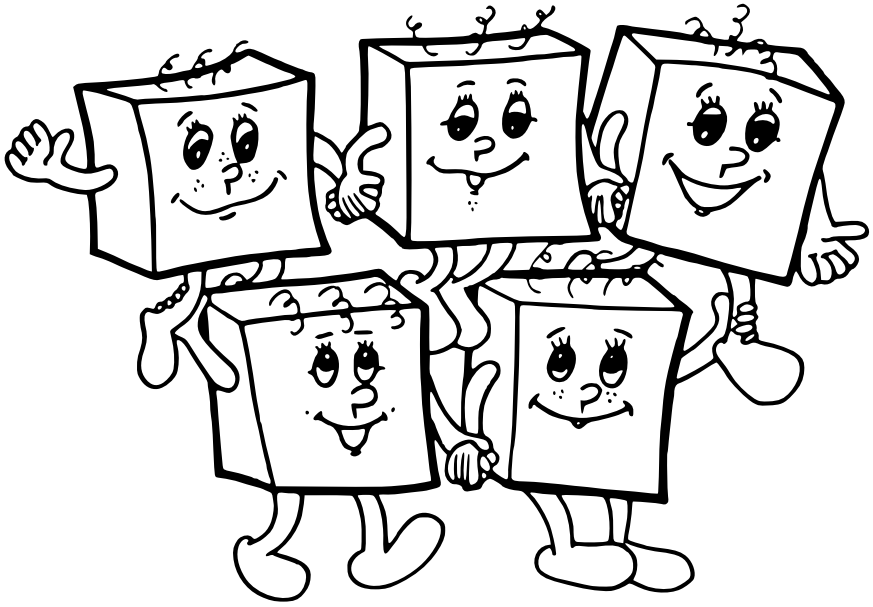
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



dvadsiaty tretí ročník
školský rok 2007/08

riešenia celoštátneho kola
kategória A

2. súťažný deň

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://www.ksp.sk/oi/`.

Riešenia druhého súťažného dňa

A-III-4 O lenivých prasiatkach

Začneme tým, že si uvedomíme, že stromy ani senníky sa nehýbu. V každom okamihu teda vieme situáciu jednoznačne popísať tak, že udáme súradnice všetkých prasiatok.

Všetkých možných situácií nie je až tak veľa, ako by sa na prvý pohľad mohlo zdať. Máme $2P$ súradníc, každá je z rozsahu od 1 po N, teda možných popisov je rádovo N^{2P}. (V skutočnosti menej, lebo prípustné sú len tie popisy, kde je všetkých P políčok navzájom rôznych.) Pre intuíciu, pre $P = 4$ a $N = 8$ je N^{2P} len 16 miliónov.

Prvé, pomerne priamočiare riešenie bude používať prehľadávanie do šírky. Budeme postupne zostrojovať situácie, ktoré sú dosiahnuteľné na jeden krok, na dva kroky, a tak ďalej.

Samozrejme, aby sme neprezerali tú istú situáciu viackrát, potrebujeme si napríklad v poli pamätať, ktoré situácie sme už videli. Na to môžeme buď použiť $2P$-rozmerné pole, alebo si napísať pomocné funkcie na kódovanie celého popisu situácie do jedného čísla a použiť pole 1-rozmerné. (Druhý prístup je o čosi lepší, lebo sa nepotrebujeme trápiť s rozmermi poľa, spravíme ho proste také veľké, ako nám pamäťový limit dovolí.)

Prvé zlepšenie

Takýto prístup však ešte na plný počet bodov nestačí. Môžeme však našu situáciu ľahko o niečo vylepšiť. Stačí si napríklad uvedomiť, že nám nezáleží na tom, ktoré prasiatko je ktoré. A teda si ich súradnice môžeme udržiavať utriedené. Takto sa nám vždy $P!$ situácii zmení na jednu. Pre intuíciu, pre $P = 4$ a $N = 8$ je $N^{2P}/P!$ už len 700 tisíc.

Prehľadávanie z oboch strán naraz

Napriek tomu nás problém s priveľa stavmi začne pre veľké hodnoty N trápiť znova.

Problém je v tom, že pri každom ďalšom ťahu môže počet dosiahnuteľných situácií narásť až exponenciálne: v prvom ťahu máme rádovo $4P$ možností, po druhom ich už môže byť až rádovo $(4P)^2$, a tak ďalej. A väčšina týchto situácií je nám úplne nanič, ale to naše riešenie (zatiaľ) nevie rozpoznať.

Zaujímavým a jednoduchým trikom, ako zmenšiť množstvo spracúvaných „zbytočných“ situácií je začať riešenie hľadať naraz z oboch strán – aj zo začiatočnej situácie, aj z cieľovej.

Ak teda má napríklad optimálne riešenie 31 krokov, nevygenerujeme všetky situácie, ktoré idú dosiahnuť zo začiatočnej na 31 krokov, ale len: (tie, ktoré idú dosiahnuť na 16 krokov) + (tie, z ktorých ide na 15 krokov dosiahnuť cieľ).

Algoritmus A*

Ešte lepšie by bolo, keby náš program vedel robiť to, čo my „od oka“ spravíme ľahko: pozrieť sa na nejakú situáciu a zhodnotiť, že ju nemá zmysel skúšať, lebo vyzerá príliš nanič.

Ako môže takéto niečo vedieť program robiť? Použijeme na to *heuristickú funkciu* h, ktorá bude hovoriť nejaký dolný odhad riešenia. Inými slovami, pre h musí platiť: Ak S je ľubovoľná situácia, tak najlepšie riešenie pre S má aspoň $h(S)$ krokov.

Načo nám je takáto funkcia h dobrá? Pomocou nej môžeme vedieť niektoré situácie rovno zahodiť. Predstavme si napríklad, že už poznáme nejaké riešenie našej úlohy s hodnotou X, a práve sme zistili, že sa zo začiatočnej situácie vieme na $d(S)$ krokov dostať do situácie S. Spočítame si $h(S)$, a ak zistíme, že $d(S) + h(S) \geq X$, tak je pre nás S nezaujímavá – určite nám nepomôže nájsť lepšie riešenie.

Rôznych funkcií, ktoré spĺňajú našu podmienku, je samozrejme veľa. Spĺňa ju napríklad aj funkcia $h(S) = 0$. Tá nám ale príliš nepomôže. Trik je samozrejme v tom, že treba zvoliť funkciu h, ktorá na jednej strane hodnotu riešenia odhadne čo najlepšie, ale na druhej strane ju musíme vedieť efektívne počítať.

Samotný algoritmus A* je v podstate len upravené prehľadávanie do šírky. Zmena bude jednoduchá: Nebudeme používať obyčajnú frontu (z ktorej ako prvú vyberieme situáciu, ktorá tam bola prvá vložená). Namiesto

toho budeme používať prioritnú frontu. Vyberať na spracovanie budeme vždy tú situáciu S, ktorá má najmenší súčet $d(S) + h(S)$. Prestaneme, akonáhle sa prvýkrát dostaneme do cieľa.

Pre ľubovoľnú funkciu h spĺňajúcu našu podmienku, bude algoritmus A* fungovať a nájde najkratšiu cestu do cieľa. Totiž keď sa dostaneme do cieľovej situácie C, pre ľubovoľnú ešte nespracovanú situáciu S platí, že $d(S)+h(S) \geq d(C)+h(C) = d(C)+0 = d(C)$, a teda cez S už lepšiu cestu od práve objavenej určite nevyrobíme.

Pritom platí, že čím lepšiu funkciu h budeme mať, tým menej situácií budeme musieť zobrať do úvahy. Vhodnou voľbou h však vieme dosiahnuť to, že ako prvé pôjde náš algoritmus prezerať tie situácie, ktoré vyzerajú najviac nádejne. Presný dôkaz je komplikovaný, ukážeme si ale aspoň oba okrajové prípady:

Keby sme zobrali $h(S) = 0$, dostali by sme pôvodné prehľadávanie do šírky, kde by sme spracúvali situácie usporiadané podľa vzdialenosti od začiatku.

Naopak, čo by sa stalo, ak by funkcia h bola presná, teda vracala počet krokov potrebný na dosiahnutie cieľa? V takomto prípade by algoritmus A* jednoducho prešiel po najkratšej ceste a skončil. Žiadne odchýlky od nej by neskúšal, keďže pre také situácie by súčet $d(S) + h(S)$ bol väčší.

Heuristická funkcia

V našom prípade môžeme použiť napríklad takúto funkciu h:

Vyskúšame všetkých $P!$ spôsobov, ako priradiť prasiatka senníkom. Pre každú možnosť zoberieme pre každé prasiatko p najkratšiu vzdialenosť d_p od jeho aktuálnej polohy k jeho senníku. Keďže prasiatko vie robiť na jeden krok posun najviac o dve políčka, určite bude potrebovať aspoň $\lceil d_p/2 \rceil$ krokov. Toto sčítame pre všetky prasiatka. Spomedzi všetkých $P!$ spôsobov vyberieme ten, kde nám výsledok vyjde najmenší, a ten vrátime ako hodnotu h.

Najkratšie vzdialenosti medzi každou dvojicou políčok si vieme na začiatku predpočítať (napr. prehľadávaniami do šírky v celkovom čase $O(N^4)$), a teda jedno volanie funkcie h má časovú zložitosť $O(P \cdot P!)$.

Ešte o niečo lepšie je pri počítaní vzdialeností brať do úvahy to, či sa tam vôbec dá skákať. Teda predpočítame si priamo najmenšie počty krokov, ktoré by prasiatko na presun potrebovalo, keby vždy, keď chce skákať, malo na správnom mieste kamaráta.

Listing programu:

```
#include <algorithm>
#include <cstdio>
#include <queue>
#include <map>
using namespace std;

#define SIZE(t) ((int)((t).size()))

#define MAX_P 4
#define MAX_N 22

int N,P; // rozmer mapy a pocet prasiatok
char M[MAX_N+4][MAX_N+4]; // mapa
int dr[] = {-1,1,0,0}, dc[] = {0,0,-1,1}; // smery pohybu
int G[600][600]; // prasacie vzdialenosti medzi polickami mapy

inline int encode(int r, int c) { return (N+3)*r + c; }
inline void decode(int co, int &r, int &c) { c=co%(N+3); r=co/(N+3); }

// spravime si class na pamatanie si situacie
class State {
  public:
  int cells[MAX_P];
  // situacie vieme normalizovat (zoradit prasiatka):
  void normalize() {
    for (int i=0; i<P; i++) for (int j=i+1; j<P; j++)
      if (cells[i]>cells[j]) swap(cells[i],cells[j]);
  }
  // konstruktor z pola suradnic
  State(int *coords) {
    for (int i=0; i<P; i++) cells[i] = encode(coords[2*i],coords[2*i+1]);
    normalize();
  }
  // situacie vieme lexikograficky zoradit
  bool operator<(const State &rhs) const {
    for (int i=0; i<P; i++) {
      if (cells[i] < rhs.cells[i]) return true;
      if (cells[i] > rhs.cells[i]) return false;
    }
    return false;
```

```
  }
};

typedef pair<int,State> Record;

// funkcia ktora vygeneruje mozne tahy zo situacie "kde"
vector<State> generuj_tahy(State kde) {
  vector<State> res;
  int x[2*MAX_P];
  for (int i=0; i<P; i++) decode(kde.cells[i],x[2*i],x[2*i+1]);
  for (int i=0; i<P; i++) M[x[2*i]][x[2*i+1]]='O';

  for (int i=0; i<P; i++) for (int d=0; d<4; d++) {
    x[2*i]+=dr[d]; x[2*i+1]+=dc[d];
    bool two = false;
    if (M[x[2*i]][x[2*i+1]]=='O') { two=true; x[2*i]+=dr[d]; x[2*i+1]+=dc[d]; }

    if (M[x[2*i]][x[2*i+1]]=='.') res.push_back(State(x));

    if (two) { x[2*i]-=dr[d]; x[2*i+1]-=dc[d]; }
    x[2*i]-=dr[d]; x[2*i+1]-=dc[d];
  }

  for (int i=0; i<P; i++) M[x[2*i]][x[2*i+1]]='.';
  return res;
}

// eval() vracia hodnotu -(d(kde)+h(kde))
int eval(int dist, const State &kde, const State &koniec) {
  int x[2*MAX_P], y[2*MAX_P], perm[MAX_P];
  for (int i=0; i<P; i++) decode(koniec.cells[i],x[2*i],x[2*i+1]);
  for (int i=0; i<P; i++) decode(kde.cells[i],y[2*i],y[2*i+1]);
  for (int i=0; i<P; i++) perm[i]=i;
  int bestdist = 987654321;
  do {
    int thisdist = 0;
    for (int i=0; i<P; i++) {
      int a = encode(x[2*i],x[2*i+1]);
      int b = encode(y[2*perm[i]],y[2*perm[i]+1]);
      thisdist += G[a][b];
    }
    bestdist = min(bestdist,thisdist);
  } while (next_permutation(perm,perm+P));
  dist += bestdist;
  return -dist;
}

int main() {
  // nacitame vstup
  scanf("%d ",&N);
  for (int i=0; i<=N+1; i++) for (int j=0; j<=N+1; j++) M[i][j]='#';
  for (int i=1; i<=N; i++) scanf("%s",M[i]+1);
  scanf("%d ",&P);
  int x[2*MAX_P];
  for (int i=0; i<2*P; i++) scanf("%d ",&x[i]);
  State zaciatok(x);
  for (int i=0; i<2*P; i++) scanf("%d ",&x[i]);
  State koniec(x);

  // spocitame prasiatkove vzdialenosti policok na mape
  memset(G,47,sizeof(G));
  for (int i=1; i<=N; i++) for (int j=1; j<=N; j++) {
    if (M[i][j]!='.') continue;
    int x = encode(i,j);
    G[x][x]=0;
    for (int d=0; d<4; d++) {
      int k,l;
      k=i+dr[d], l=j+dc[d];
      if (M[k][l]!='.') continue;
      G[x][encode(k,l)]=1;

      k=i+2*dr[d], l=j+2*dc[d];
      if (M[k][l]!='.') continue;
      G[x][encode(k,l)]=1;
    }
  }
  int TOP = (N+3)*(N+3);
  for (int k=0; k<TOP; k++)
  for (int i=0; i<TOP; i++)
  for (int j=0; j<TOP; j++) G[i][j] = min(G[i][j], G[i][k]+G[k][j]);

  // spustime algoritmus A*
  map<State,int> dist;
  dist[zaciatok] = 0;

  priority_queue<Record> Q;
  Q.push( Record( eval(0,zaciatok,koniec), zaciatok ) );
```

```
  while (!dist.count(koniec)) {
    State kde = Q.top().second; Q.pop();
    vector<State> kam = generuj_tahy(kde);
    for (int i=0; i<SIZE(kam); i++) {
      bool ok = false;
      if (!dist.count(kam[i])) ok = true;
      if (dist.count(kam[i])) if (dist[kam[i]] > dist[kde]+1) ok = true;
      if (!ok) continue;
      dist[kam[i]] = dist[kde] + 1;
      Q.push( Record( eval(dist[kam[i]],kam[i],koniec), kam[i] ) );
    }
  }

  printf("%d\n",dist[koniec]);
  return 0;
}
```

A-III-5 Posledná dobrota

Začneme terminológiou. Stav hry, teda aktuálne počty koláčov a ovocia, budeme volať *pozícia*. *Vyhrávajúca stratégia* je postup, ktorý nám zaručí, že hru vyhráme, bez ohľadu na to, ako bude ťahať protihráč. Pozícia je *vyhrávajúca*, ak hráč, ktorý je práve na ťahu, má vyhrávajúcu stratégiu. Ostatné pozície voláme *prehrávajúce*.

Prezeranie stromu hry

Najjednoduchšie riešenie, ktoré sa dá použiť pre ľubovoľnú konečnú matematickú hru, je založené na dvoch jednoduchých myšlienkach:

- Ak z danej pozície všetky ťahy vedú do vyhrávajúcich pozícií, tak je táto pozícia prehrávajúca.
- Ak z danej pozície existuje ťah do prehrávajúcej, tak je táto pozícia vyhrávajúca.

(Ak všetky ťahy vedú do vyhrávajúcich pozícií, nech si vyberieme ktorýkoľvek, vždy tým dostaneme súpera do vyhrávajúcej pozície. A ak sa potom bude súper držať nejakej vyhrávajúcej stratégie, hru prehráme. Preto takáto pozícia je prehrávajúca. Naopak, ak existuje ťah do prehrávajúcej pozície, spravíme ho, a tým dostaneme súpera do tejto, pre neho prehrávajúcej pozície.)

Túto myšlienku ľahko prepíšeme do rekurzívnej funkcie, ktorá nám o pozícii povie, či je vyhrávajúca alebo prehrávajúca.

Dynamické programovanie / memoizácia

Problém predchádzajúceho prístupu spočíva v tom, že je príliš pomalý. Hlavný dôvod je ten, že pri rekurzívnych volaniach vlastne skúša všetky možné priebehy hry, a pri tom mnohé pozície vyhodnotí veľa krát.

Tu je samozrejme ľahká pomoc – akonáhle o nejakej pozícii zistíme, či je vyhrávajúca, zapíšeme si to do pomocného poľa. Takto dosiahneme to, že každú pozíciu budeme spracúvať práve raz.

Rôznych pozícií je $O(AB)$. Preto je taká aj pamäťová zložitosť tohoto riešenia. Na vyhodnotenie pozície potrebujeme prezrieť všetky možné ťahy, ktorých je $O(K)$. Preto je časová zložitosť riešenia $O(ABK)$.

Na toto riešenie sa môžeme dívať aj z opačnej strany: Keby sme napríklad pozície spracúvali zoradené podľa celkového množstva dobrôt, tak by platilo, že v okamihu, keď vyhodnocujeme nejakú pozíciu, už vieme o všetkých pozíciách, do ktorých môžeme ťahať, či sú vyhrávajúce alebo prehrávajúce.

Šikovnejšie dynamické programovanie

Predchádzajúce riešenie ešte stále robí zbytočne veľa práce, keď skúša všetky možné ťahy. Namiesto toho môžeme využiť skutočnosť, že prehrávajúcich pozícií je málo.

Budeme postupne prechádzať pozície. Vždy, keď narazíme na prehrávajúcu pozíciu (x, y), zaznačíme si o všetkých pozíciách $(x + i, y)$, $(x, y + i)$ aj $(x + i, y + i)$ pre $1 \leq i \leq K$, že sú vyhrávajúce:

```
for (int x=0; x<=A; x++)
  for (int y=0; y<=B; y++)
    if (!vyhravajuca[x][y])
      for (int i=1; i<=K; i++)
        vyhravajuca[x+i][y] = vyhravajuca[x][y+i] = vyhravajuca[x+i][y+i] = true;
```

Zjavne každú pozíciu zmeníme z prehrávajúcej na vyhrávajúcu najviac raz. Preto má tento algoritmus časovú zložitosť $O(AB)$.

Iná formulácia hry

V ďalšom texte riešenia bude lepšie predstaviť si našu hru ináč. Predstavme si, že máme obrovskú šachovnicu, ktorá pokrýva celý prvý kvadrant súradnicovej sústavy. Políčko v ľavom dolnom rohu nech má súradnice $(0, 0)$.

Na políčku (A, B) stojí figúrka. Hráč na ťahu ju môže posunúť nanajvýš o K políčok, a to buď doľava, alebo dodola, alebo šikmo doľava dodola. Vyhráva ten, kto figúrku privedie do ľavého dolného rohu.

Táto hra je identická (presnejšie povedané, *izomorfná*) s hrou, ktorú hrajú Julka a Monika s koláčmi a ovocím. Prvá súradnica figúrky zodpovedá počtu koláčov na stole, druhá počtu kusov ovocia.

Redukcia priestoru stavov

Predchádzajúce riešenie ešte stále nestačí na riešenie úlohy pre limity dané v zadaní. Ďalším krokom môže byť napríklad odpozorovanie nejakej závislosti v tabuľke vyhrávajúcich a prehrávajúcich pozícií. Jednu takúto závislosť si teraz dokážeme:

Tvrdíme, že pozícia (A, B) je vyhrávajúca práve vtedy, keď je vyhrávajúca pozícia $(A \bmod (K + 1), B \bmod (K + 1))$.

Intuícia za týmto tvrdením: Ak je ešte hra ďaleko od konca, vieme doplniť ťah súpera tak, aby sme dokopy odobrali aj koláčov, aj ovocia buď 0 alebo $K + 1$ kusov, a tak zabezpečiť, že sa zvyšok nezmení.

Dôkaz budeme vysvetľovať pre hru s figúrkou na šachovnici. Predstavme si, že šachovnicu rozdelíme na veľké štvorce so stranou $K + 1$, začínajúc samozrejme v ľavom dolnom rohu.

Ukážeme najskôr, že ak je vyhrávajúca pozícia $(A \bmod (K + 1), B \bmod (K + 1))$, tak je vyhrávajúca aj pozícia (A, B).

Uvažujme nasledujúcu stratégiu:

- V prvom ťahu potiahneme rovnako ako by sme ťahali v pozícii $(A \bmod (K + 1), B \bmod (K + 1))$.
- Ak protihráč svojím predchádzajúcim ťahom prekročil hranicu veľkého štvorca, potiahneme tým istým smerom tak, aby sme dokopy posunuli figúrku presne o $K + 1$.
  (Toto určite vieme urobiť, a dosiahneme tým, že figúrka bude na tej istej pozícii, len v inom veľkom štvorci ako bola.)
- V opačnom prípade potiahneme aj my len v rámci veľkého štvorca, a to rovnako ako by sme ťahali v tejto pozícii v ľavom dolnom veľkom štvorci.

Ak sa jej budeme držať, zjavne hru vyhráme.

Podobne sa dá ukázať, že ak je pozícia $(A \bmod (K + 1), B \bmod (K + 1))$ prehrávajúca, tak aj pozícia (A, B) je prehrávajúca – nech začneme ľubovoľným ťahom, súperovi na to, aby vyhral, stačá použiť práve popísanú stratégiu.

Ak použijeme algoritmus z časti „Šikovnejšie dynamické programovanie“, dostaneme riešenie s časovou aj pamäťovou zložitosťou $O(K^2)$.

Vzorové riešenie

Všimnime si, že keďže nás už zaujímajú len pozície od $(0, 0)$ po (K, K), môžeme úplne zabudnúť na to, že máme nejaké obmedzenie na počet vecí, ktoré môžeme v jednom ťahu zobrať. Budeme teda odteraz uvažovať hru, kde toto obmedzenie nie je.

Pre každé x určite existuje najviac jedno y také, že (x, y) je prehrávajúca pozícia. Zoberme totiž najmenšie y také, že $(x.y)$ je prehrávajúca. Potom pre ľubovoľné $z > y$ vieme z (x, z) ťahať do (x, y), a teda (x, z) je vyhrávajúca pozícia. (Dá sa dokonca dokázať, že existuje práve jedno také y.)

Nepotrebujeme teda dvojrozmerné pole pozícii, stačí nám pamätať si v poli pre každé x jemu zodpovedajúce y, ak sme ho už zistili.

Prehrávajúce pozície budeme zostrojovať usporiadané podľa $x + y$. Budeme mať dve polia: v jednom si pamätáme, kde je prehrávajúca pozícia v ktorom riadku, v druhom, kde je na ktorej „uhlopriečke“. (Uhlopriečkami voláme množiny políčok „rovnobežné“ s hlavnou uhlopriečou tabuľky.)

Budeme v cykle prechádzať riadky od 0 po A. Vždy, keď natrafíme na riadok x, v ktorom ešte nepoznáme prehrávajúcu pozíciu, nájdeme ju jednoducho tak, že nájdeme k nemu prvý taký stĺpec y, že ešte nepoznáme prehrávajúcu pozíciu ani v stĺpci y, ani na uhlopriečke idúcej cez (x, y). Stĺpec y stačí hľadať napravo od stĺpca, ktorý sme našli k predchádzajúcemu riadku. Zaznačíme si, že v riadku x je prehrávajúca pozícia y. A zo symetrie vyplýva, že v riadku y je prehrávajúca pozícia x, to si zaznačíme tiež. V oboch prípadoch si príslušný údaj zapíšeme aj pre uhlopriečku, na ktorej práve nájdená pozícia leží.

Takto dostávame riešenie, ktoré všetky potrebné prehrávajúce pozície spočíta v čase $O(K)$. Pomocou nich vieme následne každý ťah spraviť v konštantnom čase.

Listing programu:

```
#include “dobrota.h“
#include <cstdio>
using namespace std;

int prehraR[MAX_K+1];
int prehraU[2*MAX_K+1];
int K;

void zaciatok (int A, int B, int _K) {
  K = _K;
  for (int i=0; i<=2*K; i++) prehraU[i] = -1;
  for (int i=0; i<=K; i++) prehraR[i] = -1;

  int stlpec = 0;
  for (int riadok = 0; riadok <= K; riadok++) {
    if (prehraR[riadok] >= 0) continue;
    while (prehraU[riadok-stlpec+K]>=0 || prehraR[stlpec]>=0) {
      stlpec++;
      if (stlpec > K) break;
    }
    if (stlpec > K) break;
    prehraR[riadok] = stlpec;
    prehraR[stlpec] = riadok;
    prehraU[riadok-stlpec+K] = riadok;
    prehraU[stlpec-riadok+K] = stlpec;
  }
}

void tahaj (int A, int B, int *ber_A, int *ber_B) {
  A %= K+1;
  B %= K+1;
  if (prehraR[A]!=-1 && B > prehraR[A]) { *ber_A = 0; *ber_B = B-prehraR[A]; return; }
  if (prehraR[B]!=-1 && A > prehraR[B]) { *ber_B = 0; *ber_A = A-prehraR[B]; return; }
  if (prehraU[A-B+K]!=-1 && A > prehraU[A-B+K]) { *ber_A = *ber_B = A-prehraU[A-B+K]; return; }
  printf(“zle je!\n“); *ber_A=*ber_B=0; // nemalo by nastat
}
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 40 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX