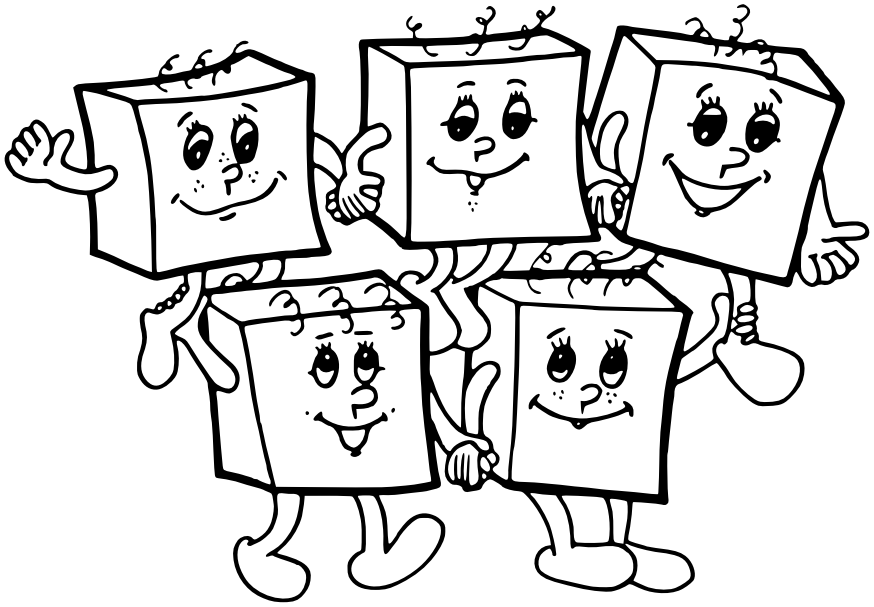
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH



dvadsiaty štvrtý ročník
školský rok 2008/09

riešenia krajského 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-II-1 Horár Jedlička

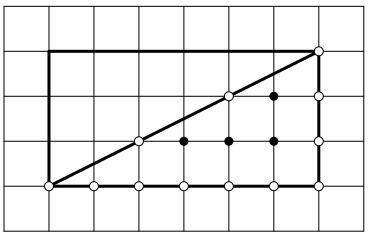
Intuitívne sa zdá, že počet mrežových bodov vo vnútri mnohouholníka bude približne úmerný jeho obsahu. A naozaj, pre všetky mnohouholníky s vrcholmi v mrežových bodoch platí tzv. *Pickova veta*: $S = V + H/2 - 1$, kde S je obsah mnohouholníka, V je počet mrežových bodov vo vnútri mnohouholníka a H je počet mrežových bodov na hranici (vrátane vrcholov). Ukážeme, ako sa dá toto tvrdenie objaviť, a náš postup bude zároveň dôkazom.

Základné pozorovanie, ktoré použijeme, je nasledovné: Predstavte si, že zoberieme mnohouholník a rozstrihneme ho po úsečke, ktorá spája dva mrežové body. Dostaneme takto dva menšie mnohouholníky. Celková plocha zjavne zostala rovnaká. Takisto zostal rovnaký počet mrežových bodov, ktoré oba dokopy obsahujú. Jediné, čo sa zmenilo, je, že keď sme strihali, tak sme možno prestrihli niekoľko mrežových bodov. Tieto doteraz boli vo vnútri jedného mnohouholníka, odteraz sú na obvode dvoch. (To intuitívne zdôvodňuje, prečo sa v Pickovej vete mrežové body na obvode rátajú len za polovicu.)

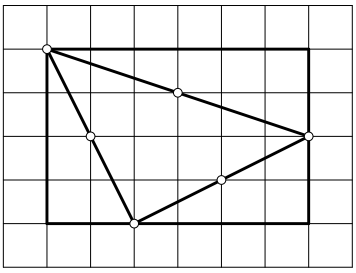
Teraz postupne dokážeme, že Pickova veta platí pre pekné obdĺžniky, pre pravouhlé trojuholníky, pre ľubovoľné trojuholníky a nakoniec pre všeobecné mnohouholníky.

Začneme najjednoduchším špeciálnym prípadom – obdĺžnik, ktorého strany sú rovnobežné so súradnicovými osami. Bez ujmy na všeobecnosti nech sú jeho protiľahlé rohy v bodoch $[0, 0]$ a $[x, y]$. Potom jeho plocha je $S = xy$, mrežových bodov vnútri je $V = (x - 1)(y - 1)$ a na obvode ich je $H = 2x + 2y$. A teda naozaj platí $S = V + H/2 - 1$.

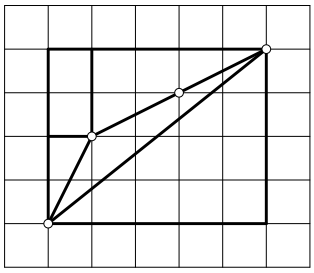
Teraz si predstavme, že takýto obdĺžnik rozstrihneme po uhlopriečke. Dostaneme dva rovnaké pravouhlé trojuholníky:



Obr. 1: Obdĺžnik a dva pravouhlé trojuholníky.



Obr. 2: Vystrihávame všeobecný trojuholník, príklad 1.



Obr. 2: Vystrihávame všeobecný trojuholník, príklad 2.

Označme si x počet mrežových bodov (iných ako rohy obdĺžnika), ktoré ležia na práve prestrihnutej uhlopriečke. Potom každý z trojuholníkov má: plochu $S' = S/2$, mrežových bodov vnútri $V' = (V - x)/2$ a mrežových bodov na obvode $H' = H/2 + x + 1$. A ľahko overíme, že platí $S' = V' + H'/2 - 1$.

Teraz už teda vieme, že Pickova veta platí pre pravouhlé trojuholníky. Čo so všeobecným trojuholníkom? Všeobecnému trojuholníku môžeme opísať obdĺžnik. Dva základné prípady, ktoré môžu nastať, sú znázornené na obrázkoch 2 a 3. Vždy dostaneme približne to isté: Obdĺžnik rozdelený na niekoľko častí. Jedna z nich je trojuholník, ktorý nás zaujíma, a ostatné sú útvary, o ktorých už vieme, že pre ne Pickova veta platí. Rovnakým postupom ako v predchádzajúcom odseku dostaneme, že potom nutne musí platiť aj pre náš trojuholník.

Ostáva nám len posledný krok – všeobecné mnohouholníky. A tento krok je vlastne opäť len zopakovanie tej istej úvahy, len tentokrát naopak. Každý mnohouholník má aspoň jednu *trianguláciu*, t. j. dá sa pozdĺž $N - 3$ vhodných uhlopriečok rozstrihnúť na $N - 2$ neprekrývajúcich sa trojuholníkov.[1] Pre každý z nich platí Pickova veta. A keď zoberieme dva mnohouholníky, o ktorých vieme, že pre ne platí, a zlepíme ich dokopy, bude Pickova

[1] Jeden možný dôkaz vyzerá nasledovne: Pre konvexné mnohouholníky triangluáciu nájdeme ľahko. Ak máme nekonvexný mnohouholník, zoberme jeho jeden nekonvexný vrchol, otočme ho tak, aby obe strany z neho išli „niekam dohora“, zoberme polpriamku z tohto vrcholu „rovno dodola“ a otáčajme ju, kým nenarazí na nejaký vrchol. V tomto okamihu sme našli uhlopriečku, ktorá leží celá vovnútri a môžeme strihať. Indukciou ďalej.

veta platiť aj pre nový mnohouholník – na jednej strane rovnice dostaneme súčet plôch, na druhej sa sčítajú mrežové body vnútri a na obvode.

To isté ešte raz a poriadnejšie: Nech mnohouholník M vznikol zlepením mnohouholníkov A a B, so spoločnou stranou na ktorej vnútrajšku leží x mrežových bodov. Označme počet mrežových bodov vnútri A ako V_A a počet mrežových bodov na hranici A ako H_A. Podobne V_B a H_B je počet mrežových bodov vo vnútri a na hranici B. Potom $V = V_A + V_B + x$ a $H = H_A + H_B - 2x - 2$. Z toho vyplýva, že $(V_A + H_A/2 - 1) + (V_B + H_B/2 - 1) = (V_A + V_B + x) + (H_A + H_B - 2x - 2)/2 - 1 = V + H/2 - 1$. Ak teda obsah A je rovný $V_A + H_A/2 - 1$ a obsah B sa rovná $V_A + H_B/2 - 1$, potom obsah M sa rovná $V + H/2 - 1$.

Trochu iný dôkaz

Pre zaujímavosť uvedieme myšlienku ešte jedného dôkazu Pickovej vety. Pre každý mrežový bod $[x, y]$ vo vnútri alebo na hranici daného mnohouholníka P označme $\varphi_{P,x,y}$ veľkosť uhla, pod ktorým z neho vidíme vnútro mnohouholníka. Teda ak $[x, y]$ je vnútri, tak $\varphi_{P,x,y} = 2\pi$, ak je na hrane, tak $\varphi_{P,x,y} = \pi$, a pre vrcholy môže nadobúdať rôzne hodnoty.

Sčítajme teraz tieto uhly pre všetky mrežové body v rovine. Presnejšie, definujme $\Psi(P) = \frac{1}{2\pi} \sum \varphi_{P,x,y}$.

Označme teraz, rovnako ako v predchádzajúcom riešení, obsah nášho N-uholníka S_P, počet mrežových bodov v jeho vnútri V_P a počet mrežových bodov na jeho obvode H_P. Zjavne každý z V_P mrežových bodov vnútri prispeje do $\Psi(P)$ jednotkou, každý z $H_P - N$ bodov na hranách jednou polovicou, a všetkých N vrcholov dokopy prispeje $(N - 2)/2$ – to preto, že súčet vnútorných uhlov N-uholníka je $2\pi(N - 2)$. Dokopy teda $\Psi(P) = V_P + (H_P - N)/2 + (N - 2)/2 = V_P + H_P/2 - 1$.

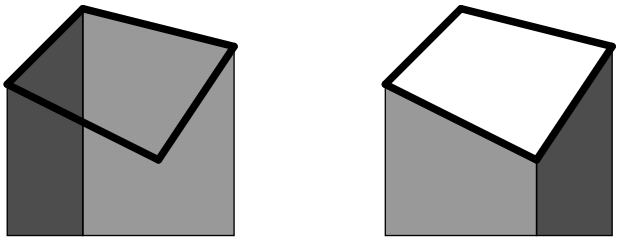
A už zostáva len dokázať, že súčasne platí, že $\Psi(P)$ je zároveň rovné ploche P. To platí napríklad z podobných dôvodov ako v predchádzajúcom riešení – platí to pre trojuholníky, a pre neprekrývajúce sa P, Q zjavne platí $\Psi(P \cup Q) = \Psi(P) + \Psi(Q)$.

Riešenie úlohy

Pomocou práve dokázanej vety vieme našu úlohu previesť na dve jednoduchšie: Stačí nám zistiť obsah zadaného mnohouholníka a počet mrežových bodov na jeho hranici.

Počet bodov na hranici vypočítame tak, že sčítame počet vrcholov a pre každú stranu počet mrežových bodov v jej vnútri. Počet mrežových bodov vo vnútri strany s koncami $[x_1, y_1]$ a $[x_2, y_2]$ vypočítame ako „najväčší spoločný deliteľ čísel $|x_2 - x_1|$ a $|y_2 - y_1|$, mínus jedna“.[2] Najväčšieho spoločného deliteľa ľahko spočítame pomocou Euklidovho algoritmu.

Obsah mnohouholníka ľahko určíme napríklad tak, že pre každú jeho stranu zostrojíme lichobežnik, ktorého jedna strana je strana mnohouholníka, dve strany sú zvislé a štvrtá leží na osi x. Ak máme mnohouholník popísaný proti smeru hodinových ručičiek, dostaneme jeho plochu tak, že sčítame plochy pre lichobežniky, pre ktoré strana mnohouholníka „ide doľava“ a odčítame plochy pre lichobežniky, ktorých strana mnohouholníka „ide doprava“. Napríklad na nasledujúcom obrázku lichobežniky znázornené vľavo pripočítame k ploche a tie znázornené vpravo odpočítame.



[2] Dôkaz: Body na uvedenej úsečke vieme parametricky vyjadriť ako $[x_1 + t(x_2 - x_1), y_1 + t(y_2 - y_1)]$ pre $t \in (0, 1)$. Mrežových bodov je teda toľko ako rôznych hodnôt t, pre ktoré sú $t(x_2 - x_1)$ aj $t(y_2 - y_1)$ celé. Nech d je najväčší spoločný deliteľ $x_2 - x_1$ a $y_2 - y_1$. Potom existujú celé a, b také, že $d = a(x_2 - x_1) + b(y_2 - y_1)$. Ak majú $t(x_2 - x_1)$ aj $t(y_2 - y_1)$ byť celé, musí byť celé číslo aj $at(x_2 - x_1) + bt(y_2 - y_1) = td$. A toto spĺňajú len $t \in \{1/d, 2/d, \ldots, (d - 1)/d\}$.

Na určenie obsahu lichobežníka použijeme známy vzorec – priemer dĺžok základní × výška. Vzhľadom k tomu, že tak ako aj v tomto vzorci ako aj vo vzorci na určenie počtu mrežových bodov sa vyskytuje delenie dvoma, budeme radšej počítať dvojnásobok obsahu, aby sme mohli používať celé čísla.

(Alternatívny spôsob počítania obsahu je pomocou vektorového súčinu. Namiesto lichobežníkov rozdelíme mnohouholník na N trojuholníkov, pričom vrcholy i-teho trojuholníka sú $[x_i, y_i]$, $[x_{i+1}, y_{i+1}]$ a $[0, 0]$. Plochu mnohouholníka dostaneme ako súčet orientovaných plôch týchto trojuholníkov, pričom orientovaná plocha takéhoto trojuholníka je polovica z-ovej súradnice vektorového súčinu vektorov $[x_i, y_i, 0]$ a $[x_{i+1}, y_{i+1}, 0]$. Na konci riešenia tejto úlohy uvádzame listing veľmi stručného riešenia v C++ založeného na tejto myšlienke.)

Časová a pamäťová zložitosť

Časová zložitosť určenia obsahu je lineárna od počtu vrcholov. Časová zložitosť určenia počtu bodov na hranici závisí na zložitosti určenia najväčšieho spoločného deliteľa. Ak použijeme Euklidov algoritmus, tak je táto zložitosť $O(\log \min(X, Y))$, kde X a Y sú obmedzenia na velkosť súradníc vrcholov mnohouholníka. Výsledná časová zložitosť je teda $O(N \log \min(X, Y))$. Okrem súradníc mnohouholníka si potrebujeme pamätať iba konštantné množstvo medzivýsledkov, pamäťová zložitosť je teda $O(N)$.

Listing programu:

```
program les;

const MAXN = 100000;

type bod = record x, y : longint; end;

var n : longint;
    body : array [1 .. MAXN] of bod;
    hranice, dobsah : longint;

procedure nacti_vstup;
var i : longint;
begin
  readln (n);
  for i := 1 to n do readln (body[i].x, body[i].y);
end;

function nsd (n1, n2 : longint) : longint;
var t : longint;
begin
  if n1 > n2 then begin  t := n1;  n1 := n2;  n2 := t;  end;
  while n1 > 0 do begin  t := n1;  n1 := n2 mod n1;  n2 := t;  end;
  nsd := n2;
end;

function pocet_vnitrnich_bodu_usecky (a, b : bod) : longint;
var dx, dy : longint;
begin
  dx := abs (a.x - b.x);
  dy := abs (a.y - b.y);
  if (dx = 0) and (dy = 0) then pocet_vnitrnich_bodu_usecky := 0
    else pocet_vnitrnich_bodu_usecky := nsd (dx, dy) - 1;
end;

function bodu_na_hranici : longint;
var i, b : longint;
begin
  b := n + pocet_vnitrnich_bodu_usecky (body[1], body[n]);
  for i := 1 to n - 1 do b := b + pocet_vnitrnich_bodu_usecky (body[i], body[i + 1]);
  bodu_na_hranici := b;
end;

function dvakrat_obsah_lichobeznika (a, b : bod) : longint;
begin
  dvakrat_obsah_lichobeznika := (a.y + b.y) * (a.x - b.x);
end;

function dvakrat_obsah_mnohouhelnika : longint;
var i, s : longint;
begin
  s := dvakrat_obsah_lichobeznika (body[n], body[1]);
```

```
  for i := 1 to n - 1 do s := s + dvakrat_obsah_lichobeznika (body[i], body[i + 1]);
  dvakrat_obsah_mnohouhelnika := abs (s);
end;

begin
  nacti_vstup;
  hranice := bodu_na_hranici;
  dobsah := dvakrat_obsah_mnohouhelnika;
  writeln (1 + (dobsah + hranice) div 2);
end.
```

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int N; cin >> N;
  vector<int> X(N+1), Y(N+1);
  for (int i=0; i<N; i++) cin >> X[i] >> Y[i];
  X[N]=X[0]; Y[N]=Y[0];
  int dvakrat_plocha = 0, na_obvode = 0;
  for (int i=0; i<N; i++) dvakrat_plocha += X[i]*Y[i+1]-X[i+1]*Y[i];
  for (int i=0; i<N; i++) na_obvode += __gcd( abs(X[i+1]-X[i]), abs(Y[i+1]-Y[i]) );
  cout << (dvakrat_plocha + na_obvode + 2)/2 << endl;
}
```

A-II-2 Babylonská kríza

Každý si určite všimol podobnosť s úlohou z domáceho kola. Táto podobnosť nie je náhodná, pretože v riešení tejto úlohy použijeme rovnakú dátovú štruktúru ako v riešení úlohy A-I-1. Táto štruktúra sa volá písmenkový strom, alebo tiež trie, a jej popis je vo vzorovom riešení práve spomenutej úlohy domáceho kola.

Zaveďme si označenie, ktoré nám uľahčí ďalší popis. W bude označovať počet wordlistov, L_W počet písmen vo všetkých wordlistoch a L_T dĺžku analyzovaného textu v znakoch.

Asi najjednoduchší spôsob, ktorý každého okamžite napadne, je postupne si pre každý jazyk spraviť samostatný písmenkový strom. Potom stačí prejsť vstupný text po jednotlivých slovách a pre každé slovo zistiť, v ktorých stromoch (a teda aj wordlistoch) sa nachádza. Každý taký výskyt započítame pre jednotlivé jazyky. Nakoniec nájdeme jazyk s maximálnym počtom takýchto výskytov a jednoducho vypíšeme odpoveď.

Takto priamočiary postup má zložitosť $O(L_T \cdot W + L_W)$, pretože každé slovo v texte vyhľadávame v písmenkovom strome toľko krát, koľko máme písmenkových stromov (a tých je toľko, koľko je wordlistov). Vytvorenie všetkých stromov nedokážeme spraviť lepšie ako v čase L_W, pretože každé slovo z každého wordlistu musí byť pridané do zodpovedajúceho stromu. Pamäťová zložitosť bude prinajhoršom $O(L_W)$.

To síce nie je zlé, ale určite vás napadla kopa zlepšení. Jedným by mohlo byť to, že postavíme spoločný písmenkový strom pre všetky wordlisty a pri každom slove si zapamätáme, do ktorých wordlistov patrí. Toto sa dá implementovať napríklad tak, že v každom vrchole budeme mať zoznam jazykov, ktoré obsahujú slovo zodpovedajúce dotyčnému vrcholu.

Analyzovaný text potom znova prečítame po jednotlivých slovách. Keď nájdeme slovo v strome, nebudeme hneď prechádzať cez všetky jazyky, ktoré ho obsahujú, ale len si v danom vrchole zvýšime počítadlo počtu výskytov dotyčného slova. Až keď dočítame celý text, tak raz celý strom prejdeme (ideálne prehľadávaním do hĺbky) a pri tom spracujeme pre každé slovo, ktoré sa v texte vyskytovalo, všetky jeho výskyty naraz.

Je dôležité si uvedomiť, že vytvorenie takého písmenkového stromu a zároveň prehľadávanie na konci nezaberie viac než $O(L_W)$. Pridané spájané zoznamy totiž celkovú časovú ani pamäťovú zložitosť neovplyvnia. Je to preto, že každý prvok v zozname zastupuje jedno slovo. V strome tak máme najviac $O(L_W)$ normálnych vrcholov a najviac $O(L_W)$ prvkov spájaného zoznamu. Pri vytváraní aj pri záverečnom prehľadávaní tak každý vrchol spracovávame práve raz, takže celková časová zložitosť je $O(L_W + L_T)$. Pamäťová je z rovnakých príčin $O(L_W)$.

Lepšiu asymptotickú časovú zložitosť nevieme dosiahnuť, pretože vstupné súbory musíme aspoň raz prečítať.

Aj keď vieme, že ďalšie zrýchlenie sa už nedá dosiahnuť, môžeme sa pokúsiť o implementačne jednoduchšie riešenie. V práve popísanom algoritme sme si pre každé slovo pamätali, v ktorých wordlistoch sa nachádza. Skúsme túto informáciu zo stromu úplne vynechať.

Po jednom prečítaní vstupného textu budeme o každom slove vedieť, koľkokrát sa v texte vyskytuje. Nevieme len, v ktorých wordlistoch sa tieto slová nachádzajú. Nič nám ale nebráni v tom, aby sme si prečítali všetky wordlisty ešte raz. Keď teraz čítame wordlist, každé jeho slovo v strome nájdeme a zodpovedajúcemu jazyku prirátame jeho počet výskytov v texte.

Takéto prejdenie bude trvať $O(L_W)$ krokov, takže to celkovú časovú zložitosť nepokazí. Navyše však bude tento algoritmus omnoho jednoduchší na implementáciu.

Listing programu:

```
#include <stdio.h>
#include <stdlib.h>

#define WORD_LEN 255
#define ALPHABET_SIZE 26

struct TRIE_NODE {
      TRIE_NODE *next[ALPHABET_SIZE];
      int count;
   };

struct DICT_DESC {
      char name[WORD_LEN];
      int size;
   };

void trie_add(TRIE_NODE *root, const char *word)
{
   TRIE_NODE *cur_root = root;

   while (*word)
   {
      if (!cur_root->next[*word-'a'])
      {
         cur_root->next[*word-'a'] = malloc(sizeof(TRIE_NODE));
         for (int i=0; i<ALPHABET_SIZE; i++)
            cur_root->next[*word-'a']->next[i] = NULL;
         cur_root->next[*word-'a']->count = 0;
      }

      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   //Nie je potrebne pamatat si konce slov, v zaverecnom pocitani sa hodnoty vo vrcholoch zanedbaju
}

void trie_increase_word_count(TRIE_NODE *root, const char *word)
{
   TRIE_NODE *cur_root = root;    //posledna faza, mame istotu, ze slovo najdeme
   while (*word && cur_root)
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   if (cur_root)
      cur_root->count++;
}

int trie_find(TRIE_NODE *root, const char *word)
//najdeme slovo v strome vratime, ze kolko krat sa vyskytlo
{
   TRIE_NODE *cur_root = root;    //posledna faza, mame istotu, ze slovo najdeme

   while (*word)
   {
      cur_root = cur_root->next[*word-'a'];
      ++word;
   }
   return cur_root->count;
}
```

```
void trie_free(TRIE_NODE *root)  //uvolnenie struktur
{
   if (!root)
      return;
   for(int i=0; i<ALPHABET_SIZE; i++)
      trie_free(root->next[i]);
   free(root);
}

TRIE_NODE *trie_root;
int N;

int main(void)
{
   FILE *fdict = fopen("slovniky.in", "r");
   FILE *ftext = fopen("text.in", "r");

   trie_root = malloc(sizeof(TRIE_NODE));
   for (int i=0; i<ALPHABET_SIZE; ++i)
      trie_root->next[i] = NULL;
   trie_root->count = 0;

   //inicializacia pismenkoveho stromu
   int dict_count;
   fscanf(fdict, "%d", &dict_count);
   for (int dict=0; dict<dict_count; dict++)
   {
      int dict_size;
      char word[WORD_LEN+1];
      fscanf(fdict, "%s %d", word, &dict_size);
      for(int cur_word=0; cur_word<dict_size; cur_word++)
      {
         fscanf(fdict, "%s", word);
         trie_add(trie_root, word);
      }
   }

   //zistenie poctu slov v texte
   while( !feof(ftext))
   {
      char word[WORD_LEN+1];
      fscanf(ftext, " %s ", word);
      trie_increase_word_count(trie_root, word);
   }

   //zistenie poctu slov v slovniku
   fclose(fdict);
   fdict = fopen("slovniky.in", "r");
   fscanf(fdict, "%d", &dict_count);
   DICT_DESC *dict_desc = malloc(sizeof(DICT_DESC) * dict_count);

   for (int dict=0; dict<dict_count; dict++)
   {
      int dict_size;
      fscanf(fdict, "%s %d", dict_desc[dict].name, &dict_size);
      dict_desc[dict].size = 0;

      for(int cur_word=0; cur_word<dict_size; cur_word++)
      {
         char word[WORD_LEN+1];
         fscanf(fdict, "%s", word);
         dict_desc[dict].size+= trie_find(trie_root, word);
      }
   }

   //najdenie slovniku s najvacsim poctom slov
   int max = 0;
   for (int dict=0; dict<dict_count; dict++)
      if (dict_desc[dict].size>max)
         max = dict_desc[dict].size;

   //vypisanie slovnikov
   FILE *fout = fopen("jazyk.out", "w");
   for (int dict=0; dict<dict_count; dict++)
      if (dict_desc[dict].size==max)
         fprintf(fout, "%s\n", dict_desc[dict].name);
   fclose(fout);
```

```
   //uvolnenie stromu
   trie_free(trie_root);
   return 0;
}
```

A-II-3 Cyklistické preteky

Hneď po prečítaní zadania je jasné, že táto úloha bude mať niečo spoločné s grafovými algoritmami. Mestá budú vrcholy, možné etapy pretekov budú hrany a počty divákov si môžeme predstaviť ako ohodnotenie hrán nezápornými celými číslami. Ako ale nájdeme riešenie?

Začneme prvou časťou úlohy, teda nájdením nejakej trasy pretekov z Vyšných Hákov do Veľkého Sumca s čo najmenej etapami (ale bez požiadavky na maximalizáciu počtu divákov). Takto zadaná úloha je v podstate učebnicovým príkladom na prehľadávanie grafu do šírky. Pre každé mesto si spočítame minimálny počet etáp nutných na jeho dosiahnutie a to tak, že Vyšné Háky dostanú nulu, ich susedia jednotku, susedia susedov dvojku atď. V grafe nám tak vzniknú akési vrstvy, pričom v k-tej vrstve sú vrcholy, do nich sa dá dostať najkratšou cestou na k etáp. Nájdenie nejakej trasy je potom jednoduché: Začneme z posledného vrcholu, teda z Veľkého Sumca, potom zoberieme nejakého jeho suseda vo vrstve o jedna nižšej a tak pokračujeme, pokiaľ sa nedostaneme do Vyšných Hákov. Toto riešenie vyžaduje lineárny čas a lineárne množstvo pamäte, tj. $O(N + M)$.

A ako nájdeme tú divácky najatraktívnejšiu trasu? Pre každé mesto už vieme najmenší počet etáp, na ktorý sa doň vieme dostať. Teraz navyše pre každé mesto spočítame, koľko najviac divákov môže vidieť niektorú najkratšiu trasu z Vyšných Hákov do tohto mesta.

Postup, ako tento údaj spočítame, bude podobný tomu z predchádzajúcich odsekov. Pre mestá susediace s Vyšnými Hákmi je to priamo počet divákov na etape medzi nimi (keďže musíme použiť najmenší počet etáp, nemáme na výber). Potom postupne spracujeme mestá, kam sa vieme dostať na 2 etapy, potom na 3, atď. Keď teraz zisťujeme optimálny počet divákov pre konkrétne mesto X vo vzdialenosti k etáp od Vyšných Hákov, nájdeme si všetkých jeho susedov $Y_1, \ldots, Y_l$, do ktorých sa z Vyšných Hákov dalo dostať na $k - 1$ etáp. Najlepšia cesta do X určite vyzerá tak, že na $k - 1$ etáp prídeme (optimálnym spôsobom, ktorý už v tomto okamihu poznáme) do niektorého mesta Y_i, a následne z neho k-tou etapou do X. Z týchto l možností si vyberieme tú najlepšiu.

Zjavne každý vrchol spracujeme práve raz a každú hranu práve dvakrát, preto je časová aj pamäťová zložitosť lineárna od veľkosti daného grafu – teda $O(M + N)$. (Aby sme dosiahli túto časovú a pamäťovú zložitosť, graf máme uložený ako zoznam okolí vrcholov – pre každý vrchol si pamätáme zoznam hrán, ktoré z neho vedú.)

Za zmienku ešte stojí malý trik, ako program rieši výpis trasy – namiesto hľadania trasy od konca a následného otočenia poradia miest, ako by naznačoval algoritmus, nájdeme cestu z Veľkého Sumca do Vyšných Hákov a pri hľadaní trasy od konca ju rovno vypisujeme – takže vypíšeme trasu z Vyšných Hákov do Veľkého Sumca.

Na konci uvádzame aj alternatívne riešenie v C++, ktoré obe časti spája do jednej a počas prehľadávania do šírky rovno spočíta aj optimálny počet divákov.

Listing programu:

```
program zavod;

const maxN = 1000000;
      maxM = 1000000;
      zacatek = 2;  { prohodili jsme startovni a cilove mesto, abychom }
      konec = 1;    { nemuseli otacet cestu }

var M,N:integer;

    mesta: array [1..maxN] of record { informace o mestech }
            vzdalenost : integer;   { pocet etap od zacatku }
            maxDivaku : integer;    { pocet divaku, ktery shledne trasu az sem}
            predchozi:integer;      { predchozi mesto na nejlepsi trase }
            stupen: integer;        { pocet sousednich mest}
            zacatek: integer;       { pozice prvniho souseda v poli sousede }
            pozice:integer; {docasna hodnota pouzivana pri nacitani}
        end;
```

```
    sousede: array[1..2*maxM] of record {pole sousedu. nejprve jsou ulozeni }
                { sousede mesta 1, pak sousede mesta 2, atd. }
            mesto, divaku:integer;
        end;
    hrany: array[1..maxM] of record   { pole hran, neboli moznych etap. Pouzito}
                                      { jen pri nacitani. }
        odkud, kam, divaku: integer;  { odkud a kam etapa vede a pocet divaku }
        end;

{ Nacteni dat ze vstupu - naplni pole mesta a sousede. }
procedure nacti;
var i, pozice:integer;
    a,b, divaku:integer; {mesta}
    p:integer;
begin
    readln(N, M);
    for i := 1 to N do begin
        mesta[i].stupen := 0;
    end;
    {nacti hrany a spocti stupne}
    for i:= 1 to M do begin
        readln(a, b, hrany[i].divaku);
        hrany[i].odkud := a; hrany[i].kam := b;
        mesta[a].stupen := mesta[a].stupen + 1;
        mesta[b].stupen := mesta[b].stupen + 1;
    end;
    { urci zacatky pro jednotliva mesta v poli sousede }
    pozice := 1; {prubezna pozice v poli sousede}
    for i := 1 to N do begin
        mesta[i].zacatek := pozice;
        mesta[i].pozice := pozice;
        pozice := pozice + mesta[i].stupen;
    end;
    { napln pole sousede }
    for i:= 1 to M do begin
        a := hrany[i].odkud; b := hrany[i].kam; divaku := hrany[i].divaku;
        {pridej souseda mestu a}
        p:= mesta[a].pozice;
        sousede[p].mesto := b;
        sousede[p].divaku := divaku;
        mesta[a].pozice := p+ 1;
        {pridej souseda mestu b}
        p:= mesta[b].pozice;
        sousede[p].mesto := a;
        sousede[p].divaku := divaku;
        mesta[b].pozice := p+ 1;
    end;

end;

{ Fronta mest ke zpracovani. Vzhledem ke zpusobu pridavani plati, ze }
{ mesto s nizsi vzdalenosti od zacatku je v poli na nizsi pozici nez }
{ mesto s vetsi vzdalenosti. }
var fronta: array[1..maxN] of integer;
    zacF,konF:integer; {zacatek a konec fronty}

{ Projde sousedy vybraneho mesta a pokusi se prodlouzit do nich trasu. }
procedure projdiSousedy(mesto:integer);
var i: integer; {index do pole sousedu}
    posledni: integer; {index posledniho souseda v poli sousede. }
    divaku, soused : integer;
begin
    posledni := mesta[mesto].zacatek + mesta[mesto].stupen - 1;
    for i:= mesta[mesto].zacatek to posledni do begin
        divaku := mesta[mesto].maxDivaku + sousede[i].divaku;
        soused := sousede[i].mesto;
        if mesta[soused].vzdalenost = -1 then begin
            {mesto jsme jeste nenavstivili.}
            konF := konF + 1;
            fronta[konF] := soused; {dej do fronty}
            mesta[soused].vzdalenost := mesta[mesto].vzdalenost + 1;
            mesta[soused].maxDivaku := divaku; { a toto je zatim nejlepsi trasa}
            mesta[soused].predchozi := mesto;
        end else if (mesta[soused].vzdalenost = mesta[mesto].vzdalenost + 1)
            and (mesta[soused].maxDivaku < divaku) then begin
            { trasa pses toto mesto je lepsi, prenastavime hodnoty u souseda.}
            mesta[soused].maxDivaku := divaku;
```

```
            mesta[soused].predchozi := mesto;
        end;
    end;
end;

{ Spocte hodnoty maxDivaku, vzdalenost a predchozi v poli mesta, cimz }
{ prakticky vyresi celou ulohu. }
procedure spocti;
var mesto: integer;
    i:integer;
begin
    {inicializace}
    for i := 1 to N do begin
        mesta[i].maxDivaku := 0;
        mesta[i].vzdalenost := -1; {zatim jsme nikde nebyli}
    end;
    {pocatecni mesto ma vzdalenost 0 a zadne divaky}
    mesta[zacatek].vzdalenost := 0;
    mesta[zacatek].maxDivaku := 0;
    fronta[1] := zacatek;
    zacF := 1; konF := 1;
    {dokud neni fronta prazdna}
    while zacF <= konF do begin
        mesto := fronta[zacF];
        zacF := zacF + 1;  {precti hodnotu z fronty}
        projdiSousedy(mesto);
    end;
end;

{ Vypise cestu od konce do zacatku. Protoze cesta, kterou najde spocti(), }
{ konci ve Vysnych Hacich, tak je vysledkem presne to, co pozaduje zadani. }
procedure vypis;
var mesto:integer;
begin
    writeln(mesta[konec].vzdalenost, ' ', mesta[konec].maxDivaku);
    mesto := konec;
    write(konec);
    while mesto <> zacatek do begin   {dokud nejsme na zacatku}
        mesto := mesta[mesto].predchozi; {najdi mesto, ze ktereho jsme prisli}
        write(' ', mesto);       {vypis ho}
    end;
    writeln;
end;


{ Hlavni program }
begin
    nacti;
    spocti;
    vypis;
end.
```

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
struct hrana { public: int ciel, pocet; };

int main() {
  int N, M; cin >> N >> M;
  vector< vector< hrana > > G(N);
  // nacitaj vstup
  for (int i=0; i<M; i++) {
    int x; hrana E;
    cin >> x >> E.ciel >> E.pocet;
    x--; E.ciel--;
    G[x].push_back(E);
    swap(x,E.ciel);
    G[x].push_back(E);
  }
  // prehladaj do sirky
  vector<int> vzdialenost(N,987654321), divakov(N), odkial(N);
  queue<int> Q;
```

```
  Q.push(0); vzdialenost[0]=divakov[0]=odkial[0]=0;
  while (!Q.empty()) {
    int kde = Q.front(); Q.pop();
    for (int i=0; i<int(G[kde].size()); i++) {
      int kam = G[kde][i].ciel, kolko = G[kde][i].pocet;
      if (vzdialenost[kam] > vzdialenost[kde]+1) {
        vzdialenost[kam] = vzdialenost[kde]+1;
        divakov[kam] = divakov[kde] + kolko;
        odkial[kam] = kde;
        Q.push(kam);
      }
      if (vzdialenost[kam] == vzdialenost[kde]+1) if (divakov[kam] < divakov[kde] + kolko) {
        divakov[kam] = divakov[kde] + kolko;
        odkial[kam] = kde;
      }
    }
  }
  // vypis vzdialenost a pocet divakov
  cout << vzdialenost[1] << " " << divakov[1] << endl;
  // zostroj a vypis cestu
  vector<int> cesta(1,1);
  while (cesta.back()!=0) cesta.push_back(odkial[cesta.back()]);
  reverse(cesta.begin(),cesta.end());
  for (int i=0; i<int(cesta.size()); i++) cout << (i?" ":"") << (cesta[i]+1); cout << endl;
}
```

A-II-4 Zásobníkové počítače

Vzorové riešenie používa len jeden zásobník. V ňom si budeme počas výpočtu udržiavať čo najjednoduchší výraz, ktorý je ekvivalentný s doteraz prečítanou časťou vstupu.

Zabudnime na chvíľku na zátvorky (budeme predpokladať, že nie sú na vstupe). Výraz budeme načítavať po znakoch. Ak načítame hodnotu `0` alebo `1`, uložíme ju na vrch zásobníka. Ak načítame nejaký operátor, tak výraz v zásobníku trošku zjednodušíme a potom načítaný operátor vložíme na vrch zásobníka. V zásobníku sa nám budú striedať operátory a hodnoty, pričom na spodku zásobníka bude hodnota.

Zjednodušovanie výrazov: Ak načítame operátor `|`, tak vtedy môžeme celý výraz, ktorý je uložený v zásobníku vyhodnotiť. Vieme to spraviť napríklad tak, že vyberieme vrchné 3 prvky zo zásobníka, ktoré budú mimochodom vyzerať ako **`hodnota, operátor, hodnota`**, spracujeme[3] a výslednú hodnotu vložíme na vrch zásobníka. Toto budeme opakovať až kým nebude zásobník obsahovať iba jednu hodnotu. To môžeme považovať za dostatočne zjednodušený výraz.

Všimnite si, že ak by v zásobníku boli za sebou (oddelené číslom) dva operátory `&` a nad nimi operátor `|`, tak by sme týmto postupom mohli prísť k zlému výsledku[4]. Preto treba zaručiť, aby sa za sebou nenachádzalo dva a viac operátorov `&`. Kvôli tomu, vždy ak načítame operátor `&`, tak predtým než ho vložíme na vrch zásobníka skontrolujeme, či by neboli za sebou dva operátory `&` (vtedy zásobník vyzerá ako **`hodnota, &, hodnota`**). Ak hej, tak danú trojicu vyberieme zo zásobníka, vyhodnotíme, a výsledok vložíme na vrch zásobníka. Toto budeme opakovať až kým nebude zásobník obsahovať iba jednu hodnotu, alebo nenarazíme na operátor `|`.

Zátvorky: Teraz sa musíme popasovať so zátvorkami. Ak načítame ľavú zátvorku, musíme rekurzívne vyhodnotiť výraz medzi touto zátvorkou a správnou pravou zátvorkou. Zásobník nám pri tom veľmi pomôže. Jednoducho vložíme zátvorku do zásobníka a budeme pokračovať vo vyhodnocovaní výrazov tak, ako sme popísali vyššie (jediný rozdiel je, že pri zjednodušovaní výrazov budeme považovať za prázdny zásobník aj zásobník, na ktorý má na vrchu ľavú zátvorku).

Keď načítame pravú zátvorku, tak vyhodnotíme výraz na zásobníku až po najbližšiu ľavú zátvorku (tak, ako pri zjednodušovaní výrazu pri načítaní `|`) a výsledok vložíme do zásobníka (čiže sme nahradili celú zátvorku jej hodnotou).

Keď dočítame vstup, tak nám ostane v zásobníku nejaký výraz, ktorý stačí vyhodnotiť presne tak, ako pri načítaní operátora `|`. Potom ostane v zásobníku práve jedna hodnota, ktorá sa rovná hodnote výrazu zo vstupu.

Tento algoritmus zjavne používa iba jeden zásobník. Časová zložitosť je trosku zložitejšia. Všimnite si, že každý operátor zo vstupu dáme najviac jeden krát do zásobníka a najviac jeden krát ho potom vyberieme.

[3] Vyhodnotíme výraz, ktorý reprezentujú

[4] Skúste si nájsť taký príklad.

Medzi vloženiami a výbermi vždy spravíme konštantný počet operácii a preto je časová zložitosť lineárna.

Listing programu:

```
program vyraz;
var s : stack of char;
    c : char;

{ Vrati operator na vrchu zasobnika. }
{ Ak tam nie je ziadny, vrati '#'. }
function vrchny_op : char;
var num, op : char;
begin
   num := pop(s);
   if empty(s) then op := '#'
               else begin op := pop(s); push(s, op); end;
   push(s, num);
   vrchny_op := op;
end;

{ Vyhodnoti operator na vrchu zasobnika. }
procedure vyhodnot;
var a, b, op, vysledok : char;
begin
   b := pop(s);
   op := pop(s);
   a := pop(s);
   if op = '&' then vysledok := a='1' and b='1';
   if op = '|' then vysledok := a='1'  or b='1';
   if op = '(' then vysledok :=           b='1';
   if vysledok then push(s, '1') else push(s, '0');
end;

begin
   while read(c) do case c of
     '0' ,
     '1' : push(s, c);        { Cislo ulozime na zasobnik. }
     '&' : begin              { Vyhodnotit vsetky '&'. }
             while vrchny_op = '&' do vyhodnot;
             push(s, c);
           end;
     '|' : begin              { Vyhodnotit vsetky '&' alebo '|'. }
             while vrchny_op in ['&', '|'] do vyhodnot;
             push(s, c);
           end;
     '(' : begin              { Pred zatvorkou ulozime pomocnu nulu. }
             push(s, '0');
             push(s, c);
           end;
     ')' : begin              { Vyhodnotit az do '('. }
             while vrchny_op <> '(' do vyhodnot;
             vyhodnot;        { Odstrani '(' z vrchu zasobniku. }
           end;
   end;
   while vrchny_op <> '#' do vyhodnot;
   write(pop(s));
end.
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX