Riešenia kategórie A

A-I-1 Medzery

Prvý pokus

Najprv si potrebujeme uvedomiť, že jediná podstatná vec na vstupe sú jednotlivé slová a ich počet. Riadkovanie môžeme ignorovať. Dokonca ak by sme riešenie nemuseli vypisovať, stačilo by si pamätať len dĺžky jednotlivých slov. Tu sa dostávame k prvému problém. Pravdepodobne budeme niekde v riešení potrebovať rýchlo zisťovať počty písmen v i za sebou idúcich slovách. Na to nám poslúžia prefixové súčty. Vyrobíme si tabuľku, ktorá bude mať na i-tej pozícií počet písmen v prvých i slovách (pre i od 0 po n, kde n je celkový počet slov). Potom ak chceme zistiť počet písmen od i-teho do j-teho slova, tak sa len pozrieme na hodnotu prefix[j]-prefix[i-1]. Toto vieme predrátať v čase $O(n)$ s pamäťou $O(n)$, čo nám riešenie nemôže pokaziť, nakoľko je to čas a pamäť potrebná na načítanie vstupu.

Teraz sa poriadne pozrieme na limity. Vidíme, že 3 body vieme dostať aj za celkom pomalé riešenie. Môžeme teda skúsiť vymyslieť brute-force. Napríklad vyskúšať všetky možnosti na zlomy riadkov. Nakoľko máme zaručené, že optimálne riešenie má najviac $k = 5$ riadkov a na vstupe je maximálne $n = 50$ slov, mohli by sme teda stihnúť vyskúšať všetky možnosti, medzi ktoré slová dáme nové riadky (tie budú najviac 4). Týchto možností je približne $\binom{49}{4} = 211\,876$. Každú ale musíme ešte otestovať, čo nám zaberie tak 50 operácií a v súčte (súčine) dostaneme približne 10 000 000 operácií, čo sa ešte stále do časového limitu zmestí. Časová zložitosť takéhoto riešenia je vo všeobecnosti $O(n\binom{n-1}{k-1})$. Pamäťová dosť závisí od implementácie, obzvlášť od toho, ako prezeráme všetky možnosti. Bude to ale približne $O(n)$.

Pažravé riešenie

Čo tak úlohu riešiť pažravo? Budeme postupne brať slová a kým sa zmestia do jedného riadku, tak ich tam budeme dávať. Čím plnší riadok, tým menšia škaredosť.

Takýto postup síce zväčša vyrobí dobré riešenie, nebude však nutne optimálne. Prečo to nezafunguje? Ak by sme napríklad mali 31 slov dĺžky 1 a potom dve dĺžky 29, tak sa stane nasledovná šarapata. V prvom riadku budeme mať 30 jednopísmenových slov a škaredosť tohoto riadku bude 0.034. V druhom budeme mať jedno jednopísmenové a jedno 29 písmenové. Tento má škaredosť 841. No a posledný má 0. Ak by sme ale čo i len jedno slovo z prvého riadku dali do druhého, tak v prvom máme škaredosť 0.32, a v druhom 364.5, čo je dokopy oveľa menej. Pažravý (greedy) prístup teda nefunguje. (Ale keď nevieme nič lepšie, tak sa ho stále oplatí implementovať a zobrať zaň slušné množstvo bodov. Obzvlášť v kombinácii s hrubou silou pre malé vstupy.)

Hlbšie zamyslenie

Prečo vlastne pažravé riešenie nefungovalo? Preto, lebo sa nám môže oplatiť zobrať do riadku aj menej slov ako sa tam práve zmestí. To, koľko presne, ale závisí od toho, ako dobre vieme potom nasádzať zvyšok.

Povedzme, že chceme nasádzať n slov. Máme dve možnosti. Buď sa všetky zmestia do jedného riadku (a teda majú škaredosť 0), alebo nie. Prvý prípad je ľahký – nič lepšie ako dať ich do jedného riadku nevymyslíme. V tom druhom prípade sa musíme rozhodnúť: koľko slov dáme do prvého riadku a koľko vysádzame potom (do ostatných niekoľkých riadkov)?

Ak by sme ale toto mali porátané pre menšie n-ká, tak porátať to pre to naše aktuálne n je ľahké. Stačí prejsť všetky možnosti m od 1 po n slov na prvom riadku, pozrieť sa, či je tento riadok validný (má apsoň dve slová a zmestí sa do 60 znakov), porátať jeho škaredosť, pozrieť sa do tabuľky, na akú najmenšie škaredosť sa dá vysádzať zvyšných $n - m$ slov a vybrať to najlepšie m-ko. Musíme si len dať pozor, aby sme škaredosť rátali rozumne a teda pomocou prefixových (respektíve sufixových) súm. Na záver už len podľa zapamätaných údajov spätne zostrojíme jedno optimálne riešenie.

Formálne si toto riešenie môžeme sformulovať nasledovne: Slová na vstupe si očíslujeme od 1 po n. Nech s_i je najmenšia možná škaredosť, ktorú vieme dosiahnuť, keď máme slová s číslami i až n vysádzať do ľubovoľného

počtu celých riadkov. Budeme postupne počítať hodnoty s_n až s_1 (čo je hľadaná celková optimálna cena). Pre každú z týchto hodnôt sme si odvodili nasledovný vzťah:

$$s_i = \min_{1 \le m \le n-i+1} (\text{škaredosť riadku obsahujúceho slová } i \text{ až } i+m-1) + s_{i+m}$$

(pričom škaredosť považujeme za nekonečnú, ak sa dotyčné slová do riadku nezmestia).

Toto riešenie má časovú zložitosť $O(n^2)$, lebo pre každý počet slov $O(n)$ sa potrebuje pozrieť na všetky možnosti toho, koľko slov je v prvom riadku $O(n)$. Nakoľko si len pamätáme tabuľku s výsledkami pre jednotlivé n-ká, tak pamäťová zložitosť bude $O(n)$.

Posledné zlepšenie

Koľko slov je ale vôbec rozmné skúšať dať do jedného riadku? No toľko, koľko sa tam zmestí. Nakoľko má ale každé slovo dĺžku aspoň 1, tak viac ako 30 slov do riadku nikdy nezmestíme. To znamená, že pre každé n sa nám stačí pozrieť len na 30 (konštantne veľa) možností pre počet slov v prvom riadku. Tak sa rázom dostávame na časovú zložitosť $O(n)$ a so správne napísaným kódom aj na plný bodový zisk.

Listing programu (C++)

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

#define For(Q,W) for(int Q=0; Q<W; Q++)

int main(){
    vector<string> text;
    string slovo;
    while( cin>>slovo) text.push_back(slovo); //nacitame text
    int pocetslov = text.size();
    reverse(text.begin(),text.end()); //prevratime poradie slov, lebo ideme ratat od konca

    vector<int> prefixsum(1,0);
    For(i,pocetslov){
        prefixsum.push_back(prefixsum[i]+text[i].size()); // vyratame si prefixove sumy dlzky slov
    }

    vector<int> vriadku(pocetslov);
    vector<double> skaredost (pocetslov+1,99999999); // na zaciatku mame velmi velku skaredost
    For(i,pocetslov){
        // chceme vysadzat i+1 slov
        if(i-1+prefixsum[i+1]<=60){   // ak sa vsetky zmestia do riadku (je to posledny riadok)
            skaredost[i+1]=0.0;       // skaredost je 0, lebo je to posledny riadok
            vriadku[i]=i+1;           // v ktorom je i+1 slov
        }
        else{
            // rozhodneme sa, kolko slov dame do tohoto riadku, pricom su to aspon dva, ale maximalne 30
            // maximalne 30 je ich preto, lebo kedze kazde slovo ma aspon jedno pismeno, tak viac ako 30
            // slov ma viac ako 30 pismen + 29 medzier co je aspon 61
            for(int j=2; j<=30 && i-j>=0; j++){
                // ak sa to nezmesti do jedneho riadku, tak nema cenu skusat viac slov
                if(prefixsum[i+1]-prefixsum[i-j+1]+j-1 >60) break;
                // skaredost toho, co nedame do prveho riadku
                double nova = skaredost[i-j+1];
                // zratame skaredost pre j slov v tomto riadku
                double medzera = (60.0-prefixsum[i+1]+prefixsum[i-j+1])/(j-1.0);
                nova+= (medzera-1)*(medzera-1)*(j-1);
                if(skaredost[i+1]> nova){      // ak sme si pomohli
                        skaredost[i+1]=nova;   // tak updatneme skaredost
                        vriadku[i]=j;          // a zapamatame si pocet slov na riadku
                }
            }
        }
    }
    //uz len spravne vypiseme text zo vstupu
    for(int i=pocetslov-1; i>=0; i--){
            cout<< text[i];
            int vypis= vriadku[i]-1; //jedno sme uz vypisali
            For(j,vypis){
                i--;
                cout<<" "<<text[i];
            }
            cout<<endl;
    }
}
```

A-I-2 Gaštanové zvieratko

Najprv si ukážeme ideu drevorubačského riešenia, ktoré je navyše pomalé.

Dá sa ukážať, že ak je zvieratko súvislé, tak vždy z neho vieme odobrať gaštan. Napríklad indukciou od počtu gaštanov dokážeme, že v každom zvieratku, ktoré má aspoň dva gaštany, existujú aspoň dva rôzne gaštany, z ktorých každý môžeme odobrať bez porušenia súvislosti.

Pre zvieratko s 2 gaštanmi je to zrejmé. Teraz predpokladajme, že tvrdenie platí pre všetky zvieratká, čo majú maximálne k gaštanov, a dokážeme, že potom platí aj pre zvieratká z $k+1$ gaštanov. Ak z dotyčného zvieratka môžeme odobrať ľubovoľný gaštan, vyhrali sme. A čo ak nie? Nech g je gaštan, ktorého odobraním sa naše zvieratko rozpadne na aspoň dve časti. Zoberme si dve tieto časti. Pre každú z nich uvažujme nasledovne: Keby som mal len túto časť a gaštan g, tak mám zvieratko tvorené maximálne k gaštanmi. Viem teda, že v ňom existujú aspoň dva gaštany, ktoré môžem odobrať bez toho aby sa rozpadlo. Aspoň jeden z týchto gaštanov nie je g. A tento gaštan sme zjavne mohli odobrať aj z pôvodného, $(k+1)$-gaštanového zvieratka bez toho aby sa rozpadlo. A keďže máme aspoň dve časti a v každej takýto gaštan, máme aspoň dva odoberateľné gaštany, čo sme chceli dokázať.

Z práve dokázaného tvrdenia vyplýva dôležité pozorovanie: keď nájdeme gaštan, ktorý sa dá odobrať, vždy ho naozaj odobrať môžeme – keďže každé zvieratko má odoberateľný gaštan, na poradí odoberania nezáleží, nemáme ako čo pokaziť.

Na tom môžeme teraz postaviť riešenie hrubou silou: Kým má zvieratko ešte nejaké gaštany, tak si nájdeme gaštan, ktorý môžeme odobrať, a odoberieme ho. A ako nájsť vhodný gaštan? Predsa opäť hrubou silou. Postupne skúšame všetky, kým nenájdeme vhodný. Vhodnosť overíme otestovaním, či zvyšok zvieratka drží pokope.

Zvieratko drží celé pokope, keď medzi každými dvoma gaštanmi existuje postupnosť paličiek, ktorá ich spája. (Odborne, keď všetky gaštany ležia v tom istom komponente súvislosti.) Rozmyslite si ale, že toto netreba testovať zvlášť pre každé dva gaštany. Stačí si zvoliť jeden gaštan a otestovať, či sa z neho po paličkách dostaneme na všetky ostatné.

Toto overenie sa dá urobiť pomocou prehľadávania do hĺbky, ktoré má nasledovný pseudokód:

```
def hladaj(x):
  boli_sme[x] = True
  for sused in susedia[x]:
    if boli_sme[sused] == False:
      hladaj(sused)

hladaj(0)
```

Ak je `boli_sme` pre všetky gaštany `True`, potom zvieratko drží pokope. V tomto kóde sa v podstate z každého gaštanu pozrieme do všetkých jeho susedov a overenie rekurzívne pustíme zo susedov, kde sme ešte neboli. Keďže každý gaštan spracujeme len raz, celková zložitosť je $O(n+m)$.

Celková zložitosť tohoto riešenia je potom $O(n^2(n+m))$, čo je dosť pomalé.

Lepšie riešenie

Idea lepšieho riešenia vyzerá nasledovne. Zoberieme pôvodné zvieratko a z ľubovoľného vrcholu spustíme prehľadávanie do hĺbky. Určite existuje gaštan, z ktorého sa už rekurízvne nezavoláme ďalej. (Napríklad má túto vlastnosť gaštan, ktorý spracujeme ako úplne posledný – ale môže takých byť aj viac.) Ľubovoľný takýto gaštan môžeme určite odstrániť, lebo neodpadne žiadna časť zvieratka. Dá sa to predstaviť aj tak, že prehľadávanie do hĺbky nám na zvieratku nakreslí strom a my odstraníme list stromu a to nevadí, lebo ostatné gaštany sú spojené zvyškom stromu.

Toto dáva návod na trochu rýchlejšie riešenie: Pustíme prehľadávanie do hĺbky a odstraníme gaštan z ktorého sa nepohneme ďalej. Toto opakujeme, kým máme gaštany. Celková časová zložitosť by bola $O(n(n+m))$.

Vzorové riešenie

Toto riešenie sa dá ešte zlepsiť. Idea je nasledovná: Bude nám stačiť len jedno jediné prehľadávanie do hĺbky. Počas neho postupne odstránime všetky gaštany. Gaštan odstránime v momente, keď ukončujeme funkciu hľadaj. Toto sposobí nasledovné: Ak sa nevoláme nikde ďalej, tak máme prípad opísaný vyššie. Ak sa voláme ďalej, tak už všetky gaštany odtiaľ boli odstránené a máme akokeby situáciu, keď sme sa nemali kam ďalej zavolať.

Toto celé nás stojí jedno prehľadanie do hĺbky sa časová zložitosť je $O(n + m)$. Rovnaká je aj pamäťová zložitosť.

Ešte jednoduchšie vzorové riešenie

Nepotrebujeme vlastne použiť prehľadávanie do hĺbky, zafungovalo by aj prehľadávanie do šírky alebo ľubovoľné iné. Počas prehľadávania nič nerobíme, len si zapisujeme poradie, v ktorom gaštany prvýkrát navštívime. (Všimnite si, že do každého gaštanu okrem prvého sme prišli z nejakého, ktorý je zapísaný skôr.) No a teraz ako odpoveď jednoducho odzadu vypíšeme poradie, ktoré sme práve dostali.

Prečo to funguje? Keď odstránime posledných k gaštanov v tomto poradí, bude prehľadávanie zvyšku zvieratka prebiehať presne rovnako, a teda všetkých zvyšných $n - k$ gaštanov zjavne drží pokope.

Listing programu (C++)

```
#include <cstdio>
#include <vector>

using namespace std;
vector<vector<int>> g;
vector<bool> visited;

void dfs(int x) {
  visited[x] = true;
  for (int i = 0; i < g[x].size(); i++) {
    if (visited[g[x][i]]) continue;
    dfs(g[x][i]);
  }
  printf("%d ", x+1);
}

int main() {
  int n, m; scanf("%d %d", &n, &m);
  g.resize(n);
  visited.resize(n);
  for (int i = 0; i < m; i++) {
    int a, b; scanf("%d %d", &a, &b); a--; b--;
    g[a].push_back(b);
    g[b].push_back(a);
  }
  dfs(0);
  printf("\n");
}
```

A-I-3 Stratil sa paket

Podúloha A

Priamočiare riešenie, ktoré človeku pravdepodobne napadne ako prvé, by si v poli premenných typu boolean pamätalo, ktoré pakety sme našli v súbore. Toto riešenie je celkom pekné a rýchle, s časovou zložitosťou $O(n)$, ale vyžaduje až $O(n)$ pamäte, čo sa nám veľmi nepozdáva.

Druhé, pomerne jednoduché, riešenie by to naopak potiahlo do opačného extrému. Postupne by pre každé číslo od 1 po n prešlo celý súbor a pozrelo sa, či sa v ňom dané číslo nachádza. Takéto riešenie by síce spotrebovalo iba konštatné množstvo pamäte, bolo by však pomalé, pretože by muselo prejsť súborom n krát a celková časová zložitosť by bola $O(n^2)$.

Ďalej by sme mohli rôznymi spôsobmi vyvažovať časové a pamäťové nároky, napríklad s použitím binárneho vyhľadávania. („Máme všetky čísla z prvej polovice?“) Dá sa to však spraviť jednoduchšie a dokonca s lepšou zložitosťou. Stačí nám jedno jednoduché pozorovanie.

Je jednoduché si spočítať, aký súčet čísel by mali pakety, keby tam boli všetky: je to jednoducho súčet všetkých čísel od 1 po n, teda $n(n+1)/2$. A keď nám chýba len jeden paket, vieme jeho číslo zistiť odčítaním očakávaného súčtu čísel paketov od reálneho súčtu čísel paketov. Dostaneme tak riešenie, ktorému stačí raz prečítať súbor a pritom sčítať všetky čísla, na ktoré narazíme. Na to nám stačí konštantne veľká pamäť a jeden prechod súboru, takže dostaneme časovú zložitosť $O(n)$.

Listing programu (C++)

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

int main () {
    FILE * fp = fopen("pakety.txt", "r");
    long long sucet = 0, kolke_cislo = 1, cislo;
    while (fscanf(fp, "%lld", &cislo) > 0) {
        sucet += kolke_cislo - cislo;
        kolke_cislo ++;
    }
    printf("%lld\n", sucet);
}
```

Podúloha B

Vyzbrojení znalosťou z podúlohy A sa môžeme pustiť do riešenia podúlohy B. Postupovať úplne rovnakým spôsobom však v tomto prípade nemôžeme, lebo by sme zistili iba súčet dvoch chýbajúcich čísel. Problémom ale je, že k väčšine takýchto súčtov existuje mnoho rôznych dvojíc čísel, ktoré dávajú takýto súčet a my si nevieme vybrať tú správnu. Preto na to musíme ísť šikovnejšie a ponúkajú sa nám dokonca dva rôzne prístupy.

Prvým možným riešením je pozorovanie, že v každom súčte dvoch rôznych čísel bude jedno z týchto čísel menšie ako polovica súčtu a druhé väčšie ako polovica súčtu. Preto ak nájdeme súčet chýbajúcich čísel s, jedno z chýbajúcich čísel bude menšie ako $\frac{s}{2}$. Následne nám stačí znova aplikovať riešenie predošlej podúlohy s tým, že budeme ignorovať akékoľvek čísla väčšie ako $\frac{s}{2}$. Toto riešenie použije dve čítania súboru, lebo zo znalosti súčtu chýbajúcich čísel a menšieho z týchto čísel ľahko dorátame aj druhé chýbajúce číslo.

Druhým možným riešením je počítať si nielen súčet čísel, ale aj súčet štvorcov čísel (ktorému vieme tiež ľahko zistiť, akú hodnotu by mal, keby žiadne číslo nechýbalo)[1]. Keď poznáme pre dve čísla ich súčet aj súčet ich štvorcov, vieme vždy dané dve čísla jednoznačne dopočítať.[2]

Keďže v našom prípade vieme, že hľadané x a y existujú a sú z rozsahu od 1 po n, nemusíme robiť výpočty popísané v poznámke pod čiarou. Namiesto toho jednoducho v cykle vyskúšame všetky možné x, ku každej zo súčtu dopočítame y a overíme, či sedí aj súčet druhých mocnín.

Obe tieto riešenia spravia jedno alebo dve čítania súboru, pracujú v čase $O(n)$ a pamäti $O(1)$.

Listing programu (C++)

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

int main () {
    FILE * fp = fopen("pakety.txt", "r");
    // ocakavame aspon paket 0 a 1, za kazde dalsie precitane cislo ocakavame o 1 viac paketov
    long long sucet = 1, kolke_cislo = 2, cislo, sucet_stvorcov = 1;
    while (fscanf(fp, "%lld", &cislo) > 0) {
        sucet += kolke_cislo - cislo;
        sucet_stvorcov += kolke_cislo * kolke_cislo - cislo * cislo;
        kolke_cislo ++;
    }
    for (long long i = 0; i <= sucet / 2; i++) {
        if (i * i + (sucet - i) * (sucet - i) == sucet_stvorcov) {
            printf("%lld %lld\n", i, sucet - i); // riesenie existuje len jedno
        }
    }
}
```

[1] Na internete viete opäť nájsť exaktný vzorec, do ktorého stačí dosadiť n. Lenivejšie riešenie spočíta tento súčet v lineárnom čase pomocou cyklu – to nám našťastie neovplyvní výslednú zložitosť.

[2] Prečo? Sústava rovníc $x+y=a$ a $x^2+y^2=b$ má pre kladné a, b najviac jedno riešenie. Totiž vieme vypočítať že $(x+y)^2=a^2$, a teda $xy=(a^2-b)/2$. No a keď poznáme $x+y=a$ aj $xy=c$, tak x a y sú nutne koreňmi kvadratickej rovnice $t^2-at+c=0$.

A-I-4 Magické siete

Postupne si vyriešime všetky podúlohy. Riešenia jednotlivých podúloh na seba nenadväzujú, dajú sa čítať v ľubovoľnom poradí.

Podúloha A

$\mathrm{OR}_2(x, y)$ platí ak aspoň jedno z x a y nie je 0. Ekvivalentná podmienka by bola $\mathrm{NAE}(x, y, 0)$, keby sme mali nulu. My síce nulu nemáme, ale vieme si vyrobiť pomocnú premennú, ktorej hodnota bude musieť byť 0. Stačí zobrať tri nové premenné a, b, c a pridať obmedzenia $\mathrm{ONE}(a)$, $\mathrm{ONE}(b)$ a $\mathrm{NAE}(a, b, c)$. Potom zjavne c musí byť nula.

Riešením je teda sieť obsahujúca vstupné premenné x, y, pomocné premenné a, b, c, a obmedzenia $\mathrm{NAE}(x, y, c)$, $\mathrm{ONE}(a)$, $\mathrm{ONE}(b)$, $\mathrm{NAE}(a, b, c)$.

Za domácu úlohu si rozmyslite, ako by sa pomocou obmedzení typu NAE a typu ONE simulovalo obmedzenie OR_3. (Uvedomte si tiež, že podúloha C ukazuje, že budete musieť použiť aspoň jedno ONE.)

Podúloha B

V duchu hesla „nepriateľ môjho nepriateľa je mojim priateľom“: $\mathrm{EQ}(x, y)$ platí (teda premenné x a y sa rovnajú) práve vtedy, ak sa obe líšia od tej istej pomocnej premennej a. No a to vieme ľahko zapísať pomocou obmedzení typu NAE.

Riešením je teda sieť obsahujúca vstupné premenné x a y, pomocnú premennú a a obmedzenia $\mathrm{NAE}(x, a, a)$ a $\mathrm{NAE}(y, a, a)$.

Pre zaujímavosť uvádzame, že úloha má aj riešenie, pri ktorom v každom obmedzení použijeme tri rôzne premenné. Všimnime si obmedzenia $\mathrm{NAE}(x, a, b)$, $\mathrm{NAE}(x, a, c)$ a $\mathrm{NAE}(x, b, c)$. Tieto nám hovoria, že v každej z dvojíc (a, b), (a, c) a (b, c) musí aspoň jedna premenná mať hodnotu opačnú od x. Inými slovami, spomedzi a, b a c má najviac jedno tú istú hodnotu ako x, a aspoň dve majú hodnotu opačnú.

Keď teraz pridáme rovnaké tri obmedzenia ale s premennou y namiesto x, dostaneme magickú sieť, ktorá zjavne má platné ohodnotenie premenných vtedy a len vtedy keď $x = y$.

Podúloha C

Treba si všimnúť, že $\mathrm{NAE}(x, y, z)$ je splnené práve vtedy, keď je splnené $\mathrm{NAE}(1 - x, 1 - y, 1 - z)$. Ak by teda sieť zo samých NAE prijala vstup $(1, 1, 1)$, tak môžeme úplne všetky premenné (vstupné aj pomocné) znegovať a vidíme, že tá istá sieť musí prijať aj vstup $(0, 0, 0)$. To je ale zle, pretože OR_3 pre ohodnotenie $(1, 1, 1)$ splnené je, ale pre ohodnotenie $(0, 0, 0)$ nie.

TRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Vladimír Boža, Michal Forišek, Vladimír Macko, Jaroslav Petrucha
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2014