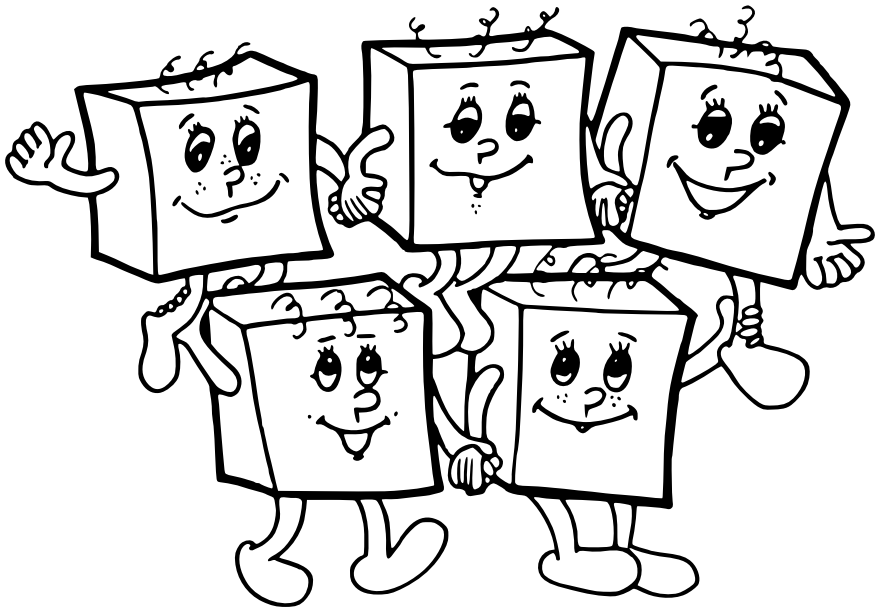
# OLYMPIÁDA V INFORMATIKE
## NA STREDNÝCH ŠKOLÁCH



**dvadsiaty druhý** (a zároveň prvý) **ročník**
školský rok 2006/07

**riešenia krajského kola**
**kategória B**

- **Olympiáda v informatike** je od tohto školského roku samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://www.ksp.sk/oi/`.
- Novinkou je zavedenie **dvoch kategórií**. Nová kategória B je určená pre mladších riešiteľov.

# Riešenia kategórie B

## B-II-1 Kerfúr ten je lacnejší

Pokúsime sa zostrojiť nejaké poradie supermarketov, ktoré by zodpovedalo všetkým reklamám. Ak sa nám to podarí, bude výstup `ANO`, naopak, ak zistíme, že sa to nedá, bude výstup `NIE`.

Majme teda niekoľko supermarketov. Rozdeľme si ich na dve kôpky. Na prvú dáme tie, o ktorých nik iný netvrdí, že je od nich lacnejší. Na druhej teda samozrejme zostanú tie, od ktorých poznáme aspoň jeden lacnejší.

Pokiaľ je prvá kôpka prázdna, hľadané usporiadanie zjavne neexistuje – žiaden zo supermarketov nemôže byť najlacnejší.

(Rozmyslite si, že ak táto situácia nastala, znamená to, že na vstupe sme dostali *cyklus* – postupnosť supermarketov $s_1, \ldots, s_k$ takú, že pre každé $i$ je $s_i$ lacnejší ako $s_{i+1}$, a navyše aj $s_k$ má byť lacnejší ako $s_1$. Keby bolo treba, takýto cyklus nájdeme ľahko: začneme ľubovoľným supermarketom a vždy prejdeme na nejaký, ktorý má od práve spracúvaného byť lacnejší, až kým sa nám nejaký supermarket v takto získanej postupnosti nezopakuje.)

Pokiaľ je na prvej kôpke aspoň jeden supermarket, môžeme hociktorý z týchto supermarketov vyhlásiť za najlacnejší. Rozmyslite si, že tým naozaj nič nemôžeme pokaziť.

Vyberieme si teda ľubovoľný supermarket z prvej kôpky, prehlásime ho za najlacnejší a úplne naň zabudneme. A čo sme takto dostali? Tú istú úlohu, len máme o jeden supermarket menej. Rovnakým postupom teda pokračujeme ďalej, a sú len dve možnosti: buď časom narazíme na cyklus (a zistíme, že riešenie neexistuje), alebo sa nám časom všetky supermarkety minú a máme hotové celé poradie.

Ako to šikovne naprogramovať? Pre každý supermarket si budeme pamätať, koľko supermarketov má byť od neho lacnejších a tiež zoznam všetkých, ktoré majú byť od neho drahšie. Okrem toho si budeme v jednom poli pamätať čísla tých supermarketov, ktoré už od seba nemajú mať nikoho lacnejšieho.

Výber jedného supermarketu bude prebiehať nasledovne: Pozrieme sa do poľa s kandidátmi a vyberieme posledného z nich. Teraz ho potrebujeme odstrániť. Prejdeme všetky supermarkety, ktoré majú od neho byť väčšie. Každému z nich zmenšíme počítadlo menších o jedna, a ak kleslo na nulu, pridáme ho na koniec poľa kandidátov.

Spracovanie jedného supermarketu teda trvá lineárne dlho v závislosti od počtu reklám, ktoré ho chvália. Spracovanie všetkých supermarketov dokopy bude teda trvať lineárne dlho od počtu všetkých reklám. Lepšie to ani nejde, lebo rádovo rovnaký čas nám zoberie už len načítanie vstupu.

### Listing programu:

```
program reklamy;
var mensich, vacsich, kandidati : array[1..1000] of longint;
    vacsie : array[1..1000,1..1000] of longint;
    N, M, i, j, x, y, kandidatov, teraz : longint;

begin
   readln(N,M);
   for i:=1 to N do begin mensich[i]:=0; vacsich[i]:=0; end;
   for i:=1 to M do begin
      readln(x,y);
      inc(vacsich[x]);  inc(mensich[y]);
      vacsie[ x ][ vacsich[x] ] := y;
   end;
   { inicializujeme pole kandidatov }
   kandidatov := 0;
   for i:=1 to N do if mensich[i]=0 then begin
      inc(kandidatov);
      kandidati[ kandidatov ]:=i;
   end;
   { N-krat skusime vybrat najlacnejsi supermarket }
   for i:=1 to N do begin
      { ak nemame ziadneho kandidata, zle je }
      if kandidatov=0 then begin writeln('NIE'); halt; end;
      { ak mame, odstranime ho }
      teraz := kandidati[ kandidatov ];
      dec(kandidatov);
      for j:=1 to vacsich[teraz] do begin
         x := vacsie[teraz][j];
         dec(mensich[x]);
         if mensich[x]=0 then begin { mame noveho kandidata }
            inc(kandidatov);
            kandidati[kandidatov] := x;
         end;
      end;
   end;
   writeln('ANO');
end.
```

Poznámka. Úloha sa dá preformulovať do jazyka teórie grafov. Jednotlivé supermarkety budú predstavovať vrcholy grafu, informáciu „je lacnejší“ si zaznačíme ako orientovanú hranu. Hľadané usporiadanie supermarketov sa volá *topologické usporiadanie* vrcholov grafu. Ukázali sme, že takéto usporiadanie existuje práve vtedy, ak je graf zodpovedajúci zadaniu acyklický.

## B-II-2 Televízna súťaž

Keďže čísla sú z príliš veľkého rozsahu na to, aby sme si mohli značiť v poli, ktoré sme koľkokrát videli, budeme si musieť pomôcť ináč.

Jedným možným riešením je čísla, ktoré dostaneme zadané na vstupe, uložiť do poľa a toto pole utriediť. Ak to spravíme použitím šikovného triediaceho algoritmu (napríklad QuickSort), budeme na utriedenie čísel potrebovať rádovo $N \log N$ operácií. Keď už máme čísla utriedené, stačí raz prejsť poľom a spočítať si, ktorá hodnota sa v ňom koľkokrát vyskytuje. (Vľmnite si, že po utriedení budú všetky výskyty konkrétneho čísla tvoriť v poli súvislý úsek.)

### Listing programu:

```
program sutaz1;
var a : array[1..10047] of longint;
    n,i,kandidat,pkand,pteraz : longint;

procedure QuickSort(zaciatok, koniec: longint);
   { utriedi cisla, ktore su v poli A na poziciach zaciatok .. koniec }
var pivot, malych, strednych, i, pom : longint;
begin
   if (koniec <= zaciatok) then exit;        { ak je usek kratky, skonci }
   pivot := A[zaciatok];               { vyber jeden z prvkov ako pivota }
   malych := 0;                     { presunieme "male" prvky na zaciatok }
   for i:=zaciatok to koniec do
      if (A[i] < pivot) then begin
         pom:=A[i];  A[i]:=A[zaciatok+malych];  A[zaciatok+malych]:=pom;
         Inc(malych);
      end;
        { a mame "male" prvky na poziciach zaciatok .. zaciatok+malych-1 }
   strednych := malych;            { za ne presunieme prvky rovne pivotu }
   for i:=zaciatok+malych to koniec do
      if (A[i] = pivot) then begin
         pom:=A[i];  A[i]:=A[zaciatok+strednych];  A[zaciatok+strednych]:=pom;
         Inc(strednych);
```

```
      end;                          { a mame prvky rovne pivotu na poziciach
                                      zaciatok+malych .. zaciatok+strednych-1 }
                    { teraz samostatne utriedime usek "malych" prvkov ... }
   QuickSort(zaciatok, zaciatok+malych-1);
   QuickSort(zaciatok+strednych, koniec);  { ... a usek "velkych" prvkov }
end;

begin
   read(n);
   for i:=1 to n do read(a[i]);
   QuickSort(1,n);
    { v premennej "kandidat" je doteraz najcastejsi prvok, v "pkand" je
      pocet jeho vyskytov, v "pteraz" je pocet vyskytov aktualneho prvku }
   kandidat:=-1; pkand:=0; pteraz:=0;
   for i:=1 to n do begin
      if (i=1) or (a[i-1]=a[i]) then inc(pteraz) else pteraz:=1;
      if (pteraz > pkand) then begin kandidat:=a[i]; pkand:=pteraz; end;
   end;
   writeln(kandidat);
end.
```

Existujú však ešte lepšie riešenia, založené na metóde *hašovania*. Predstavme si, že nám z neba spadla funkcia $f$, ktorá každému číslu zo vstupu priradí číslo povedzme z intervalu od 0 do $N-1$. Navyše chceme, aby výstupné hodnoty $f$ boli približne rovnako pravdepodobné. (Teda ak vypočítame hodnotu $f$ pre veľa náhodných vstupov, dostaneme každý výsledok približne rovnako veľa krát.)

Ako by nám takáto funkcia pomohla pri riešení úlohy? Predstavme si, že máme $N$ vedier označených číslami od 0 do $N-1$. Pre každé číslo $x$ zo vstupu spočítame hodnotu $f(x)$ a do toho vedra $x$ hodíme. Potom stačí každé vedro spracovať samostatne.

Získali sme tým niečo? Na vstupe bolo najviac $N$ rôznych hodnôt, preto môžeme očakávať, že v priemere bude v každom vedre len jedna hodnota. Vedrá, v ktorých skončila len jedna hodnota, ľahko spracujeme v konštantnom čase. Pre každé z ostatných vedier môžeme napríklad použiť vyššie popísaný algoritmus – čísla utriedime a raz prejdeme.

Samozrejme, najhorší možný prípad je, že na vstupe je $N$ rôznych čísel, ale všetky skončia v tom istom vedre. Takýto prípad je ale veľmi nepravdepodobný. V priemernom prípade má náš algoritmus časovú zložitosť lineárnu od počtu čísel na vstupe, v najhoršom prípade bude jeho časová zložitosť naďalej úmerná časovej zložitosti triedenia, teda $O(N \log N)$.

Posledná otázka: odkiaľ vziať funkciu $f$? Môžeme pre jednoduchosť uvažovať funkciu $f(x) = x \bmod N$. (V praxi sa používajú zložitejšie hašovacie funkcie.)

### Listing programu:

```
program sutaz2;
type pzaznam = ^tzaznam;
     tzaznam = record next : pzaznam; hodnota : longint; end;
var vedra : array[0..10047] of pzaznam;
    prve : array[0..10047] of longint;
    pocet : array[0..10047] of longint;
    N,i,x : longint;
    kandidat,pkand : longint;

{ pridaj(kam,co) prida hodnotu "co" na zaciatok zoznamu "kam" }
procedure pridaj(kam : pzaznam; co : longint);
var tmp : pzaznam;
begin getmem(tmp,sizeof(tzaznam)); tmp^.hodnota:=co; tmp^.next:=kam; end;

{ hashovacia funkcia f }
function f(x : longint) : longint; begin f:=x mod N; end;

begin
   read(N);
   for i:=0 to n-1 do begin pocet[i]:=0; vedra[i]:=nil; end;
   for i:=1 to n do begin
      read(x);
      if pocet[f(x)]=0 then begin
         prve[f(x)]:=x; pocet[f(x)]:=1;
      end else begin
         if prve[f(x)]=x then inc(pocet[f(x)]) else pridaj( vedra[ f(x) ], x );
      end;
   end;
   kandidat:=-1; pkand:=0;
   for i:=0 to n-1 do begin { spracujeme vedro i }
      if pocet[i]>pkand then begin pkand:=pocet[i]; kandidat:=prve[i]; end;
      if vedra[i]<>nil then begin
         { ... prepiseme zvysny obsah vedra do pola ... }
         { ... pouzijeme na pole predchadzajuci program ... }
      end;
   end;
   writeln(kandidat);
end.
```

## B-II-3 Kartári

Lackov balíček kariet zodpovedá dátovej štruktúre známejšej pod menom fronta. Môžeme si ho predstaviť ako takú dlhú rúru, do ktorej jedným koncom karty vkladáme a druhým koncom z nej *v tom istom poradí* vypadávajú.

Pomalé riešenie je pamätať si karty v poli a pri každom výbere ich o jedno posunúť. Takto totiž bude náš program tým pomalší, čím viac kariet bude mať Lacko v kôpke.

Lepšie riešenie je použiť na reprezentáciu Lackovej kôpky kariet spájaný zoznam. Takto vieme zobrať kartu zo začiatku aj pridať kartu na koniec priamo, bez toho, aby nás trápilo, koľko kariet je medzi nimi.

### Listing programu:

```
program fronta1;
type pzaznam = ^tzaznam;
     tzaznam = record
        prev, next : pzaznam;
        hodnota : char;
     end;
var zaciatok, koniec : pzaznam;
    prikaz : longint;
    karta : string;

procedure pridaj(co : char);
var tmp : pzaznam;
begin
   { vytvorime a inicializujeme novy zaznam }
   getmem(tmp,sizeof(tzaznam));
   tmp^.hodnota := co; tmp^.next:=nil;

   { ak je fronta prazdna, vytvorime novu, inak ho pridame na koniec }
   if (zaciatok=nil) then begin
      tmp^.prev:=nil; zaciatok:=tmp;
   end else begin
      tmp^.prev:=koniec; koniec^.next:=tmp;
   end;
   koniec:=tmp;
end;

function vyber : char;
var tmp : pzaznam;
begin
   { vratime hodnotu z prveho zaznamu a vymazeme ho z pamate }
   vyber := zaciatok^.hodnota;
   tmp := zaciatok;
   zaciatok := zaciatok^.next;
   freemem(tmp);
end;

begin
   zaciatok := nil; koniec := nil;
   while true do begin
      read(prikaz);
      case prikaz of
         1 : begin readln(karta); pridaj(karta[2]); end;
         2 : writeln(vyber);
         0 : halt;
      end;
   end;
end.
```

Rovnako efektívne riešenie, ale jednoduchšie na naprogramovanie dostaneme nasledovne: budeme si Lackove karty pamätať ako súvislý úsek prvkov poľa. Keď príde nová karta, pridáme ju na koniec úseku (ten sa tým natiahne „doprava“), keď máme zahrať kartu, zoberieme ju zo začiatku (tým sa nám úsek skráti).

Ak ale zvolíme pole konečnej dĺžky, ako zabezpečiť, aby sme z neho časom nevybehli? Jednoducho: Vždy, keď takto prídeme úsekom na koniec poľa, začneme nové prvky pridávať znova od začiatku. A ešte jeden detail: Aby sa nám to celé nepomiešalo, zvolíme si pole väčšej dĺžky ako je celkový počet kariet.

### Listing programu:

```
program fronta2;
var fronta : array[0..146] of char;
    zac, kon : longint; { zac=zaciatok useku, kon=prve volne miesto za nim }
    prikaz : longint;
    karta : string;

procedure pridaj(co : char);
begin
   fronta[kon] := co;
   kon := (kon+1) mod 147;
end;
function vyber : char;
begin
   vyber := fronta[zac];
   zac := (zac+1) mod 147;
end;

begin
   zac:=0; kon:=0;
   while true do begin
      read(prikaz);
      case prikaz of
         1 : begin readln(karta); pridaj(karta[2]); end;
         2 : writeln(vyber);
         0 : halt;
      end;
   end;
end.
```

Obe uvedené riešenia spracujú každú inštrukciu s konštantnou časovou zložitosťou.

## B-II-4 Assembler II

Prvú úlohu vyriešime ľahko. Presunieme obsah $R_j$ do $R_7$. Zmažeme obsah $R_i$. Na záver presunieme obsah $R_7$ naraz aj do $R_i$, aj do $R_j$. Musíme si dať pozor na situácie, kedy je v $R_i$ alebo $R_j$ nula.

Definujeme teraz pomocnú inštrukciu „**add** $R_i$, $R_j$“, ktorá k hodnote v $R_i$ pridá hodnotu z $R_j$ (a tú nechá nezmenenú). Toto spravíme rovnako ako pri inštrukcii **mov**, len vynecháme dva riadky, ktoré nulujú obsah $R_i$.

Pomocou inštrukcie **add** teraz definujeme inštrukciu **mul** zo zadania. Presunieme hodnotu $R_i$ do $R_6$. Skopírujeme hodnotu $R_j$ do $R_5$. Teraz hodnotu z $R_6$ (hodnota z $R_5$)-krát pripočítame k hodnote (pôvodne teda nule) v $R_i$. Nakoniec už len vynulujeme $R_6$.

```
///// mov R_i R_j /////
a: test R_j, R_j
   jz b
   dec R_j
   inc R_7
   jmp a
b: dec R_i
   jnz b
c: test R_7, R_7
   jz k
   dec R_7
   inc R_i
   inc R_j
   jmp c
k:
```

```
///// mul R_i R_j /////
a: test R_j, R_j
   jz b
   dec R_j
   inc R_6
   jmp a
b: mov R_j, R_5
c: test R_5, R_5
   jz d
   dec R_5
   add R_i, R_6
   jmp c
d: dec R_6
   jnz d
```

Všimnime si ešte, ako dlho trvá vykonanie nami definovaných inštrukcií. Na vykonanie **mov** potrebujeme inštrukcií lineárne veľa od súčtu hodnôt $R_i$ a $R_j$. Pri **add** nemusíme nulovať $R_i$, takže počet inštrukcií je priamo úmerný hodnote v $R_j$. Pri inštrukcii **mul** bude počet krokov lineárne závisieť od výsledného súčinu. *(V špeciálnom prípade, kedy je jeden z činiteľov 0, nebude počet inštrukcii*

*0, ale bude úmerný obsahu druhého z násobených registrov. Toto nás ale pri našom využívaní tejto inštrukcie trápiť nebude.)*

S počítaním odmocniny to bude trochu zložitejšie. Optimálne riešenie je pomerne jednoduché, ale ukážeme si predtým niekoľko iných, pomalších riešení.

Začnime riešením, na ktoré netreba v podstate nič vymýšľať. Postupne budeme zväčšovať hodnotu v registri $R_0$. Zakaždým spočítame (pomocou inštrukcie **mul**) jej druhú mocninu a tú porovnáme s hodnotou v $R_1$. Akonáhle bude vypočítaná druhá mocnina väčšia ako obsah $R_1$, zmenšíme obsah $R_0$ o jedna a skončíme.

Jediná vec, ktorú ešte nevieme robiť, je „tú porovnáme s hodnotou v $R_1$“.

Definujme inštrukciu na porovnanie registrov „**jg** $R_i$, $R_j$, x“ (jump if greater), ktorá bude fungovať nasledovne: Porovnaj obsahy $R_i$ a $R_j$. Oba registre vynuluj. Ak bol obsah $R_i$ väčší, pokračuj skokom na návestie x, inak pokračuj ďalšou inštrukciou.

```
///// jg R_i R_j x /////
```

```
a:      test R_i, R_i
        jz zle
        dec R_i
        test R_j, R_j
        jz dobre
        dec R_j
        jmp a
dobre:  dec R_i
        jz x
        jmp dobre
zle:    dec R_j
        jnz zle
```

Aké efektívne je toto riešenie? Nech je správna odpoveď $k$. Potom náš algoritmus postupne vypočíta hodnoty $1^2$, $2^2$, …, $(k+1)^2$, a každú z nich porovná s $k$. Na výpočet hodnôt potrebujeme čas úmerný súčtu vypočítaných hodnôt, teda $O(k^3)$ krokov. Porovnanie dvoch hodnôt trvá čas úmerný väčšej oboch

hodnôt. V našom prípade je to skoro vždy obsah registra $R_1$, a ten je približne rovný $k^2$. Porovnaní robíme $k$, preto pri všetky porovnaniach dokopy spravíme $O(k^3)$ krokov.

Ak označíme hodnotu v $R_1$ ako $n$, náš algoritmus potrebuje $O(n\sqrt{n})$ krokov.

Funkcia na porovnanie hodnôt sa dá v našom prípade napísať aj šikovnejšie, ak si trochu zmeníme, ako má fungovať. Spolu so zmenšovaním oboch hodnôt budeme zväčšovať hodnotu v novom pomocnom registri. Keď teraz vynulujeme niektorý z registrov, zapamätáme si (skokom do vhodnej časti programu), ktorý register to bol, a spravíme opačný proces. Pomocný register budeme nulovať, pôvodné dva zároveň s tým zväčšovať. Skončíme v situácii, v ktorej sme začínali, iba navyše vieme, ktorý z registrov má väčší obsah. Takto upravené porovnávanie je lineárne od **menšieho** z oboch porovnávaných čísel. (Asymptotickú časovú zložitosť predchádzajúceho riešenia to nezlepší, ale zlepšenie to je.)

Prefíkanejšie riešenie je neskúšať postupne všetky hodnoty $k$, ale len niektoré. Použijeme niečo podobné binárnemu vyhľadávaniu: postupne skúsime ako $k$ hodnoty 1, 2, 4, 8, 16, ..., až kým $k^2$ neprekročí obsah $R_1$. V tomto okamihu máme v $R_0$ hodnotu $2^x$ a vieme, že správne $k$ je v rozsahu $2^{x-1} \leq k < 2^x$. A teraz už môžeme použiť sľubované binárne vyhľadávanie a na ďalších $x - 1$ „otázok“ nájsť presnú hodnotu $k$.

(Príklad: Ak sme zistili, že $k = 8$ ešte nie je veľa a $k = 16$ už veľa je, budeme postupne skúšať $k = 12$, a napr. ďalej $k = 14$ a $k = 13$.)

Oproti predchádzajúcemu algoritmu sme na tom lepšie: zatiaľ čo ten potreboval vyskúšať $k$ hodnôt, tomuto stačí $2 \log_2 k$. Náš nový algoritmus teda spraví $O(k^2 \log k)$ krokov. Ak označíme hodnotu v $R_1$ ako $n$, tento algoritmus potrebuje $O(n \log n)$ krokov.

Stále to ale ide aj lepšie, a prekvapujúco jednoducho. Predstavme si, že v premennej $y$ máme štvorec premennej $x$. Ako vypočítať štvorec premennej $x+1$? Platí $(x + 1)^2 = x^2 + 2x + 1$. Takže stačí k $y$ pripočítať dvojnásobok $x$ a ešte 1.

V našom programe budeme robiť presne to isté, len namiesto pripočítavania k $y$ budeme rovno odpočítavať od hodnoty v registri $R_1$. Bude teda platiť, že

keď v $R_0$ bude hodnota $x$, v $R_1$ bude $n - x^2$. A jednoducho povedané, keď sa nám $R_1$ minie, skončíme.

Časová zložitosť tohoto riešenia je zjavne lineárna od obsahu $R_1$, teda $O(n)$. Program vyzerá nasledovne:

```
///// odmocnina /////
a:      sub R_1, R_0
        sub R_1, R_0
        jz koniec
        dec R_1
        inc R_0
        jmp a     // teraz plati: v R_0 je x, v R_1 je n-x^2
koniec:
```

(V programe používame inštrukciu „**sub** $R_i$, $R_j$“. Tá vyzerá presne rovnako ako „**add** $R_i$, $R_j$“, len namiesto inštrukcie „**inc** $R_i$“ je v nej „**dec** $R_i$“. Výsledok je teda taký, že od hodnoty v $R_i$ odčíta hodnotu z $R_j$.)

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

**DVADSIATY DRUHÝ ROČNÍK OLYMPIÁDY V INFORMATIKE**

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX