Olimpiada INTERNAȚIONALĂ de INFORMATICĂ

Vă prezentăm soluțiile oficiale ale celor șase probleme pe care au încercat să le rezolve participanții la IOI 2001. Implementările algoritmilor prezentați sunt disponibile pentru download la www.ginfo.ro/revista/11_7/main.html.

P060101: Telefoane mobile

Timo Tossavainen, Jyrki Nummenmaa

Este mai ușor să găsim soluția dacă luăm în considerare doar cazul unidimensional (un vector de dimensiune N) pentru care avem actualizări și interogări pentru un interval. După găsirea unei soluții pentru această variantă, va fi relativ simplu să o generalizăm pentru două (sau chiar mai multe) dimensiuni.

Cazul unidimensional

Dacă valorile vectorului sunt memorate în maniera clasică, atunci determinarea sumei unui interval necesită $O(N)$ operații. Răspunsul la o interogare referitoare la suma valorilor dintr-un interval $[X, Y]$ poate fi determinat dacă sunt cunoscute sumele pentru intervalele $[1, X - 1]$ și $[1, Y]$; el va fi calculat scăzând cele două valori. Sumele corespunzătoare intervalelor de forma $[1, I]$ pot fi stocate într-un tabel și, în acest caz, răspunsul la interogările referitoare la intervale de forma $[X, Y]$ poate fi calculat într-un timp $O(1)$. Totuși, dacă dorim să păstrăm aceste sume, o actualizare ar necesita $O(N)$ operații.

Arborele indexat binar este o structură de date care permite determinarea sumelor și efectuarea actualizărilor în timp $O(\log N)$ și nu necesită spațiu suplimentar de memorie, cantitatea de memorie ocupată fiind egală cu cea pe care ar ocupa-o vectorul clasic. În figura 1 este prezentat un astfel de arbore.

Elementele tabloului sunt indexate cu numere cuprinse între 1 și N, iar elementul de la indicele I conține suma corespunzătoare intervalului $[I - 2^k + 1, I]$, unde k reprezintă numărul zerourilor aflate după ultimul 1 din reprezentarea binară a valorii I. Suma corespunzătoare unui interval poate fi determinată folosind $2 \cdot O(\log N)$ operații.

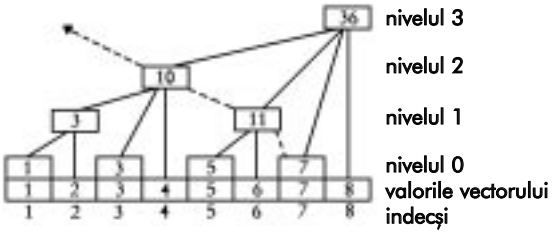

Figura 1. Un arbore indexat binar. Liniile continue indică direcțiile (traseele de la un element la altul) folosite pentru actualizare, iar linia întreruptă indică direcția folosită pentru interogarea referitoare la suma elementelor din intervalul [1, 7] (7 + 11 + 10 = 28). Tabloul inițial nu trebuie memorat; în memorie se află doar liniile care reprezintă niveluri ale arborelui.

În cazul în care numărul de telefoane active dintr-o anumită celulă I crește/scade cu x, va trebui să actualizăm valorile păstrate. O actualizare necesită modificarea tuturor elementelor care păstrează sume ale unor intervale, care conțin și poziția I. Vom porni de la elementul cu indicele I și vom aduna/scădea valoarea x. Pentru a determina următorul element care trebuie actualizat, va trebui să adunăm la I valoarea 2^k unde k reprezintă numărul zerourilor aflate după ultimul 1 din reprezentarea binară a valorii I. Vom continua până în momentul în care obținem o valoare mai mare decât N. Se observă că actualizările necesită $O(\log N)$ operații.

Pentru interogări referitoare la suma elementelor unui interval de forma $[1, I]$ vom porni tot de la elementul cu indicele I. Pentru a determina următorul element va trebui să scădem din I valoarea 2^k, unde k reprezintă numărul zerourilor aflate după ultimul 1 din reprezentarea binară a valorii I. Vom continua până în momentul în care obținem valoarea 0. Se observă că o astfel de interogare necesită $O(\log N)$ operații. Datorită faptului că, pentru a determi-

na suma corespunzătoare unui interval de forma [*X*, *Y*], avem nevoie de două astfel de interogări, numărul total de operații va fi $2 \cdot O(\log N)$. Răspunsul la interogare este calculat însumând valorile tuturor elementelor traversate.

Interogările pot fi optimizate pentru intervale mici, dacă se observă că cele două sume care trebuie calculate sunt obținute, folosind anumite elemente comune. Datorită faptului că răspunsul final este dat de diferența celor două sume, se observă că valorile elementelor comune sunt adunate și apoi scăzute. Așadar, căutarea se poate opri în momentul în care se ajunge la un element comun.

Exemplu pentru actualizare

Presupunem că dorim să actualizăm valoarea elementului de pe poziția 3 și avem $N = 8$. Vom porni de la indicele $3 = 11_2$. Vom actualiza valoarea, și apoi vom trece la elementul următor. Nu există nici un 0 după ultimul 1 din reprezentarea binară a lui 3, deci vom trece la poziția $3 + 2^0 = 4 = 100_2$. Actualizăm valoarea și apoi trecem la elementul următor. Există două zerouri după ultimul 1 din reprezentarea binară a lui 4, deci vom trece la poziția $4 + 2^2 = 8 = 1000_2$. După actualizarea valorii vom trece la elementul $8 + 2^3 = 16$, deoarece există trei zerouri după ultimul 1 din reprezentarea binară a lui 8. Observăm că am depășit valoarea *N* și ne oprim.

Exemplu pentru interogare

Presupunem că dorim să aflăm suma elementelor din intervalul [1, 7]. Vom porni de la indicele $7 = 111_2$. Preluăm valoarea din această poziție într-o variabilă care va conține suma parțială și apoi trecem la pasul următor. Nu există nici un 0 după ultimul 1 din reprezentarea binară a lui 7, deci vom trece la poziția $7 - 2^0 = 6 = 110_2$. Preluăm valoarea și o adunăm la suma parțială, apoi trecem la pasul următor. Există un zero după ultimul 1 din reprezentarea binară a lui 6, deci vom trece la poziția $6 + 2^1 = 4 = 100_2$. Preluăm și această valoarea (adăugând-o la suma parțială) și apoi vom obține valoarea $4 - 2^2 = 0$, deoarece există două zerouri după ultimul 1 din reprezentarea binară a lui 4. Am obținut valoarea 0, deci ne oprim; suma obținută în acest moment corespunde intervalului [1, 7].

Cazul bidimensional

Soluția cazului unidimensional poate fi generalizată pentru oricâte dimensiuni. Pentru rezolvarea acestei probleme sunt necesare două dimensiuni. Arborii sunt construiți în același mod ca și în cazul unidimensional, formându-se un "arbore de arbori". În această structură, elementul de pe poziția [*X*, *Y*] va conține suma corespunzătoare dreptunghiului care are colțul stânga sus în poziția $[X - 2^k + 1, Y - 2^l + 1]$, unde *k* reprezintă numărul zerourilor aflate după ultimul 1 din reprezentarea binară a valorii *X*, iar unde *l* reprezintă numărul zerourilor aflate după ultimul 1 din reprezentarea binară a valorii *Y*.

Folosind această structură vom putea răspunde într-un timp $O((\log N)^2)$ la interogări referitoare la suma elementelor din dreptunghiul care are colțul din stânga-sus în poziția [1, 1] și colțul din dreapta-jos în poziția [*X*, *Y*]. Pentru a determina suma *S* a elementelor unui dreptunghi care are colțul din stânga-sus în poziția $[X_1, Y_1]$ și colțul din dreapta-jos în poziția $[X_2, Y_2]$ vom avea nevoie de patru interogări (toate referitoare la dreptunghiuri care au colțul din stânga-sus în poziția [1, 1]):

- dreptunghi cu colțul din dreapta-jos în poziția $[X_2, Y_2]$ (S_1);
- dreptunghi cu colțul din dreapta-jos în poziția $[X_1, Y_2]$ (S_2);
- dreptunghi cu colțul din dreapta-jos în poziția $[X_2, Y_1]$ (S_3);
- dreptunghi cu colțul din dreapta-jos în poziția $[X_1, Y_1]$ (S_4).

Vom avea $S = S_1 - S_2 - S_3 + S_4$.

În cazul bidimensional poate fi realizată aceeași optimizare ca în cazul unidimensional. Astfel, pentru dreptunghiuri mici, căutarea se oprește în momentul în care se găsesc celule comune în momentul calculării unor sume care apar cu semne diferite în formula finală.

Observație

Pentru ca proprietățile folosite pentru actualizare și interogare să fie respectate, indicii folosiți pentru un arbore indexat binar trebuie să înceapă cu valoarea 1. În această problemă, indicii încep cu valoarea 0, deci trebuie efectuată o corecție a acestora (adunarea valorii 1 după citirea lor).

Exemple pentru actualizare

Să presupunem că dorim să actualizăm valoarea elementului de pe poziția (2, 5) și avem $N = 8$. Va trebui să determinăm elementele de pe linia a doua care trebuie actualizate. Pentru o linie folosim același procedeu ca și în cazul unidimensional. Vom actualiza elementele de pe coloanele $5 = 101_2$, $5 + 2^0 = 6 = 110_2$ și $6 + 2^1 = 8 = 1000_2$. Vom trece apoi la linia următoare: $2 + 2^1 = 4 = 100_2$ și actualizăm elementele de pe aceleași coloane. Ultima linie va fi $4 + 2^2 = 8 = 1000_2$ și se vor actualiza elementele de pe aceleași coloane.

În figura 2 sunt prezentate traseele parcurse pentru actualizarea elementului de pe poziția (5, 2).

Vom prezenta și exemplul actualizării elementului de pe poziția (1, 1). Traseele parcurse sunt ilustrate în figura 3.

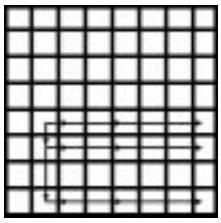

Figura 2. Traseele parcurse pentru actualizarea structurii de date în cazul modificării numărului de telefoane mobile active din celula de coordonate (5, 2)

Exemple pentru interogare

Să presupunem că dorim să aflăm suma elementelor din dreptunghiul care are colțul din stânga-sus în poziția (1, 1)

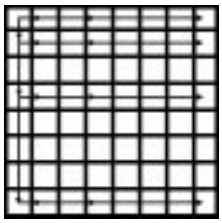

Figura 3. Traseele parcurse pentru actualizarea structurii de date în cazul modificării numărului de telefoane mobile active din celula de coordonate (1, 1)

și colțul din dreapta-jos în poziția (7, 7). Va trebui să determinăm elementele de pe linia a șaptea care trebuie actualizate. Pentru o linie folosim același procedeu ca și în cazul unidimensional. Vom adăuga la suma parțială elementele de pe coloanele $7 = 111_2$, $7 - 2^0 = 6 = 110_2$ și $6 - 2^1 = 4 = 100_2$. Vom trece apoi la linia următoare: $7 - 2^1 = 6 = 110_2$ și vom adăuga la suma parțială elementele de pe aceleași coloane. Ultima linie va fi $6 - 2^1 = 4 = 100^2$ și se vor aduna elementele de pe aceleași coloane.

În figura 4 sunt prezentate traseele parcurse pentru determinarea sumei elementelor din dreptunghiul considerat.

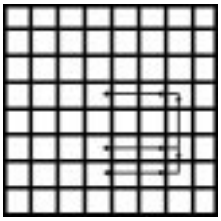

Figura 4. Traseele parcurse pentru determinarea sumei corespunzătoare dreptunghiului care are colțul din stânga-sus în poziția (1, 1) și colțul din dreapta-jos în poziția (7, 7).

Vom prezenta și exemplul determinării sumei elementelor din dreptunghiul care are colțul din stânga-sus în poziția (1, 1) și colțul din dreapta-jos în poziția (3, 6). Traseele parcurse sunt ilustrate în figura 5.

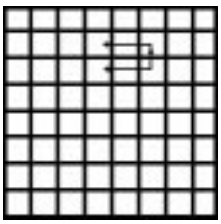

Figura 5. Traseele parcurse pentru determinarea sumei corespunzătoare dreptunghiului care are colțul din stânga-sus în poziția (1, 1) și colțul din dreapta-jos în poziția (3, 6).

Dreptunghiuri corespunzătoare unor celule

În figura 6 sunt reprezentate dreptunghiuri corespunzătoare anumitor celule. Celula este hașurată în negru, iar dreptunghiul corespunzător este hașurat în gri.

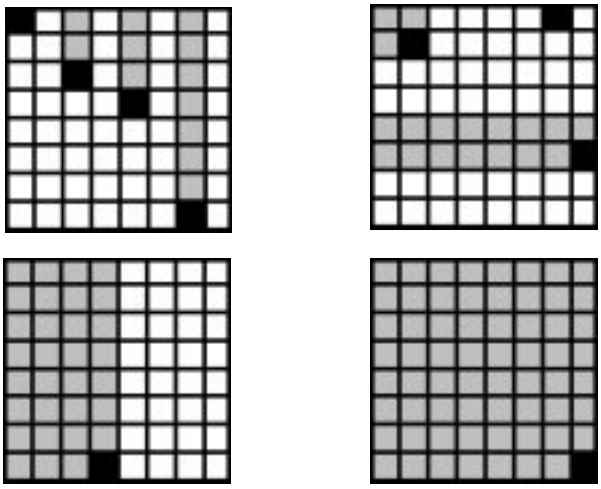

Figura 6. Dreptunghiurile (hașurate în gri) corespunzătoare anumitor celule (hașurate în negru)

P060102: Jocul Ioiwari

Gyula Horvath

Jocul *Ioiwari* este o adaptare a jocurilor existente din familia *Mancala*. Diferite caracteristici (numărul de adâncituri, numărul și distribuția mărgelelor și, în special, regulile jocului) au fost modificați pentru ca următoarele condiții să fie satisfăcute:

- jocul poate fi câștigat de Jucătorul 1;
- în majoritatea situațiilor strategia de câștig nu se găsește ușor;
- există o soluție eficientă care folosește o cantitate rezonabilă de resurse (memorie și timp *CPU*);
- jocul se poate termina la egalitate;
- numărul diferitelor instanțe ale jocului este suficient de mare pentru testarea programelor concurenților.

Considerăm graful orientat ale cărui noduri sunt perechi de forma (w, B) unde w poate avea valoarea 1 sau valoarea 2 și identifică jucătorul aflat la mutare, iar B este o posibilă configurație a tablei (ignorând băncile).

Există un arc de la nodul (u, A) la nodul (v, B) dacă și numai dacă $u + v = 3$ și configurația B poate fi obținută din configurația A, pe baza unei mutări valide. Datorită faptului că, după fiecare mutare, numărul de mărgele din adâncituri scade, graful este aciclic.

Fie $Dif(w, B)$ cea mai mare diferență de scor pe care o poate obține Jucătorul 1 în cazul în care se pornește din configurația B și Jucătorul 2 joacă optim. Ajungem la următoarea formulă:

$Move(B, i)$ reprezintă configurația tablei după efectuarea mutării corespunzătoare adâncituri i (mutarea care începe cu alegerea adâncituri i), iar $D(B, i)$ reprezintă di-

$$Dif(w,B)=\begin{cases}0, \text{ dacă este configurația în care nu mai}\\ \quad\text{există mărgele în nici o adâncitură}\\ \max\{Dif(2, Move(B,i)+D(B,i))\}, \text{ pentru}\\ \quad\text{toate mutările valide } i, \text{ dacă } w=1\\ \min\{Dif(1, Move(B,i)+D(B,i))\}, \text{ pentru}\\ \quad\text{toate mutările valide } i, \text{ dacă } w=2\end{cases}$$

ferența dintre numărul mărgelelor aflate în banca Jucătorului 1 și numărul mărgelelor aflate în banca Jucătorului 2 după efectuarea mutării corespunzătoare adânciturii *i*.

Este evident că există o strategie sigură de câștig pentru Jucătorul 1, dacă și numai dacă $Dif(1, B) > 0$.

Algoritmul recursiv

Formula recursivă de determinare a valorilor *Dif* poate fi folosită pentru a găsi o soluție rapidă a problemei: Jucătorul 1 va alege întotdeauna adâncitura *i* căreia îi corespunde o valoare maximă pentru *Dif*. Este evident că acest algoritm nu este eficient, deoarece trebuie să determine la fiecare pas (înaintea fiecărei mutări) valoarea funcției *Dif*. Mai mult, este recalculată valoarea corspunzătoare unei configurații de fiecare dată când aceasta este "accesată" (valoarea corespunzătoare apare în formulă).

Algoritmul de memoizare

Putem îmbunătăți eficiența algoritmului recursiv dacă memorăm valorile $Dif(w, B)$ după calcularea lor pentru prima dată și apoi le preluăm de fiecare dată când avem nevoie de ele, fără a mai fi necesară recalcularea lor.

Pentru a memora aceste valori, vom atribui câte un număr unic de identificare pentru fiecare configurație posibilă a tablei. Deoarece, pe parcursul jocului, fiecare adâncitură poate conține cel mult cinci mărgele, fiecare configurație poate fi identificată în mod unic printr-un număr de șapte cifre scris în baza 6.

La început, programul va determina valorile optime pentru subproblemele corespunzătoare determinării valorilor $Dif(w, B)$ și păstrează mutarea optimă într-un tabel. Calculele sunt realizate folosind o tehnică recursivă cu memoizare. În timpul jocului, algoritmul va căuta mutarea optimă în acest tabel.

Cantitatea de memorie necesară este de $3 \cdot MaxN$ bytes, unde *MaxN* este 279935 (cel mai mare număr de șapte cifre scris în baza 6).

Se observă că fiecare configurație este echivalentă cu alte configurații obținute prin rotație. Spațiul necesar poate fi redus la jumătate dacă atribuim fiecărei configurații cel mai mic număr obținut considerând toate configurațiile echivalente obținute prin rotație.

Timpul necesar execuției programului este proporțional cu numărul arcelor grafului (la care se adaugă timpul necesar inițializării tabelelor). O aproximație (prin adaos) a limitei superioare este $7 \cdot MaxN$.

Algoritmi greedy

Poate fi implementat un algoritm care folosește o strategie de tip *greedy*. Algoritmul va "privi în viitor" doar un singur pas (o singură mutare). Jucătorul va alege întotdeauna mutarea care corespunde optimului local. Această strategie este optimă doar pentru câteva instanțe ale jocului. Deși un algoritm, care "privește în viitor" doi pași ar trebui să fie mult mai bun, în practică un astfel de algoritm pierde aproape întotdeauna jocul.

Algoritmul probabilist

O altă posibilitate de rezolvare a problemei o constituie un algoritm care încearcă să ghicească mutarea corectă. Probabilitatea de a câștiga jocul alegând la întâmplare (aleator) mutările efectuate este foarte mică.

P060103: Doi-cinci

Sergey Melnik

Soluția se bazează pe o funcție *numconts* care, pentru o mulțime arbitrară dată, formată din litere care au pozițiile fixate, determină numărul de posibilități de a plasa restul literelor. Se va încerca poziționarea literelor pe marginile unui contur al unui tablou *Young* (același tip de contur care apare și în cazul problemei ***Depozit*** - vezi pagina 20).

Dacă toate literele au fost poziționate, atunci există o singură posibilitate de continuare: nu se mai realizează nici o altă operație. Deci, pentru o stare completă (toate literele sunt poziționate), *numconts* va inițializa cu `1` valoarea `snum[states - 1]`. În continuare se va completa tabelul folosind funcția *calcstate*.

Funcția *calcstate* returnează numărul posibilităților de a alege poziții pentru literele rămase, dându-se literele poziționate anterior și conturul corespunzător acestei poziționări. Se va încerca poziționarea următoarei litere în toate pozițiile valide și se vor însuma valorile obținute prin autoapeluri recursive corespunzătoare noilor contururi obținute. Rezultatele intermediare sunt păstrate în tabelul *snum* pentru a nu fi necesară recalcularea lor.

Rezolvarea problemei folosind funcția *numconts* este relativ simplă. Să presupunem că dorim să determinăm numărul corespunzător unui anumit cuvânt. Vom păstra o valoare corespunzătoare numărului primului cuvânt (în ordine alfabetică), care poate fi generat folosind literele care sunt fixate în acel moment. Vom fixa literele, una câte una, începând cu cea mai semnificativă. O literă va fi fixată alegând la început valoarea `'A'` și apoi vom alege succesiv valorile următoare până când ajungem la cea dorită. De fiecare dată când alegem o nouă valoare pentru litera curentă, folosim funcția *numconts* pentru a determina numărul cuvintelor peste care "s-a sărit" pe parcursul efectuării operației și valoarea curentă este actualizată corespunzător. După fixarea tuturor literelor am obținut soluția corectă.

Efectuarea operației inverse este asemănătoare. Vom poziționa literele una cîte una, incrementând valorile lor până în momentul în care valoarea curentă depășește valoarea dorită. În momentul apariției depășirii efectuăm "un pas înapoi" (decrementăm valoarea literei), și trecem la litera următoare. La sfârșit, valoarea curentă este egală cu valoarea dorită și am obținut cuvântul corect.

Funcția *calcstate* determină numărul literelor care se află între litera curentă și ultima literă inserată în contur. Folosind acest număr, este posibil să verificăm numărul continuărilor peste care s-a "sărit" alegând continuarea curentă și care au fost "pierdute". Acest număr al continuărilor "pierdute" este scăzut din numărul continuărilor

posibile. Adăugând literele în acest mod (folosind o tehnică recursivă), vom observa că unele litere pot fi așezate într-o singură poziție. Aceste litere sunt fixate; aceste poziții fixe sunt folosite la plasarea literelor.

P060104: Score

Timo Poranen, Jyrki Nummenmaa

Testarea anumitor tipuri de proprietăți ale unui proces cu număr de stări poate fi privită ca un joc *MC*. Câștigarea unui joc *MC* indică dacă proprietatea considerată este respectată pentru un anumit proces. Jocul ***Score*** este o adaptare a jocurilor *MC* de conectivitate a ciclurilor, o categorie specială a jocurilor *MC*. Noțiunea de ***conectivitate a ciclurilor*** indică faptul că toate ciclurile cuprinse într-o componentă tare conexă a grafului jocului au cel puțin un nod comun. Strategia optimă pentru jocul ***Score*** poate fi determinată în timp liniar folosind un algoritm bazat pe căutarea în adâncime.

Din enunțul problemei rezultă că starea inițială a jocului permite întotdeauna alegerea unei săgeți astfel încât, dacă jocul durează destul de mult, jetonul ajunge din nou în poziția inițială. Aceasta înseamnă că ciclurile grafului au cel puțin un nod comun și anume cel corespunzător poziției inițiale.

Strategia de câștig implică alegerea unui drum de la poziția curentă la poziția de pornire în cazul în care după parcurgerea drumului scorul Jucătorului 1 este mai mare decât scorul adversarului. Dacă acest lucru nu este posibil, atunci se alege un drum după parcurgerea căruia scorul adversarului este minim.

Pentru a găsi cea mai bună alegere (cea mai convenabilă săgeată) pentru poziția curentă, se folosește un algoritm care combină căutarea în adâncime cu următoarele reguli recursive de alegere a săgeților:

- dacă există unul sau mai multe drumuri după a căror parcurgere scorul Jucătorului 1 este mai mare decât cel al adversarului, atunci se alege drumul care duce la obținerea celui mai mare scor;
- dacă după parcurgerea oricăruia dintre drumurile care pornesc din poziția curentă, scorul adversarului este mai mare, atunci se alege drumul pentru care scorul adversarului este minim.

Algoritmi greedy

Jucătorul 1 poate folosi o strategie în care "privește în viitor" un singur pas. El va alege întotdeauna o săgeată care va duce la o poziție a cărui posesor este și căreia îi corespunde o valoare cât mai mare.

Dacă nici o săgeată care pleacă din poziția curentă nu ajunge într-o poziție a Jucătorului 1, atunci se va alege o săgeată care duce în poziția adversarului cu cea mai mică valoare.

Această strategie poate fi îmbunătățită dacă se "privește în viitor" cu mai mulți pași. Totuși, găsirea strategiei optime poate fi găsită doar prin căutarea tuturor drumurilor posibile până la atingerea poziției inițiale.

Algoritmul probabilist

O altă posibilitate de rezolvare a problemei o constituie un algoritm care încearcă să ghicească săgeata corectă. Probabilitatea de a câștiga jocul alegând la întâmplare (aleator) săgețile este foarte mică.

P060105: Criptare dublă

Tom Verhoeff

Criptarea dublă, așa cum este ea aplicată în această problemă, oferă o securitate suplimentară mult mai mică decât s-ar părea la prima vedere. Dacă lungimea cheii este de n biți, atunci o căutare exhaustivă trebuie să ia în considerare toate cele 2^n chei. S-ar putea crede că criptarea dublă, folosind două chei independente a câte n biți, corespunde unei criptări simple cu o cheie având lungimea de $2 \cdot n$ biți. Totuși așa-numitul ***meet-in-the-middle attack*** (*atac cu întâlnire la mijloc*) arată că această criptare nu oferă o securitate mai mare decât o criptare simplă cu o cheie de lungime $n + 1$. În practică este folosită criptarea triplă, cunoscută sub numele de ***Triple DES***.

Atacul cu întâlnire la mijloc funcționează astfel: textul p este criptat folosind toate cele 2^n chei posibile și rezultatele sunt stocate (folosind o tehnică oarecare) într-un tabel; în continuare, textul obținut după criptarea dublă este decriptat folosind toate cele 2^n chei posibile; fiecare rezultat al decriptării este căutat în tabel și în cazul în care s-a întâlnit o concordanță, atunci cheia k_2 este cheia curentă, iar cheia k_1 este cheia corespunzătoare poziției din tabel la care s-a găsit concordanța. Teoretic, este posibilă apariția mai multor concordanțe, dar este foarte puțin probabil să apară un astfel de caz. Pentru această problemă este suficientă găsirea unei singure perechi de chei.

Vom studia acum puțin mai detaliat efortul computațional de care este nevoie pentru găsirea celor două chei. Pentru a traversa spațiul cheilor posibile avem nevoie de două operații:

```
FirstKey(var key:Block);
NextKey(var key:Block):Boolean;
```

unde valoarea returnată de `NextKey` indică existența sau inexistența unui succesor pentru cheia curentă.

Structura de date corespunzătoare tabelului care păstrează rezultatele criptării textului p trebuie să pună la dispoziție următoarele operații:

```
EmptyTable;
Store(const msg:Block; const key:Block);
Retrieve(const msg:Block;
         var found:Boolean; var key:Block);
```

Operația `EmptyTable` inițializează un tabel vid.

Cu ajutorul operației `Store` este adăugată o pereche (`msg`, `key`) cu proprietatea `msg = Encrypt(p,key)`.

Prin intermediul operației `Retrieve` este verificată prezența unui mesaj `msg` și, în cazul găsirii unui astfel de mesaj, este furnizată valoarea cheii corespunzătoare `k` pentru care `msg = Encrypt(p,key)`.

Mai trebuie adăugată o operație care permite verificarea egalității a două mesaje (structuri de tip `Block`).

Operațiile `Store` și `Retrieve` trebuie să fie efectuate foarte rapid (preferabil în timp constant), fapt care sugerează folosirea unei tabele de dispersie (tabelă *hash*) deschise.

P060106: Depozit

Jyrki Nummenmaa, Erkki Mäkinen

Fără a restrânge generalitatea putem presupune că avem containere etichetate cu numere cuprinse între 1 și *N*.

Algoritmul de căutare exhaustivă

Vom genera toate permutările numerelor 1, 2, ..., *N*. Pentru fiecare permutare vom genera plasamentul containerelor în depozit folosind metoda muncitorului și vom compara plasamentul obținut cu plasamentul furnizat la intrare. Vom furniza la ieșire toate permutările cărora le corespunde un plasament identic cu cel de la intrare.

Totuși, un program care implementează această versiune a algoritmului nu se încadrează în limita de timp admisă decât pentru valori mici ale lui *N*.

Algoritm care folosește metoda backtracking

În continuare vom prezenta un algoritm bazat pe următoarea idee: vom elimina succesiv containere, determinând starea depozitului pentru situația în care containerul eliminat ar fi fost ultimul sosit în depozit. Determinarea stărilor după eliminarea unui container poate implica mutarea unor containere pe liniile de deasupra, operație opusă celei de la inserare (care implică mutări pe liniile de dedesubt).

Pentru fiecare container de pe prima linie vom apela metoda `RecursiveDelete`. Aceasta are parametrii `depot`, `item` și `solution`; pentru primul apel parametrul `solution` va fi un șir vid (nici un container).

Dacă `item` este singurul container rămas în depozit, atunci acesta este adăugat în lista `solution` și soluția obținută este furnizată la ieșire.

Dacă `item` nu este singurul container rămas în depozit, atunci acesta este adăugat în lista `solution`, este eliminat din depozit și pentru toate containerele de pe prima linie este apelată metoda `PullUp(depot,item,solution)`.

Metoda `PullUp` are trei parametri: `depot`, `item` și `solution`. Dacă ne aflăm pe ultima linie, atunci determinăm, pe baza valorilor corespunzătoare, toate containerele care pot înlocui containerul `item` din linia anterioară. Fiecare astfel de container va fi eliminat și pentru toate containerele `newitem` de pe prima linie vom apela `RecursiveDelete(depot,newitem,solution)`.

Acest algoritm ia în considerare toate posibilitățile de a elimina containerele din depozit, deci va determina toate soluțiile posibile ale problemei.

Din nefericire, acest algoritm nu este eficient. Sunt "favorizate" intrările care conțin linii scurte (în cazul cel mai favorabil avem *N* linii cu câte un container). În cazul în care avem o singură linie cu *N* containere, algoritmul nu este mai eficient decât cel care determină toate permutările posibile. Mai mult, datorită detaliilor de implementare, timpul de execuție este în unele cazuri mai mare.

O optimizare evidentă poate fi observată imediat. În momentul în care nu mai există containere decât pe prima linie, putem construi restul soluției în timp liniar, deoarece containerele respective trebuie să fi fost inserate în ordinea în care se află în acel moment. În caz contrar, inserarea unui container ar fi trebuit să ducă la mutarea pe linia următoare a unui alt container.

Chiar și folosind această optimizare, algoritmul nu este eficient. Dar, să observăm că `PullUp` execută mult mai multe operații decât ar fi necesar. Dacă studiem mai atent problema, "descoperim" lema prezentată în continuare.

Lemă
Presupunem că *d* este un depozit care conține *N* - 1 containere și în care intră un container. Considerăm operația de mutare a unui container *y* de pe linia *r* pe linia *r* + 1 și inserarea containerului *x* pe linia *r*, în locul containerului *y*. Este evident că avem $x < y$ și $y' < x$ pentru fiecare container *y*' aflat înaintea containerului *y* pe linia *r* înaintea mutării acestuia pe linia următoare.

Folosind această lemă ajungem la următorul rezultat:

Lemă
Presupunem că *d* este un depozit care conține *N* containere. Considerăm operația de mutare a unor containere de pe linia *r* + 1 pe linia *r*. În acest caz, fiecare container de pe linia *r* + 1 poate fi luat în considerare doar dacă este găsit înlocuitorul pentru exact un container de pe linia anterioară.

Folosind această lemă putem micșora foarte mult spațiul de căutare. Pentru a mări viteza de căutare putem păstra pentru fiecare container o listă cu containerele de pe linia următoare care pot fi mutate în locul său.

Dacă luăm în considerare următoarele rezultate, poate fi găsit un alt algoritm:

Lemă
În momentul inserării unui container, ultimul container care trebuie poziționat va ajunge pe ultima poziție a unei linii, iar linia de deasupra sa nu poate fi mai scurtă decât linia pe care se află.

Lemă
Despre fiecare container aflat la sfârșitul unei linii care nu este la fel de lungă ca linia de deasupra, se poate spune că a fost ultimul adăugat.

Aceste leme sugerează o căutare similară cu cea prezentată anterior, dar începând cu ultima linie. Avantajul principal îl constituie faptul că vom începe doar cu containere care pot fi mutate pe linia anterioară printr-o operație inversă inserării, spre deosebire de algoritmul anterior care începe căutarea de la prima linie și ar putea porni cu un container care nu poate fi ultimul container inserat.