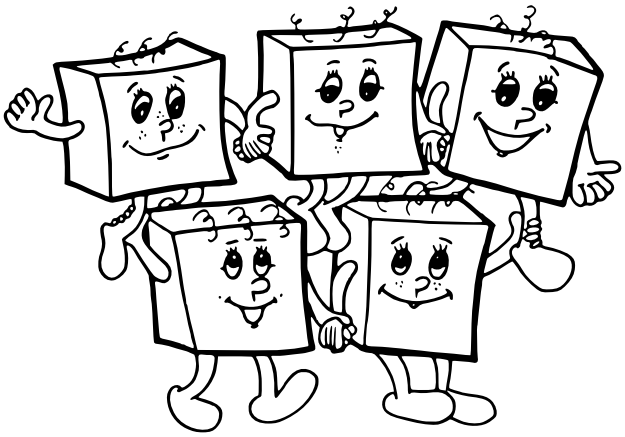
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty piaty ročník

školský rok 2009/10

riešenia krajského kola
kategória A

A-II-1 Tobogany

Keby sme mali celú situáciu riešiť ručne a nie programom, dalo by sa postupovať napríklad nasledovne:

Zoženieme si dostatočne veľa dobrovoľníkov. Na začiatku budú všetci len tak sedieť pri vstupe na tobogany a budú všetci pasívni. Teraz budeme postupne spracúvať nasadnutia na tobogan v poradí, v akom sa naozaj odohrali. Čo sa stane, ak máme niekoho poslať dole toboganom? Ak máme hore nejakého aktívneho dobrovoľníka, jedného z nich (je jedno, ktorého) pošleme dole toboganom. Ak nie, zoberieme pasívneho dobrovoľníka, prehlásime ho za aktívneho a pošleme dole toboganom. Aktívny dobrovoľník má samozrejme za úlohu vybehnúť späť na štart, len čo zo svojho toboganu vypadne. No a tvrdíme, že počet aktívnych dobrovoľníkov na konci dňa je rovný minimálnemu počtu ľudí, ktorí mohli v daný deň tobogany používať.

Sedliackemu rozumu by sa mohlo zdať, že takéto riešenie vyzerá celkom dobre – koniec koncov, nového aktívneho dobrovoľníka pridáme len keď musíme. Toto však samo o sebe nie je dôkazom. Zlé jazyky by sa mohli opýtať: „A nešlo by to predsa len lepšie tak, že by sme niekedy skôr pridali viac aktívnych dobrovoľníkov a tým niekedy neskôr ušetrili?“

Správnosť nášho riešenia preto radšej dokážeme poriadnejšie:

Predpokladajme, že máme ľubovoľné optimálne riešenie O a že máme riešenie N vyrobené naším algoritmom. Ukážeme, že O sa dá konečným počtom krokov prerobiť na N, pričom nezmeníme počet ľudí, ktorých potrebujeme.

Všimnime si prvú udalosť, kedy sa O a N líšia – t. j. prvýkrát pošlú v konkrétnom čase dole konkrétnym toboganom rozličné osoby. Ako sa to mohlo stať? A čo s tým spraviť? Rozoberieme niekoľko prípadov:

- V riešení N sme „vyrobili“ nového aktívneho dobrovoľníka len vtedy, ak práve nebol žiaden aktívny dobrovoľník k dispozícii. Keďže riešenie O sa doteraz s N zhodovalo, v takomto prípade tiež nemáme nikoho aktívneho k dispozícii, a teda nutne spravíme to isté ako v N. Tento prípad teda nenastal.

- Vieme teda, že v riešení N sme dole toboganom poslali niektorého aktívneho dobrovoľníka x, ktorý práve čakal na štarte. Ak sme v O poslali iného aktívneho dobrovoľníka y, upravíme O na O' tak, že od tohto okamihu všetky jazdy x robí y a naopak.
- Zostáva teda posledný prípad: V riešení N sme dole toboganom poslali niektorého aktívneho dobrovoľníka x, zatiaľ čo v našom konkrétnom optimálnom riešení O sme vyrobili nového aktívneho dobrovoľníka z a poslali toho. (Inými slovami, v O prišiel človek, ktorý sa v ten deň ešte nespustil.)

  V tomto prípade opäť môžeme upraviť O na O' jednoducho tak, že vymeníme z a x.

Ukázali sme, že ak sa O a N líšia, tak bez ohľadu na to, ktorý prípad nastal, vždy vieme O upraviť tak, aby sme dostali rovnako dobré riešenie, ktoré sa s N líši až v neskoršom kroku. Po konečne veľa opakovaniach vyššie uvedeného postupu teda z riešenia O nutne vyrobíme naše riešenie N, čím sme dokázali, že aj naše riešenie N je nutne optimálne.

Ako to implementovať? Program sme písali tak, aby sa navyše podľa neho dal priamo aj jeden rozvrh zostrojiť. Vstup reprezentujeme ako 3 fronty (pre každý tobogan máme jednu). Taktiež výstupy z toboganov (t. j. časy, keď z neho vystúpia dobrovoľníci, ktorí sa ním práve vezú) si udržujeme v ďalších 3 frontách. Nasledujúcu udalosť vždy zistíme ako minimum z hodnôt vo vstupných frontách. Pre každú udalosť sa pozrieme na situáciu na výstupe. Nájdeme tam najskoršie vystupujúceho dobrovoľníka. Ak stíha túto jazdu, vyberieme ho z patričnej fronty. Ak nie, musíme pridať nového človeka.

Časová aj pamäťová zložitosť tohto riešenia je $O(N)$, kde N je celkový počet jázd.

Pomalšie riešenia

Vyššie uvedené riešenie sa dá zostrojiť aj ináč – tak, že budeme uvažovať sekvenčne. Zoberieme prvého človeka, pustíme ho na úplne prvú jazdu, potom na prvú ďalšiu čo stíha, atď. Ak nám ešte zostali nespravené jazdy, tak pridáme druhého, tretieho, atď., kým nepokryjeme všetky jazdy.

Dôkaz správnosti tohto riešenia vyzerá podobne ako pri našom vzorovom riešení. (Presnejšie, ak by sme v našom vzorovom riešení pridali podmienku „ak máš k dispozícii viacero aktívnych dobrovoľníkov, pošli toho s najmenším poradovým číslom“, zostrojili by sme úplne to isté riešenie ako týmto postupom.)

Priamočiara implementácia má v najhoršom možnom prípade časovú zložitosť $O(N^2)$. Existuje ale aj implementácia tohto riešenia s časovou zložitosťou $O(N \log N)$.

Za zmienku taktiež stojí pomalšie riešenie, ktoré je ale univerzálnejšie a dalo by sa napríklad použiť aj v situácii, kedy rôzne tobogany začínajú a končia na rôznych miestach, a teda pešie presuny medzi ich koncami a začiatkami trvajú rôzne dlho.

Prevedieme úlohu na hľadanie maximálneho párovania v bipartitnom grafe. Jednu partíciu budú tvoriť časy konca jázd, druhú časy začiatku, a hrana znamená, že po danom konci sa stíha daný začiatok. Každému platnému rozvrhu zodpovedá nejaké párovanie v tomto grafe a naopak. Totiž každá hrana, ktorú vyberieme do párovania, vlastne hovorí, že človek, ktorý práve dokončil jednu jazdu, má ako nasledujúcu spraviť príslušnú druhú jazdu. Zjavne každou vybranou hranou ušetríme jedného človeka, preto hľadaný minimálny počet ľudí vieme vypočítať ako N mínus veľkosť maximálneho párovania v našom grafe.

Listing programu:

```
#include <cstdio>
#include <queue>

using namespace std;

#define MAX_TIME 2000000100

int main()
{
        int t[3], d, n[3], a;
        queue<int> nastup[3];
        queue<int> vystup[3];
        scanf("%d %d %d %d", &t[0], &t[1], &t[2], &d);
```

```
        for(int i = 0; i < 3; i++)
        {
                scanf("%d", &n[i]);
                for(int j = 0; j < n[i]; j++)
                {
                        scanf("%d", &a);
                        nastup[i].push(a);
                }
                nastup[i].push(MAX_TIME); //vlozime fiktivny posledny nastup
        }
        int prvyNastupPos, prvyNastupCas;
        int prvyVystupCas, prvyVystupPos;
        int pocetDeti = 0;
        while(1)
        {
                prvyNastupCas = MAX_TIME;
                prvyNastupPos = -1;
                for(int i = 0; i < 3; i++) //najdeme prvy nespracovany nastup
                {
                        if(nastup[i].front() < prvyNastupCas)
                        {
                                prvyNastupCas = nastup[i].front();
                                prvyNastupPos = i;
                        }
                }
                if(prvyNastupCas == MAX_TIME) break; //vycerpali sme vsetko, koncime
                prvyVystupCas = MAX_TIME;
                for(int i = 0; i < 3; i++) //najdeme prvy nepouzity vystup
                {
                        if(vystup[i].size() == 0) continue;
                        if(vystup[i].front() < prvyVystupCas)
                        {
                                prvyVystupCas = vystup[i].front();
                                prvyVystupPos = i;
                        }
                }
                if(prvyVystupCas <= prvyNastupCas) //mozeme tento vystup pouzit
                        vystup[prvyVystupPos].pop();
                else
                        pocetDeti++;
                nastup[prvyNastupPos].pop();
                vystup[prvyNastupPos].push(prvyNastupCas+t[prvyNastupPos]+d);
        }
        printf("%d\n", pocetDeti);
}
```

A-II-2 Oplotenie farmy

Vzorové riešenie tohto príkladu bude využívať postup, ktorý sme použili v úlohe o čokoláde z domáceho kola. Každý štvorcový úsek farmy môžeme jednoznačne identifikovať bodom, v ktorom sa nachádza jeho pravý dolný roh a dĺžkou jeho strany. Základnou myšlienkou riešenia je, že namiesto počítania pestrých štvorcov môžeme spočítať štvorce všetky a od nich odčítať počet jednofarebných.

Keby sme pre každý bod (r, s) vedeli počet jednofarebných štvorcov, ktorých pravý dolný roh je (r, s), potom by sme už vedeli spočítať hľadaný výsledok. Totiž počet všetkých štvorcov (jednofarebných a pestrých dokopy), ktoré majú pravý dolný roh na (r, s) je $\min(r, s)$. A teda vieme počet pestrých štvorcov pre (r, s) vypočítať ako $\min(r, s)$ mínus počet tých, ktoré sú jednofarebné.

Pozrime sa bližšie na štvorce rôznych veľkostí ktoré majú pravý dolný roh na (r, s). Zjavne platí, že akonáhle je niektorý z nich pestrý, sú pestré aj všetky väčšie od neho – lebo ho celý obsahujú.

Označme $K(r, s)$ dĺžku strany najväčšieho štvorca, ktorý má pravý dolný roh na (r, s) a v ktorom sú všetky plodiny rovnaké. Keď sa teraz pozrieme na štvorce, ktoré majú pravý dolný roh na (r, s), tak zjavne tie, ktoré majú dĺžku strany od 1 po $K(r, s)$ sú jednofarebné a všetky väčšie sú pestré. Preto $K(r, s)$ je zároveň rovné počtu jednofarebných štvorcov, ktorých pravý dolný roh je (r, s).

A ako vypočítať hodnotu $K(r, s)$? Uvažujme políčka $(r, s-1)$, $(r-1, s)$ a $(r-1, s-1)$. Ak je v aspoň jednom z týchto políčok iná plodina ako na políčku (r, s), potom je $K(r, s)$ rovné jednej, pretože štvorec s dĺžkou strany

dva už je pestrý. V prípade, že sa plodiny na týchto políčkach zhodujú s plodinou na políčku (r, s), potom platí:

$$K(r,s) = 1 + \min(\, K(r-1,s),\, K(r,s-1),\;\; K(r-1,s-1)\,)$$

Dôkaz tejto rovnosti sa zhoduje s dôkazom uvedeným v riešeniach domáceho kola.

Keďže na výpočet $K(r,s)$ stačí poznať hodnoty $K(r-1,s), K(r,s-1), K(r-1,s-1)$, môžeme postupovať po riadkoch od hora dole a v rámci riadku zľava doprava a každú hodnotu získať v konštantnom čase s využitím hodnôt, ktoré už poznáme. Preto má toto riešenie časovú zložitosť $O(RS)$. Ak by sme si načítali celý vstup, mali by sme aj pamäťovú zložitosť $O(RS)$. Avšak, podobne ako v domácom kole, môžeme túto zložitosť zlepšiť na $O(R)$, keď si uvedomíme, že stačí v pamäti držať len dva riadky vstupu a tiež dva riadky hodnôt $K(r,s)$, pretože predchádzajúce riadky sú už pre nás zbytočné.

Listing programu:

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

int R,S,K;
int M[2][2500],D[2][2550];
long long res;

int min(int a, int b){return a < b ? a : b;}

int main(){
  scanf("%d %d %d ",&R,&S,&K);
  //uplne hore a uplne vlavo si domyslime imaginarne policka
  //s plodinou cislo K, aby sa nam lahsie pocitalo
  for(int i=0;i<=S;i++) D[0][i] = K;
  for(int i=1;i<=R;i++){
    int novy = i%2;
    int stary = (novy+1)%2;
    D[novy][0] = K;
    for(int j=1;j<=S;j++) scanf("%d ",&D[novy][j]);
    for(int j=1;j<=S;j++){
      if (D[novy][j-1] != D[novy][j] || D[stary][j] != D[novy][j] || D[stary][j-1] != D[novy][j])
        M[novy][j] = 1;
      else
        M[novy][j] = 1 + min( min( M[novy][j-1], M[stary][j] ), M[stary][j-1] );
      res += min(i,j) - M[novy][j];
    }
  }
  printf("%Ld\n",res);
  return 0;
}
```

A-II-3 Obmedzovač rýchlosti

Prvé pozorovanie tejto úlohy je, že nech si vyberieme akúkoľvek trasu, optimálne nastavenie obmedzovača rýchlosti je rovné minimu z obmedzení rýchlosti na našej trase – menej sa neoplatí, viac nesmieme.

Dôsledkom je, že pre dosiahnutie najnižšieho času cesty medzi dvoma mestami má zmysel nastavovať obmedzovač rýchlosti len na jednu z rýchlostí, ktoré sú na vstupe.

Pomalšie riešenie

Uvažujme problém, kedy už máme nastavené obmedzenie na nejakú rýchlosť v. Za aký čas sa teraz vieme dostať z mesta x do mesta y?

Nech G_v je podgraf pôvodného grafu, ktorý obsahuje len hrany, kde je rýchlosť v ešte povolená. T. j. ak máme nastavené obmedzenie na rýchlosť v, potom môžeme cestovať len po hranách grafu G_v. Keďže rýchlosť jazdy už máme určenú, zrejme najlepší čas jazdy dosiahneme vtedy, ak si zvolíme najkratšiu možnú trasu (v kilometroch).

Tým sa nám ponúka nasledujúce riešenie:

```
pre každú rýchlosť v zo vstupu
    vytvoríme graf G_v;
```

`riešime problém (dĺžkovo) najkratšej cesty v grafe` G_v`;`
`pre každú dvojicu miest` x `a` y `poznačíme riešenie s časom` $dist_v(x,y)/v$`;`

kde $dist_v(x,y)$ určuje dĺžku (v kilometroch) najkratšej cesty medzi mestami x, y v grafe G_v.

Ako je to s časovou zložitosťou takéhoto riešenia? Podľa zadania a nášho úvodného pozorovania stačí pre v uvažovať všetky možné rýchlosti na vstupe, ktorých je dohromady M. Graf G_v vieme zakaždým ľahko zostrojiť v čase $O(M)$ tak, že prejdeme cez všetky hrany pôvodného grafu. Otázkou už iba ostáva, ako vyriešiť „klasický“ problém najkratšej cesty v grafe G_v medzi každou dvojicou vrcholov. Dva možné prístupy:

- Algoritmus Floyd-Warshall, ktorý tento problém rieši v čase $O(N^3)$.
- N-krát spustený Dijkstrov algoritmus (raz z každého vrcholu). Tým vieme získať opäť časovú zložitosť $O(N^3)$, alebo v prípade využitia haldy získame riešenie v čase $O(NM \log N)$.

Nakoľko celý cyklus sa spúšťa pre každú rýchlosť na vstupe (t.j. najviac M-krát), celková časová zložitosť takéhoto riešenia je $O(MN^3)$, resp. $O(M^2N \log N)$ a pamäťová zložitosť $O(N^2)$.

Vzorové riešenie

Naše vzorové riešenie využíva podobnú myšlienku ako algoritmus Floyd-Warshall.

Začneme tým, že zoradíme všetky hrany zo vstupu do postupnosti $e_1, e_2, \ldots, e_m$ tak, aby pre ich maximálne povolené rýchlosti platilo $v_1 \geq v_2 \geq v_3 \geq \ldots \geq v_m$.

Úlohu teraz vyriešime metódou dynamického programovania. Postupne pre každé k spočítame všetky hodnoty $d_k(x,y)$ – dĺžku najkratšej cesty (v kilometroch) medzi mestami x a y, ak sa smieme pohybovať len po hranách $e_1, e_2, \ldots, e_k$. (Resp. $d_k(x,y) = \infty$, ak taká cesta neexistuje.)

Ekvivalentne, podľa definície z predchádzajúceho riešenia môžeme povedať, že $d_k(x,y)$ je dĺžka najkratšej cesty medzi mestami x a y v grafe G_{v_k}. Ako sme konštatovali skôr, všetko, čo by sme potrebovali, je vyriešiť tento problém pre každú z hodnôt $k = 1, 2, \ldots, M$.

Našou ideou teraz bude v k-tom kroku „sprístupniť“ hranu e_k a prerátať hodnoty $d_k(x,y)$ pre všetky dvojice miest x, y. Pri určení hodnoty $d_k(x,y)$ uvažujeme dve možnosti.

- V prípade, že najkratšia cesta hranu e_k nepoužíva, je $d_k(x,y) = d_{k-1}(x,y)$.
- V opačnom prípade sa dá najkratšia cesta z x do y rozdeliť na 3 úseky: prídeme z x do niektorého vrcholu hrany e_k, prejdeme ňou a z jej druhého konca prídeme do y. Aj v prvom, aj v treťom úseku sa už vyskytujú len hrany s číslom menším ako k, a teda už vieme ich optimálnu dĺžku.

  Formálne, označme u_{k1} a u_{k2} vrcholy spojené hranou e_k. Potom platí buď $d_k(x,y) = d_{k-1}(x,u_{k,1}) + |e_k| + d_{k-1}(u_{k,2},y)$ alebo $d_k(x,y) = d_{k-1}(x,u_{k,2}) + |e_k| + d_{k-1}(u_{k,1},y)$ – podľa toho, ktorým smerom naša trasa prechádza hranou e_k.

My vždy máme na výber, či hranu e_k použiť alebo nie, a ak áno, tak v ktorom smere. Vyberieme si samozrejme najlepšiu z uvedených troch možností. Preto $d_k(x,y)$ je vždy rovné minimu z uvedených troch možností.

Tým sme získali rekurzívny vzťah pre hodnoty $d_k(x,y)$. Každú z týchto MN^2 hodnôt pomocou neho spočítame v konštantnom čase, takéto riešenie má teda časovú zložitosť $O(MN^2)$.

Pre zníženie pamäťovej zložitosti si stačí všimnúť, že hodnoty d_k závisia len od hodnôt d_{k-1}, a teda si vystačíme s pamäťou veľkosti $O(N^2)$.

Listing programu:

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

struct hrana{
   double m,d;
   int x,y;
   hrana(){}
   hrana(int x, int y, double m, double d): x(x), y(y), m(m), d(d) {}
};
```

```
int operator < (const hrana &h1, const hrana &h2){
   return h1.m > h2.m;
}

const double INF=1e50;
const int maxM=5000, maxN=100;
int n,m;
hrana h[maxM];
double vysl[maxN][maxN];      //tu si pamatame celkovy vysledok, tj. casy medzi mestami
double d[maxN][maxN];         //v d si pamatame vzdialenost pouzitim prvych k-1 hran

int main(){
   scanf("%d %d",&n,&m);
   for(int k=0;k<m;k++){
      scanf("%d %d %lf %lf",&h[k].x,&h[k].y,&h[k].d,&h[k].m);
      h[k].x--; h[k].y--;
   }
   sort(h,h+m*sizeof(hrana));
   for(int i=0;i<n;i++)
      for(int j=0;j<n;j++) d[i][j]=vysl[i][j]=(i==j)?0.0:INF;

   for(int k=0;k<m;k++){
      double v=h[k].m;
      for(int i=0;i<n;i++)
         for(int j=0;j<n;j++){
            d[i][j]=min( d[i][j] , d[i][h[k].x] + h[k].d + d[h[k].y][j]);
            d[i][j]=min( d[i][j] , d[i][h[k].y] + h[k].d + d[h[k].x][j]);
            vysl[i][j]=min(vysl[i][j], d[i][j]/v);
         }
   }
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++) printf("%.3lf ",vysl[i][j]);
      printf("\n");
   }
}
```

A-II-4 Počítač s gumenou rúrou

Podúloha a: Lucasove čísla

(V celom riešení tejto podúlohy ticho predpokladáme, že všetky sčítania sú sčítaniami modulo 65 536.)

Použijeme dva registre a a b. Na začiatku do nich uložíme hodnoty $L_0 = 2$ a $L_1 = 1$.

Predstavme si teraz, že v a máme hodnotu L_{x-2}, v b hodnotu L_{x-1} a rúra je prázdna. Teraz vieme spočítať hodnotu L_x: vložíme do rúry hodnotu z a, vložíme tam aj hodnotu z b a vykonáme príkaz `add`.

Následne do rúry vložíme ešte raz hodnotu z registra b. V rúre teda teraz máme za sebou najskôr hodnotu $L_{x-2} + L_{x-1} = L_x$ a potom hodnotu L_{x-1}. Prvú z nich načítame do registra b a druhú do registra a.

Tým sme sa dostali do „o jedno posunutej“ situácie: začínali sme s hodnotami L_{x-2} a L_{x-1}, skončili sme s hodnotami L_{x-1} a L_x.

A teraz už môžeme ľahko napísať celý program riešiaci súťažnú úlohu: Číslo z rúry uložíme do registra n. Inicializujeme registre a a b na L_0 a L_1. Vyššie uvedený postup n-krát zopakujeme, čím dostaneme v registroch a a b hodnoty L_n a L_{n+1}. A na záver hodnotu z registra a vypíšeme na výstup.

```
get n
put 2 ; get a
put 1 ; get b
label loop
jz n koniec
put n ; put 1 ; sub ; get n
put a ; put b ; add
put b
get b
get a
jump loop
```

```
label koniec
put a
print
```

Podúloha b: hľadanie výskytu čísla 47

Testovať, či je číslo rovné 47, vieme napríklad pomocou inštrukcie `jeq` – skok v prípade rovnosti hodnôt v dvoch registroch. Na to, aby sme ju mohli používať, však musíme dostať do nejakého registra hodnotu 47. A to nevieme spraviť len tak, že ju vložíme do rúry a hneď načítame do registra – lebo rúru už máme plnú vstupu.

Tento problém vyriešime tak, že obsah rúry „pretočíme“. Začneme tým, že si do rúry vložíme hodnotu 0. Táto bude slúžiť ako „zarážka“ za vstupnou postupnosťou kladných čísel. Za túto 0 vložíme hodnotu 47.

Teraz budeme dokola opakovať: načítame hodnotu z rúry do registra, a ak to nie je nula, vložíme ju naspäť do rúry. Takto sa bude obsah rúry postupne otáčať, až kým niekedy do registra nenačítame našu nulu. V tomto okamihu vieme, že prvé číslo v rúre je naša 47 a za ňou nasleduje celá vstupná postupnosť. Hodnotu 47 si teda načítame do registra z.

Teraz sme presne v rovnakej situácii ako na začiatku, s jediným rozdielom – v registri z máme číslo 47. S tým teraz každú hodnotu v zadanej postupnosti porovnáme. Ak sa nám rúra vyprázdni, vložíme do nej hodnotu 0, vypíšeme ju na výstup a skončíme. Ak niekedy narazíme na hodnotu 47, najskôr v cykle vyprázdnime zvyšok rúry, potom do nej vložíme hodnotu 1, vypíšeme ju na výstup a skončíme.

```
put 0
put 47
label pretoc
get a ; jz a dalej ; put a
jump pretoc

label dalej
get z
label testuj
jempty nenasiel
get a
jeq a z nasiel
jump testuj

label nasiel
jempty pis1
get x
jump nasiel

label pis1
put 1 ; print
jump koniec

label nenasiel
put 0 ; print
jump koniec

label koniec
```

Pre zaujímavosť uvedieme ešte jednu možnosť, ako riešiť druhú polovicu úlohy. Za postupnosť si opäť vložíme nulu ako zarážku. Teraz postupne spracúvame zadanú postupnosť a vždy, keď nájdeme hodnotu 47, do rúry vložíme číslo 1. Po tom, ako z rúry načítame nulu-zarážku, do rúry vložíme ešte jednu 0.

V tomto okamihu teda máme v rúre najskôr niekoľko jednotiek (a to presne toľko, koľkokrát bola v zadanej postupnosti hodnota 47) a následne jednu nulu. Prvé číslo z rúry vypíšeme na výstup a skončíme.

Podúloha c: výpis párnych čísel

Keď chceme zistiť, či je nejaké číslo párne, môžeme spočítať zvyšok, aký dáva po delení 2, a overiť, že je 0. Zvyšok vieme spočítať inštrukciou `mod`. Zároveň si ale musíme niekde „odložiť“ aj pôvodnú hodnotu čísla, lebo `mod` svoje vstupy z rúry odstráni.

V prvej fáze nášho riešenia si pripravíme obsah rúry tak, aby sme mohli použiť inštrukciu `mod` na vypočítanie zvyškov po delení dvoma. Použijeme rovnaký trik ako v predchádzajúcej podúlohe – začneme tým, že za vstupnú postupnosť vložíme 0 ako zarážku. Teraz budeme po jednom čítať vstupnú postupnosť a za každé prečítané číslo a do rúry postupne vložíme hodnoty 1, a, 2, a. Hodnota 1 nám bude hovoriť „ešte nasleduje ďalší prvok“, nasledujúce dve hodnoty a a 2 použijeme ako vstupy pre `mod`, a posledná hodnota a zostane zachovaná.

Druhú fázu nášho riešenia opäť začneme tým, že na koniec postupnosti pridáme 0 ako zarážku. Teraz dokola opakujeme: Načítame z rúry hodnotu do registra. Ak je to 0, už sme spracovali celú postupnosť a prejdeme na tretiu fázu. Ak nie, vykonáme inštrukciu `mod`. Tá zoberie zo začiatku rúry hodnoty a a 2 a namiesto nich vloží do rúry a mod 2. A následne načítame z rúry hodnotu do registra a vložíme ju späť.

Ak sme na začiatku mali v rúre postupnosť $(s_1, \ldots, s_k)$, tak po skončení druhej fázy nášho riešenia tam máme postupnosť $(s_1 \bmod 2, s_1, s_2 \bmod 2, s_2, \ldots, s_k \bmod 2, s_k)$.

No a v tretej, poslednej fáze použijeme spočítané zvyšky na to, aby sme určili, ktoré čísla vypísať a ktoré nie. Prečítame zvyšok, ak je 1, nasledujúce číslo zahodíme, ak je 0, nasledujúce číslo vypíšeme na výstup. Keď sa rúra vyprázdni, skončíme.

```
put 0
label mark
get a ; jz a druha_faza
put 1 ; put a ; put 2 ; put a
jump mark

label druha_faza
put 0
label dalej
get a ; jz a vypis
mod
get a ; put a
jump dalej

label vypis
jempty koniec
get b
jz b chceme
get a
jump vypis

label chceme
print
jump vypis

label koniec
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY PIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX