# 55. ročník Matematické olympiády – 2005/2006

*Úlohy domácího kola kategorie P*

Řešení každého příkladu musí obsahovat podrobný popis použitého algoritmu, zdůvodnění jeho správnosti a diskusi o efektivitě zvoleného řešení (tzn. posouzení časových a paměťových nároků programu).

V úlohách P-I-1, P-I-2 a P-I-3 je třeba k řešení připojit odladěný program zapsaný v jazyce Pascal, C nebo C++. Program se odevzdává v písemné formě (jeho výpis je tedy součástí řešení) i na disketě, aby bylo možné otestovat jeho funkčnost. Slovní popis řešení musí být ovšem jasný a srozumitelný, aniž by bylo nutno nahlédnout do zdrojového textu programu. V úloze P-I-4 je nutnou součástí řešení program pro paralelizátor.

Řešení úloh domácího kola MO kategorie P vypracujte a odevzdejte nejpozději do 15. 11. 2005. Vzorová řešení úloh naleznete po tomto datu na Internetu na adrese *http://mo.mff.cuni.cz/*. Na stejném místě jsou stále k dispozici veškeré aktuální informace o soutěži a také archiv soutěžních úloh a výsledků minulých ročníků.

## P-I-1 Pluky

Na monitoru se právě schyluje k velké bitvě mezi armádou hráče a armádou jeho počítače. Síly jsou vyrovnané, obě armády mají stejný počet pluků, ovšem jednotlivé pluky mohou být tvořeny různým počtem vojáků. Na začátku bitvy se pluky obou armád seřadí do dvou řad tak, že proti každému hráčovu pluku stojí právě jeden pluk patřící počítači. Potom začne vlastní boj. Pluky stojící proti sobě na sebe zaútočí. A protože v množství je síla, zvítězí ten z nich, který má více vojáků. Pokud náhodou mají soupeřící pluky stejný počet vojáků, vyhraje pluk patřící počítači.

Hráčova armáda má schopné špióny, kteří před bitvou zjistili, kolik vojáků má nepřítel v kterém pluku a jak jsou jeho pluky rozmístěny. Vaším úkolem je rozmístit na základě těchto informací hráčovy pluky tak, aby co nejvíce z nich svůj souboj vyhrálo.

**Soutěžní úloha:** Napište program, který vám poradí, jak nejlépe rozmístit pluky, které máte k dispozici. Na vstupu dostanete počet pluků $N$ v každé armádě a počty vojáků v každém z $2N$ pluků na bojišti. Výstupem programu bude jediné celé číslo – maximální počet pluků hráče, které mohou vyhrát svůj souboj při nějakém rozestavení.

**Formát vstupu:** První řádek vstupního souboru `pluky.in` obsahuje jedno celé číslo $N$ ($1 \leq N \leq 10\,000$) – počet pluků v každé z armád. Na druhém řádku je mezerami odděleno $N$ celých čísel $A_1, \ldots, A_N$ ($1 \leq A_i \leq 100\,000\,000$) – počty vojáků v jednotlivých plucích hráče. Na třetím řádku je mezerami odděleno $N$ celých čísel $B_1, \ldots, B_N$ ($1 \leq B_i \leq 100\,000\,000$) – počty vojáků v jednotlivých plucích patřících počítači.

**Formát výstupu:** Jediný řádek výstupního souboru `pluky.out` bude obsahovat jedno celé číslo – maximální počet pluků hráče, které mohou najednou vyhrát svůj souboj.

**Příklady:**

| `pluky.in` | `pluky.out` |
|---|---|
| 5<br>7 12 1 7 47<br>7 12 1 7 47 | 3<br>*(Pokud hráč rozmístí své pluky správně, zvítězí jeho pluky velikosti 47, 12 a jeden z pluků velikosti 7.)* |

| `pluky.in` | `pluky.out` |
|---|---|
| 4<br>10 10 10 10<br>10 10 10 10 | 0<br>*(Při jakémkoliv rozestavení všechny pluky hráče prohrají.)* |

| `pluky.in` | `pluky.out` |
|---|---|
| 5<br>1 3 5 7 9<br>2 4 6 8 10 | 4<br>*(Hráč obětuje svůj nejmenší pluk, pošle ho proti pluku velikosti 10. Ostatní pluky potom lze rozmístit tak, aby vyhrály.)* |

## P-I-2 Teleport

Vědcům se konečně podařilo vymyslet efektivní způsob cestování v časoprostoru. Jejich testovací středisko se skládá z několika lokalit. V každé lokalitě je umístěno několik teleportů. Když vstoupíme do teleportu, přemístí nás na předem zadanou lokalitu (což bychom od teleportu očekávali), ale navíc nás přemístí také v čase o zadaný počet minut (buď dopředu nebo dozadu). Vědci by chtěli zjistit, jak je cestování pomocí teleportů výhodné. Právě se nacházejí u centrálního počítače a chtěli by se jít nasvačit do bufetu. A protože čas jsou peníze, chtěli by být v bufetu co nejdříve. Pohybovat se v čase a prostoru samozřejmě chtějí jen pomocí již postavených teleportů.

**Soutěžní úloha:** Program dostane na vstupu počet lokalit $N$, které budeme označovat čísly $1, \ldots, N$. Centrální počítač je umístěn v lokalitě číslo 1, bufet má číslo $N$. Následuje celkový počet postavených teleportů $M$ a seznam těchto teleportů. Pro každý teleport je určena počáteční lokalita, koncová lokalita a změna času v minutách, jež nastane při průchodu tímto teleportem (kladné číslo znamená posun do budoucnosti, záporné do minulosti a 0 znamená, že se v koncové lokalitě ocitneme ve stejném čase, v jakém jsme nastoupili do teleportu).

Každý teleport se může použít jen tím směrem, který je uveden na vstupu. Mezi dvěma lokalitami může být vybudováno více teleportů. Dokonce může existovat i teleport, který nás přesune pouze v čase (tedy počáteční a koncová lokalita jsou u něj totožné).

Program má určit čas, kdy nejdříve se můžeme dostat do lokality $N$, jestliže se v lokalitě 1 nacházíme v čase 0. Pokud tam dokážeme být libovolně brzo (tzn. můžeme pomocí teleportů cestovat neomezeně do minulosti), nebo pokud se tam vůbec nemůžeme dostat, program o tom vydá příslušnou zprávu.

**Formát vstupu:** První řádek vstupního souboru `teleport.in` obsahuje dvě čísla $N$ a $M$ ($2 \leq N \leq 1\,000$, $0 \leq M \leq 50\,000$) oddělená mezerou. Následuje $M$ řádků, na každém z nich jsou tři čísla $A_i$, $B_i$, $T_i$ ($1 \leq A_i, B_i \leq N$, $|T_i| \leq 10\,000$) popisující teleport z lokality $A_i$ do lokality $B_i$ se změnou času $T_i$ minut.

**Formát výstupu:** Jediný řádek výstupního souboru `teleport.out` bude obsahovat zprávu „`Vedci umrou hlady`“, jestliže se od centrálního počítače nedá dostat do bufetu, resp. zprávu „`Vedci poznaji vznik vesmiru`“, jestliže můžeme cestovat do nekonečna do minulosti. Jinak bude obsahovat jedno celé číslo představující čas v minutách, kdy nejdříve se vědci dokážou dostat do bufetu.

**Příklady:**

`teleport.in`
```
3 4
1 2 5
2 3 −7
1 3 −1
1 3 16
```
`teleport.out`
```
−2
```
*(Prvním teleportem se vědci dostanou do lokality 2 v čase 5, odtud druhým do lokality 3 v čase $5 + (-7) = -2$. Ostatní možnosti jsou horší.)*

`teleport.in`
```
2 2
1 1 −1
1 2 0
```
`teleport.out`
```
Vedci poznaji vznik vesmiru
```
*(Dříve než se vědci druhým teleportem přesunou do bufetu, mohou prvním odcestovat libovolně daleko do minulosti.)*

`teleport.in`
```
4 3
1 2 −1
2 3 0
4 3 10
```
`teleport.out`
```
Vedci umrou hlady
```
*(Poslední teleport nemohou vědci použít na přesun z lokality 3 do lokality 4, jedině naopak.)*

### P-I-3 Posádky

Žil jednou jeden starý král. Jeho království tvořilo $N$ měst a mezi nimi vedla sem tam nějaká cesta. Jelikož králové bývají od přírody lakomí, v celém království nebylo zrovna mnoho udržovaných cest. Přesněji řečeno, cest bylo právě $N - 1$ a byly vedeny tak, aby se mezi každými dvěma městy v království dalo po cestách dojet (ať už přímo, nebo přes jiná města). V řeči teorie grafů takovéto síti cest říkáme strom.

Na stará kolena krále navštívila teta Paranoia a našeptala mu, že sousedé chtějí napadnout jeho království. Proto se král rozhodl, že lakota musí jít stranou a že postaví ve městech vojenské posádky. Paranoia však šeptala dál: „Zbláznil ses? Když budou dvě posádky v sousedních městech, budou si mezi sebou posílat zprávy. A víš, jak to dopadne... Nech hodně vojáků pohromadě a vzbouří se proti tobě!“

Tři dny a tři noci král nespal, až vymyslel následující kompromis: Vybere několik měst, v nichž postaví vojenské posádky. Aby mu nehrozila vzpoura, rozhodl se, že nikdy nesmějí být pohromadě více než tři posádky. Teď sedí nad mapou a vymýšlí, jak je má jenom rozmístit, aby království bylo co nejlépe zabezpečeno.

**Soutěžní úloha:** Ještě jednou si formálněji zopakujme, o co královi vlastně jde.

Na vstupu máte zadán počet měst $N$ a popis cest mezi nimi. Cest je právě $N - 1$, nikde se nekřižují, jimi tvořená síť je souvislá a spojuje všechna města. Pro každé město $i$ známe číslo $b_i$ – toto číslo udává, kolik přidá vojenská posádka v $i$-tém městě k bezpečnosti království. Královým (a vaším) úkolem je vybrat množinu měst, v nichž budou umístěny posádky. Tato množina musí splňovat následující podmínky:

- Každá její souvislá podmnožina má velikost nejvýše 3. (Množinu měst nazýváme souvislou, jestliže se mezi libovolnými dvěma městy z této množiny dá dojet po cestách, aniž bychom při tom navštívili město, které do této množiny nepatří.)
- Ze všech takovýchto množin má maximální možný součet hodnot $b_i$ – bezpečnost království.

**Formát vstupu:** První řádek vstupního souboru `posadky.in` obsahuje jedno číslo $N$ ($1 \leq N \leq 100\,000$) – počet měst v království. Města jsou očíslována od 1 do $N$. Každý z následujících $N - 1$ řádků obsahuje dvě čísla měst, která jsou spojena cestou. Můžete předpokládat, že síť cest je souvislá.

Poslední řádek vstupního souboru obsahuje $N$ celých čísel $b_1, \ldots, b_N$ ($0 \leq b_i \leq 10\,000$), která udávají, kde je jak výhodné umístit vojenskou posádku.

**Formát výstupu:** První řádek výstupního souboru `posadky.out` bude obsahovat jedno celé číslo – nejlepší dosažitelnou bezpečnost království. Druhý řádek bude obsahovat několik čísel oddělených mezerami – jednu vhodnou množinu měst, pro kterou se uvedené bezpečnosti dosáhne.

**Příklady:**

| posadky.in | posadky.out |
|---|---|
| 7<br>1 2<br>2 3<br>3 4<br>4 5<br>5 6<br>6 7<br>1 1 1 1 1 1 1 | 6<br>1 2 3 5 6 7<br>*(Všude je zisk z posádky stejný, chceme jich umístit co nejvíce.)* |

| posadky.in | posadky.out |
|---|---|
| 5<br>1 5<br>2 5<br>3 5<br>4 5<br>1 6 5 2 1000 | 1011<br>2 3 5<br>*(Zjevně chceme mít posádku ve městě 5. Potom už ale můžeme vybrat jen dvě z ostatních měst.)* |

| posadky.in | posadky.out |
|---|---|
| 5<br>1 5<br>2 5<br>3 5<br>4 5<br>4 4 4 4 5 | 16<br>1 2 3 4<br>*(Ne vždy se vyplatí vybrat město s nejvyšší hodnotou $b_i$.)* |

### P-I-4 Paralelizátor

Za sedmero horami a sedmero řekami vymyslel vynálezce Kleofáš podivný stroj, který nazval *paralelizátor*. Na první pohled vypadal paralelizátor jako obyčejný počítač... Byl tu však jeden malý, ale o to důležitější rozdíl. Za určitých okolností dokázal paralelizátor paralelně (tj. současně) spustit více větví programu, aniž by ho to jakkoliv zpomalilo. Kleofáš rychle pochopil, že jen ze slovního popisu tohoto zázraku by nikdo nebyl moc moudrý, a tak vymyslel i programovací jazyk, v němž je možné psát programy pro jeho paralelizátor.

Programy pro paralelizátor se budou od klasických lišit mimo jiné tím, že nebudou mít žádný výstup. Budeme pouze rozlišovat, zda program skončil *úspěšně* nebo *neúspěšně*. U klasických programů by to znamenalo, že nás zajímá jen tzv. exit code (návratová hodnota) programu.

Kleofášův programovací jazyk je téměř přesnou kopií jazyka Pascal. Oproti klasickému Pascalu v něm nemáme k dispozici generátor náhodných čísel (a tedy například funkci `random`), takže je předem dáno, jak bude výpočet každého programu vypadat. Zato přibyly čtyři nové příkazy: **Accept**, **Reject**, **Both**($x$) a **Some**($x$) (kde $x$ je proměnná typu integer).

Příkaz **Accept** *úspěšně* ukončí běžící program.

Příkaz **Reject** ukončí běžící program, ale *neúspěšně*. Stejný význam má i provedení standardního Pascalského příkazu **Halt** a ukončení výpočtu programu přechodem přes koncové **End.**, příkaz **Reject** definujeme jen kvůli názornosti.

V následujícím textu budeme *vytvořením kopie programu* rozumět to, že se v operační paměti vytvoří úplně přesná kopie celého programu včetně obsahu jeho proměnných – výsledek bude stejný, jako kdybychom už od začátku daný program spustili ne jednou, ale dvakrát.

Příkaz **Both**($x$) zastaví aktuálně běžící program. Vytvoří se dvě jeho identické kopie. V první z nich je hodnota proměnné $x$ nastavena na 0, v druhé na 1. Obě kopie programu jsou paralelně spuštěny, přičemž jejich výpočet pokračuje příkazem následujícím za příslušným příkazem **Both**.

Pokud obě kopie úspěšně skončí, v následujícím taktu procesoru úspěšně skončí i původní program. Jestliže jedna z kopií skončí neúspěšně (druhá přitom skončit ani nemusí), původní program v následujícím taktu skončí také neúspěšně. Ve všech ostatních případech (tj. když jedna kopie nikdy neskončí a druhá buď rovněž nikdy neskončí, nebo skončí úspěšně) původní program nikdy neskončí.

Příkaz **Some**($x$) funguje podobně. Rovněž zastaví aktuálně běžící program. Opět se vytvoří dvě jeho identické kopie, v první z nich je hodnota proměnné $x$ nastavena na 0, v druhé na 1. Obě kopie programu jsou paralelně spuštěny, přičemž jejich výpočet pokračuje příkazem následujícím za příslušným příkazem **Some**.

Jakmile některá z kopií úspěšně skončí, v následujícím taktu procesoru úspěšně skončí i původní program. Pokud obě kopie skončí neúspěšně, v následujícím taktu procesoru skončí neúspěšně také původní program. Ve všech ostatních případech (tj. když jedna kopie nikdy neskončí a druhá buď rovněž nikdy neskončí, nebo skončí neúspěšně) původní program nikdy neskončí.

Slovně můžeme tyto operace popsat následovně: Příkaz **Both** provádí „paralelní and“ – ověří, zda obě větve úspěšně skončí. Příkaz **Some** provádí „paralelní or“ – ověří, zda aspoň jedna z větví úspěšně skončí.

Netrvalo dlouho a Kleofáš si uvědomil, že na takovémto zázračném zařízení dokáže některé problémy řešit až neuvěřitelně rychle. Například testování prvočíselnosti je skutečně snadné.

**Příklad 1:** V proměnné $N$ je přirozené číslo. Napište program pro paralelizátor, který pro každou hodnotu $N$ skončí, přičemž *úspěšně* skončí právě tehdy, když $N$ je prvočíslo.

**Řešení:** Pomocí volání příkazu **Both** paralelně vygenerujeme všechna čísla od 2 do $N-1$ a najednou pro každé z nich ověříme, zda dělí $N$. Každá větev výpočtu úspěšně skončí, jestliže „její“ číslo nedělí $N$. Aby původní program úspěšně skončil, musí úspěšně skončit všechny větve, tedy žádné z vygenerovaných čísel nesmí dělit $N$. Časová složitost programu je $O(\log N)$.

```
{ VSTUP:  N : integer; }

var moc2, pocet_cifer : integer;
    cislo : integer;
    i,x : integer;

begin
   { ošetříme okrajový případ }
   if N = 1 then Reject;

   { zjistíme, kolik má N cifer ve dvojkové soustavě }
   moc2 := 1;
   pocet_cifer := 0;
   while moc2 < N do begin
      moc2 := moc2 * 2;
      inc(pocet_cifer);
   end;

   { vygenerujeme čísla od 0 do 2^pocet_cifer - 1 }
   cislo := 0;
   for i:=1 to pocet_cifer do begin
      Both(x);
      cislo := 2*cislo + x;
   end;

   { moc malé dělitele zkoušet nebudeme, prohlásíme za dobré }
   if cislo <= 1 then Accept;
   { ani příliš velké dělitele zkoušet nebudeme }
   if cislo >= N then Accept;
   { jinak zkoušíme, zda vygenerované číslo dělí N }
   if N mod cislo <> 0 then Accept;
   Reject;
end.
```

Názorně si ukážeme, jak vypadá výpočet paralelizátoru na tomto programu pro $N = 3$ a pro $N = 6$. Kopie programu, které vznikají během výpočtu, budeme číslovat v pořadí, v jakém vznikají.

Pro $N = 3$ bude výpočet probíhat následovně:

- Spustí se kopie #1 (tedy vlastně originál).
- Spočítá, že *pocet_cifer* = 2.
- Spustí se for-cyklus pro i = 1.
- Kopie #1 se zastaví, vzniknou kopie #2 a #3.
- V kopii #2 je *cislo* = 0, v kopii #3 je *cislo* = 1.
- V obou běžících kopiích pokračuje for-cyklus pro i = 2.
- Kopie #2 a #3 se zastaví, z #2 vzniknou #4 a #5, z #3 vzniknou #6 a #7.
- V kopiích #4 až #7 bude mít proměnná *cislo* hodnoty 0 až 3.
- Kopie #4 a #5 úspěšně skončí, neboť čísla 0 a 1 nechceme testovat jako dělitele.
- Kopie #2 úspěšně skončí, neboť už úspěšně skončily obě kopie, které z ní vznikly.
- Kopie #7 úspěšně skončí, neboť ani číslo 3 nechceme testovat.
- Kopie #6 úspěšně skončí, neboť 2 nedělí 3.
- Kopie #3 úspěšně skončí, neboť už úspěšně skončily obě kopie, které z ní vznikly.
- Kopie #1 (tedy původní program) úspěšně skončí, neboť už úspěšně skončily obě kopie, které z ní vznikly.

Pro $N = 6$ bude výpočet probíhat následovně:

- Podobně jako při $N = 3$ se dostaneme do situace, kdy běží kopie #8 až #15, proměnná *cislo* v nich má hodnoty postupně od 0 do 7.
- Kopie #8 a #9 (s příliš malým číslem) úspěšně skončí.
- Kopie #4 (z níž vznikly #8 a #9) úspěšně skončí.

- Kopie #14 a #15 (s příliš velkým číslem) úspěšně skončí.
- Kopie #7 (z níž vznikly #14 a #15) úspěšně skončí.
- Kopie #10 až #13 skončí – a to: #12 a #13 úspěšně (4 ani 5 nedělí 6), #10 a #11 neúspěšně (2 a 3 dělí 6).
- Kopie #5 skončí neúspěšně (obě její „děti“ skončily neúspěšně), kopie #6 skončí úspěšně.
- Kopie #2 skončí neúspěšně (neboť kopie #5 skončila neúspěšně), kopie #3 skončí úspěšně.
- Kopie #1 (tedy původní program) skončí neúspěšně.

**Příklad 2:** V proměnných $N$ a $K$ jsou přirozená čísla. Napište program pro paralelizátor, který pro každé $N$ skončí, přičemž *úspěšně* skončí právě tehdy, když $N$ má nějakého dělitele z množiny $M = \{2, 3, \ldots, 2^K - 1\}$.

**Řešení:** Pomocí volání příkazu **Some** paralelně projdeme všechna čísla $m \in M$, stačí nám, když libovolné jedno z nich dělí $N$.

(Jiný pohled na totéž řešení: Pomocí volání příkazu **Some** „uhodneme“ dělitele $m \in M$ a ověříme, zda jsme ho uhodli správně. Na náš program se můžeme dívat tak, že se nevětví, ale každé volání **Some** „uhodne“ a do $x$ dosadí „správnou“ hodnotu. Jestliže tedy $N$ má v množině $M$ dělitele, najdeme ho, jinak skončíme s nějakým číslem, které $N$ nedělí.)

Časová složitost programu je $O(K)$.

```
{ VSTUP:  N, K : integer; }

var cislo : integer;
    i,x : integer;

begin
   { paralelně zkoušíme čísla od 0 do 2^K - 1 }
   cislo := 0;
   for i:=1 to K do begin
      Some(x);
      cislo := 2*cislo + x;
   end;

   { 0 a 1 do množiny M nepatří }
   if cislo <= 1 then Reject;
   { zkusíme, zda vygenerované číslo dělí N }
   if N mod cislo = 0 then Accept;
   Reject;
end.
```

**Soutěžní úloha:**

**a)** V proměnných *jehla* a *seno* jsou dva znakové řetězce. Napište co nejrychlejší program pro paralelizátor, který pro každý vstup skončí, přičemž *úspěšně* skončí právě tehdy, když se řetězec *jehla* nachází v řetězci *seno* jako souvislý podřetězec. Váš program by tedy měl úspěšně skončit, jestliže například:

$$\textit{jehla} = \texttt{abcd}, \quad \textit{seno} = \texttt{aaabcdddaa}$$
$$\textit{jehla} = \texttt{ddda}, \quad \textit{seno} = \texttt{aaabcdddaa},$$

ale ne v případech:

$$\textit{jehla} = \texttt{abcd}, \quad \textit{seno} = \texttt{aaabcEdddaa}$$
$$\textit{jehla} = \texttt{jajsemjehla}, \quad \textit{seno} = \texttt{vtetokupesenajehlaneni}.$$

**b)** Nad polem přirozených čísel můžeme postavit „pyramidu“. Spodní řádek pyramidy bude tvořit samotné pole. Každý vyšší řádek bude o 1 kratší než předcházející, přičemž $i$-tý prvek v novém řádku je roven součtu $i$-tého a $(i+1)$-tého prvku z řádku pod ním, modulo 10 000 (tzn. pokud by součet vyšel větší než 9 999, necháme z něho v pyramidě jen jeho poslední čtyři cifry). Vrchní řádek pyramidy je tvořen jediným číslem.

V proměnné $N$ máme přirozené číslo. V poli $A$ na pozicích 1 až $N$ máme $N$ přirozených čísel menších než 10 000. V proměnné $V$ je nezáporné celé číslo menší než 10 000.

Napište co nejrychlejší program pro paralelizátor, který pro každý vstup skončí, přičemž *úspěšně* skončí právě tehdy, když hodnota $V$ je na vrcholu pyramidy postavené nad polem $A$.

**Příklad:**

| Vstup: | Výstup: | *(Pyramida vypadá následovně:)* |
|---|---|---|
| $N = 4$ | skončí neúspěšně | 45 |
| $A = (6, 3, 9, 3)$ | | 21 24 |
| $V = 17$ | | 9 12 12 |
| | | 6 3 9 3 |

| Vstup: | Výstup: | *(Pyramida vypadá následovně:)* |
|---|---|---|
| $N = 4$ | skončí úspěšně | 20 |
| $A = (1, 2, 3, 4)$ | | 8 12 |
| $V = 20$ | | 3 5 7 |
| | | 1 2 3 4 |