49. ročník matematické olympiády na středních školách

Kategorie P

In: Leo Boček (editor); Karel Horák (editor); Jaromír Šimša (editor); Jaroslav Švrček (editor); Pavel Töpfer (editor): 49. ročník matematické olympiády na středních školách. Zpráva o řešení úloh ze soutěže konané ve školním roce 1999/2000. 41. mezinárodní matematická olympiáda. 12. mezinárodní olympiáda v informatice. (Czech). Praha: Jednota českých matematiků a fyziků, 2005. pp. 101–146.

Persistent URL: http://dml.cz/dmlcz/405016

Kategorie P

Texty úloh

P – I – 1

V továrně bude zahájena výroba nového typu výrobku. Výroba je popsána přesným technologickým plánem, jenž stanoví, jaké všechny výrobní operace je třeba vykonat a jak dlouho každá z těchto operací trvá. Pro každou výrobní operaci P je dále znám seznam operací, které musí být provedeny před začátkem operace P (nazveme je *předcházející operace*).

Majitelé továrny chtějí výrobu zorganizovat tak, aby bylo možné dokončit první nový výrobek v co nejkratším čase. V továrně lze současně provádět libovolný počet výrobních operací, každou operaci je však možné zahájit až po dokončení všech jí předcházejících operací. Jinak mohou být jednotlivé výrobní operace prováděny v jakémkoliv pořadí. Můžete předpokládat, že výroba podle zadaného technologického plánu je možná.

Napište program, který ze vstupního textového souboru `TOVARNA.IN` přečte technologický plán výroby nového výrobku a do výstupního textového souboru `TOVARNA.OUT` vypíše nejkratší čas, v jakém lze vyrobit první nový výrobek.

Soubor `TOVARNA.IN` obsahuje na prvním řádku počet výrobních operací N ($N \leqq 100$). Na dalších N řádcích se nacházejí informace o jednotlivých výrobních operacích, na i-tém řádku o operaci číslo i. Informace o každé výrobní operaci se skládá z několika čísel oddělených mezerou. První číslo udává, jak dlouho operace trvá, dále následují čísla předcházejících operací. Řádek je ukončen číslem -1. Pro zhotovení nového výrobku je třeba vykonat všech N zadaných výrobních operací. Jednotlivé operace je možné provádět v libovolném pořadí, každá operace však může být zahájena až po dokončení všech jí předcházejících operací.

Soubor TOVARNA.OUT bude obsahovat jediný řádek, na kterém bude zapsán čas, za jak dlouho od zahájení výroby lze dokončit první nový výrobek.

Příklad: Soubor TOVARNA.IN

```
4
2 3 -1
3 3 -1
3 -1
1 1 2 -1
```

Soubor TOVARNA.OUT

```
7
```

P – I – 2

Je dána množina $A = \{a_1, a_2, a_3, \ldots, a_N\}$ $(N \geqq 1)$, kterou budeme nazývat *abeceda*. Prvky abecedy budeme nazývat *znaky*. *Řetězcem* nad abecedou A je konečná posloupnost prvků z množiny A (znaků). *Vybraný podřetězec* vznikne z řetězce vynecháním některých jeho znaků, přičemž pořadí zbývajících znaků řetězce zachováme beze změn. *Permutace* prvků množiny A je řetězec, v němž se každý prvek množiny A vyskytuje právě jednou.

Příklad: Nechť $N = 3$, $A = \{a, b, c\}$. Potom z řetězce *abcabccb* můžeme vytvořit například vybrané podřetězce *ab*, *aac*, *bbcb*, *abbb*, *abc*, *cab*. Z nich pouze *abc* a *cab* jsou permutace množiny A.

Soutěžní úloha. Nalezněte co nejkratší řetězec nad abecedou $A = \{a, b, c, \ldots, o\}$ $(N = 15)$, který obsahuje všechny permutace prvků množiny A jako vybrané podřetězce, přičemž každý znak se v něm vyskytuje nejvýše N-krát. Podrobně popište také způsob, jak jste tento řetězec našli, a přiložte a popište všechny algoritmy a programy, které jste přitom použili. Výsledný řetězec uložte do souboru RETEZEC.TXT.

Příklad: Například pro abecedu $A = \{a, b, c\}$ $(N = 3)$ vyhovuje řetězec *acbacab*, protože se v něm nacházejí všechny permutace *abc*, *acb*, *bac*, *bca*, *cab*, *cba* jako vybrané podřetězce. Zároveň je tento řetězec nejkratším řetězcem nad abecedou A splňujícím tuto podmínku.

P – I – 3

V jisté zemi existuje N politických stran. Každá ze stran má mezi obyvatelstvem určité preference, přičemž žádné dvě strany nemají stejné preference. Agentury pro výzkum veřejného mínění dokážou provést průzkum, který pro libovolné dvě politické strany umožňuje přesně zjistit, která

z nich má preference nižší a která vyšší. Tento průzkum je však poměrně drahý a dá se použít vždy jen pro dvě vybrané politické strany.

Jedna agentura chce zjistit, která ze stran v zemi má nejvyšší a která nejnižší preference. Bude postupovat tak, že pro různé dvojice stran provede výše popsaný průzkum veřejného mínění. Chce přitom vykonat co nejméně průzkumů.

Napište program, který bude řídit činnost agentury. Tento program nejprve přečte počet stran N ($1 \leqq N \leqq 100$). Strany si pro jednoduchost očíslujeme čísly $1, 2 \ldots, N$. Program potom v cyklu vždy vypíše, pro které dvě strany se má provést průzkum, a následně přečte ze vstupu výsledek tohoto průzkumu — číslo strany s většími preferencemi. Když program tímto způsobem získá dostatek údajů potřebných k tomu, aby určil politické strany s nejvyššími a s nejnižšími preferencemi, vypíše odpověď a skončí.

Dbejte zejména na to, aby program dal vždy správnou odpověď a aby použil co nejméně průzkumů. Odhadněte, kolik nejvíce průzkumů může váš program potřebovat pro N stran.

Příklad: Uvedeme příklad činnosti programu pro tři strany, přičemž texty vypisované programem jsou od začátku řádku a odpovědi uživatele jsou zadány v řádcích začínajících znakem >.

```
Zadej počet stran:
> 3
Proveď průzkum 1,2
> 2
Proveď průzkum 1,3
> 3
Proveď průzkum 2,3
> 2
Nejvyšší preference má strana 2, nejnižší 1
```

P – I – 4

Minského registrový stroj

Minského registrový stroj je jednoduché výpočetní zařízení. K dispozici má několik registrů označených $R_0, R_1, R_2, \ldots$, přičemž v každém registru může být uloženo jedno libovolně velké nezáporné celé číslo.

Minského registrový stroj může mít jeden nebo více vstupů, jejichž hodnoty jsou na začátku výpočtu uloženy v registrech $R_1, \ldots, R_k$, kde k je počet těchto vstupů. Ostatní registry jsou na začátku výpočtu inicia-

lizovány na hodnotu nula. Po skončení výpočtu Minského registrového stroje je výsledkem hodnota uložená v registru R_0.

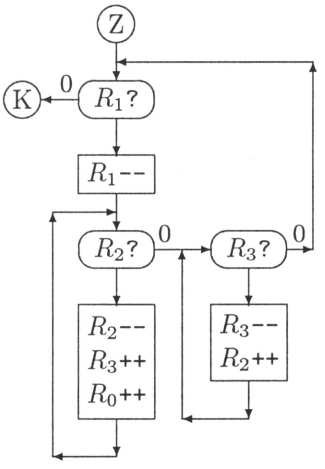
Každý Minského registrový stroj je řízen pevně daným programem. V programu se mohou vyskytovat tyto příkazy:

- ▷ *Zvýšení* hodnoty v daném registru o 1.
- ▷ *Snížení* hodnoty v daném registru o 1 (pokud je v registru nula, hodnota se nezmění).
- ▷ *Test na nulu* zjistí, zda je v daném registru nula.

Registr je určen svým číslem. Číslo registru je v příkazu vždy pevně zadáno, není tedy možné k určení registru použít obsah jiného registru.

Programy budeme zakreslovat do schémat, přičemž zvýšení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i++, snížení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i--, test na nulu v registru R_i značíme oválem s nápisem R_i?. Začátek programu označujeme písmenem Z v kroužku a konec programu písmenem K v kroužku (program může mít i více konců, ale jen jeden začátek). Jednotlivé příkazy jsou pospojovány šipkami, které určují tok řízení programu. Z obdélníku vychází vždy jediná šipka. Z oválu vycházejí dvě šipky, přičemž jedna z nich je označena nulou (po ní pokračuje výpočet tehdy, když byla v testovaném registru nula, v opačném případě výpočet sleduje druhou šipku). Jestliže v programu za sebou následuje několik příkazů zvýšení nebo snížení, můžeme je napsat pod sebe do jednoho obdélníku.

Příklad: Sestrojte stroj, který bude mít na vstupu čísla x a y (uložená v registrech R_1 a R_2) a který vypočítá jejich součin $x \cdot y$ (uloží ho do registru R_0).



ŘEŠENÍ. Stroj řešící tuto úlohu vidíme na obrázku. Tento stroj vždy odečte od registru R_1 jedničku a k registru R_0 přičte číslo y (obsah registru R_2). To provádí tak dlouho, dokud v registru R_1 není nula. Číslo y se tedy do registru R_0 přičte celkem x-krát, takže dostaneme správný výsledek.

Podívejme se nyní na to, jak se provádí přičtení čísla y k R_0. Opět v cyklu snižujeme hodnotu registru R_2 a zvyšujeme hodnotu registru R_0, dokud nemáme v registru R_2 nulu. Tehdy jsme

do R_0 přidali přesně y. Problém však spočívá v tom, že v registru R_2 máme nyní nulu, ale pro další průběh výpočtu tam potřebujeme vložit zpět hodnotu y. Proto v cyklu kromě snížení R_2 a zvýšení R_0 ještě zvýšíme R_3. Tím si zajistíme, že po skončení cyklu máme hodnotu y uloženou v registru R_3. Nyní ji musíme odtamtud přenést zpět do registru R_2. To provedeme v dalším cyklu, v němž současně snižujeme R_3 a zvyšujeme R_2, dokud v R_3 není nula. V tom okamžiku máme v R_2 opět původní hodnotu y a můžeme začít s dalším kolem výpočtu.

Soutěžní úlohy:

a) Sestrojte Minského registrový stroj s jedním vstupem n, který vypočítá číslo 2^n.

b) Sestrojte Minského registrový stroj s jedním vstupem n, který vypočítá Fibonacciho číslo F_n. Váš stroj přitom může používat pouze registry R_0, R_1, R_2, R_3.
Fibonacciho čísla jsou definována následujícím vztahem:

$$F_n = \begin{cases} 1, & \text{jestliže } n < 2, \\ F_{n-1} + F_{n-2}, & \text{jestliže } n \geqq 2. \end{cases}$$

P – II – 1

Síť

Ve výpočetním středisku mají N počítačů očíslovaných od 1 do N. Počítače jsou navzájem propojeny jednosměrnými spoji. Jestliže je „počítač i připojen k počítači j“, znamená to, že počítač i může posílat zprávy počítači j (ale ne naopak). Jestliže je počítač i připojen k počítači j, může, ale nemusí být připojen také počítač j k počítači i. (Žádný počítač není připojen sám k sobě.)

Do výpočetního střediska nyní koupili nový centrální počítač. Je třeba zajistit, aby tento nový počítač mohl poslat zprávu všem ostatním počítačům. Musíme ho proto připojit k některým dalším počítačům tak, aby se z něj dala poslat zpráva libovolnému počítači p buď přímo (tj. centrální počítač je připojen k počítači p), nebo přes několik jiných počítačů (tj. existují počítače $a_1, a_2, \ldots, a_k$ takové, že centrální počítač je připojen k počítači a_1, pro $i = 1, 2, \ldots k-1$ je počítač a_i připojen k počítači a_{i+1} a počítač a_k je připojen k počítači p). Kvůli úspoře kabelů je přitom nutné minimalizovat počet počítačů, ke kterým bude nový centrální počítač připojen.

Soutěžní úloha. Je dán počet počítačů N. Pro každý počítač i ($1 \leqq i \leqq N$) je dán počet počítačů d_i, k nimž je daný počítač připojen, a dále je dán seznam těchto počítačů $a_{i1}, a_{i2}, \ldots a_{id_i}$. Napište program, který vypíše seznam počítačů, k nimž je třeba připojit nový centrální počítač tak, aby jejich počet byl minimální. Pokud je možné centrální počítač připojit více způsoby, vypište libovolný jeden z nich.

Příklad: *vstup*:	*výstup*:
$N = 7$	2,4,7
$d_1 = 1$, připojení: 2	(Jiné možné řešení je 1,4,7.)
$d_2 = 2$, připojení: 1 5	
$d_3 = 1$, připojení: 6	
$d_4 = 2$, připojení: 3 6	
$d_5 = 1$, připojení: 3	
$d_6 = 1$, připojení: 5	
$d_7 = 0$, připojení: $\emptyset$	

P – II – 2

Posloupnost

Nechť $A = (a_1, a_2, \ldots, a_M)$ je posloupnost celých čísel. Posloupnost A' nazveme *vybranou podposloupností* této posloupnosti, jestliže vznikne z posloupnosti A vynecháním některých jejích členů, přičemž pořadí ostatních prvků zachováme. Společná vybraná podposloupnost posloupností A, B je každá taková posloupnost C, která je vybranou podposloupností obou posloupností A i B.

Příklad: Nechť $A = (1, 11, 2, 1, 4, 99), B = (9, 4, 1, 2, 7, 1, 99)$. Posloupnost $(1, 2, 1, 99)$ je společnou vybranou podposloupností posloupností A, B. Posloupnost $(1, 2, 4)$ je vybranou podposloupností posloupnosti A, ale není vybranou podposloupností posloupnosti B, takže není ani společnou vybranou podposloupností A a B.

Soutěžní úloha. Máme dány dvě posloupnosti celých čísel $A = (a_1, a_2, \ldots, a_M)$ a $B = (b_1, b_2, \ldots, b_N)$ s délkami M, resp. N. Tyto posloupnosti jsou uloženy v polích `a[1..M]`, resp. `b[1..N]`, jejichž obsah není dovoleno měnit. Uvažujme takovou společnou vybranou podposloupnost posloupností A a B, ve které součet všech jejích členů je největší možný. Napište program, který vypíše součet členů takovéto posloupnosti.

Poznámka. Existuje algoritmus, který tuto úlohu řeší v čase úměrném $M \cdot N$ a paměti, jejíž velikost je úměrná menšímu z čísel M, N.

Příklad:

vstup:	*výstup*:
$M = 6, N = 7$	103
$A = (1, 11, 2, 1, 4, 99)$	(Vybrané podposloupnosti se součtem 103
$B = (9, 4, 1, 2, 7, 1, 99)$	existují dvě: $4, 99$ a $1, 2, 1, 99$.)

P – II – 3

Volby

V jisté zemi právě skončily volby. Každý volič v nich hlasoval pro jednoho z navržených kandidátů. Vítězem voleb se stane kandidát, pro kterého hlasovala nadpoloviční většina voličů. Máte k dispozici výsledky hlasování jednotlivých voličů a máte co nejrychleji zjistit, který kandidát vyhrál volby.

Soutěžní úloha. Voleb se zúčastnilo N voličů očíslovaných od 1 do N a M kandidátů očíslovaných od 1 do M (někteří kandidáti však nemuseli získat ani jeden hlas). Výsledky hlasování jsou uloženy v poli a, přičemž platí, že volič i ($1 \leqq i \leqq N$) hlasoval pro kandidáta číslo a_i. Obsah tohoto pole nesmíte měnit.

Napište program, který zjistí, zda existuje kandidát, pro kterého hlasovalo více než $\frac{1}{2}N$ voličů. Jestliže takový kandidát existuje, program vypíše jeho číslo, pokud ne, program vypíše, že takový kandidát neexistuje.

Poznámka. Čísla M a N mohou být *velmi* velká. Snažte se proto, aby váš program pracoval co nejefektivněji a aby používal co nejméně paměti.

Příklad:

vstup:	*výstup*:
$N = 9$	Vyhrál kandidát 2.
$A = \{2, 3, 2, 2, 3, 50\,001, 3, 2, 2\}$	
vstup:	*výstup*:
$N = 10$	Nevyhrál žádný kandidát.
$A = \{2, 2, 1, 4, 3, 2, 1, 2, 100, 2\}$	

P – II – 4

Minského registrové stroje

(Oproti domácímu kolu je popis registrového stroje rozšířen o nový druh příkazu — výpočet hodnoty funkce.)

Minského registrový stroj je jednoduché výpočetní zařízení. K dispozici má několik registrů označených $R_0, R_1, R_2, \ldots$, přičemž v každém registru může být uloženo jedno libovolně velké nezáporné celé číslo.

Minského registrový stroj může mít jeden nebo více vstupů, jejichž hodnoty jsou na začátku výpočtu uloženy v registrech $R_1, \ldots, R_k$, kde k je počet těchto vstupů. Ostatní registry jsou na začátku výpočtu inicializovány na hodnotu nula. Po skončení výpočtu Minského registrového stroje je výsledkem hodnota uložená v registru R_0.

Každý Minského registrový stroj je řízen pevně daným programem. V programu se mohou vyskytovat tyto příkazy:

- ▷ *Zvýšení* hodnoty v daném registru o 1.
- ▷ *Snížení* hodnoty v daném registru o 1 (pokud je v registru nula, hodnota se nezmění).
- ▷ *Test na nulu* zjistí, zda je v daném registru nula.
- ▷ *Výpočet hodnoty funkce F.* Tento příkaz vezme obsah určeného vstupního registru (označme tuto hodnotu registru jako x), vypočítá hodnotu $F(x)$ a uloží ji do určeného výstupního registru. Obsah vstupního registru po provedení tohoto příkazu není definován (může v něm být libovolné nezáporné číslo).

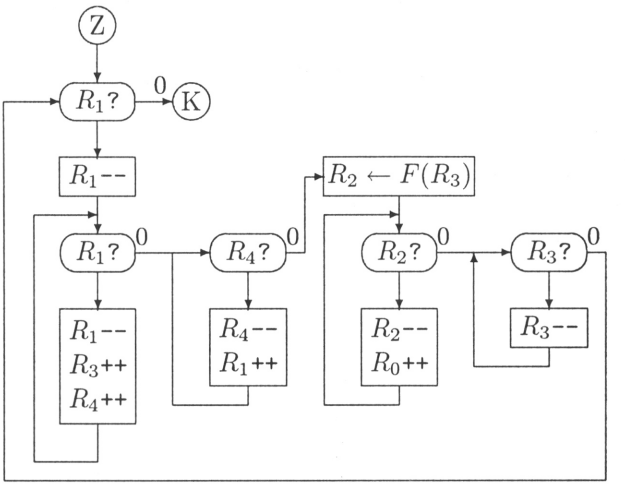
Registr je určen svým číslem. Číslo registru je v příkazu vždy pevně zadáno, není tedy možné k určení registru použít obsah jiného registru.

Programy budeme zakreslovat do schémat, přičemž zvýšení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i++, snížení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i--, test na nulu v registru R_i značíme oválem s nápisem R_i?. Příkaz na výpočet funkce F se vstupním registrem R_i a výstupním registrem R_j značíme obdélníkem s nápisem $R_j \leftarrow F(R_i)$ (vstupní a výstupní registr musí být navzájem různé).

Začátek programu označujeme písmenem Z v kroužku a konec programu písmenem K v kroužku (program může mít i více konců, ale jen jeden začátek). Jednotlivé příkazy jsou pospojovány šipkami, které určují tok řízení programu. Z obdélníku vychází vždy jediná šipka. Z oválu vycházejí dvě šipky, přičemž jedna z nich je označena nulou (po ní pokračuje výpočet tehdy, když byla v testovaném registru nula, v opačném případě výpočet sleduje druhou šipku). Jestliže v programu za sebou následuje několik příkazů zvýšení, snížení a výpočtu hodnoty funkce, můžeme je napsat pod sebe do jednoho obdélníku.

Příklad. Nechť $F(x)$ je předem daná, ale neznámá funkce. Sestrojte

registrový stroj, který dostane na vstupu jedno číslo n (uložené v registru R_1) a který spočítá součet $F(0) + F(1) + \ldots + F(n-1)$ (uloží ho do registru R_0).



ŘEŠENÍ. Stroj řešící zadanou úlohu vidíme na obrázku. Tento stroj pracuje v cyklu. Při každém průchodu cyklem sníží hodnotu uloženou v registru R_1 o jedničku, vypočítá hodnotu funkce $F(R_1)$ a přičte ji k registru R_0. Budeme potřebovat několik pomocných registrů. Registr R_3 bude sloužit jako vstupní registr pro výpočet funkce F. Do tohoto registru zkopírujeme obsah registru R_1 tak, že nejprve „přesypeme" obsah registru R_1 do registrů R_3 a R_4 a potom zpět obsah R_4 do R_1. Nyní můžeme použít příkaz na výpočet funkce F. Výsledek výpočtu funkce se zapíše do registru R_2. Na dokončení jednoho průchodu cyklem nyní stačí „přisypat" obsah R_2 do registru R_0 a vynulovat registr R_3.

Soutěžní úlohy:

a) Nechť $F(x)$ je nějaká předem daná, ale neznámá rostoucí funkce. (Funkce F je rostoucí, jestliže pro všechna x platí $F(x+1) > F(x)$.) Sestrojte registrový stroj s jedním vstupem y, který bude počítat funkci $G(y)$ inverzní k funkci F. Přesněji řečeno, výsledek výpočtu registrového stroje $G(y) = x$, kde x je nejmenší takové nezáporné celé číslo, pro které platí $F(x) \geqq y$.
Příklad. Předpokládejme, že $F(x) = x^2 + 6$. Potom pro vstupní hodnotu $y = 5$ by měl stroj vypočítat výsledek 0 (protože $F(0) \geqq 5$

a 0 je nejmenší nezáporné celé číslo). Pro vstup $y = 10$ je správným výstupem 2, pro vstup $y = 11$ je správná výstupní hodnota 3.

b) Sestrojte Minského registrový stroj s jedním vstupem n, který zjistí, zda je zápis čísla n ve dvojkové soustavě palindromem. Palindrom je posloupnost cifer, která je stejná, když se čte zepředu i zezadu (není povoleno dopisovat nuly na začátek zápisu čísla). Stroj bude na konci výpočtu obsahovat v registru R_0 jedničku v případě, že dvojkový zápis čísla n je palindrom, nebo nulu v případě, že palindromem není.
Příklad. Zápisy čísel $0_{10} = 0_2$ nebo $21_{10} = 10101_2$ ve dvojkové soustavě jsou palindromy, zápisy čísel $10_{10} = 1010_2$ a $11_{10} = 1011_2$ palindromy nejsou.

P – III – 1

Jednosměrky

V centru města je N důležitých křižovatek očíslovaných od 1 do N, které jsou pospojovány M cestami. Cestou rozumíme asfaltový koberec spojující dvě konkrétní křižovatky. Mezi libovolnými dvěma křižovatkami může vést nejvýše jedna cesta. Všechny cesty jsou obousměrné a na každou křižovatku se lze po těchto cestách dostat ze všech ostatních křižovatek, tzn. pro libovolnou dvojici křižovatek i a j, $i \neq j$, existují křižovatky $a_1, a_2, \ldots, a_k$ takové, že $a_1 = i$, $a_k = j$ a pro $i = 1, 2, \ldots, k-1$ vede mezi křižovatkami a_i a a_{i+1} cesta. Pro účely této úlohy budeme křižovatkou označovat i místo, do něhož vede jen jediná cesta (případně dvě cesty).

V rámci zvyšování dopravní bezpečnosti se městská rada rozhodla co nejvíce cest „zjednosměrnit“. To znamená, že pokud mezi křižovatkami i a j existuje cesta, po „zjednosměrnění od i k j“ (označujeme $i \to j$) bude po této cestě povoleno jet z křižovatky i na křižovatku j, ale ne naopak. Městská rada má jedinou podmínku: z každé křižovatky musí být možné dojet na všechny ostatní křižovatky při dodržení přikázaného směru jízdy.

Soutěžní úloha. Je dán počet křižovatek N a počet cest M. Dále je dáno takových M dvojic křižovatek $\{i, j\}$, $i \neq j$, že mezi křižovatkami i a j vede cesta. Napište program, který vypíše všechny „zjednosměrněné“ cesty a jejich povolený směr jízdy tak, aby byl počet zbývajících obousměrných cest minimální. Jestliže existuje více možných řešení, vypište jedno libovolné z nich.

Příklad: *vstup*: $N = 8$, $M = 10$	*výstup*:
$\{1, 2\}$	$2 \to 3$
$\{2, 3\}$	$3 \to 8$
$\{2, 8\}$	$8 \to 2$
$\{3, 4\}$	$4 \to 5$
$\{3, 8\}$	$5 \to 6$
$\{4, 5\}$	$6 \to 7$
$\{4, 7\}$	$7 \to 4$
$\{5, 6\}$	$5 \to 7$
$\{5, 7\}$	(Jiným možným řešením je
$\{6, 7\}$	přesměrovat $\{5, 7\}$ na $7 \to 5$.)

P – III – 2

Volby

V jisté zemi právě skončily volby. Každý volič v nich hlasoval pro jednoho z navržených kandidátů. Za poslance místního shromáždění jsou zvoleni všichni kandidáti, pro které hlasovala více než jedna k-tina všech voličů. Všimněte si, že v takovýchto volbách je možné zvolit nejvýše $k - 1$ kandidátů (možná jich ale bude zvoleno méně). Máte k dispozici výsledky hlasování jednotlivých voličů a máte co nejrychleji zjistit, kteří kandidáti byli zvoleni do zastupitelstva.

Soutěžní úloha. Voleb se zúčastnilo N voličů očíslovaných od 1 do N a M ($M \leqq N$) kandidátů očíslovaných od 1 do M (někteří kandidáti však nemuseli získat ani jeden hlas). Výsledky hlasování jsou uloženy v poli a, přičemž platí, že volič i ($1 \leqq i \leqq N$) hlasoval pro kandidáta číslo $a[i]$. Obsah tohoto pole nesmíte měnit. Dále máte dáno číslo $k \geqq 2$ (velmi malé v porovnání s N). Napište program, který vypíše čísla všech kandidátů, pro něž hlasovalo více než N/k voličů.

Poznámka. Čísla M a N mohou být *velmi* velká. Snažte se proto, aby váš program pracoval co nejefektivněji a aby používal co nejméně paměti.

Příklad 1: *vstup*:	*výstup*:
$N = 11$, $k = 3$, $M = 7$	Zvoleni jsou kandidáti 1, 2.
$A = (1, 2, 3, 2, 1, 1, 7, 2, 3, 2, 1)$	
Příklad 2: *vstup*:	*výstup*:
$N = 8$, $k = 4$, $M = 4$	Nebyl zvolen žádný kandidát.
$A = (1, 2, 3, 3, 4, 2, 4, 1)$	

P – III – 3

Minského registrové stroje

Definice. (Oproti domácímu kolu se studijní text liší příkladem sestavení Minského stroje z bloků.)

Minského registrový stroj je jednoduché výpočetní zařízení. K dispozici má několik registrů označených R_0, R_1, R_2, ..., přičemž v každém registru může být uloženo jedno libovolně velké nezáporné celé číslo.

Minského registrový stroj může mít jeden nebo více vstupů, jejichž hodnoty jsou na začátku výpočtu uloženy v registrech $R_1, \ldots, R_k$, kde k je počet těchto vstupů. Ostatní registry jsou na začátku výpočtu inicializovány na hodnotu nula. Po skončení výpočtu Minského registrového stroje je výsledkem hodnota uložená v registru R_0.

Každý Minského registrový stroj je řízen pevně daným programem. V programu se mohou vyskytovat tyto příkazy:

- ▷ *Zvýšení* hodnoty v daném registru o 1.
- ▷ *Snížení* hodnoty v daném registru o 1 (pokud je v registru nula, hodnota se nezmění).
- ▷ *Test na nulu* zjistí, zda je v daném registru nula.

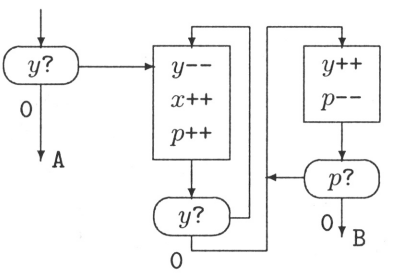
Registr je určen svým číslem. Číslo registru je v příkazu vždy pevně zadáno, není tedy možné k určení registru použít obsah jiného registru.

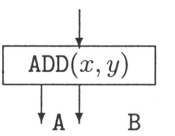
Programy budeme zakreslovat do schémat, přičemž zvýšení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i++, snížení hodnoty registru R_i budeme značit obdélníkem s nápisem R_i−−, test na nulu v registru R_i značíme oválem s nápisem R_i?. Začátek programu označujeme písmenem Z v kroužku a konec programu písmenem K v kroužku (program může mít i více konců, ale jen jeden začátek). Jednotlivé příkazy jsou pospojovány šipkami, které určují tok řízení programu. Z obdélníku vychází vždy jediná šipka. Z oválu vycházejí dvě šipky, přičemž jedna z nich je označena nulou (po ní pokračuje výpočet tehdy, když byla v testovaném registru nula, v opačném případě výpočet sleduje druhou šipku). Jestliže v programu za sebou následuje několik příkazů zvýšení nebo snížení, můžeme je napsat pod sebe do jednoho obdélníku.

Použití bloků. Při kreslení složitějších schémat se nám může stát, že některé části potřebujeme použít vícekrát. Abychom se vyhnuli opakovanému kreslení stejných skupin příkazů, budeme je seskupovat do bloků. *Blok* je skupina příkazů s jedním určeným počátečním příkazem a s jednou nebo více výstupními šipkami. Každý blok musíme nejprve definovat, tj. nakreslit příslušnou skupinu příkazů. Definovaný blok potom

můžeme použít na libovolném místě při kreslení schématu registrového stroje (případně při definování jiných bloků). Blok značíme obdélníkem s vepsaným jménem bloku, z tohoto obdélníku vychází příslušný počet šipek. V definici bloku můžeme některé registry označit proměnnými. Za tyto proměnné se dosadí konkrétní registry až na místě, kde se blok použije (v tom případě napíšeme do obdélníku za jméno bloku do závorek registry, které se mají dosadit za jednotlivé proměnné). Proměnným, za které nedosadíme žádný registr, se nakonec přiřadí nepoužité registry (každá kopie proměnné má vlastní registr). Použití bloků nejlépe objasní příklad.

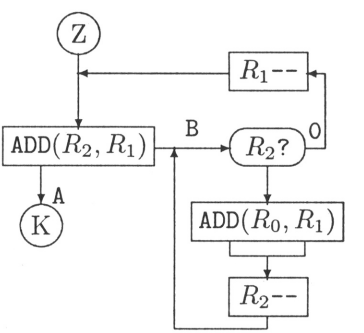
Úloha: Sestrojte stroj, který bude mít na vstupu číslo n (uložené v registru R_1) a který vypočítá součet druhých mocnin čísel od 0 do n (výsledný součet uloží do registru R_0).





Blok ADD a jeho schématická značka

ŘEŠENÍ. Nejprve nadefinujeme blok ADD(x, y), který k registru x přičte obsah registru y (obsah registru y se přitom zachová). Tento blok bude používat jeden pomocný registr p, kterému není explicitně přiřazen žádný konkrétní registr. Bude mít dvě výstupní šipky: jestliže je v registru y nula, použije se šipka A, v opačném případě použijeme šipku B.



Náš registrový stroj bude pracovat následovně: K obsahu registru R_0 (který je na začátku nulový) budeme postupně přičítat $n^2, (n-1)^2, \ldots, 1^2$. V registru R_1 bude vždy uloženo číslo x, jehož druhou mocninu právě přičítáme. Jestliže $x = 0$, končíme (větev A bloku $ADD(R_2, R_1)$). V opačném případě pomocí $ADD(R_2, R_1)$

zkopírujeme obsah R_1 do R_2 (obsah R_2 byl předtím nulový) a pomocí cyklu a dalšího bloku ADD přičítáme do R_0 x-krát číslo x. (Všimněte si, že v R_2 je po proběhnutí tohoto cyklu opět nula.) Tento postup opakujeme, přičemž snižujeme obsah R_1 o jedničku.

Soutěžní úloha. Konečná množina nezáporných celých čísel $M = \{c_1, c_2, \ldots, c_n\}$ se dá jednoznačně zakódovat do jednoho nezáporného celého čísla m takto: $m = \sum_{i=1}^{n} 2^{c_i}$. (V množině se může každý prvek nacházet nejvýše jednou, tzn. $c_i \neq c_j$ pro $i \neq j$.) Jestliže si představíme zápis čísla m ve dvojkové soustavě, potom číslo c patří do množiny M právě tehdy, když c-tý bit zápisu m je jednička. Bity číslujeme zprava doleva, tj. nejpravější bit má číslo 0, jeho levý soused číslo 1, atd.

Sestrojte Minského registrový stroj, který dostane na vstupu kód m nějaké množiny M a číslo s a zjistí, zda lze číslo s vyjádřit jako součet některých prvků množiny M, tj. zda existuje taková množina $M' \subset M$, že $\sum_{c \in M'} c = s$. Číslo m je na začátku výpočtu uloženo v registru R_1, číslo s v registru R_2. Po skončení výpočtu musí registr R_0 obsahovat jedničku v případě, že s je součtem některých prvků množiny M a nulu v opačném případě. Součet nula prvků má definitoricky hodnotu nula.

Příklad. Pro hodnoty $m = 203$ (11001011 v dvojkové soustavě, tzn. $M = \{0, 1, 3, 6, 7\}$), $s = 10$ je správný výsledek 1 ($10 = 3 + 7$). Pro $m = 203$, $s = 12$ je výsledek 0 (protože 12 nelze získat součtem čísel vybraných z množiny M). Pro m libovolné, $s = 0$ je správný výsledek 1.

P – III – 4

Pramen

Program: `pramen.pas` / `pramen.cpp`
Vstup: `pramen.in`
Výstup: `pramen.out`

V jistém lázeňském městečku mají N minerálních pramenů očíslovaných $1, 2, \ldots, N$. Prameny jsou navzájem pospojovány pěšinami, přičemž pěšina vede mezi každou dvojicí pramenů. Turisté obvykle chtějí ochutnat vodu z každého pramenu, ale protože pěšiny nejsou nijak označené, občas se stane, že některého zbloudilého turistu najdou až ve vedlejší dolině. Správce lázní se proto rozhodl umístit ke každému pramenu právě jednu směrovku ukazující na další pramen, a to tak, že turista začínající svou

procházku u libovolného pramenu a pokračující ve své cestě podle směrovek obejde postupně všechny prameny.

Správce tedy nechal vyrobit N směrovek s čísly pramenů $1, \ldots, N$. Dělníci ale příliš nepřemýšleli a rozmístili směrovky k pramenům úplně náhodně. Tak se stalo, že u každého pramene sice byla umístěna právě jedna směrovka ukazující k nějakému prameni, ale mohlo se stát, že pokud vyšel turista od některého pramene, tak se k některým jiným pramenům vůbec nedostal. Mohlo se dokonce stát, že směrovka u některého pramene ukazovala nazpět na ten samý pramen, u kterého byla umístěna.

Správce ví, že na dělníky se může spolehnout jen v této jednoduché operaci: vzájemně zaměnit směrovky u dvou pevně daných pramenů p a q, tzn. jestliže směrovka u pramene p ukazovala na pramen i a směrovka u pramene q ukazovala na pramen j, tak po této výměně bude u pramene p směrovka ukazovat na pramen j a u pramene q bude směrovka ukazovat na pramen i.

Napište program pro správce lázní, který zjistí, jaké výměny směrovek mají dělníci provést, aby bylo možné podle směrovek obejít všechny prameny a aby přitom počet provedených výměn byl nejmenší možný.

Vstupní soubor: Vstupní soubor `pramen.in` obsahuje na prvním řádku počet pramenů N ($1 \leqq N \leqq 30\,000$). Následuje N čísel (oddělených mezerami nebo konci řádků), přičemž i-tým z nich je číslo pramene, na který ukazuje směrovka umístěná u i-tého pramenu.

Výstupní soubor: První řádek výstupního souboru `pramen.out` obsahuje minimální počet výměn M, které je třeba provést. Následuje M řádků popisujících jednotlivé výměny: $(i+1)$-tý řádek výstupního souboru ($1 \leqq i \leqq N$) obsahuje dvě čísla p, q oddělená jednou mezerou, označující dvojici pramenů, u nichž je třeba vyměnit směrovky. Výměny se provádějí v uvedeném pořadí.

Příklad.

`pramen.in`	`7`	`pramen.out`	`2`
	`5 1 4 6 2 3 7`		`1 3`
			`7 1`

Poznámka. Uvedený výstupní soubor je jedním z možných správných výstupních souborů.

P – III – 5

Kina

Program: `kina.pas / kina.cpp`
Vstup: `kina.in`
Výstup: `kina.out`

V jisté zemi právě začal Mezinárodní Filmový Festival Umělecké Kinematografie (MFF UK). V zemi je n ($1 \leqq n \leqq 50$) měst označených čísly $1, 2, \ldots, n$. V každém městě je jedno kino. O každém kině je známo, který film z nabídky filmů MFF UK očíslovaných $1, 2, \ldots, t$ ($1 \leqq t \leqq n$) v něm promítají. Konkrétně, v kině ve městě číslo i promítají každý večer stejný film číslo f_i. Města jsou navzájem pospojována m obousměrnými cestami. Pro každou cestu c ($1 \leqq c \leqq m$) známe čísla měst z_c a k_c ($z_c \neq k_c$), která tato cesta spojuje, a také známe její délku l_c ($0 \leqq l_c \leqq 1\,000$). Mezi každou dvojicí měst vede nejvýše jedna cesta.

Předpokládejme, že máte rozpis k večerů ($0 \leqq k \leqq 1\,000$), přičemž pro každý večer i ($1 \leqq i \leqq k$) je stanoveno číslo filmu p_i, který byste chtěli v ten večer vidět. Některé filmy se v rozpisu mohou vyskytnout i vícekrát. Abyste mohli večer sledovat naplánovaný film, musíte se během dne dopravit do některého města (nezáleží na tom do kterého), v němž tento film uvádějí. Protože neradi cestujete, chtěli byste, aby celková vámi procestovaná vzdálenost byla co nejmenší. První den ráno se nacházíte ve městě číslo 1. Při přejíždění z města do města je možné projíždět přes libovolná jiná města. Z každého města se lze dostat do každého a přeprava se stihne vždy za jeden den. Nezáleží na tom, ve kterém městě zůstanete po shlédnutí posledního filmu.

Napište program, který pro každé město přečte číslo promítaného filmu, dále načte popis jednotlivých cest a rozpis filmů, které chcete v jednotlivých dnech vidět, a vypíše, ve kterém městě máte jít který večer do kina, aby vámi procestovaná dráha byla minimální. Dále vypíše délku této minimální trasy.

Vstupní soubor: Vstupní soubor `kina.in` obsahuje na prvním řádku tři čísla n, m, k. Druhý řádek obsahuje n čísel $f_1, f_2, \ldots, f_n$. Následuje m řádků popisujících cesty, každý z nich obsahuje trojici čísel z_c, k_c, l_c. Poslední řádek obsahuje k čísel $p_1, p_2, \ldots, p_k$. Každá dvojice po sobě následujících čísel v řádku je oddělena jednou mezerou. Můžete předpokládat, že každý film p_i ($1 \leqq i \leqq k$), který chcete vidět, se promítá alespoň v jednom městě.

Výstupní soubor: Výstupní soubor `kina.out` obsahuje na prvním řádku délku celkově procestované dráhy mezi městy. Druhý řádek popisuje jednu optimální trasu. Obsahuje k čísel (každá dvě po sobě jdoucí čísla jsou oddělena mezerou), i-té číslo v pořadí označuje město, v němž byste podle zvolené trasy měli vidět film p_i.

Poznámka. Jestliže výstupní soubor obsahuje správnou minimální délku, ale buď neobsahuje žádný popis optimální trasy, nebo obsahuje popis trasy, který je chybný, bude řešení hodnoceno jako částečně správné a získá pro daný vstup poloviční počet bodů.

Příklad:

`kina.in`

```
6 7 7
2 1 2 3 1 4
1 2 13
2 3 7
3 4 5
4 1 4
1 5 8
5 3 10
2 6 0
1 2 1 4 3 2 1
```

`kina.out`

```
49
5 3 2 6 4 3 2
```

P – I – 1

Pro výrobní operaci i označíme jako h_i nejkratší čas od spuštění výroby, v němž může být operace i dokončena. Protože v továrně lze vykonávat najednou libovolný počet výrobních operací, můžeme výrobu uspořádat tak, že každá operace i bude skutečně dokončena v čase h_i. Na zhotovení celého výrobku potřebujeme dokončit všechny stanovené výrobní operace, takže výrobek je možné dokončit nejdříve v čase $\max\{h_1, h_2, \ldots, h_n\}$.

Zbývá nám spočítat čas h_i. Jestliže výrobní operace i nemá žádné předcházející operace, můžeme ji okamžitě spustit. Hodnota h_i je v takovém případě přímo rovna délce trvání operace i. Pokud operace i má nějaké předcházející operace $p_1, p_2, \ldots, p_k$, je třeba vykonat nejprve tyto předcházející operace. Výrobní operace i může začít nejdříve v čase, který je maximem z časů $h_{p_1}, h_{p_2}, \ldots, h_{p_k}$. Když víme, kdy operace i začne a známe délku jejího trvání, umíme triviálně spočítat, kdy skončí.

Hodnoty h_i budeme počítat rekurzívně. Program obsahuje rekurzívní funkci *urci_cas*(i), která spočítá hodnotu h_i. Pokud operace i nemá předchůdce, bude to jednoduše čas jejího trvání. Jestliže operace i má nějaké předchůdce, spustí se funkce *urci_cas* pro každého předchůdce operace i, z takto získaných časů se vezme maximum a k němu se připočítá čas trvání operace i. Navíc, jakmile vypočítáme hodnotu h_i pro nějaké i, uložíme si ji do pomocného pole. Když později zavoláme funkci *urci_cas* pro stejné i znovu, tato funkce už nebude znovu počítat, ale jednoduše vrátí už vypočítanou hodnotu z pole.

Nejprve dokážeme, že popsaný program vždy skončí. Předpokládejme, že bychom počítali nějakou hodnotu h_i a při jejím výpočtu bychom potřebovali vypočítat nějakou hodnotu h_{j_1} pro předcházející operaci j_1. Při výpočtu h_{j_1} bychom potřebovali vypočítat nějakou hodnotu h_{j_2} a tak bychom se vnořovali hlouběji a hlouběji, až bychom zjistili, že k výpočtu nějakého h_{j_k} potřebujeme znát h_i. To by způsobilo nekonečné volání rekurze. Takovýto případ však může nastat jedině tehdy, jestliže není možné vykonat operaci i, protože operace i se může provést až po skončení j_1 a j_1 lze provést až po skončení j_2 atd., a j_k je možné vykonat až po skončení i. Dostali jsme tedy cyklus, takže i není možné provést vůbec. V zadání je však uvedeno, že všechny operace lze provést, takže tento případ nemůže nastat a náš algoritmus vždy skončí.

Na závěr zhodnotíme výpočtovou složitost algoritmu. Předpokládejme, že pro každou operaci máme seznam předcházejících operací uložen ve spojovém seznamu. Označme n počet výrobních operací a m celkový počet předcházejících operací pro všechny operace v technologickém plánu výrobku. V první části algoritmu načítáme a ukládáme potřebné údaje. Tato část má časovou složitost $O(m+n)$.

V druhé části algoritmu zavoláme funkci *urci_cas*(i) pro každé i od 1 do n. V rámci každého takového volání se funkce může ještě dále rekurzívně volat. Funkci *urci_cas*(i) voláme pro každé i jednou z hlavní části algoritmu a pro každou operaci, pro níž je operace i předchůdcem, tuto funkci opět zavoláme. Celkem se tedy vykoná $m+n$ volání funkce *urci_cas*. Pro každou hodnotu i se ale hodnota *urci_cas*(i) počítá pouze jednu a při dalších voláních se jen najde výsledek v poli v konstantním čase. Máme tedy přesně m volání funkce *urci_cas*, která proběhnou v konstantním čase. Zbývajících n volání počítá hodnotu. Při volání, které počítá hodnotu, se projde seznam všech předcházejících operací a hledá se maximum z časů. Délka takovéhoto volání je tedy úměrná počtu předcházejících operací (pokud zanedbáme čas potřebný na vnořená rekurzivní volání). Celkový čas všech n volání, která počítají hodnotu, bude tedy $O(m+n)$. Všech $m+n$ volání bude také trvat dobu $O(m+n)$. Celková časová složitost algoritmu je proto $O(m+n)$. Paměťová složitost je rovněž $O(m+n)$.

Dvě poznámky na závěr. Pokud bychom vypočítané hodnoty ve funkci *urci_cas* neukládali do pole, ale počítali bychom je vždy znovu, dostaneme algoritmus s exponenciální časovou složitostí. Všimněte si, že úlohu je možné jednoduše přetransformovat do teorie grafů — operace budou představovat vrcholy grafu a hrany (orientované) povedou vždy od operace k jejímu předchůdci. Úkolem je potom nalézt orientovanou cestu s největším součtem časů ve vrcholech. Algoritmus, který jsme uvedli, je v grafové terminologii pouze jednoduchou modifikací prohledávání do hloubky.

```
program P_I_1;

type ppostupy=^postupy;
     postupy=record          {seznam operací}
       postup:integer;
       dalsi:ppostupy;
     end;
```

```
var predpostupy:array[1..100] of ppostupy;
       {seznam předcházejících operací pro každou výrobní operaci}
    cas,hotovo:array[1..100] of integer;
       {cas[i] je čas, kolik trvá operace i,
        hotovo[i] je nejnižší čas, kdy může být operace i hotova}
    n:integer;              {počet operací}

procedure nacti;
var t:text;
    i,j:integer;
    novy:ppostupy;
begin
  assign(t,'TOVARNA.IN');
  reset(t);
  readln(t,n);              {načtení počtu operací}
  for i:=1 to n do begin
    read(t,cas[i]);         {načtení času jednotlivých operací}
    predpostupy[i]:=nil;    {seznam předcházejících operací}
    read(t,j);
    while j<>-1 do begin
      new(novy);            {vložit operaci j do seznamu}
      novy^.postup:=j;
      novy^.dalsi:=predpostupy[i];
      predpostupy[i]:=novy;
      read(t,j);
    end;
  end;
  close(t);
end;

function urci_cas(postup:integer):integer;
{vrátí nejmenší čas, v němž může být operace hotova}
var max,kdy:integer;
    pom:ppostupy;
begin
  if hotovo[postup]=-1 then begin
   {pokud jsme čas ještě nepočítali, vypočítáme ho
    a uložíme do pole „{"hotovo}, jinak neděláme nic}
    max:=0;
    pom:=predpostupy[postup];
    while pom<>nil do begin
      kdy:=urci_cas(pom^.postup);
      if kdy>max then max:=kdy;
      pom:=pom^.dalsi;
    end;
```

```
    {max je čas, kdy jsou hotové všechny předcházející operace}
    hotovo[postup]:=cas[postup]+max;
  end;
  urci_cas:=hotovo[postup];
end;

procedure vypocitej;
var i,max,kdy:integer;
    t:text;
begin
  for i:=1 to n do          {inicializace}
    hotovo[i]:=-1;
  max:=0;                   {zjistíme, kdy skončí poslední operace}
  for i:=1 to n do begin
    kdy:=urci_cas(i);
    if kdy>max then max:=kdy;
  end;
  assign(t,'TOVARNA.OUT');
  rewrite(t);
  writeln(t,max);
  close(t);
end;

begin
  nacti;
  vypocitej;
end.
```

P – I – 2

Je známo více algoritmů na konstrukci řetězce obsahujícího všechny permutace znaků dané abecedy jako podřetězce. Nejprve uvedeme příklad jednoduchého algoritmu, který pro N-prvkovou abecedu sestrojí řetězec délky $N^2 - N + 1$. Potom ukážeme komplikovanější algoritmus, který sestrojí řetězec délky pouze $N^2 - 2N + 4$ (pro $N \geqq 3$). Není známo, zda existuje i nějaký kratší řetězec (existuje dolní odhad počtu znaků takovéhoto řetězce, tento odhad je však nižší než $N^2 - 2N + 4$).

Nejprve si ukážeme jednodušší konstrukci řetězce. Mějme N-prvkovou abecedu $\{a_1, a_2, \ldots, a_N\}$. Řetězec bude tvořen $N - 1$ úseky tvaru $a_1 \times$ $\times a_2 \ldots a_N$ a ukončen bude znakem a_1. Například pro abecedu $\{a, b, c\}$ sestrojíme řetězec *abcabca*. Tento řetězec obsahuje celkem $(N-1) \cdot N +$ $+1 = N^2 - N + 1$ znaků. Pro $N = 15$ tedy dostáváme řetězec délky 211.

Zbývá dokázat, že takto sestrojený řetězec skutečně obsahuje každou permutaci znaků abecedy jako vybraný podřetězec. Rozdělíme si tedy řetězec na $N-1$ úseků tvaru $a_2a_3\ldots a_N$. Mezi každými dvěma těmito úseky, stejně jako za posledním a před prvním z nich, se nachází znak a_1. Vezměme nyní libovolnou permutaci znaků abecedy. Nejprve z této permutace vynecháme znak a_1. Zbytek permutace je jistě vybraným podřetězcem našeho řetězce, neboť z každého úseku tvaru $a_2a_3\ldots a_N$ můžeme vybrat právě jeden znak tak, abychom z j-tého úseku vybrali j-tý znak permutace různý od a_1.

Nechť je nyní znak a_1 umístěn v permutaci na i-tém místě, tzn. před ním leží $i-1$ znaků. Potom z řetězce vybereme v pořadí i-tý výskyt znaku a_1. Před ním se nachází $i-1$ úseků tvaru $a_2a_3\ldots a_N$, a tedy a_1 bude i ve vybraném podřetězci na i-tém místě. Dokázali jsme, že pro libovolnou permutaci znaků abecedy můžeme nalézt takový vybraný podřetězec našeho řetězce, který se této permutaci rovná.

Nyní si popíšeme jinou, složitější konstrukci. Budeme vytvářet posloupnost řetězců $T(1), T(2), T(3), \ldots$, přičemž $T(N)$ obsahuje všech $N!$ permutací N znaků jako podřetězce. Ukážeme si způsob, jak z $T(N)$ sestrojit $T(N+1)$ (pro $N \geqq 3$).

Řetězce $T(1)$, $T(2)$ a $T(3)$ zvolíme pevně: $T(1) = a_1$, $T(2) = a_1a_2a_1$ a $T(3) = a_1a_3a_2a_1a_3a_1a_2$. Jsou to nejkratší možné řetězce pro $N = 1, 2, 3$ (to lze snadno dokázat ověřením všech možností).

Mějme nyní abecedu $\{a_1, a_2, \ldots, a_N\}$. K řetězci $T(N)$ vytvoříme posloupnost *základních bodů*. První základní bod řetězce $T(N)$ je index prvního výskytu znaku a_1 v $T(N)$. Pro $i > 1$ je i-tým základním bodem index prvního výskytu znaku a_i za $(i-1)$-ním základním bodem v řetězci. Například v $T(3) = a_1a_3a_2a_1a_3a_1a_2$ je posloupnost základních bodů $(1, 3, 5)$.

Algoritmus pro konstrukci řetězce $T(N)$ z řetězce $T(N-1)$:

1. Pro každé $i = 2, 3, \ldots, N-1$ vsunout těsně před i-tý základní bod řetězce $T(N-1)$ znak a_N.
2. Na konec takto vzniklého řetězce připojit úsek

$$a_2, a_3, \ldots, a_{N-3}, a_N, a_1, a_{N-1}.$$

Podle prvního bodu algoritmu jsme vložili $N-2$ znaků a podle druhého bodu jsme vložili $N-1$ znaků. Celkově se tedy řetězec prodloužil o $2N-3$ znaků. Všechny základní body řetězce $T(N-1)$ budou i základními body řetězce $T(N)$ (jen se posunou kvůli vsouvání znaků a_N

podle bodu 1) a N-tým základním bodem se stane znak a_N přidaný v bodě 2. Například pro $N = 4$ dostáváme pomocí tohoto algoritmu $T(4) = a_1a_3a_4a_2a_1a_4a_3a_1a_2a_4a_1a_3$.

Řetězec $T(N)$ sestrojený naším algoritmem (pro $N \geqq 3$) má délku N^2-2N+4. Toto tvrzení dokážeme matematickou indukcí. Délka řetězce $T(3)$ je 7, což je rovno $3^2-2\cdot3+4$. Předpokládejme nyní, že délka $T(N-1)$ je $(N-1)^2-2(N-1)+4$ a dokážeme, že potom délka řetězce $T(N)$ je N^2-2N+4. Řetězec $T(N)$ vznikl přidáním $2N-3$ znaků do řetězce $T(N-1)$, jeho délka je $(N-1)^2-2(N-1)+4+2N-3 = N^2-2N+4$. Dokázali jsme tedy, že délka řetězce $T(N)$ je skutečně N^2-2N+4.

Nakonec je třeba dokázat, že řetězec $T(N)$ obsahuje všech $N!$ permutací znaků abecedy jako podřetězce. Tento důkaz je poměrně zdlouhavý, takže uvedeme jen základní myšlenku. Řetězec $T(N)$ rozdělíme na úseky. První úsek začíná prvním základním bodem a končí druhým základním bodem. Pro $2 \leqq i \leqq N-1$ bude i-tý úsek začínat znakem a_1, který je hned za i-tým základním bodem, a končit znakem a_i, který se vyskytuje poprvé za $(i+1)$-ním základním bodem.

Matematickou indukcí (vzhledem k N) lze dokázat, že každý z takto vytvořených úseků obsahuje znaky $a_1, a_2, \ldots, a_N$ a začíná znakem a_1. Řetězec obsahuje $N-1$ takovýchto úseků a každý z nich obsahuje právě jeden základní bod s výjimkou prvního úseku, který obsahuje dva základní body. Další vlastností dvou sousedních úseků i-tého a $(i+1)$-ního je jejich vzájemné překrytí ve dvou znacích, a to a_1 a a_i.

Nechť má naše permutace znak a_1 na i-tém místě. Uvažujme nejprve případ že $i \leqq N-1$. V permutaci vybereme první výskyt znaku a_1 z i-tého úseku řetězce $T(N)$. Z i-tého úseku je možné použít ještě jeden znak, neboť se v něm vyskytují všechny znaky, a to až za vybraným znakem a_1. Před i-tým úsekem je $i-1$ úseků a za ním je $N-i-$ -1 úseků. Při výběru jednoho znaku z každého úseku tedy dostaneme všechny permutace se znakem a_1 v i-té pozici. Překrytí sousedních úseků nezpůsobí problémy, protože při výběru znaku a_1 z i-tého úseku je možné všechny ostatní znaky a_1 ignorovat a další společné znaky se stávají hraničními prvky úseků. Hraniční prvek je považován za prvek patřící jen do jednoho úseku.

Zbývá dokázat tvrzení v případě, že a_1 je v permutaci na N-tém místě. V tomto případě řetězec $T(N)$ rozdělíme na úseky se základními body jako hraničními body úseků. Počet úseků je $N-1$ a každý z nich obsahuje znaky $a_1, a_2, \ldots, a_N$. Jestliže na poslední místo vybereme poslední výskyt znaku a_1 v řetězci $T(N)$, tento znak nepatří do žádného z výše uvedených

úseků. Řetězec $T(N)$ tedy obsahuje všechny permutace se znakem a_1 na posledním místě.

Uvedený algoritmus pro $N = 15$ dává následující řetězec délky 199:

```
acdefghijklmnobadefghijklmnocabefghijklmnodacbfghijklmnoead
bcghijklmnofaebcdhijklmnogafbcdeijklmnohagbcdefjklmnoiahbcd
efgklmnojaibcdefghlmnokajbcdefghimnolakbcdefghijnomalbcdefg
hijkonambcdefghijkloan
```

Na závěr uvádíme program, který řetězec $T(15)$ generuje výše popsaným způsobem.

```
program P_I_2;
const  N=15;                          {počet prvků abecedy}
       abeceda:string='abcdefghijklmno';  {abeceda řetězce}
var T: string;                        {hledaný řetězec}
    BB: array[1..N] of integer;       {indexy základních bodů}
    i: integer;
    ff: text;

function Dalsi(n: integer; T:string):string;
{funkce dostane řetězec všech permutací prvních n-1 prvků
 a vrátí řetězec všech permutací prvních n prvků (n>3)}
var i, j: integer;
    TN: string;
begin
  TN:='';
  i:=BB[1];
  j:=2;

  {přidání nových prvků před základní body}
  while j<n do begin
    TN:=TN+copy(T,i,BB[j]-BB[j-1]);
    i:=BB[j];
    j:=j+1;
    TN:=TN+abeceda[n];
  end;
  TN:=TN+copy(T,BB[n-1],length(T));

  {přidání prvků na konec řetězce}
  for i:=2 to n-3 do TN:=TN+abeceda[i];
  TN:=TN+abeceda[n];                  {nový základní bod}
  TN:=TN+abeceda[1]+abeceda[n-1];

  {úprava základních bodů}
```

```
  BB[n]:=length(TN)-2;                {nový základní bod}
  for i:=2 to n-1 do                  {původní body se posunuly}
    BB[i]:=BB[i]+i-1;

  Dalsi:=TN;
end; {Dalsi}

procedure VytvorT3;
{vytvoří řetězec všech permutací pro tříprvkovou abecedu}
begin
  T:=abeceda[1]+abeceda[3]+abeceda[2]+abeceda[1]
     +abeceda[3]+abeceda[1]+abeceda[2];
  BB[1]:=1; BB[2]:=3; BB[3]:=5;
end;

begin
  assign(ff,'P-1-2.TXT');
  rewrite(ff);
  VytvorT3;
  for i:=4 to N do
     T:=Dalsi(i,T);
  writeln(ff,T);
  close(ff);
end.
```

P – I – 3

Tuto úlohu si můžeme představit tak, že máme dáno pole n navzájem různých čísel $a[1], a[2], \ldots, a[n]$ (preference politických stran) a máme za úkol nalézt nejmenší a největší prvek tohoto pole, přičemž ale k poli můžeme přistupovat pouze prostřednictvím funkce *pruzkum*(i, j), která nám říká, zda je $a[i] < a[j]$, nebo $a[j] < a[i]$.

Ukážeme si nejprve řešení pro sudé n. Rozdělíme všechny prvky pole do dvojic a v každé dvojici prvky porovnáme (pomocí funkce *pruzkum*). Prvky se nám rozdělí na dvě podmnožiny — do množiny X dáme ty, které byly při porovnávání v dvojici větší, a do množiny Y ty, které byly při porovnávání menší. Je zřejmé, že žádný prvek z množiny X nebude nejmenším prvkem v poli, neboť existuje aspoň jeden menší prvek (ten, který s ním byl ve dvojici). Proto při hledání minima stačí hledat mezi prvky množiny Y. Podobně žádný prvek z Y nebude největším prvkem pole, a proto stačí maximum hledat mezi prvky množiny X.

Nejmenší z prvků množiny Y najdeme jednoduše. V proměnné *min* si budeme pamatovat nejmenší dosud nalezený prvek. Na začátku to bude libovolný prvek množiny Y. Potom budeme nejmenší dosud nalezený prvek porovnávat vždy s dalším a dalším prvkem množiny Y. Pokaždé, když bude porovnávaný prvek menší než prvek uložený v proměnné *min*, stane se on dosud nejmenším nalezeným prvkem (uloží se do proměnné *min*). Podobně budeme hledat i největší prvek v množině X.

V případě, že n je liché, budeme postupovat stejným způsobem, jenom do dvojic rozdělíme jen prvních $n-1$ prvků a poslední prvek nakonec přidáme do množiny X i do množiny Y. V některých případech můžeme ještě jedno porovnávání ušetřit — pokud se tento poslední prvek stane nejmenším prvkem množiny Y (a tudíž i nejmenším prvkem celého pole), pak ho můžeme z množiny X vynechat, neboť jistě nebude současně největším prvkem. Případ $n = 1$ ošetříme zvlášť. V tomto případě není třeba nic porovnávat — jediná strana má současně nejvyšší i nejnižší preference.

Nyní spočítáme, kolik porovnání v nejhorším případě vykonáme pro dané n. Je-li n sudé, vznikne nám $\frac{1}{2}n$ dvojic. Pro každou dvojici provedeme jedno porovnání, abychom zjistili, který prvek je větší. Množiny X a Y budou mít každá $\frac{1}{2}n$ prvků. Pro nalezení minima (nebo maxima) v množině s x prvky použijeme $x-1$ porovnání (do proměnné *min* uložíme nejprve první prvek, potom tuto proměnnou postupně porovnáme se všemi ostatními $x-1$ prvky). K nalezení minima v množině Y tedy potřebujeme $\frac{1}{2}n-1$ porovnání a k nalezení maxima v množině X dalších $\frac{1}{2}n-1$ porovnaní. Celkový počet porovnání tedy bude

$$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \left(\frac{n}{2} - 1\right) = \frac{3n-4}{2}.$$

Pro liché n budeme mít $\frac{1}{2}(n-1)$ dvojic, takže na začátku vykonáme $\frac{1}{2}(n-1)$ porovnání ve dvojicích. Množiny X a Y však budou mít $\frac{1}{2}(n-1)+1$ prvků, takže k nalezení minima nebo maxima z takovéto množiny potřebujeme $\frac{1}{2}(n-1)$ porovnání. Celkový počet provedených porovnání proto bude

$$\frac{n-1}{2} + \frac{n-1}{2} + \frac{n-1}{2} = \frac{3n-3}{2}.$$

Došli jsme k závěru, že pro n sudé vykonáme nejvýše $\frac{1}{2}(3n-4)$ porovnání a pro n liché nejvýše $\frac{1}{2}(3n-3)$ porovnání. Tento výsledek lze zapsat i jedním vzorcem s použitím horní celé části a dolní celé části

takto:

$$2\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor - 2.$$

Na závěr pár slov o implementaci popsaného algoritmu. Množiny X a Y nebudeme vytvářet na začátku výpočtu celé, ale průběžně. Nejprve vezmeme první dva prvky v poli, porovnáme je a menší z nich se stane počáteční hodnotou proměnné *min*, zatímco větší bude počáteční hodnotou proměnné *max*. Potom budeme brát vždy další a další dvojici, porovnáme vždy její prvky navzájem, následně menší z nich porovnáme s proměnnou *min* (a pokud bude třeba, tak obsah *min* změníme) a větší porovnáme s proměnnou *max*. Pro lichá n je třeba nakonec zvlášť zpracovat poslední prvek. Takto naprogramovaný algoritmus bude mít lineární časovou složitost (tzn. $O(n)$) a konstantní paměťovou složitost.

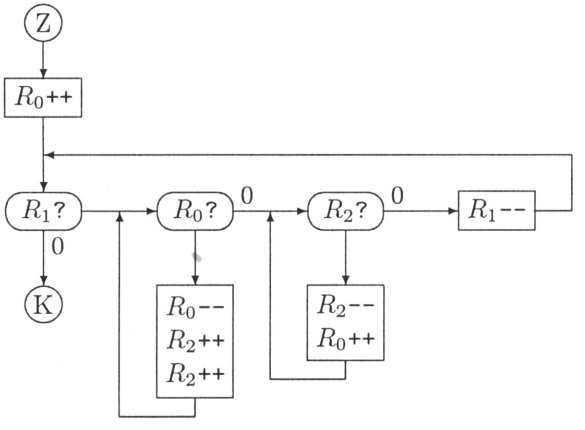
```
program P_I_3;
var n,i,min,max,a,b,pom:integer;

procedure pruzkum(i,j:integer; var min, max:integer);
{provede průzkum pro i a j
 stranu s menšími preferencemi vrátí v min, s většími v max}
begin
  writeln('Proveď průzkum ',i,',',j);
  write('> ');
  readln(max);
  if max=i then min:=j
           else min:=i;
end;

begin
  writeln('Zadej počet stran:');
  write('> '); readln(n);
  if n=1 then begin       {speciální případ}
    min:=1; max:=1;
  end
  else begin              {inicializace - porovnáme první pár}
    pruzkum(1,2,min,max);
  end;
  i:=3;
  while i+1<=n do begin     {ostatní páry}
    pruzkum(i,i+1,a,b);     {porovnáme navzájem}
    pruzkum(min,a,min,pom); {menší porovnáme s minimem}
    pruzkum(max,b,pom,max); {větší s maximem}
    i:=i+2;
```

```
  end;
  if i=n then begin          {poslední prvek pro liché n}
    pruzkum(min,n,min,pom); {porovnáme s min i s max,
                             pokud je třeba}
    if min<>n then
      pruzkum(max,n,pom,max);
  end;
  writeln('Nejvyšší preference má strana ',max,', nejnižší ',min);
end.
```

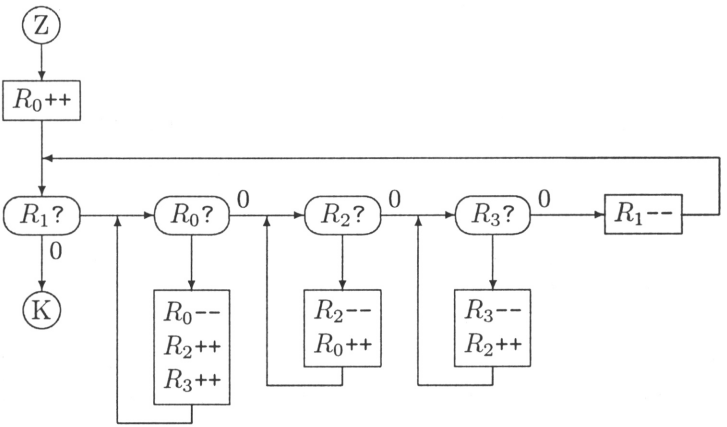
P – I – 4

a) Úkolem bylo sestrojit stroj, který pro vstup n vypočítá číslo 2^n. Číslo n máme uloženo v registru R_1. Na začátku výpočtu uložíme do R_0 číslo 1 (tj. 2^0). Potom budeme postupovat tak, že v každém kroku snížíme R_1 o 1 a registr R_0 vynásobíme 2. To provádíme tak dlouho, dokud v R_1 není 0. Tehdy máme v R_0 uloženo hledané číslo 2^n.

Zbývá už jen popsat, jak násobíme registr R_0 dvěma. Jednou možností by bylo uložit do některého dalšího registru hodnotu 2 a potom použít algoritmus násobení uvedený v příkladu ve studijním textu. My však využijeme trochu zjednodušený algoritmus, který lze použít pouze pro násobení konstantou. V cyklu budeme po jedné snižovat hodnotu R_0, dokud neklesne na nulu, a při každém snížení R_0 dvakrát zvýšíme o 1 hodnotu registru R_2. Po skončení cyklu máme v R_2 dvojnásobek hodnoty, kterou jsme měli původně v R_0. Zbývá už jen přesunout hodnotu z R_2 zpět do R_0, což provedeme dalším cyklem, který vždy jednou sníží R_2 a zvýší R_0, dokud nebude v R_2 nula.



b) Úkolem je vypočítat n-té Fibonacciho číslo F_n. Podle definice $F_0 = F_1 = 1$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geqq 2$. Jestliže si však zavedeme pomocný člen této posloupnosti $F_{-1} = 0$, bude platit, že $F_n = F_{n-1} + F_{n-2}$ i pro $n = 1$, což sníží počet případů, které je třeba v programu pro Minského stroj speciálně ošetřit.



Náš stroj bude pracovat tak, že v cyklu bude snižovat registr R_1 a počítat další člen Fibonacciho posloupnosti, dokud nebude v R_1 nula. Po k průchodech tohoto cyklu bude registr R_0 obsahovat F_k a registr R_2 bude obsahovat F_{k-1}. Celkem se provede n průchodů, takže stroj správně vypočítá hodnotu F_n.

Před prvním prováděním cyklu (tj. po nula průchodech) potřebujeme mít v registru R_0 číslo $F_0 = 1$ a v registru R_2 číslo $F_{-1} = 0$. Toho dosáhneme snadno příkazem R_0++.

Zbývá nám prozkoumat, co potřebujeme provádět v těle cyklu. V registru R_0 máme číslo F_k a v registru R_2 číslo F_{k-1}. Chceme dosáhnout toho, aby v registru R_0 bylo $F_{k+1} = F_k + F_{k-1}$, tj. součet registrů R_0 a R_2, a v registru R_2 aby bylo F_k, tj. číslo, které bylo původně v R_0. Budeme postupovat tak, že hodnotu v R_0 přičítáme k registrům R_2 a R_3 (a to tak, že budeme v cyklu R_0 snižovat, dokud neklesne na nulu, a pokaždé při tom zvýšíme R_2 a R_3). Po skončení této operace máme v R_0 nulu, v R_2 máme F_{k+1} a v R_3 máme F_k. Potřebujeme nyní ještě obsahy registrů přemístit tak, abychom do R_0 dostali obsah R_2 a do R_2 obsah R_3. To provedeme opět obvyklým způsobem (dvěma „přesýpacími“ cykly).

Jednotlivé počítače nazvime vrcholmi. Z vrchola i do vrchola j nech vedie hrana práve vtedy, keď je počítač i pripojený k počítaču j. Vznikne nám tak orientovaný graf G. Našou úlohou je vybrať množinu vrcholov M s minimálnym počtom prvkov takú, že pre ľubovoľný vrchol v existuje vrchol $u \in M$ taký, že z u do v vedie cesta (pripúšťame cestu dĺžky 0, t.j. $u = v$).

Množinu vrcholov U, pre ktorú platí, že

1. pre ľubovoľné dva vrcholy $u, v \in U, u \neq v$ existuje cesta z u do v vedúca len cez vrcholy patriace do U
2. zo žiadneho vrchola $w \in G$ nepatriaceho do U nevedie hrana do žiadneho vrchola $u \in U$

nazvime *maximálny silne súvislý komponent* (MSSK) grafu G (poznamenajme, že vrchol, do ktorého nevedie žiadna hrana tvorí maximálny silne súvislý komponent). Každé dva rôzne MSSK sú disjunktné. Keby MSSK U a MSSK V ($U \neq V$) neboli disjunktné, potom existuje u, ktorý patrí do obidvoch a vrchol v, ktorý patrí do U a nepatrí do V (alebo naopak). Z vrchola v vedie cesta do vrchola u. Na nej niekde musia za sebou nasledovať vrchol v', ktorý do V nepatrí a vrchol u', ktorý už do V patrí — množina V nemá vlastnosť 2 MSSK — spor.

Ľahko vidno, že z každého MSSK grafu G musí nejaký vrchol patriť do množiny M. Zároveň stačí, aby z každého MSSK bol v množine M jeden ľubovoľný vrchol.

Budeme ofarbovať vrcholy grafu rôznymi farbami. Začnime z ľubovoľného neofarbeného vrchola v a ofarbime farbou f všetky neofarbené vrcholy (vrátane v), do ktorých existuje cesta z v vedúca cez doteraz neofarbené vrcholy. Farbenie realizujeme prehľadávaním grafu do hĺbky. Vrchol v vyhlásime za *reprezentanta* farby f. Potom si zvolíme ďalšiu farbu f a iný neofarbený vrchol v a opakujeme, kým sa neminú všetky neofarbené vrcholy. Do každého vrchola zrejme existuje cesta z niektorého z reprezentantov. Niektorí reprezentanti sú však zbytoční: ak do vrchola r_1, reprezentujúceho farbu f_1 vedie cesta z vrchola r_2 reprezentujúceho farbu f_2, z r_2 sa dá dostať do všetkých vrcholov ofarbených farbou f_1, a preto r_1 je zbytočný.

Ako rýchlo zistiť, ktorí reprezentanti sú zbytoční? Nech r_i je zbytočný reprezentant. Potom existuje reprezentant r_j taký, že z r_j vedie do r_i cesta prechádzajúca len cez vrcholy ofarbené farbami f_j a f_i, pričom farby na ceste sa nestriedajú, t.j. cesta vedie najprv cez vrcholy farby

f_j a potom cez vrcholy farby f_i. Posledný vrchol na tejto ceste ofarbený farbou f_j označme v, prvý vrchol ofarbený f_i nech je u. Vrcholy farby f_j boli zrejme ofarbované neskôr ako vrcholy farby f_i. Preto ak by sme v okamihu ofarbovania vrchola v vedeli, že z vrchola u sa dá dostať do vrchola r_i, veľmi ľahko by sme prišli na zbytočnosť r_i. Preto si pre každú farbu f_i a pre každý vrchol u ofarbený touto farbou spočítame, či sa z vrchola u dá dostať do reprezentanta farby f_i t.j. vrchola r_i. Táto práca sa dá tiež zveriť rekurzívnej ofarbovacej procedúre. Na záver treba skontrolovať, pre každú hranu, ktorá vedie medzi vrcholmi rôznych farieb, či sa z jej koncového vrchola dá dostať do reprezentanta farby koncového vrchola. Ak áno, označíme reprezentanta tejto farby ako zbytočného. Táto kontrola sa dá tiež robiť počas ofarbovania.

Procedúra `Prehladaj(v : integer)` dostane ako argument číslo vrchola, z ktorého má prehľadávať. Označme $N(v)$ množinu vrcholov, do ktorých vedie z v hrana. Pred spustením procedúry `Prehladaj` z vrchola r_i si poznačíme, že z vrchola r_i sa dá dostať do reprezentanta r_i. Táto procedúra

- ofarbí vrchol v aktuálnou farbou f_i;
- rekurzívne sa zavolá pre všetky vrcholy $u \in N(v)$, ktoré ešte nie sú ofarbené;
- ak existuje vrchol $u \in N(v)$ farby f_i, z ktorého sa dá dostať do reprezentanta r_i, poznačíme si, že aj z v sa dá dostať do r_i;
- ak existuje vrchol $u \in N(v)$ ofarbený farbou $f_j \neq f_i$ a pritom z u existuje cesta do reprezentanta r_j, poznačíme si, že reprezentant r_j je zbytočný.

Ostáva ukázať, že množina všetkých reprezentantov, ktorí nie sú zbytoční, tvorí našu hľadanú množinu M. Ľahko vidno, že do každého zbytočného reprezentanta r vedie cesta z nejakého reprezentanta, ktorý nie je zbytočný. Nech to tak nie je. Potom existuje reprezentant r_1 taký, že z r_1 vedie do r cesta. Ak by aj ten bol zbytočný, existuje r_2 taký, že z neho vedie cesta do r_1 atď. Buď po nejakom čase prídeme k reprezentantovi, do ktorého už nič nevedie, alebo sa nám začnú reprezentanti opakovať, teda nejakí dvaja, r_i a r_j sa v postupnosti vyskytnú aspoň dvakrát. Jeden z nich musel byť ofarbovaný skôr, nech je to r_i. Potom ale z r_i vedie do r_j cesta, a preto by mal byť r_j ofarbený farbou f_i (alebo inou, použitou skôr ako f_i) — spor. Z množiny nezbytočných reprezentantov sa teda dá dostať do ľubovoľného reprezentanta, a teda aj do ľubovoľného vrchola. Zároveň žiadni dvaja reprezentanti nemôžu ležať v rovnakom MSSK

(lebo inak by museli byť ofarbení rovnakou farbou), teda ich je naozaj minimálny možný počet.

Časová a pamäťová zložitosť: Prehľadávanie každého vrchola je úmerné počtu hrán, ktoré z neho vedú. Žiadny vrchol sa neprehľadáva viac než raz, preto celková časová zložitosť algoritmu je $O(M+N)$, kde M je celkový počet hrán v grafe. Pamäťová zložitosť je tiež $O(M+N)$, pretože si potrebujeme zapamätať celý graf.

P – II – 2

Riešenie tohoto príkladu používa metódu dynamického programovania. Označme $A_i = a[1], a[2], \ldots, a[i]$ postupnosť utvorenú z prvých i členov postupnosti a, analogicky $B_j = b[1], \ldots, b[j]$. Budeme riešiť všeobecnejšiu úlohu: Pre každé i, j, $(0 \leqq i \leqq M,\ 0 \leqq j \leqq N)$ vypočítame, aký je maximálny súčet vybranej podpostupnosti postupností A_i a B_j. Tieto maximálne súčty si budeme zapisovať do tabuľky $p[0..M, 0..N]$, kde $p[i,j]$ je súčet maximálnej vybranej podpostupnosti postupností A_i a B_j. Hľadaný maximálny súčet bude teda hodnota $p[M,N]$.

Tabuľku p budeme vypĺňať po riadkoch s využitím predpočítanej informácie v predošlom riadku. Riadok $p[0]$ obsahuje samé nuly, pretože neexistuje vybraná podpostupnosť prázdnej postupnosti. Riadok $p[i]$ (pre $i > 0$) vyplníme podľa riadku $p[i-1]$ takto: Políčko $p[i,0]$ je zrejme nula. Políčko $p[i,j]$ (pre $j > 0$) vieme vyplniť pomocou hodnôt $p[i-1,j]$, $p[i,j-1]$ a $p[i-1,j-1]$. Ak sa čísla $a[i]$ a $b[j]$ nezhodujú, každá vybraná podpostupnosť postupností A_i a B_j je zároveň vybranou podpostupnosťou postupností A_{i-1} a B_j alebo A_i a B_{j-1}. Teda v tomto prípade je $p[i,j]$ rovné maximu z čísel $p[i-1,j]$ a $p[i,j-1]$. Ak $a[i] = b[j]$, každá vybraná podpostupnosť postupností A_i a B_j je vybranou podpostupnosťou postupností A_{i-1} a B_j alebo A_i a B_{j-1}, alebo vybranou podpostupnosťou postupností A_{i-1} a B_{j-1} s pridaným členom $a[i] = b[j]$. Preto $p[i,j]$ je rovné maximu z čísel $p[i-1,j]$, $p[i,j-1]$ a $p[i-1,j-1]+a[i]$.

Navrhnutý algoritmus má časovú zložitosť $O(M \cdot N)$. Pamäťová zložitosť je tiež $O(M \cdot N)$. Keďže každý riadok tabuľky p závisí iba na predchádzajúcom riadku, stačí si pamätať len posledné dva riadky (pri počítaní riadku $p[i]$ si pamätáme predchádzajúci riadok $p[i-1]$. V programe `p1` značí riadok $p[i-1]$ a `p2` riadok $p[i]$.), pamäťová zložitosť je $O(M+N)$.

Úlohu budeme riešiť v dvoch prechodoch. V prvom prechode nájdeme kandidáta — prvok, ktorý ako jediný môže mať nadpolovičnú väčšinu. V druhom prechode len overíme, či sa tento prvok nachádza v poli a viac ako $\frac{1}{2}N$-krát.

Kandidáta budeme hľadať nasledovným spôsobom: pre každý prvok k, ktorý sa v poli a aspoň raz vyskytuje, si budeme počítať jeho *silu* s_k. Na začiatku položme silu všetkých prvkov rovnú 0. Silu prvkov budeme meniť takým spôsobom, aby v každom okamihu bola nenulová pre nanajvýš jeden prvok. Tento prvok nazvime kandidátom a označme ho K. Ak majú všetky prvky silu nulovú, kandidátom je buď prvok $a[1]$ (pred začiatkom výpočtu), alebo prvok, ktorý bol kandidátom v predchádzajúcom kroku.

Pri spracovávaní prvku $a[i]$ môžu nastať tieto situácie:

1. $K = a[i]$, t.j. ďalší spracovávaný hlas patrí kandidátovi. Zvýšime s_K o 1.
2. $K \neq a[i]$, $s_K > 0$. spracovávaný hlas nepatrí kandidátovi, preto znížime s_K o 1.
3. $K \neq a[i]$, $s_K = 0$. Zvýšime silu $s_{a[i]}$ prvku $a[i]$ o 1. Tým sa prvok $a[i]$ stane novým kandidátom K.

Tento postup opakujeme, kým nespracujeme všetky prvky poľa.

Je zrejmé, že si netreba pamätať silu všetkých prvkov, stačí si pamätať silu kandidáta a to, ktorý prvok je kandidátom. Na to nám stačia dve premenné typu `integer`.

Lemma. *Nech sa nejaký prvok K vyskytuje v poli a M-krát, kde $M > \frac{1}{2}N$. Potom po spracovaní všetkých prvkov poľa bude K kandidátom so silou $s_K \geqq 2M - N > 0$.*

Označme počet zvýšení sily kandidáta K ako k_+, počet znížení jeho sily k_-, počet zvýšení sily ľubovoľného iného kandidáta l_+ a počet znížení sily iných kandidátov l_-. Zníženie sily kandidáta K, ako aj zvýšenie sily iného kandidáta je spôsobené jedine výskytom prvku rôzneho od K. Takýchto operácií teda bude najviac $N - M$. Každý výskyt prvku K spôsobí buď zvýšenie sily K, alebo zníženie sily iného kandidáta (prvky rôzne od K si však môžu znižovať silu aj navzájom), preto týchto operácií bude najmenej M.

$$\begin{aligned} N - M &\geqq k_- + l_+, \\ k_+ + l_- &\geqq M, \\ l_+ - l_- &\geqq 0. \end{aligned}$$

Posledná nerovnosť vyplýva z toho, že počet znížení u žiadneho prvku nepresiahne počet zvýšení. Po sčítaní nerovností a úprave dostaneme

$$k_+ - k_- \geqq 2M - N > 0,$$

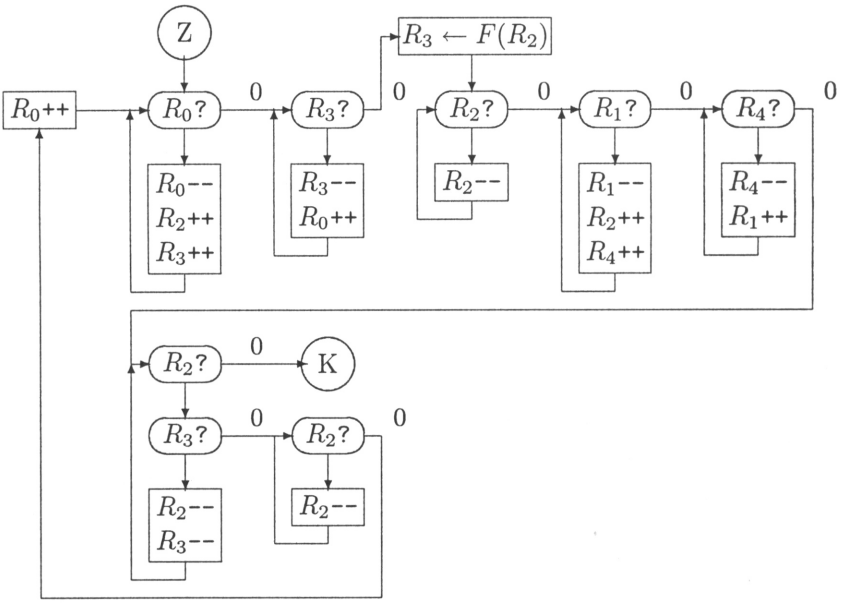
a teda po skončení algoritmu má prvok K kladnú silu. To je možné len tak, že bude na konci kandidátom.

Časová a pamäťová zložitosť: algoritmus vyžaduje dva prechody poľom, každý z nich je v čase $O(N)$. Pamäťová zložitosť je konštantná.

P – II – 4

Časť A. Stroj, riešiaci túto úlohu bude postupne skúšať ako x čísla $0, 1, 2, \ldots$ Vypočíta funkčnú hodnotu $F(x)$ v každom z týchto čísel a porovná ju s hodnotou y v registri R_1. Ak je väčšia alebo rovná, máme riešenie, ak nie, zvýšime hodnotu x o 1 a pokračujeme. V registri R_0 budeme uchovávať hodnotu x, v registri R_1 bude hodnota y. Funkčnú hodnotu $F(x)$ uložíme do registra R_3. Registre R_2 a R_4 slúžia ako pomocné registre. Jeden cyklus stroja sa bude skladať z týchto operácií:

- ▷ skopírovanie obsahu registra R_0 do registra R_2, ktorý bude slúžiť ako vstupný register pre výpočet funkcie F. Najprv presunieme obsah registra R_0 do registrov R_2 a R_3 (za každé zníženie registra R_0 raz zvýšime každý z registrov R_2 a R_3), potom presunieme naspäť obsah R_3 do registra R_0.
- ▷ Použijeme inštrukciu na výpočet funkcie F. R_2 je vstupný register, výsledok sa uloží do výstupného registra R_3. Po vykonaní tejto inštrukcie je obsah registra R_2 nedefinovaný, preto ho musíme vynulovať.
- ▷ Prekopírujeme obsah registra R_1 do registra R_2 za pomoci registra R_4 rovnakou technikou ako v prvom kroku.
- ▷ Porovnáme veľkosti čísel uložených v registroch R_2 a R_3. Postupne budeme od oboch registrov odpočítavať jednotku, až kým sa nám jeden z registrov nevynuluje. Ak sa prvý vynuluje register R_2 (alebo sa vynulujú oba registre naraz), znamená to, že $y \leqq F(x)$. Keďže sme hodnoty x skúšali od najmenšieho, je to najmenšie x s touto vlastnosťou. Hodnota x je prezieravo uložená v registri R_0, takže môžeme skončiť. Ak sa naopak prvý vynuluje register R_3, znamená to, že $F(x) < y$, a preto musíme pokračovať v cykle. Vynulujeme register R_2 a vrátime sa na začiatok.



Časť B. Činnosť stroja je založená na jednoduchej myšlienke: vstupné číslo n si skopírujeme do pomocného registra, vypočítame číslo, ktoré vznikne obrátením binárneho zápisu n (označme ho n_2^R). Potom porovnáme obe tieto čísla. Ak sú rovnaké, odpoveď stroja bude 1, v opačnom prípade bude odpoveď 0.

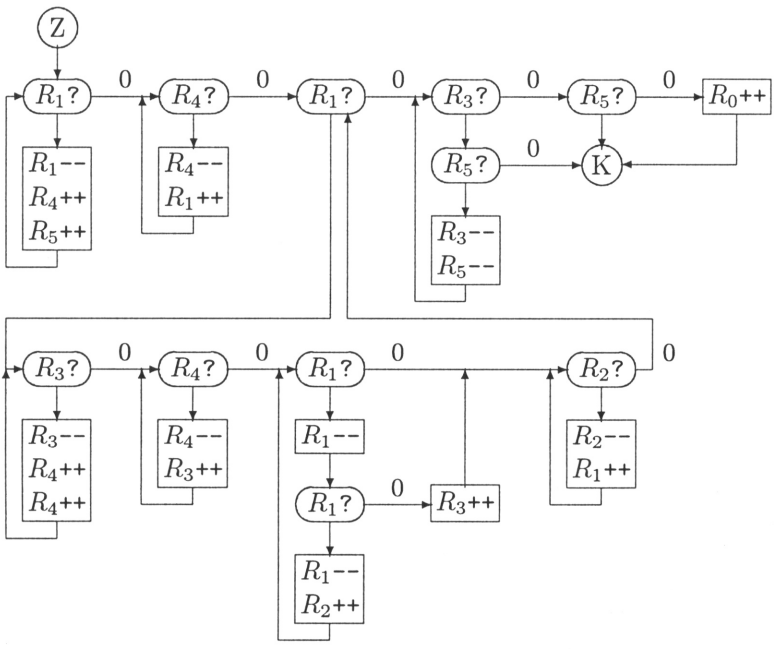
Poďme sa na činnosť stroja pozrieť trochu bližšie. Najprv obsah registra R_1, obsahujúceho vstupné číslo n prekopírujeme za pomoci registra R_4 do registra R_5. Potom budeme v cykle v registri R_3 vyrábať číslo, ktoré vznikne otočením binárneho zápisu čísla n. Nech $n = 2^k b_k + 2^{k-1} b_{k-1} + \ldots + 2^0 b_0$ je binárny zápis čísla n. Po i prechodoch cyklu $(0 \leqq i \leqq k+1)$ bude platiť:

$$R_1 = 2^{k-i} b_k + 2^{k-i-1} b_{k-1} + \ldots + 2^0 b_i,$$
$$R_3 = 2^{i-1} b_0 + 2^{i-1} b_1 + \ldots + 2^0 b_{i-1}.$$

Na začiatku teda $R_1 = n$, $R_3 = 0$. V jednom prechode cyklom najprv vynásobíme obsah R_3 dvoma, potom vydelíme obsah R_1 dvoma. Obe tieto operácie sa dajú realizovať s jedným pomocným registrom tak, že na každé zníženie obsahu R_3 dvakrát zvýšime obsah pomocného registra, resp. na každé dve zníženia R_1 raz zvýšime obsah pomocného registra. Potom stačí len presypať obsah pomocného registra naspäť do R_3,

resp. R_1. Ak nám pri delení vznikne zvyšok, vieme, že posledná cifra zápisu R_1 bola 1, a preto nastavíme poslednú cifru aj číslu v registri R_3 (t.j. pripočítame k R_3 jednotku).

Keď je v registri R_1 nula, zrejme platí $i > k$, a teda v R_3 je číslo n_2^R. Posledná vec, ktorú treba urobiť, je porovnať obsah registrov R_3 a R_5. Budeme postupne znižovať obsah oboch registrov o 1, až kým sa jeden z nich nebude nulový. V prípade, že je aj druhý nulový, zvýšime obsah registra R_0 (lebo n je palindróm). V opačnom prípade zanecháme v registri R_0 nulu a skončíme.



P – III – 1

Vytvoríme graf G, ktorého vrcholy sú križovatky a medzi vrcholmi i a j vedie hrana práve vtedy, ak sú križovatky i a j spojené cestou. Hranu, ktorej odobratie spôsobí rozpadnutie grafu na dve časti (komponenty súvislosti), nazývame *most*. Ukážeme, že mosty sú práve tie hrany, ktoré musia zostať obojsmerné. Zrejme hranu e z vrchola u do v, ktorá je mostom, nemôžeme orientovať (ak sme ju zorientovali povedzme z u do v,

nedalo by sa dostať z v do u, pretože e je most). Z algoritmu vyplynie, že všetky ostatné hrany môžu byť „zjednosmernené".

Algoritmus je založený na myšlienke prehľadávania grafu do hĺbky. Prehľadávanie do hĺbky je rekurzívna procedúra s jediným parametrom — vrcholom v. Tento vrchol označíme za už prehľadaný a rekurzívne voláme tú istú procedúru pre všetkých ešte neprehľadaných susedov vrchola v. Volajme týchto susedov *potomkami* vrchola v. Nazvime *hlavnou* každú hranu, ktorá vedie z predka do niektorého z jeho potomkov, a hrany, ktoré nie sú hlavné, volajme *spätné*.

Pre naše účely priradíme naviac pri prehľadávaní každému vrcholu v poradové číslo c_v označujúce, koľkáty v celkovom poradí bol vrchol v prvýkrát objavený. Zároveň každú ešte neorientovanú (obojsmernú) hranu orientujeme („zjednosmerníme") smerom od vrchola v. Orientáciu už orientovaných hrán nemeníme.

Ukážeme, ako sa dá tento algoritmus použiť na nájdenie mostov v grafe. Aby sme zistili, či je hrana z u do v most, potrebujeme overiť, či sa dá dostať z v do u po hranách rôznych od hrany (u, v). Ak sa dá, hrana (u, v) mostom byť nemôže. Naopak, ak sa nedá, hrana (u, v) je most. Keďže každý most je hlavnou hranou, predpokladajme, že v je potomok u, teda prehľadávacia procedúra pre v bola spustená v prehľadávacej procedúre pre u. Nech w je vrchol s najmenším číslom c_w taký, do ktorého sa dá dostať z vrchola v po orientovaných hranách.

Ak je hrana (u, v) most, pre každý vrchol w' objavený volaním prehľadávacej procedúry z v platí $c_{w'} \geqq c_v$. Zároveň žiaden z vrcholov, do ktorých sa vieme prehľadávaním z v dostať (nepoužívame hlavné, t.j. už orientované, hrany), nemohol byť v okamihu volania procedúry pre v objavený. Teda $c_w \geqq c_v$.

Naopak, predpokladajme, že $c_v \leqq c_w$. Označme P množinu prehľadaných vrcholov tesne pred spustením procedúry pre v a Q množinu vrcholov, ktoré objavíme touto procedúrou. Okrem hrany (u, v) nemôže viesť žiadna hlavná hrana z vrchola z P do vrchola z Q, ani naopak. (Ak by viedla z P do Q, vrchol v Q by bol už prehľadaný pred volaním procedúry, čo je spor. Ak by viedla z Q do P, táto hrana by nemohla byť hlavná, pretože vedie do už objaveného vrchola — opäť spor.) Ak $c_v \leqq c_w$, neexistuje žiadna spätná hrana z Q do P (inak by platilo $c_v > c_w$), a teda neexistuje žiadna hrana medzi P a Q okrem (u, v). Z toho vyplýva, že hrana (u, v) je most.

Zostáva nám ukázať, že hrany, ktoré nie sú mostami, sú orientované vhodne, t.j. že je možné prejsť z každého vrchola do ľubovoľného iného

vrchola dodržujúc orientáciu hrán. Predpokladajme, že graf G nemá mosty. Nech u a v sú také vrcholy, že $c_u = 1$ a $c_v = 2$. Z predošlého vieme, že pre hlavnú hranu (u, v), ktorá nie je mostom, platí $c_v > c_w$, teda $c_w = 1$. Teda existuje orientovaný cyklus (postupnosť vrcholov $v_1, \ldots, v_k$ taká, že $v_1 = v_k$ a pre každé $i = 1, \ldots, k-1$ vedie z vrchola v_i do v_{i+1} orientovaná hrana) prechádzajúci vrcholmi u, v. Označme S množinu vrcholov z cyklu. Pre množinu S platí, že sa vieme z každého do každého z jej vrcholov dostať po orientovaných hranách. Ak S obsahuje všetky vrcholy, ukázali sme, že orientácia grafu vyhovuje. V opačnom prípade nech x je vrchol mimo S taký, že existuje vrchol y v S, že z y vedie do x orientovaná hrana. Z x sa dá dostať po orientovaných hranách do niektorého vrcholu z S, pretože inak by bola hrana (y, x) most. Takže sa dá dostať aj z x do u, aj z u do x, a teda ak vrchol x pridáme do S, zostane zachovaná vlastnosť, že z každého do každého vrchola v S sa dá dostať. Induktívne môžme pridať do S všetky vrcholy, z čoho vyplýva, že navrhnutá orientácia G je vhodná.

Analogickú argumentáciu možno použiť aj keď sa v grafe G mosty nachádzajú. Nech G_1 je (neorientovaný) graf, ktorý vznikne z G po odobraní všetkých mostov. Označme $C_1, \ldots, C_l$ komponenty súvislosti G_1 (z každého vrchola v C_i sa dá dostať do každého vrchola z C_i po hranách z G_1, pre $i = 1, \ldots, l$). Existuje aspoň jeden komponent C_i, do ktorého viedol jediný most. (Ak si zostrojíme graf, v ktorom každému komponentu zodpovedá jeden vrchol a dva vrcholy sú spojené hranou práve vtedy, keď medzi zodpovedajúcimi komponentami v G vedie most, tento graf je súvislý a neobsahuje kružnice, musí to byť teda strom. Každý strom má aspoň jeden list.) Pre tento komponent možno použiť argumenty z predchádzajúceho odstavca, C_i z grafu vynechať a induktívne pokračovať v dôkaze pre ostatné C_j.

Implementácia. Pre každý vrchol `v` je v poli `kriz` uložený zoznam jeho susedných vrcholov. Teda každá hrana je v tomto poli uložená dvojnásobne a tieto dve kópie na seba navzájom ukazujú pomocou smerníkov `dual`. Atribút `aktiv` hovorí, či sa dá v tom smere po danej hrane prechádzať (využíva sa pri orientovaní, keď povoľujeme len jeden z dvoch možných smerov hrany). Pole `sus[v]` uchováva počet susedov vrchola `v`, pole `c[v]` obsahuje c_v a v poli `naj[v]` je uložené číslo c_w. Najskôr pomocou procedúry `prehladaj` podľa vyššie uvedeného algoritmu orientujeme všetky hrany grafu, pričom mosty úplne vymažeme (obidvom hranám nastavíme `aktiv` na `false`). Potom vypíšeme všetky aktívne hrany.

Pamäťová zložitosť algoritmu je $O(M+N) = O(M)$. Časová zložitosť

je zhodná so zložitosťou prehľadávania do hĺbky, teda tiež $O(M)$ (po každej hrane prejdeme práve raz).

P – III – 2

Úlohu budeme riešiť analogicky ako v krajskom kole — v prvom prechode nájdeme kandidátov a v druhom overíme pre každého z nich, či sa nachádza v poli a viac ako N/k-krát.

Kandidátov je zjavne najviac $k-1$. Budeme ich hľadať nasledovne: pre každý prvok p, ktorý sa v poli a aspoň raz vyskytuje, si budeme počítať jeho *silu* s_p. Na začiatku položme silu všetkých prvkov rovnú 0. Silu prvkov budeme meniť takým spôsobom, aby v každom okamihu bola nenulová pre nanajvýš $k-1$ prvkov. Tieto prvky nazvime kandidátmi a označme ich $K_1, \ldots, K_{k-1}$.

Pri spracovávaní prvku $a[i]$ môžu nastať tieto situácie:

1. $\exists j$: $K_j = a[i]$, t.j. ďalší spracovávaný hlas patrí niektorému kandidátovi. Zvýšime s_{K_j} o 1.
2. $\forall j$: $K_j \neq a[i]$, $\forall u$: $s_{K_u} > 0$. Spracovávaný hlas nepatrí žiadnemu kandidátovi, preto znížime každému kandidátovi silu o 1.
3. $\forall j$: $K_j \neq a[i]$, $\exists u$: $s_{K_u} = 0$. Zvýšime silu $s_{a[i]}$ prvku $a[i]$ o 1. Preto sa prvok $a[i]$ stane novým kandidátom K_u.

Tento postup opakujeme, kým nespracujeme všetky prvky poľa.

Je zrejmé, že si netreba pamätať silu všetkých prvkov, stačí si pamätať sily $k-1$ kandidátov a to, ktoré prvky sú kandidátmi. Na to nám stačia dve polia veľkosti $O(k)$.

Lemma. *Nech sa nejaký prvok P vyskytuje v poli a M-krát, kde $M > N/k$. Potom po spracovaní všetkých prvkov poľa bude P kandidátom so silou $s_P > 0$.*

Dôkaz. Nazvime operácie 1 a 3 zvýšením a operáciu 2 znížením. Dokážme najskôr, že znížení je najviac N/k. Sporom. Všimnime si, že sila každého prvku je nezáporné celé číslo, teda súčet síl všetkých prvkov je na konci určite nezáporný. Ak by bolo znížení viac ako N/k, (t.j. aspoň $\lfloor N/k \rfloor + 1$) znamenalo by to, že sa celkový súčet síl znížil aspoň o $(\lfloor N/k \rfloor + 1) \cdot (k-1)$, zatiaľ čo sa zvýšil najviac o $N - (\lfloor N/k \rfloor + 1)$. To ale znamená, že súčet síl prvkov na konci je

$$S \leqq N - \left(\left\lfloor \frac{N}{k} \right\rfloor + 1\right) - \left(\left\lfloor \frac{N}{k} \right\rfloor + 1\right)(k-1) = N - k\left(\left\lfloor \frac{N}{k} \right\rfloor + 1\right) < 0,$$

čo je spor, preto je naozaj znížení najviac N/k.

Všimnime si teraz prvok P. Nech jeho výskyt A-krát spôsobil zníženie, B-krát zvýšenie. Opäť sporom. Ukážeme, že $s_P > 0$. Ak by mal prvok P na konci silu 0, znamenalo by to, že bolo aspoň $A + B$ znížení — A-krát ho spôsobil prvok P, B iných znížení muselo prvku P znížiť silu na 0. Počet znížení je teda aspoň $A + B = M > N/k$, čo je spor s vyššie dokázaným tvrdením, že zvýšení je najviac N/k. Preto má prvok P na konci nenulovú silu. To je možné len tak, že bude na konci kandidátom.

Časová a pamäťová zložitosť: algoritmus vyžaduje dva prechody poľom, každý z nich je v čase $O(kN)$. Pamäťová zložitosť je $O(k)$.

P – III – 3

Stroj riešiaci túto úlohu je už pomerne zložitý a veľmi ťažko by sa kreslil naraz, bez toho, aby sme ho rozložili na menšie celky. Predtým ako ho budeme konštruovať, si povedzme, ako by sa takýto problém riešil na normálnom počítači.

Prvé riešenie. Jeden z prístupov by bol backtrackom skúšať všetky možné podmnožiny množiny M, pre každú vypočítať súčet a overiť či sa nerovná danému číslu s. Skončili by sme, ak by sme našli podmnožinu s vyhovujúcim súčtom, alebo keby sme vyskúšali všetky podmnožiny množiny M. Klasické backtrackové riešenie však používa zásobník, ktorý by sme pomocou registrového stroja simulovali len s veľkou námahou. Pekný trik, ako vyskúšať všetky podmnožiny množiny M je takýto: Množina M má kód m. Postupne budeme skúšať všetky také množiny N, ktorých kód n je menší alebo rovný m. Pre každú takúto množinu vypočítame kód prieniku $M \cap N$, čo je vlastne logický súčin (`AND`) po bitoch čísel m a n. Niektoré podmnožiny síce vygenerujeme viackrát, ale to vôbec nevadí. Pre každý prienik (t.j. pre jeho kód m `AND` n) potom spočítame súčet jeho prvkov a skontrolujeme, či sa náhodou nerovná hľadanému súčtu s.

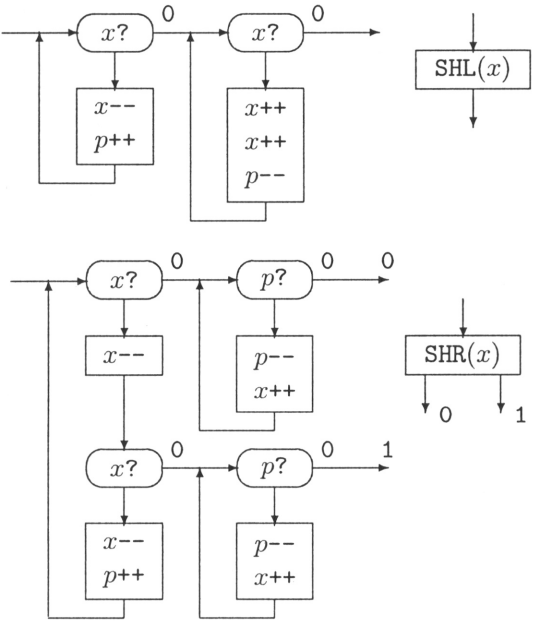
Blok pre bitový logický súčin skonštruujeme podobne ako blok pre logický súčet (`OR`, viď. ďalej). Na výpočet súčtu prvkov v množine môžeme použiť blok `SHR`(x), ktorý zisťuje hodnotu najnižšieho bitu čísla v registri x a zároveň register x celočíselne vydelí dvoma (viď. ďalej) a blok `ADD`(x, y), definovaný v príklade zo zadania.

Druhé riešenie. Ako vzorové uvádzame iné riešenie, ktoré využíva myšlienku dynamického programovania. Postupne budeme budovať množiny súčtov, ktoré sa dajú vytvoriť len z k najmenších prvkov množiny M. Označme tieto množiny $S_0, S_1, \ldots, S_k$ a ich kódy $s_1, s_2, \ldots, s_k$. Jediné

číslo, ktoré sa dá utvoriť súčtom nula prvkov, je číslo 0. Preto $S_0 = \{0\}$ a $s_0 = 1$. Predstavme si, že už máme vytvorenú množinu S_i, a nech $(i+1)$-vý najmenší prvok v množine M je p. Ku každému prvku z množiny S_i pripočítame číslo p. Dostaneme tak množinu S'_i, $S'_i = \{c + p : c \in S_i\}$. Keďže každý súčet z prvých $i+1$ prvkov sa dá dosiahnuť buď s použitím alebo bez použitia prvku p, množina S_{i+1} dostaneme zjednotením množín S_i a S'_i. Kód s'_i množiny S'_i dostaneme veľmi jednoducho: $s'_i = s_i \cdot 2^p$. Zjednotenie množín zase dosiahneme bitovým logickým súčtom ich kódov, t.j. $s_{i+1} = s_i \texttt{OR} s'_i$. Pred tým, ako do detailov rozoberieme činnosť nášho stroja, si definujeme a popíšeme niekoľko blokov.

Všetky popisované bloky pre správnu činnosť predpokladajú, že všetky použité pomocné registre sú pred vstupom do bloku vynulované. Pred výstupom z bloku sa tieto registre opäť vynulujú.

Popis blokov.

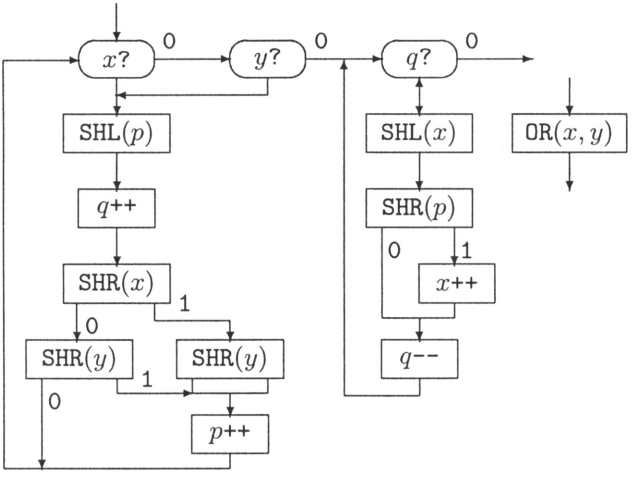


Blok ADD(x, y) bol popísaný a definovaný v zadaniach.

Blok SHL(x) vynásobí číslo v registri x dvoma. Potrebuje jeden pomocný register p, do ktorého „presype“ obsah registra x, potom obsah p presýpa do x, pričom za každé zníženie registra p dvakrát zvýši obsah registra x.

Blok SHR(x) celočíselne delí číslo v registri x dvoma. Má dva výstupy označené 0 a 1, pričom výstup 0 sa použije, ak bolo číslo v registri x pri vstupe do bloku párne a výstup 1, ak bolo nepárne. Pracuje podobne ako blok SHL s tým, že najprv za každé dve zníženia registra x raz zvýšime pomocný register a potom presýpame obsah pomocného registra naspäť do x.

Blok OR(x, y) je najzložitejší. Jeho funkciou je priradiť do registra x bitový OR čísel uložených v registroch x a y a zároveň vynulovať register y. Budeme to robiť podobne ako vo vzorovom riešení krajského kola. Pomocou blokov SHR odoberieme posledný bit zo zápisu oboch čísel x, y. Ak je aspoň jeden z odobratých bitov jednotkový, pridáme na koniec zápisu čísla v pomocnom registri p jednotku (t.j. p vynásobíme dvoma pomocou bloku SHL a potom k nemu pripočítame jednotku), v opačnom prípade pridáme na koniec p nulu (t.j. vynásobíme ho dvoma). Takto sa nám bude v registri p postupne objavovať bitový súčet čísel x a y, avšak s bitmi zapísanými v obrátenom poradí. Na koniec teda musíme obsah registra p otočený zapísať do registra x, čo spravíme opäť odoberaním posledného bitu zápisu p a jeho pridávaním na koniec zápisu x. Aby sme vedeli, koľkokrát máme túto operáciu spraviť, počítame si počet platných bitov registra p v pomocnom registri q.

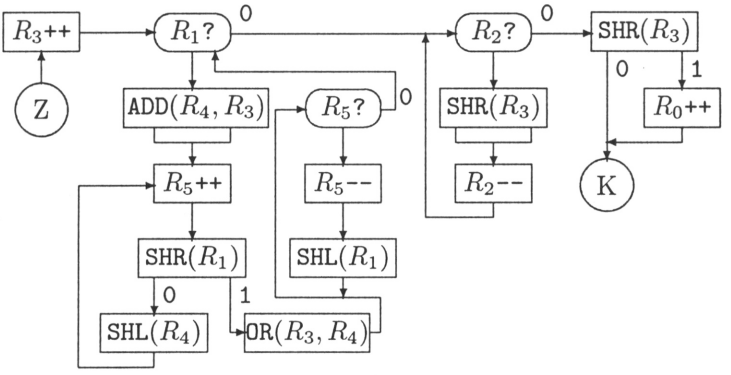


Nakoniec nám zostáva popísať samotný stroj. Skladá sa z dvoch hlavných cyklov. V prvom cykle sa v registri R_3 postupne počítajú kódy mno-

žín S_i, po skončení i-teho cyklu R_3 obsahuje kód s_i. Zároveň sa v každom prechode odoberie z množiny M jej najmenší prvok.

Odoberanie prvku sa robí v dvoch menších cykloch spolu s výpočtom kódu množiny S'_{i-1}, ktorý bude uložený v registri R_4. Na začiatku si kód s_{i-1} skopírujeme z registra R_3 aj do registra R_4. V prvom cykle postupne delíme obsah registra R_1 (t.j. kód množiny M) dvoma a obsah registra R_4 naopak násobíme dvoma, pričom si počítame počet prechodov cyklu v registri R_5. Keď sa nám nepodarí vydeliť obsah R_1 dvoma bezo zvyšku, narazili sme na najmenší prvok. V tomto okamihu máme v R_4 kód S'_{i-1}, ktorý pomocou bloku OR pridáme k obsahu R_3, čím nám v tomto registri vznikne kód množiny S_i. Na záver v druhom cykle po sebe upraceme, t.j. vynásobíme obsah R_1 takou mocninou dvojky, akou sme ho v prvom cykle vydelili. Všimnite si, že týmto postupom sa nám automaticky vynuloval najnižší nenulový bit registra R_1, t.j. odobrali sme najmenší prvok množiny M.

Keď z M odoberieme posledný prvok, jej kód uložený v registri R_1 sa vynuluje a riadenie prejde do druhého hlavného cyklu. V tomto cykle overíme, či číslo s uložené v registri R_2, patrí do množiny, ktorej kód je uložený v registri R_3. Hodnotu R_3 stačí s krát vydeliť dvoma a potom zistiť, či posledný bit registra je 1.



P – III – 4

Smerovník pri prameni číslo i nech ukazuje na prameň číslo $s[i]$, pre $1 \leqq i \leqq N$. Hovoríme, že pramene $p_1, p_2, \ldots, p_k$ tvoria *cyklus*, ak od

prameňa p_1 ukazuje smerovník k prameňu p_2, od p_2 k p_3 a tak ďalej, až od prameňa p_k k prameňu p_1.

Ak vyštartujeme z ľubovoľného prameňa p_1 a sledujúc smerovníky prechádzame postupne pramene $p_2, p_3, \ldots$, po najviac N krokoch sa nám stane, že prídeme k prameňu p_{k+1}, pri ktorom sme už boli, t.j. $p_{k+1} = p_j$ pre nejaké $j \leqq k$. Keďže smerovník ukazujúci na prameň j bol vyrobený iba jeden, musí byť buď $p_k = p_{j-1}$, alebo $j = 1$. Ak je k najmenšie také, že $p_{k+1} = p_j$, $j \leqq k$, potom $j = 1$, a teda pramene $p_1, \ldots, p_k$ tvoria cyklus. Takýmto spôsobom vieme pre každý prameň určiť cyklus, do ktorého patrí.

Vzorový program využíva fakt, že ak vymeníme smerovníky pri dvoch prameňoch, ktoré patria do rovnakého cyklu, tento cyklus sa nám rozdelí na dva, t.j. počet cyklov sa zvýši o 1. Ak naopak vymeníme smerovníky pri prameňoch z rôznych cyklov, tieto dva cykly sa spoja do jedného.

Označíme si pramene, ktoré patria do toho istého cyklu ako prameň 1. Postupne budeme hľadať ďalšie cykly, ktoré budeme pripájať k cyklu obsahujúcemu prameň 1. Vždy, keď nájdeme nový cyklus, označíme si všetky pramene, ktoré doňho patria a zároveň spravíme výmenu smerovníkov, ktorou sa tento cyklus pripojí k cyklu obsahujúcemu prvý prameň. Všimnime si, že novovzniknutý cyklus obsahuje práve doposiaľ označené pramene. Ďalší cyklus potom hľadáme medzi neoznačenými prameňmi.

Implementácia. Na označovanie prameňov používame pole s. Prameň j považujeme za označený, ak $s[j] \leqq 0$. Nový cyklus začíname hľadať od takého neoznačeného prameňa i, že všetky pramene s číslom menším ako i sú označené. Prameň i ako reprezentanta nového cyklu označíme $s[i] = -1$, ostatné pramene cyklu označíme nulou. Keď pri prechádzaní cyklom opäť natrafíme na prameň i, cyklus sme uzatvorili. Rovnakým postupom hľadáme ďalší cyklus, pričom vieme, že každý prameň na tomto cykle má číslo väčšie ako i.

Na záver vymeníme jeden smerovník z každého cyklu so smerovníkom pri niektorom prameni z cyklu obsahujúceho prameň 1, t.j. postupne vymieňame smerovník pri prameni 1 so smerovníkmi pri prameňoch označených -1 (reprezentanti cyklov). Potrebný počet výmen je o jednotku menší ako počet cyklov.

Časová a pamäťová zložitosť. Pamäťová zložitosť je zrejme $O(N)$. Časová zložitosť algoritmu je tiež $O(N)$, pretože s každým prvkom poľa s pracujeme maximálne trikrát (dvakrát pri hľadaní cyklov a na záver robíme ešte jeden prechod poľom s).

Úlohu zo zadania si trochu zjednodušme a venujme sa len podstate úlohy. Je jasné, že ak sa z akéhokoľvek dôvodu budeme presúvať medzi dvoma mestami, tak optimálne bude, ak sa medzi nimi budeme presúvať najkratšou možnou cestou. Nech teda $h_{i,j}$ $(1 \leqq i, j \leqq N)$ označuje dĺžku najkratšej cesty medzi mestom i a mestom j. Riešenie podúlohy, ako zistiť dĺžky najkratších ciest je uvedené o pár odstavcov nižšie.

Takto zredukovanú úlohu budeme riešiť metódou dynamického programovania. Označme $E_{i,j}$ $(1 \leqq i \leqq N,\ 0 \leqq j \leqq K)$ *dĺžku* najkratšej trasy končiacej v deň j v meste i, takej, že sme videli filmy $p_1, p_2, \ldots, p_j$ a pritom sme posledný film p_j videli v meste i. Ak hodnota $E_{i,j}$ neexistuje (t.j. neexistuje trasa popísaných vlastností), položíme $E_{i,j} = \infty$.

Hodnoty $E_{i,j}$ budeme postupne počítať z iných skôr vypočítaných hodnôt $E_{i,j}$ a z hodnôt $h_{i,j}$. Začneme zrejme takto: $E_{1,0} = 0$ a $E_{i,0} = \infty$ $(2 \leqq i \leqq N)$.

Počítajme hodnotu $E_{i,j}$ pre $1 \leqq j \leqq N$. Máme dve možnosti: V meste i nehrajú film p_j. Trasa požadovaných vlastností končiaca v meste i neexistuje, preto $E_{i,j} = \infty$.

Druhá možnosť je, že film p_j v meste i hrajú. Rozoberme túto možnosť. Na to, aby sme videli filmy $p_1, p_2, \ldots, p_j$ sme museli vidieť filmy $p_1, p_2, \ldots, p_{j-1}$. Film p_{j-1} sme mohli vidieť v nejakom meste s. Do mesta s sme sa pritom zrejme dostali najkratšou trasou, končiacou v tomto meste. Dĺžka tejto trasy je $E_{s,j-1}$. Z mesta s do mesta i sme takisto museli ísť najkratšou cestou. Teda dĺžka takejto trasy bude $E_{s,j-1} + h_{s,i}$. Prostým vyskúšaním všetkých možných miest s dostaneme dĺžku najkratšej cesty $E_{i,j}$:

$$E_{i,j} = \min\{E_{s,j-1} + h_{s,i} \colon 1 \leqq s \leqq N\}.$$

Najkratšia trasa, potrebná na zhliadnutie všetkých K filmov končí v nejakom meste i. Stačí nám teda vybrať z dĺžok trás, ktoré končia v jednotlivých mestách tú najmenšiu. Pre dĺžku optimálnej trasy E teda platí

$$E := \min\{E_{i,K} \colon 1 \leqq i \leqq N\}.$$

Ostáva nám ešte zistiť, cez ktoré mestá vlastne optimálna trasa vedie. Označme $D_{i,j}$ mesto, z ktorého sme prišli v deň j do mesta i, ak by sme išli po optimálnej trase končiacej v meste i v deň j. Hodnotu $D_{i,j}$ budeme počítať súbežne s hodnotou $E_{i,j}$. $D_{i,j}$ bude vlastne to mesto s, pre ktoré bude $E_{s,j-1} + h_{s,i}$ minimálne.

Mesto, v ktorom končí hľadaná optimálna trasa (t.j. také, pre ktoré je $E_{i,K}$ minimálne) označme x_K. Potom predchádzajúce mesto na optimálnej trase bude $x_{K-1} = D_{x_K,K}$. Vo všeobecnosti mesto na optimálnej trase, z ktorého sme prišli do mesta x_i bude mesto $x_{i-1} = D_{x_i,i}$.

Nakoniec venujme pár slov otázke, ako vzdialenosti $h_{i,j}$ ($1 \leqq i, j \leqq N$) medzi jednotlivými mestami počítať. Použijeme štandardný Floyd-Warshallov algoritmus. Vstupom tohto algoritmu je matica $h_{i,j}$ ($1 \leqq i, j \leqq N$), obsahujúca dĺžky ciest, spájajúcich jednotlivé dvojice miest (pre cestu vedúcu medzi mestami i a j s dĺžkou l položíme $h_{i,j} = h_{j,i} = l$, ak medzi i a j nevedie žiadna cesta, položíme $h_{i,j} = h_{j,i} = \infty$). Výstupom algoritmu je matica h, v ktorej $h_{i,j}$ je minimálna vzdialenosť, ktorú musíme precestovať, aby sme sa dostali z mesta i do mesta j.

Algoritmus bude pracovať v N cykloch. Po vykonaní k-teho cyklu ($0 \leqq k \leqq N$) bude platiť, že $h_{i,j}$ je dĺžka najkratšej trasy z mesta i do mesta j, ktorá prechádza len cez mestá s číslom menším alebo rovným k. Na začiatku (t.j. po vykonaní 0 cyklov) je v $h_{i,j}$ uložená dĺžka priamej trasy bez prechádzania cez iné vrcholy. Pri vykonávaní k-teho cyklu, dĺžka trasy $h_{i,j}$ môže byť buď rovnaká ako v predchádzajúcom cykle (ak nevyužijeme možnosť viesť trasu z mesta i do mesta j cez mesto k), alebo rovná $h_{i,k} + h_{k,j}$ (ak trasu z i do j vedieme cez mesto k). Vždy si samozrejme vyberieme kratšiu možnosť.

Implementácia. Dĺžky ciest načítavame priamo do poľa `h`, v ktorom aj počítame vzdialenosti medzi mestami F-W algoritmom. Na výpočet si nepotrebujeme pamätať všetky hodnoty $E_{i,j}$, stačí si pamätať iba dva stĺpce pre j a $j + 1$. To robíme v poli `optim`, pričom `optim[0]` obsahuje hodnoty $E_{i,j}$ pre j párne a `optim[1]` pre j nepárne. Ak však chceme zrekonštruovať optimálnu trasu, potrebujeme si pamätať aspoň hodnoty $D_{i,j}$. Na tie nám však stačí typ `byte`. Hodnoty $D_{i,j}$ máme v poli `pred`, ktoré je vo vzorovom programe alokované dynamicky, aj keď obmedzenia v zadaní dovoľovali použiť statické pole.

Časová a pamäťová zložitosť. Časová zložitosť celého algoritmu bude $O(N^3 + KN^2)$, z toho $O(N^3)$ je Floyd-Warshallov algoritmus. Pamäťová zložitosť bude $O(N^2 + KN)$, kde $O(N^2)$ pamäti zaberá matica h a $O(NK)$ zaberajú matice E a D.

Poznámka. Existuje algoritmus, ktorý nepotrebuje úvodné predvypočítanie vzdialeností F-W algoritmom a ktorý na výpočet každého stĺpca matice E používa modifikáciu Dijkstrovho algoritmu. Tento algoritmus má časovú zložitosť $O(K(M \log N))$, resp. $O(KN^2)$ (podľa implementácie Dijkstrovho algoritmu) a pamäťovú zložitosť $O(M + NK)$.