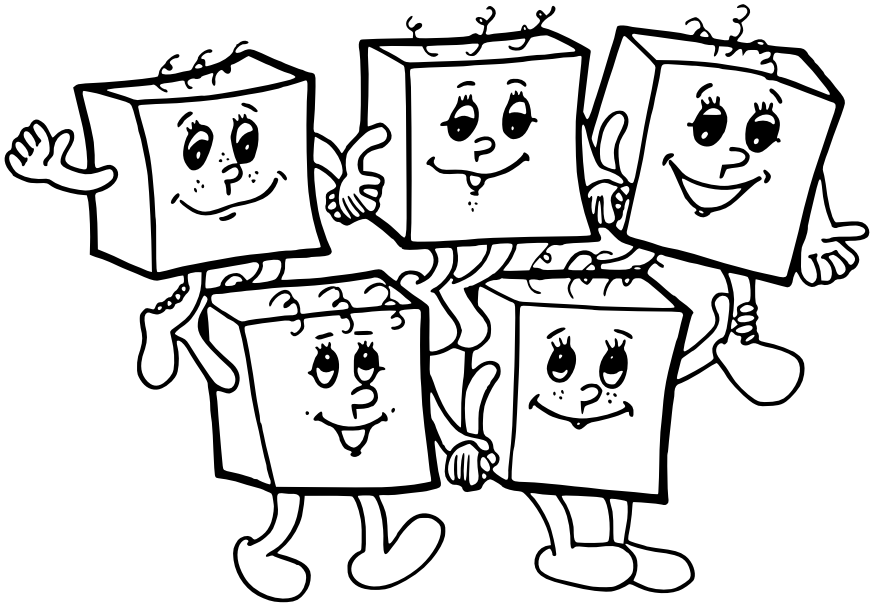
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH



dvadsiaty piaty ročník
školský rok 2009/10

riešenia domáceho kola
kategória A

- **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://oi.sk/`.

Riešenia kategórie A

A-I-1 Maliar Bonifác

Na získanie 3 bodov stačilo robiť presne to, čo mal pôvodne robiť Bonifác: spravíme si K-prvkové pole, kde každý prvok bude predstavovať jeden meter chodníka. Teraz postupne čítame príkazy a zakaždým príslušný úsek chodníka prefarbíme, t. j., do príslušných premenných poľa priradíme novú farbu.

Toto riešenie má pamäťovú zložitosť $O(K)$ a časovú $O(NK)$. Čo môžeme zlepšiť?

Predstavme si, že Bonifác nebude najskôr nič maľovať. Vypočuje si všetky požiadavky, potom zoberie kriedu a pre každú požiadavku spraví na chodník dve čiary – na začiatku a na konci dotyčného úseku. Takto rozdelí chodník na $U \leq 2N + 1$ úsekov. A je zjavné, že každý z úsekov, ktoré takto dostaneme, bude celý jednej farby. Stačí teda pre každý z nich zistiť jeho farbu.

Takto sme sa úspešne zbavili premennej K udávajúcej dĺžku chodníka. Použitému triku sa zvykne hovoriť *kompresia súradníc*. A teraz nám teda stačí spraviť si pole dĺžky U a ofarbovať jeho prvky (predstavujúce jednotlivé úseky chodníka). Takéto riešenie má časovú zložitosť $O(N^2)$ a dalo sa zaň získať 6 bodov.

Ukážeme teraz dva rôzne prístupy, ktoré povedú k rôznym riešeniam za plný počet bodov.

Prvý prístup bude založený na myšlienke, že sa prejdeme po chodníku z jedného konca na druhý a o každom mieste zistíme, akú bude mať na konci farbu.

Druhý prístup bude založený na simulácii požiadaviek v takom poradí, v akom ich Bonifác dostal, ale použijeme lepšiu dátovú štruktúru ako obyčajné pole.

Zametanie

V tomto riešení pôjdeme po chodníku, pričom si budeme v každom okamihu pamätať množinu čísel príkazov, ktoré obsahujú miesto, kde práve sme. Samozrejme, aktuálne miesto má farbu podľa toho z nich, ktorý bol posledný – a teda má najväčšie číslo. A kedy sa množina príkazov zmení? Vtedy, keď prídeme na miesto, kde nejaký príkaz začína alebo končí.

Celé riešenie bude teda vyzerať nasledovne: Do poľa uložíme všetky začiatky a konce príkazov. Toto pole utriedime. Takto rozdelíme chodník na $2N + 1$ úsekov. O každom z nich vieme rovno povedať jeho dĺžku. (Ak na niektorom mieste začínalo alebo končilo viacero príkazov, budú mať niektoré úseky dĺžku nulovú, ale to nám nevadí.) A keď ich budeme spracúvať postupne, vieme si udržiavať množinu aktívnych príkazov.

V našej implementácii sme na uloženie množiny „aktívnych“ príkazov použili `set`. Celé toto riešenie sa však dá ľahko implementovať aj v Pascale – vystačíme si napríklad s dvoma haldami. V jednej budú uložené všetky začiatky a konce príkazov, ktoré ešte treba spracovať, v druhej budú čísla príkazov, ktoré sú momentálne aktívne.

Takto dostávame riešenie s časovou zložitosťou $O(N \log N)$ a pamäťovou zložitosťou $O(N)$.

Listing programu:

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

class udalost {
  public:
    int pozicia, prikaz; bool zacina;
    udalost(int po=0, int pr=0, bool z=true) : pozicia(po), prikaz(pr), zacina(z) {}
};
bool skor(const udalost &a, const udalost &b) { return a.pozicia < b.pozicia; }

int N, F, K;
set<int> aktivne;           // cisla momentalne aktivnych prikazov
vector<udalost> udalosti;   // zaciatky a konce prikazov
vector<int> farby;          // ku kazdej farbe pocet litrov ktore treba
vector<int> farby_prikazov; // ku kazdemu prikazu jeho farba
```

```
int main() {
  // nacitame vstup
  cin >> N >> F >> K;
  for (int i=0; i<N; i++) {
    int z,k,f;
    cin >> z >> k >> f;
    farby_prikazov.push_back(f);
    udalosti.push_back(udalost(z,i,true));
    udalosti.push_back(udalost(k,i,false));
  }

  // utriedime udalosti
  sort( udalosti.begin(), udalosti.end(), skor );

  // spracujeme udalosti
  farby.resize(F+1,0);
  aktivne.insert( udalosti[0].prikaz );
  for (int i=1; i<2*N; i++) {
    int farba = 0, dlzka = udalosti[i].pozicia - udalosti[i-1].pozicia;
    if (!aktivne.empty()) farba = farby_prikazov[ *aktivne.rbegin() ];
    farby[farba] += dlzka;
    if (udalosti[i].zacina) aktivne.insert( udalosti[i].prikaz );
    else aktivne.erase( udalosti[i].prikaz );
  }

  // vypiseme vysledok
  for (int f=1; f<=F; f++) cout << farby[f] << endl;
}
```

Simulácia – prvá možnosť

Potrebujeme teraz vymyslieť efektívny spôsob, ako si pamätať ofarbenie chodníka – ešte lepší ako ten v 6-bodovom riešení. Existuje hneď viacero možností, ako na to, ale všetky majú spoločné jednu vec – potrebujeme sa vedieť efektívne „zbaviť“ úsekov, ktoré práve spracúvaným príkazom celé prekryjeme.

Ak máme k dispozícii vyvažovaný binárny strom (napr. `set` v STL), nepotrebujeme ani strácať čas kompresiou súradníc. Stačí si jednoducho v strome pamätať všetky jednofarebné úseky aktuálne ofarbeného chodníka, utriedené podľa začiatku.

Napríklad pre $K = 20$ budeme po spracovaní prikazov „od 3 po 7 farbou 1“ a „od 4 po 5 farbou 2“ mať päť jednofarebných úsekov: od 0 po 3 farbou 0, od 3 po 4 farbou 1, atď. Pre každý z týchto úsekov by sme mali jeden záznam v strome.

Ako teraz spracujeme novú požiadavku? Začneme tým, že nájdeme v strome posledný úsek, ktorý začína skôr ako ona, a prvý úsek, ktorý neskôr končí. (Toto vieme spraviť v $O(\log N)$, keďže náš strom je vyvážený a má $O(N)$ vrcholov. V `set`e na to slúži napr. metóda `lower_bound`.)

Oba tieto úseky skrátime po hranicu novej požiadavky. (Pozor, mohlo sa stať, že ide o ten istý úsek, ktorý sa nám týmto rozdelil na dva.) Následne zo stromu postupne vyhádžeme všetky úseky medzi nimi – tie naša nová požiadavka celé prekryje – a úplne na záver vložíme do stromu úsek zodpovedajúci novej požiadavke.

Mohlo by sa zdať, že takéto riešenie nemusí byť efektívne – môže sa predsa stať, že tých prekrytých intervalov bude niekedy strašne veľa, nie?

Môže sa to stať, ale nie často. Trik je v tom, že dokopy do stromu vložíme $O(N)$ intervalov, a vyhodiť ich môžeme len toľko, koľko sme ich predtým vložili. Preto dokopy spravíme s naším stromom $O(N)$ operácií. A keďže každú potrebnú operáciu vie vyvažovaný strom spraviť v čase $O(\log N)$, má aj toto riešenie časovú zložitosť $O(N \log N)$.

Listing programu:

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

class zaznam {
  public:
    int zac, kon, farba;
    zaznam(int zac=0, int kon=0, int farba=0) : zac(zac), kon(kon), farba(farba) {}
};
```

```
bool operator< (const zaznam &a, const zaznam &b) { return a.zac < b.zac; }

int N, F, K;
set<zaznam> S;

int main() {
  cin >> N >> F >> K;
  S.insert(zaznam(-1,K+1,0));
  while (N--) {
    zaznam cur, next;
    cin >> cur.zac >> cur.kon >> cur.farba;

    // spracuje interval obsahujuci lavy koniec poziadavky
    set<zaznam>::iterator left = S.lower_bound(cur);
    left--;
    if (left->kon > cur.kon) S.insert(zaznam(cur.kon,left->kon,left->farba));
    if (left->kon > cur.zac) {
      next = zaznam(left->zac,cur.zac,left->farba);
      S.erase(left);
      S.insert(next);
    }

    // pravy koniec poziadavky
    next = zaznam(cur.kon,K+1,0);
    set<zaznam>::iterator right = S.upper_bound(next);
    right--;
    if (right->zac < cur.kon) { S.insert(zaznam(cur.kon,right->kon,right->farba)); S.erase(*right);}

    // vymaze vsetky intervaly medzi nimi
    left = S.lower_bound(cur);
    right = S.upper_bound(next); right--;
    S.erase(left,right);

    // a vlozi novy
    S.insert(cur);
  }
  vector<int> farby(F+1,0);
  for (set<zaznam>::iterator it=S.begin(); it != S.end(); ++it)
    farby[ it->farba ] += (it->kon)-(it->zac);
  for (int f=1; f<=F; f++) cout << farby[f] << endl;
}
```

Simulácia – druhá možnosť

(Riešenie podľa Tomáša Belana)

Predchádzajúce riešenie vieme pomocou STL (alebo podobnej sady knižníc v inom jazyku) implementovať ešte jednoduchšie. Namiesto toho, aby sme si pamätali celé intervaly, budeme si pamätať len usporiadanú množinu záznamov tvaru „na pozícii x začína úsek farby f“. Na toto použijeme dátovú štruktúru `map<int,int> zaciatky`, pričom záznam (x, f) uložíme príkazom `zaciatky[x]=f;`.

Čo presne musíme spraviť, keď nám príde nový interval (x, y, f)? V prvom rade zistíme, akej farby bude interval, ktorý bude od tejto chvíle začínať na pozícii y. Toto spravíme tak, že pomocou metódy `upper_bound` nájdeme posledný záznam v mape `zaciatky`, ktorého kľúč je menší alebo rovný ako y. Jeho farba a odteraz bude začínať na pozícii y. Teraz do mapy `zaciatky` uložíme záznamy (x, f) a (y, a). A na záver pomocou metódy `erase` zmažeme všetky záznamy, ktoré už nie sú aktuálne – teda tie s kľúčmi medzi x a y.

Pre každý interval vložíme do mapy dva nové záznamy, a každý záznam z mapy najviac raz vymažeme. Každá z týchto operácií prebehne v čase $O(\log N)$, preto spracovanie všetkých záznamov má časovú zložitosť $O(N \log N)$. Po spracovaní posledného záznamu už len v lineárnom čase prejdeme cez všetky záznamy, ktoré máme na konci v mape, a spočítame odpoveď. Výsledná implementácia je až neuveriteľne stručná:

Listing programu:

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;

int N, F, K, zac, kon, farba;
map<int, int> zaciatky;
vector<int> farby;
```

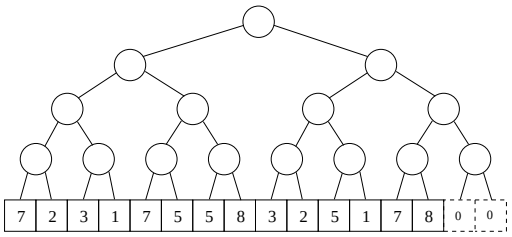
```
int main() {
  cin >> N >> F >> K;
  zaciatky[0] = zaciatky[K] = 0;
  while (N--) {
    cin >> zac >> kon >> farba;
    int after = (--zaciatky.upper_bound(kon))->second;
    zaciatky[zac] = farba;
    zaciatky[kon] = after;
    zaciatky.erase( zaciatky.upper_bound(zac), zaciatky.lower_bound(kon) );
  }

  farby.resize(F+1,0);
  map<int,int>::iterator it = zaciatky.begin();
  while (it->first < K) {
    farba = it->second, zac = it->first, kon = (++it)->first;
    farby[farba] += kon-zac;
  }
  for (int f=1; f<=F; f++) cout << farby[f] << endl;
}
```

Simulácia – tretia možnosť

Na záver si ukážeme dátovú štruktúru, ktorá sa dá rozumne ľahko implementovať aj v Pascale a umožní nám robiť simuláciu rovnako efektívne.

Začneme tým, že rovnako ako v 6-bodovom riešení spravíme kompresiu súradníc, čím rozdelíme chodník na U úsekov. Pre jednoduchosť budeme predpokladať, že U je mocnina dvoch. (Ak by nebolo, zväčšíme ho na najbližšiu mocninu dvoch. Uvedomte si, že tým sa zväčší menej ako na dvojnásobok pôvodnej hodnoty, a teda naďalej $U = O(N)$.)

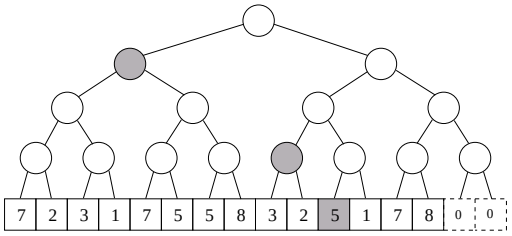
Použijeme dátovú štruktúru známu pod menom *intervalový strom*. Ten bude vyzerať takto:



Listy intervalového stromu zodpovedajú jednotlivým úsekom chodníka a budeme si v nich samozrejme ukladať ich farby. Všimnite si, že vnútorný vrchol, ktorý je k úrovní nad listami, zodpovedá intervalu obsahujúcemu 2^k po sebe idúcich úsekov. Tie intervaly, ktoré zodpovedajú vrcholom nášho stromu, budeme volať *jednoduché*.

Načo je intervalový strom dobrý? Ukážeme, že ľubovoľný interval úsekov vieme šikovne „poskladať“ z jednoduchých intervalov.

Zaoberajme sa najskôr intervalom, ktorý obsahuje úseky od 1 po k. Tvrdíme, že tento vieme zložiť z najviac $\lg N$ jednoduchých intervalov. Toto dokážeme tak, že budeme z jeho ľavej strany odkrajovať čo najväčšie jednoduché intervaly, až kým sa neminie. Najjednoduchšie je to vidieť na príklade. Napr. interval „od 1 po 11“ vieme rozdeliť na „od 1 po 8“, „od 9 po 10“ a „od 11 po 11“.

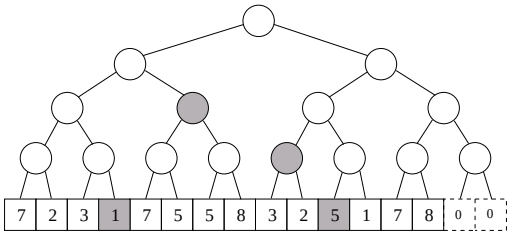


Vyznačené vrcholy zodpovedajú jednoduchým intervalom,
ktoré dokopy tvoria interval „od 1 po 11“.

Odhad počtu použitých intervalov vyplýva napríklad z toho, že v každom kroku odkrojíme viac ako polovicu intervalu.

Všimnite si, že postup krájania zodpovedá schádzaniu z koreňa dole po intervalovom strome. V každom vrchole sa pozrieme, či nerozkrájaná časť zadaného intervalu leží celá v ľavom podstrome. Ak áno, nič nekrájame a zlezieme doň. Ak nie, odkrojíme interval zodpovedajúci ľavému podstromu a zlezieme do pravého.

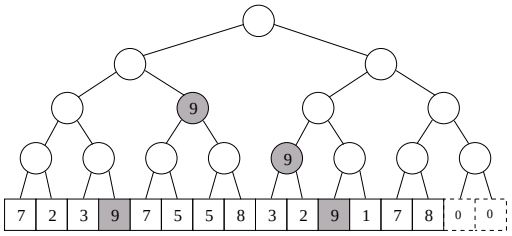
Vo všeobecnej situácii, keď chceme naskladať interval úsekov od k po l, budeme na tom podobne, vystačíme si s $2 \lg N$ jednoduchými intervalmi. Dôkaz je podobný, pôjdeme dole intervalovým stromom. Akonáhle niekedy zistíme, že zadaný interval zasahuje do oboch podstromov, rozkrojíme ho na dve časti. No a každá z častí už teraz zodpovedá jednoduchšiemu prípadu, ktorý sme rozobrali vyššie.



Všeobecný prípad: interval „od 4 po 11“.

Ukázali sme teda, že v čase $O(\log N)$ vieme ľubovoľný interval rozdeliť na niekoľko častí, ktoré zodpovedajú vrcholom stromu.

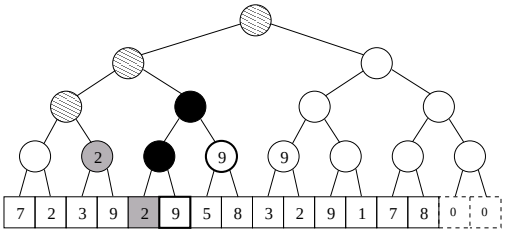
Načo to bude dobré? Vnútorné vrcholy stromu použijeme na to, aby sme vedeli rýchlo simulovať Bonifácove požiadavky. Dohodneme sa, že ak je vo vnútornom vrchole stromu iná hodnota ako nula, znamená to, že celý zodpovedajúci interval má príslušnú farbu – bez ohľadu na to, aké hodnoty sú uložené vo vrcholoch v tomto podstrome. Toto nám umožní spracovať každú požiadavku v čase $O(\log N)$: nájdeme v strome vrcholy zodpovedajúce dotyčnému intervalu a zaznačíme si do nich príslušnú farbu.



Stav po prefarbení intervalu „od 4 po 11“ na farbu 9. V prázdnych vrcholoch sú nuly.

Potrebujeme si ešte rozmyslieť, čo presne sa stane, ak počas spracúvania požiadavky prechádzame v strome cez vrchol, ktorý je momentálne celý zafarbený. To, že cez tento vrchol prechádzame, znamená, že časť jeho podstromu ideme prefarbiť. Preto odteraz bude v tomto vrchole nula – už nebude jednofarebný. Namiesto toho (skôr, ako sa pohneme hlbšie) zafarbíme jeho farbou oba vrcholy pod ním.

Na nasledujúcom obrázku je znázornené, ako sa zmenia údaje v strome z predchádzajúceho obrázku po spracovaní požiadavky „prefarbi interval od 3 po 5 na farbu 2“. Šedé pozadie majú, rovnako ako na predchádzajúcich obrázkoch, vrcholy zodpovedajúce aktuálnej požiadavke. Šrafované vrcholy sú tie, ktoré počas spracúvania požiadavky navštívime, ale nič sa v nich neudeje. Zaujímavé veci sa začnú diať, ked dorazíme k hornému čiernemu vrcholu, v ktorom bola doteraz hodnota 9. Tú presunieme do jeho synov, a následne to isté ešte raz zopakujeme v druhom čiernom vrchole. Oba čierne vrcholy teda teraz obsahujú nuly. Hodnota 9 sa presunula z horného čierneho vrchola do dvoch hrubo orámovaných vrcholov. Všimnite si, že tieto presne zodpovedajú tej časti pôvodného intervalu, ktorú sme neprefarbili.



Kompresiu súradníc vieme spraviť v čase $O(N \log N)$, napríklad pomocou triedenia a binárneho vyhľadávania. (Tak to robíme v nižšie uvedenom programe.) Úvodné nastavenie hodnôt v strome spravíme v čase $O(N)$. Každý z N príkazov spracujeme v čase $O(\log N)$, a na záver v čase $O(N)$ prejdeme strom a spočítame výsledok. Celková časová zložitosť tohto riešenia je teda $O(N \log N)$ a pamäťová zložitosť je $O(N)$.

(Náš intervalový strom má menej ako $4N$ listov, a každá vyššia vrstva má polovičný počet vrcholov oproti tej bezprostredne pod ňou. Preto náš intervalový strom obsahuje nanajvýš $8N$ záznamov, a teda je pamäťová zložitosť skutočne lineárna.)

Listing programu:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int N, F, K, D, L;
struct poziadavka { int zac, kon, farba; };
struct vrchol { int farba, dlzka; };
vector<poziadavka> P; // pole s poziadavkami
vector<vrchol> T; // intervalovy strom
vector<int> farby;

// nacitanie vstupu
void load() {
  cin >> N >> F >> K;
  poziadavka tmp;
  for (int i=0; i<N; i++) {
    cin >> tmp.zac >> tmp.kon >> tmp.farba;
    P.push_back(tmp);
  }
}

// najdenie indexu prvku x v utriedenom poli
int najdi(int x, const vector<int> &pole) {
  int lo=0, hi=pole.size();
  while (hi-lo > 1) { int med=(hi+lo)/2; if (pole[med]<=x) lo=med; else hi=med; }
  return lo;
}

void build_tree() {
  // utriedime vsetky hranice poziadaviek
  vector<int> hranice;
  hranice.push_back(0); hranice.push_back(K);
  for (int i=0; i<N; i++) { hranice.push_back( P[i].zac ); hranice.push_back( P[i].kon ); }
  sort( hranice.begin(), hranice.end() );
  // vyhadzeme duplikaty
  D=1;
  for (int i=1; i<2*N+2; i++) if (hranice[i]!=hranice[i-1]) swap(hranice[i],hranice[D++]);
  hranice.resize(D);
  D--;
  // vyrobime strom
  for (int i=1; ; i*=2) if (i>D+7) { L=i; break; }
  T.resize(2*L);
  for (int i=0; i<D; i++) T[L+i].dlzka = hranice[i+1]-hranice[i];
  for (int i=L-1; i>=1; i--) T[i].dlzka = T[2*i].dlzka + T[2*i+1].dlzka;
  // prepiseme zaciatky a konce poziadaviek do noveho cislovania
  for (int i=0; i<N; i++) {
    P[i].zac = najdi( P[i].zac, hranice );
    P[i].kon = najdi( P[i].kon, hranice );
  }
}
```

```
void color(int x, int kde=1, int left=0, int length=L) {
  if (P[x].kon <= left || P[x].zac>=left+length) return; // mimo
  if (P[x].zac <= left && left+length <= P[x].kon) { T[kde].farba=P[x].farba; return; } // dnu
  if (kde<L && T[kde].farba!=0) {
    T[2*kde].farba=T[2*kde+1].farba=T[kde].farba;
    T[kde].farba=0;
  }
  color(x,2*kde,left,length/2);
  color(x,2*kde+1,left+length/2,length/2);
}

void evaluate(int kde) {
  if (kde>=L || T[kde].farba>0) farby[ T[kde].farba ] += T[kde].dlzka;
  else { evaluate(2*kde); evaluate(2*kde+1); }
}

int main() {
  load();
  build_tree();
  for (int i=0; i<N; i++) color(i);
  farby.resize(F+1,0);
  evaluate(1);
  for (int f=1; f<=F; f++) cout << farby[f] << endl;
}
```

A-I-2 Čokoláda

Najskôr ukážeme riešenie s časovou zložitosťou $O(R^2S)$. Bude založené na jednoduchej myšlienke: vyskúšame všetky dvojice riadkov, a pre každú dvojicu v $O(S)$ spočítame všetky štvorce, ktoré práve tam majú svoj horný a dolný riadok.

Keď sme si už zvolili horný a dolný riadok, máme pás políčok. Niektoré jeho stĺpce sú celé, tie môžeme použiť. A niektoré obsahujú aspoň jedno chýbajúce políčko, a tie použiť nemôžeme.

Ak by sme vedeli, ktoré stĺpce použiť môžeme, a ktoré nie, dostávame nasledujúcu úlohu: Máme dané číslo K (dĺžku strany štvorca) a pole $A[1..S]$ obsahujúce len nuly a jednotky (zlé a dobré stĺpce). Koľko existuje v poli A úsekov dĺžky K, ktoré obsahujú samé jednotky?

Túto úlohu vieme ľahko vyriešiť v lineárnom čase od dĺžky poľa A. Existuje viacero spôsobov, ukážeme jeden z nich. Všimneme si, že úsek je dobrý práve vtedy, ak je jeho súčet rovný K. Spravíme si nové pole $B[0..S]$, kde $B[0] = 0$ a pre všetky $i > 0$ je $B[i] = A[i] + B[i-1]$. Zjavne platí, že $B[i]$ je súčet prvých i prvkov poľa A. (Hodnoty v poli B voláme *prefixovými sumami* poľa A.)

Potom ale úsek, ktorý začína na pozícii p, je dobrý práve vtedy, ak $B[p+K-1] - B[p-1] = K$. (Rozmyslite si, že hodnota $B[p+K-1] - B[p-1]$ predstavuje súčet prvkov poľa A na pozíciách p až $p+K-1$.)

Pole B vieme spočítať v čase $O(S)$, a následne vieme o každom z $S-K+1$ úsekov dĺžky K v konštantnom čase zistiť jeho súčet, a podľa toho povedať, či je tento úsek dobrý alebo nie. Celkovo teda našu jednorozmernú úlohu vieme riešiť v čase $O(S)$.

Zostáva doriešiť, odkiaľ vezmeme informáciu o tom, ktoré stĺpce môžeme použiť a ktoré nie. Na to nám stačí spracúvať dvojice riadkov v systematickom poradí. Pre dvojicu (r_1, r_1) túto informáciu máme „zadarmo“ priamo vo vstupe. A keď pre dvojicu (r_1, r_2) vieme, ktoré stĺpce sa ešte dajú použiť, pre dvojicu (r_1, r_2+1) túto informáciu ľahko zistíme – sú to tie stĺpce, ktoré sa dali použiť pre (r_1, r_2), a zároveň majú jednotku aj v riadku r_2+1.

Listing programu:

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

int R, S;
int A[5012][5012];
int zije[5012], sucet[5012];

int main() {
  scanf("%d %d ",&R,&S);
  for (int r=0; r<R; r++) for (int s=0; s<S; s++) scanf("%d",&A[r][s+1]);
  long long result = 0;
```

```
  for (int r1=0; r1<R; r1++) {
    for (int s=1; s<=S; s++) zije[s]=1;
    for (int r2=r1; r2<R; r2++) {
      for (int s=1; s<=S; s++) zije[s] &= A[r2][s];
      sucet[0]=0;
      for (int s=1; s<=S; s++) sucet[s]=sucet[s-1]+zije[s];
      for (int d=r2-r1+1, zac=1; zac+d-1<=S; zac++)
        if (sucet[zac+d-1]-sucet[zac-1] == d)
          result++;
    }
  }
  printf("%Ld\n",result);
  return 0;
}
```

Lepšie riešenie

Podobný trik ako sme robili pri jednorozmernej úlohe v predchádzajúcom riešení vieme spraviť aj v dvojrozmernom prípade. Začneme teda tým, že pre každý riadok aj pre každý stĺpec zadanej matice si predpočítame prefixové sumy. Vďaka nim vieme o ľubovoľnom kuse riadku alebo stĺpca v konštantnom čase povedať, či je celý dobrý. Na toto predpočítanie nám stačí čas $O(RS)$.

Teraz postupne pre každé políčko zodpovieme otázku: „Aký najväčší štvorec má pravý dolný roh na tomto políčku?“

Ak je na danom políčku 0, odpoveď je zjavne 0, inak je odpoveď aspoň 1. Budeme teraz postupne zväčšovať veľkosť strany štvorca, až kým „nenarazíme“ – nezistíme, že už náš štvorec obsahuje nejakú dieru, prípadne prekročil hranice pôvodného obdĺžnika.

Predstavme si, že skúmame pravý dolný roh na súradniciach (r, s), a už vieme, že tam má pravý dolný roh štvorec veľkosti K. Ako zistiť, či tam máme aj štvorec veľkosti $K + 1$? Jednoducho – je tam práve vtedy, ak sú v stĺpci $s - K$ dobré všetky políčka od $r - K$ po r, a zároveň v riadku $r - K$ musia byť dobré všetky políčka od $s - K$ po s. Obe veci vieme overiť v konštantnom čase vďaka predpočítaným prefixovým sumám.

Takto dostávame riešenie, ktorého časová zložitosť je $O(RS + X)$, kde X je počet štvorcov vo vstupe. V najhoršom prípade bude toto riešenie rovnako rýchle ako to predchádzajúce, ale na „riedkych“ vstupoch (s veľa dierami) bude výrazne rýchlejšie.

Vzorové riešenie

Vidíme, že ak chceme lepšiu časovú zložitosť, nesmieme štvorce počítať po jednom.

Použijeme podobný prístup ako v predchádzajúcom riešení. Nech $K(r, s)$ je veľkosť najväčšieho plného štvorca, ktorý má pravý dolný roh na políčku (r, s). Potom hľadanú odpoveď získame jednoducho ako súčet hodnôt $K(r, s)$ pre všetky prípustné r a s.

Ukážeme teraz, ako hodnoty $K(r, s)$ šikovne spočítať. Ak je na políčku (r, s) diera, tak zjavne $K(r, s) = 0$. Predpokladajme teda, že na (r, s) diera nie je.

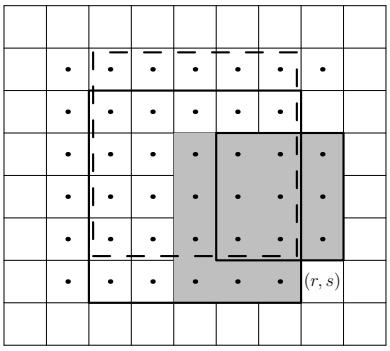
V prvom rade si uvedomme, že platia nasledujúce nerovnosti:

- $K(r, s) \leq K(r - 1, s) + 1$
- $K(r, s) \leq K(r, s - 1) + 1$
- $K(r, s) \leq K(r - 1, s - 1) + 1$

Totiž ak má na (r, s) pravý dolný roh štvorec veľkosti x, tak keď zoberieme štvorec veľkosti $x - 1$ s pravým dolným rohom na niektorom z políčok $(r - 1, s)$, $(r, s - 1)$, alebo $(r - 1, s - 1)$, tak tento menší štvorec bude celý vo vnútri v tom pôvodnom – a teda bude určite dobrý.

Dostávame teda, že platí: $K(r, s) \leq 1 + \min(K(r - 1, s), K(r, s - 1), K(r - 1, s - 1))$.

Dokážeme teraz, že v skutočnosti v predchádzajúcom vzťahu vždy platí rovnosť. Nech totiž $\min(K(r - 1, s), K(r, s - 1), K(r - 1, s - 1)) = x$. Keď zoberieme zjednotenie štvorcov, ktoré majú veľkosť x a pravý dolný roh na týchto troch políčkach, dostaneme presne štvorec veľkosti $x + 1$ s rohom na (r, s) – s jedinou výnimkou, ktorou je samotné políčko (r, s). O tom sme ale predpokladali, že je dobré, a teda naozaj máme štvorec veľkosti $x + 1$.



Príklad situácie z dôkazu. Bodkami sú vyznačené dobré políčka. Máme $K(r-1,s) = 3$ a $K(r,s-1) = K(r-1,s-1) = 5$. Prvé dva zodpovedajúce štvorce sú vyznačené hrubou čiarou, tretí čiarkovanou. Dostávame $x = \min(3,5,5) = 3$. Zjednotenie príslušných troch štvorcov 3×3 je vyplnené. Všimnite si, že spolu s políčkom (r,s) dostávame štvorec 4×4.

Každú hodnotu $K(r,s)$ teda vieme spočítať v konštantnom čase z predchádzajúcich hodnôt. Časová zložitosť tohto riešenia je teda $O(RS)$. Za zmienku ešte stojí, že pamäťová zložitosť sa dá zredukovať na $O(R)$, keďže nám vždy stačí pamätať si hodnoty K pre predchádzajúci a aktuálny riadok.

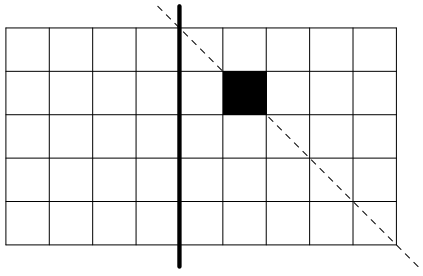
Listing programu:

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

int R, S, A;
long long result = 0;
int K[2][5012];

int main() {
  scanf("%d%d",&R,&S);
  for (int r=1, cur=0; r<=R; r++, cur=1-cur)
    for (int s=1; s<=S; s++) {
      scanf("%d",&A);
      if (A==0) K[cur][s]=0; else K[cur][s]=1+min(K[cur][s-1],min(K[1-cur][s],K[1-cur][s-1]));
      result += K[cur][s];
    }
  printf("%Ld\n",result);
  return 0;
}
```

A-I-3 Koláč

Podúloha a)

Hru vyhrá Marienka. V prvom ťahu spraví rez po priamke $x = 4$ (na obrázku hrubou čiarou). Takto vyrobí štvorec 5×5, ktorý je, vrátane polohy ropuchy, symetrický vzhľadom na uhlopriečku (na obrázku čiarkovane).



Od tohto okamihu bude Marienka ťahať symetricky s Jankom – nech ten spraví rez, aký chce, Marienka spraví jeho zrkadlový obraz (podľa čiarkovanej osi). Teda ak Janko reže vodorovne, tak Marienka zvisle a naopak. Marienka zjavne vždy bude môcť spraviť svoj ťah, a vždy po jej ťahu zostane koláč symetrický. Preto nutne tým, kto už nebude môcť rezať, bude Janko.

Podúloha b)

Riešenie začneme tým, že vysvetlíme základné pojmy z teórie kombinatorických hier. Stav hry, teda aktuálne rozmery obdĺžnika a polohu ropuchy na ňom, 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 kombinatorickú 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.

Koľko je rôznych pozícii, do ktorých sa hra môže dostať? Každá pozícia je určená štyrmi súradnicami: ľavý, pravý, horný a dolný okraj aktuálneho obdĺžnika. Ľavý okraj musí ležať medzi 0 a r_x, vrátane, pravý medzi $r_x + 1$ a X, analogicky pre zvyšné dva.

Dokopy máme teda $(r_x + 1)(X - r_x)(r_y + 1)(Y - r_y)$ pozícii, čo je $O(X^2Y^2)$. Taká bude aj pamäťová zložitosť nášho riešenia. Možných ťahov je v každej pozícii $O(X + Y)$, takže časová zložitosť tohto riešenia je $O(X^2Y^2(X + Y))$.

Na toto riešenie sa môžeme dívať aj z opačnej strany: Keby sme napríklad pozície spracúvali zoradené podľa plochy, tak by platilo, že v okamihu, keď vyhodnocujeme nejakú pozíciu, už vieme o všetkých pozíciách, do ktorých môžeme ťahať, či sú vyhrávajúce alebo prehrávajúce. Takto implementované riešenie má rovnakú časovú aj pamäťovú zložitosť ako to predchádzajúce.

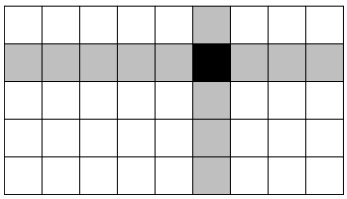
Listing programu:

```
#include <iostream>
#include <cstring>
using namespace std;

int X,Y,rx,ry;
int memo[50][50][50][50];

int winning(int x1, int y1, int x2, int y2) {
  // ak uz sme tuto poziciu riesili, rovno vrat hodnotu
  if (memo[x1][y1][x2][y2] >= 0) return memo[x1][y1][x2][y2];
  // ak nie, inicializuj ju na prehravajucu
  memo[x1][y1][x2][y2]=0;
  // a ak najdes tah do prehravajucej, zmen sucasnu na vyhravajucu
```

```
  for (int x=x1+1; x<x2; x++) {
    int nx1, nx2;
    if (x<=rx) nx1=x, nx2=x2; else nx1=x1, nx2=x;
    if (!winning(nx1,y1,nx2,y2)) return memo[x1][y1][x2][y2]=1;
  }
  for (int y=y1+1; y<y2; y++) {
    int ny1, ny2;
    if (y<=ry) ny1=y, ny2=y2; else ny1=y1, ny2=y;
    if (!winning(x1,ny1,x2,ny2)) return memo[x1][y1][x2][y2]=1;
  }
  return 0;
}

int main() {
  cin >> X >> Y >> rx >> ry;
  memset(memo,-1,sizeof(memo));
  cout << (winning(0,0,X,Y) ? "Marienka" : "Janko") << " vyhra." << endl;
}
```

Optimálne riešenie

Všimnime si štyri pásiky políčok, ktoré sú na nasledujúcom obrázku šedé. Bez ohľadu na to, ako budeme rezať, vždy skrátime práve jeden z nich. A naopak, ak si povieme, ktorý pásik a o koľko chceme skrátiť, vždy vieme taký rez urobiť.[1] Každú pozíciu môžeme teda popísať dĺžkami týchto štyroch pásikov (a, b, c, d).



```
1 =     1
3 =    11
5 =   101
3 =    11
```

Dokážeme teraz nasledujúce tvrdenie: Majme pozíciu (a, b, c, d). Zoberme čísla a, b, c, d, preveďme ich do dvojkovej sústavy a zapíšme pod seba. Tvrdíme, že pozícia je prehrávajúca práve vtedy, ak je v každom stĺpci párny počet jednotiek.[2]

Ak toto chceme dokázať, potrebujeme ukázať, že platia obe vlastnosti, ktoré majú prehrávajúce pozície mať. Presnejšie, musíme ukázať, že:

- Z pozície, kde je v každom stĺpci párny počet jednotiek, každý ťah vedie do pozície, kde to neplatí.
- V každej inej pozícii existuje ťah, po ktorom bude v každom stĺpci párny počet jednotiek.

Prvé tvrdenie očividne platí. Ak spravíme ťah, zmeníme tým práve jedno z čísel. Keď sa teraz pozrieme na binárne zápisy našich čísel, vidíme, že sa nám zmenil práve jeden riadok. Vyberme si niektorý bit v ňom, ktorý sa zmenil. Potom v jeho stĺpci sa nutne zmenila parita počtu jednotiek, a teda je ich tam teraz nepárny počet.

Druhé tvrdenie dokážeme nasledovne: Všimnime si najľavejší stĺpec, kde je nepárny počet jednotiek. Vyberme si ľubovoľný riadok, ktorý má v tomto stĺpci jednotku. Tú zmeníme na nulu. Následne zmažeme zvyšok tohto riadku a znova ho dopočítame tak, aby v každom stĺpci bol párny počet jednotiek. Keďže najľavejší bit, ktorý sme menili, sme zmenili z jednotky na nulu, bez ohľadu na to, ako sme menili ostatné bity, hodnotu čísla sme zmenšili, a teda ide o platný ťah.

Ukážeme si to celé na príklade. Majme pozíciu $(58, 9, 43, 22)$. Keď si ju zapíšeme v dvojkovej sústave, zistíme, že v 4., 3. a 2. stĺpci sprava je nepárny počet jednotiek. Vyberieme si číslo 43, ktoré má v stĺpci 4 jednotku, a túto zmeníme na nulu. Následne dopočítame hodnoty ostatných bitov (tieto bity sú v strednom stĺpci dočasne nahradené bodkami).

```
       v
58 = 111010        111010        111010 = 58
 9 =   1001          1001          1001 =  9
43 = 101011        100...        100101 = 37
22 =  10110         10110         10110 = 22
```

[1] Pre tých, čo už sú v kombinatorickej teórii hier doma: Týmto sme práve dokázali, že naša hra je izomorfná so 4-kopovým NIMom, pričom dĺžky pásikov zodpovedajú veľkostiam kôp.

[2] Inými slovami, vtedy, keď $a \oplus b \oplus c \oplus d = 0$, kde $\oplus$ je bitový xor.

Ak teda v začiatočnej pozícii našej hry platí, že v niektorom stĺpci binárnych zápisov dĺžok pásikov je nepárny počet jednotiek, hru vyhrá Marienka. Stačí jej vždy nájsť a spraviť taký ťah, po ktorom bude všade počet jednotiek párny. Už sme ukázali, že taký ťah vždy existuje. A naopak Janko bude vždy na ťahu v pozícii, v ktorej je všade počet jednotiek párny, a nech ťahá, ako chce, vždy toto poruší.

Keďže v koncovej pozícii sú všetky dĺžky pásikov rovné nule, znamená to, že v nej je všade počet jednotiek párny, a teda hráčom na ťahu (ktorý práve prehral) je Janko.

Naopak, ak je pre začiatočnú pozíciu v každom stĺpci binárnych zápisov dĺžok pásikov párny počet jednotiek, hru vyhrá Janko – po tom, ako Marienka spraví prvý ťah, bude Janko v pozícii s nepárnym počtom jednotiek v niektorom stĺpci a môže použiť tú istú stratégiu ako v opačnom prípade mala Marienka.

Práve sme teda ukázali riešenie, ktoré dokáže o ľubovoľnej pozícii v konštantnom čase povedať, či je vyhrávajúca. (Rozmyslite si, že dokonca vieme pre vyhrávajúcu pozíciu nájsť v konštantnom čase všetky možné vyhrávajúce ťahy.)

A-I-4 Počítač s gumenou rúrou

Podúloha a)

Ak celé číslo N nie je prvočíslo, tak má nejakého deliteľa d takého, že $1 < d < N$. A nie len to. Číslo $e = N/d$ je celé, a tiež je deliteľom N (lebo $N/e = d$, pochopiteľne). Navyše tiež platí, že $1 < e < N$. Ak by aj d, aj e bolo viac ako $\sqrt{N}$, tak máme $N = d \cdot e > \sqrt{N} \cdot \sqrt{N} = N$, čo je spor. Preto je jedno z nich nanajvýš rovné $\sqrt{N}$.

Práve sme teda dokázali tvrdenie: Ak celé číslo N nie je prvočíslo, tak má deliteľa d, pre ktorého platí $1 < d \leq \sqrt{N}$.

Budeme teda postupne skúšať všetky hodnoty d z tohto rozsahu. Ak nájdeme nejakú, ktorá delí N, vypíšeme, že nie je prvočíslo. Ak ani jedna z nich N nedelí, je to prvočíslo. Na začiatku ešte samozrejme ošetríme špeciálne prípady $N = 0$ a $N = 1$.

```
get n
put 1 ; get j
jz n bad
jeq n j bad
put 2 ; get d

label cyklus
  put n ; put d ; div ; get e
  jgt d e good
  put n ; put d ; mod ; get e
  jz e bad
  put d ; put 1 ; add ; get d
jump cyklus

label good ; put 1 ; print ; stop
label bad ; put 0 ; print ; stop
```

V cykle vždy najskôr porovnáme hodnoty $\lfloor N/d \rfloor$ a d. Ak už je prvá menšia, znamená to, že určite $d > \sqrt{N}$ a môžeme prestať skúšať. Následne spočítame hodnotu N mod d a ak je 0, tak d delí N, a tiež môžeme prestať, len s opačným výsledkom. V opačnom prípade zvýšime d a ideme na ďalšiu iteráciu cyklu.

Najhorším vstupom pre tento program je samozrejme veľké prvočíslo. Najväčšie prvočíslo, ktoré môžeme na vstupe dostať, je 65 521. Preň tento program spraví niečo vyše 4 000 krokov.

Podúloha b)

Najjednoduchšie riešenie je zmeniť každé číslo v rúre na 1, a následne použiť program z Príkladu 2 v študijnom texte, ktorý tieto jednotky sčíta a vypíše ich súčet – ktorý je zároveň ich počtom.

Prvý krok spravíme tak, že využijeme, že čísla v rúre sú kladné. Vložíme si teda na koniec rúry nulu. Potom v cykle opakujeme: prečítame číslo z rúry. Ak to nie je 0, namiesto neho vložíme do rúry 1. Ak to 0 je, už sme spracovali všetky čísla, v rúre máme namiesto každého z nich jednotku, a môžeme ísť sčítať.

(Drobný detail: aby náš program fungoval aj ak mal na začiatku rúru prázdnu, vložíme teraz do rúry ešte jednu nulu. Tá nám súčet nezmení, a rúra prestane byť prázdna.)

Celý program môže vyzerať napríklad takto:

```
put 0
label prepisuj
  get a ; jz a dalej ; put 1
jump prepisuj
label dalej
put 0
label cyklus
  get a ; jempty koniec ; put a ; add
jump cyklus
label koniec
put a ; print
```

Časová zložitosť je lineárna od počtu čísel, ktoré sú na začiatku v rúre, lebo najskôr raz prejdeme celú rúru, a potom použijeme lineárny program na zistenie súčtu.

Podúloha c)

Aj táto úloha je riešiteľná. Porovnávať hodnoty vieme, stačí ich načítať do rôznych registrov. Problém, ktorý potrebujeme vyriešiť, je, že akonáhle načítame hodnotu do registra, nevieme ju už presunúť do iného. Potrebovali by sme si teda pamätať, kde máme doteraz nájdené maximum. Trik je v tom, že toto si pamätať vieme – pozíciou v programe. Na začiatku máme maximum v registri `a` a nové čísla načítavame do `b`. Ak niekedy do `b` načítame väčšiu hodnotu ako tú, čo máme v `a`, skočíme do inej časti programu, kde sa správame opačne – maximum máme v `b` a nové čísla načítavame do `a`. A ak sa v `a` opäť vyskytne väčšie číslo ako je teraz v `b`, skočíme späť do prvej časti programu. A tak dokola, až kým nedočítame celú postupnosť.

```
get a

label maximum_je_v_a
  jempty koniec_a
  get b ; jgt b a maximum_je_v_b
jump maximum_je_v_a

label maximum_je_v_b
  jempty koniec_b
  get a ; jgt a b maximum_je_v_a
jump maximum_je_v_b

label koniec_a ; put a ; print ; stop
label koniec_b ; put b ; print ; stop
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 400 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX

© Slovenská komisia Olympiády v informatike, 2009