53. ročník matematické olympiády na středních školách

Kategorie P

In: Karel Horák (editor); Jan Kára (editor); Jaromír Šimša (editor); Jaroslav Švrček (editor); Pavel Töpfer (editor); Jaroslav Zhouf (editor): 53. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 2003/2004. 45. mezinárodní matematická olympiáda. 16. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2006. pp. 98–152.

Persistent URL: http://dml.cz/dmlcz/405076

Terms of use:

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* http://dml.cz

Kategorie P

Texty úloh

P – I – 1

Síť

Firma Připojse poskytovala svým zákazníkům velmi spolehlivé připojení na Internet. Vybudovala si pro tento účel svou vlastní datovou síť spojující několik největších měst v zemi. Síť se skládá z uzlů umístěných ve městech a z linek, které vedou mezi městy. Každá linka spojuje některé dva uzly. Síť měla tu vlastnost, že v případě přerušení jedné libovolné linky zůstala plně funkční, tzn. zbývající linky zajišťovaly propojení mezi každými dvěma městy.

Nedávno se firma rozhodla rozšířit své služby i pro zákazníky v dalších městech. Zřídila proto řadu nových linek, pomocí nichž byla tato města připojena k již existující síti. Nyní by ve firmě potřebovali vědět, zda je jejich síť stále ještě dostatečně spolehlivá, tj. zda i nadále existuje možnost spojení mezi každými dvěma městy i v případě výpadku jedné libovolné linky.

Soutěžní úloha. Napište program, který přečte ze vstupu seznam uzlů a linek tvořících síť a zjistí, zda má síť tu vlastnost, že pokud se libovolná jedna linka přeruší, všechna města v síti mohou mezi sebou nadále komunikovat pomocí zbývajících linek.

Formát vstupu: První řádek vstupního textového souboru `sit.in` obsahuje dvě kladná celá čísla n a m, kde n je počet měst propojených v síti a m je počet linek ($n \leqq 100$). Pro jednoduchost jsou města označena čísly $1, 2, \ldots, n$. Na každém z následujících m řádků vstupního souboru jsou zapsána dvě čísla, která určují města spojená linkou.

Můžete předpokládat, že je možné prostřednictvím existujících linek komunikovat mezi každými dvěma městy v síti a že žádné dvě linky nespojují stejnou dvojici měst.

Formát výstupu: Výstupní textový soubor `sit.out` obsahuje jediný řádek, na němž je zapsáno slovo „ANO“, jestliže lze v síti zadané na

vstupu komunikovat mezi libovolnými dvěma městy i po přerušení libovolné jedné linky, a slovo „NE“, pokud síť tuto vlastnost nemá.

Příklad 1:		*Příklad 2:*	
`sit.in`	`sit.out`	`sit.in`	`sit.out`
`5 6`	`ANO`	`4 4`	`NE`
`1 2`		`1 2`	*(Pokud se přeruší linka*
`2 3`		`2 3`	*mezi městy 3 a 4, tato*
`3 1`		`3 1`	*města spolu nebudou*
`3 4`		`3 4`	*moci komunikovat.)*
`4 5`			
`2 5`			

P – I – 2

AttoSoft

Vašek si založil maličkou programátorskou firmu, kterou nazval příznačně AttoSoft.[1] Nedávno se mu konečně podařilo získat prvního klienta. Ten mu dal za úkol napsat N jednoduchých programů.

Vašek je však líný sám programovat, a proto si na tuto práci najal N programátorů. Když ráno všichni přišli do práce, ukázalo se, že každý z nich umí naprogramovat pouze jeden z potřebných programů. Naštěstí každý z požadovaných programů uměl někdo z nich naprogramovat, takže se mohli pustit do práce. Objevil se však další problém. Firma AttoSoft vlastní pouze jeden počítač a na něm může v jednom okamžiku pracovat jen jeden programátor.

Vašek si uvědomil, že je důležité zvolit správné pořadí, v němž budou jednotliví programátoři pracovat u počítače. Podle podepsané smlouvy totiž programátory neplatí za vykonanou práci. Každého programátora platí za čas, který uplyne od začátku práce na celé zakázce až do okamžiku, kdy tento programátor dokončí svůj program. Vašek o každém z programátorů ví, kolik mu musí zaplatit za jednu hodinu a jak dlouho mu napsání jeho programu bude trvat. Pomozte mu zjistit, v jakém pořadí má poslat programátory pracovat na počítači, aby jim dohromady mohl zaplatit co nejméně.

Příklad. Mějme tři programátory A, B, C. Programátor A chce 100 Kč za hodinu a na svůj program potřebuje 2 hodiny. B dostane 20 Kč za ho-

---

[1] Mikro je 10^{-6}, nano je 10^{-9}, piko je 10^{-12}, femto je 10^{-15}, atto je 10^{-18}.

dinu a potřebuje na práci 5 hodin času. Programátor C požaduje 500 Kč za hodinu a bude pracovat 20 hodin.

Pokud by programovali v pořadí A, B, C, musel by Vašek zaplatit programátorovi A za 2 hodiny, programátorovi B za 7 hodin a programátorovi C za 27 hodin, což by ho přišlo celkově na 13 840 Kč. Jestliže budou programovat v pořadí C, B, A, bude to Vaška stát jenom $500 \times 20 + 20 \times 25 + 100 \times 27 = 13\,200$ Kč, takže toto pořadí je výhodnější. (Ale není to ještě nejlepší možné řešení.)

Soutěžní úloha. Napište program, který přečte ze vstupu počet programátorů, jejich hodinové mzdy a časy potřebné na jejich práci a spočítá, v jakém pořadí má Vašek nechat programátory pracovat, aby jim dohromady zaplatil nejmenší možnou částku. Má-li úloha více různých řešení, program najde a vypíše jedno libovolné z nich.

Formát vstupu: První řádek vstupního souboru `attosoft.in` obsahuje jedno kladné celé číslo N ($1 \leqq N \leqq 10\,000$), které udává počet programátorů. Následuje dalších N řádků; i-tý z nich obsahuje dvě celá čísla m_i, t_i ($0 < m_i, t_i \leqq 30\,000$), kde m_i je hodinová mzda i-tého programátora a t_i je čas v hodinách, který i-tý programátor potřebuje na napsání svého programu.

Formát výstupu: Výstupní textový soubor `attosoft.out` obsahuje N řádků. Na j-tém z nich je zapsáno jedno celé číslo z rozmezí od 1 do N — číslo programátora, který bude programovat jako j-tý v pořadí.

Příklad:

attosoft.in	attosoft.out
3	1
100 2	3
20 5	2
500 20	*(Vašek zaplatí 11 740 Kč.)*

P – I – 3

Součty

Je dáno pole $A[1..N]$ celých čísel. Napište program, který bude umět co nejrychleji provádět následující příkazy:

▷ změň hodnotu $A[x]$ na y,

▷ vypiš součet prvků $A[x] + A[x+1] + \ldots + A[y]$.

Váš program si na začátku výpočtu může pole A v rozumném čase vhodně předzpracovat.

Formát vstupu: Vstupní textový soubor `soucty.in` obsahuje předem neznámý počet řádků. Na prvním řádku souboru je uvedeno je-

diné celé číslo N ($1 \leqq N \leqq 2\,000$). Druhý řádek obsahuje původní hodnoty uložené v poli A — celá čísla $a_1, a_2, \ldots, a_N$ oddělená mezerami ($0 \leqq a_i \leqq 1\,000\,000$).

Vstupní soubor pokračuje několika řádky s příkazy, z nichž každý má některý z následujících tvarů:

$1\ x\ y\ (1 \leqq x \leqq N,\ 0 \leqq y \leqq 1\,000\,000)$	změň hodnotu $A[x]$ na y
$2\ x\ y\ (1 \leqq x \leqq y \leqq N)$	vypiš hodnotu $A[x] + \ldots + A[y]$

Vstup je ukončen řádkem obsahujícím jediné číslo 0.

Formát výstupu: Program musí vykonat všechny příkazy v pořadí, ve kterém jsou uvedeny na vstupu. Pro každý příkaz na vypsání součtu nějakých prvků pole musí do výstupního souboru `soucty.out` zapsat jeden řádek obsahující jedno celé číslo — součet příslušných prvků pole A v daném okamžiku.

Příklad:

`soucty.in`	`soucty.out`
`14`	`26`
`1 4 3 1 1 3 1 1 3 1 4 1 1 1`	`8`
`2 1 14`	`8`
`2 3 6`	`5`
`1 2 0`	
`2 3 6`	
`1 3 0`	
`2 3 6`	
`0`	

P – I – 4

Registrový počítač

V této úloze se budeme zabývat *registrovými počítači. Registr* je něco podobného jako proměnná. V registru může být uloženo *libovolně velké* nezáporné celé číslo. Na rozdíl od proměnných, které mezi sebou můžeme sčítat, odčítat a násobit, s registrem lze provádět jen tři jednoduché operace: zvětšit jeho obsah o 1, zmenšit jeho obsah o 1 (pokud se pokusíme zmenšit obsah registru obsahujícího hodnotu 0, zůstane v něm 0) a otestovat registr, zda je v něm 0. Na začátku výpočtu jsou ve všech registrech nuly.

Registrový počítač může používat neomezený počet registrů označených R_0, R_1, R_2 atd. Vedle registrů má k dispozici ještě *konečně velkou* pomocnou paměť.

Program pro registrový počítač budeme zapisovat v jazyce velmi podobném programovacímu jazyku Pascal. Programovací jazyk registrového počítače bude oproti Pascalu rozšířen například o příkazy pro práci s registry, naopak některé příkazy z Pascalu v něm budou zakázány.

Registrový počítač bude řešit úlohy následujícího typu: počítač dostane na vstupu zadáno slovo (řetězec písmen) a po nějakém čase odpoví, zda je toto slovo *správné* nebo *špatné*. Aby nám mohl odpovědět, zavedeme do programovacího jazyka speciální příkazy *Accept* a *Reject*. Jakmile se během výpočtu vykoná příkaz *Accept*, vstupní slovo je správné a výpočet končí. Jestliže se provede příkaz *Reject*, slovo je špatné a výpočet končí. Pokud se výpočet zacyklí nebo pokud skončí, aniž by se provedl příkaz *Accept* nebo *Reject*, zadané vstupní slovo je rovněž špatné.

Příkaz „přičti 1 k obsahu registru R" budeme značit $Inc(R)$, „odečti 1 od obsahu registru R" budeme značit $Dec(R)$. Výraz $Zero(R)$ je pravdivý, jestliže je v registru R nula, v opačném případě je nepravdivý. Na začátku výpočtu jsou ve všech registrech nuly.

V každém programu můžeme použít jen konečně mnoho registrů. Kromě nich můžeme použít už jen konstantní počet pomocných proměnných typu *byte*[2] (nemůžeme tedy používat pole!) a jednu speciální proměnnou *vstup* typu *char*. Obsah proměnné *vstup* lze měnit pouze provedením příkazu *Read*(*vstup*). Jestliže počítač ještě nedočetl vstupní slovo, příkaz *Read*(*vstup*) z něj přečte jedno další písmeno a uloží ho do proměnné *vstup*. Pokud počítač již vstupní slovo dočetl, příkaz *Read*(*vstup*) uloží do proměnné *vstup* speciální znak $.

Jelikož registrový počítač má kromě registrů jen konečně mnoho paměti, nemůže si dovolit používat rekurzi (neměl by si kde pamatovat návratové adresy). My pro jistotu úplně zakážeme definovat a používat v programu procedury a funkce. Zakázáno je i volání všech standardních procedur a funkcí jazyka Pascal. V aritmetických výrazech lze používat pouze proměnné (tedy ne registry!), celočíselné konstanty, celočíselné operátory +, `-`, `*`, `div`, `mod` a závorky. V podmínkách se mohou používat výrazy $Zero(R_i)$, běžné relační operátory (<, <=, …), logické spojky a závorky.

Z klíčových slov jazyka Pascal jsou tedy v programovacím jazyku registrového počítače povolena pouze následující: `var`, `begin`, `end`, `if`, `then`, `else`, `case`, `of`, `while`, `do`, `repeat`, `until`, `for`, `to`, `downto`, `div`, `mod`, `and`, `or`, `not` a `xor`.

---

[2] Taková proměnná obsahuje jedno celé číslo z rozmezí od 0 do 255.

Příklad 1. Napište program pro registrový počítač, který bude řešit následující úlohu. Na vstupu je zadán řetězec písmen a, b, c. Nechť α označuje počet těch písmen a ve vstupním řetězci, za kterými už není žádné c. Podobně nechť β je počet b, za nimiž není žádné c. Počítač má vstupní řetězec označit za správný právě tehdy, když $\alpha = \beta$.

Řešení. V registru R_1 si budeme pamatovat aktuální hodnotu α, v registru R_2 hodnotu β. Pokaždé, když přečteme ze vstupu písmeno c, oba registry vynulujeme. Na konci výpočtu jednoduše porovnáme hodnoty uložené v registrech. Rozmyslete si, že by stačilo použít jen jeden registr (v němž bychom měli hodnotu $\alpha - \beta$).

```
var vstup:char;
begin
   Read(vstup);
   while (vstup<>'$') do begin
      if (vstup='a') then Inc(R1);
      if (vstup='b') then Inc(R2);
      if (vstup='c') then begin
         while not Zero(R1) do Dec(R1);
         while not Zero(R2) do Dec(R2);
      end;
      Read(vstup);
   end;
   while not Zero(R1) do begin
      Dec(R1);
      if (Zero(R2)) then Reject;
      Dec(R2);
   end;
   if Zero(R2) then Accept;
end.
```

Příklad 2. Napište program pro registrový počítač, který bude řešit následující úlohu. Na vstupu bude zadán řetězec písmen a. Počítač ho označí za správný právě tehdy, když je jeho délka mocninou tří.

Řešení. Přečteme vstupní slovo, přičemž si do R_1 uložíme jeho délku. Potřebujeme zjistit, zda je to mocnina tří. Uloženou hodnotu proto budeme dělit třemi, dokud to půjde. Jestliže nakonec dostaneme podíl 0 a zbytek 1, původní číslo bylo mocninou tří, jinak nebylo.

```
var vstup:char;
    zbytek:byte;
```

```
begin
   Read(vstup);
   while (vstup<>'$') do begin Inc(R1); Read(vstup); end;
   if Zero(R1) then Reject;
   while true do begin
      zbytek:=0;
      while not Zero(R1) do begin
         Dec(R1);
         zbytek:=(zbytek+1) mod 3;
         if (zbytek=0) then Inc(R2);
      end;
      if (Zero(R2) and (zbytek=1)) then Accept;
      if (zbytek<>0) then Reject;
      while not Zero(R2) do begin Dec(R2); Inc(R1); end;
   end;
end.
```

Soutěžní úloha. Napište program pro registrový počítač, který bude řešit následující úlohu. Vstupem programu bude řetězec písmen a, b, c, d. Počítač ho označí jako správný právě tehdy, jestliže obsahuje nejvíce písmen a (tzn. počet písmen a obsažených ve vstupním slově je větší než počet písmen b, zároveň počet písmen a je větší než počet písmen c a zároveň také počet písmen a je větší než počet písmen d).

Například vstupní slovo baacd je tedy správné, zatímco vstupní slovo baacdbb je špatné (neboť obsahuje více písmen b než a) a ani vstup ac není správný (neboť písmen a a c je v něm stejný počet).

Základním kritériem hodnocení kvality navrženého programu bude počet registrů, které program používá. Pokuste se napsat program, kterému jich stačí co nejméně. Druhým kritériem pak bude doba výpočtu programu.

P – II – 1

Síť

Firma Truhlík a syn má ve městě N budov a chce všechny svoje budovy propojit počítačovou sítí. Vedení firmy rozhodlo, že pro K ($1 \leqq K \leqq N$) budov zakoupí vysokorychlostní připojení na Internet. Kromě toho mezi některými dvojicemi budov vybudují propojení optickým kabelem.

Dvě budovy se nacházejí v téže komponentě sítě, pokud lze mezi nimi komunikovat pomocí optických kabelů (buď mají přímé spojení, nebo

jsou spojeny nepřímo přes několik jiných budov). Aby bylo možné komunikovat mezi dvěma budovami ležícími v různých komponentách sítě, musí každá z těchto komponent obsahovat aspoň jeden počítač připojený na Internet.

Soutěžní úloha. Na vstupu jsou dána čísla N a K a pro každou dvojici budov jedno kladné celé číslo — cena za vybudování optického kabelu, který by propojil tuto dvojici budov. Navrhněte efektivní algoritmus, jenž určí, kterých K budov se má připojit na Internet a které dvojice budov se mají propojit optickým kabelem tak, aby mezi každými dvěma budovami bylo možné komunikovat a přitom aby celková cena vybudovaných optických kabelů byla co nejmenší.

Příklad:

Vstup:	*Výstup*:
$N = 4$, $K = 2$	Na Internet připojíme budovy 1 a 2,
Ceny spojení:	kabelem spojíme dvojice budov $(1,3)$, $(2,4)$ a $(3,5)$.
(1,2): 100	Cena kabelů bude 47.
(1,3): 10	
(1,4): 100	
(1,5): 300	
(2,3): 100	
(2,4): 10	
(2,5): 300	
(3,4): 47	
(3,5): 27	
(4,5): 74	

P – II – 2

AttoSoft

Vaškova programátorská firma AttoSoft je známa z úlohy P–I–2. Vaškovi se nyní podařilo získat druhého klienta. Ten mu dal opět za úkol naprogramovat N jednoduchých programů.

Vašek chce tentokrát ušetřit ještě více, a proto místo programátorů zaměstnal N studentů, na každý program jednoho studenta. Firma AttoSoft vlastní stále jen jeden počítač a na něm může v každém okamžiku pracovat jen jeden student. Hlavní problém ale spočívá v tom, že studenti mohou pracovat jen ve volných chvílích mezi přednáškami.

Student i potřebuje p_i hodin času na napsání přiděleného programu, přijde do firmy v čase s_i a musí odejít nejpozději v čase t_i. Svůj program nemusí psát najednou, může občas práci přerušit a počítač uvolnit jiným studentům.

Soutěžní úloha. Napište program, jenž určí, kdy má počítač používat který student, aby všichni stihli napsat své programy za jeden den, nebo zjistí, že to není možné.

Příklad 1:

Vstup:
$N = 3$
$p_1 = 1,\ s_1 = 3,\ t_1 = 4$
$p_2 = 2,\ s_2 = 2,\ t_2 = 5$
$p_3 = 5,\ s_3 = 1,\ t_3 = 10$

Výstup:
Student 1 pracuje od 3 do 4.
Student 2 pracuje od 2 do 3 a od 4 do 5.
Student 3 pracuje od 5 do 10.

Příklad 2:

Vstup:
$N = 2$
$p_1 = 200,\ s_1 = 300,\ t_1 = 500$
$p_2 = 200,\ s_2 = 400,\ t_2 = 600$

Výstup:
Nelze.

P – II – 3

Bageta

Kleofáš dostal dnes ráno hlad a rozhodl se připravit si obloženou bagetu se sýrem. Bagetu si můžeme představit jako úsečku dlouhou N cm, nebo přesněji jako uzavřený interval $\langle 0, N\rangle$. Každý kousek sýra tvoří rovněž uzavřený interval celočíselné délky. Jelikož Kleofáš je pedant, pokládá na bagetu kousky sýra tak, aby souřadnice jejich začátků i konců byla celá čísla. Kleofáš by chtěl, aby bageta byla pokryta sýrem přesně podle jeho představ. Na to ale potřebuje jednoduchý počítačový program, který by mu pomohl.

Soutěžní úloha. Na vstupu je dána velikost bagety N (N je celé číslo, $1 \leqq N \leqq 10^6$). Následuje P příkazů ($1 \leqq P \leqq 10^9$), přičemž každý z nich má jeden z následujících možných tvarů:

`PRIDEJ` a b	Kleofáš přidal kousek sýra sahající od a do b.
`KOLIK` c	Program vypíše zprávu, kolik kusů sýra leží na pozici c.

Váš program musí zpracovávat příkazy v pořadí, v jakém jsou uvedeny na vstupu. Pro každý příkaz `KOLIK` vypište jedno číslo — počet dosud

položených kusů sýra, které leží nad souřadnicí c. (Jelikož kousky sýra jsou uzavřené intervaly, počítají se i ty kousky, pro něž je c souřadnice jejich začátku nebo konce.)

Příklad:	*Vstup*:	*Výstup*:
	$N = 20$	
	`PRIDEJ 1 10`	
	`PRIDEJ 6 12`	
	`KOLIK 5`	1
	`KOLIK 6`	2
	`PRIDEJ 4 14`	
	`KOLIK 5`	2
	`KOLIK 16`	0

P – II – 4

Registrový počítač

V této úloze se budeme zabývat tzv. *dvousměrnými* registrovými počítači. Od registrových počítačů z úlohy P–I–4 se liší tím, že se při čtení vstupního slova dovedou vracet zpět. Oproti původní definici registrových počítačů nyní proměnná *vstup* vždy obsahuje hodnotu aktuálního písmene ze vstupu. Na začátku výpočtu je aktuálním písmenem první písmeno zadané na vstupu. Polohu aktuálního písmene můžeme v programu změnit provedením příkazů *Left* (aktuálním se stane předcházející písmeno) a *Right* (aktuálním se stane následující písmeno). Pokud by se po provedení jednoho z těchto příkazů mělo aktuální písmeno nacházet mimo zadané vstupní slovo, proměnná *vstup* bude obsahovat speciální znak `$`. (Můžeme si představovat, že před i za vstupním slovem je zapsáno dostatečně mnoho znaků `$`.)

Příklad 1. Napište program pro dvousměrný registrový počítač, který bude řešit následující úlohu: Na vstupu bude zadán řetězec písmen `a`, `b`, `c`. Nechť α označuje počet písmen `a` ve vstupním řetězci, β nechť je počet `b` a γ počet `c`. Počítač má vstupní řetězec označit za správný právě tehdy, když $\alpha = \beta = \gamma$.

Řešení. Projdeme vstupní slovo zleva doprava a v registrech R_1 a R_2 si přitom spočítáme hodnoty α a β. Když vstupní slovo dočteme, porovnáme obsahy obou registrů. Vrátíme se na začátek, při druhém průchodu vstupním slovem spočítáme v R_1 a R_2 hodnoty β a γ a opět je porovnáme. Rozmyslete si, že by stačilo použít jediný registr, v němž bychom měli hodnotu $\alpha - \beta$, resp. $\beta - \gamma$.

```
var vstup: char;
begin
   while (vstup<>'$') do begin
      if (vstup='a') then Inc(R1);
      if (vstup='b') then Inc(R2);
      Right;
   end;
   while not Zero(R1) do begin
      Dec(R1); if Zero(R2) then Reject else Dec(R2);
   end;
   if not Zero(R2) then Reject;
   { bylo stejně 'a' a 'b', vrátíme se na začátek }
   Left; while (vstup<>'$') do Left;
   Right;

   while (vstup<>'$') do begin
      if (vstup='b') then Inc(R1);
      if (vstup='c') then Inc(R2);
      Right;
   end;
   while not Zero(R1) do begin
      Dec(R1); if Zero(R2) then Reject else Dec(R2);
   end;
   if Zero(R2) then Accept;
end.
```

Příklad 2. Napište program pro dvousměrný registrový počítač, který bude řešit následující úlohu: Na vstupu bude zadán řetězec písmen a. Počítač ho označí za správný právě tehdy, když je jeho délka mocninou tří.

Řešení. Řešení bude vypadat stejně jako na jednosměrném registrovém počítači. Projdeme vstupním slovem zleva doprava, přičemž si do registru R_1 uložíme délku tohoto slova. Potřebujeme zjistit, zda je to mocnina tří. Uloženou hodnotu proto budeme dělit třemi, dokud to půjde. Jestliže nakonec dostaneme podíl 0 a zbytek 1, původní číslo bylo mocninou tří, jinak nebylo.

```
var vstup: char;
    zbytek: byte;
begin
```

```
   while (vstup<>'$') do begin Inc(R1); Right; end;
   if Zero(R1) then Reject;
   while true do begin
      zbytek:=0;
      while not Zero(R1) do begin
         Dec(R1);
         zbytek:=(zbytek+1) mod 3;
         if (zbytek=0) then Inc(R2);
      end;
      if (Zero(R2) and (zbytek=1)) then Accept;
      if (zbytek<>0) then Reject;
      while not Zero(R2) do begin Dec(R2); Inc(R1); end;
   end;
end.
```

Soutěžní úloha. Napište program pro dvousměrný registrový počítač, který bude řešit následující úlohu:

Vstupem programu bude řetězec písmen a, b, c, d. Počítač ho označí jako správný právě tehdy, jestliže je to palindrom, tzn. je stejný při čtení zepředu i zezadu. Formálně řečeno: slovo $a_1 a_2 a_3 \ldots a_{n-1} a_n$ je palindrom, jestliže $a_1 = a_n$, $a_2 = a_{n-1}$, ..., $a_{\lfloor n/2 \rfloor} = a_{\lceil n/2 \rceil}$. Tedy například vstupy bacab a dd jsou palindromy, zatímco vstupy baacdbb a bacabdccc nejsou.

P – III – 1

Agenti

Jistá nejmenovaná tajná společnost má N agentů. Z důvodu utajení může každý agent vydávat rozkazy jen několika dalším agentům. Agent, který dostane rozkaz, pošle tento rozkaz všem agentům, jimž může vydávat rozkazy. Šéfem společnosti je takový agent, který když vydá rozkaz, tak ho časem dostanou všichni agenti. (Společnost může mít i více šéfů, případně nemusí mít žádného šéfa.)

Soutěžní úloha. Na vstupu je dán počet agentů N. Agenti jsou označeni čísly od 7 (přesněji 007) do $N + 6$. Pro každého agenta je také dán seznam agentů, kterým může vydat rozkaz. Navrhněte efektivní algoritmus, který určí šéfa tajné společnosti (pokud jich existuje více, stačí nalézt jednoho libovolného z nich) nebo zjistí, že tajná společnost žádného šéfa nemá.

Příklad 1:

Vstup:	*Výstup*:
$N = 3$	Šéfem je agent 7.
Agent 7 rozkazuje agentovi 8.	
Agent 8 rozkazuje agentovi 9.	
Agent 9 rozkazuje agentovi 7.	

Příklad 2:

Vstup:	*Výstup*:
$N = 4$	Žádný agent není šéfem.
Agent 7 nerozkazuje nikomu.	
Agent 8 nerozkazuje nikomu.	
Agent 9 rozkazuje agentům 7 a 8.	
Agent 10 rozkazuje agentům 7 a 8.	

P – III – 2

Teploty

Meteorologická stanice měří každou minutu teplotu vzduchu. Meteorologové by potřebovali program, který by jim v každém okamžiku sděloval, jaká nejnižší teplota byla naměřena během posledních K minut. Vaším úkolem bude napsat tento program.

Na vstupu je dáno číslo K, následuje posloupnost naměřených teplot ukončená hodnotou $-1\,000$. Váš program musí po přečtení každé teploty ihned vypsat nejnižší z posledních K načtených teplot (resp. nejnižší ze všech načtených teplot, jestliže jich dosud bylo méně než K).

Příklad:	*Vstup*:	*Výstup*:
	$K = 3$	
	teploty:	
	9,0	9,0
	4,7	4,7
	5,3	4,7
	2,1	2,1
	9,0	2,1
	9,8	2,1
	17,0	9,0
	9,5	9,5
	−1000	

P – III – 3

Registrový počítač

V této úloze se budeme zabývat tzv. *jednosměrnými registrovými počítači* — stejnými jako v úloze P–I–4 (tam je obsažena i jejich definice).

Soutěžní úloha. a) Nechť R je řetězec tvořený písmeny `a`, `b`, `c`, `A`, `B`, `C`. Označme $m(R)$ řetězec tvořený malými písmeny obsaženými v R (ve stejném pořadí, v jakém se vyskytují v R). Analogicky označme $v(R)$ řetězec tvořený velkými písmeny v R. Řetězec $upcase(R)$ dostaneme z R tak, že nahradíme všechna malá písmena odpovídajícími velkými písmeny.

Např. jestliže R = `aaAcB`, potom $m(R)$ = `aac`, $v(R)$ = `AB` a $upcase(R)$ = `AAACB`.

Napište program pro jednosměrný registrový počítač, který bude řešit následující úlohu: Na vstupu dostane řetězec R tvořený písmeny `a`, `b`, `c`, `A`, `B`, `C`. Počítač ho má označit za správný právě tehdy, když $upcase(m(R)) = v(R)$. (Vyjádřeno slovně: když velká písmena obsažená v R tvoří „stejné“ slovo jako malá.)

Například vstupní řetězce `aA`, `Aa` a `abAcBaCABb` jsou tedy správné, zatímco řetězce `aa`, `BcbC` a `acACa` jsou špatné.

Váš program může použit libovolný konečný počet registrů, hodnotí se jen jeho správnost. Pokud si myslíte, že takový program neexistuje, dokažte to.

b) Dokažte, že pro libovolnou úlohu platí: Jestliže umíme sestrojit program pro registrový počítač, který řeší danou úlohu pomocí tří registrů, potom dokážeme sestrojit také program, který tuto úlohu řeší pomocí dvou registrů.

Jinými slovy: Ukažte postup, kterým lze libovolný existující program používající tři registry přepsat na ekvivalentní program, jenž potřebuje pouze dva registry. Nezapomeňte zdůvodnit správnost svého postupu.

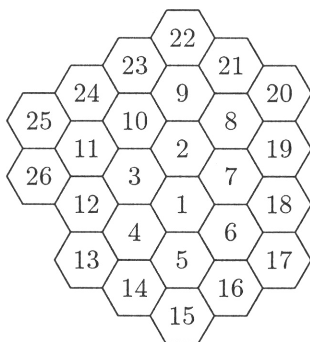
P – III – 4

Psíci

Program: `psici.pas` / `psici.c` / `psici.cpp`
Vstup: `psici.in`
Výstup: `psici.out`

„Tak, a teď budete skákat vždycky, když zapískám. A každý jinak!“ rozkázal Konrád svým dvěma psíkům. Chudáci malí, musí teď oba poskakovat po louce tak dlouho, dokud se oba současně neschovají.

Svět je nekonečná šestiúhelníková síť. Políčka tvořící svět jsou očíslována přirozenými čísly počínaje od 1 po spirále. Louku tvoří prvních N políček světa. Na obrázku je příklad louky pro $N = 26$:



Na louce stojí naši dva psíci na políčkách S_1, S_2. Skrýš pro prvního psíka je na políčku T_1, pro druhého na políčku T_2. Na M políčkách louky rostou bodláky a psíci tam za žádnou cenu neskočí.

Na jedno písknutí přeskočí každý psík na libovolné sousední políčko, pokud na něm nerostou bodláky. Oba psíci nemohou nikdy skočit současně stejným směrem a nemohou také oba dopadnout najednou na stejné políčko. Při každém písknutí musí každý z nich přeskočit na jiné políčko (i kdyby už stál ve své skrýši).

Vaším úkolem je zjistit, na kolik nejméně písknutí se mohou oba psíci dostat současně do svých skrýší.

Vstup: Ve vstupním souboru `psici.in` následuje po sobě popis několika (maximálně pěti) problémů. Každý problém má na svém prvním řádku dvě čísla N ($2 \leqq N \leqq 500$), M ($0 \leqq M \leqq N-2$), na druhém řádku čísla políček S_1, T_1, S_2, T_2 (v uvedeném pořadí, $1 \leqq S_1, T_1, S_2, T_2 \leqq N$). Následuje dalších M řádků s čísly políček, kde rostou bodláky. Vstupní soubor je ukončen řádkem obsahujícím dvě nuly ($M = N = 0$).

Můžete předpokládat, že vždy $S_1 \neq S_2$, $T_1 \neq T_2$ a že na políčkách S_1, S_2 nejsou bodláky. Na políčkách T_1 nebo T_2 bodláky být mohou — v takovém případě však úloha jistě nemá řešení.

Výstup: Do výstupního souboru `psici.out` zapište pro každý problém jeden řádek s nejmenším počtem písknutí, po němž mohou oba psíci

současně stát ve svých skrýších. Pokud není možné tohoto výsledného stavu dosáhnout, vypište do výstupního souboru místo počtu písknutí řádek se slovem „nelze“.

Příklad:

Vstupní soubor psici.in	*Výstupní soubor* psici.out
11 0	2
3 10 6 7	3
11 1	nelze
3 10 6 7	
1	
10 2	
3 10 7 8	
2	
9	
0 0	

P – III – 5

AttoSoft

Program: attosoft.pas / attosoft.c / attosoft.cpp
Vstup: attosoft.in
Výstup: attosoft.out

Programátorské firmě AttoSoft se podařilo získat dalšího klienta, který potřebuje naprogramovat N programů. Vaškovi a jeho programátorům se však do práce moc nechtělo, a tak když přišel termín odevzdání, programy ještě stále nebyly hotové. Vašek se lekl a začal studovat smlouvu, kterou se zákazníkem podepsal.

Ve smlouvě byl pro každý program uveden vzorec, podle kterého se počítá pokuta za opožděné odevzdání programu v závislosti na délce zdržení. Naštěstí není třeba zaplatit součet pokut za všechny opožděné programy, ale jen nejvyšší pokutu ze všech. Vašek se proto nyní snaží naplánovat práci na programech tak, aby zaplatil co možná nejnižší pokutu. Stejně jako dříve má i nyní k dispozici jen jeden počítač, a proto není možné pracovat na více programech najednou. Započatou práci na programu není možné přerušit.[3]

[3] Šikovnější z vás si po přečtení zbytku zadání uvědomí, že i kdyby se to smělo, stejně by se to nevyplatilo.

Soutěžní úloha. Na vstupu je pro každý z nedokončených programů uveden vzorec na výpočet pokuty a údaj, kolik dní práce je zapotřebí na jeho dokončení. Napište program, který určí rozvrh na dokončení programů, při němž Vašek zaplatí nejmenší pokutu. Vzorec na výpočet pokuty má tvar polynomu nejvýše třetího stupně $ax^3 + bx^2 + cx + d$, ve kterém jsou koeficienty a, b, c, d celočíselné nezáporné a x je počet dní, o něž se odevzdání programu opozdilo.

Vstup: První řádek vstupního souboru `attosoft.in` obsahuje kladné celé číslo N $(1 \leqq N \leqq 5\,000)$ — počet programů. Následuje N řádků, i-tý z nich obsahuje pět celých čísel l_i, a_i, b_i, c_i, d_i $(1 \leqq l_i \leqq 100, 0 \leqq$ $\leqq a_i, b_i, c_i, d_i \leqq 5\,000)$ kde l_i je počet dní potřebný na dokončení i-tého programu a a_i, b_i, c_i, d_i jsou koeficienty vzorce na výpočet pokuty. Můžete předpokládat, že za 100 000 dní se stihnou napsat všechny programy.

Výstup: Výstupní soubor `attosoft.out` obsahuje N čísel oddělených mezerami nebo konci řádků. Tato čísla představují čísla jednotlivých programů v pořadí, v němž je třeba programy dokončit, aby byla pokuta nejmenší možná. Pokud má úloha více řešení, vypište jedno libovolné z nich.

Příklad:

Vstupní soubor `attosoft.in`	*Výstupní soubor* `attosoft.out`
`3`	`1`
`10 1 0 0 0`	`3`
`3 0 0 0 10`	`2`
`1 0 0 5 0`	

Zde pokuta za program číslo 1 dokončený po deseti dnech je $10^3 = 1\,000$, za program číslo 2 dokončený po 14 dnech je pokuta 10 a za program číslo 3 dokončený po 11 dnech je $5 \cdot 11 = 55$. Vašek tedy zaplatí pokutu 1 000.

P – I – 1

Na úvod pár slov pro ty, kdo dosud neměli příležitost seznámit se alespoň se základy *teorie grafů.* V našem chápání je graf tvořen několika body, které budeme nazývat *vrcholy* grafu, některé dvojice bodů jsou spojeny čárami, kterým budeme říkat *hrany* grafu. Formálněji řečeno, (neorientovaný) graf je dvojice $G = (V, E)$, kde V je množina vrcholů a $E \subseteq \{\{x, y\}: x, y \in V\}$ je množina neuspořádaných dvojic vrcholů, tj. hran. Právě takový graf máme v naší úloze zadán na vstupu — města v zemi představují vrcholy grafu a linky jsou hrany vedoucí mezi nimi.

Řekneme, že graf je *souvislý,* jestliže se dá po jeho hranách přejít z libovolného vrcholu do libovolného jiného. Podle zadání naší úlohy je graf zadaný na vstupu souvislý. Máme zjistit, zda odstranění některé hrany souvislost grafu poruší. Hranu s touto vlastností nazýváme *most.*

Jakým způsobem můžeme zjistit, zda je zkoumaný graf souvislý? Existuje na to více různých algoritmů. Nejčastějšímu algoritmu řešícímu tento problém se říká *obarvování vrcholů* nebo také *prohledávání* grafu. Základní myšlenka algoritmu je následující. Začneme v nějakém (libovolně zvoleném) vrcholu grafu a postupně obarvujeme všechny vrcholy, kam se dokážeme po hranách grafu dostat. Když už není možné obarvit žádný další vrchol, stačí se podívat, zda jsou obarveny všechny vrcholy grafu. Vrcholy je samozřejmě třeba obarvovat systematicky tak, abychom žádný z dostupných vrcholů nevynechali. Můžeme postupovat například prohledáváním do hloubky.

Prohledávání do hloubky je podobné postupu, jakým člověk zkoumá neznámé město. Začneme tím, že se postavíme do nějakého vrcholu a obarvíme ho. Nadále budeme barvit všechny vrcholy i hrany grafu, které navštívíme. Jestliže z vrcholu, kde právě jsme, vede nějaká ještě nepoužitá (tj. neobarvená) hrana, vydáme se po ní. Pokud přijdeme do dosud nenavštíveného (tj. neobarveného) vrcholu, obarvíme ho a rekurzívně zavoláme prohledávání z něj (tedy opět se snažíme najít nepoužitou hranu, atd.). Když přijdeme do již navštíveného, a tedy obarveného vrcholu, okamžitě se vrátíme po té hraně, kterou jsme do něj přišli. Jsme-li ve vrcholu, z něhož vedou samé obarvené hrany, vrátíme se zpět tou hranou, po které jsme do vrcholu přišli poprvé. Až se tímto způsobem budeme chtít vracet z vrcholu, kde jsme začínali, prohledávání končí.

Popsaným postupem projdeme právě dvakrát (tam a zpět) po každé z hran, k nimž se dokážeme dostat, a navštívíme všechny vrcholy, ke kterým lze dojít z počátečního vrcholu. Algoritmus je tedy korektní a jeho časová složitost je $O(M + N)$. Algoritmus je možné snadno rekurzívně implementovat, jak dokládá program uvedený na konci tohoto řešení.

Nejjednodušším řešením zadané úlohy by bylo postupně vyzkoušet odstranit každou jednotlivou hranu z grafu a vždy se podívat, zda je výsledný graf ještě stále souvislý. Takové řešení by mělo časovou složitost $O(M \cdot (M + N))$ — pro každou hranu potřebujeme spustit jedno prohledávání.

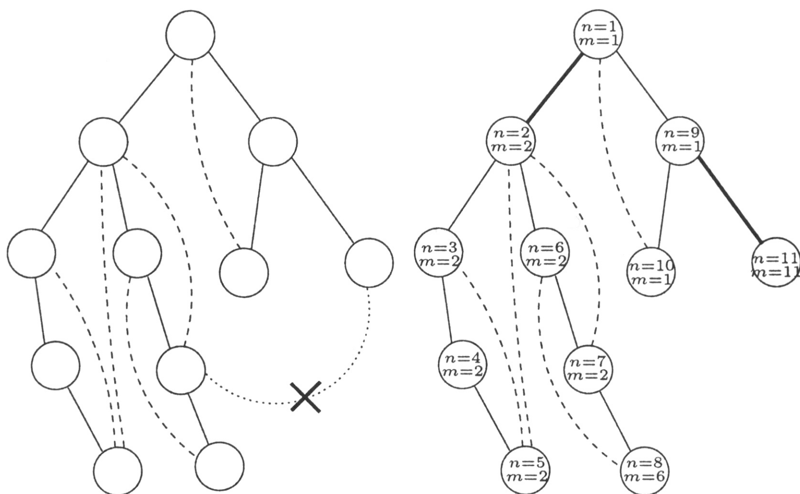
Ukážeme si však jiný algoritmus, který úlohu vyřeší v čase $O(M+N)$ (tedy s optimální časovou složitostí) a vyhledá přitom v grafu všechny mosty. Tento algoritmus je drobnou modifikací prohledávání do hloubky. Dříve než vysvětlíme samotné řešení, seznámíme se s několika potřebnými vlastnostmi prohledávání do hloubky. Začneme tedy s prohledáváním do hloubky v našem souvislém grafu. Všimněte si těch hran grafu, jimiž jsme během prohledávání přišli do dosud nenavštíveného vrcholu. Takových hran je přesně $N - 1$ (jedna pro každý vrchol grafu kromě toho, ve kterém jsme začínali s prohledáváním). Graf jimi tvořený je strom, neboť je souvislý a neobsahuje kružnice. Tento strom budeme nazývat *DFS strom* (DFS = depth-first search = prohledávání do hloubky). Vrchol, z něhož jsme graf začínali prohledávat, nazveme *kořenem* DFS stromu. Z každého jiného vrcholu x vede po *stromových* hranách (tj. po hranách DFS stromu) do kořene právě jedna cesta. Vrcholy ležící na této cestě budeme nazývat předky vrcholu x, zatímco o vrcholu x budeme říkat, že je jejich potomkem. Speciálně každý vrchol je sám sobě předkem i potomkem. Všichni potomci vrcholu x a stromové hrany vedoucí mezi nimi tvoří podstrom s kořenem x.

Ostatní hrany mohou být teoreticky dvou typů. Jestliže hrana spojuje vrchol s nějakým jeho předkem nebo potomkem, budeme ji nazývat *zpětná*, ostatní hrany nazveme *příčné*. Nechť uv je hrana, která není stromová. Všimněte si podstromů s kořeny u, v. Jsou dvě možnosti — pokud je jeden z nich podgrafem druhého, hrana uv je zpětná, jinak musí být tyto podstromy disjunktní a hrana uv je příčná. V DFS stromu však žádné příčné hrany nemohou být. To snadno zdůvodníme sporem. Nechť uv je příčná hrana. Bez újmy na obecnosti můžeme předpokládat, že během prohledávání jsme do u přišli dříve než do v. Všimněte si nyní okamžiku, kdy se při prohledávání chceme vrátit z vrcholu u zpět. Je-li uv příčná hrana, nesměli jsme dosud vrchol v navštívit (jinak by v byl

potomkem u a hrana uv by byla zpětná). Vrchol v je tedy dosud nenavštívený soused vrcholu u, proto bychom se z u ještě neměli vracet zpět, ale měli bychom se vydat do v, což je spor.

Všechny hrany grafu tedy můžeme rozdělit na stromové a zpětné. Je zřejmé, že leží-li hrana na nějaké *kružnici* (cyklu), po jejím odstranění graf zůstane souvislý. Každá zpětná hrana uv leží na kružnici tvořené hranou uv a cestou z u do v po hranách DFS stromu. Mosty se proto mohou nacházet jen mezi stromovými hranami. Každý most rozděluje graf na dvě části, přičemž v jedné z nich se nachází kořen DFS stromu.

Představte si, že náš graf zavěsíme za kořen. Nyní se vydáme z kořene dolů po stromových hranách. Uvažujme jednu konkrétní stromovou hranu uv, kde u je vrchol ležící blíže ke kořeni než v. Kdy je hrana uv mostem? Tehdy, když ji nedokážeme obejít. Jinými slovy řečeno když se z podstromu s kořenem v nemůžeme dostat do vrcholu u (nebo ekvivalentně: do u nebo libovolného jeho předka) bez použití hrany uv.



Budeme tedy chtít pro každou hranu uv určit, zda existuje cesta z v do u nebo do nějakého jeho předka, která nepoužívá hranu uv. Hledejme takovou cestu, která používá nejmenší počet zpětných hran a ze všech takových cest je nejkratší. Co o ní umíme říci? Její poslední hrana bude určitě zpětná, neboť po stromových hranách se do vrcholu u či nad u nedostaneme. Všechny její vrcholy kromě posledního budou ležet v podstromu s kořenem v, protože jakmile se dostaneme nad u, skončíme. Do všech vrcholů ležících v podstromu s kořenem v se ale jistě můžeme do-

stat z v stromovými hranami. Ukázali jsme tedy, že pokud nějaká hledaná cesta existuje, pak existuje i taková, při níž jdeme nejprve několika stromovými hranami a potom jednou zpětnou hranou. Stačí nám proto pro každou stromovou hranu v grafu ověřit, zda existuje takováto cesta. Jak to uděláme?

Během prohledávání budeme číslovat vrcholy v pořadí, v jakém do nich budeme poprvé vstupovat. Číslo vrcholu x označíme $num(x)$. Je zřejmé, že všechny vrcholy ležící v podstromu s kořenem u mají číslo větší než $num(u)$. Na druhé straně všichni předci vrcholu u mají číslo menší než $num(u)$. Kdybychom pro v znali nejmenší číslo vrcholu, do kterého se můžeme dostat bez použití hrany uv (což musí být předek vrcholu v, neboť příčné hrany neexistují), měli bychom vyhráno — hrana uv je mostem právě tehdy, když je toto číslo větší než $num(u)$. Ukázali jsme si ale, že nám stačí uvažovat cesty, které vedou nejprve několika stromovými hranami „dolů" a potom jednou zpětnou hranou „nahoru". Budeme si tedy pro každý vrchol přímo během prohledávání počítat nejmenší číslo vrcholu, do kterého se z něj dokážeme dostat takovouto cestou.

Tím máme algoritmus řešení úlohy téměř hotov, zbývá už jen celý postup shrnout. Budeme prohledávat zkoumaný graf do hloubky a zároveň si pro každý vrchol x budeme počítat dvě čísla: $num(x)$ (kolikátý navštívený vrchol to je) a $up(x) = \min\{num(y)$:do y vede z x cesta výše uvedeného tvaru$\}$. Jak vypočítat $num(x)$ je zřejmé. Hodnota $up(x)$ je rovna minimu z $num(x)$, ze všech hodnot $up(x_i)$ pro syny vrcholu x a ze všech hodnot $num(y_i)$ vrcholů, do nichž vede z x zpětná hrana. Hodnotu $up(x)$ tedy umíme spočítat v okamžiku, kdy se při prohledávání vracíme z vrcholu x. V tomto okamžiku dokážeme také rozhodnout o hraně vedoucí z vrcholu x do jeho otce y, zda je mostem — stačí porovnat hodnoty $up(x)$ a $num(y)$ (resp. $up(x)$ a $num(x)$).

```
program Sit;
var G : array[1..100,1..100] of integer; {graf}
 deg,num,up: array[1..100] of integer; {stupně vrcholů a obě čísla pro ně}
 visited : array[1..100] of boolean;     {byl jsem už v tomto vrcholu?}
 N,M,C : integer;                   {počet vrcholů, hran, navštívených vrcholů}
 ok : boolean;

procedure Load;
var i,x,y : integer;
begin
   read(N,M); fillchar(deg,sizeof(deg),0);
   for i:=1 to M do begin
      read(x,y);
      inc(deg[x]); G[x][deg[x]]:=y;
      inc(deg[y]); G[y][deg[y]]:=x;
```

```
   end;
end;

procedure DFS(v,parent : integer);
var i : integer;
begin
   visited[v]:=true;
   num[v]:=C; up[v]:=C; inc(C); { nastavíme obě čísla ve vrcholu }
   for i:=1 to deg[v] do if not visited[G[v][i]] then begin
      DFS(G[v][i],v);
      if up[G[v][i]]<up[v] then up[v]:=up[G[v][i]];
   end else begin { zpětná hrana }
      if G[v][i]<>parent then
        if num[G[v][i]]<up[v] then up[v]:=num[G[v][i]];
   end;
   if num[v]=up[v] then ok:=false; { hrana v-parent je most }
end;

begin
   Load;
   fillchar(visited,sizeof(visited),0); C:=1; ok:=true;
   DFS(1,1);
   if ok then writeln('ANO') else writeln('NE');
end.
```

P – I – 2

Uvažujme libovolné pořadí, v němž budou programátoři pracovat, a podívejme se na dva po sobě napsané programy — nechť jsou to programy i a j. Napsání programu budeme nadále označovat jako událost. První z našich událostí, tedy i, začne v čase T_0, bude trvat po dobu t_i a Vašek za ni proto zaplatí částku $(T_0 + t_i) \cdot m_i$. Druhá událost, j, začne v čase $T_0 + t_i$ (tzn. ihned po skončení události i) a bude stát $(T_0 + t_i + t_j) \cdot m_j$ — každého programátora platíme nejen za dobu, kdy pracuje, ale od úplného začátku.

Po sečtení zjistíme, že když se obě uvažované události vykonají v pořadí i, j, Vašek za ně bude muset zaplatit částku $S_{i,j} = T_0 \cdot (m_i + m_j) + t_i \cdot m_i + (t_i + t_j) \cdot m_j$.

Co by se stalo, kdybychom zaměnili pořadí událostí i a j? Podobně jako v předchozím případě můžeme spočítat, kolik bude muset Vašek zaplatit za tyto dvě události. Za první z nich (tedy j) to bude $(T_0 + t_j) \cdot m_j$ a za druhou $(T_0 + t_j + t_i) \cdot m_i$, což dohromady činí $S_{j,i} = T_0 \cdot (m_i + m_j) + t_j \cdot m_j + (t_j + t_i) \cdot m_i$.

Porovnejme nyní tyto dva výsledky. Označme si pro jednoduchost jejich společnou část $A := T_0 \cdot (m_i + m_j) + t_i \cdot m_i + t_j \cdot m_j$. Po snadných

úpravách dostáváme:

$$S_{i,j} = A + m_i \cdot m_j \cdot \frac{t_i}{m_i}, \qquad S_{j,i} = A + m_i \cdot m_j \cdot \frac{t_j}{m_j}.$$

Zajímá nás, která z těchto hodnot je menší, ale to je zjevně ta, která má menší poměr t_k/m_k. To tedy znamená, že pokud $t_i/m_i > t_j/m_j$, výměnou pořadí těchto událostí dosáhneme nižší výsledné částky. (Je zřejmé, že záměna pořadí dvou po sobě následujících událostí neovlivní částku, kterou zaplatíme ostatním programátorům.)

Z uvedených úvah vyplývá, že pokud v nějakém pořadí událostí najdeme dvě po sobě jdoucí takové, že první z nich má větší poměr t_k/m_k než druhá, jejich vzájemnou výměnou získáme nové pořadí událostí, které je levnější. Optimální pořadí událostí bude proto takové, v němž jsou poměry t_k/m_k uspořádány od nejmenšího po největší.

Samotný program je potom už velmi jednoduchý — stačí události utřídit vzestupně podle poměru t_k/m_k, což dokážeme provést v průměrném čase $O(n \cdot \log n)$ například algoritmem QuickSort.

```
program AttoSoft;
type Tprg = record
            m,t,idx: integer;
            tm: real;
            end;

var N,i: integer;
    prg: array[1..10000] of Tprg;
    sum,t: integer;

procedure QSort(l,r: integer);
var y: Tprg;
    x: real;
    i,j: integer;
begin
    i:=l; j:=r; x:=prg[(l+r) div 2].tm;
    repeat
    while prg[i].tm<x do inc(i);
    while x<prg[j].tm do dec(j);
    if i<=j then
    begin
        y:=prg[i]; prg[i]:=prg[j]; prg[j]:=y;
        inc(i); dec(j);
    end;
    until i>j;

    if l<j then QSort(l, j);
    if i<r then QSort(i, r);
end;
```

```
begin
   assign(input,'attosoft.in'); reset(input);
   assign(output,'attosoft.out'); rewrite(output);

   read(N);
   for i:=1 to N do
   begin
     read(prg[i].m, prg[i].t);
     prg[i].idx:=i;
     prg[i].tm:=prg[i].t/prg[i].m;
   end;

   QSort(1,N);

   for i:=1 to N do writeln(prg[i].idx);

   {Výsledná částka:
   sum:=0; T:=0;
   for i:=1 to N do
   begin
     sum:=sum+(T+prg[i].t)*prg[i].m;
     T:=T+prg[i].t;
   end;
   writeln('Výsledná částka: ',sum);
   }

   close(input); close(output);
end.
```

P – I – 3

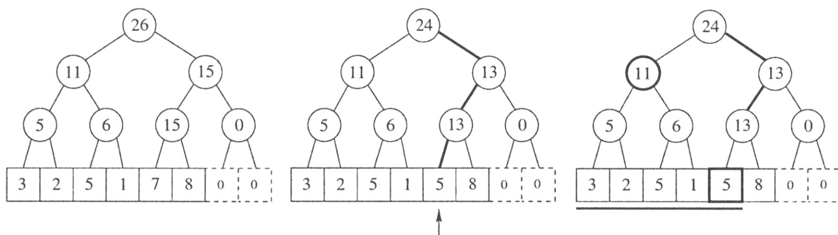
Pro zjednodušení dalších úvah zvětšíme nejprve pole A tak, aby jeho velikost byla rovna nejbližší vyšší mocnině dvou. Tím se pole A prodlouží maximálně na dvojnásobek původní délky, takže tato úprava neovlivní časovou složitost výsledného algoritmu. Nadále tedy předpokládejme, že prodloužené pole má délku $N = 2^K$.

Představme si, že nad polem A vybudujeme úplný binární strom. Jeho listy budou odpovídat jednotlivým prvkům pole A, každý vyšší vrchol tohoto stromu odpovídá nějakému intervalu v poli A (přesněji řečeno odpovídá prvkům pole určeným listy z jeho podstromu). V každém vrcholu stromu si budeme pamatovat součet čísel v příslušném intervalu pole. Tuto datovou strukturu budeme nazývat intervalový strom.

V nejspodnější vrstvě našeho stromu se nachází N vrcholů, v předcházející vyšší vrstvě jich je $N/2$, ve třetí odspodu $N/4$, atd. V celém stromě je tedy $2N - 1$ vrcholů, proto budeme potřebovat na jeho uložení paměť velikosti $\Theta(N)$ (čti: lineární). V průběhu předzpracovaní pole A musíme tuto paměť naplnit, proto na předzpracování bude zapotřebí čas $\Omega(N)$

(čti: aspoň lineární). Snadno zjistíme, že v lineárním čase dokážeme náš strom skutečně vytvořit — stačí ho zaplňovat po vrstvách zdola nahoru.

Co se stane s naším stromem, když změníme hodnotu prvku $A[j]$? Musíme změnit zapamatované hodnoty pro všechny intervaly, v nichž je změněný prvek pole obsažen. Ty ale odpovídají právě vrcholům intervalového stromu ležícím na cestě z j-tého listu do kořene. Je jich tedy $K + 1 = O(\log N)$. Změnit hodnotu v poli A tudíž dokážeme v logaritmickém čase.



Zbývá ukázat, jak lze pomocí intervalového stromu odpovídat na otázky ze zadání. Řešme nejprve jednodušší úlohu: Jakou hodnotu má součet $S(x) = A[1]+\ldots+A[x]$? Začneme v kořeni našeho stromu. Mohou nastat dvě možnosti: Jestliže interval od 1 do x leží celý v levém podstromu, zavoláme rekurzívní výpočet pro levého syna. Pokud ne, tak tento interval zabírá celý levý podstrom a ještě část pravého. Vezmeme proto součet všech prvků pole odpovídajících levému podstromu (ten máme spočítaný v levém synovi) a zavoláme rekurzívní výpočet pro pravého syna a zbytek intervalu.

Takto postupně v našem stromu procházíme dolů po cestě od kořene do x-tého listu, přitom na každé úrovni vykonáme jen konstantní počet operací. Proto pro libovolné x dokážeme hodnotu $S(x)$ spočítat v čase $O(\log N)$. To je ale vše, co potřebujeme vědět, neboť $A[x] + \ldots + A[y] = S(y) - S(x-1)$ (dodefinujeme $S(0) = 0$).

Pomocí intervalového stromu tedy dokážeme každý příkaz ze zadání úlohy zpracovat v logaritmickém čase. Naše řešení potřebuje lineární paměť a lineární čas na předzpracování.

Nejjednodušší implementací intervalového stromu je uložit ho v jednom poli podobně jako haldu. Kořen stromu bude umístěn v poli na pozici 1, synové vrcholu x jsou na pozicích $2x$ a $2x+1$. Prvky původního pole A odpovídají listům stromu a začínají v poli na pozici N. V praxi se někdy paměťová složitost snižuje na polovinu tím, že si ukládáme jen součty v levých synech, implementace je potom ale o něco náročnější.

```
program Soucty;

var T : array[1..10000] of longint; { strom }
    oldN,N,prikaz,i : longint;
    x,y : longint;
    pom : longint;

function Soucet(delka, koren, interval : longint) : longint;
{delka - délka intervalu, jehož součet počítáme
 koren - kořen podstromu, ve kterém počítáme
 interval - délka intervalu odpovídajícího kořenu
            (abychom ji nemuseli počítat)}

begin
   if delka=0 then begin Soucet:=0; exit; end;
   if interval=1 then begin Soucet:=T[koren]; exit; end;
   if delka<=(interval div 2)
      then Soucet:=Soucet(delka,2*koren,interval div 2)
      else Soucet:=T[2*koren]+
         Soucet(delka-(interval div 2),2*koren+1,interval div 2);
end;

begin
   fillchar(T,sizeof(T),0);
   read(oldN);
   N:=1; while N<oldN do N:=N*2; { upravíme velikost pole }
   for i:=1 to oldN do read(T[N+i-1]);
   for i:=N-1 downto 1 do T[i]:=T[2*i]+T[2*i+1];
   read(prikaz);
   while prikaz>0 do begin
      if prikaz=1 then begin
         { měníme hodnotu }
         read(x,y); i:=x+N-1; pom:=y-T[i];
         while i>=1 do begin Inc(T[i],pom); i:=i div 2; end;
      end else begin
         { počítáme součet }
         read(x,y);
         writeln(Soucet(y,1,N)-Soucet(x-1,1,N));
      end;
      read(prikaz);
   end;
end.
```

P – I – 4

Nejjednodušším řešením je použít čtyři registry a v každém si počítat počet písmen jednoho typu. Když dočteme slovo, v R_0 máme počet přečtených písmen a, v R_1 počet b, atd. Nyní budeme najednou zmenšovat hodnoty ve všech čtyřech registrech. *Accept* zavoláme právě tehdy, když registr R_0 zůstane nejdéle nenulový.

Počet použitých registrů lze snadno snížit na tři: Nechť jsme dosud přečetli α písmen a, β písmen b, γ písmen c a δ písmen d. V registrech si budeme ukládat absolutní hodnoty výrazů $\alpha - \beta$, $\alpha - \gamma$, $\alpha - \delta$, ve třech proměnných si budeme pamatovat jejich znaménka (např. 0, pokud je v příslušném registru nula, 1, pokud tam je kladné číslo, a 255, když je záporné.) V každém okamžiku výpočtu pak dokážeme snadno určit, zda bylo dosud na vstupu písmen a nejvíce — to platí právě tehdy, když jsou všechny tři zapamatované hodnoty kladné (tzn. všechna tři jejich znaménka rovna 1).

Naše řešení bude potřebovat jen dva registry. Je možné ukázat (v tomto vzorovém řešení to ale neuděláme), že jeden registr na vyřešení této úlohy nestačí. Naše řešení bude tudíž vzhledem k počtu registrů optimální.

V průběhu výpočtu si v R_0 budeme pamatovat číslo $2^{\alpha}3^{\beta}5^{\gamma}7^{\delta}$, registr R_1 budeme používat pouze na pomocné výpočty. Když například přečteme ze vstupu jako další písmeno b, pomocí registru R_1 vynásobíme obsah registru R_0 třemi. Po dočtení vstupu potřebujeme porovnat hodnoty α, β, γ a δ. Podobně jako v prvním řešení je budeme najednou zmenšovat (což v tomto případě znamená dělit obsah R_0 vhodným číslem) a akceptujeme právě tehdy, když nám na konci zůstane kladná mocnina 2.

Samotný program je sice trochu delší, ale je jen přímočarou implementací uvedené myšlenky.

```
var c:char;
    d,e,f:byte;

begin
 { čteme vstup a kódujeme do R0, kolik v něm čeho je }
 Inc(R0);
 Read(c);
 while c<>'$' do begin
    case c of
       'a': d:=2;
       'b': d:=3;
       'c': d:=5;
       'd': d:=7;
       end;
    while not Zero(R0) do begin   { R1 := R0 * d, R0 := 0 }
       Dec(R0);
       for e:=1 to d do Inc(R1);
 end;
    while not Zero(R1) do begin   { R0 := R1, R1 := 0 }
       Dec(R1);
       Inc(R0);
   end;
```

```
   Read(c);
   end;

 { v každé iteraci z R0 odebereme jedno "a" }
 { a po jednom z dosud zbývajících ostatních písmen }
 while true do begin
  e := 0;                    { e := R0 mod 210, R1 := R0 div 210, R0 := 0 }
  while not Zero(R0) do begin   { (210 = 2*3*5*7) }
     Dec(R0);
     e := (e+1) mod 210;
 if e=0 then Inc(R1);
     end;
  d := 1;                    { zjistíme, čím vším bylo R0 ještě dělitelné }
                             { ale stačí, když budeme testovat e místo R0 }
  if e mod 2 = 0 then d := d*2;
  if e mod 3 = 0 then d := d*3;
  if e mod 5 = 0 then d := d*5;
  if e mod 7 = 0 then d := d*7;
  if d=2 then Accept;           { už zbývají jen a-čka, což je dobré }
  if e mod 2 <> 0 then Reject;
                          { a-čka došla, ale zbyla jiná písmena => špatné }
  while not Zero(R1) do begin { V R0 má být původní R0 div d, což získáme}
     Dec(R1);             { tak, že nejprve spočteme (210 div d) * R1 ... }
     for f := 1 to 210 div d do Inc(R0);
  end;
  for f := 1 to e div d do Inc(R0);
                             { ... a pak přičteme e div d; R1 máme nulové }
  end;
end.
```

P – II – 1

Zadanou úlohu si převedeme do řeči teorie grafů. Budovy firmy představují *vrcholy* našeho grafu, *hrany* grafu odpovídají možným propojením optickým kabelem. Úlohu vyřešíme nejprve pro případ $K = 1$. V tomto případě je naším úkolem vybrat takovou množinu hran, aby všechny vrcholy byly navzájem propojeny (ne nutně přímo). Taková množina hran se nazývá *kostra grafu*, a jelikož chceme, aby součet cen hran v kostře byl co nejmenší, řešíme problém hledání *minimální kostry*.

Rozmyslete si, že minimální kostra grafu neobsahuje žádný cyklus — kdyby totiž nějaký obsahovala, mohli bychom jeho libovolnou hranu odstranit. Na druhé straně když do minimální kostry přidáme libovolnou hranu, vznikne nám cyklus, neboť vrcholy, mezi nimiž vede přidaná hrana, byly už spojeny pomocí nějakých hran kostry.

Hledání minimální kostry (Primův algoritmus). Algoritmus je založen na následující myšlence. Vrcholy grafu rozdělíme na dvě skupiny: na připojené a nepřipojené. Na začátku algoritmu zvolíme libovolný vrchol

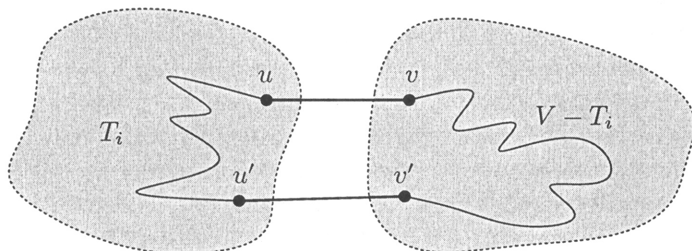
a prohlásíme ho za připojený, ostatní vrcholy jsou zatím nepřipojené. V každém kroku algoritmu připojíme jeden vrchol k dosud vytvořené síti následujícím způsobem. Najdeme nejkratší hranu spojující připojený a nepřipojený vrchol. Tuto hranu přidáme do sítě a její druhý konec se stane připojeným vrcholem. Skončíme ve chvíli, když jsou všechny vrcholy připojeny.

Aby byl algoritmus efektivní, potřebujeme umět rychle nalézt nejkratší hranu spojující připojený a nepřipojený vrchol. To zařídíme tak, že pro každý dosud nepřipojený vrchol si budeme pamatovat, ze kterého připojeného vrcholu k němu vede nejkratší hrana. Pokaždé, když přidáme k připojeným vrcholům další vrchol, musíme si informaci o nejbližších připojených vrcholech aktualizovat. Projdeme všechny nepřipojené vrcholy a pokud je nově připojovaný vrchol bližší, naši informaci změníme.

Skutečnost, že výsledná množina hran tvoří kostru, je zřejmá. Je však třeba dokázat, že je tato kostra minimální. Představme si libovolnou minimální kostru (dále ji budeme označovat MK) a porovnávejme ji s výsledkem našeho algoritmu (dále VNA).

Jestliže MK a VNA jsou shodné, VNA je minimální kostra. Předpokládejme tedy, že MK a VNA nejsou shodné. Nechť T_i je množina připojených vrcholů po i-tém kroku našeho algoritmu. Seřadíme hrany ve VNA podle toho, jak jsme je přidávali, a najdeme první hranu, která se vyskytuje ve VNA, ale není obsažena v MK. Nechť tato hrana byla přidána v kroku $i+1$ a nechť spojuje vrchol $u \in T_i$ a vrchol $v \notin T_i$.

Přidejme hranu (u, v) do MK. Tím vznikne v MK cyklus, který začíná v T_i, přejde po hraně (u, v) ven z T_i a potom se vrátí nějakou cestou zpět do T_i (obr. 41). Na této cestě musí existovat aspoň jedna hrana (u', v'),



Obr. 41. Přidáním hrany (u, v) vznikne v MK cyklus, který začíná v T_i, přejde po hraně (u, v) ven z T_i a potom se vrátí nějakou cestou zpět do T_i.

která má jeden konec v T_i a druhý konec mimo T_i. Cena této hrany musí být aspoň taková, jako je cena hrany (u, v). V opačném případě by si

náš algoritmus v kroku $i + 1$ musel vybrat hranu (u', v') namísto hrany (u, v). Proto pokud hranu (u', v') odebereme z MK a přidáme tam místo ní hranu (u, v), cena MK se nezvýší. Nemůže se však ani snížit, neboť MK je minimální. Upravená MK bude tedy nadále minimální kostrou v grafu. Navíc VNA a MK se nyní shodují v prvních $i + 1$ hranách. Stejným způsobem postupně přeměníme MK na VNA, přičemž nezvýšíme její cenu, takže VNA musí být také minimální kostrou.

Řešení pro obecné K. Dosud jsme předpokládali $K = 1$. Jestliže $K > 1$, nemusíme hranami pospojovat všechny vrcholy. Ke komunikaci totiž můžeme využít také Internet. Stačí, když se naše síť bude skládat z K souvislých částí, v každé z těchto souvislých částí vybereme jeden vrchol, který připojíme na Internet, a tak bude moci komunikovat každý vrchol s každým.

Takovouto síť můžeme získat například odebráním $K - 1$ nejdražších hran z minimální kostry MK. Tím se nám totiž MK rozpadne právě na K souvislých částí. Jediným problémem je ukázat, že toto řešení je skutečně nejlevnější možné.

Označme tedy symbolem P množinu $K - 1$ nejdražších hran kostry MK. Jejich odebráním z MK dostaneme množinu hran Q, která se skládá z K souvislých částí. Nechť existuje levnější množina hran T, která rovněž tvoří síť složenou z K souvislých částí.

Budeme uvažovat graf tvořený kostrou MK a hranami z množiny T. Jelikož už MK je souvislá, tento graf je jistě souvislý. Proto lze zvolit několik hran z MK, jimiž se dají jednotlivé komponenty T pospojovat. Každá přidaná hrana spojí dvě komponenty do jedné větší, takže stačí přidat $K-1$ hran. Množinu těchto přidaných hran označíme symbolem S.

Všimněte si následujících dvou skutečností:

- ▷ Množina hran S určitě není dražší než P, neboť obě obsahují $K - 1$ hran z kostry MK, ale P jsme vybrali tak, aby obsahovala nejdražší hrany.
- ▷ Podle našeho předpokladu množina hran T je levnější než množina hran Q.

Z toho ale vyplývá, že kostra $S \cup T$ je levnější než MK $= P \cup Q$, což je spor s tím, že MK je minimální kostra. Tím jsme ukázali, že k vyřešení úlohy stačí z MK odebrat $K - 1$ nejdražších hran.

Časová složitost. Při hledání minimální kostry se v každém kroku přidá jeden vrchol do množiny připojených, vykoná se tedy celkem $N - 1$ kroků. V každém kroku nejprve v čase $O(N)$ najdeme nejkratší hranu

spojující připojený a nepřipojený vrchol. Potom aktualizujeme informaci o nejbližším připojeném vrcholu pro všechny dosud nepřipojené vrcholy. Tato aktualizace představuje opět provedení $O(N)$ operací. Celková časová složitost je proto kvadratická, tj. $O(N^2)$.

Z výsledné minimální kostry potom potřebujeme odebrat $K-1$ nejdražších hran. To můžeme udělat tak, že hrany kostry setřídíme. Třídění lze provést v čase $O(N \log N)$, v tomto případě nám ovšem stačí použít jednoduché třídění pracující v čase $O(N^2)$. Celková časová složitost je $O(N^2)$.

```
program Sit_P_II_1;
const
   maxn = 1000;
   nekonecno = 10000;

var
   N,K: integer; { počet budov, počet komponent }
   a: array[1..maxn,1..maxn] of integer; { ceny spojení }
   sit: array[1..maxn,1..2] of integer; { seznam hran výsledné sítě }

procedure nacti_vstup;
var
   i,j: integer;
begin
   write('Počet budov N:'); readln(N);
   write('Počet internetových připojení K:'); readln(K);
   for i:=1 to N do
      for j:=i+1 to N do begin
         write('Cena (',i,',',j,'):'); readln(a[i,j]);
         a[j,i]:=a[i,j];
      end;
end; {nacti_vstup}

procedure minimalni_kostra;
{najde minimální kostru a uloží její hrany do pole sit}
var
   pripojene: array[1..maxn] of boolean;
   nej: array[1..maxn] of integer;
   i,j: integer;
   min, nejlepsi: integer;

begin
   {na začátku je jenom vrchol 1 připojený}
   pripojene[1]:=true;
   for i:=2 to N do begin
      {v poli nej si budeme pro každý dosud nepřipojený vrchol
       udržovat nejbližší připojený vrchol}
      pripojene[i]:=false;
      nej[i]:=1;
   end;

   for i:=1 to N-1 do begin
```

```
                    {najdeme nejkratší hranu, která spojuje
                     připojený a nepřipojený vrchol}
      min:=nekonecno;
      for j:=1 to N do begin
         if not pripojene[j] then begin
            if a[j,nej[j]]<min then begin
               nejlepsi:=j; min:=a[j,nej[j]]
            end;
         end;
      end;

      {nalezený vrchol připojíme}
      pripojene[nejlepsi]:=true;
      {spojení (nejlepsi,nej[nejlepsi]) patří do sítě}
      sit[i,1]:=nejlepsi; sit[i,2]:=nej[nejlepsi];

      {přepočítáme pole nej: pro každý vrchol zjistíme, zda právě
      připojený vrchol nezkrátí jeho vzdálenost k připojeným vrcholům}
      for j:=1 to N do begin
         if not pripojene[j] then begin
            if a[j,nej[j]]>a[j,nejlepsi] then nej[j]:=nejlepsi;
         end;
      end;
   end;
end; {minimalni_kostra}

procedure utrid_hrany_site;
{setřídí hrany sítě od nejlevnější po nejdražší}
var
   i,j,k,min: integer;
begin
   for i:=1 to N-1 do begin
      min:=i;
      for j:=i+1 to N-1 do begin
         if a[sit[j,1],sit[j,2]]<a[sit[min,1],sit[min,2]] then
            min:=j;
      end;
      k:=sit[min,1]; sit[min,1]:=sit[i,1]; sit[i,1]:=k;
      k:=sit[min,2]; sit[min,2]:=sit[i,2]; sit[i,2]:=k;
   end;
end; {utrid_hrany_site}

procedure vypis_vysledek;
{vypíše výsledné hrany sítě}
var
   i: integer;
begin
   writeln('Je třeba vybudovat spojení mezi následujícími budovami:');
   for i:=1 to N-K do begin
      writeln('(',sit[i,1],',',sit[i,2],')');
   end;
end; {vypis_vysledek}

begin
   nacti_vstup;
```

```
   minimalni_kostra;
   utrid_hrany_site;
   vypis_vysledek;
end.
```

P – II – 2

Uvažujme, který ze studentů má pracovat u počítače v daném okamžiku t. Zřejmě to musí být jeden z těch studentů, kteří už přišli do firmy, ale ještě nedokončili svůj program. O těchto studentech řekneme, že jsou *aktivní* v čase t. Dokážeme, že v optimálním řešení vždy můžeme poslat pracovat na počítači toho z aktivních studentů, který musí odejít nejdříve (nechť je to student a). Kdyby totiž existoval rozvrh, ve kterém v čase t pracuje nějaký jiný student b, pak se student a musí dostat k počítači ještě v čase t' někdy mezi t a časem odchodu t_a. Student b však odchází v čase $t_b \geqq t_a$. Proto můžeme sestrojit nový rozvrh, v němž necháme studenta a chvíli pracovat na počítači v čase t a stejně dlouhou dobu potom necháme studenta b pracovat na počítači v čase t'. Jestliže byl původní rozvrh správný, je správný i takto modifikovaný rozvrh, neboť každý pracuje stejně dlouho jako v původním rozvrhu a každý pracuje před svým odchodem.

Dokázali jsme, že v každém okamžiku může pracovat ten z aktivních studentů, který musí nejdříve odejít. Kdy tedy může dojít ke změně obsazení počítače? Buď tehdy, když přijde nový student a potřebuje odejít dříve, než student právě pracující (v tom případě se vystřídají u počítače), nebo když nějaký student dokončí svůj program a uvolní počítač.

Náš algoritmus bude sestrojovat rozvrh obsazení počítače postupně od začátku do konce. Studenty seřadíme podle času jejich příchodu a u každého si zaznamenáme, jak dlouho ještě potřebuje pracovat. Budeme si také udržovat množinu aktivních studentů. V každém kroku algoritmu najdeme nejbližší událost, jež může ovlivnit rozvrh. Touto událostí je buď příchod studenta nebo ukončení práce právě pracujícího studenta. V obou případech zaktualizujeme datové struktury a potom najdeme aktivního studenta s nejbližším odchodem a přidělíme mu počítač. Pokud tento student již nemá dost času na dokončení svého programu, oznámíme, že všechny programy nelze dokončit. Správnost tohoto tvrzení přímo vyplývá z důkazu uvedeného výše.

Seřazení studentů je možné provést v čase $O(N \log N)$. Počet událostí je $2N$, neboť každý student jednou přijde a jednou dokončí program. Při každé události potřebujeme najít aktivního studenta s nejmenším časem

odchodu. Kdybychom kvůli tomu vždy procházeli všechny aktivní studenty, dostali bychom algoritmus s časovou složitostí $O(N^2)$. Algoritmus se však dá zefektivnit, jestliže uložíme aktivní studenty do haldy uspořádané podle času jejich odchodu. V takovém případě zpracování jedné události trvá jen $O(\log N)$ — když někdo přišel, vložíme ho do haldy, když někdo dokončil práci, z haldy ho odstraníme. Poté se podíváme na minimum v haldě — studenta, který bude od této chvíle pracovat u počítače. Celková časová složitost algoritmu s použitím haldy je tedy pouze $O(N \log N)$.

```
program AttoSoft_P_II_2;
type student =
   record
      prichod, odchod, zbyva: integer;
   end;

const Nekonecno = 10000;

var A: array [1..1000] of student;      { pole studentů }
    N: integer;                         { počet studentů }
    Halda: array [1..1000] of integer;  { halda podle času odchodu }
    Halda_N: integer;                   { počet studentů v haldě }

procedure trid;
begin
   { vynechána kvůli úspoře místa }
end; { trid }

procedure vloz_do_haldy(student: integer);
var i, rodic, tmp:  integer;
begin
   { vlož studenta na konec haldy a posouvej ho nahoru }
   Halda_N := Halda_N + 1;
   Halda[Halda_N] := student;
   i := Halda_N;
   while i>1 do begin
      rodic := i div 2;
      if A[Halda[i]].odchod < A[Halda[rodic]].odchod then begin
         tmp := Halda[i];
         Halda[i] := Halda[rodic];
         Halda[rodic] := tmp;
      end;
      i := rodic;
   end;
end; { vloz_do_haldy }

procedure vyber_z_haldy;
var i, dite, tmp:  integer;
begin
   { první prvek nahraď posledním a posouvej ho dolů }
   Halda[1] := Halda[Halda_N];
   Halda_N := Halda_N - 1;
```

```
   i := 1;
   while i*2<=Halda_N do begin
      dite := i*2;
      if (i*2+1<=Halda_N)
         and (A[Halda[i*2+1]].odchod < A[Halda[dite]].odchod) then begin
            dite := i*2+1;
      end;
      if A[Halda[i]].odchod > A[Halda[dite]].odchod then begin
         tmp := Halda[i];
         Halda[i] := Halda[dite];
         Halda[dite] := tmp;
      end;
      i := dite;
   end;
end; { vyber_z_haldy }

procedure nacti;
var i: integer;
begin
   readln(N);
   for i := 1 to N do begin
      readln(A[i].prichod, A[i].odchod, A[i].zbyva);
   end;
end; { nacti }

function min(a,b: integer): integer;
begin
   if a<b then min := a
   else min := b;
end; { min }

var Pracuje, Skonci: integer;       { kdo právě pracuje a kdy skončí }
    Stary_cas, Novy_cas: integer;   { aktuální a předcházející událost }
    i: integer;                     { index do pole A }

begin
   nacti;                           { načti studenty do pole A }
   A[N+1].prichod := Nekonecno;     { zarážka }
   trid;                      { seřaď prvky pole A podle položky "prichod" }
   Pracuje := -1;                   { nikdo nepracuje }
   Skonci := Nekonecno;
   Halda_N := 0;                    { inicializace haldy }
   i := 1;
   Stary_cas := 0;
   while (i <= N) or (i = N+1) and (Pracuje > 0) do begin
      { Najdi novou událost }
      Novy_cas := min(A[i].prichod, Skonci);
      writeln('Čas ',Novy_cas);
      { Pokud někdo pracoval u počítače, vyhoď ho }
      if Pracuje > 0 then begin
         writeln(Novy_cas, ': student ', Pracuje, ' od počítače.');
         A[Pracuje].zbyva := A[Pracuje].zbyva - (Novy_cas - Stary_cas);
         if A[Pracuje].zbyva = 0 then vyber_z_haldy;
      end;
```

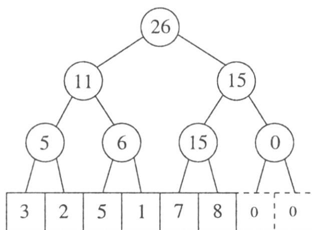
```
      { Jestliže událostí je příchod, }
      { vlož do haldy a jdi na další příchod }
      if (i <= N) and (A[i].prichod < Skonci) then begin
         vloz_do_haldy(i);
         i := i+1;
      end;

      { Najdi nového studenta k počítači }
      if Halda_N > 0 then begin
         Pracuje := Halda[1];
         Skonci := Novy_cas + A[Pracuje].zbyva;
         if Skonci > A[Pracuje].odchod then begin
            writeln('Rozvrh neexistuje!');
            exit;
         end;
         writeln(Novy_cas, ': student ', Pracuje, ' k počítači.');
      end
      else begin
         Pracuje := -1;
         Skonci := Nekonecno;
      end;
      Stary_cas := Novy_cas;
   end;
end.
```

P – II – 3

Na příkaz KOLIK c musí náš program odpovídat, v kolika dosud zadaných intervalech c leží. Na úvod uvedeme dvě triviální řešení. První je založeno na tom, že si budeme jednoduše pamatovat všechny dosud zadané intervaly a při každém příkazu KOLIK je všechny projdeme. Paměťová složitost tohoto řešení je $O(P)$, časová v nejhorším případě až $O(P^2)$. (Příkaz PRIDEJ dokážeme zpracovat v čase $O(1)$, ale na KOLIK potřebujeme v nejhorším případě až $O(P)$.) Trochu lepší řešení využívá pomocné pole velikosti $N+1$, v němž si pro každou celočíselnou pozici budeme pamatovat počet intervalů, které ji obsahují. Jeden interval přidáme v čase $O(N)$, na otázku odpovíme v čase $O(1)$. Výsledná časová složitost tohoto algoritmu je $O(N \cdot P)$, paměťová $O(N)$.

Uvědomte si, co vlastně potřebujeme zjistit, když nám přijde příkaz KOLIK c. Potřebujeme určit S — počet intervalů, které začínají na pozici $\leqq c$ a končí na pozici $\geqq c$. Nechť $Z(x)$ je počet intervalů, které začínají na pozici $\leqq x$, a $K(x)$ je počet intervalů, které končí na pozici $\leqq x$. Potom $S = Z(c) - K(c-1)$. (Intervaly, které končí před c, jsou započteny v $Z(c)$ i v $K(c-1)$, a proto je do S nezapočítáváme.) Stačilo by nám tedy umět rychle zjišťovat hodnoty $Z(x)$ a $K(x)$.

V řešení úlohy budeme využívat myšlenku z úlohy P–I–3 — datovou strukturu, kterou jsme nazvali *intervalový strom.*[4] Připomeňme si, o co šlo: Představte si, že nad polem A (jehož délku N jsme zvětšili na nejbližší mocninu dvou) vybudujeme úplný binární strom. Jeho listy budou odpovídat jednotlivým prvkům pole A, každý vyšší vrchol tohoto stromu bude odpovídat nějakému intervalu v poli A (přesněji bude odpovídat prvkům určeným listy z jeho podstromu). V každém vrcholu stromu si budeme pamatovat součet čísel v příslušném intervalu pole. Změnit hodnotu v poli A (a příslušně upravit součty ve vrcholech stromu) dokážeme v čase $O(\log N)$, zjistit součet libovolného intervalu v poli A dokážeme rovněž v čase $O(\log N)$.



$Z(c)$ je vlastně součet počtů intervalů začínajících na pozicích $0, 1, 2, \ldots, c$. Budeme mít pole, ve kterém si tyto počty budeme pamatovat, a nad ním vybudovaný *intervalový strom.* Každé přidání intervalu změní jednu hodnotu v poli, tuto změnu dokážeme uskutečnit v čase $O(\log N)$. Analogicky budeme používat druhé pole (a druhý *intervalový strom*) pro počty intervalů, které na jednotlivých pozicích končí. Pomocí těchto datových struktur dokážeme každou hodnotu Z a K spočítat v čase $O(\log N)$.

Detailnější popis obou operací s *intervalovým stromem* a jeho implementaci v poli najdete v řešeních P–I–3. Časová složitost našeho vzorového řešení je $O(P \log N)$ a paměťová $O(N)$. Všimněte si, že by nám stačilo udržovat jedno pole. Přidání intervalu $\langle a, b\rangle$ by znamenalo např. zvýšení hodnoty na pozici a a snížení hodnoty na pozici $b + 1$.

```
program Bageta_P_II_3;
var ZZ,KK: array[0..2100000] of longint; { stromy pro Z a K }
    N,oldN,a,b,c,kde: longint;
    prikaz,pom: char;

function
Soucet(var T: array of longint; delka, koren, interval: longint): longint;
```

[4] Neplést si s intervaly ze zadání!

```
{T - pole, v němž počítáme součty (všimněte si: "var T", ne "T" - proč?)
 delka - délka intervalu, jehož součet počítáme
 koren - kořen podstromu, ve kterém ho počítáme
 interval - délka intervalu odpovídajícího kořenu
 (abychom ji nemuseli počítat)}
begin
   if (delka=0) then begin Soucet:=0; exit; end;
   if (interval=1) then begin Soucet:=T[koren]; exit; end;
   if (delka<=(interval div 2))
      then Soucet:=Soucet(T,delka,2*koren,interval div 2)
      else Soucet:=T[2*koren]+Soucet(T,delka-(interval div 2),
                     2*koren+1,interval div 2);
end;

begin
 fillchar(ZZ,sizeof(ZZ),0);
 fillchar(KK,sizeof(KK),0);
 readln(oldN); N:=1; while (N<oldN+1) do N:=N*2; {upravíme velikost pole}

 while not eof do begin
   read(prikaz); pom:=prikaz; while (pom<>' ') do read(pom);
   if (prikaz='P') then begin
      readln(a,b);
      kde:=a+N; while (kde>=1) do begin Inc(ZZ[kde]); kde:=kde div 2; end;
      kde:=b+N; while (kde>=1) do begin Inc(KK[kde]); kde:=kde div 2; end;
   end else begin
      readln(c);
      writeln(Soucet(ZZ,c+1,1,N) - Soucet(KK,c,1,N));
   end;
 end;
end.
```

P – II – 4

Představte si, že bychom kromě registrů měli k dispozici ještě jeden *zásobník*[5]. Potom bychom již úlohu dokázali snadno vyřešit: Procházíme vstupním slovem zleva doprava a přečtená písmena vkládáme do zásobníku. Až potom budeme ze zásobníku písmena odebírat, budeme je dostávat v opačném pořadí, než v jakém byla do zásobníku vložena. Vrátíme se proto na začátek slova a budeme porovnávat, zda je slovo stejné odpředu jako odzadu. Vždy přečteme jedno písmeno ze vstupu, vyzvedneme jedno písmeno ze zásobníku a porovnáme je. Skončíme, když někdy dostaneme dvě různá písmena (slovo je špatné) nebo když dočteme celé vstupní slovo (slovo je správné).

Kdybychom tedy měli k dispozici zásobník, máme úlohu vyřešenou. Zásobník si však dokážeme simulovat v jednom registru (s pomocí dru-

[5] Zásobník je datová struktura, která podporuje operace „vlož prvek“ a „odeber naposledy vložený prvek“.

hého)! Jak na to? Písmena a, b, c, d budou odpovídat číslicím 1, 2, 3, 4. Číslo uložené v registru R_1 bude představovat obsah zásobníku — když ho zapíšeme v poziční soustavě o základu 5, jednotlivé cifry budou představovat hodnoty vložené do zásobníku (cifra na místě jednotek bude naposledy vložená hodnota). Například když do prázdného zásobníku vložíme postupně písmena a, c, b, a, bude v R_1 hodnota $a\times 5^3+c\times 5^2+b\times 5+a = 1\times 5^3+3\times 5^2+2\times 5+1 = 125+75+10+1 = 211$.

Jak ale s takovýmto registrem-zásobníkem pracovat? Vložit novou hodnotu x je jednoduché — pomocí registru R_2 vynásobíme obsah R_1 pěti a potom ho x-krát zvětšíme o 1. Vyzvednout naposledy vloženou hodnotu také není těžké — je to přesně opačná operace. Vydělíme obsah registru R_1 pěti, zbytek po dělení je naposledy vložená hodnota, podíl (který dostaneme v R_2) je nový obsah zásobníku bez této hodnoty.

Máme tedy funkční řešení úlohy, které potřebuje dva registry. Pokusíme se však nalézt řešení ještě lepší. Jen s jedním registrem se nám už nepodaří simulovat zásobník a musíme proto vymyslet něco jiného.

Nejprve trochu terminologie: aktuální písmeno se bude v našem řešení pohybovat sem a tam po vstupním slově. Kvůli názornosti místo „aktuální je i-té písmeno vstupního slova", resp. „přesuneme aktuální písmeno doleva/doprava" budeme říkat „stojíme na pozici i", resp. „jdeme doleva/doprava". Délku vstupního slova budeme značit n.

Představte si, že stojíme na pozici i (přičemž ale i si nijak nepamatujeme, v R_1 je nula). Chtěli bychom písmeno na této pozici porovnat s jemu odpovídajícím písmenem na pozici $n+1-i$. Náš program ovšem nezná n ani i. Jak na to? Písmeno na naší pozici si zapamatujeme v proměnné. Nyní si zjistíme i. Jdeme doleva, dokud nepřijdeme na začátek vstupního slova, a zvyšujeme R_1. Odpovídající písmeno je i-té od konce. Není tedy těžké dojít k němu — přejdeme na konec slova, potom zmenšujeme R_1 a jdeme doleva, dokud v R_1 není nula. Písmena porovnáme, a jsou-li různá, končíme. Jinak se potřebujeme vrátit zpět na pozici, kde jsme začínali. K tomu použijeme úplně stejný postup: Cestou doprava spočítáme v registru R_1 potřebný počet kroků, přesuneme se na začátek slova a vykonáme stejný počet kroků směrem doprava. Tím jsme se dostali do stejné situace, v níž jsme začínali, jen máme porovnané aktuální písmeno s jemu odpovídajícím písmenem. Celý tento postup budeme nazývat *porovnání*.

Chtěli bychom postupně porovnat všechny navzájem si příslušející dvojice písmen. To ale není problém. Začínáme na prvním písmenu vstupu a provedeme *porovnání*. Pokud není první písmeno stejné jako poslední, skončili jsme, jinak pokračujeme. Přesuneme se doprava (na druhé

písmeno) a vykonáme další *porovnání*. Takto pokračujeme tak dlouho, dokud neporovnáme n-té písmeno s prvním (a nezjistíme, že právě porovnané písmeno bylo již posledním písmenem vstupního slova).

```
var vstup: char;
    pismeno: byte;
begin
   while (vstup<>'$') do begin
         { v R1 máme nulu, začínáme porovnání }
      if (vstup='a') then pismeno:=1;
      if (vstup='b') then pismeno:=2;
      if (vstup='c') then pismeno:=3;
      if (vstup='d') then pismeno:=4;
         { spočítáme, kde jsme }
      while (vstup<>'$') do begin Inc(R1); Left; end;
         { přejdeme na pravý konec }
      Right;
      while (vstup<>'$') do Right;
         { přejdeme na odpovídající pozici }
      while not Zero(R1) do begin Dec(R1); Left; end;
         { kontrola }
      if (pismeno=1) and (vstup<>'a') then Reject;
      if (pismeno=2) and (vstup<>'b') then Reject;
      if (pismeno=3) and (vstup<>'c') then Reject;
      if (pismeno=4) and (vstup<>'d') then Reject;
         { návrat zpět }
      while (vstup<>'$') do begin Inc(R1); Right; end;
      Left;
      while (vstup<>'$') do Left;
      while not Zero(R1) do begin Dec(R1); Right; end;
         { posun na další písmeno, které je třeba zkontrolovat }
      Right;
   end;
   Accept; { všechno správně }
end.
```

P – III – 1

Úlohu si můžeme reprezentovat pomocí orientovaného grafu. Agenti představují vrcholy grafu. Skutečnost, že agent a může vydat rozkaz agentovi b, vyjádříme orientovanou hranou (a, b). Naším úkolem je nalézt v tomto grafu vrchol, z něhož se můžeme dostat do všech ostatních vrcholů.

Začneme prohledáváním grafu do hloubky z libovolného zvoleného vrcholu. Jestliže při tomto prohledávání navštívíme všechny vrcholy, našli jsme šéfa — je jím vrchol, kterým jsme prohledávání začali. V opačném případě pokračujeme tak, že zahájíme nové prohledávání do hloubky v jednom z vrcholů, které jsme dosud nenavštívili (dříve navštívené vrcholy grafu přitom necháme označené jako navštívené). To opakujeme

tak dlouho, dokud nenavštívíme všechny vrcholy našeho grafu. Nechť r je vrchol, v němž jsme zahájili poslední prohledávání.

Tvrzení. *Má-li náš graf aspoň jednoho šéfa, potom vrchol r je šéfem.*

DŮKAZ. Předpokládejme, že náš graf má šéfa a že vrchol r není šéfem. Nechť je šéfem vrchol s. Musíme uvažovat dvě možnosti:

▷ *Vrchol s byl navštíven při posledním prohledávání.* To by ale znamenalo, že se do tohoto vrcholu můžeme dostat z vrcholu r (neboť vrchol r je počátkem posledního prohledávání), tudíž se můžeme dostat z vrcholu r do libovolného jiného vrcholu grafu přes s, což je však v rozporu s naším předpokladem, že r není šéfem.

▷ *Vrchol s byl navštíven dříve než při posledním prohledávání.* Jelikož se však z vrcholu s dá dojít do libovolného vrcholu, museli bychom také vrchol r navštívit ve stejném prohledávání jako s, takže vrchol r nemůže být začátkem posledního prohledávání.

Zbývá tedy už jen ověřit (opět prohledáváním do hloubky), zda r je skutečně šéfem grafu; v opačném případě graf nemá žádného šéfa. Časová složitost celého algoritmu je $O(M + N)$, kde N je počet vrcholů a M je počet hran grafu.

```
program Agenti;

const MAXM = 10000;
      MAXN = 100;

var rozkazuje: array [1..MAXM] of integer;
    ind_od, ind_do: array [1..MAXN] of integer;
    {agent i+6 rozkazuje agentům rozkazuje[ind_od[i]]...
               rozkazuje[ind_do[i]]}
    N: integer;
    navstiven: array [1..MAXN] of boolean;

procedure Nacti;
var i,M,agent: integer;
begin
   write('Počet agentů:'); readln(N);

   M:=0;
   for i:=1 to N do begin
      write('Agent ',i+6,' rozkazuje: (ukonči -1)');
      ind_od[i]:=M+1;
      read(agent);
      while (agent>0) do begin
           M:=M+1;
           rozkazuje[M]:=agent-6;
           read(agent);
      end;
      ind_do[i]:=M;
```

```
      end;
   end; {procedure Nacti}

procedure Prohledej(i: integer);
var j: integer;
begin
   if not navstiven[i] then begin
      navstiven[i]:=true;
      for j:=ind_od[i] to ind_do[i] do begin
         Prohledej(rozkazuje[j]);
      end;
   end;
end; {procedure Prohledej}

var i,posledni: integer;
    je_sef: boolean;

begin
   Nacti;
   for i:=1 to N do navstiven[i]:=false;
   for i:=1 to N do begin
      if not navstiven[i] then begin
         posledni:=i;
         Prohledej(i);
      end;
   end;

   for i:=1 to N do navstiven[i]:=false;
   Prohledej(posledni);

   je_sef:=true;
   for i:=1 to N do je_sef:=je_sef and navstiven[i];

   if je_sef then
      writeln('Šéfem je agent ', posledni+6)
   else
      writeln('Žádný agent není šéfem');
end. {program Agenti}
```

P – III – 2

Snadno sestrojíme řešení, které potřebuje čas $O(K)$ na zpracování jedné hodnoty ze vstupu. Stačí si v cyklicky přepisovaném poli pamatovat posledních K vstupních hodnot. Pokaždé, když přečteme další číslo ze vstupu, pole jednoduše projdeme a vypíšeme nejmenší z hodnot uložených v poli.

Vzorové řešení vystačí s časem $O(\log K)$ na zpracování jednoho čísla. Představme si, že bychom si aktuálních K hodnot udržovali v haldě. Novou hodnotu do této haldy lehce přidáme v čase $O(\log K)$. Předtím, než vypíšeme minimum (které je uloženo v kořeni haldy), potřebujeme

však ještě z haldy odstranit nejstarší hodnotu. Jak ale máme vědět, která z nich to je?

Pomůžeme si tím, že hodnoty, které nám budou přicházet, vložíme nejen do haldy, ale také do fronty. Mezi těmito dvěma datovými strukturami si budeme udržovat vzájemné odkazy, abychom v každém okamžiku dokázali o každém prvku fronty říci, kde je v haldě, a naopak.

Když tedy přijde nová hodnota, vložíme ji do haldy a na konec fronty. Následně ze začátku fronty odstraníme nejstarší hodnotu, pomocí odkazu ji najdeme v haldě a odstraníme ji také odtamtud. Nyní už jen vypíšeme hodnotu uloženou v kořeni haldy.

Obě operace s haldou mají časovou složitost $O(\log K)$, zbývající operace dokážeme provést v konstantním čase. Paměť spotřebovaná haldou i frontou je $O(K)$.

```
#include <stdio.h>
#define MAXK 100047
#define INFTY 1e10
#define SWAP(x,y) pom=(x); (x)=(y); (y)=pom

typedef struct { double val; int ptr; } tZaznam;
int K;                     // ze zadání
tZaznam H[MAXK],Q[MAXK]; // halda a fronta
int qs;                    // začátek fronty
tZaznam pom;

void init(void) { // naplníme haldu i frontu "nekonečně" velkými hodnotami
  int i;
  qs=0; H[0].val=-10000;
  for (i=0;i<K;i++)
  { Q[i].val=H[i+1].val=INFTY; Q[i].ptr=i+1; H[i+1].ptr=i; }
}
void bubbleup(int idx) { // bublej prvkem nahoru v haldě
  int next=idx,p1,p2;
  if (H[idx/2].val > H[idx].val) next=idx/2;
  if (next!=idx) {
    p1=H[idx].ptr; p2=H[next].ptr;
    SWAP(H[idx],H[next]);
    Q[p1].ptr=next; Q[p2].ptr=idx;
    bubbleup(next);
  }
}
void bubbledown(int idx) { // bublej prvkem dolů v haldě
  int next=idx,p1,p2;
  if (2*idx<=K)   if (H[2*idx  ].val < H[next].val) next=2*idx;
  if (2*idx+1<=K) if (H[2*idx+1].val < H[next].val) next=2*idx+1;
  if (next!=idx) {
    p1=H[idx].ptr; p2=H[next].ptr;
    SWAP(H[idx],H[next]);
    Q[p1].ptr=next; Q[p2].ptr=idx;
    bubbledown(next);
```

```
  }
}

int main(void) {
 int delptr;
 double x;
 scanf("%d ",&K); init();
 while (1) {
   scanf("%lf ",&x); if (x==-1000) break;
   delptr=Q[qs].ptr; // najdeme hodnotu, kterou je třeba vymazat z haldy
   H[delptr].val=H[K].val; H[delptr].ptr=H[K].ptr; Q[H[K].ptr].ptr=delptr;
   K--; bubbledown(delptr); bubbleup(delptr); K++;
                                                 // smažeme a upravíme haldu
   Q[qs].val=H[K].val=x; Q[qs].ptr=K; H[K].ptr=qs; bubbleup(K); // vložíme
   qs=(qs+1)%K;
   printf("%g\n",H[1].val);
 }
 return 0;
}
```

P – III – 3

a) Na úvod si připomeňme, že v řešení krajského kola jste se mohli kromě jiného dočíst, jak lze pomocí dvou počítadel simulovat zásobník. Pro jistotu si zopakujeme, jak na to:

Zásobník si můžeme simulovat v jednom registru (s pomocí druhého). Písmena a, b, c budou odpovídat číslům $1, 2, 3$. Číslo uložené v registru R_1 bude představovat náš zásobník — když ho zapíšeme v poziční soustavě o základu 4, jednotlivé cifry budou představovat vložené hodnoty (cifra na místě jednotek bude naposledy vložená hodnota). Například když do prázdného zásobníku vložíme postupně písmena a, c, b, a, bude v R_1 hodnota $\mathtt{a} \times 4^3 + \mathtt{c} \times 4^2 + \mathtt{b} \times 4 + \mathtt{a} = 1 \times 4^3 + 3 \times 4^2 + 2 \times 4 + 1 = 64 + {} + 48 + 8 + 1 = 121$.

Jak ale s takovýmto registrem-zásobníkem pracovat? Vložit novou hodnotu x je jednoduché — pomocí registru R_2 vynásobíme obsah R_1 čtyřmi a potom ho x-krát zvětšíme o 1. Rovněž odebrání naposledy vložené hodnoty není těžké — je to přesně opačná operace. Vydělíme obsah registru R_1 čtyřmi. Zbytek po dělení je naposledy vložená hodnota, podíl (který dostaneme v R_2) je obsah zásobníku bez této hodnoty.

Nyní můžeme již přikročit k řešení zadané úlohy. Jednou možností je simulovat (pomocí dvou zásobníků) frontu, v níž si udržujeme ta písmena, jejichž pár jsme ještě neviděli. Toto řešení je poměrně komplikované a jeho základní myšlenka spočívá v tom, že přicházející písmena vkládáme do prvního zásobníku, písmena na kontrolu vybíráme z druhého zásobníku

a vždy, když se nám druhý zásobník vyprázdni, do něj přesypeme obsah prvního zásobníku.

Ukážeme si raději jednodušší řešení. Budeme opět používat dva zásobníky. Do prvního budeme vkládat všechna přicházející malá písmena (jako hodnoty 1, 2, 3), do druhého velká (také jako hodnoty 1, 2, 3). Po dočtení vstupního slova jednoduše porovnáme obsahy obou zásobníků. Vstupní slovo bylo správné právě tehdy, je-li jejich obsah stejný. To již snadno ověříme.

```
var vstup: char;
    i,co: byte;
begin
   Read(vstup);
   while vstup<>'$' do begin
      if vstup>='a' then begin
         if vstup='a' then co:=1;
         if vstup='b' then co:=2;
         if vstup='c' then co:=3;
         while not Zero(R1) do
            begin Dec(R1); for i:=1 to 4 do Inc(R0); end;
         while not Zero(R0) do begin Dec(R0); Inc(R1); end;
         while co>0 do begin Inc(R1); co:=co-1; end;
      end else begin
         if vstup='A' then co:=1;
         if vstup='B' then co:=2;
         if vstup='C' then co:=3;
         while not Zero(R2) do
            begin Dec(R2); for i:=1 to 4 do Inc(R0); end;
         while not Zero(R0) do begin Dec(R0); Inc(R2); end;
         while co>0 do begin Inc(R2); co:=co-1; end;
      end;
      Read(vstup);
   end;
   while not Zero(R1) and not Zero(R2) do begin Dec(R1); Dec(R2); end;
   if Zero(R1) and Zero(R2) then Accept;
end.
```

b) Pro zvýšení přehlednosti označíme původní registry R_1, R_2, R_3 a nové registry Q_1, Q_2.

Použijeme myšlenku, kterou známe již z řešení úlohy P–I–4. Zakódujeme obsah všech tří registrů do jediného, druhý registr budeme používat jako pomocný při práci s prvním. Místo tří registrů s obsahem a, b, c budeme mít tedy jeden registr Q_1 s obsahem $2^a 3^b 5^c$. Při simulování každé operace použijeme Q_2 jako pomocný registr. Na začátku i po provedení každé operace v něm bude uložena nula.

Operaci $Inc\,(R_x)$ v původním programu nahradíme tím, že obsah nového registru Q_1 vynásobíme 2, 3, resp. 5. Podobně příkaz $Dec\,(R_x)$ nahradíme příslušným dělením.

Nahradit podmínku $Zero\,(R_x)$ bude trochu komplikovanější. Během vyhodnocování nějaké složené podmínky totiž nemůžeme provádět operace s registry — zjistit, zda je v R_x nula, tedy musíme *před* vyhodnocením příslušné podmínky. Navíc drobné problémy způsobí skutečnost, že tato podmínka se může vyskytovat i v podmínce pro příkaz `while`, kde bude vyhodnocována při každé iteraci (nejen před prvním voláním `while`), a že v jedné podmínce můžeme testovat více proměnných.

Definujme si „makro“ (kus výpočtu) *SpocitejZ*, které bude fungovat následovně: Pro každý registr R_x nastaví proměnnou z_x tak, aby v ní byla kladná hodnota právě tehdy, je-li v R_x nula, jinak bude $z_x = 0$. Výpočet makra začne tím, že obsah Q_1 vydělíme příslušným prvočíslem, přičemž si (v proměnné z_x) zapamatujeme zbytek, který jsme dostali při tomto dělení. Vrátíme obsah Q_1 do původního stavu. Jestliže obsah Q_1 byl dělitelný příslušným prvočíslem (tedy neplatí $Zero\,(R_x)$), bude v z_x nula, jinak tam bude kladný zbytek. Výraz $Zero(R_x)$ má tedy v tomto okamžiku stejnou pravdivostní hodnotu jako výraz $(z_x > 0)$.

Každý příkaz „`if` P `then` *příkazy*`;`“ nahradíme makrem *SpocitejZ* a příkazem „`if` P' `then` *příkazy*`;`“, kde podmínka P' vznikla z P tak, že jsme v ní místo všech výskytů výrazu $Zero\,(R_x)$ dali výraz $(z_x > 0)$.

Každý příkaz „`while` P `do` *příkazy*`;`“ nahradíme voláním makra *SpocitejZ* před cyklem a na konci každé iterace, tedy následujícím kusem výpočtu: „*SpocitejZ*`;` `while` P' `do begin` *příkazy*`;` *SpocitejZ*`;` `end;`“

Nová „makra“ *Inc*, *Dec* (jimiž nahradíme každý výskyt těchto příkazů v původním programu) a *SpocitejZ* (na simulaci *Zero*) budou tedy vypadat následovně:

```
var x,y,z1,z2,z3,i: byte;
{nové proměnné, které nebyly v původním programu}

{ Inc (Rx) — x je v proměnné x, předpokládáme, že Q2 = 0 }
  if x=1 then y:=2 else if x=2 then y:=3 else y:=5;
  {vynásobíme obsah Q1 číslem y}
  while not Zero(Q1) do begin
     Dec(Q1);
     for i:=1 to y do Inc(Q2);
  end;
  while not Zero(Q2) do begin
     Dec(Q2); Inc(Q1);
  end;

{ Dec (Rx) — x je v proměnné x, předpokládáme, že Q2 = 0 }
  if x=1 then y:=2 else if x=2 then y:=3 else y:=5;
  z:=0;
  {vydělíme obsah Q1 číslem y}
  while not Zero(Q1) do begin
```

```
      Dec(Q1);
      if Zero(Q1) then begin z:=1; break; end; {v Rx byla nula}
      for i:=1 to y-1 do Dec(Q1);
      Inc(Q2);
   end;
   if z=1 then begin
      {obnovíme původní stav Q1 - nic se nemění}
      while not Zero(Q2) do begin
         Dec(Q2); for i:=1 to y do Inc(Q1);
      end;
      Inc(Q1);
   end else begin
      {přesuneme do Q1 podíl}
      while not Zero(Q2) do begin
         Dec(Q2); Inc(Q1);
      end;
   end;

{ SpocitejZ — předpokládáme, že Q2 = 0 }
   {nejdříve chceme spočítat z1, čili budeme dělit dvěma}
   y:=2;
   {vydělíme obsah Q1 číslem y}
   z1:=0;
   while not Zero(Q1) do begin
      Dec(Q1);
      z1:=z1+1;
      if z1=y then begin z1:=0; Inc(Q2); end;
   end;
   {vrátíme zpět původní hodnotu do Q1}
   while not Zero(Q2) do begin
      Dec(Q2); for i:=1 to y do Inc(Q1);
   end;
   for i:=1 to z1 do Inc(Q1);
   {a v proměnné z1 máme hledaný zbytek}
   {opakujeme totéž pro z2, y:=3 a z3, y:=5}
```

Dva důležité detaily, kterých jste si mohli povšimnout:

1. Nesmíme zapomenout ošetřit situaci, že registr R_i obsahuje nulu. V tom případě i po provedení *Dec* (R_i) musí v R_i (podle definice) zůstat nula.
2. Když jsme při simulování příkazů *Inc*, *Dec* a *Zero* potřebovali použít proměnné, muselo se jednat o *nové* proměnné, které se dosud v programu nevyskytovaly. (Co kdyby například původní program obsahoval část „`for i:=1 to 3 do` *Inc* (R_1)“ a my bychom při simulaci *Inc* (R_1) použili proměnnou `i`?) Volných jmen pro nové proměnné máme k dispozici nekonečně mnoho, lehce tedy najdeme nějaké nepoužité.

Snadno nahlédneme, že když tímto způsobem upravíme libovolný program, bude upravený program ekvivalentní s původním — tj. bude dá-

vat pro každý vstup stejný výstup jako původní program. Přitom pokud původní program používal tři registry, upravený program už používá jen dva.

Uvědomte si, že aplikováním tohoto postupu na program používající $k > 3$ registrů dostaneme program používající $k - 1$ registrů. Proto platí poměrně překvapivý výsledek: K libovolné úloze, kterou dokážeme řešit na registrovém počítači, existuje program, jemuž na její řešení stačí dva registry.

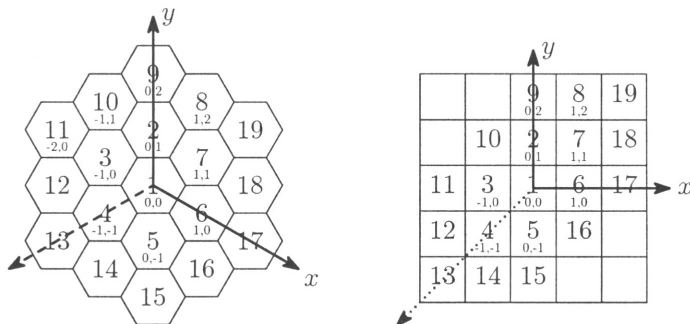
P – III – 4

Na poskakování psíků se můžeme dívat jako na hru. Stav hry lze jednoznačně popsat pozicí obou psíků. Skok psíků představuje tah. Když oba psíci skočí, změní stav hry. Povolené stavy hry budou ty, které odpovídají povoleným pozicím psíků. Pro každý stav hry budeme zkoumat, kolika tahy se do něho dá dostat z počátečního stavu (když jsou oba psíci na výchozích místech). Tento počet tahů budeme označovat jako vzdálenost daného stavu.

Vzdálenost počátečního stavu je 0. Všechny stavy, do nichž se lze z něho dostat jedním tahem, budou ve vzdálenosti 1. Nyní projdeme všechny stavy ve vzdálenosti 1 a hledáme, do kterých nových stavů se z nich dostaneme — ty budou zjevně ve vzdálenosti 2. Takto můžeme analogicky pokračovat pro stavy ve vzdálenosti $3, 4, \ldots$ Skončíme, když najdeme koncový stav (oba psíci jsou na svých cílových místech), nebo když už nenajdeme žádný nový stav. Tato technika prohledávání stavů se nazývá prohledávání do šířky.

Otázkou zůstává, jak pro každý stav určit, do kterých dalších (sousedních) stavů se z něho lze dostat jedním tahem. Pomohlo by nám, kdybychom uměli pro každé políčko na louce určit čísla jeho sousedů. Sousední stavy bychom potom určili snadno. Ze všech možných pohybů oběma psíky 6 směry (36 možností) vyškrtáme skákání stejným směrem, skákání na bodláky, skok některého psíka mimo louku a současný skok obou psíků na totéž políčko.

Abychom našli sousední políčka snadněji, ukážeme si, jak se dá šestiúhelníkový plán louky reprezentovat v obyčejném dvojrozměrném poli. Na políčku 1 si zvolíme dva směry. Jeden určuje rostoucí směr první souřadnice, druhý směr druhé souřadnice. Takto jsme přiřadili každému políčku souřadnice x, y, kterým odpovídají indexy v obyčejném dvojrozměrném poli. V dvojrozměrném poli je už nalezení sousedů lehké. Konkrétně



při naší volbě souřadnicových os budou mít sousedi políčka (x, y) souřadnice $(x, y+1)$, $(x-1, y)$, $(x-1, y-1)$, $(x, y-1)$, $(x+1, y)$ a $(x+1, y+1)$.

Jak zjistíme pro políčko s číslem k jeho souřadnice? Všimněte si, že spirálu můžeme rozložit na vrstvy šestiúhelníkového tvaru. Nejprve určíme, na kolikáté vrstvě spirály se k nachází, potom stranu na této vrstvě, pozici políčka na straně a je to.

Nultá vrstva spirály obsahuje 1 políčko, i-tá vrstva pak $6i$, jelikož každá vrstva má 6 stran a na každé straně je i políček. Celkový počet políček ve spirálách $0 \dots v$ je tedy $1 + \sum_{i=1}^{v} 6i = 3v^2 + 3v + 1$.

A jak zjistíme, kde se nachází políčko k? Nejdříve spočteme vrstvu — najdeme kladné řešení rovnice $k = 3v^2+3v+1$ a zaokrouhlíme ho nahoru. Po vyřešení dostaneme $v = \left\lceil -\frac{1}{2} + \sqrt{\frac{1}{3}k - \frac{1}{12}} \right\rceil$. (Jednodušší a trochu pomalejší postup: zvyšujeme v, dokud počet políček nedosáhne k.)

Když už víme vrstvu v, kde se políčko nachází, pořadové číslo políčka ve vrstvě (číslováno od 0) dopočítáme snadno: odečteme od k celkový počet políček na dřívějších vrstvách a ještě 1. Na vrstvě v má každá strana v políček, rozdíl tedy stačí vydělit v. Číslo strany s bude podíl, pozice na straně p bude zbytek po tomto dělení.

Již umíme pro dané políčko k spočítat jeho vrstvu v, stranu s a pozici na straně p. Z těchto hodnot dostaneme souřadnice podle následující tabulky (platí pro $v \geqq 1$):

s	x	y
0	$+v-1-p$	$+v$
1	$-1-p$	$+v-1-p$
2	$-v$	$-1-p$
3	$-v+1+p$	$-v$
4	$+1+p$	$-v+1+p$
5	$+v$	$+1+p$

Implementace: Stav hry je jednoznačně reprezentován souřadnicemi x_1, y_1, x_2, y_2 obou psíků. Při prohledávání do šířky si pamatujeme seznam stavů, které jsme již dosáhli, ale ještě jsme z nich nezkoušeli prohledávat nové vrcholy. K tomu nám poslouží fronta. Dále potřebujeme umět pro každý stav rychle zjistit, zda jsme ho ještě neviděli, už viděli, nebo zda se do něj nedá jít (bodláky). K tomu používáme čtyřrozměrné pole, kde je to přímo zapsáno. (Přesně totéž se dalo reprezentovat dvojrozměrným polem, které bychom indexovali původními souřadnicemi. Navíc bychom ale potřebovali umět ze souřadnic určit původní číslo políčka.)

Abychom nemuseli při prohledávání do šířky stále kontrolovat, zda se nedostaneme ven z louky, postavíme okolo celé louky bodláky. Zakážeme také stavy, v nichž by byli oba psíci na stejném místě.

Prohledáváme prostor velikosti řádově $O(N^2)$, kde N je velikost louky. Časová i paměťová složitost prohledávání je lineární vzhledem k velikosti tohoto prostoru, tedy $O(N^2)$.

```
program Psici;
const
   EPS = 1.0E-6;     {max. chyba vzniklá v reálných číslech}
   MAX_V = 16;       {největší možná vrstva (i se zarážkou)}
   MAX_STAVU = MAX_V*MAX_V*MAX_V*MAX_V;
   INF = 299999;     {největší možný počet skoků}
   MAX_SMER = 6;     {počet směrů, jimiž se mohou psíci hýbat}

type
   TSour =  record x,y : integer; end; {souřadnice jednoho psíka}
   TStav = record p1, p2 : TSour; end; {souřadnice dvou psíků}
   {prostor pro 2 psíky: [x1,y1,x2,y2] říká, zda tam mohou být}
   TMrizka = array [-MAX_V..MAX_V,-MAX_V..MAX_V,
                    -MAX_V..MAX_V,-MAX_V..MAX_V] of integer;

function dekVrstvu(k: integer): integer; {číslo vrstvy, kde se k nachází}
begin
   dekVrstvu:= trunc(0.5 + sqrt(k/3.0 - 1.0/12.0 - EPS) );
end;

function zakVrstvu(v: integer): integer; {poslední prvek na dané vrstvě}
begin
   zakVrstvu:= 3*v*v - 3*v +1;
end;

function dekoduj(k: integer): TSour; {Dekóduj číslo políčka na souřadnice}
var v, pv, s, ps: integer;
    sour: TSour;
begin
   if k=1 then begin  {pro k=1 (vrstva 0) naše vzorce nefungují}
      sour.x:= 0;
      sour.y:= 0;
   end else begin
```

```
      v:= dekVrstvu(k);
      pv:= k - (3*v*v - 3*v + 1); {pozice ve vrstvě}
      s:= pv div v; {strana šestiúhelníka, kde se pv nachází}
      ps:= pv mod v; {pozice na straně šestiúhelníka}

      case s of
      0: begin sour.x:= +v-1-ps; sour.y:= +v     ; end;
      1: begin sour.x:=   -1-ps; sour.y:= +v-1-ps; end;
      2: begin sour.x:= -v     ; sour.y:=   -1-ps; end;
      3: begin sour.x:= -v+1+ps; sour.y:= -v     ; end;
      4: begin sour.x:=   +1+ps; sour.y:= -v+1+ps; end;
      5: begin sour.x:= +v     ; sour.y:=   +1+ps; end;
      else writeln('Velká chyba v dekoduj');
      end;
   end;
   dekoduj:=sour;
end;

var
   n, m, s1, t1, s2, t2: integer; {vstup}
   v: integer; {max. použitá vrstva pro dané n}
   A: TMrizka; {prohledávaný prostor}
   F: array [0..MAX_STAVU] of TStav; {fronta}

{zakáže všechny situace, kde se nachází bodlák na daném místě}
procedure pridejBodlak(b: TSour);
var x, y: integer;
begin
   for x:= -v to v do for y:= -v to v do begin
      A[x, y, b.x, b.y]:= INF;        {2. psík stojí na bodláku}
      A[b.x, b.y, x, y]:= INF;        {1. psík stojí na bodláku}
   end;
end;

{vyčistí celý prohledávaný prostor}
procedure inicializace;
var x1, y1, x2, y2, i, last: integer;
begin
   {vyčištění prostoru}
   for x1:= -v to v do for y1:= -v to v do
      for x2:= -v to v do for y2:= -v to v do
         A[x1, y1, x2, y2]:= -1;
   {zarážky při okrajích - přidáme umělé bodláky}
   last:= zakVrstvu(v+1);
   for i:=n+1 to last do pridejBodlak(dekoduj(i));
   {zakážeme být oběma psíkům na stejném místě}
   for x1:= -v to v do for y1:= -v to v do A[x1, y1, x1, y1]:= INF;
end;

{posunutí souřadnice daným směrem}
function pohyb(a: TSour; s: integer): TSour;
const smer: array [1..MAX_SMER] of TSour = (
   (x: 0; y: 1), (x:-1; y: 0), (x:-1; y:-1),
   (x: 0; y:-1), (x: 1; y: 0), (x: 1; y: 1) );
begin
```

```
   a.x:= a.x+smer[s].x; a.y:= a.y+smer[s].y; pohyb:=a;
end;

function pracuj: integer;
{prohledávání prostoru, kde se mohou psíci nacházet}
var
   zac: integer;    {pozice prvního prvku ve frontě}
   kon: integer;    {pozice za posledním prvkem ve frontě = volné místo}
   i, j, vzd: integer;
   p1, p2, q1, q2: TSour;
   pt1, pt2: TSour; {dekódované pozice skrýší pro psíky}
begin
   zac:=0; kon:=1; {přidáme počátek do fronty}
   p1:= dekoduj(s1);
   p2:= dekoduj(s2);
   F[zac].p1:= p1; F[zac].p2:= p2;
   A[p1.x, p1.y, p2.x, p2.y]:= 0; {začínáme ve vzdálenosti 0}

   pt1:=dekoduj(t1); pt2:=dekoduj(t2);

   while zac<>kon do begin
      {vybereme stav z fronty a najdeme k němu vzdálenost}
      p1:=F[zac].p1; p2:=F[zac].p2;
      vzd:= A[p1.x, p1.y, p2.x, p2.y];
      inc(zac);

      {zkoušíme všechny kombinace směrů, kam mohou skákat}
      for i:=1 to MAX_SMER do for j:=1 to MAX_SMER do begin
        if i=j then continue; {nemohou skákat stejně}

        {nové pozice psíků}
        q1:= pohyb(p1, i); q2:= pohyb(p2, j);

        {již navštívená pozice resp. zarážka ?}
        if A[q1.x, q1.y, q2.x, q2.y] >= 0 then continue;

        {nově objevený stav -> přidáme do fronty}
        A[q1.x, q1.y, q2.x, q2.y]:= vzd+1;
        F[kon].p1:=q1; F[kon].p2:=q2;
        inc(kon);

        {našli jsme koncový stav?}
        if (pt1.x=q1.x) and (pt1.y=q1.y) and (pt2.x=q2.x) and (pt2.y=q2.y)
           then begin pracuj:=vzd+1; exit; end;
      end;
   end;
   pracuj:= -1;
end;

var x, i: integer;
begin
   while true do begin
      read(n, m);
      if (n=0) and (m=0) then break;
```

```
      v:= dekVrstvu(n);
      inicializace();

      read(s1, t1, s2, t2);
      for i:=1 to m do begin read(x); pridejBodlak(dekoduj(x)); end;
      if (s1=t1) and (s2=t2) then x:=0 {speciální případ} else x:=pracuj;
      if x>=0 then writeln(x) else writeln('nelze');
   end;
end.
```

P – III – 5

Označme $S = \sum_{i=1}^{N} l_i$ počet dní, které AttoSoft potřebuje na dokončení všech programů. Poslední program tedy dokončíme po S dnech.

Pro každý program spočítáme pokutu, kterou bychom za něj zaplatili, kdybychom ho dokončili až po S dnech. Program s nejmenší takovou pokutou zařadíme do rozvrhu jako poslední. Je-li to program číslo i, zbývá nám naplánovat všechny zbývající programy na prvních $S - l_i$ dní, což provedeme stejným způsobem (tzn. opět vybereme jako poslední program s nejnižší pokutou po $S - l_i$ dnech, atd.)

Správnost uvedeného algoritmu dokážeme indukcí vzhledem k počtu programů, které potřebujeme dokončit. Pokud je třeba dokončit jeden program, existuje jen jediný možný rozvrh, a náš algoritmus tedy funguje jistě správně.

Nechť tedy počet programů, které je třeba dokončit, je N a nechť pro libovolný menší počet programů náš algoritmus funguje správně. Označme G řešení získané naším algoritmem (v tomto řešení je posledním programem program číslo i). Nechť existuje jiné, levnější řešení O, které končí programem číslo j. Jestliže $i = j$, pak rozdíl mezi G a O musí být v pořadí prvních $N - 1$ programů. Podle indukčního předpokladu však toto pořadí v řešení G je optimální, proto řešení O nemůže být levnější.

V opačném případě vytvoříme nový rozvrh O' následujícím způsobem. Nechť rozvrh O dokončuje programy v pořadí $o_1, o_2, \ldots, o_N$ a nechť $o_k = i$. Podle rozvrhu O' dokončíme programy v následujícím pořadí: $o_1, o_2, \ldots, o_{k-1}, o_{k+1}, \ldots, o_N, i$. Všimněte si, že řešení O' je nejvýše tak drahé, jako řešení O. Pokuta za programy $o_{k+1}, \ldots, o_N$ je totiž nižší než v řešení O, neboť je dokončíme dříve (pokuta roste s počtem dní po termínu). Navíc pokuta za program i dokončený po S dnech určitě nepřesahuje pokutu za program o_N dokončený po S dnech, jelikož program i jsme vybrali tak, aby tato pokuta byla nejmenší možná.

Řešení O' ale nemůže být levnější než řešení G (platí tu stejný argument, jako v předchozím případě). Proto ani řešení O nemůže být levnější než G. Dokázali jsme, že žádné levnější řešení než G neexistuje, řešení určené naším algoritmem je tedy optimální.

V každém kroku algoritmu musíme spočítat příslušnou pokutu pro každý program, který jsme dosud nezařadili do rozvrhu. Proto časová složitost algoritmu je $O(N^2)$.

Při výpočtu si ještě musíme dávat pozor na to, že pokuta může být až $5\,000 \cdot 100\,000^3 + 5\,000 \cdot 100\,000^2 + \ldots + 5\,000$, tedy přibližně $5 \cdot 10^{18}$, a tak velké číslo se již nevejde do *longint*u a nemůžeme si dovolit použít ani typ *real*, jelikož potřebujeme i u tak velkých čísel rozlišovat rozdíly na řádu jednotek. Většina překladačů naštěstí nabízí 64bitový celočíselný typ — v Turbo Pascalu je to typ *comp*, ve Free Pascalu například typ *QWord*.

```
program Attosoft;

const MAXN = 10000;

var a,b,c,d,l: array [1..MAXN] of longint;
    pouzite: array [1..MAXN] of boolean;
    N,S: longint;

function Cena(prog:integer; den: comp): comp;
begin
   cena:=((a[prog]*den+b[prog])*den+c[prog])*den+d[prog];
end; {function Cena}

procedure Nacti;
var i: integer;
begin
   readln(N);
   S:=0;
   for i:=1 to N do begin
      readln(l[i],a[i],b[i],c[i],d[i]);
      S:=S+l[i];
   end;
end; {procedure Nacti}

procedure Spocitej_rozvrh(d: longint);
var minprog: integer;
    min,cc: comp;
    i: integer;
begin
   {najdi nepoužitý program, který má nejnižší pokutu po d dnech}
   minprog:=-1;
   for i:=1 to N do begin
      if not pouzite[i] then begin
         cc:=Cena(i,d);
         if (minprog=-1) or (cc<min) then begin
```

```
            min:=cc;
            minprog:=i;
         end;
      end;
   end;

   if minprog>-1 then begin
      {označ program jako použitý}
      pouzite[minprog]:=true;
      {sestav zbytek rozvrhu}
      Spocitej_rozvrh(d-l[minprog]);
      {vypiš poslední program na konci rozvrhu}
      writeln(minprog);
   end;
end; {procedure Spocitej_rozvrh}

var i:integer;

begin
   Nacti;
   for i:=1 to N do pouzite[i]:=false;
   Spocitej_rozvrh(S);
end.
```