MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

52. ročník, školský rok 2002/2003

Zadania úloh 1. kola kategórie P

Matematická olympiáda je súťaž pre študentov stredných škôl našej republiky. **Kategória P** je zameraná na programovanie a je určená študentom všetkých ročníkov.

Organizácia súťaže v kategórii P

Kategória P matematickej olympiády má tri postupové kolá — domáce, krajské a celoštátne.

V **I. kole** účastníci riešia štyri úlohy uvedené v tomto letáku. Riešenia odovzdajú svojmu učiteľovi informatiky do **15. novembra 2002**. Učitelia informatiky odošlú riešenia v tomto termíne zo školy na príslušnú adresu podľa krajov takto:

Košický, Prešovský:
RNDr. G. Andrejková, CSc., KMI PF UPJŠ, Jesenná 5, 041 54 Košice
Žilinský:
RNDr. P. Varša, KI FRI Žilinská univerzita, Moyzesova 20, 010 26 Žilina
Banskobystrický:
Mgr. L. Huraj, KI FPV UMB, Tajovského 40, 974 01 Banská Bystrica
Trnavský, Trenčiansky, Nitriansky:
Doc. Ing. V. Štoffová, CSc., KI FPV UKF, tr. A. Hlinku 1, 949 74 Nitra
Bratislavský:
RNDr. A. Blaho KVI FMFI UK, Mlynská dolina, 842 48 Bratislava

Najúspešnejší riešitelia domáceho kola sú pozvaní do **II. kola** (krajského), kde riešia štyri teoretické úlohy. Do **III. kola** (celoštátneho) sú pozývaní najúspešnejší riešitelia všetkých krajských kôl, pričom riešia tri teoretické úlohy a dve praktické úlohy pri počítači. Z najlepších riešiteľov tohto kola SK MO vyberie družstvá pre Medzinárodnú informatickú olympiádu a Stredoeurópsku informatickú olympiádu.

Predbežné termíny 52. ročníka MO kat. P

I.	Domáce kolo	15. novembra 2002
II.	Krajské kolo	7. januára 2003
III.	Celoštátne kolo	apríl 2003

Usporiadateľ súťaže

Matematickú olympiádu vyhlasuje *Ministerstvo školstva SR* v spolupráci s *Jednotou slovenských matematikov a fyzikov* a *Slovenskou komisiou Matematickej olympiády*. Súťaž organizuje *Slovenská komisia MO* a v jednotlivých krajoch ju riadia *krajské komisie MO* pri pobočkách JSMF. Na jednotlivých školách ju zaisťujú učitelia matematiky a informatiky. Celoštátne kolo MO, tlač materiálov MO a ich distribúciu po organizačnej stránke zabezpečuje IUVENTA – zariadenie pre voľný čas detí, mládeže i dospelých MŠ SR v tesnej súčinnosti so slovenskou komisiou MO.

Formálna úprava riešenia

Riešenia súťažných úloh domáceho kola kategórie P pozostávajú z dvoch častí:

Popis riešenia. Riešenia musia obsahovať podrobný popis použitého algoritmu, zdôvodnenie jeho správnosti a diskusiu o efektivite zvoleného riešenia (t.j. posúdenie časových a pamäťových nárokov programu). Algoritmus by mal byť jasný už z popisu riešenia bez toho, aby bolo potrebné nahliadnuť do programu.

Program. V úlohách **P–I–1** a **P–I–3** je potrebné k riešeniu pripojiť odladený program napísaný v jazyku Pascal, C alebo C++. Program sa odovzdáva v písomnej forme (jeho výpis je teda súčasťou riešenia) i na diskete, aby bolo možné otestovať jeho funkčnosť.

Súbory na diskete pomenujte `p1`x`.pas/.c/.cpp`, kde x je číslo súťažnej úlohy. Disketu označte menom riešiteľa. Z jednej školy možno poslať všetky riešenia na jednej diskete. V tomto prípade pre každého riešiteľa vytvorte podadresár označený jeho priezviskom a disketu označte adresou školy.

V úlohe **P–I–2** stačí uviesť dostatočne podrobný popis algoritmu, program netreba. V úlohe **P–I–4** je potrebné zapísať navrhnutý algoritmus ako reverzibilnú procedúru.

Písomnú časť riešenia vypracujte čitateľne na listy formátu A4. **Každú úlohu začnite na novom liste** a v záhlaví uveďte vaše meno, triedu, adresu školy a označenie príkladu podľa tohto letáku. Zadania úloh nemusíte opisovať. Ak sa vám riešenie nezmestí na jeden list, uveďte na ďalších listoch vľavo hore svoje meno a označenie úlohy a očíslujte strany.

P-I-1

Pán Nyi bol dvorným pestovateľom čaju cisára Tiang-tonga. Bol skutočne vyhláseným pestovateľom a jeho čajové lístky putovali nielen do blízkych šálok cisára Tianga, ale aj do ďalekých zemí za oceánom. Tajomstvo Nyiovho skvelého čaje spočívalo predovšetkým v dôkladnosti, s akou sa staral o svoje čajové kríky. Nyi bol tak dôkladný, že si o každom svojom kríku viedol záznamy. Písal si dokonca aj to, koľko vetvičiek vychádza z ktorého miesta kríka. Po smrti pána Nyia boli záznamy rozkradnuté a jeho nástupca pan Myi tak mal o veľa ťažšiu prácu. Rozhodol sa preto, že záznamy získa späť. Problém ale je, že veľa rôznych podvodníkov mu ponúka falošné záznamy. Tie však našťastie väčšinou obsahujú nezmyselné počty vetvení, a tak sa dajú ľahko odhaliť. Pána Myia ustavičné overovanie pravosti záznamu už unavuje, a preto vás požiadal, aby ste mu napísali program, ktorý mu s overovaním pomôže.

Váš program dostane na vstup počet významných miest N na údajnom čajovníku. Významným miestom na čajovníku je buď miesto, kde sa čajovník vetví, alebo miesto, kde končí niektorá vetva čajovníku. Pretože žiadne dve vetvy čajovníku nemôžu zrásť, nemôžu vznikať "cykly" z vetví. Ďalej je na vstupe programu zadaných N kladných celých čísel $c_1, c_2, \ldots, c_N$, kde c_i určuje počet častí kmeňa, ktoré vychádzajú z i-teho významného miesta. Na výstup program vypíše správu, či môže existovať čajovník, ktorý bude mať takéto počty vetvení.

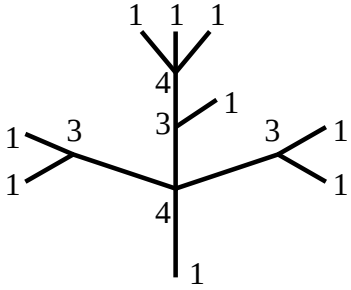
Formát vstupu. Vstupný textový súbor `caj.in` obsahuje dva riadky. Na prvom riadku je uvedené jediné celé číslo N, $1 \leq N \leq 1000$. Druhý riadok obsahuje celé čísla $c_1, c_2, \ldots, c_N$ oddelené medzerami, $1 \leq c_i \leq N - 1$.

Formát výstupu. Výstupný textový súbor `caj.out` obsahuje jediný riadok tvorený buď slovom `EXISTUJE` alebo slovom `NEEXISTUJE`.

Príklad:

Súbor `caj.in`
```
14
1 4 3 1 1 3 1 1 3 1 4 1 1
1
```

Súbor `caj.out`
```
EXISTUJE
```
(pozri obrázok vpravo)



Súbor `caj.in`
```
6
3 3 3 1 1 1
```

Súbor `caj.out`
```
NEEXISTUJE
```

P-I-2

Knihovníčka Milka potrebuje objednať ďalšiu skriňu s poličkami do svojej knižnice, nevie však sama spočítať jej optimálne rozmery. Milka by chcela

do novej skrine umiestniť N kníh. Každá kniha má priradený jednoznačný číselný kód a tieto kódy určujú poradie kníh v skrini. Kniha s menším kódom sa má nachádzať na rovnakej alebo vyššie umiestnenej poličke ako kniha s väčším kódom. Na každej poličke majú byť knihy s menšími kódmi umiestené naľavo od kníh s väčsími kódmi. Vstupom pre váš program bude postupnosť N čísel v_i, $1 \leq i \leq N$, kde v_i je výška i-tej knihy (v poradí podľa kódov od najmejšieho po najväčší). Pre jednoduchosť predpokladajte, že všetky knihy majú rovnakú hrúbku 1 cm. Váš program by mal zo zadaných údajov spočítať nasledujúce údaje:

- Šírku skrine – označme ju s.
- Počet poličiek v skrini – označme ho p.
- Výšku w_i i-tej poličky pre každé $1 \leq i \leq p$.
- Rozmiestenie kníh do skrine s vypočítanými parametrami, ktoré rešpektuje požiadavky na poradie kníh zmienené v zadaní tohto príkladu.

Navyše, knihovníčka Milka si praje, aby skriňa bola čo najužšia a pritom aby sa vošla do miestnosti vysokej 250 cm. Rozmiestenie kníh, ktoré váš program nájde, musí teda spĺňať ešte nasledujúce podmienky:

- Výška ľubovoľnej z kníh umiestených do i-tej poličky je najviac w_i.
- Súčet hrúbok kníh umiestených do jednej poličky je najviac s cm, t.j. táto polička obsahuje najviac s kníh.
- Výška skrine, ktorá sa rovná $w_1 + \ldots + w_p + (p+1) \times 1$ cm(predpokladáme, že šírka dosiek oddeľujících poličky v skrini je 1 cm), nesmie presiahnuť výšku miestnosti 250 cm.
- s je najmenšie možné.

Príklad:

Predpokladajme, ze Milka chce do skrine umiestiť 11 kníh, ktorých výšky v poradí podľa ich kódu sú nasledujúce: 40 cm, 10 cm, 40 cm, 25 cm, 40 cm, 25 cm, 50 cm, 40 cm, 40 cm, 25 cm a 40 cm. Jedno z optimálnych riešení vyzerá takto: Skriňa bude mať šírku pre 3 knihy a spolu 4 poličky s nasledujúcimi výškami: 40 cm, 40 cm, 50 cm a 40 cm. Výška skrine je v tomto prípade 175 cm. Na každej z horných troch poličiek budú umiestnené 3 knihy, na najspodnejšej poličke budú 2 knihy.

P-I-3

Jedna z metód spracovania textu používa nasledujúci transformačný algoritmus: Na vstupe je n-znakový reťazec $C = c_1c_2...c_n$, ktorého všetky znaky sú navzájom rôzne. Keď presunieme prvé jeho písmeno na koniec, dostaneme zrotovaný reťazec. Reťazec $c_{k+1}c_{k+2}...c_nc_1...c_k$ budeme nazývať k-krát zrotovaný reťazec a budeme ho značiť C_k. Napríklad reťazec `eldat` je 3-krát zrotovaný reťazec `datel`.

Napíšme teraz do riadkov pod seba reťazce $C = C_0$, C_1, ..., C_{n-1}. Takto dostaneme tabuľku n reťazcov. Riadky tejto tabuľky zotriedime lexikograficky (t.j. podľa abecedy). Z výslednej tabuľky si zapamätáme posledný stĺpec S a číslo riadku r, v ktorom sa po zotriedení nachádzal pôvodný reťazec. Aj keď to vyzerá dosť magicky, dvojica (S, r) nám stačí na to, aby sme vedeli jednoznačne určiť pôvodný reťazec.

Príklad transformácie

Na vstupe máme reťazec `datel`.

Pôvodná tabuľka:	Zotriedená tabuľka:
`datel`	`ateld`
`ateld`	`datel`
`telda`	`eldat`
`eldat`	`ldate`
`ldate`	`telda`

Výsledkom transformácie je teda reťazec `dltea` a číslo 2, lebo slovo `datel` je v druhom riadku zotriedenej tabuľky.

Súťažná úloha

Váš program dostane na vstupe reťazec S dĺžky n ($1 \leq n \leq 100$) a číslo r ($1 \leq r \leq n$). Všetky znaky reťazca S budú navzájom rôzne a (kvôli ľahšiemu načítavaniu) žiaden z nich nebude medzera. Úlohou vášho programu je nájsť reťazec C taký, aby výsledkom vyššie popísanej transformácie reťazca C bola práve dvojica (S, r). Môžete predpokladať, že taký reťazec existuje. Uvedomte si, že pri použití v praxi môžu mať spracovávané vstupy stovky kilobajtov, preto by bolo dobré, keby časové aj pamäťové nároky vášho algoritmu boli lepšie ako kvadratické.

Formát vstupu. Na prvom riadku vstupného súboru `bw.in` sa nachádza reťazec S, na druhom riadku jedno celé číslo r.

Formát výstupu. Výstupný súbor `bw.out` má obsahovať jediný riadok a na ňom hľadaný reťazec C.

Príklad:

bw.in	**bw.out**
`dltea` `2`	`datel`

P-I-4

Pri hľadaní úspornejších polovodičových technológii sa zistilo, že najviac energie sa spotrebúva pri mazaní informácii. Teda optimálne sú tie výpočty, pri ktorých sa žiadne informácie nestrácajú. Takýmto výpočtom sa hovorí *reverzibilné*, lebo vďaka tejto vlastnosti môžu prebiehať obomi smermi – dá sa určiť nielen zo vstupu výstup, ale aj z výstupu vstup. Aj my sa teraz vydáme do tohoto zvláštneho symetrického sveta a preskúmame, ako sa programuje „ekologicky“.

Začneme tým najjednoduchším, čo v klasických programovacích jazykoch máme, a to priradzovacím príkazom. Tu si nič také bohužiaľ nemôžeme dovoliť, lebo by sme stratili pôvodný obsah premennej, do ktorej priradzujeme! Namiesto toho zavedieme niekoľko príkazov, ktoré budú premennú modifikovať vratne:

- `premenná += hodnota` – pripočíta hodnotu k premennej
- `premenná -= hodnota` – odpočíta hodnotu od premennej
- `premenná ^= hodnota` – prixoruje hodnotu k premennej
- `premenná =:= premenná` – vymení obsah dvoch premenných

Operácia **xor** je bitová operácia, ktorá má pre dve jednobitové čísla výsledok 1 práve vtedy, keď sú rôzne. Teda $(0 \text{ xor } 0) = (1 \text{ xor } 1) = 0$ a $(0 \text{ xor } 1) = (1 \text{ xor } 0) = 1$. Viacbitové čísla sa xorujú po bitoch – i-ty bit výsledku je i-ty bit prvého čísla **xor** i-ty bit druhého čísla. Napr. $5 \text{ xor } 14 = (0101)_2 \text{ xor } (1110)_2 = (1011)_2 = 11$. Operácia **xor** má veľa užitočných vlastností, okrem iného $(x \text{ xor } y) = (y \text{ xor } x)$, $(x \text{ xor } x) = 0$, $(x \text{ xor } 0) = x$ a $((x \text{ xor } y) \text{ xor } z) = (x \text{ xor } (y \text{ xor } z))$. Podobne sa dajú zaviesť aj bitový **and** a **or**: je $(0 \text{ and } 0) = (0 \text{ and } 1) = (1 \text{ and } 0) = 0$, $(1 \text{ and } 1) = 1$, $(0 \text{ or } 0) = 0$, $(0 \text{ or } 1) = (1 \text{ or } 0) = (1 \text{ or } 1) = 1$. Tieto operácie nás až tak nebudú zaujímať, lebo nie sú reverzibilné.

Aby sme sa vyhli problémom s pretečením (čo je potom inverzná operácia?), dohodneme sa, že budeme počítať len s nezápornými celými číslami od 0 do `maxword`. Takýmto číslam budeme odteraz hovoriť *prirodzené*. Všetky operácie budeme robiť modulo (`maxword`+1), čiže výsledkom každej z operácii na prirodzených číslach bude opäť prirodzené číslo. Navyše príkaz `-=` bude naozaj inverzný k `+=` a naopak. Príkazy `^=` a `=:=` sú zjavne inverzné samy k sebe.

Čo všetko ale môže byť `hodnota`? Iste to môže byť ľubovoľná konštanta. Takisto to môže byť ľubovoľná premenná, samozrejme okrem tej, do ktorej priradzujeme. Inak by sme mohli napísať napr. „`a -= a`“, čo zjavne nie je reverzibilné. Ešte by sme mali povoliť základné aritmetické operácie – tie samy nemusia byť reverzibilné, stačí, keď reverzibilne spracujeme ich výsledok. Každý zložitejší výraz potom môžeme prepísať na výrazy s jednou operáciou, napr. „`x ^= (a*b)+(c*d)`“rozpíšeme takto:

```
t1 += a*b;
t2 += c*d;
x ^= t1+t2;
t2 -= c*d;
t1 -= a*b;
```

Pritom `t1` a `t2` sú pomocné premenné, ktoré sú na počiatku výpočtu nulové a po dopočítaní výrazu sa opäť k nulovým vrátia, takže ich môžeme znovu použiť. Podobne sa dá rozpísať do reverzibilného tvaru výpočet ľubovoľného výrazu, takže odteraz môžeme používať aj zložené výrazy (bez toho, aby sme ich museli rozpisovať).

Trik s odpočítavaním medzivýsledkov a spúšťaním častí programu odzadu sa nám ešte môže hodiť. Zadefinujme si teda, že `undo prikaz` znamená spustiť príkaz odzadu a že `wrap prikaz1 on prikaz2` najskôr vykoná `prikaz1`, potom `prikaz2` a nakoniec `undo prikaz1`. Náš príklad s postupným výpočtom výrazu by sme teda mohli prepísať nasledovne:

```
wrap
  begin
    t1 += a*b;
    t2 += c*d;
  end
on x ^= t1+t2;
```

Podmienené príkazy `if-then-else` môžeme používať, ak zaručíme, že po vykonaní podmieneného príkazu bude pravdivosť podmienky rovnaká ako pred jeho vykonaním (napríklad preto, že žiadnu z premenných, ktoré sú v podmienke, v podmienenej časti programu nemeníme). Potom totiž vieme aj pri vykonávaní výpočtu odzadu rozhodnúť, ktorou vetvou sa má výpočet vydať.

Ťažšie to bude u cyklov. Tam si nevystačíme s nemeniacou sa podmienkou – to by cyklus buď neprebehol ani raz, alebo sa opakoval do nekonečna. My budeme používať cykly typu `for`. Tie budú reverzibilné, ak vnútri cyklu nikde nemeníme riadiacu premennú cyklu ani jej hranice. Toto nie je až také veľké obmedzenie – nesmie sa to robiť ani v niektorých obyčajných programovacích jazykoch. Navyše aby nás netrápilo, čo bolo v riadiacej premennej pred začiatkom cyklu a čo je v nej po jeho skončení,

dohodneme sa, že príkaz `for` si túto premennú sám vytvorí a na konci ju zase zruší.

Príkaz `goto` pre istotu zakážeme úplne.

Procedúry môžu fungovať reverzibilne, ale musíme sa vyhnúť kopírovaniu parametrov a výsledkov. Všetky premenné preto budeme procedúre odovzdávať odkazom (v Pascale `var`, v C `*`). Lokálne premenné procedúry budú pri jej spustení nulové a procedúra ich musí pred skončením opäť uviesť do tohoto stavu. Rekurzia funguje bez problémov.

Teraz už máme všetko pripravené na to, aby sme vybudovali reverzibilný programovací jazyk. Ten náš bude príbuzný Pascalu a bude vyzerať takto:

Dátové typy. K dispozícii máme typy `word` (nezáporné celé číslo), `bit` (jednobitové číslo, t.j. 0 alebo 1, používa sa aj pre pravdivostné hodnoty ako pascalovský `boolean`) a pole `array[x..y] of typ` (`x` a `y` udávajú rozmedzie indexov a okrem čísel to môžu byť aj výrazy, ktorých hodnota sa počas existencie poľa nezmení). Prvky polí môžu byť aj polia, takto získame viacrozmerné polia. Svoj vlastný typ si môžete zaviesť deklaráciou `type identifikator = typ`, napr.:

```
type boolean = bit;
type screen = array[0..199] of array[0..319] of bit;
```

Identifikátory slúžia na pomenovanie typov, premenných a procedúr a sú to ľubovoľné reťazce písmen, číslic a znakov '_', ktoré nezačínajú číslicou a nezhodujú sa s niektorým z kľúčových slov jazyka. Malé a veľké písmená sa nerozlišujú.

Procedúry sa deklarujú konštrukciou:

```
procedure identifikator ( parametre );
  (deklarácie lokálnych typov, premenných a procedúr)
begin
  (príkazy oddelené bodkočiarkami)
end;
```

Parametre procedúry majú syntax `var meno : typ`, kde `meno` je identifikátor, ktorým sa na parameter odkazujeme vnútri procedúry. Ak je parametrov viac, oddeľujú sa pri deklarácií procedúry bodkočiarkami. Ak sú parametre rovnakého typu, môžeme zápis skracovať, napr. `procedure X(var m,n : word; var A:array[1..n] of bit);` Všetky objekty deklarované vnútri procedúry (parametre, typy, premenné aj vnorené procedúry) existujú len počas behu procedúry. Každá procedúra vidí svoje lokálne premenné a navyše aj lokálne premenné všetkých procedúr, vnútri ktorých je deklarovaná (ak sa ich názvy líšia od názvov jej lokálnych premenných). Presne rovnako to funguje v Pascale.

Premenné sú pomenované identifikátormi, musia sa vytvoriť dekla-

ráciou `var identifikator : typ;`. Pri vstupe do procedúry, v ktorej sú deklarované, majú nulovú hodnotu (v prípade poľa: všetky jeho prvky majú nulovú hodnotu) a predtým, ako premenná na konci procedúry zanikne, musí byť jej hodnota opäť nulová. Deklaráciu viac premenných toho istého typu môžeme skrátene zapísať `var i1, i2, ..., in : typ`.

Výrazy môžu obsahovať:

- konštanty (prirodzené čísla a konštanta `maxword` – najväčšie prirodzené číslo)
- premenné
- prvky polí (`pole[vyraz]`)
- aritmetické operácie, ktorých vstupom aj výstupom sú prirodzené čísla: $+$, $-$, $*$, **div** (celá časť podielu), **mod** (zvyšok po delení), **and**, **or**, **xor** (bitové operácie, definície viď vyššie) a **not** (prehodenie nulových a jednotkových bitov) – výsledky operácii sa automaticky berú modulo (`maxword`+1)
- relačné operácie (vstupom sú dve prirodzené čísla, výstupom bitová hodnota 1, ak relácia platí a 0, ak neplatí): $<$, $>$, $=$, $<=$, $>=$, $<>$
- zátvorky

Príkazy môžu byť nasledovných druhov:

- Blok: `begin` (príkazy oddelené bodkočiarkami) `end` – spôsobí postupné vykonanie príkazov, ktoré obsahuje, v danom poradí.
- Modifikačné príkazy: `premenna += vyraz` – spôsobí vyhodnotenie výrazu a jeho pripočítanie k premennej. Pritom `premenna` môže byť aj prvok poľa indexovaný nejakým výrazom. Premenná (resp. prvok poľa), ktorú príkaz modifikuje, sa už nikde v tomto príkaze nesmie vyskytnúť. Analogicky príkazy `-=` a `^=`.
- Vymieňací príkaz: `premenna =:= premenna` – vymení obsah dvoch premenných rovnakého typu. Ak sa jedná o prvky polí, nesmie sa žiadne z týchto polí používať vo výrazoch určujúcich indexy.
- Podmienený príkaz: `if podmienka then prikaz1 else prikaz2` – vyhodnotí sa podmienka (výraz s bitovým výsledkom), ak je výsledok 1, vykoná sa `prikaz1`, inak sa vykoná `prikaz2`. Platnosť podmienky sa vykonaním príslušného príkazu nesmie zmeniť. Časť `else` sa môže vypustiť, prípadné `else` sa vzťahuje k najbližšiemu neukončenému `if`.
- Príkaz cyklu: `for var identifikator = d to h do prikaz` – založí novú premennú daného mena, daný príkaz postupne vykonáva pre túto premennú nadobúdajúcu hodnoty `d`, `d`+1 až `h` a nakoniec riadiacu premennú opäť zruší. Hranice `d` a `h` sú celočíselné výrazy,

ak `d>h`, cyklus sa ani raz nevykoná. Príkaz `prikaz` musí zachovávať hodnotu riadiacej premennej aj oboch hraníc cyklu (presnejšie: môže ich modifikovať, ale na konci prechodu cyklom musia mať tú istú hodnotu ako mali na začiatku príslušného prechodu). Takisto sa dá použiť konštrukcia `h downto d` namiesto `d to h`, potom bude riadiaca premenná nadobúdať hodnoty v opačnom poradí, t.j. `h`, `h`-1, až `d`.

- Volanie procedúry: `nazov_procedury ( par1, ..., parN )` – zavolá procedúru so zadanými parametrami. Parametre môžu byť premenné alebo prvky poľa (potom ale výraz určujúci index prvku musí mať po návrate z procedúry rovnakú hodnotu ako pred vstupom do nej). Počet parametrov aj ich typy musia zodpovedať deklarácii procedúry.
- Príkaz obrátenia výpočtu: `undo príkaz` – vykoná daný príkaz „odzadu“podľa nasledujúcich pravidiel:

`undo begin`
`  p1; ...; pn`
`end`
	`begin`
	`  undo pn; ...; undo p1`
	`end`

`undo x += y`	`x -= y`
`undo x -= y`	`x += y`
`undo x ^= y`	`x ^= y`
`undo x =:= y`	`x =:= y`
`undo if x then y else z`	`if x then undo y` `  else undo z`
`undo for x = d to h do p`	`for x = h downto d do` `  undo p`
`undo P(x1; ...; xn)`	`undo` tela procedúry (`begin ... end`)
`undo undo p`	`p`

  Konštrukcia `begin p; undo p; end` teda nevykoná nič. (Aj keď počítať môže pomerne dlho.)
- Príkaz lokálneho výpočtu: `wrap prikaz1 on prikaz2` – skrátený zápis konštrukcie `begin prikaz1; prikaz2; undo prikaz1; end`.

Komentáre. Čokoľvek uzavreté v zložených zátvorkách (‘`{`’ a ‘`}`’) je komentár, ktorý počítač ignoruje (ako keby namiesto neho boli medzery). Komentár nesmie obsahovať zložené zátvorky.

Hlavný program nebudeme zavádzať. Aby sme sa vyhli problémom so vstupom a výstupom, budeme všetko programovať ako procedúry. Tie dostanú ako svoje parametre premenné obsahujúce vstupné dáta a premenné, ktoré majú byť predpísaným spôsobom modifikované podľa výstupu.

Časová a pamäťová zložitosť sú definované podobne ako v klasickom programovaní: Časová zložitosť je počet vykonaných príkazov. Veľkosť pamäte využitej v danom okamihu spočítame ako súčet veľkostí všetkých lokálnych premenných (typy bit a word majú jednotkovú veľkosť, veľkosť poľa je súčet veľkostí jeho prvkov), počtu všetkých parametrov práve zavolaných procedúr a počtu práve zavolaných procedúr. Parametre sa bez ohľadu na ich typ počítajú ako jednotka, lebo sa odovzdávajú odkazom. Potom pamäťová zložitosť je maximum množstva využitej pamäte počas behu programu. Pozor! Keďže aj náš program je procedúra, jeho vstupy a výstupy sa do pamäťovej zložitosti započítavajú len ako konštanty, aj keď to môžu byť veľké polia.

Príklad 1

Procedúra na prehodenie obsahu dvoch premenných (ktorá vlastne ukazuje, že príkaz `=:=` vieme odvodiť pomocou ostatných príkazov). Časová aj pamäťová zložitosť sú konštantné, teda $O(1)$.

```
procedure Vymen(var x,y : word);
begin         { x = X, y = Y (X,Y sú pôv. hodnoty) }
  x ^= y;     { x = X xor Y, y = Y }
  y ^= x;     { x = X xor Y, y = Y xor (X xor Y) = X }
  x ^= y      { x = (X xor Y) xor X = Y, y = X }
end;
```

Príklad 2

Procedúra na výpočet maxima zo zadaných `n` čísel. Dané je pole `X` celých čísel a premenná `max`, ku ktorej máme hľadané maximum pripočítať. To dokážeme takto: najskôr predpočítame pole `M`, kde `M[i]` bude maximum spomedzi čísel `X[1]` až `X[i]`. Potom pripočítame `M[n]` ku `max` a nakoniec `M[i]` opäť vyprázdnime. Časová aj pamäťová zložitosť sú lineárne, teda $O(n)$.

```
procedure Maximum(var n:word; var X:array [1..n] of word;
                  var max:word);
var M:array [0..n] of word;
begin
  wrap for var i=1 to n do
         if X[i]>M[i-1] then
           M[i] += X[i]
         else
           M[i] += M[i-1]
  on max += M[n];
end;
```

Súťažná úloha

Napíšte reverzibilnú procedúru `Najdi(var n:word; var X:array[1..n] of word; var co, kde:word)`. Táto procedúra má za úlohu zistiť, či sa v `n`-prvkovom poli `X` vyskytuje hodnota `co` a ak áno, prirátať k premennej `kde` pozíciu jej výskytu, teda také `i`, že `X[i]=co`. Naviac viete, že pole `X` je usporiadané vzostupne, t.j. pre každé `i`, `j`, `i<j`, platí `X[i]<X[j]`. Preto sa hodnota `co` vyskytuje v poli najviac raz. Snažte sa, aby vaše riešenie malo čo najmenšiu časovú a pamäťovú zložitosť.

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

52. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Zadania 1. kola kategórie P

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek