la pasul P. Comparația este simplă dacă ținem cont de faptul că un subșir de lungime 2^{P+1} este format din doua jumătăți de lungime 2^P. Deoarece în cazul indicilor aflați în aceeași grupă primele jumătăți sunt egale, trebuie comparată jumătatea a doua a celor două șiruri; pentru aceasta se determină în ce grupă se aflau jumătățile finale, deci indicii $2^P + i$ și $2^P + j$. Această informație poate fi construită pe parcurs, deci comparația are ordinul de complexitate $O(1)$.

Cunoscând aceste detalii, la pasul P+1 putem sorta elementele din fiecare grupă de la pasul P, folosind un algoritm de complexitate $O(NR \cdot \log NR)$, unde NR este numărul de elemente din grupă (maxim L).

Pentru aflarea rezultatului se vor testa indicii din prima grupă. Aceștia determină șiruri egale de lungime 2^P. Pentru a compara șiruri de lungime N se testează ultimele 2^P caractere din aceste șiruri; aceasta se realizează folosind indici de forma $i + (L - 2^P)$.

Analiza complexității

În continuare vom calcula ordinul de complexitate al algoritmului eficient prezentat.

Avem în total $P = \log L$ pași, iar la fiecare pas se realizează câte o sortare în fiecare grupă. Cazul cel mai defavorabil se obține când una din grupe este mare în comparație cu celelalte. în acest caz, ordinul de complexitate al algoritmului este $O(L \cdot \log L)$.

În concluzie, ordinul total de complexitate pentru fiecare set de date este $O(L \cdot \log^2 L)$.

P070303: Petrecerea nebunatică

În continuare vom rezolva această problemă pentru un singur test, urmând ca soluția să fie aplicată pentru fiecare test din fișierul de intrare.

Cunoaștem că n este numărul de participanți la petrecere și considerăm permutarea identică cu n elemente (1, 2, ..., n).

Împărțim această permutare în două permutări, astfel încât fiecare dintre ele să aibă $[n/2]$, respectiv $n - [n/2]$ elemente.

Dacă n este număr par, adică $n = 2 \cdot k$, atunci pentru a inversa pe fiecare dintre cele două jumătăți avem nevoie de $k \cdot (k - 1)/2$ operații. În total se efectuează $k \cdot (k - 1)$ operații, adică $(n^2 - 2 \cdot n)/4$ operații.

Dacă n este număr impar, adică $n = 2 \cdot k + 1$, atunci pentru a inversa o jumătate din permutare se efectuează $k \cdot (k - 1)/2$ operații, iar pentru cealaltă $k \cdot (k + 1)/2$ operații. Rezultă că, în total, se efectuează $(n^2 - 2 \cdot n + 1)/4$ operații.

În timpul concursului trebuia ținut cont de faptul că rezolvarea se putea trimite de mai multe ori. Se trimitea o rezolvare de tipul celei prezentate anterior și se obținea răspunsul ***Accepted*** (*Soluție acceptată*), fără a se mai verifica dacă existau soluții mai bune. Nu existau.

Analiza complexității

Pentru un singur test ordinul de complexitate a rezolvării este $O(1)$ deoarece ordinul de complexitate a operațiilor de citire, scriere și calculare a timpului necesar pentru ca participanții să fie aranjați în ordine inversă este $O(1)$.

Dacă luăm în considerare toate testele și notăm cu t numărul acestora, ordinul de complexitate devine $O(t)$.

P070304: Supermagazin

În continuare vom prezenta soluția acestei probleme pentru un singur test.

Vom încerca să construim soluția optimă în ordine "cronologică invers".

Fie T momentul de timp maxim la care se poate vinde un produs.

În ultima unitate de timp pot fi vândute doar produse cu timpul limită de vânzare T. Este bine să profităm de această unitate de timp, deci alegem unul din acele produse și îl vindem. Mai exact, alegem produsul pe care obținem profit maxim.

La momentul $T - 1$ poate fi vândut fie un produs care are timpul limită de vânzare $T - 1$, fie unul din produsele care aveau timpul limită de vânzare T și care nu a fost vândut. Se alege produsul pe care se obține profit maxim.

La momentul de timp $K < T$ putem vinde orice produs cu timpul limită de vânzare mai mare sau egal cu K și care nu a fost deja vândut. Se alege produsul pe care se obține profit maxim.

Un set de date are n produse, maxim 10000. Dacă la fiecare pas alegem minimul cu metoda clasică, complexitatea este $O(N^2)$. Pentru un set de date cu multe teste mari, teoretic acest algoritm nu s-ar încadra în timp. Soluția este să folosim un *heap* pentru extragerea maximului. La fiecare moment de timp K (K descrește pornind de la T) se introduc în *heap* profiturile corespunzătoare produselor cu timpul limită de vânzare K, apoi se extrage maximul și se adună la soluție.

Analiza complexității

Operația de citire a datelor de intrare pentru un singur test are ordinul de complexitate $O(n)$, iar cea de scriere a rezultatului are ordinul de complexitate $O(1)$.

Calcularea profitului planificării optime are ordinul de complexitate $O(n \cdot \log n)$, deoarece fiecare dintre cele n produse este introdus în *heap*, iar ordinul de complexitate de introducere și extragere de elemente din acesta este $O(\log n)$.

Dacă m este numărul de teste dintr-un set de date, atunci ordinul de complexitate al algoritmului de rezolvare pentru această problemă devine $O(m \cdot n \cdot \log n)$.

P070305: Stăpânul inelului

Problema se rezolvă prin metoda *backtracking*. Ținând cont de testele date în concurs, limitele din enunț sunt dimensionate mai sus decât ar trebui. Astfel, testul maxim a constat în 200 de tufe, situate la distanța 1 unele de altele. Celelalte teste aveau mai puțin de 10 tufe, distanța între cele mai depărtate două fiind cel mult 50. Pe nici un test produsul cerut nu a depășit 50000. O dimensionare a limitelor corespunzătoare cu testele date ar fi încurajat mai mulți concurenți să abordeze problema.

Pentru a calcula produsul distanțelor dintre tufele consecutive se vor determina pozițiile tufelor. Prima tufă va avea coordonata 0, iar a doua coordonata *MAX*, unde *MAX* este maximul distanțelor.

După citirea datelor se creează doi vectori: unul care conține distanțele distincte, altul care conține numărul de apariții al fiecărei distanțe.

Abordarea *backtracking* constă în alegerea, la fiecare pas, a poziției în care va fi plasată o tufă. Mai exact, după ce le-am plasat pe primele K (prima la coordonata 0, a doua la coordonata *MAX*), tufa $K + 1$ va fi plasată la o poziție între coordonata tufei K și *MAX*. Făcând abstracție de tufa 2, coordonatele se vor obține în ordine crescătoare.

Pentru fiecare variantă de plasare se va verifica dacă există distanțele între tufa $K + 1$ și primele K tufe în vectorul de distanțe, folosind căutarea binară, și dacă numărul de apariții este strict pozitiv. În caz afirmativ, se scade numărul de apariții al distanțelor respective (deci considerăm distanțele ca fiind alocate) și se încearcă plasarea celorlaltor tufe. Numărul de apariții va fi restaurat la abandonarea variantei curente.

Limitarea produsului ne asigură că, dacă există o soluție, majoritatea distanțelor între tufe consecutive vor fi egale cu 1. Altfel spus, de multe ori *backtracking*-ul va înainta rapid, plasând mai multe tufe consecutive, fiecare la distanța 1 față de precedenta.

Dacă pe parcursul generării soluției, produsul distanțelor dintre tufele consecutive deja plasate depășește limita din enunț, varianta curentă nu este corectă. De asemenea, dacă distanța dintre tufa curentă și ultima tufă este mai mică decât numărul de tufe care încă nu au fost plasate, varianta curentă este abandonată.

Analiza complexității

Algoritmul de rezolvare a acestei probleme are ordinul de complexitate exponențial. Un calcul mai exact ar trebui să țină cont de limitele din enunț sau de limitele din testele date (cazuri diferite). Lăsăm efectuarea acestui calcul ca exercițiu pentru cititor.

În orice caz, un program care implementează această rezolvare rulează instantaneu pe testele date.

P070306: Subsecvența comună

Problema cere calcularea celei mai lungi subsecvențe comune a două șiruri de caractere. Aceasta este una din primele probleme care se învață la *metoda programării dinamice*.

În continuare prezentăm modul de rezolvare, pentru un singur test, bazat pe metoda *programării dinamice*.

Notăm șirurile cu X și Y și cu M și N lungimile acestora. Construim o matrice A cu M linii și N coloane, în care $A_{i,j}$ reprezintă lungimea celei mai lungi subsecvențe comune care se poate obține folosind doar primele i litere din X și primele j litere din Y.

În continuare prezentăm modul de construire al elementelor matricei A.

În cazul în care $X_i = Y_j$, atunci $A_{i,j} = A_{i-1,j-1} + 1$, deoarece subsecvența respectivă va fi formată din subsecvența comună a șirurilor, folosind primele i - 1, respectiv j - 1 litere, la care se va adăuga litera X_i.

În caz contrar, se alege maximul dintre $A_{i-1,j}$ și $A_{i,j-1}$, pentru că una din literele finale nu va intra în subsecvență.

Soluția problemei este dată, în final, de valoarea elementului $A_{M,N}$.

Analiza complexității

Operația de citire a datelor pentru un test are ordinul de complexitate $O(M + N)$, iar operația de afișare a rezultatelor are ordinul de complexitate $O(1)$.

Calculul unui element al matricei A se realizează în timp constant. Ordinul de complexitate al construirii matricei A este $O(M \cdot N)$, unde M și N sunt lungimile celor două șiruri.

În final, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(T \cdot M \cdot N)$, unde T reprezintă numărul de teste.

P070307: Rețeaua electrică

Rețeaua electrică este un graf orientat, care se transformă într-o rețea de transport astfel:

- capacitatea unui arc reprezintă transportul maxim pe arcul respectiv;
- se introduce o sursă fictivă din care pleacă spre toți producătorii de curent arce de capacitate $+\infty$;
- se introduce o destinație fictivă în care sosesc arce de capacitate $+\infty$ de la toți consumatorii.

Pe această rețea de transport se aplică un algoritm de flux maxim.

În final, soluția problemei este dată de suma fluxului pe arcele care pleacă din sursa fictivă.

Subiectul fluxului într-o rețea de transport este mult prea amplu pentru a fi tratat în cadrul acestui articol.

În continuare vom examina algoritmul general *Ford-Fulkerson*, împreună cu câteva optimizări. Amintim și existența algoritmilor de preflux, ceva mai dificil de înțeles, dar mai eficienți. Sugerăm cititorului interesat să studieze capitolul despre flux din cartea *Introducere în algoritmi*, *T. H. Cormen*, *C. E. Leisserson*, *R. R. Rivest*.

Pe scurt, fluxul pe arcele rețelei este dat de valorile $l(i, j)$ transportate de fiecare arc, iar capacitatea arcelor este dată de $L_{max}(i, j)$.

Fie un flux valid prin rețea. Acesta trebuie să respecte condițiile de conservare (ceea ce intră într-un nod intermediar este egal cu ceea ce iese) și mărginire (fluxul pe orice arc este cel mult egal cu capacitatea arcului respectiv).

Evident, fluxul 0 pe toate arcele este un flux valid.

Algoritmul *Ford-Fulkerson* este următorul:

- se pornește de la fluxul inițial 0 pe toate arcele;
- cât timp acest flux poate fi mărit (există un drum de creștere de la sursă la destinație), se modifică fluxul pe arcele drumului de creștere, obținându-se un flux mai bun.

Un drum de creștere este un drum de la sursă la destinație, folosind:

- arce (i, j) pe care fluxul este mai mic decât capacitatea (deci mai poate fi crescut);
- arce fictive (i, j) pentru care fluxul pe arcul (j, i) este nenul (deci poate fi scăzut). Practic folosirea unor arce din această categorie conduce la redirecționarea fluxului prin rețea.

Determinarea existenței unui drum de creștere de la sursă la destinație are ordinul de complexitate $O(m)$, printr-o parcurgere a grafului care contine arcele de mai sus. Pentru a calcula complexitatea totală a algoritmului este necesar să estimăm câte drumuri de creștere vor fi găsite până la determinarea fluxului maxim.

Dacă la fiecare pas se alege un drum oarecare, complexitatea pe cazul cel mai defavorabil este $O(m \cdot F_{max})$, unde F_{max} este valoarea fluxului maxim. Această limită se atinge în cazul în care fluxul în retea crește cu 1 la fiecare pas. Evident, pentru grafuri cu capacități mari pe muchii (deci unde fluxul maxim poate avea valori mari) acest algoritm nu este acceptabil.

O variantă este algoritmul *Edmonds-Karp*. Acest algoritm implementează căutarea drumului de creștere printr-o

parcurgere în lățime, obținându-se la fiecare pas un drum de creștere cu un număr minim de arce. Se poate demonstra că această variantă are ordinul de complexitate $O(n \cdot m^2)$.

O altă variantă este dată de algoritmul de flux maxim prin scalare. Astfel, fie C capacitatea maximă a unei muchii, $P = [\log C]$ și $K = 2^P$.

Primele drumuri de creștere căutate trebuie să aibă capacitatea cel puțin K, adică arcele pe care se introduce flux să permită introducerea a cel putin K unități, iar arcele de pe care se "scoate" (se redirecționează pe alte arce) flux să aibă fluxul cel puțin egal cu K. În momentul în care nu mai există astfel de drumuri se vor căuta drumuri de capacitate $K/2$, apoi $K/4$ etc.

Prin această metodă, se introduce o cantitate importanta în rețea încă de la primele iterații.

Se poate demonstra că acest algoritm are ordinul de complexitate $O(m^{2 \cdot \log C})$.

În practică, toți algoritmii de mai sus au timpi de execuție mult mai mici decât indică ordinul de complexitate. Astfel, la căutarea unui drum de creștere nu se examinează întotdeauna toate cele m muchii.

Pentru valori ale lui m apropiate de n (de exemplu, cazul grafurilor planare), metoda prin scalare poate conduce la performanțe excelente, ordinul de complexitate devenind $O(n^{2 \cdot \log C})$ față de $O(n^3)$ la algoritmii de preflux.

Analiza complexității

Operația de citire a datelor (mai dificilă pentru programatorii *Pascal* datorită parantezelor și virgulelor) are ordinul de complexitate $O(m + n)$.

Operația de transformare a grafului într-o rețea de transport are ordinul de complexitate $O(n)$.

Datorită faptului că ordinul de complexitate al oricărui algoritm de determinare a fluxului maxim într-o rețea de transport este mai mare decât celelalte operații suplimentare, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este dat de ordinul de complexitate al algoritmului de flux maxim folosit pe structura cu ajutorul căreia s-a reținut graful.

P070308: Pompierii

Această problemă se poate rezolva analizând toate posibilitățile de a înlocui semnele de întrebare cu operatori și evaluând de fiecare dată expresia rezultată. Pentru 10 semne de întrebare se obțin $4^{10} = 2^{20} = 1048576$ variante; aparent sunt destul de multe, dar calculatoarele din concurs aveau procesoare cu frecvența 2,6 *GHz*.

În momentul în care s-a obținut o expresie care dă rezultatul corect se întrerupe generarea posibilităților pentru testul respectiv și se afișează `yes`. Dacă nu se obține nici o astfel de expresie, se afișează `no`.

Pentru a optimiza pasul de evaluare a expresiei se pot determina poziția semnelor, a parantezelor, valorile operanzilor inițiali și ordinea aproximativă a operațiilor înainte de generarea tuturor variantelor. Mai exact, ordinea operațiilor dintre paranteze (dacă acestea există) este fixată datorită precizărilor din enunț.

Analiza complexității

Pentru un singur test, operația de citire a datelor de intrare are ordinul de complexitate $O(L)$, unde L este lungimea expresiei, iar operația de scriere a rezultatelor are ordinul de complexitate $O(1)$.

Fie Op numărul de operatori și S numărul de semne de întrebare.

Ordinul de complexitate al operației de generare a tuturor expresiilor posibile este $O(4^S)$.

Pentru fiecare expresie, evaluarea ei are ordinul de complexitate $O(Op)$.

Ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N \cdot 4^S)$ pentru fiecare set de date.

Dacă luăm în considerare toate testele din setul de date, atunci ordinul de complexitate al algoritmului de rezolvare pentru această problemă devine $O(T \cdot N \cdot 4^S)$, unde T este numărul total de teste.

P070309: Polinoame binare

Problema se rezolvă prin metoda forței brute.

Se generează toți cei $C(n, k)$ vectori binari. Pentru fiecare din acești vectori se evaluează funcția dată. Se afișează numărul de vectori pentru care rezultatul este 1.

Pentru generare se poate folosi fie metoda *backtracking*, fie o simplă numărare (adică un ciclu cu 2^n pași). În cazul numărării, deși se parcurg toți cei 2^n vectori cu n elemente (și nu numai prin cei cu k de 1), timpul total de lucru este redus, deoarece pentru vectorii care nu au exact k de 1 funcția nu se va evalua.

La evaluare, pentru ca un anumit produs de coeficient 1 să fie 1 este necesar ca toate componentele sale să fie 1. Numărul de ordine al produsului trebuie să fie deci conținut în reprezentarea binară a numărului de ordine al vectorului (ținând cont de generarea prin numărare și de toți cei 2^n vectori). Excepție face cazul în care a_0 este 1, deci valoarea funcției este 1 pentru orice vector.

În momentul în care unul dintre produsele cu coeficienți 1 este 1, valoarea funcției este 1, deci se întrerupe evaluarea. Aceasta face ca evaluarea să fie, în general, destul de rapidă, deși teoretic ordinul de complexitate al acestei operații este $O(2^n)$.

Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate $O(2^n)$, iar scrierea datelor are ordinul de complexitate $O(1)$.

Pentru fiecare vector, evaluarea are ordinul de complexitate, teoretic, $O(2^n)$. În total sunt $C(n, k)$ vectori.

Ordinul total de complexitate al algoritmului este, teoretic, $O(2^n \cdot C(n, k))$. Datorită întreruperii evaluării, un program care implementează această rezolvare va rula destul de rapid.