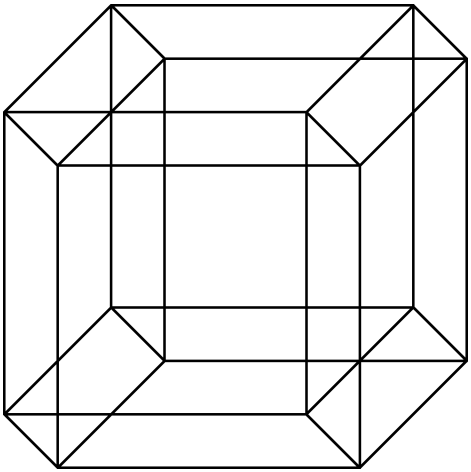
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

VIII. ročník – 1995/96



MATFYZPRESS
Vydavatelství
Matematicko-fyzikální fakulty
University Karlovy

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

VIII. ročník – 1995/96

Matfyzpress
Vydavatelství
Matematicko-fyzikální fakulty
University Karlovy

ISBN-00-00000-00-0

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož osmý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF UK pro zájemce o informatiku a programování z řad studentů středních škol. Řešíce úlohy našeho semináře, středoškoláci získávají praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé discipliny informatiky. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy, MFF UK z toho nevyjímaje. To ovšem neznamená, že nemá vůbec smysl takové problémy řešit – při troše přemýšlení není příliš obtížné nějaké (i když někdy ne to nejlepší) řešení nalézt. Nakonec – posuďte sami.

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání několika (čtyř či pěti) úloh, v klidu domácího krbu je (ne nutně všechny) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na níže uvedenou adresu. My je poté (víceméně obratem) opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

Za jeden školní rok obvykle proběhnou čtyři série, v letech hojnějších pak pět. Závěrečným bonbónkem je pak pravidelné *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě).

Náš korespondenční seminář není ojedinělou aktivitou svého druhu v Evropě – existují korespondenční semináře z fyziky a matematiky při MFF UK, jakož i jiné programátorské semináře (kupříkladu brněnský a bratislavský). Rozhodně si však nekonkurujeme, právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

Velice rádi vám odpovíme na libovolné dotazy ohledně studia informatiky na naší fakultě, jakož i jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@ksvi.mff.cuni.cz*

Zadání úloh

8-1-1 Rozděl a panuj — 10 bodů

Zlatá horečka postihla skupinu N zlatokopů, kteří se vydali rýžovat zlato na Aljašku. Rýžovali a rýžovali a rýžovali, až nakonec získali N^2 valounů zlata v hodnotě od 1 do N^2 tolarů.

Nu a přišlo dělení a jelikož zlatokopové jsou horké hlavy a jsou trochu hloupí, je potřeba zlato mezi ně spravedlivě rozdělit tak, aby každý dostal zlato se stejnou hodnotou a aby měl každý též stejný počet valounů.

Navrhněte tedy algoritmus a napište program, který pro dané N najde rozdělení hodnot 1 až N^2 do N skupin tak, aby součet hodnot byl ve všech skupinách stejný.

8-1-2 Sedlový bod — 10 bodů

Mějme matici A celých čísel o velikosti $M \times N$. Sedlovým bodem takovéto matice je prvek a_{ij} takový, že je maximální ve svém sloupci a minimální ve svém řádku. Řečeno slovy matematika:

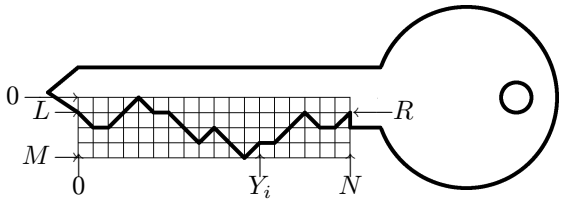
$$\text{pro všechna } k, l \ (1 \leq k \leq M, 1 \leq l \leq N) \text{ platí: } a_{kj} \leq a_{ij} \leq a_{il}.$$

Navrhněte algoritmus a napište program, který pro zadanou matici určí, zda má nějaký sedlový bod a v případě, že ano, vypíše souřadnice alespoň jednoho sedlového bodu.

8-1-3 Klíč — 13 bodů

Všichni jistě víte, jak vypadá klíč. Klíč je popsán celými čísly N, M, L, R a $(N+1)$-ticí nezáporných celých čísel $Y_0, \ldots, Y_N$, pro kterou platí:

- $Y_0 = L$, $Y_N = R$
- pro všechna i, $0 \leq i \leq N$ je $0 \leq Y_i \leq M$
- pro všechna i, $1 \leq i \leq N$ je $\|Y_i - Y_{i-1}\| \leq 1$



A teď úloha: Navrhněte algoritmus a napište program, který určí počet všech různých klíčů, je-li dáno M, N, L, R.

Poznámka: Tato úloha pochází původně z dílny zámečnického mistra Čílka, ovšem v současné době se touto otázkou zabývají i některé skupiny z podsvětí.

8-1-4 MIDI — 11 bodů

MIDI (Musical Instrument Digital Interface) je standard pro komunikaci mezi počítačem a elektronickými hudebními nástroji. Část tohoto standardu definuje příkazy, které ovládají zapínání a vypínání jednotlivých tónů na hudebním nástroji. Tyto příkazy se sestavují do sekvencí (programů), kdy každý příkaz má vyznačen čas, ve kterém se má provést. Jednotlivé příkazy jsou uspořádány vzestupně podle svého času.

Pro zapnutí resp. vypnutí generování tónu slouží příkaz `ON` resp. `OFF`, za nímž následuje číslo noty, jíž se to týká.

Například následující MIDI program odpovídá současnému spuštění tří tónů (60, 70 a 80) po dobu 10 časových jednotek a následného spuštění tónu 62 po dvě časové jednotky:

```
 0 ON  60
 0 ON  70
 0 ON  80
10 OFF 60
10 OFF 80
10 OFF 70
10 ON  62
12 OFF 62
```

Většina existujících skladeb nemůže být přímo přepsána do MIDI programu, neboť občas je již daný tón spuštěn, když se v původní skladbě požaduje jeho opětné zaznění. Například:

```
 0 ON  60
10 ON  60
12 OFF 60
20 OFF 60
```

Syntezátor bude tento program interpretovat tak, že nechá zaznít tón 60 na 12 časových jednotek a ne na 20, jak jsme předpokládali. Neuslyšíme tedy oddělené zaznění dvou tónů 60, neboť zapnutí tónu, který je již zapnut, nemá žádný efekt.

Pokud je tedy tón již zapnut a v programu je příkaz `ON`, aby tón zazněl znovu, je třeba do programu vložit vypnutí onoho tónu jednu časovou jednotku před tímto druhým spuštěním. Zároveň je potřeba odstranit příkaz pro vypnutí tohoto tónu, který následuje jako nejbližší. Takový program bude již interpretován jako dvojité následné zaznění tohoto tónu.

Dalším problémem je vypnutí a zapnutí tónu ve stejný časový okamžik. V tomto případě záleží na pořadí příkazů `ON` a `OFF` uvedených v programu, zda tón bude znít nebo ne. Například srovnej následující programy:

```
 0 ON  60      0 ON  60
10 ON  60     10 OFF 60
10 OFF 60     10 ON  60
20 OFF 60     20 OFF 60
```

V levém programu se tón 60 vypne již v čase 10. V pravém programu je naproti tomu zase čas, kdy je tón vypnut, nedostatečný, aby jej lidské ucho postřehlo. V obou případech se pro úpravu programu použije stejné řešení – přesune se první příkaz `OFF` o 1 časovou jednotku před druhé `ON`.

Pokud `OFF` vložené jednu časovou jednotku před `ON` nastává ve stejný okamžik jako jiné `ON`, pak vložené `OFF` a po něm následující `ON` budou z programu zcela vyjmuty.

Vaším úkolem je napsat program, který přečte MIDI program v textové podobě a upraví jej dle námi popsaných pravidel.

8-1-5 Konečné automaty — 10 bodů

Do tohoto ročníku KSP jsme zařadili několik na sebe navazujících úloh zabývajících se *konečnými automaty.* Ti z vás, kteří nevědí, co to konečný automat je, naleznou podrobné vysvětlení na straně 18.

Sestrojte konečné automaty, které rozeznávají následující jazyky:

$L_1 = \{w \in \{\mathbf{a}, \mathbf{b}\}^* \,;\; w = x\mathbf{baba} \;\&\; x \in \{\mathbf{a}, \mathbf{b}\}^*\}$
$L_2 = \{w \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}^* \,;\; w$ je ostře rostoucí posloupnost $\}$

Do jazyka L_1 tedy patří všechna slova, která končí na "**baba**". Do jazyka L_2 například patří 135, 12, e, nepatří do něj např. 534, 12321, 1223.

Poznámka: Sestrojením konečného automatu rozumíme vypsání složek pětice a nakreslení obrázku podle uvedených pravidel.

8-2-1 O líných novinářích — 12 bodů

Lidé jsou leniví a tato vlastnost se nevyhýbá ani mnohým novinářům. Někteří novináři jsou dokonce tak líní, že do svých článků píší některé pasáže vícekrát, aby lehce dosáhli požadované délky textu. Šéfredaktor proto potřebuje program, který pro daný text článku zjistí nejdelší úsek, jenž se v něm opakuje.

Navrhněte tedy algoritmus a napište program, který pro daný text délky maximálně 1000 znaků zjistí nejdelší úsek, v něm se opakující. V případě, že maximálních podúseků je více, nalezněte všechny.

8-2-2 Rozmarná princezna — 11 bodů

V jednom království si rozmarná princezna vymyslela, že si každý večer s někým zahraje hru *nim* a pokud dotyčný prohraje, půjde spát na noc do chlívku. Přes týden, kdy byli protihráči z práce unaveni, se hrála jednodušší varianta, v neděli pak těžší.

Jednodušší varianta hry má tato pravidla: na stůl se položí n sirek, kde n je nějaké kladné celé číslo. Poté střídavě hráči odebírají jednu, tři nebo pět sirek. Kdo odebere poslední sirku, vyhrál.

Těžší varianta má obdobná pravidla, ovšem lze odebrat pouze počet sirek, který odpovídá libovolné mocnině čísla 2, tj. 1, 2, 4, 8, ...

Zda začíná odebírat princezna nebo její protihráč, určí se losem.

Naším úkolem je pomoci protihráči princezny, tj. pro oba druhy hry *nim* navrhnout vhodnou strategii hry a napsat program, který se jí bude řídit tak, aby protihráč, pokud je to možné, vyhrál.

8-2-3 U zeleného stromu — 10 bodů

V holičství "U zeleného stromu" bylo $2N + 1$ stoliček, na nichž sedělo N trpaslíků a N hobitů a zbylá stolička byla prázdná. Stoličky byly vyrovnány do řady a očíslovány čísly 1 až $2N + 1$. Holič bral na ostříhání zákazníky v pořadí, v jakém seděli na stoličkách.

Ježto hobiti jsou známí svou mírnou povahou a tím, že nikam nespěchají, požádal občas holič čekající zákazníky, aby si povyměňovali místa tak, aby všichni trpaslíci seděli na stoličkách před hobity. A aby během vyměňování nevznikl zmatek, muselo to probíhat tak, že se vždy někdo zvedl a sedl si na volnou stoličku. V okamžiku, kdy bylo dosaženo kýženého cíle, tedy trpaslíci seděli před hobity, pokračoval holič ve své práci. Jelikož trpaslíci byli nedočkaví, přemýšleli, jaké musí být pořadí při vyměňování, aby celá operace proběhla na co nejmenší počet přesunů.

A právě vaším úkolem je navrhnout algoritmus a napsat program, který pro dané rozmístění N hobitů a N trpaslíků na $2N + 1$ stoličkách určí, v jakém pořadí má dojít k výměnám, aby byl počet výměn nejmenší možný.

8-2-4 Náhrdelník — 10 bodů

Rozmarná princezna z předešlé úlohy nosí na krku náhrdelník z perel. Jelikož se jí často stane, že se jí při hře na zahradě náhrdelník rozsype, očíslovala si perly čísly od jedné do N. Vždy když se jí pak náhrdelník rozsype, musí sluhové perly posbírat, zjistit, zda nějaká nechybí a pak jej znovu svázat. Často se stane, že chybí právě jedna perla. Pak ji musí sluhové hledat tak dlouho, dokud ji nenajdou, přičemž hledají perlu s konkrétním číslem a jiné nalezené perly si mohou ponechat.

Vaším úkolem je navrhnout algoritmus a napsat program, který pro dané N a *neuspořádanou* posloupnost čísel jedna až N zjistí, zda některé číslo v posloupnosti chybí a v případě, že ano, tak které. Chybějící číslo je maximálně jedno a žádné číslo se v posloupnosti neopakuje.

Příklad: $N = 7$, posloupnost $\{4, 2, 3, 5, 1, 7\}$. Chybí perla číslo 6.

8-2-5 Konečné automaty — 10 bodů

Připravili jsme pro vás další dva jazyky, pro které byste měli sestrojit konečné automaty. Jeden z nich je o něco snazší, druhý z nich je obtížnější.

$L_1 = \{w \in \{\mathbf{a}, \mathbf{b}\}^* \,;$ slovo w neobsahuje **baba** $\}$
$L_2 = \{w \in \{\mathbf{a}, \mathbf{b}\}^* \,;$ slovo w obsahuje **baba** nebo **abba** $\}$

Příklad: První jazyk například neobsahuje slova **baba**, **ababab**, **bbbbbababbbbb** a další, druhý obsahuje slova **baba**, **abba**, **bababba**, **aaaabba** a další.

8-3-1 Kouzelné zrcadlo — 13 bodů

Byla jednou jedna docela obyčejná královna, která vlastnila jedno ne zcela obyčejné zrcadlo a z poměrně nejasných pohnutek po něm stále žádala, aby jí ukázalo, kdo že je to na tom světě nejkrásnější, načež se divila, že se zrcadlo nechová tak, jak se od běžného zrcadla obvykle očekává. Jistě nemusíme pokračovat – tuto pohádku dozajista všichni dobře znáte.

Napište program, který se bude chovat přesně jako dotyčná královna a pro daný text na vstupu rozhodne, je-li to jeho vlastní zdrojový text (v takovém případě odpoví něco ve stylu "Tak přeci je to zrcadlo kouzelné!", v případě opačném pak "Xakru, zase to nefunguje!"). Program ovšem nesmí používat žádné soubory.

8-3-2 Magické čtyřky — 13 bodů

$\mathcal{X}$ byl profesionální mág a současně matematik amatér. A nebo opačně – vyberte si. Vzhledem k tomu, že měl velice omezené možnosti, počítal pouze s malými celými čísly v rozsahu 0 až $p-1$ (za p si z nějakého rozmaru vybíral pouze prvočísla) a zavedl si s nimi následující operace:

$$a \oplus b = (a + b) \bmod p$$
$$a \ominus b = (a - b + p) \bmod p$$
$$a \otimes b = (a \cdot b) \bmod p$$
$$a \oslash b = (a \cdot b^{p-2}) \bmod p$$

(vyzkoušejte si, že se tyto operace chovají podobně jako normální sčítání, odčítání, násobení a dělení reálných čísel).

$\mathcal{X}$ navíc považoval číslo 4 za číslo magické (čtyři přeci jsou živly, světové strany, strany čtverce, kola od vozu rozměry časoprostoru a jiné důležité věci). Proto se rozhodl studovat, kolik čísel 4 spolu s popsanými celočíselnými operacemi stačí k zapsání každého čísla od 0 do $p-1$. Takovýto minimální počet čtyřek pro dané p nazval *magickou konstantou* svého číselného systému a ihned odvodil, že pro $p = 5$ to je 3:

$$0 = 4 \ominus 4, \quad 1 = 4 \otimes 4, \quad 2 = 4 \oplus 4 \oplus 4, \quad 3 = 4 \oplus 4$$

Záhy však zjistil, že pro velké p je poměrně náročné hodnotu magické konstanty zjistit, a tak se obrací na Vás, abyste mu napsali program, který pro dané prvočíslo $p > 4$ příslušnou hodnotu určí.

8-3-3 Nešťastný pátek — 10 bodů

Říká se, že neštěstí nechodí po horách, ale ... například po pátcích třináctého. Za rok obzvláště šťastný je dozajista možno považovat takový, v němž není ani jeden pátek třináctého, naopak za nad míru nešťastný pak ten, kde jsou takové tři nebo dokonce více. Napište program, který pro daný rok $r > 1582$ zjistí, kolik je v něm zmíněných nešťastných pátků.

Pro ty z vás, kteří neznají přesná pravidla kalendáře, podotýkáme, že v Gregoriánském kalendáři zavedeném v roce 1582 a používaném dodnes jsou přestupné ty roky, které jsou dělitelné čtyřmi, pokud ovšem nejsou dělitelné stem. Jsou-li dělitelné stem, jsou přestupné jen tehdy, jsou-li navíc dělitelné čtyřmi sty. Tedy například rok 1900 přestupný nebyl, zatímco rok 2000 bude.

8-3-4 Základní kámen — 10 bodů

Vítejte ve světě matematické logiky. Všude okolo jsou logické výrazy, logické operace a logické proměnné. Pojďme si je nyní prohlédnout zblízka. Nejprve operace:

$$\begin{array}{llll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & 0 \oplus 0 = 0 & 0 \Rightarrow 0 = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & 0 \oplus 1 = 1 & 0 \Rightarrow 1 = 1 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & 1 \oplus 0 = 1 & 1 \Rightarrow 0 = 0 \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & 1 \oplus 1 = 0 & 1 \Rightarrow 1 = 1 \end{array}$$

$$\begin{array}{llll} 0 \Leftarrow 0 = 1 & 0 \barwedge 0 = 1 & 0 \barvee 0 = 1 & \neg 0 = 1 \\ 0 \Leftarrow 1 = 0 & 0 \barwedge 1 = 1 & 0 \barvee 1 = 0 & \neg 1 = 0 \\ 1 \Leftarrow 0 = 1 & 1 \barwedge 0 = 1 & 1 \barvee 0 = 0 & \\ 1 \Leftarrow 1 = 1 & 1 \barwedge 1 = 0 & 1 \barvee 1 = 0 & \end{array}$$

Pak jsou zde proměnné označované malými písmeny latinské abecedy (tedy a až z) a konečně výrazy složené z operací a proměnných podle následujících pravidel:

* Každá proměnná je platný výraz.
* Je-li v platný výraz, je též $\neg v$ platný výraz.
* Jsou-li u a v platné výrazy, pak i $(u \vee v)$, $(u \wedge v)$, $(u \oplus v)$, $(u \Rightarrow v)$, $(u \Leftarrow v)$, $(u \barwedge v)$ a $(u \barvee v)$ jsou platné výrazy.
* Jiné platné výrazy neexistují.

Operace $\barwedge$ a $\barvee$ jsou ovšem něčím zvláštní – jsou to totiž *základní kameny matematické logiky* – pomocí každé z nich (a bez operací ostatních) je možno zapsat libovolný logický výraz, například

$$\neg(a \vee b) = (((a \barwedge a) \barwedge (b \barwedge b)) \barwedge ((a \barwedge a) \barwedge (b \barwedge b))).$$

Vaším úkolem je napsat program, který danou logickou funkci F přepíše pomocí operace $\barwedge$. Pro počítač budeme ovšem logické operace zapisovat pomocí standardních ASCII znaků takto: $\wedge \rightarrow$ `*`, $\vee \rightarrow$ `+`, $\neg \rightarrow$ `!`, $\Rightarrow \rightarrow$ `>`, $\Leftarrow \rightarrow$ `<`, $\barwedge \rightarrow$ `&`, $\barvee \rightarrow$ `|`, $\oplus \rightarrow$ `^`.

8-3-5 Konečně automaty — 15 bodů

A nakonec pokračování seriálu o konečných automatech – další dva jazyky (ten druhý je o něco obtížnější):

$L_1 = \{w \in \{\mathbf{a}, \mathbf{b}\}^* \,;\ w$ obsahuje podslovo **abba** a neobsahuje **baba** $\}$
$L_2 = \{w \in \{\mathbf{0}, \mathbf{1}\}^* \,;\ w = \mathbf{1}^n\mathbf{0}^n$ pro nějaké celé kladné číslo $n\}$

Příklad: První jazyk obsahuje slovo **ababbab** a neobsahuje **babbaba**, druhý obsahuje **11110000** a neobsahuje **11000** ani **1110001**.

8-4-1 Cesta tam a zase zpátky — 13 bodů

Bylo jednou jedno království, ve kterém ke značné nelibosti místní fauny i flory začala jezdit auta. Jak již tomu bývá, auta jezdí po silnicích, které vedou mezi městy. Všechny silnice v říši jsou však jednosměrné (což neznamená, že z jednoho města do druhého nemůže vést jedna jednosměrka tam a druhá zpátky). Časem ale lidé zjistili, že se v některých oblastech začínají hromadit auta, neboť je možné, aby se z jednoho města dostali do druhého a zpátky se už nemohli vrátit ani oklikou.

Plynula léta, starého krále vystřídal na trůně jeho druhorozený syn (ten starší si totiž jednoho dne vyjel na obhlídku říše a nikdy se již nevrátil) a rozhodl se dostavět silniční síť tak, aby vedla cesta mezi libovolnými dvěma městy. Svému rádci pro dopravu zadal úkol, aby na příští zasedání královské rady připravil seznam silnic, jež je nutné dostavět, aby se dalo dostat (byť

oklikou) z každého města do každého. Rádce se navíc rozhodl, že na krále udělá dobrý dojem tím, že připraví seznam nejkratší možný (tedy s co nejmenším počtem silnic, i když možná budou dost dlouhé) a po krátkém uvažování dospěl k názoru, že je to problém pro počítač, a najal vás, abyste za drobný bakšiš vytvořili program, který pro zadaný počet měst, počet dosud existujících silnic a jejich seznam nalezne alespoň jedno minimální řešení.

Příklad 1: 5 měst, dosud 3 silnice: $\langle 1,2\rangle$, $\langle 2,3\rangle$, $\langle 3,4\rangle$. Odpověď: je třeba ještě vybudovat $\langle 4,5\rangle$ a $\langle 5,1\rangle$.

Příklad 2: 3 města, dosud 4 silnice: $\langle 1,2\rangle$, $\langle 2,3\rangle$, $\langle 3,2\rangle$, $\langle 2,1\rangle$. Odpověď: Silniční síť je vyhovující.

8-4-2 Ďábelské mocniny — 10 bodů

Královský matematický ústav pořádá soutěž o nejlepšího matematika v říši. Všichni soutěžící dostanou zadáno spočíst danou poměrně vysokou mocninu nějakého velkého čísla a mají na to k dispozici mechanický kalkulátor nevalné kvality schopný za poměrně dlouhý čas vynásobit dvě zadaná čísla a k tomu také tužku a papír na poznámky (čísla jsou ovšem natolik dlouhá, že byť sečíst je ručně dá příliš mnoho práce).

Vy byste se též rádi soutěže zúčastnili – zkuste tedy vymyslet algoritmus, který pro daná přirozená čísla a a b nalezne postup, jak spočíst a^b nejmenším možným počtem násobení.

Příklad: V prvním ročníku soutěže (tomu již jest drahně let) bylo zadáno 1337^6. Výherce si nejprve spočetl $1337^2 = 1337 \cdot 1337 = 1787569$ a poté $1337^6 = 1337^2 \cdot 1337^2 \cdot 1337^2 = 5712003219749941009$.

8-4-3 Top secret — 11 bodů

Královští vyzvědači zachytili tajnou depeši vyslance znepřátelené země. Aby mohli armádní experti na kryptoanalýzu tuto depeši rozluštit, potřebují pro dané číslo n zjistit jeho rozklad na součin dvou jiných čísel $a, b > 1$. Poté, co jste se stali slavnými vyřešením dopravních problémů království, najali vás, abyste jim dodali program na řešení takovýchto problémů. Z dobře informovaných kruhů je známo, že číslo n je součin dvou prvočísel.

Ve svých řešeních můžete předpokládat, že se všechna čísla vejdou do 32-bitového celočíselného typu (*long int* v C, *longint* v Pascalu apod.). Algoritmus ale koncipujte tak, aby byl schopen zpracovávat co možná největší čísla n.

8-4-4 Penězokazi — 10 bodů

Vrchní bankéř královské banky pojal podezření, že v království začínají řádit penězokazi a chce si ověřit, zda je oprávněné. Pokusil se shromáždit co

nejvíce bankovek z jedné podezřelé série (v každé sérii je 100000 bankovek číslovaných od nuly do 99999) a rád by zjistil, zda některé dvě z nich mají stejné číslo. Činiti tak ručně by byla práce na mnoho dní, takže si chce zjednodušit práci pomocí počítače. Jenže bankovní počítač je poněkud staršího data výroby – má pouze 32KB programové paměti a 4KB datové paměti, zato však dostatek volného místa na disku.

Na vás je napsat program, který pro daných N čísel v rozsahu 0 až 99999 při splnění uvedených podmínek zjistí, zda se tam nějaké vyskytuje vícekrát. Rozhodující je rychlost programu – přístup na disk je nesrovnatelně pomalejší než přístup do paměti.

8-4-5 (Ne)konečné automaty — 15 bodů

Náš seriál úloh o konečných automatech pokračuje...

a) Jsou dány jazyky

$$L_1 = \{w \in \{\mathbf{a},\mathbf{b}\}^* ;\ w \text{ obsahuje podslovo } \mathbf{bbba}\ \}$$
$$L_2 = \{w \in \{\mathbf{a},\mathbf{b}\}^* ;\ w \text{ obsahuje podslovo } \mathbf{baaa}\ \}$$
$$L_3 = \{w = xy ;\ x \in L_1 \text{ a zároveň } y \in L_2\ \}$$

Sestrojte konečný automat přijímající jazyk L_3.

b) Cenzorní úřad našeho království, jak se ostatně dalo předpokládat, pozorně sleduje i úlohy *KSP* a uvědomil si, že by se mu hodil inteligenční potenciál řešitelů k tomu, aby mohli lépe a radostněji cenzorovat novinové články. V první fázi by ocenili, kdyby byli rychle schopni zjistit, zda se v textu vyskytuje příslušné nevhodné či hanlivé slovo. Zakoupili totiž superrychlý interpreter konečných automatů, do nějž vloží dodaný text a příslušný konečný automat a on jim řekne, zda automat text přijme (což je pro autora špatné, neboť automat detekoval nepřípustné slovo), a tak jim stačí sestrojit pro každé nevhodné slovo konečný automat testující, zda se v textu toto slovo vyskytuje.

Jenže dlouhá léta vývoje obohatila národní jazyk království o neuvěřitelné množství nevhodných slov (za slovo se v tomto případě dokonce považuje i spojení více "klasických" slov pomocí mezer a interpunčních znamének) a kromě toho se jazyk vyvíjí, čili nevhodná slova přibývají, jména vladařů se mění (takže místo "Xaver je osel" se časem stane nevhodným "Rudpert je osel"), pročež potřebují práci zautomatizovat a navrhnout algoritmus, který pro dané slovo vygeneruje příslušný konečný automat, jenž přijímá právě texty toto slovo obsahující.

Běžné novinové články obsahují malá a velká písmena, mezery, čárky, tečky, vykřičníky a otazníky – konečný automat by tedy měl být definován nad touto abecedou. Pokuste se zvolit nějakou *rozumnou* formu výpisu definice automatu.

Poznámka: V terminologii konečných automatů je "slovo" vlastně celý námi zpracovávaný text, zatímco "podslovo" je hledané slovo v textu. Tudíž hledaný automat přijímá ta slova, která obsahují zadané podslovo.

Příklad: Pro slovo "baba" nám program nageneruje v *KSP* již zprofanovaný automat (přijme nejen větu "Královna je baba.", ale i větu "Dejte si pozor, lupič Řimbaba utekl z šatlavy!").

8-5-1 Rozmarný princ — 12 bodů

Bylo-nebylo. V jednom malém království pod vysokými horami žil krásný princ, jemuž se neustále dvořily princezny. Dokonce se mu dvořila i rozmarná princezna z království za horami, která byla ošklivá a hloupá k tomu. Prince již pomalu začínaly davy vdavek (i říše) chtivých princezen nudit, a tak se poradil se svým komořím-matfyzákem a vymysleli si hru, kterou si s ním měla každá princezna ucházející se o jeho přízeň zahrát. Princezny potom samozřejmě neměly šanci, ale byly ochotny zaplatit za každý nápad, jak prince porazit.

Vaším úkolem je napsat pro princeznu program a popis strategie (algoritmu), jenž jí pomůže vyhrát, pokud to ovšem lze. Napište též, je-li (a pokud ano, tak kdy a v kterém tahu) možné v průběhu hry říci, že princezna, která používá vaši strategii, vyhraje, a složitost vašeho algoritmu.

Pravidla: Zda začíná princezna nebo princ, určí se losem. Na začátku je náhodně zvolen počet hromádek (1 až 10), počet sirek na každé z nich (na počátku všech stejně, a to 1 až 1000) a kolik sirek lze maximálně najednou odebrat z jedné hromádky (1 až 10). Poté začne hra a oba hráči se střídají v odebírání sirek z hromádek. V jednom tahu smí hráč odebírat vždy jen z jedné hromádky alespoň jednu a nejvýše onen zvolený počet sirek. Kdo odebere poslední sirku z libovolné hromádky, prohrál.

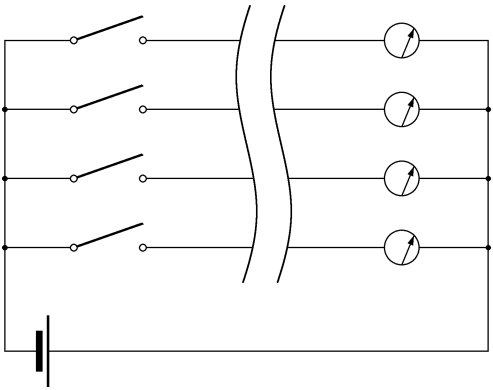
8-5-2 Telefon do pekla — 11 bodů

V pekle se rozhodli, že si zavedou telefon. Nejprve všude natahali telefonní kabely, každý obsahoval stínění a kolem stovky drátů. Když se snažili kabely připojit k telefonní ústředně, s hrůzou, jaká je pouze peklu vlastní, zjistili, že nevědí, který drát na jednom konci odpovídá kterému na konci druhém (jen stínění bylo zcela jasně propojeno, neboť bylo pouze jedno). Navíc ještě některé dráty, ač vodit měly, prokazatelně nevodily. Dospěli tedy k závěru, že by si měli postavit speciální testovací zařízení, které bude na jednom konci umět připojovat napájecí napětí k jednotlivým drátům (jako zem se použije stínění) a na druhém konci napětí měřit (viz obrázek).

Vás si najali jakožto proslulé odborníky přes programování (jistě jste neodmítli, neboť mít protekci v pekle není od věci), abyste vymysleli algoritmus pro optimální obsluhu tohoto testeru – tedy takový, který použije co nejmenší

počet kroků k tomu, aby zjistil propojení všech drátů v kabelu a které dráty jsou nevodivé.

Na počátku je dán počet drátů N (pro oba konce kabelu stejný) a všechny vypínače jsou ve vypnutém stavu. Program v každém kroku vydá příkaz k přivedení napětí na drát (‘`+X`’, kde X je číslo drátu), k odpojení napětí od drátu (‘`-X`’) či ke změření napětí na druhém konci (‘`?X`’). Uživatel na každý příkaz ‘`?`’ odpoví ‘`+`’ nebo ‘`-`’ podle toho, zda nějaké nenulové napětí naměří či nikoliv. Až si bude program jist výsledkem, vypíše pro každý drát ‘`X -> Y`’, pokud je drát X na jednom konci připojen k drátu Y na konci druhém, přičemž nepřipojené dráty vynechá.

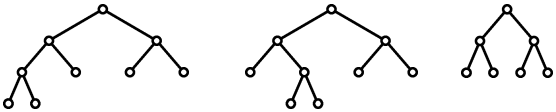


8-5-3 Pomsta šíleného zahradníka — 12 bodů

Jeden zahradník pěstoval stromy. A to ne ledajaké – byly to stromy binární. Pokud přesně nevíte, co je to binární (přesněji binární kořenový) strom, pak vězte, že:

- Prázdná množina je binární strom.
- Jsou-li x a y binární stromy, pak (x, y) je též binární strom.
- Žádné jiné binární stromy neexistují.

Tedy například $(((\emptyset,\emptyset),\emptyset),(\emptyset,\emptyset))$, $((\emptyset,(\emptyset,\emptyset)),(\emptyset,\emptyset))$ a $((\emptyset,\emptyset),(\emptyset,\emptyset))$ jsou binární stromy. Trochu názornější je obvyklá grafická podoba – podle jejího vzezření se též stromy jmenují. Naše ukázky by vypadaly takto:



O dvou binárních stromech x a y náš zahradník říkal, že jsou *isomorfní*, pokud se liší pouze natočením větví – tento vztah značil $x \simeq y$. Exaktně to lze definovat splněním alespoň jedné z následujících podmínek:

- $x = y = \emptyset$
- $x = (a, b)$, $y = (c, d)$, $a \simeq c$, $b \simeq d$
- $x = (a, b)$, $y = (c, d)$, $a \simeq d$, $b \simeq c$

Z našich ukázek jsou první a druhý strom navzájem isomorfní, zatímco třetí není isomorfní s žádným jiným.

Napište program, který našemu zahradníkovi pro dva zadané binární stromy určí, zda jsou isomorfní či nikoliv.

8-5-4 Čekej, Snad Dojede — 12 bodů

V jednom bývalém království měli přeslavnou železniční síť. Vlaky jezdily z místa na místo, ba dokonce, ať tomu věříte či nikoliv, dodržovaly jízdní řády. Nicméně síť tratí byla tak složitá, že téměř nikdo nedokázal nalézt rozumné spojení z jednoho města do druhého.

Pro každý vlak je známo jeho číslo (1 až 9999), počáteční i cílová stanice, jakož i všechny další, v nichž staví, spolu s příslušnými časy (čas příjezdu a čas odjezdu pro každou stanici). Jména stanic jsou až 16-znakové jednoznačné řetězce. Mezi nádražími není možno cestovat jinak než železnicí. Na přestup je potřeba vždy alespoň jedna minuta (tedy přijede-li vlak v 15.20, můžete odjet až v 15.21).

Napište program, který načte popis železniční sítě ve zmíněném formátu a poté odpovídá na dotazy všetečných cestujících, jak se v nejkratším možném čase dostat ze stanice $\mathcal{A}$ do stanice $\mathcal{B}$ (ve stanici $\mathcal{A}$ jsme v daném čase t). Pro každé řešení nechť program vypíše seznam stanic, v nichž budou použité vlaky stavět, spolu s časem příjezdu, časem odjezdu a číslem vlaku, jímž stanici opustíme. Pokud je více stejně rychlých řešení, stačí vypsat pouze jedno.

8-5-5 Automata finita — 10 bodů

Jistě všichni dychtivě očekáváte, že na tomoto místě naleznete zadání dalšího konečného automatu. Dlouho jsme přemýšleli, jaký automat vám zadat, když už vlastně z předešlé série máte program, který vám je bryskně generuje a už o nich všechno podstatné víte.

Naštestí jsme nakonec přeci jen nalezli vhodné zadání. Tady tedy je:

a) Sestrojte konečný automat, který rozpoznává Céčkové komentáře. Céčkový komentář je slovo ze znaků ASCII, které začíná posloupností znaků '`/*`' a končí posloupností znaků '`*/`', přičemž uvnitř neobsahuje '`*/`'. '`/*/`' se nepovažuje za Céčkový komentář.

b) Sestrojte konečný automat rozpoznávající Céčkové komentáře s vnořením definované takto:

- Každý Céčkový komentář je také vnořeným Céčkovým komentářem, pokud neobsahuje posloupnost znaků '`/*`' vyjma svých uvozovačů.
- Dále je to každé slovo z ASCII znaků, které začíná a končí posloupnostmi '`/*`' a '`*/`' a které, mimo jiné, může obsahovat další Céčkové komentáře s vnořením a neobsahuje '`/*`' ani '`*/`' jinde než v těchto vnořených komentářích a svých uvozovačích.
- Jiné Céčkové komentáře s vnořením nejsou.

Tedy například (každý řádek je chápán samostatně):

```
/* toto je komentar */
/* toto (**) je take /* komentar */
/* toto neni */ komentar */
/* ani toto */ /* komentar neni */
toto neni komentar ani nahodou
```

Či pro druhou podúlohu:

```
/* toto je komentar */
/* toto neni /* komentar */
/* toto také neni */ komentar */
/* toto /* je */ /* komentar */ */
/* toto /* je /*take*/ *//*komentar*/ */
/* a /* toto */ pro */ zmenu */ neni*/
```

Konečné automaty

Letošní ročník *KSP* obsahuje seriál úloh o *konečných automatech.* Podívejme se tedy, co to vlastně konečné automaty jsou:

Definice:

- **Konečným automatem** (dále jen KA) nazýváme každou pětici

$$A = (Q, \Sigma, \delta, q_0, F)\,, \text{kde}$$

  Q je konečná množina stavů KA
  Σ je konečná neprázdná množina (vstupní abeceda)
  δ je zobrazení $Q \times \Sigma \to Q$ (přechodová funkce)
  q_0 je počáteční stav ($q_0 \in Q$)
  F je množina koncových stavů ($F \subseteq Q$)
- **Slovo, délka slova, zřetězení slov**

Je-li Σ libovolná konečná množina (abeceda), pak Σ^+ označuje množinu všech konečných neprázdných posloupností utvořených z prvků množiny Σ, e označuje prázdnou posloupnost a definujeme $\Sigma^* = \Sigma \cup \{e\}$.

Posloupnost $(a_1, \ldots, a_n) \in \Sigma^*$, budeme zapisovat $a_1 \ldots a_n$. Každou takovou posloupnost nazýváme **slovem** v abecedě Σ, e nazýváme **prázdným slovem.**

O Σ^* mluvíme jako o **množině všech slov nad abecedou** Σ, o Σ^+ jako o **množině všech neprázdných slov nad** Σ.

Jestliže $u = a_1 \ldots a_n$ a $v = b_1 \ldots b_m \in \Sigma^*$, pak řetěz $uv = a_1 \ldots a_n b_1 \ldots b_m$ nazveme **zřetězením slov** u a v. Speciálně $eu = ue = u$. Symbol u^n bude označovat **n-násobné zřetězení slova** u, tzn. $u^1 = u$, $u^2 = uu$, $u^{i+1} = u^i u$. **Délku slova** u budeme značit $|u|$, tj. $|a_1 \ldots a_n| = n$ a $|e| = 0$.

- **Jazyk**

  Je-li Σ konečná abeceda a $L \subset \Sigma^*$, pak L nazýváme **jazykem nad abecedou** Σ.

- **Zobecněná přechodová funkce**

  Přechodovou funkci $\delta : Q \times \Sigma \to Q$ konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$ rozšíříme na **zobecněnou přechodovou funkci** $\delta^* : Q \times \Sigma^* \to Q$ takto:

  1. $\delta^*(q, e) = q \quad \forall q \in Q$.
  2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a) \quad \forall q \in Q,\ w \in \Sigma^*,\ a \in \Sigma$.

  Jazykem rozpoznávaným konečným automatem A pak nazveme jazyk $L(A) = \{w; w \in \Sigma^* \ \&\ \delta^*(q_0, w) \in F\}$.

  Říkáme, že **slovo** $w \in \Sigma^*$ **je přijímáno automatem** A právě když $w \in L(A)$.

- **Rozpoznatelný jazyk**

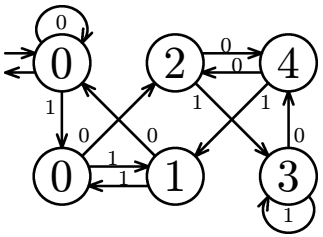
  Existuje-li konečný automat A takový, že jazyk $L = L(A)$, říkáme, že **jazyk** L **je rozpoznatelný konečným automatem.**

A teď o čem to vlastně je:

Tato definice je ovšem pro naše použití poněkud suchá, a proto se ji pokusíme převést do srozumitelnější řeči (či obrázků). Zavedeme si proto znázornění KA pomocí koleček a šipeček následovně:

Každý stav je zakreslen právě jedním kolečkem (ve kterém může být eventuálně i netriviální jméno stavu). Každá hodnota přechodové funkce $\delta(q, a) = p,\ q, p \in Q,\ a \in \Sigma$ je znázorněna šipkou ve směru od stavu q do stavu p a ohodnocenou prvkem abecedy a. Jediný počáteční stav je znázorněn neohodnocenou šipkou vedoucí dovnitř a každý koncový stav je znázorněn neohodnocenou šipkou vedoucí ven. Slovo w je KA přijímáno právě tehdy, když existuje cesta z *počátečního* stavu do nějakého *koncového* stavu ve směru šipek a posloupnost ohodnocení šipek na této cestě je totožná se slovem w.

Asi nejlepší bude, uvedeme-li si příklad:



Toto je KA rozpoznávající jazyk, jenž obsahuje binární zápis čísel dělitelných šesti. Projděme si nyní definici KA a podívejme se na jednotlivé složky pětice:

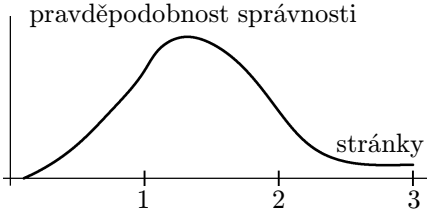
$\Sigma = \{\mathbf{0}, \mathbf{1}\}$	Abeceda obsahuje pouze **0** a **1**
$Q = \{D_0, D_1, D_2, D_3, D_4, D_5\}$	Tento automat má 6 stavů D_0 až D_5
$q_0 = D_0$	Počáteční stav je D_0
$F = \{D_0\}$	KA obsahuje jediný koncový stav D_0

$$\begin{array}{ll} \delta\,(D_0, \mathbf{0}) = D_0 & \delta\,(D_0, \mathbf{1}) = D_1 \\ \delta\,(D_1, \mathbf{0}) = D_2 & \delta\,(D_1, \mathbf{1}) = D_3 \\ \delta\,(D_2, \mathbf{0}) = D_4 & \delta\,(D_2, \mathbf{1}) = D_5 \\ \delta\,(D_3, \mathbf{0}) = D_0 & \delta\,(D_3, \mathbf{1}) = D_1 \\ \delta\,(D_4, \mathbf{0}) = D_2 & \delta\,(D_4, \mathbf{1}) = D_3 \\ \delta\,(D_5, \mathbf{0}) = D_4 & \delta\,(D_5, \mathbf{1}) = D_5 \end{array}$$

Dříve než začnete řešit "automatové" úlohy, dobře si rozmyslete, proč tento automat přijímá pouze čísla dělitelná šesti! (Nápověda: Jsme-li v průběhu čtení čísla ve stavu D_i, pak již přečtené číslo dává zbytek po dělení šesti právě i. Plyne to z toho, že přečtení další 0 resp. 1 odpovídá vynásobení již přečteného čísla dvěmi a přičtení 0 resp. 1 s tím, že příslušně změní zbytek po dělení šesti.) Ti z vás, které problematika zaujme víc, anebo ti, kteří budou mít pocit, že výše uvedený popis není dostatečný, mohou sáhnout třeba po knížce *Michal Chytil: Automaty a gramatiky (SNTL 1984).*

Vzorová řešení

Programy v některých řešeních *KSP* jsou dlouhé tři, čtyři někdy i více stran. V takto dlouhých programech se téměř vždy snadno objeví chyby. Na základě dlouhotrvajících výzkumů jsme sestavili následující graf vyjadřující závislost mezi délkou programu a pravděpodobností, že je tento program správným řešením průměrné úlohy *KSP:*



Z tohoto grafu jasně vyplývá, že optimální délka zdrojového kódu programu v řešení je zhruba jedna stránka a kousek. Při větší délce programu rapidně klesá pravděpodobnost, že je správným řešením úlohy. Co z toho plyne? Přesáhnu-li při psaní programu k úloze z *KSP* hranici dvou stránek, je asi něco v nepořádku a měl bych si to ještě jednou promyslet.

8-1-1 Rozděl a panuj

Jan Kotas

Z názvu úlohy by čtenář mohl usoudit, že její řešení bude založeno na známé metodě pro konstrukci efektivních algoritmů *Divide et Impera* (Rozděl a panuj). Bohužel, správné řešení s touto metodou vůbec nesouvisí.

Rozdělíme přirozeným způsobem hodnoty $1 \ldots N^2$ valounů do matice $N \times N$. V průsečíku r-tého řádku a s-tého sloupce je hodnota $(r-1) \cdot N + s$:

1	2	3	...	N
$N+1$	$N+2$	...	...	...
$2 \cdot N + 1$	...	...	...	...
...	...	...	...	...
...	...	...	...	N^2

Dáme-li každému zlatokopovi právě jeden valoun z každého sloupce a každého řádku, budou valouny rozděleny rovnoměrně: (r_i, s_i jsou souřadnice hodnoty i-tého valounu nějakého pevně zvoleného zlatokopa)

$$\sum_{i=1}^{N} [(r_i - 1) \cdot N + s_i] = N \cdot \sum_{i=1}^{N} (r_i - 1) + \sum_{i=1}^{N} s_i =$$

$$= N \cdot \sum_{i=1}^{N}(i-1) + \sum_{i=1}^{N} i = N \cdot \frac{N \cdot (N-1)}{2} + \frac{N \cdot (N+1)}{2} =$$

$$= \frac{N \cdot (N^2+1)}{2}$$

Nejjednodušší způsob, jak provést uvedené rozdělení, je přiřazovat valouny zlatokopům po diagonálách v rozšířené matici (stejným způsobem rozšířená matice se používá například při počítání determinantů Sarrusovým pravidlem):

1	2	3	...	N
$N+1$	$N+2$	...	...	...
$2 \cdot N + 1$	...	...	...	...
...	...	...	...	...
...	...	...	...	N^2
1	2	3	...	N
$N+1$	$N+2$	...	...	...
...	...	...	...	...

První zlatokop dostane valouny z hlavní diagonály. Druhý zlatokop dostane valouny z diagonály začínající $N+1$ a končící N, třetí z diagonály začínající $2 \cdot N + 1$ atd. Je zřejmé, že při tomto způsobu rozdělování dostane každý zlatokop právě jeden valoun z každého sloupce a každého řádku.

```
program KSP811;

var
 n,              { počet zlatokopů }
 r,              { n-násobek čísla řádku }
 s,              { číslo sloupce }
 i, j : integer; { řídící proměnné cyklu }

begin
  write('Počet zlatokopů: '); readln(n);

  for i:=1 to n do { pro každého zlatokopa }
    begin
      write('Zlatokop ', i:2, ' :');

      r := 0; s:= i;

      for j:=1 to n do { rozděluj }
        begin
          write(' ', r+s:3);
          r := r+n;
          if s=n then s:=1 else s:=s+1;
        end;

      writeln;
```

```
    end;
end.
```

Časová složitost algoritmu je $O(N^2)$, lepší ani být nemůže, protože vždy potřebujeme vypsat N^2 čísel. Paměťová náročnost je při vhodné implementaci konstatní (tj. $O(1)$).

Algoritmus je zřejmě konečný, protože obsahuje pouze dva do sebe vnořené for-cykly. Správnost algoritmu byla ukázána výše.

8-1-2 Sedlový bod

Martin Mareš

Jedná se o tak jednoduchou úlohu, že je až ku podivu, kolik se toho na ní dá zkazit: Základním problémem ve značné části řešení bylo, že si autoři neuvědomili, že se jedná o *neostré* maximum resp. minimum a že jich tedy na jednom řádku může být více. Typické příklady toto ukazující jsou:

$$\begin{pmatrix} 2 & 2 & 2 \\ 3 & 1 & 3 \\ 3 & 1 & 3 \end{pmatrix} \qquad \text{a} \qquad \begin{pmatrix} 2 & 1 & 1 \\ 2 & 4 & 4 \\ 2 & 1 & 1 \end{pmatrix}$$

V prvním z nich je sedlový bod na souřadnicích $[1, 2]$, v tom druhém na $[2, 1]$. Autoři jiných řešení sice pochopili, že minim může být více, ale procházeli otrocky všechna z nich neuvědomivše si, že takovýto postup má za následek zvýšení časové složitosti v nejhorším případě (např. při matici plné stejných hodnot) na $O(N^3)$. Jiným nešvarem jsou "přebujelé" vstupy a výstupy obsahující spoustu "bells & whistles" – příkazů, které dělají povyk na všechny strany (barvičky, pískání apod.), ale ve skutečnosti k ničemu užitečnému neslouží – tak na takových věcech opravdu v tomto semináři nezáleží. Hlavní je (pokud není řečeno jinak) algoritmus.

Vzorové řešení si nejprve spočte minima na jednotlivých řádcích (jejich hodnoty, nikoliv polohy) do pole R, a poté hledá maxima ve sloupcích, přičemž každé nalezené porovnává s hodnotou minima příslušného řádku, čímž objevuje sedlové body. Časová složitost tohoto algoritmu je $O(N^2)$ a lépe to ani nelze, neboť je nutno prozkoumat každou z N^2 hodnot alespoň jednou. Správnost algoritmu plyne z definice sedlového bodu.

8-1-3 Klíč

Vít Novák

Tak jak se dalo očekávat, mnozí řešitelé přistoupili ke klíčům jako programátoři a zvolili pěkný algoritmus založený na tzv. dynamickém programování ... Tedy pro orientaci, v čem to spočívalo:

Vím, že do bodů B,C,D, daných souřadnicemi $(3, 5)$, $(3, 6)$, $(3, 7)$, vede z počátečního bodu A $(0, 5)$ postupně 7, 6 a 3 cest. Pak ale do bodu $(4, 6)$ vede právě $7 + 6 + 3$ cest. Doplňme si nyní ještě omezení v krajích a můžeme, vrstvu

po vrstvě, vyplnit počet cest, kterými se dostaneme do daných bodů čtvercové sítě. Tak zjistíme počet klíčů daných parametrů v čase $N \cdot M \cdot k$, kde k je řád výsledku, tedy

$$\log \binom{n}{\frac{n}{2}} = \log \frac{n!}{(n/2)!^2}$$

(Nezapomeňte, že na počítači veškeré operace jako sčítání mají časovou složitost rovnou řádu čísel, se kterými počítáte, i když se to obvykle zanedbává.)

Dobrá tedy, jde to rychleji, je k tomu však potřeba trochu matematiky.

Nejprve se podívejme na to, kolik klíčů bychom získali, kdybychom neměli horní a dolní limit. Takových klíčů je přesně tolik, jako cest ve čtvecové síti z počátečního do předepsaného koncového bodu. Nechť rozdíl výšek těchto bodů je δ, pak je potřeba těchto δ bodů rozmístit a dále můžeme přidat několik úseků nahoru, stejně úseků dolů a doplníme to úseky rovnými. Máme tedy dva variabilní prvky:

1. počet přidaných stoupání a klesání – budeme tedy počítat součet pro $l = 0..|\frac{n-\delta}{2}|$
2. rozmístění jednotlivých úseků – to je snadná kombinatorika

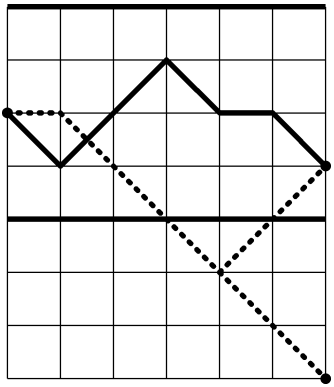
Celkově pak takových cest tedy je

$$\sum_{l=0}^{|\frac{n-\delta}{2}|} \frac{n!}{l! \cdot (n-l)!} \cdot \frac{(n-l)!}{(l+\delta)! \cdot (n-2l-\delta)!} =$$

$$= \sum_{l=0}^{|\frac{n-\delta}{2}|} \frac{n!}{l! \cdot (l+\delta)! \cdot (n-2l-\delta)!}$$

Jak nyní začleníme limity? Každá z cest, která překročí dolní limit, musela nutně projít bodem s y-ovou souřadnicí -1. Pokud nyní zbytek cesty obrátíme právě podle této přímky (-1), dostaneme se nakonec do bodu (-2) – (výška koncového bodu) (viz obrázek). Počet cest, které překročí dolní limit pak je tedy roven počtu cest z počátečního bodu do takto zrcadlově překlopeného koncového bodu. Pro horní limit se zachováme obdobně.

Tak jsme si vyjádřili počet klíčů jako tři součty. Uvědomíme-li si, že počet sčítání takto je řádově n, faktoriály až do $n!$ si můžeme předpočítat a operace násobení a sčítání opět budou mít složitost asi $\log(n) \cdot k$, dostali jsme algoritmus o časové složitosti $n \cdot k \cdot \log(n)$. Samotný program považuji za triviální, pročež jej zde ani neuvádím.



8-1-4 MIDI *Martin Bělocký*

Tato úloha patřila k algoritmicky jednodušším, leč svým zadáním nadělala spoustu nepříjemností, protože mnozí vůbec nepochopili, o co vlastně jde a co přesně se má dělat. Chtěli jsme po vás, abyste napsali efektivní program, který bude syntetizátoru předzpracovávat MIDI program. Takový program by měl být tak rychlý jako syntetizátor, aby mohl zpracovávat řádky stejnou rychlostí jako syntetizátor (tj. program s lineární časovou složitostí). Takový program by měl být též lehce "zadrátovatelný", aby se mohl připojit přímo na vstup syntetizátoru. (tj. program nemůže mít žádné veliké paměťové nároky). Mnozí jste však místo toho dělali programy, které by se ani nevešly do paměti a měly složitost minimálně N^2 (kde N je počet řádků). Syntetizátor může být schopen generovat několik desítek tisíc různých tónů, ale naráz jich může být jen malý počet (v praxi to bývá 32 nebo 64). Jedna skladba může být hodně dlouhá, např. plná disketa, rychlost běhu programu tedy nesmí záležet na počtu řádků.

Vstupní a výstupní zařízení nebylo přímo určeno, používáme tedy disk. Načtením (zápisem) časové jednotky rozumíme načtení (zápis) všech příkazů ve stejném čase.

Náš program bude mít v paměti vždy všechny příkazy dvou časových jednotek. Když totiž budeme zpracovávat aktuální příkaz v čase T, může to mít vliv na obsah pole s předchozí časovou jednotkou, (např. vložení `OFF` v čase $T-1$). Obě časové jednotky budeme mít uloženy v polích (`akt_cas`, `pr_cas`). V každém kroku algoritmu vždy načteme do pole aktuálního času (`akt_cas`) časovou jednotku, potom dle algoritmu zpracujeme po jednom každý příkaz. Když je to hotovo, uložíme obsah pole předchozí čas (`pr_cas`) na disk, obsah pole `akt_cas` zkopírujeme do `pr_cas`, a do `akt_cas` načteme novou časovou jednotku. Tyto kroky opakujeme až do konce vstupního souboru.

Program používá ještě pole s názvem `zapnuta`, kde jsou čísla zapnutých tónů a ke každému tónu číslo, které značí, kolik se má vypustit příkazů `OFF`

s číslem daného tónu. Zpracování jedné časové jednotky se provádí lineárně – příkaz po příkazu tak, jak to popisuje vlasní algoritmus řešení.

Složitost: Paměťová složitost tedy bude konstantní (3 pole po 256-ti informacích), neboť velikost paměti je nezávislá na vstupních datech. Časová složitost bude lineární – $O(N)$, kde N je počet řádků programu, ježto pro každý řádek provedeme několik konstantních akcí, které nejsou závislé na délce vstupních dat. Program tedy může přímo průběžně předávat svá data syntetizátoru a ne na disk, což bylo úkolem.

Algoritmus zpracování jednoho příkazu:

- Na řádku je `ON`:
  - Tón není zapnut (není v poli `zapnuty`) ⇒ Tón se vloží do pole `zapnuty`.
  - Tón je zapnut:
    - Předcházející časová jednotka má čas jen o 1 nižší a obsahuje zapnutí (`ON`) stejného tónu ⇒ aktuální řádek se zruší (číslo tónu se nastaví na 0) a do pole `zapnuty` se vloží požadavek na odstranění následujícího `OFF Ton`. (odstraněním `ON` musím také odstranit `OFF`)
    - V opačném případě do pole `pr_cas` vložím řádek `CAS-1 OFF Ton` a do pole `zapnuty` vložím požadavek na odstranění `OFF Ton`.
- Na řádku je `OFF`:
  - V poli `zapnuty` není požadavek na odstranění `OFF Ton` ⇒ Zjistíme, je-li některý ještě nezpracovaný řádek aktuální časové jednotky (`akt_cas`) tvaru `CAS ON Ton`.
    - Takový řádek existuje ⇒ na konec pole `akt_cas` vložíme stejný řádek jako je řádek aktuální (tedy `CAS OFF TON`) a aktuální řádek odstraníme. Prakticky jde pouze o přehození pořadí zapnutí a vypnutí tónu (na konečný výsledek to nemá vliv, neb dle zadání se oba případy zpracují stejně).
    - Řádek existuje ⇒ Tón se vypne, tj. odstraní z pole `zapnuty`.
  - V poli `zapnuty` je požadavek na odstranění `OFF Ton` ⇒ řádek se odstraní, tj. nastaví se jeho tón na 0.

Rozmyslete si, že není třeba dělat žádné další úpravy, neboť tento algoritmus obstará vše, co bylo v zadání.

8-1-5 Konečné automaty

Michal Koucký

Mile nás překvapilo, jak mnoho řešitelů se pokusilo tuto úlohu řešit, neb jsme ani nedoufali v to, že někdo z naší definice pochopí, co to vlastně konečný automat je. Kupodivu také většina došlých řešení byla správná.

Nyní již k jednotlivým jazykům:

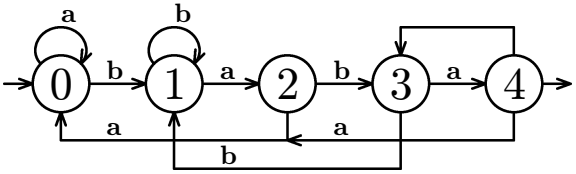
a) Jazyk slov, která končí na **baba**. Se sestrojením konečného automatu pro tento jazyk jste většinou neměli problémy. Jediné, s čím jste občas měli potíže, bylo určit, do kterého stavu musí automat přejít, pokud zatím načtené slovo končí na **bab** (dle autorského řešení jsme ve stavu 3) a na vstupu je písmeno **b**. V tomto případě totiž automat nepřechází do počátečního stavu (stav 0), ale do stavu, kdy slovo končí na písmeno **b** (stav 1).

Nyní k tomu, jak mohl vypadat konečný automat:

$$KA = (\Sigma, Q, \delta, 0, F)$$
$$\Sigma = \{\mathbf{a}, \mathbf{b}\}$$
$$Q = \{0, 1, 2, 3, 4\}$$
$$F = \{4\}$$

$$\begin{array}{ll} \delta\,(0, \mathbf{a}) = 0 & \delta\,(0, \mathbf{b}) = 1 \\ \delta\,(1, \mathbf{a}) = 2 & \delta\,(1, \mathbf{b}) = 1 \\ \delta\,(2, \mathbf{a}) = 0 & \delta\,(2, \mathbf{b}) = 3 \\ \delta\,(3, \mathbf{a}) = 4 & \delta\,(3, \mathbf{b}) = 1 \\ \delta\,(4, \mathbf{a}) = 0 & \delta\,(4, \mathbf{b}) = 3 \end{array}$$

Stav 1 značí, že načtené slovo končí na **b**
Stav 2 značí, že načtené slovo končí na **ba**
Stav 3 značí, že načtené slovo končí na **bab**
Stav 4 značí, že načtené slovo končí na **baba**
Stav 0 značí, že načtené slovo nekončí na nic z výše uvedených možností



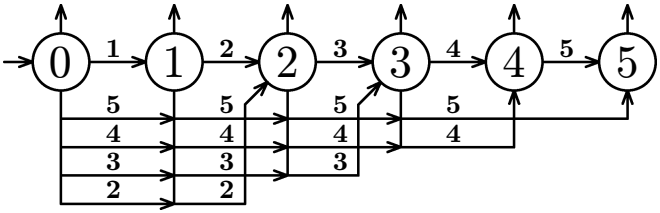
b) Jazyk ostře rostoucích posloupností čísel 0 až 5. Toto byl relativně jednoduchý jazyk. Jediné, kde se mohl vyskytnout problém, bylo to, zda posloupnost neobsahující žádné číslo nebo číslo právě jedno je také posloupnost. To v zadání nebylo explicitně uvedeno, avšak za posloupnost se to považuje.

KA pro tento jazyk bude mít šest stavů – počáteční stav, a pak pro každou číslici jeden stav, který bude určovat hodnotu poslední načtené číslice. Je zřejmé, že pokud budeme ve stavu pro některou číslici n a přijde nám číslice m, pak pokud je tato číslice vyšší než ta, kterou jsme načetli minule, tedy n, pak je zatím všechno v pořádku a přejdeme do stavu pro číslici m. Pokud je

však m menší nebo rovno n, posloupnost není ostře rostoucí, tudíž ji nesmíme přijmout.

Nepřijmutí lze udělat dvěma způsoby: Buď zavedeme stav zvaný *stoupa,* který nebude koncový a všechny přechody z něj budou zpět do něj a zavedeme do něj také všechny hrany, jež nemají přijímat, a nebo můžeme hrany, které nesmí přijímat, prostě vůbec nedefinovat. V definici přijímání KA totiž stojí, že automat má skončit v koncovém stavu. Pokud však cestou nějaký přechod chybí (není definován), pak zřejmě dle definice slovo není přijato.

Automat tedy bude vypadat takto:



$$KA = (\Sigma, Q, \delta, D_0, F)$$

$$\Sigma = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$$

$$Q = \{D_0, D_1, D_2, D_3, D_4, D_5\}$$

$$F = \{D_0, D_1, D_2, D_3, D_4, D_5\}$$

$$\begin{array}{ll}
\delta\,(D_0, \mathbf{1}) = D_1 & \delta\,(D_2, \mathbf{3}) = D_3 \\
\delta\,(D_0, \mathbf{2}) = D_2 & \delta\,(D_2, \mathbf{4}) = D_4 \\
\delta\,(D_0, \mathbf{3}) = D_3 & \delta\,(D_2, \mathbf{5}) = D_5 \\
\delta\,(D_0, \mathbf{4}) = D_4 & \\
\delta\,(D_0, \mathbf{5}) = D_5 & \delta\,(D_3, \mathbf{4}) = D_4 \\
 & \delta\,(D_3, \mathbf{5}) = D_5 \\
\delta\,(D_1, \mathbf{2}) = D_2 & \\
\delta\,(D_1, \mathbf{3}) = D_3 & \delta\,(D_4, \mathbf{5}) = D_5 \\
\delta\,(D_1, \mathbf{4}) = D_4 & \\
\delta\,(D_1, \mathbf{5}) = D_5 &
\end{array}$$

8-2-1 O líných novinářích

Jan Kotas

Uvažme, jak asi postupoval zkušený řešitel *KSP* při řešení této úlohy. Nejdříve si důkladně přečetl zadání a tiše si pro sebe řekl: "Délka textu nejvýše 1000 znaků, hmmm, tím asi chtěl autor úlohy naznačit, že můžeme pro uložení textu použít normální pole znaků, není nutné pracovat s textem jako se souborem. A je možné aby se opakující se úseky překrývaly? Pro jistotu budu

předpokládat, že to možné je – stejně to asi ovlivňuje jenom meze nějakých hloupých cyklů. A co by vlastně mělo být výstupem programu? Kdybych já byl šéfredaktor, bylo by asi nejlepší, kdyby program vypisoval vždy na jedné řádce všechny pozice, na kterých se opakuje jeden druh úseku." (Pokud jste se na tyto otázky dívali jinak, nepovažovali jsme to za chybu.)

A pak začal přemýšlet o tom, jakým algoritmem by se úloha dala vyřešit. První algoritmus, který jej napadl, hledal postupně pro všechny možné dvojice začátků úseků délku shodného úseku. Když byla délka shody větší než dosud nalezené maximum, zapomněl všechno, co dosud našel, a zapamatoval si právě nalezenou dvojici úseků. Když byla délka shody rovna dosud nalezenému maximu, přidal k již nalezeným úsekům právě nalezenou dvojici. Tento algoritmus měl časovou složitost $O(N^3)$, kde N je délka textu.

Našemu zkušenému řešiteli ale bylo hned jasné, že nám tento algoritmus nepošle, protože za tak málo nápadů by nikdo 12 bodů nedal. Tedy se zkusil zamyslet ještě jednou – napadly jej vyhledávací stromy, připomnělo mu to nějaké algoritmy pro kompresi, atd. Když si ale představil délku zdrojového kódu řešení založených na těchto nápadech, udělalo se mu špatně, a proto šel raději spát. (Na rozdíl od organizátorů *KSP,* kteří právě opravují elaboráty tohoto typu z minulé série.) Až ráno, které je, jak praví známé české přísloví, moudřejší večera, po příjemně pro (žité | sněné) noci, přišel na následující algoritmus s časovou složitostí $O(N^2)$:

Dva úseky textu jsou od sebe vzdáleny vždy o nějakou vzdálenost z intervalu $\langle 1, N-1 \rangle$. Pro každou možnou vzdálenost úseků se tedy provede prohledávací průchod, při kterém se zjišťují shodné úseky. (Jinými slovy: Posouváme dvě (myšlené) kopie textu vůči sobě a zjišťujeme nejdelší shodné úseky.)

Nyní ale stál před problémem, jak celou věc zrealizovat, aby z algoritmu vypadly výsledky ve formě, kterou si na začátku zvolil, a aby mu to zároveň nezvýšilo časovou složitost algoritmu. Tento problém vyřešil tím, že se prohledávání textu bude provádět dvakrát:

Při prvním průchodu se pouze zjistí délka nejdelších opakujících se úseků.

Při druhém průchodu se budou zpracovávat pouze dvojice úseků již zjištěné maximální délky. Budeme je zařazovat do skupin (podobně jako při zjišťování souvislých komponent grafu pomocí faktorových množin). Skupina bude obsahovat indexy začátků stejných úseků (tj. to, co se bude vypisovat vždy na jeden řádek). Pokud se najdou dva úseky, z nichž ani jeden není ještě v žádné skupině, založí se nová skupina. Této skupině se musí přiřadit nějaký unikátní identifikátor – číslo. Způsobů, jak toto číslo vybrat, je více. V programu se jednoduše vezme index začátku jednoho z úseků. Pokud se najdou dva úseky, z nichž je právě jeden již zařazen do nějaké skupiny, druhý se dá do téže skupiny. Pokud jsou nalezené úseky každý v jiné skupině, je třeba skupiny sloučit. To bude sice trvat čas $O(N)$, ale složitost celého algoritmu – $O(N^2)$ – nám to

nezvýší, protože slučování se bude provádět zřejmě nejvýše N-krát.

Teprve teď si sedl k počítači a napsal program.

Poznámky:

Pokud bychom vytvářeli skupiny rovnou při prvním průchodu, mohlo by se stát, že pracně najdeme postupným slučováním skupinu několika úseků, kterou vzápětí zapomeneme, protože nalezneme shodu s větší délkou. Složitost takového algoritmu by mohla překročit $O(N^2)$. Jelikož je ale možné předpokládat, že shoda je relativně málo častý jev, byl by asi v praxi použit právě tento algoritmus.

Paměťovou náročnost programu v průměrném případě by bylo možné snížit tím, že by pro ukládání výsledků použilo vhodné hashování.

```
const MaxN = 1000; { maximalni delka textu }

var
 A: array [1..MaxN] of char; { pole pro ulozeni textu }
 N: Integer; { delka textu }
 G: array [1..MaxN] of Integer; { pole pro ulozeni vysledku, G[h] obsahuje }
     { identifikator skupiny useku, do ktere patri usek zacinajici na pozici h }
 I,J,K,L: Integer; { pomocne promenne }
 Max: Integer; { maximalni delka shodneho useku }
 Current: Integer; { aktualni delka shodneho useku }

begin
 N := 0;
 while not eof do { precteni textu ze standardniho vstupu }
  begin
   N := N + 1;
   if N > MaxN then Halt(1); { kdyby nas chtel nekdo zmast, tak se kousnem }
   if eoln then begin A[N] := ' '; readln end { konce radku nahradime mezerama }
    else read(A[N])
  end;

 { nejdrive pouze najdeme delku nejdelsiho opakujiciho se useku }

 Max := 0;
 for J := 1 to N - 1 do { pro vsechny mozne posuny }
  begin
   Current := 0;
   for I := 1 to N - J do { prohledej text }
    if A[I] = A[I + J] then
      begin
       Current := Current + 1;
       if Current > Max then Max := Current
      end
     else Current := 0;
  end;

 if Max = 0 then
   begin writeln('Neexistuje zadny opakujici se usek.'); exit end;

 for I := 1 to N do G[I] := 0; { vynulovani pole s vysledky }
  { G[I] = 0 -> usek nepatri do zadne skupiny }
```

```
 for J := 1 to N - 1 do { jeste jednou na ostro }
  begin
   Current := 0;
   for I := 1 to N - J do
    if A[I] = A[I + J] then
      begin
       Current := Current + 1;
       if Current = Max then { nasli jsme dvojici }
         begin
          if G[I - Max + 1] = 0 then
            if G[I + J - Max + 1] = 0 then
              begin { ani jeden usek zatim nepatri do zadne skupiny }
               G[I - Max + 1] := I - Max + 1; { zalozime jim tedy novou }
               G[I + J - Max + 1] := I - Max + 1;
              end
             else
              { prvni usek dame skupiny jako druheho }
              G[I - Max + 1] := G[I + J - Max + 1]
           else
            if G[I + J - Max + 1] = 0 then
              { druhy blok dame do skupine prvniho }
              G[I + J - Max + 1] := G[I - Max + 1]
             else
              begin
               K := G[I + J - Max + 1];
               { skupinu prvniho bloku pridame ke skupine druhe pouze, kdyz se lisi! }
               if K <> G[I - Max + 1] then
                 for L := 1 to N do
                  if G[L] = K then G[L] := G[I - Max + 1];
              end;
         end;
       end
      else Current := 0;
  end;

 { vypsani vysledku }

 writeln('Maximalni delka opakujicich se useku je ', Max);
 writeln('Pozice opakujicich se useku:');
 for I := 1 to N do
  if G[I] > 0 then
    begin
     write(I);
     for J := I + 1 to N do
      if G[I] = G[J] then
        begin write(' - ', J); G[J] := 0 end;
     writeln;
     G[I] := 0;
    end;
end.
```

8-2-2 Rozmarná princezna

Martin Bělocký

U první varianty se dá na první pohled říci, kdo vyhraje – záleží to jen na tom, kdo odebírá první, a na počtu sirek na stole. Pokud je na stole lichý počet, vyhrává ten, kdo je první na řadě. Důkaz: Odečtením lichého čísla od lichého počtu sirek dostaneme vždy sudý počet. Odečtením lichého čísla od sudého počtu dostaneme lichý počet $\Rightarrow$ pokud se oba hráči pravidelně střídají a protože lze odebírat jen lichý počet (1, 3 nebo 5), na jednoho musí vycházet vždy jen sudý počet sirek a na druhého lichý. Program tedy pouze odebírá nejvyšší povolený počet sirek, aby se zkrátila hra, ale na postupu odebírání jinak vůbec nezáleží.

Druhá varianta: odebírají se pouze mocniny dvou. Nejprve si dokážeme dvě větičky:

1. Žádné číslo dělitelné třemi není mocnina dvou. Důkaz: Každá mocnina dvou má ve svém prvočíselném rozkladu pouze dvojky, kdežto každý násobek tří alespoň jednu trojku.
2. Z žádného čísla A dělitelného třemi nelze odebráním mocniny dvou M dostat žádné číslo B dělitelné třemi, tedy neplatí $A - M = B$, $3|A$, $3|B$. Důkaz sporem: Kdyby platilo $A - M = B$, pak také musí platit $A - B = M$. Víme, že A i B jsou násobky tří, tedy můžeme vytknutím trojky psát $3 \cdot (C - D) = M$, tedy číslo M musí být dělitelné třemi, pročež dle 1. M nemůže být mocnina dvou.

Pokud tedy budeme hrát tak, aby měla princezna na stole vždy jen počet sirek dělitelný třemi, pak nemůže vyhrát, neboť 0 je dělitelná třemi a odebráním libovolné mocniny dvou se na ni nelze dle tvrzení 2 dostat, tedy princezna nemůže vyhrát. Toto se v programu realizuje velice snadno testováním zbytku po dělení počtu sirek třemi. Pokud vyjde nula, pak vyhrává prozatím princezna, my odebereme 1 sirku, abychom hru prodloužili a měli větší šanci, že princezna udělá chybu.

Povšimněte si v programu finty na testování, zda je číslo mocnina dvou: Číslo $A > 0$ typu integer je mocnina dvou právě tehdy když platí `(A-1) AND A = 0`.

```
Program sirky;
var Typ_hry:char;
    konec  :boolean;
    psir   :integer;

procedure princezna;       {odebira princezna}
var pom:integer;
begin
  if typ_hry ='1' then
  begin
    repeat
    writeln('Na stole je ',psir,'sirek,jste na rade, princezno, kolik sirek odeberete ?');
```

```
    read(pom);
    if ((pom<>1)and(pom<>3)and(pom<>5))or(psir-pom<0) then
      writeln('Myslel jsem, ze princezny nesvindluji!! Odebirejte znovu!');
    until not(((pom<>1)and(pom<>3)and(pom<>5))or(psir-pom<0));
  end
  else
  begin
    repeat
      writeln('Na stole je ',psir,'sirek, jste na rade, princezno,');
      writeln('Kolik sirek odeberete?');
      read(pom);
      if not( ((pom-1)and(pom) =0)and(pom>0)and(psir-pom>=0)) then
       writeln('Myslel jsem, ze princezny nesvindluji!! Odebirejte znovu!');
    until (((pom-1)and(pom)) =0)and(pom>0)and(psir-pom>=0);
  end;
  psir:=psir-pom;
  if psir=0 then
  begin
        writeln('Vyhrala jste, Velicenstvo, a ja jdu spat do chlivku...');
    konec:=true;
  end;
end;

procedure hra1;                {varianta hry 1}
var pom:integer;
begin
 write('Na stole je ',psir,' sirek, jsi na rade, odebiras ');
 if psir>5 then pom:=5
 else if psir>3 then pom:=3
 else pom:=1;
 writeln(pom, 'sirek');
 psir:=psir-pom;
 if psir=0 then
  begin
    writeln('Vyhral jsi, (za trest) pujdes spat s princeznou...');
    konec:=true;
  end;
end;

procedure hra2;                          {2.varianta hry}
var pom,pom1,a:integer;
begin
 write('Na stole je ',psir,'sirek, jsi na rade a odebiras ');
 if psir mod 3 =0 then pom:=1
 else pom:=psir mod 3;
 writeln(pom, 'sirek');
 psir:=psir-pom;
 if psir=0 then
  begin
    writeln('Vyhral jsi, (za trest) pujdes spat s princeznou...');
    konec:=true;
  end;
end;


begin
  writeln('S kolika sirkami se bude hrat?');
  readln(psir);
```

```
  writeln ('jaka varianta hry se bude hrat?');
  writeln('1...lze odebirat pouze 1,3,5 sirek');
  writeln('2...lze odebirat lib. mocninu dvou sirek');
  readln(Typ_hry);
  konec:=false;
  case typ_hry of
    '1':begin
       if not (odd(psir)) then writeln ('mate to marne, princezno !')
       else writeln('nema to cenu, prohraju to...');
       while not konec do
       begin
         princezna;
         if not konec then hra1;
       end;
      end;

    '2':begin
       if (psir mod 3 )=0 then
         writeln ('mate to marne, princezno, ale prece si s vama zahraju !')
       else
       begin
        writeln('pokud jste tak chytra, jak se o vas rika princezno,');
        writeln('tak mi jiste nedate sanci. Jinak si o vasem IQ budu myslet sve...');
       end;
       while not konec do
       begin
         princezna;
         if not konec then hra2;
       end;
      end;
    else writeln ('zadna hra se hrat nebude, kdyz nechces hrat...');
  end;
end.
```

8-2-3 U zeleného stromu

Michal Koucký

Jako vždy si někteří řešitelé špatně přečetli zadání a bezdůvodně předpokládali, že na začátku je židle s číslem $2 \cdot N + 1$ volná či že na konci tomu má být také tak. Ani jedno však nebyla pravda a tuto skutečnost jsme ohodnotili stržením jednoho až dvou bodů.

Z hlediska časové složitosti se vyskytla dvě řešení – s kvadratickou časovou složitostí a s lineární časovou složitostí. To první jsme samozřejmě hodnotili hůře.

Obě řešení vypadala takto: vyhledávám špatně sedící hobity (trpaslíky) a ty pak přemisťuji na volnou židli. Rozdíl byl v tom, jak, či spíše kde, vyhledávám špatně sedící hobity (trpaslíky). Ti, kteří hledali vždy od začátku (resp. jiné pevné meze), dosáhli kvadratické časové složitosti. Ti, kteří hledali jen tam, kde můžou ještě být špatně sedící hobiti (trpaslíci), dosáhli lineární časové složitosti. Kde se tedy mělo hledat? Mělo se hledat od místa, kde jsme tuto osobu našli posledně. Tu jsme totiž přesadili, před ní určitě již nikdo není (jinak bychom při

minulém hledání vzali toho, kdo tam je) a stačí tedy hledat v neprohledaném zbytku.

Jaký to ale mělo vliv na časovou složitost? Pro zkoumání prohledávání od začátku si vezměme nejhorší případ, tj. případ, kdy jsou všichni hobiti v první části a všichni trpaslíci v druhé části:

$$\texttt{HH}\ldots\texttt{HHTT}\ldots\sqcup\ldots\texttt{TTTT}$$

(na pozici mezery nezáleží). V tom případě při i-tém vyhledávání nalezneme i-tého hobita. i-tý hobit je na i-té pozici, tudíž než k němu dojdeme, otestujeme i prvků. Hobita budeme muset vyhledat N-krát (každého špatně sedícího hobita musíme najít). Při prvním hledání tedy prohledáváme jeden prvek, při druhém dva, atd. až při N-tém N prvků. To dohromady dává

$$\sum_{i=1}^{N} i \;=\; \frac{N \cdot (N+1)}{2}$$

testů při vyhledávání. Pokud se nějaká část programu opakuje $c.N^2$, pak program musí mít minimálně kvadratickou časovou složitost – $O\left(N^2\right)$.

Podívejme se nyní na případ, kdy hledáme vždy od minule nalezeného. Je sice pravda, že v nepříznivém případě může jedno konkrétní hledání trvat až N kroků, ale o to méně máme špatně sedících hobitů. V každém případě, každý prvek otestuji maximálně jednou, takže dohromady otestuji maximálně N prvků, tedy celková doba strávená ve vyhledávání je maximálně N testů, plus nějaká režie na proběhnutí testovacího cyklu N-krát a nějaká konstantní režie na maximálně N-násobné spuštění prohledávání. Celkově je tedy doba vyhledávání přímo úměrná velikosti N, tedy pokud někde něco ještě nepokazíme, můžeme dosáhnout lineární časové složitosti.

Tento rozbor jsme udělali pro hobity, ale pro trpaslíky je to zcela stejné.

Jediné co se dalo ještě pokazit ("Co se může pokazit, to se taky pokazí" – Murphy's laws) bylo to, že někteří řešitelé byli sklerotičtí a vždy znovu museli hledat prázdnou židli, ač si mohli pamatovat, že volná je ta, ze které se naposledy někdo přesadil.

Teď už asi všichni tušíte, jak mělo vypadat správné řešení. Tedy:
Stav na konci musí vypadat takto:

- na židli 1 až N sedí trpaslíci a jedna židle může být volná (tedy nesedí zde žádný hobit).
- na prostřední židli může sedět kdokoliv, případně je volná.
- na $N+2$ až $2 \cdot N+1$ sedí hobiti, nebo je tu jedna židle volná (nesmí tu sedět žádný trpaslík).

Pokud jsou tato pravidla splněna, sedí všichni trpaslíci před hobity, což požadovalo zadání. Jinak tento požadavek splnit nelze (rozmyslete si).

Z koncové situace je vidět, že musíme hobity z židlí 1 až N přesadit do zadní části a trpaslíky z $N + 2$ až $2 \cdot N + 1$ do přední. Kdo bude sedět na prostřední židli, je nám jedno.

Postupovat budeme tak, že pokud je volná židle s číslem 1 až N, vyhledáme špatně sedícího trpaslíka (na židlích $N+2$ až $2 \cdot N+1$) a toho na ni přesadíme. Pokud bude volná židle v části $N + 2$ až $2 \cdot N + 1$, tak tam přesadíme špatně sedícího hobita z židle 1 až N.

Pokud bude volná prostřední židle, což může nastat pouze na začátku, neboť pokud tam někdo sedí, tak s ním nehýbeme, přesadíme tam buď špatně sedícího hobita, nebo trpaslíka. Je to jedno, neboť v tu chvíli je počet špatně sedících hobitů i trpaslíků stejný. To plyne z počtu židlí v levé a pravé části a počtu hobitů a trpaslíků.

Zbývá ukázat, že pokud je volná židle 1 až N a není to již srovnané, tak existuje trpaslík sedící špatně, tj. na židli $N + 1$ až $2 \cdot N + 1$. Pokud by to tedy ještě nebylo srovnané a onen trpaslík tam neseděl, znamená to, že nám srovnanost narušuje nějaký hobit sedící na židli 1 až N. Jelikož všichni trpaslíci jsou již na židlích 1 až $N+1$, dále je v dolní části volná židle a minimálně jeden špatně sedící hobit, museli bychom tam mít alespoň $N+2$ židlí (N trpaslíků + volná + alespoň jeden hobit). My jich ovšem máme jen $N+1$, tedy nutně tento případ nastat nemůže a určitě existuje špatně sedící trpaslík – toho vezmeme a přesadíme.

Pro situaci, kdy je volná židle v horní části, se obdobně ukáže, že existuje špatně sedící hobit. Neustále tedy přesazujeme špatně sedící hobity a trpaslíky (dokonce střídavě, ale toho si nemusíme všímat, nechceme-li).

Na úplný závěr ještě ukážeme, že tímto způsobem přesadíme trpaslíky a hobity na nejmenší možný počet tahů. To je ovšem triviální: Při každém přesazení nám ubyde jedna špatně sedící bytost, takže provedeme tolik přesazení, kolik je na začátku špatně sedících bytostí. Na druhou stranu méně udělat nemůžeme, neboť každá špatně sedící osoba musí být přesazena a žádným přesazením nelze dosáhnout toho, aby nám ubyly dvě špatně sedící osoby. Tedy algoritmus je korektní.

8-2-4 Náhrdelník

Martin Mareš

Tuto spíše oddychovou úlohu vyřešili téměř všichni řešitelé správně, nicméně zdaleka ne každé řešení bylo "rozumné."

Pravděpodobně nejméně vhodné bylo posloupnost celou načíst do paměti, tam ji nějakou více či méně efektivní metodou setřídit a poté kontrolovat, na kterém místě se čísla v posloupnosti liší o 2. Za tuto "metodu hrubé síly" pochopitelně moc bodů nebylo, jelikož je velice pomalá i paměťově náročná.

Jiní se rozhodli zavést pomocné pole a v něm si "odškrtávat", která čísla již byla načtena, což též není příliš rozumné, poněvadž princezna byla pravděpodobně velice bohatá a pro značkování jejích perel by mohla být paměť počítače příliš malá (i když použijete bitové pole). Navíc jste mnohdy zapomněli na to, že takovéto pomocné pole by bylo záhodno na začátku práce vynulovat (Pascal globálním proměnným implicitně žádnou hodnotu nepřiřazuje, zatímco C nuly zaručuje).

Elegantní řešení této úlohy je založeno na tom prostém faktu, že součet kompletní posloupnosti čísel 1 až N se od součtu posloupnosti, v níž právě jedno číslo chybí liší o přesně tolik, kolik jest hodnota onoho chybějícího čísla (pokud vyjde 0, žádná perla nechyběla). Navíc ještě existuje velice jednoduchá metoda, kterak stanovit součet čísel 1 až N bez jakéhokoliv cyklu: Buď

$$s = 1 + 2 + 3 + 4 + \ldots + N,$$

pak jistě též platí (sčítání je komutativní):

$$s = N + (N-1) + (N-2) + \ldots + 3 + 2 + 1.$$

Pokud tyto dvě rovnosti sečteme, dostáváme:

$$2 \cdot s = \overbrace{(N+1) + (N+1) + \ldots + (N+1)}^{N\text{-krát}},$$

a tak

$$s = \frac{N \cdot (N+1)}{2}.$$

Tento vzorec je všeobecně známý, mnozí řešitelé si na něj vzpomněli a použili jej, nicméně často naprosto zbytečně hodnotu počítali v typu *real,* což rozhodně není vhodné – kompilátor kvůli tomu musí přilinkovat matematické knihovny či spoléhat na to, že máte koprocesor, a navíc je to o něco pomalejší než naprosto zjevné celočíselné řešení $s = (N \cdot (N+1))$ div 2.

Řešení založená na tomto prostém, leč účinném triku tak dosahují konstantní paměťové náročnosti a lineární časové složitosti, což je zjevně nejlepší možné, neboť každé číslo na vstupu musíme "vzít do ruky" alespoň jednou. Má však ještě jednu vadu na kráse: pro velká N může dojít k přetečení, neboť s roste s druhou mocninou N. Tomuto problému se dá snadno vyhnout, použijeme-li místo sčítání a odčítání funkci *xor,* která nejen že nemůže přetéci, ale je též sama k sobě funkcí inverzní (tedy (x xor y) xor $y = x$). Tato funkce byla popsána též v úloze *Základní kámen* v minulé sérii.

Zbývá ještě jedno – jak snadno spočíst

$$x_N = 1 \text{ xor } 2 \text{ xor } 3 \text{ xor } \ldots \text{ xor } N.$$

Tvrdím, že x_N závisí na tom, jaký zbytek z dá číslo N při dělení čtyřmi:

- Pro $z = 0$ je $x_N = N$.
- Pro $z = 1$ je $x_N = 1$.
- Pro $z = 2$ je $x_N = N + 1$.
- Pro $z = 3$ je $x_N = 0$.

Důkaz tohoto tvrzení (matematickou indukcí) je poměrně snadný: Pro $N = 1$ (takže $z = 1$) věta jistě platí. Pokud již platí pro nějaké známé N (zapsatelné ve dvojkové soustavě jako $a_1 a_2 a_3 \ldots a_k z_1 z_2$, kde $z_1 z_2$ je binární vyjádření z), pak také platí pro $N+1$, neboť: (označme si $z_0 = N \bmod 4$, $z = (N+1) \bmod 4$)

- Pro $z = 0$ je $z_0 = 3$, takže $x_N = 0$, $x_{N+1} = x_N$ xor $(N + 1) = 0$ xor $(N + 1) = N + 1$.
- Pro $z = 1$ je $z_0 = 0$, takže $x_N = N = a_1 \ldots a_k 00$, tedy $N + 1 = a_1 \ldots a_k 01$, $x_{N+1} = x_N$ xor $(N+1) = 1$ (každý bit a_i se vyxoruje s ním samým, takže na příslušném místě vznikne nula a 00 xor 01 = 01).
- Pro $z = 2$ je $z_0 = 1$, takže $x_N = 1$, $N + 1 = b_1 \ldots 10$, tož $x_{N+1} = x_N$ xor $(N + 1) = 1$ xor $(N + 1) = b_1 \ldots 11 = N + 2$ (každý bit b_i se vyxoruje s nulou, takže se nezmění, a 10 xor 01 = 11).
- Pro $z = 3$ je $z_0 = 2$, takže $x_N = N + 1$ a $x_{N+1} = x_N$ xor $(N + 1) = (N + 1)$ xor $(N + 1) = 0$ (cokoliv vyxorováno samo se sebou dá nulu).

Tím jsme důkaz uzavřeli.

Program je natolik triviální, že jej snad ani není nutno blíže popisovati (pro ty otrlejší přidávám ještě "miniaturizovanou" jednořádkovou verzi akceptující čísla perel jako parametry, vracející číslo chybějící perly jako návratový kód a předpokládající, že perly nejsou všechny).

```
#include <stdio.h>

int N;                                  /* Počet perel */

void main (void)
{
int s;                                  /* "Mezisoučet" */
int p;                                  /* Právě načítaná perla */
    scanf ("%d", &N);                   /* Načteme počet perel */
    switch (N % 4)                      /* Stanovíme 1 xor ...N */
        {
        case 0: s=N; break;
        case 1: s=1; break;
        case 2: s=N+1; break;
        case 3: s=0;
        }
```

```
    do                                  /* Očekáváme čísla ukončená nulou */
        {
        scanf ("%d",  &p);
        s ^= p;
        }
    while (p);
    if (!s)                             /* A co zbývá? */
        puts ("Nechybí žádná perla.");
    else
        printf ("Chybí perla číslo %d\n",  s);
}
```

```
main(a,b)char**b;{int z=(~a&1)*a+!((a-1)&2);while(a--)z^=atol(*++b);return z;}
```

8-2-5 Konečné automaty

Michal Koucký

Automat pro první z jazyků, tj. pro jazyk slov, která neobsahují slovo **baba**, se dá sestrojit tak, že nejdříve sestrojíme pomocný konečný automat KA' pro rozpoznávání jazyka, jehož slova naopak slovo **baba** obsahují. Pokud tedy bude slovo přijaté automatem KA', bude to znamenat, že obsahuje slovo **baba**.

Z takovéhoto automatu pak ovšem snadno sestrojíme automat KA, jenž rozpoznává požadovaný jazyk, a to tak, že stavy, které byly přijímající v automatu KA' prohlásíme za nepřijímající a naopak stavy nepřijímající prohlásíme za přijímající.

Zjevně tedy náš výsledný automat bude přijímat právě to, co KA' nepřijímal, což jsme právě chtěli.

V řeči matematického formalismu by se tato úvaha dala zapsat takto:

$$L_1 = \{w \in \{\mathbf{a},\mathbf{b}\}^* \; ; \; w \text{ neobsahuje } \mathbf{baba}\}$$
$$L_1' = \{w \in \{\mathbf{a},\mathbf{b}\}^* \; ; \; w \text{ obsahuje } \mathbf{baba}\}$$

tedy

$$L_1' = \{\mathbf{a},\mathbf{b}\}^* - L_1$$
$$KA' = (\Sigma, Q, \delta, q_0, F')$$
$$KA = (\Sigma, Q, \delta, q_0, F)\,,$$

kde $F = Q - F'$.

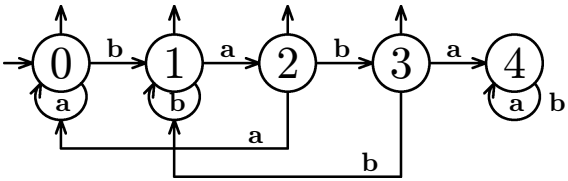
Nyní již konkrétně, jak bude vypadat náš automat KA pro jazyk L_1:

$$KA = (\Sigma, Q, \delta, D_1, F)$$
$$\Sigma = \{\mathbf{a},\mathbf{b}\}$$
$$Q = \{D_0, D_1, D_2, D_3, D_4\}$$
$$F = \{D_0, D_1, D_2, D_3\}$$

$$\begin{array}{ll} \delta\,(D_0, \mathbf{a}) = D_0 & \delta\,(D_0, \mathbf{b}) = D_1 \\ \delta\,(D_1, \mathbf{a}) = D_2 & \delta\,(D_1, \mathbf{b}) = D_1 \\ \delta\,(D_2, \mathbf{a}) = D_0 & \delta\,(D_2, \mathbf{b}) = D_3 \\ \delta\,(D_3, \mathbf{a}) = D_4 & \delta\,(D_3, \mathbf{b}) = D_1 \\ \delta\,(D_4, \mathbf{a}) = D_4 & \delta\,(D_4, \mathbf{b}) = D_4 \end{array}$$

Význam jednotlivých stavů je tento:

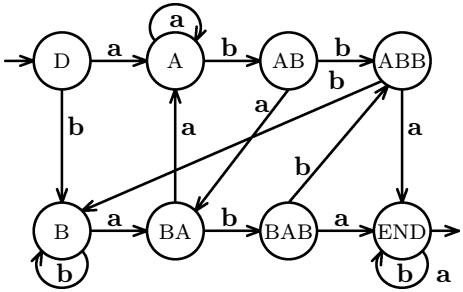
- D_0 nic z následujícího
- D_1 zatím načtené slovo končí na znak **b**
- D_2 zatím načtené slovo končí na **ba**
- D_3 zatím načtené slovo končí na **bab**
- D_4 zatím načtené slovo již obsahovalo **baba**



Při pohledu na náš automat a když si vzpomeneme na to, co jsme si říkali minule, zjistíme, že stav D_4 by se dal z automatu úplně vypustit. V okamžiku, kdy automat vstoupí do tohoto stavu, víme, že slovo nebude přijato. S ohledem na to, jak je definováno přijímaní, pokud všude, kde δ určuje přejít do tohoto stavu, nebudeme δ vůbec definovat a tento stav zrušíme, dosáhneme tím právě požadovaného efektu. Náš automat by tedy šel zmenšit o jeden zbytečný stav.

Automat pro druhý jazyk byl o něco obtížnější, neboť jsme měli sledovat výskyt hned dvou slov zároveň.

Stavy našeho vzorového automatu označíme (mnemotechnicky) posloupnostmi písmen, jež odpovídají pro nás zajímavému zakončení slova, které nás do tohoto stavu mohlo přivést.



$$
\begin{aligned}
L_2 &= \{w \in \{\mathbf{a},\mathbf{b}\}^* \ ; \ w \text{ obsahuje } \mathbf{baba} \text{ nebo } \mathbf{abba}\} \\
KA &= (\Sigma, Q, \delta, D, F) \\
\Sigma &= \{\mathbf{a},\mathbf{b}\} \\
Q &= \{D, DA, DB, DAB, DBA, DABB, \\
&\qquad DBAB, END\} \\
F &= \{END\}
\end{aligned}
$$

$$
\begin{array}{ll}
\delta\ (D,\mathbf{a}) = DA & \delta\ (D,\mathbf{b}) = DB \\
\delta\ (DA,\mathbf{a}) = DA & \delta\ (DA,\mathbf{b}) = DAB \\
\delta\ (DB,\mathbf{a}) = DBA & \delta\ (DB,\mathbf{b}) = DB \\
\delta\ (DAB,\mathbf{a}) = DBA & \delta\ (DAB,\mathbf{b}) = DABB \\
\delta\ (DBA,\mathbf{a}) = DA & \delta\ (DBA,\mathbf{b}) = DBAB \\
\delta\ (DABB,\mathbf{a}) = END & \delta\ (DABB,\mathbf{b}) = DB \\
\delta\ (DBAB,\mathbf{a}) = END & \delta\ (DBAB,\mathbf{b}) = DABB \\
\delta\ (END,\mathbf{a}) = END & \delta\ (END,\mathbf{b}) = END
\end{array}
$$

Stavy s označením $Dxxx$ jsou stavy, ve kterých se automat nachází, dokud nenalezne výskyt alespoň jednoho z hledaných slov. Pokud nějaký nalezne, přejde do stavu END, kde setrvá až do konce výpočtu. Tento stav je jediný přijímající, což plyne z toho, co jsme právě uvedli.

Pro ověření správnosti automatu se podívejme na následující: Označení stavů je takové, že pokud existují stavy $Dxyz$ a Dyz, pak slovo, které nás přivede do stavu Dyz určitě nemá na místě před y (předpředposlední písmeno) písmeno z, neboli v případě, že pro dané slovo v úvahu přicházejí dva různé stavy, kde bychom se mohli nacházet, jistě se nacházíme v tom s delším označením. Tato podmínka nám zaručuje, že nemůžeme nějaké slovo jaksi "přehlédnout".

Pro ověření této vlastnosti stačí prozkoumat, zda se při nějakém přechodu z jednoho stavu do druhého tato vlastnost neporuší. Postupně tedy probereme všechny možné přechody přechodové funkce δ a ověříme, že za předpokladu, že výchozí stav tuto vlastnost splňuje, bude i v cílovém tato vlastnost v pořádku.

Speciálně si tuto vlastnost musíme ověřit pro výchozí stav, kam se dostaneme "z ničeho nic" na počátku výpočtu. Pro něj to však zřejmě platí, neboť žádný jiný stav nemůže odpovídat prázdnému slovu, a tak je triviálně nejdelší ze všech možných.

Pro ilustraci se podívejme na přechod $\delta\ (DA,\mathbf{b}) = DAB$. Stav DA dle předpokladu vlastnost splňuje $\Longrightarrow$ načtené slovo nemohlo vypadat jako $xxxBA$, neboť jinak bychom byli ve stavu DBA, jediném delším stavu končícím na A. Připojením písmene **b** přejdeme do stavu DAB.

Z uvedené vlastnosti slova pro stav DA vyplývá, že pokud se dostaneme do DAB právě pomocí tohoto přechodu, pak vstupní slovo nemůže být $xxxBAB$.

Tedy vlastnost stavu DAB o vyloučení právě slova $xxxBAB$ z důvodů existence delšího stavu $DBAB$ není porušena.

Tento nástin důkazu správnosti našeho automatu, jak si jistě mnozí z vás všimli, zhruba odpovídá důkazu matematickou indukcí.

Tento druhý automat šel zkonstruovat též jiným, univerzálním postupem, který funguje pro jazyky, jež jsou sjednocením nějakých jiných jazyků. V našem případě je totiž hledaný jazyk sjednocením dvou různých jazyků – jazyka slov, která obsahují slovo **baba**, a jazyka slov, která obsahují **abba**.

Pokud najdeme automat pro první i druhý jazyk, což je relativně jednoduchá úloha, můžeme z nich sestrojit automat, který bude přijímat sjednocení obou jazyků.

Jak to však udělat? Mohli bychom oba automaty pustit jaksi zároveň a jakmile by skončily výpočet, podívat se, zda alespoň jeden vstupní slovo přijal. Pokud ano, patří toto slovo do jednoho z jazyků, tudíž i hledaný automat pro sjednocení ho má přijmout a my bychom ho přijali.

Jak však provedeme spuštění obou automatů najednou? Sestrojíme automat, jehož stavy budou všechny dvojice stavů našich automatů, tj. pokud jeden má stavy označeny písmeny a až x a druhý čísly 1 až k, pak nový automat může mít stavy označeny jako $\langle a,1\rangle$, $\langle a,2\rangle$, $\ldots$ $\langle a,k\rangle$, $\langle b,1\rangle$, $\ldots$ $\langle x,k\rangle$ nebo lépe $a1$, $a2$, $\ldots$, ak, $b1$, $\ldots$ xk (na označení, jak víme, nezáleží). Každý stav nového automatu pak odpovídá tomu, že oba výchozí automaty jsou v jemu odpovídajících stavech.

Přechodová funkce nového automatu se snadno sestrojí sloučením přechodových funkcí automatů výchozích, a to tak, aby přechod při přijetí nějakého písmene z jednoho stavu do druhého bral v úvahu přechody výchozích automatů, jako by byly spuštěny současně:

$\delta\,(en,i) = fm$ pokud v prvním automatu $\delta\,(e,i) = f$ a v druhém automatu $\delta\,(n,i) = m$.

Nyní je již vidět, jak bude vypadat množina přijímajících stavů našeho automatu: Bude obsahovat všechny stavy, které odpovídají tomu, že alespoň jeden z výchozích automatů se nachází v přijímajícím stavu. Výchozí stav nového KA je stav, který odpovídá výchozím stavům obou automatů.

Jak je vidět, toto je velice jednoduchý postup, jak takový automat pro sjednocení dvou jazyků sestrojit. Podobným způsobem se dá zkonstruovat i automat pro příjem průniku dvou jazyků.

Tato metoda je velice efektní a efektivní, má pouze jedinou nevýhodu – výsledek bývá zbytečně velký. Naštěstí existuje algoritmus, který vezme konečný automat a sestrojí z něj jiný automat, přijímající to samé a navíc nejmenší ze všech možných. Je zajímavé, že tento minimální automat je pro daný jazyk pouze jeden, tedy mám-li dva automaty, které přijímají to samé, pak pro ověření této skutečnosti stačí pro oba automaty sestrojit automat minimální a

pokud je stejný, jsou i oba jazyky skutečně stejné, pokud ne, nejsou. Tento postup minimalizace (v literatuře označovaný jako redukce automatu) zde však nyní vysvětlovat nebudeme, lze ho nalézt v již dříve uvedené knize Michala Chytila "Automaty a gramatiky".

Tato kniha je velice pěkná a užitečná, i když trochu náročnější, neboť je určena především vysokoškolákům. Přesto do ní můžete nahlédnout, protože když jste došli až sem, jistě kapitolám o konečných automatech porozumíte. Tuto knihu vydanou v SNTL někdy v 70. letech ale dnes již nejspíš v knihkupectví neuvidíte, možná jedině v nějaké vysokoškolské prodejně skript.

Tím jsem zodpověděl jeden z dotazů, kde se o konečnéch automatech můžete dozvědět více. Zůstává však ještě nerudovská otázka *"Kam s ním",* nebo ještě lépe *"K čemu to vlastně je."* Věřte nebo ne, konečné automaty mají i reálné uplatnění. Pominu-li to, že většina zařízení, se kterými se člověk na světě potká (počítač nevyjímaje), jsou konečné automaty, nalézají tyto automaty výborné uplatnění v programátorském prostředí.

První, s čím se setká libovolný zdrojový text libovolného programu napsaný v libovolném jazyce při kompilaci libovolným kompilátorem, je právě konečný automat provádějící takzvanou lexikální analýzu. Lexikální analýza je proces, při kterém se vstupní text rozdělí na jednotlivé lexikální elementy – identifikátory, speciální symboly, čísla, řetězce, komentáře atd. Lexikální analyzátor, což je ten, kdo ji provádí, není nic jiného, než konečný automat, jenž průchodem přes některé význačné stavy ohlašuje to, že našel konec dalšího lexikálního elementu.

Další uplatnění nacházejí konečné automaty například v různých (třeba textových) prohledávačích a vyhledávačích. Vy sami jste již sestrojili jednoduché konečné automaty například na vyhledávání slov **baba** v textu z písmen **a** a **b**. Je to právě ten automat, které rozpoznává jazyk slov končících na **baba**. Když totiž tento automat spustíte nad nějakém textem, průchod koncovém stavem signalizuje právě to, že jste detekovali další výskyt slova **baba**. Dokonce tento automat detekuje i překrývající se výskyty. Takovéto automaty použité na vyhledávání mají tu výhodu, že každý znak "vezmou do ruky" jen jednou, a to při rozhodování, do jakého stavu přejít – k jednou zpracovanému znaku se již nemusí vracet.

Kdybychom ale chtěli napsat takový vyhledávač ručně, tak by vypadal asi tak, že bychom postupně pro každý možný začátek vyzkoušeli, zda tam náhodou není hledané slovo. Pokud tam není, posuneme se na další znak a znovu zkoušíme. Takto vezmeme potenciálně do ruky každé znak až k-krát, kde k je délka hledaného slova. Tím ovšem typicky dosáhneme také k-násobného zpomalení.

A skutečně, dobré textové vyhledávače pracují tak, že si nejprve sestrojí konečný automat na vyhledávání požadovaného slova a ten poté spustí nad pro-

hledávaným textem. Výsledky můžete sami posoudit v prostředcích s různými názvy jako `grep` atd.

8-3-1 Kouzelné zrcadlo *Jan Kotas*

Zabývejme se nejdříve jednodušší variantou úlohy: hledejme program, který vypíše sám sebe. (Tato úloha patří mezi programátorskou klasiku. Princip, na kterém je založena, se velmi často používá v mnoha oblastech matematiky a informatiky např. v podobě vět o pevných bodech.)

Zamýslíme-li se nad možnostmi, které nám poskytuje Pascal (resp. Céčko), zjistíme, že zdrojový text programu musí být v programu obsažen jako textová konstanta. Ovšem tato konstanta nemůže obsahovat celý zdrojový text, protože by musela obsahovat sama sebe. Věc vyřešíme tak, že tato konstanta bude obsahovat zdrojový text programu až na tu část, ve které je její obsah zapsán. Program pak nejdříve vypíše část konstanty, která předchází její definici, pak tuto konstantu, a nakonec část konstanty, která následuje po její definici.

Ve většině programovacích jazyků ale vypadá zápis řetězců ve zdrojovém textu trochu jinak než při vypsání stejného textu na obrazovku. V Pascalu tento jev nastává u apostrofů, které se zdvojují. Problém je možné řešit buď soustředěním všech řetězců v programu na jedno místo (viz následující program v Pascalu), nebo napsáním "formátovače" (viz program pro původní úlohu v Céčku).

Kdybychom se snažili najít nejkratší možné řešení, byli bychom sváděni k psaní extrémně dlouhých řádků, případně ke kódování znaků pomocí ASCII hodnot. Tyto praktiky ale nejsou příliš pěkné, a proto je nebudeme používat.

```
const a: array [0..12] of string = (’’’’,
’const a: array [0..12] of string = (’,
’,’,
’);’,
’’,
’var i,j:integer;’,
’’,
’begin’,
’ writeln(a[1], a[0], a[0], a[0], a[0], a[2]);’,
’ for i := 1 to 11 do writeln(a[0], a[i], a[0], a[2]);’,
’ writeln(a[0], a[12], a[0], a[3]);’,
’ for i := 4 to 12 do writeln(a[i]);’,
’end.’);

var i,j:integer;

begin
 writeln(a[1], a[0], a[0], a[0], a[0], a[2]);
 for i := 1 to 11 do writeln(a[0], a[i], a[0], a[2]);
 writeln(a[0], a[12], a[0], a[3]);
 for i := 4 to 12 do writeln(a[i]);
end.
```

Existují i krátká elegantní řešení. Bohužel z nich ale není příliš zřejmá myšlenka, na které je úloha založena. V Pascalu může takové řešení vypadat třeba takto:

```
const p='const p='';begin writeln(copy(p,1,9),copy(p,1,9),copy(p,9,94),copy(p,9,2));
writeln(copy(p,11,92)) end.';
begin writeln(copy(p,1,9),copy(p,1,9),copy(p,9,94),copy(p,9,2)); writeln(copy(p,11,92)) end.
```

V Céčku můžeme použít málo známý operátor `#`, který dělá z parametru makra řetězec:

```
#define P(x) main() { printf(x,#x); }
P("#define P(x) main() { printf(x,#x); }\nP(%s)")
```

Poznámka: Tento program není správným programem v `C++`, protože nedefinuje návratovou hodnotu u funkce *main* a není v něm definován prototyp pro funkci printf. Když jej chcete přeložit např. v Borland `C++`, musíte v Options vybrat kompilaci pouze podle normálního Céčka. Ještě jiná možnost by byla:

```
main(){char *s="main(){char *s=%c%s%c;printf(s,34,s,34);}";printf(s,34,s,34);}
```

Teď jistě každý dokáže napsat program pro původní úlohu (ve vzorovém řešení byla preferována čitelnost před délkou zápisu):

```
#include <stdio.h>

char *a[43] = {
"#include <stdio.h>",
"",
"char *a[43] = {",
"\"\" };",
"",
"int checkChar(int c) { return getchar() == c ; }",
"",
"int checkLine(char *s) {",
" while(*s) if(!checkChar(*s++)) return 0;",
" return checkChar('\\n');",
"}",
"",

...

"",
"void main() {",
" if(checkProgram()) puts(\"Tak preci je to zrcadlo kouzelne\\n\");",
" else puts(\"Xakru, zase to nefunguje!\");",
"}",
"" };

int checkChar(int c) { return getchar() == c ; }
```

```
int checkLine(char *s) {
 while(*s) if(!checkChar(*s++)) return 0;
 return checkChar('\n');
}

int checkProgram() {
 int i;
 char *s;

 for(i=0; i<3; i++) if(!checkLine(a[i])) return 0;

 for(i=0; i<42; i++) {
  if(!checkChar('"')) return 0;

  for(s=a[i]; *s; s++)
   switch(*s) {
    case '"':
    case '\\': if(!checkChar('\\')) return 0;
    default : if(!checkChar(*s)) return 0;
   }

  if(!checkChar('"')) return 0;
  if(!checkChar(',')) return 0;
  if(!checkChar('\n')) return 0;
 }

 for(i=3; i<42; i++) if(!checkLine(a[i])) return 0;

 return checkChar(EOF);
}

void main() {
 if(checkProgram()) puts("Tak preci je to zrcadlo kouzelne\n");
 else puts("Xakru, zase to nefunguje!");
}
```

Zabývat se časovou a prostorovou složitostí u úlohy tohoto typu nemá valný smysl. Pro úplnost ale dodejme, že obě jsou konstantní tj. $O(1)$ – nezávisí na velikosti vstupu, nýbrž pouze na délce programu, která je konstantní.

8-3-2 Magické čtyřky

Vít Novák

Přes všechno naše snažení o jednoznačné zadání této poměrně jednoduché úlohy se přeci jen vloudila jedna drobnost: nebylo zjevné, zda v magických výrazech smějí nebo nesmějí být závorky. Vězte tedy, že původním záměrem bylo závorky povolit, poněvadž bez nich by úloha byla až nebetyčne triviální. Nicméně přiznáváme svou chybu a i řešení s existencí závorek nepočítající byla akceptována.

Algoritmus je založen na jednoduché úvaze, že pokud číslo lze napsat pomocí n čtyřek, pak lze napsat jako $a \circ b$, kde na napsání a a b potřebuji dohromady n čtyřek a $\circ$ je jedna z uvedených magických operací. Program tedy postupně bude brát všechna čísla, která lze napsat 1 čtyřkou a $(n-1)$ čtyřkami, 2

čtyřkami a $(n-2)$ čtyřkami ... a bude z nich kombinovat čísla zapsatelná n čtyřkami. Každé takové číslo, kterého bylo nově dosaženo, se uloží...

Ještě připomeňme jeden starý trik: kvůli časové složitosti bývá výhodné zdlouhavé výpočty, které se často opakují (např. faktoriály nebo v našem případě inverze), uložit do pole.

S časovou složitostí algoritmu je to poměrně složité, my se spokojíme pouze s horním odhadem (tedy nejhorším případem, o kterém ovšem nevíme, zda může skutečně nastat): Konstruujeme-li čísla zapsatelná n čtyřkami, pak jistě neprovedeme víc operací, než kdybychom kombinovali všechna již známá čísla se všemi, což je méně jak N^2 možností. Jenže jak rozumně odhadnout, kolik čtyřek bude nejvýše potřeba? Představte si, že se budete každé číslo snažit zapsat ve čtyřkové soustavě – tehdy dostanete nejvýše $\log_4 N$ sčítanců, každý z nich bude nějaká mocnina čtyř (spočteme z nejvýše $\log_4 N$ čtyřek), případně znásobená dvěma či třema (například tím, že je do součtu uvedeme vícekrát – konstanta na věci nic nemění). Tímto způsobem dostaneme kterékoliv číslo za pomoci $O(\log^2 N)$ čtyřek, takže náš odhad časové složitosti je $O(N^2 \log^2 N)$, i když by se jistě dala prozkoumat důkladněji.

8-3-3 Nešťastný pátek

Martin Mareš

Jistě mi budete věřit, že počet inkriminovaných nešťastných pátků v daném roce závisí pouze na tom, kterým dnem v týdnu zkoumaný rok r začíná a zda je přestupný či nikoliv. Pokud budeme vědět obojí, pravděpodobně nejjednodušší možností je podívat se do tabulky na příslušný výsledek (tabulka bude mít pouhých 14 položek, což je jistě únosné). Tedy nejsou zapotřebí žádné složité cykly přes všechny měsíce roku či jiné extrémně pomalé a (nebo) kompilkované metody, které zvolili mnozí řešitelé.

Posoudit přestupnost daného roku lze velice snadno (viz pravidlo uvedené v zadání této úlohy), horší je to se dnem v týdnu. Naštěstí zde existuje jeden velice jednoduchý trik: vezmeme si nějaký "pevný bod" – rok, u nějž víme, jakým dnem v týdnu začínal, a zjistíme, o kolik se tento den v týdnu liší od hledaného (zvolíme si nějaký rok v daleké minulosti, aby byly rozdíly vždy kladné). Za každý nepřestupný rok se počáteční den v týdnu posune o 1 (365 mod 7), za každý přestupný pak o 2 (366 mod 7). Pokud si navíc náš pevný bod zvolíme jako nějaký rok r_0 ve tvaru $400k+1$ (například rok 1201), bude pro výsledný posun ve dnech δ platit (rozdíl $r - r_0$ si označíme d):

$$\delta = d + d \operatorname{div} 4 - d \operatorname{div} 100 + d \operatorname{div} 400.$$

Zkrátka a dobře vezmeme celkový počet roků (každý přispívá alespoň jedním dnem), připočteme jedničku za každý přeskočený rok dělitelný čtyřmi (tím jsme připočetli další den pro všechny přestupné roky, nicméně i pro některé další,

což spravíme za chvíli), dále odečteme jedničku pro všechny roky dělitelné stem (slíbená korekce, avšak udělali jsme další chybu – nebereme v úvahu dělitelnost 400) a konečně přičteme zpět jedničku pro všechny roky dělitelné čtyřmi sty (konečne opravena i druhá chyba).

Ovšem jak zjistíme, který den v týdnu byl 1. 1. 1201? A jak se vyrovnáme s tím, že před rokem 1582 byl jiný kalendář? Velice jednoduše – budeme počítat s *fiktivním* rokem 1201, chápaným tak, jako by se užíval gregoriánský (nynější) kalendářní systém již od tohoto roku, což neuškodí, neboť stejně máme počítat až od roku 1583. A příslušný den v týdnu prostě nastavíme (například jednoduchým vyzkoušením všech možností) tak, aby nějaký rok v současnosti (např. 1996) začínal správně (pondělím – tomu přiřadíme číslo 0). Tímto mírně špinavým trikem nemůžeme nic zkazit, neboť díky pevné diferenci v počtu dní v týdnu mezi 1. 1. 1201 a 1. 1. 1996 existuje právě jedno řešení, a tak nalezená hodnota musí být nutně správně. A dokonce k našemu úžasu vyjde, že 1. 1. fiktivního roku 1201 bylo též pondělí, takže posun je nulový!

Zajímavým důsledkem našich úvah je, že v každém roce se pátek třináctého musí vyskytnout alespoň jednou, na druhou stranu ovšem nenastane více než třikrát...

Sestrojit na základě tohoto algoritmu program je záležitostí okamžiku a jistě si to nežádá jakéhokoliv dodatečného komentáře.

```
#include <stdio.h>

static int bad_luck[2][7] = {
    { 2,  2,  1,  3,  1,  1,  2 },
    { 2,  1,  2,  2,  1,  1,  3 } };

void main (void)
{
int y,  o,  d,  l;

    printf ("Rok:␣");
    scanf ("%d",  &y);
    o = y-1201;                          /* Budeme počítat od fiktivního roku 1201 */
    o = o +  (o/4) -  (o/100) +  (o/400);      /* Posun začátku roku */
    d = o % 7;                           /* Den v týdnu pro 1. leden */
    l = ! (y%4) &&  ( (y%100) || ! (y%400));      /* Je přestupný? */
    printf ("%d\n",  bad_luck[l][d]);
}
```

8-3-4 Základní kámen

Pavel Machek

Tato úloha nebyla algoritmicky příliš složitá. Alespoň v tom smyslu, že napsat funkční program nebyl žádný problém. I přesto se ale na programu dalo

dost zkazit: Byla viděna řešení na půl stránky (jedno z nich se stalo "vzorem" vzorového řešení), ale také řešení na 2 listy (s použitím objektů).

Velké problémy byly s určováním časové a paměťové složitosti. Obě byly totiž exponenciální vzhledem k délce vstupního řetězce. (Zajímavé, mnoho lidí prostě poznamenalo, že časová složitost je lineární, ale už nenapsali k čemu.) Vysvětlení je jednoduché: Ve výrazu `((A and B) and (C and D)) ...` závisí délka výstupního výrazu na délce vstupního výrazu exponenciálně. Takže by bylo docela zajímavé, kdyby existovalo řešení lineární na délce vstupního řetězce (výstup se musí alespoň vytisknout).

Program rekurzivně zpracovává zadaný výraz (v podobě řetězců). Pro každý operátor nejprve rozvine jeho operandy a poté je dosadí do standardního "tabulkového" rozvinutí daného operátoru:

$\neg a$	$a\bar{\wedge}a$
$a \wedge b$	$(a\bar{\wedge}b)\bar{\wedge}(a\bar{\wedge}b)$
$a \vee b$	$(a\bar{\wedge}a)\bar{\wedge}(b\bar{\wedge}b)$
$a \oplus b$	$((a\bar{\wedge}b)\bar{\wedge}a)\bar{\wedge}((a\bar{\wedge}b)\bar{\wedge}b)$
$a \Rightarrow b$	$(a\bar{\wedge}b)\bar{\wedge}a$
$a \Leftarrow b$	$(a\bar{\wedge}b)\bar{\wedge}b$
$a\bar{\wedge}b$	$a\bar{\wedge}b$
$a\bar{\vee}b$	$((a\bar{\wedge}a)\bar{\wedge}(b\bar{\wedge}b))\bar{\wedge}((a\bar{\wedge}a)\bar{\wedge}(b\bar{\wedge}b))$

8-3-5 Konečně automaty

Michal Koucký

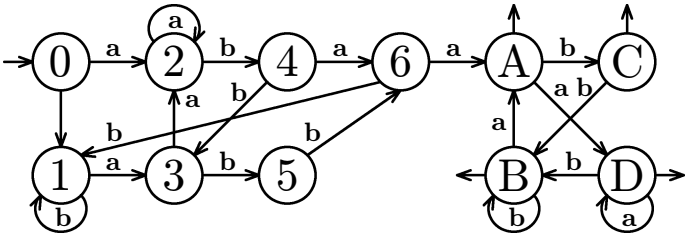
První úloha jako vždy nečinila přílišné potíže, neboť tam v podstatě žádné nebyly.

Konečný automat pro tento jazyk vypadal zhruba tak, že v první části hlídal výskyt slov **abba** i **baba**. Pokud se vyskytlo slovo **baba**, pak automat vstup nepřijímá, tudíž přešel do nějakého stavu *stoupa* nebo poslední přechod vůbec nedefinoval.

Vyskytne-li se slovo **abba**, automat přejde do své druhé části, ve které již hlídá pouze výskyty slova **baba**. Pokud toto slovo nalezne, opět přejde do nějakého stavu *stoupa* či nedefinuje přechod. Automat tedy může vypadat takto:

$$\begin{aligned} KA &= (\Sigma, Q, \delta, 1, F) \\ \Sigma &= \{\mathbf{a}, \mathbf{b}\} \\ Q &= \{0, 1, 2, 3, 4, 5, 6, A, B, C, D\} \\ F &= \{A, B, C, D\} \end{aligned}$$

$$
\begin{array}{ll}
\delta\,(0,\mathbf{a}) = 2 & \delta\,(0,\mathbf{b}) = 1 \\
\delta\,(1,\mathbf{a}) = 3 & \delta\,(1,\mathbf{b}) = 1 \\
\delta\,(2,\mathbf{a}) = 2 & \delta\,(2,\mathbf{b}) = 4 \\
\delta\,(3,\mathbf{a}) = 2 & \delta\,(3,\mathbf{b}) = 5 \\
\delta\,(4,\mathbf{a}) = 3 & \delta\,(4,\mathbf{b}) = 6 \\
 & \delta\,(5,\mathbf{b}) = 6 \\
\delta\,(6,\mathbf{a}) = A & \delta\,(6,\mathbf{b}) = 1 \\
\delta\,(A,\mathbf{a}) = D & \delta\,(A,\mathbf{b}) = C \\
\delta\,(B,\mathbf{a}) = A & \delta\,(B,\mathbf{b}) = B \\
 & \delta\,(C,\mathbf{b}) = B \\
\delta\,(D,\mathbf{a}) = D & \delta\,(D,\mathbf{b}) = B
\end{array}
$$



Sestrojit druhý z automatů však činilo problém téměř všem. Nebylo divu – hledaný konečný automat totiž sestrojit nelze.

Některá z došlých řešení předváděla automaty s nekonečným počtem stavů či automaty rozpoznávající jazyk pro nějaké omezené n. Tyto snahy jsme obvykle ohodnotili jistým počtem bodů. Na druhou stranu nás potěšilo, že se nám nikdo nesnažil vnutit ten zaručeně pravý konečný automat řešící naší úlohu.

Když tvrdíme, že takový automat neexistuje, měli bychom nějak naše tvrzení podpořit. To se v našem případě provede celkem snadno: Předpokládejme, že by nějaký takový konečný automat pro náš jazyk existoval. Takový automat má nějaký počet stavů (konečný), označme si jej n.

Nyní se podívejme, jak náš hypotetický automat zpracuje slova 1^i pro i od jedné do $n+1$. Jelikož náš automat má konečný počet stavů, musí pro nějaká dvě různá i – označme si je k, l – přejít po načtení slova 1^i do stejného stavu. Toto plyne z toho, že stavů je n a testovaných slov $n+1$.

Co se ovšem stane, když bychom spustili náš automat na slova 1^k0^k a 1^l0^k. Zřejmě náš automat přejde pro obě slova do stejného stavu, neboť po přečtení úseku jedniček je v obou případech ve stejném stavu a dále již výpočet pokračuje v obou případech stejně. Avšak pozor! Slovo 1^k0^k jsme přijmout měli, ale slovo 1^l0^k nikoliv. Přitom jsme v jednom stavu, který je buď přijímající, nebo ne. Tudíž obě slova jsou buď přijata, nebo ne. To je ovšem chyba.

Tedy někde v našich úvahách musí být chyba. Kde? Jedině v předpokladu,

že jsme ten automat našli, protože vše ostatní již plynulo pouze z této skutečnosti. Tím jsme ukázali, že automat pro daný jazyk neexistuje a existovat nemůže. (Pro znalce: použili jsme důkaz sporem.)

Tento důkaz byl aplikací obecnějšího tvrzení, tzv. Nerodovy věty. Tuto větu však v této chvíli rozebírat nebudeme a dychtivý čtenář si ji snadno vyhledá v již mnohokráte zmiňované knize pana Chytila.

8-4-1 Cesta tam a zase zpátky

Jan Kotas

Překvapilo nás, že tuto úlohu nikdo nevyřešil správně. Přitom její obtížnost je s ostatními úlohami srovnatelná. Je to asi tím, že to není žádný zprofanovaný evergreen.

Každému, kdo se trochu vyzná v teorii grafů, muselo být jasné, že jde o *minimální zesilněsouvislení orientovaného grafu.*

Nadefinujme si nejdříve několik pojmů v duchu zadání:

Okres (jinak též silně souvislá komponenta) je maximální množina měst, ve které je možné se dostat z každého města do každého. *Vstupní okres* je ten, do kterého nevedou žádné silnice. *Výstupní okres* je ten, ze kterého nevedou žádné silnice (jistě existuji i vstupně-výstupní okresy).

Je zřejmé, že hledaný algoritmus musí do každého vstupního okresu zavést alespoň jednu silnici a zároveň z každého výstupního okresu vyvést alespoň jednu silnici. To znamená, že musí přídat minimálně tolik, kolik činí maximum z počtu vstupních a výstupních okresů – tyto počty totiž nemusí být stejné.

Algoritmus, který přidává právě tento minimální počet silnic, tedy najde vždy optimální řešení. Tuto vlastnost splňuje např. algoritmus následující:

1. Dokud existují vstupní okres A a výstupní okres B takové, že z okresu A nevede cesta do okresu B, zaveď silnici z okresu B do okresu A (resp. z libovolného města okresu B do libovolného města okresu A).

V tomto kroku se určitě nezmění počet okresů ve státě – ke snížení počtu okresů může dojít pouze tehdy, když se mezi nějakými vytvoří cyklus, což je vzhledem k podmínce nemožné; zvýšení počtu okresů přidáním silnice taktéž není možné. S každou přidanou silnicí ubyde právě jeden vstupní a právě jeden výstupní okres. Nové vstupní resp. výstupní okresy se nevytvoří.

2. Dokud existuje vstupní okres A a od něj různý výstupní okres B, zaveď silnici z okresu B do okresu A.

V tomto kroku se přidáním každé silnice spojí nějaké okresy v jeden. Zároveň ubyde právě jeden vstupní a právě jeden výstupní okres. Nový vstupní resp. výstupní okres vznikne pouze tehdy, když před přidáním hrany existoval právě jeden vstupní resp. výstupní okres (vyplývá z podmínky v prvním kroku algoritmu).

Po ukončení druhého kroku algoritmu exituje pouze jeden vstupně výstupní okres – tedy je možné se dostat z každého města do každého.

Program si zadané silnice ukládá do matice sousednosti. Z této matice pak klasickým algoritmem spočte matici dostupnosti (tranzitivní uzávěr) o složitosti $O(N^3 \log N)$.

V této matici pak hledá silnice, které je nutné přidat, podle výše uvedených pravidel. Po každém přidání silnice patřičně opraví matici dostupnosti. Vzhledem k tomu, že silnic nikdy nebude přidáno více jak N, je složitost implementace každého z kroků algoritmu $O(N^3)$.

Časová složitost implementace je tedy: $O(N^3 \log N)$. Použitím vhodnějších datových struktur je možné časovou složitost snížit, paměťové nároky jsou $O(N^2)$. Algoritmus je zjevně konečný.

8-4-2 Ďábelské mocniny

Martin Mareš

Tato úloha posloužila jako dokonalý "chyták" – polovina účastníků použila rozkladu na prvočinitele (nejspíše se inspirovala úlohou následující), který ovšem má k optimálnímu řešení velice daleko. Jeden protipříklad za všechny: počítáme $a^{2^{20}}$. Rozklad na prvočinitele nám dá stejný počet násobení (1048575), jako kdybychom počítali přímo, zatímco zvolím-li $a_0 = a$, $a_{i+1} = a_i \cdot a_i$, pak a_{20} bude kýžený výsledek za pomoci pouhých 20 násobení.

Základním trikem je tedy (ne, pravda, k dosažení opravdu *optimálního* počtu násobení – to je netriviální, ale asymptoticky optimálního ano) rozklad exponentu do dvojkové soustavy (tedy jako $b = 2^{a_0} + \cdots + 2^{a_k}$, kde $k \le \log_2 b$), čímž dostaneme $a^b = a^{2^{a_0}} \cdot \ldots \cdot a^{2^{a_k}}$, přičemž všechna $a^{2^{a_i}}$ dokážeme postupným umocňováním na druhou spočítat v čase $O(\log N)$ a poté je v témže čase vynásobit, tedy celkově $O(\log N)$.

Největší číslo dosažitelné k násobeními je jistě a^{2^k} (důkaz nechť si laskavý čtenář provede sám), tudíž alespoň pro čísla tohoto tvaru není lepší řešení než logaritmické, pročež asymptoticky to lépe než logaritmicky nelze.

```
int mocnina (int a, int b)
{
    int k = 1;                          /* Právě zkoumaná mocnina dvojky */
    int l = 1;                          /* Mezivýsledek */
    while (k<b) k = k*2;                /* Hledáme nejvyšší potřebnou mocninu */
    while (k)                           /* A násobíme... */
      {
        l = l*l;                        /* Trik – zkuste to pochopit... */
        if (k & b)
          l = l*a;
        k = k/2;                        /* Nižší mocninu */
```

```
        }
    return l;
}
```

8-4-3 Top secret *Pavel Machek*

Uvedený příklad je speciálním případem velmi praktické úlohy, na kterou ale není známo opravdu rychlé řešení: rozkladu na prvočinitele. Toho se hojně využívá v šifrování – mnoho šifrovacích algoritmů je založených na tom, že rozložit na prvočinitele *opravdu velké* číslo nikdo v rozumném čase neumí. Dokonce se pořádají soutěže (RSA factoring challenge a jiné) v rozkladu čísel – firmy zabývající se šifrováním zajímá, která čísla je možno rozložit a která ne.

Pro zajímavost uvádím, že s pravděpodobností kolem 20% je možné rozložit náhodně zvolené 250-ti místné číslo.

Vhodnou metodou pro daný příklad je "triviální dělení" – pro všechna čísla $i \in \{2, \ldots, \sqrt{N}\}$ zkusíme spočítat N mod i a vyjde-li 0, máme řešení. Vzhledem k zadání (číslo je součinem dvou prvočísel) lze druhé prvočíslo dostat jako n/i a není nutné zjišťovat, jestli není $i \leq \sqrt{N}$. Dále není nutné testovat prvočíselnost čehokoliv – pokud jdeme od dvojky nahoru, tak první číslo, které bude dělit N bude určitě prvočíslo. Popsaný algoritmus má časovou složitost $O(\sqrt{N})$. Jistého zlepšení (ovšem pouze konstanta krát) lze dosáhnout tím, že se na začátku zjistí, jestli N není dělitelné 2-mi nebo 3-mi (v případě, že ano, je výsledek jasný), a v případě že ne, testovat jen čísla tvaru $6k - 1$ a $6k + 1$.

Poměrně častou chybou byly různé pokusy o dělení pouze prvočísly. Takový algoritmus měl typicky složitost kolem $O((\sqrt{N})^2) \approx O(N)$, t.j. výrazně pomalejší než jednoduché řešení.

8-4-4 Penězokazi *Martin Bělocký*

Tuto úlohu všichni řešitelé nějak vyřešili. Potíž bylo v tom "nějak". Záleželo totiž na tom, jak jste pochopili zadání úlohy. My jsme dle zadání považovali přístup k vnější paměti za daleko pomalejší, než je přístup k hlavní paměti, které však bylo pouze 4 kB = 4096 B = 32768 bitů. Data bankovek byla také uložena ve vnější paměti, kdo by se přeci namáhal psát skoro 100000 čísel.

Obecně existují dvě možnosti, jak si zapamatovat číslo – mít ho uložené v datové části, nebo si ho pamatovat pomocí stavu programu (ti, co pochopili teorii automatů z příkladu na pokračování, vědí, o co jde). Druhou možnost ovšem vylučuje limit velikosti programu uvedený v zadání.

Většina účastníků úlohu řešila z jiného pohledu, totiž aby onen program běžel na vašem počítači pod vaším systémem rychle. Chtěli jsme jako vždy řešení nezávislé na systému. Podle toho také řešení mělo vypadat. Bohužel

spousta řešitelů používala speciální operace `blockread/write`, které nejsou na jiných systémech definovány. Naše řešení se o to opírat nemůže – nevíme nic o tom, jak ona vnější paměť vypadá, a pokud je to disk, tak jaká je jeho struktura – ta vypadá podle toho jak ji připravil operační systém. Tedy již chapete, že nelze považovat `blockread` stovky čísel za jeden přístup k vnější paměti, nýbrž za 100 přístupů. Pokud toto vezmeme v úvahu, tak bychom si správné řešení představovali takto:

Na vstupu máme velké množství čísel (okolo 100000) typu `longint`. Definujeme si pole bitů o velikosti 32400, tj. 4050 bytů, zbytek ponecháme na proměnné. Jednou přečteme celý vstup a jsme schopni říci, zda se v intervalu 1..32400 něco opakuje, neboť pro každé číslo z tohoto intervalu si budeme pamatovat, zda jsme ho už četli, nastavením bitu v poli s pořadím hodnoty onoho čísla. Např. pro číslo 127 nastavíme 127. bit. Poté potřebujeme ještě otestovat soubory na disku, které jsme vytvořili. Pro každý soubor (jsou celkem nejvýše 4), existuje-li, vymažeme celé bitové pole, načítáme čísla a opět přitom nastavujeme bity v poli jako v prvním případě.

Složitost algoritmu: Je zřejmé, že je lineární, nás však zajímá počet přístupů na vnější paměť (disk), které program nejvíce zpomalují. Nejprve čteme $4N$ bytů (N longintů), přičemž přibližně $2/3 \cdot N$ čísel typu integer opět zapisujeme, načež je čteme zpět. Celkem tedy přeneseme mezi diskem a vnitřní pamětí přibližně $7N$ bytů.

Poznámka k programu pro neznalé bitových operací: `shr` je rotace bitů doprava – např: A `shr` 2 vydělí celé kladné číslo čtyřmi. Podobně A `shl` 1 násobí dvěma. Je to daleko rychlejší než dělení či násobení. A `and` 15 je bitový součin, což je totéž, jako A mod 16, ale opět rychlejší, neboť se neprovádí žádné dělení.

```
Program penezokazi;
type word=0..64000;
var data:array[0..4049] of byte;
    source:file of longint;
    f:array[1..3] of file of word;
    x:array[1..3] of word;                      {počet čísel v jednotlivých souborech}
    a,b:word;
    pom:longint;
begin
for a:=0 to 4049 do data[a]:=0;
assign(source,'sss.sss');
assign(f[1],'cesta_2');
assign(f[2],'cesta_3');
assign(f[3],'cesta_4');
reset(source);
while not eof(source) do
begin
  read(source,pom);                             {čtení vstupu}
  if(pom>=1)and(pom<=32400) then begin          {test na 1. interval}
          dec(pom);
```

```
          if (data[pom shr 3] and (1 shl (pom and 15)))=0 then
          {test a nastavení bitů v poli}
             data[pom shr 3]:=data[pom shr 3] or (1 shl (pom and 15))
          else writeln('nasla se falesna bankovka',pom);
          end
  else if (pom>=32401)and(pom<=64800)then begin   {1. soubor - vzhledem k velikosti}
               inc(x[1]);                         {intervalu už nemusím ukládat longinty}
               if x[1]=1 then rewrite(f[1]);      {1.číslo intervalu založí soubor}
               a:=pom-32401;
               write(f[1],a);
               end
  else if (pom>=64801)and(pom<=97200) then begin  {2. interval se ukládá do 2. souboru}
               inc(x[2]);
               if x[2]=1 then rewrite(f[2]);
               a:=pom-64801;
               write(f[2],a);
               end
  else if (pom>=97201)and(pom<=99999) then begin   {3. malý interval do 3. souboru}
               inc(x[3]);
               if x[3]=1 then rewrite(f[3]);
               a:=pom-97201;
               write(f[3],a);
               end
  else write (' v souboru je falesna bankovka mimo rozsah serie');
  end;
  close(source);
  for a:=1 to 3 do   {načítám jednotlivé soubory }
    if x[a]>0 then begin   {existuje soubor ? }
      if x[a]>32400 then
        { velikost intervalu je 32400, je-li čísel více, pak je jistě nějaká falešná! }
        write ('bankovek je prilis hodne->v souboru je falesna bankovka');
      reset(f[a]);
      for pom:=0 to 4049 do data[pom]:=0;
      while not eof(f[a]) do
      begin
        read(f[a],b);
        if (data[b shr 3] and (1 shl (b and 15)))=0 then
         data[b shr 3]:=data[b shr 3] or (1 shl (b and 15))
        else writeln('nasla se falesna bankovka');
      end;
      close(f[a]);
    end;
end.
```

8-4-5 (Ne)konečné automaty *Michal Koucký*

Došlých řešení této úlohy nebylo mnoho a ne všechna byla správně. Mnozí řešitelé se pustili pouze do části **a**, tj. do sestrojení předepsaného automatu. Část **b**, tj. napsání generátoru konečných automatů, řešilo pouze asi 15 řešitelů, ideální řešení však k naší lítosti nenalezl nikdo, ač mnozí byli blízko.

Nejvíce problémů s částí **a** bylo v pochopení, jaký jazyk se má rozpoznávat. Zadání říkalo něco takového:

$$L_1 = \{w \in \{\mathbf{a},\mathbf{b}\}^* \,;\; w \text{ obsahuje podslovo } \mathbf{bbba}\ \}$$
$$L_2 = \{w \in \{\mathbf{a},\mathbf{b}\}^* \,;\; w \text{ obsahuje podslovo } \mathbf{baaa}\ \}$$

$$L_3 = \{w = xy\ ;\ \ x \in L_1 \text{ a zároveň } y \in L_2\ \}$$

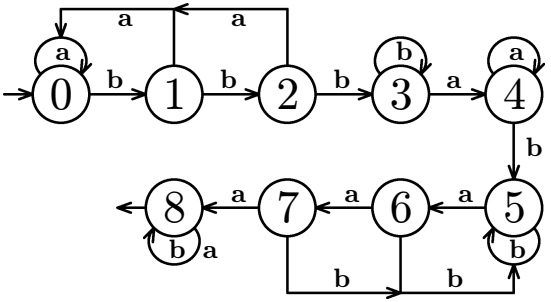
což znamená tolik, že slova z jazyka L_3 se mají dát rozdělit na dvě nepřekrývající se části, a to tak, že první část bude z prvního jazyka, tedy bude obsahovat slovo **bbba**, a druhá bude z druhého jazyka, tedy bude obsahovat slovo **abbb**.

"No problem," řekl nejspíše každý, kdo správně pochopil zadaní, sedl a automat napsal.

Ani my nemáme v podstatě k automatu co dodat, a tudíž Vám jej rovnou předkládáme:

$$\begin{aligned}
KA &= (\Sigma, Q, \delta, D_0, F)\\
\Sigma &= \{\mathbf{a}, \mathbf{b}\}\\
Q &= \{D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}\\
F &= \{D_8\}
\end{aligned}$$

$$\begin{array}{ll}
\delta\,(D_0, \mathbf{a}) = D_0 & \delta\,(D_0, \mathbf{b}) = D_1\\
\delta\,(D_1, \mathbf{a}) = D_0 & \delta\,(D_1, \mathbf{b}) = D_2\\
\delta\,(D_2, \mathbf{a}) = D_0 & \delta\,(D_2, \mathbf{b}) = D_3\\
\delta\,(D_3, \mathbf{a}) = D_4 & \delta\,(D_3, \mathbf{b}) = D_3\\
\delta\,(D_4, \mathbf{a}) = D_4 & \delta\,(D_4, \mathbf{b}) = D_5\\
\delta\,(D_5, \mathbf{a}) = D_6 & \delta\,(D_5, \mathbf{b}) = D_5\\
\delta\,(D_6, \mathbf{a}) = D_7 & \delta\,(D_6, \mathbf{b}) = D_5\\
\delta\,(D_7, \mathbf{a}) = D_8 & \delta\,(D_7, \mathbf{b}) = D_5\\
\delta\,(D_8, \mathbf{a}) = D_8 & \delta\,(D_8, \mathbf{b}) = D_8
\end{array}$$



Část **b** byla o poznání těžší než část předešlá, a proto ji také málokdo řešil a vyřešil.

Nejlepší došlá řešení pracovala v čase $O\,(N^2)$, kde n je délka hledaného slova. Tato řešení obvykle získala plný počet bodů, ostatní řešení, pokud vůbec fungovala, dostala méně.

Správný algoritmus se dal vytvořit ve dvou krocích. V prvním dosáhneme časové složitosti $O\,(A \cdot N)$, kde A je velikost abecedy, a ve druhém pak

toto řešení drobnou úpravou převedeme na řešení s lineární časovou složitostí $O(N)$. Lepšího času lze jistě těžko dosáhnout (minimálně musíme to slovo projít, abychom věděli, pro co generujeme automat).

Základní otázkou, kterou si každý řešitel musel položit a obvykle si na ni i odpovědět, bylo, jak vůbec může náš hledaný automat vypadat.

Správná odpověď na tuto otázku byla ta, že bude mít $n + 1$ stavů. Každý stav bude mít přiřazeno jméno, které odpovídá nějaké počáteční části hledaného slova, a to tak, že i-tému stavu bude odpovídat prvních i znaků hledaného slova. Nultému stavu samozřejmě odpovídá prázdný řetězec a poslednímu stavu celé hledané slovo.

Od automatu pak budeme požadovat to, aby v průběhu jeho výpočtu nad vstupním textem označení stavu, ve kterém se nachází, odpovídalo tomu, co zatím přečetl, přesněji – zakončení zatím přečteného kusu vstupního textu.

Například pro hledaní slova "ksp" budeme mít automat se čtyřmi stavy: '', 'k', 'ks', 'ksp', který se po přečtení úseku textu "ošklivý ks" bude nacházet ve stavu 'ks'.

Dalším požadavkem na tento automat bude to, že pokud se automat může nacházet ve více stavech, pak se bude nacházet ve stavu s delším názvem (srovnej stavy 'šošo'a 'šo' pro slovo "šošolka" po přečtení "občané byli šo" a "pták měl velkou šošo".) Splnění této podmínky nám zajistí jednu podstatnou vlastnost automatu – v okamžiku, kdy se na konci zatím přečteného kusu textu vyskytne hledané slovo, budeme ve stavu, který je označen tímto celým hledaným slovem a nikde jinde. Nikdy jindy se tam také dostat nemůžeme, a tedy pokud se tam dostaneme, pak jsme vyhráli, našli jsme hledané slovo, už se odsud nehneme a zbytek textu ignorujeme.

Skončí-li tedy automat výpočet v tomto posledním stavu, musel text obsahovat hledané slovo. Pokud ne, pak ani text toto slovo neobsahoval.

Zbývá už jen zodpovědět otázku, jak bude vypadat přechodová funkce našeho automatu.

Je zřejmé, že všechny přechody z posledního stavu povedou zpět do tohoto stavu.

Dále z toho, co jsme řekli vyplývá, že nejlépe bude, pokud tuto funkci budeme pro ostatní stavy sestrojovat postupně od nultého stavu do předposledního.

Přechodová funkce z nultého stavu bude vypadat tak, že pro všechny znaky mimo jednoho bude vést zpět do tohoto nultého stavu. Vyjímku tvoří první znak hledaného slova, pro který se přesuneme do následujícího, prvního, stavu.

Je vidět, že toto splňuje naše podmínky, aby stav, ve kterém jsme, odpovídal tomu, co jsme přečetli. Pokud jsme totiž v nultém stavu, pak konec přečteného textu neodpovídá žádnému počátečnímu úseku hledaného slova, tudíž přečtením dalšího znaku se to určitě nemůže zlepšit, resp. nemůžeme se

pohnout dále než na první stav, kam se pohneme, právě když přečteme první písmenko hledaného slova.

Pro nultý stav máme tedy přechodovou funkci nalezenou. Jak ji nalezneme pro další stavy?

Pojďme generovat přechodovou funkci pro stav s označením y, přičemž pro kratší stavy ji už známe. Vezměme si stav s třeba i prázdným označením u takovým, že $y = xu$, pro nějaké slovo x, a u je kratší než y. Zřejmě takový stav u existuje – minimálně nultý stav s prázdným označením – a už pro něj jistě známe přechodovou funkci.

Pokud se automat dostane do stavu y, znamená to, že slovo zatím končilo na toto slovo, tedy na xu, neboli pokud odhlédneme od požadavku, že se vždy nacházíme v nejdelším možném stavu, mohli bychom být klidně ve stavu u. Všechny přechody ze stavu u by se tedy daly použít i pro stav y, s tím, že by však mohlo dojít k porušení podmínky maximální délky, čili pokud se po přečtení nějakého znaku Z ze stavu u přejde do stavu w, pak by se tam klidně mohlo přejít i z y. ($w = vZ, u = cv, y = xu \Rightarrow y = xcv \Rightarrow$ přechod z y do w neporuší podmínku o jménech stavů)

Pokud bychom si však původně ze všech možných stavů u vybrali ten s nejdelším označením, pak ve všech případech, kromě přechodu pro další znak hledaného slova, který se musí nastavit zvlášť (viz. dále), k porušení podmínky maximálnosti nedojde. To z toho důvodu, že přechody ze stavu u jsou již pro stav u maximální. Pokud by se totiž pro nějaký znak Z' mělo přejít ze stavu y do nějakého stavu w' kratšího než je y, pak to znamená, že $w' = v'Z'$ a $y = pv'$, pro nějaká v' a p. Ovšem to speciálně znamená, že v' je také stav našeho automatu kratší než y, který je taktéž jeho zakončením, ale z výběru u, jakožto maximálního stavu zakončujícího y vyplývá, že $v'Z'$ musí být i přechod pro u, protože v' musí být též zakončením u.

Nakonec nadefinujeme přechodovou funkci pro stav y a pro znak, kterým pokračuje hledané slovo, jako přechod do stavu s o jedna delším označením než y.

Shrnutí: přechodová funkce pro stav y je stejná jako funkce pro nejdelší stav zakončující y s tou výjimkou, že pro další znak je rovna dalšímu stavu za y.

Z toho okamžitě vyplývá náš algoritmus.

Zbývá pouze zodpovědět otázku, jak rychle nalézt pro stav y nejdelší jeho zakončení. Odpověď je nasnadě. Do stavu y jsme přešli ze stavu y' přes nejaký znak. Hledaný stav u je stav, kam by přešlo y', kdyby nemuselo přejít do y, což je stav, kam by vedla přechodová funkce, kdyby se při hledání přechodové funkce pro y' nepoužilo na závěr pravidlo o výjimce pro pokračovací znak. Stačí si tedy potřebnou informaci zjištěnou při generování přechodové fce pro y' zapamatovat pro dobu generování funkce pro y.

Takový algoritmus bude mít časovou složitost $O(A \cdot N)$, protože pro každý stav musíme opsat přechodovou funkci nějakého předchozího stavu a změnit ji v jednom bodě, tedy n-krát provedeme opsání, které trvá $O(A)$.

Tento algoritmus se dá ještě zlepšit na časovou složitost $O(n)$, když si všimneme, že většina přechodů jde zpět do nultého stavu ''. Pro každý stav si tedy bude stačit pamatovat pouze ty přechody, které nevedou zpět do nultého stavu. Těchto přechodů je maximálně $2N$, včetně těch do následujícího stavu. Jelikož časová složitost našeho algoritmu spočívala v kopírování přechodové funkce, tak ježto nyní kopírujeme maximálně $2N$ položek, máme časovou složitost $O(n)$.

Důkaz toho, že je maximálně $2N$ netriviálních přechodů zde nebudeme uvádět. Vyplývá to z činnosti jiného možného algoritmu, řešícího naši úložku, jehož časová složitost sice je $O(N^2)$, ale nalezne pouze těchto $2N$ netriviálních přechodů.

Paměťová složitost našeho algoritmu je pro pomalejší verzi $O(A \cdot N)$ a pro rychlejší $O(N)$.

Autorský program uvádíme pouze pro druhou (rychlejší) verzi algoritmu. V poli `ekviv` je pro každý stav uloženo číslo nejdelšího menšího stavu, který je jeho zakončením. (Toto pole není ve skutečnosti potřeba, protože čteme vždy poslední zapsanou položku a místo něj by stačila jedna proměnná, je zde však z didaktických důvodů.)

V poli `delta` je uložena přechodová funkce hledaného automatu. V první verzi programu je tam uložena vždy celá, ve druhé pak pouze její netriviální část. Více již komentáře přímo v programech.

```
program KSP845;         { Konecne automaty v case O( n ) }

const MaxDelka = 200;    { Maximalni delka slova }

var delta : array [0..2*MaxDelka] of   { Netrivialni slozka prechodove funkce }
            record
               znak : char;          { Znak pro tento prechod }
               kam : integer;        { Kam se prejde }
            end;
  pocdelta : integer;                { Pocet polozek v poli delta }
  stavy : array [0..MaxDelka] of     { Informace pro jednotlive stavy }
         record
           bdelta: integer; { Zacatek useku prechodove funkce v poli delta pro tento stav}
           ndelta: integer; { Delka useku prechodove funkce v poli delta pro tento stav}
           ekviv : integer; { Cislo nejdelsiho kratsiho "ekvivalentniho stavu}
         end;
  stav : integer;        { Pomocna promenna }
  i,j : integer;         {    - " -         }
  s : string;            { Vstupni slovo, pro ktere vytvarime KA }

begin
   repeat                { Nacteme vstup }
   write('Zadej vstupni slovo : ');readln(s);
```

```
  until (s<>'') and (length(s)<MaxDelka);

  stavy[0].bdelta:=0;   { Delta pro nulty stav }
  stavy[0].ndelta:=1;
  delta[0].znak:=s[1];
  delta[0].kam:=1;
  pocdelta := 1;

  stavy[1].ekviv:=0;      { Prvni stav je ekv. nultemu }
                  { Spocteme prechodovou funkci v ostatnich stavech krome posledniho }
  for stav:=1 to length(s)-1 do
   begin
      stavy[stav+1].ekviv:= 0; { Nastavime ekviv dalsiho stavu prozatim na nulu }

      stavy[stav].bdelta:= pocdelta;  { Nastavime pocatecni hodnoty }
      stavy[stav].ndelta:= 1;
                             { Nastavime implic. prech. fci. tohoto stavu }
      delta[pocdelta].znak := s[stav+1];
      delta[pocdelta].kam  := stav+1;
      pocdelta := pocdelta+1;

      { Prechodova funkce je az na s[stav+1] stejna jako ekviv }
      for i:=stavy[stavy[stav].ekviv].bdelta to
        stavy[stavy[stav].ekviv].bdelta + stavy[stavy[stav].ekviv].ndelta - 1 do
                  { Pokud ma ekv. stav prechod pro s[stav+1], }
                  { pak je to ekv. nasledujiciho stavu }
         if delta[i].znak = s[stav+1] then stavy[stav+1].ekviv := delta[i].kam
         else
          begin     { Jinak nastavime rechodovou funkci tohoto stavu }
             delta[pocdelta]:=delta[i];
             stavy[stav].ndelta := stavy[stav].ndelta + 1;
             pocdelta := pocdelta + 1;
          end;
   end;

  { Vystup }
  writeln('Konecny automat pro toto slovo je tento:');
  writeln('KA = ( { predepsana abeceda }, Q, delta, D0, { D',length(s),' } )');
  writeln(' Q = { D0 .. D',length(s),' }');

  for stav:=0 to length(s)-1 do
     for i:=stavy[stav].bdelta to stavy[stav].bdelta+stavy[stav].ndelta - 1 do
        writeln('delta( D',stav,' ''',delta[i].znak,''' ) = D',delta[i].kam);
  writeln('delta( D',length(s),', pro vsechny znaky ) = D',length(s));
  writeln('Vsechny vyse neuvedene prechody jsou do stavu D0');
end.
```

8-5-1 Rozmarný princ *Martin Bělocký*

Úkolem bylo najít dobrou strategii, díky níž by princezna pokud možno vždy vyhrála. Většina řešitelů psala pouze programy, které by sice princezně pomohly, ale pouze v jistých situacích, a ke všemu neuvedli důkazy svých tvrzení, takže si ani nemohli být správností jisti. Je pravda, že pokud je na počátku na všech hromádkách stejné množství sirek a začíná princ a počet hromádek je sudý, tak stačí opakovat tahy po princi, ale na jiné hromádce, a tento postup vede jistě k vítězství, leč toto vítězství má velké předpoklady a pravděpodobnost splnění těchto předpokladů je malá.

Definice: Nechť M je maximální počet najednou odebratelných sirek z jedné hromádky, N je počet hromádek, h_i je i-tá hromádka, $i = 1, 2, \ldots, N$, p_i je počet sirek na h_i. *Strom hry* je struktura složená z *uzlů* a *hran.* Uzly odpovídají herním pozicím, každý uzel nese úplnou informaci o všech hromádkách. Kořen stromu odpovídá výchozí pozici. Z každého uzlu vede tolik hran, kolik je z dané pozice možných regulérních tahů. Algoritmus generuje sekvence *půltahů* z dané pozice, dokud není dosaženo pozice, která je pro hráče na tahu vyhraná. Pokud projde všechny možnosti a nenajde pozici, která je pro hráče vyhraná, odebere jednu sirku z největší hromádky. Půltahem je myšleno odebírání sirek jednoho hráče, tj. např. princezny. *Vyhraná pozice* je taková, že ať z této pozice protihráč odebírá jakkoliv, potom hráč používající algoritmus uvedený níže určitě vyhraje.

Algoritmus: Vyberu jen ty hromádky, pro něž platí $p_i \bmod (M+1) \geq 1$. Nanaleznu-li žádnou takovou, neexistuje z aktuální pozice sekvence půltahů vedoucí k výhře a skončím. Pro všechny vybrané hromádky provedu: Je-li $p_i > 1+M$, pak $p_i := p_i \bmod (M+1)$. Pro takto vzniklé hromádky provedu: Zkusím odebrat z libovolne hromádky nějaký přípustný počet sirek, pokud nelze udělat žádné přípustné odebrání, aniž bych prohrál, skončím s výsledkem *false,* jinak pokračuji.

Pro daný výběr rekurzivně zavolám algoritmus od prvního kroku (generuji půltah soupeře). Pokud se z rekurze vrátím s výsledkem *true,* provedu jiný výběr sirek (jiný počet a z jiné hromádky tak, abych postupně prošel všechny možnosti). Pokud se vrátím s výsledkem *false,* zapamatuji si hromádku H_1, ze které jsem vybíral, a počet sirek, který jsem z ní odebral. Našel jsem půltah vedoucí k vítězství a skončím s výsledkem *true.* Pokud vždy dostanu *true,* neexistuje v tuto chvíli půltah vedoucí k vítězství, a tak skončím s hodnotou *false.*

Pokud tento postup skončí s výsledkem *true,* existuje vyhrávající posloupnost. Ze vstupní množiny hromádek odeberu ze zapamatované hromádky (H_1) zapamatovaný počet sirek. Pak nechám hrát soupeře a zapamatuji si, ze které hromádky odebíral a kolik odebral (P). Pokud odebral z hromádky, na níž je

ještě více než $1+M$ sirek, pak z ní odeberu $M+1-P$ sirek. Pokud ne, postupuji od kroku 1.

Tvrzení: Pokud bude jednou výsledkem výše uvedeného algoritmu, podle něhož hraje počítač, hodnota *true,* pak hráč, jenž je právě na tahu, používající tento postup, vyhraje.

Důkaz: (nejprve pro jednu hromádku)

- T_1: Jsem-li na tahu a je-li na hromádce některý z tohoto počtu sirek: $2 \ldots M+1$ (je to vyhraná pozice), pak mohu odebrat tak, aby na hromádce zbyla jen jedna, a tudíž soupeř prohraje.
- T_2: Přidám-li k vyhrané pozici (např. s P sirkami) $M+1$ sirek, pozice zůstane stále vyhraná, neboť odebere-li soupeř z hromádky P sirek, mohu odebrat $M+1-P$ sirek, abych se dostal do původní vyhrané pozice. Důsledek: Je-li na hromádce $K \cdot (M+1)+1$ sirek (K je přirozené či nula), je tato hra pro hráče na tahu prohraná.
- T_3: Je-li na hromádce P sirek a $P \bmod (M+1) = 1$, pak z této pozice nelze vyhrát – je to důsledek T_1 z pohledu protihráče.

Mějme nyní více hromádek... Definujme hru H_1 tak, že vypustíme z hry hromádky, na nichž je tolik sirek, že platí $P \bmod (M+1) = 1$ (podmínka P_1), kde P je počet sirek na dané hromádce.

- T_5: Jsem-li na tahu, stačí hledat sekvenci půltahů pouze ve hře H_1. Nalezu-li v H_1 sekvenci půltahů vedoucí k vítězství a pokud soupeř odebere z některé z hromádek, které nejsou v H_1 (např. hromádka H_i), pak lze dle T_2 z téže hromádky odebrat sirky tak, aby podmínka P_1 stále platila, a tudíž se opět dostaneme do vyhrávající pozice.
- T_6: Jsem-li na tahu a algoritmus skončí s výsledkem *true,* pak nenašel pro pozici soupeře žádnou sekvenci půltahů vedoucí k jeho vítězství. Pak ať odebere soupeř cokoliv, prohraje. Provede-li soupeř tah, potom lze opakováním algoritmu, tj. průchodem stromu pozic, opět nalézt vyhrávající pozici, neboť algoritmus předtím prozkoumal všechny soupeřovy půltahy. Pokud algoritmus skončí s výsledkem *false,* pak odebere jednu sirku z největší hromádky.

Celý algoritmus je obdoba minimaxu a jeho důkaz je uveden v mnoha knihách, proto je zbytečné jej rozepisovat do podrobností. Tento univerzální způsob se používá i v jiných hrách, jako jsou třeba šachy, dáma, piškvorky, ... Složitost algoritmu je však značná díky rekurzivnímu prohledávání. Velkou roli hraje konstanta M, díky níž se nemusí prohledávat všechny možnosti, ale pouze hromádky do velikosti $M+1$. Složitost v tomto rozmezí je však nepolynomiální vzhledem k počtu sirek a hromádek.

8-5-2 Telefon do pekla *Martin Mareš*

Tento pekelný spojařský problém je možno vyřešit velice efektivně za pomoci metody "Rozděl a panuj", tedy rozdělením úlohy na dvě menší části, s nimiž se později vypořádáme analogicky. Většina účastníků si tento postup zvolila a více či méně elegantně tak dosáhla časové složitosti $O(N \cdot \log N)$, stejně jako u následujícího algoritmu:

Mějme "levé konce" označené $l_1, \ldots, l_n$ a "pravé konce" $r_1, \ldots, r_m$, o nichž víme, že jsou navzájem propojeny s výjimkou případů, kdy drát nevede (musíme počítat s tím, že n nemusí být rovno m – to nastane u kabelů s nevodivými dráty). Nyní rozdělíme levé konce na dvě pokud možno stejně velké části – řekněme $l_1, \ldots, l_k$ a $l_{k+1}, \ldots, l_n$ a k jedné (řekněme první) časti připojíme napětí, zatímco od druhé jej odpojíme. Poté změříme napětí na pravých koncích – ty, na kterých něco naměříme, jsou jistě připojeny pouze k levým koncům z první části (tedy k těm, na něž jsme napětí připojili), zatímco ty zbylé jsou buďto připojeny k levým koncům z části druhé nebo nepřipojeny. Takto jsme rozdělili dráty na dvě (přibližně stejně velké) části, které můžeme zpracovávat stejným způsobem.

Některé případy jsou ovšem natolik triviální, že si zasluhují zvláštní pozornost:

- $n = 0 \wedge m > 0$: k pravým koncům není evidentně nic připojeno (kdyby bylo, jistě by se to dle předpokladů nacházelo mezi zkoumanými levými konci, které ovšem žádné nemáme).
- $n = 1 \wedge m = 1$: pokud byl na r_1 alespoň jednou naměřen proud, jistě je l_1 propojen s r_1. Pokud nikoliv, je to třeba vyzkoušet, neboť je zde ještě možnost, že by byl drát přerušen.
- $n > 0 \wedge m = 0$: k levým koncům není evidentně nic připojeno.

Ukázkový program pracuje přesně podle tohoto algoritmu – funkce `test` zabezpečuje otestování nějakých k sobě patřících levých a pravých konců, přičemž její argumenty udávají první a poslední číslo zkoumaných levých konců (jistě můžeme postupovat tak, abychom zkoumali vždy souvislý úsek), zatímco pravá čísla konců k těmto patřící jsou uložena v poli r a je předán pouze počátek a konec úseku v tomto poli. Dále je funkci dodána informace o tom, zda je úsek zrovna zapnut nebo vypnut, a protékal-li úsekem již alespoň jednou proud.

Funkce `test` po ošetření triviálních případů úsek rozdělí volbou $k = \lfloor n/2 \rfloor$ a buďto připojí napětí k první polovině (v případě, že úsek nebyl pod napětím) nebo jej od druhé poloviny odpojí (byl-li již pod napětím), čímž se minimálním počtem přepnutí dostaneme do kýženého testovacího stavu. Dále testujeme stav jednotlivých pravých konců a prohazujeme jejich čísla v poli r tak, aby na konci r_1 až r_l byly ty konce, na nichž jsme napětí naměřili, a r_{l+1} až r_m

ty zbylé. Poté voláme funkci `test` rekurzivně. Hloubka rekurze může být nejvýše $O(\log N)$, v každé úrovni provádíme $O(N)$ operací, takže celková časová složitost je přesně dle očekávání.

Pro interakci s příslušným služebným čertem obsluhujícím tester jsou použity funkce `On`, `Off` a `Measure` snad dostatečně výmluvných názvů.

```
#include <stdio.h>
#include <stdlib.h>

int *R;                                 /* Čísla pravých konců */

void test (int la, int lb,              /* Úsek levých konců */
    int ra, int rb,                     /* Úsek pravých konců */
    int on, int flag)                   /* Připojeno (teď resp. alespoň jednou) */
{
    int k;                              /* Dělící bod */
    int a, b, c;                        /* Pomocné */

    if (la > lb || ra > rb)             /* S jistotou nepřipojeno? */
        return;
    if (la == lb && ra == rb)           /* Triviální případ? */
        {
            if (!flag)
                {
                    On (la);
                    if (!Measure (ra))      /* Přerušeno */
                        return;
                }
            printf ("%d -> %d\n", la, R[ra]);
            return;
        }

    k = (la + lb) / 2;                  /* Testovací situace */
    if (on)
        for (a=k+1; a<=lb; a++)
            Off (a);
    else
        for (a=la; a<=k; a++)
            On (a);

    b = rb+1;                           /* Konec úseku pravých konců bez napětí */
    a = ra;                             /* Testujeme ("rozděl") */
    while (a<b)
        if (Measure (R[a]))             /* Zapojen – nechat tam, kde je */
            a++;
```

```
        else
            {
                b−−;
                c = R[a];                    /* Nezapojen – vyměnit */
                R[a] = R[b];
                R[b] = c;
            }
    test (la, k, ra, b−1, 1, 1);             /* A panuj... */
    test (k+1, lb, b, rb, 0, flag);
}
int main (void)
{
int N, i ;
    printf ("Počet drátů?␣");
    scanf ("%d", &N);
    R = malloc (sizeof (int) * N), R−−;
    for (i=1; i<=N; i++)
        R[i] = i;
    test (1, N, 1, N, 0, 0);
}
```

8-5-3 Pomsta šíleného zahradníka *Vít Novák*

Ve vzorovém řešení použijeme následující myšlenku: jsou-li isomorfní stromy a a b a dále jsou isomorfní b a c, pak jsou i a a c isomorfní *(tranzitivita isomorfismu)*. Pokud tedy umíme pro každý strom najít jakýsi standardní strom k němu isomorfní, pak stačí zjistit, zda jsou tyto normalizované stromy stejné. Tranzitivitu dokážeme indukcí podle počtu listů:

1. Pro $a = b = c = \emptyset$ není o čem mluvit.
2. Nechť tvrzení platí pro stromy s nejvýše n listy, nechť $a,\ b,\ c$ mají $n+1$ listů (isomorfní stromy zřejmě mají stejně listů). Nechť $a = (a_1, a_2),\ b = (b_1, b_2),\ c = (c_1, c_2)$, nechť BÚNO (Bez Újmy Na Obecnosti – další případy obdobně) $a_1 \simeq b_1$, $b_1 \simeq c_1$ a $a_2 \simeq b_2$, $b_2 \simeq c_2$. Pak podle předpokladu $a_1 \simeq c_1$ a $a_2 \simeq c_2$, a tedy podle definice $a \simeq c$.

Nyní tedy stačí najít rychlý algoritmus, který ke každému stromu najde jeho normalizaci.

Normalizovaný strom bude strom, pro který platí, že je to buď $\emptyset$, nebo je to strom (a, b), kde $a \geq b$ a a a b jsou normalizované stromy. Přitom relace $\geq$ je libovolné lineární uspořádání normalizovaných stromů (tedy takové, že pro pro každé normalizované stromy a a b platí právě jedna z možností $a < b$, $a > b$,

$a = b$). Pro náš případ bude stačit třeba porovnání řetězcových zápisů stromů (pro normalizované stromy je jednoznačné a jak se ukáže, bude dostatečně rychlé).

Paměťová složitost tak zůstává lineární k délce zápisu stromu (označme ji N). Jaká je časová složitost? Při průchodech porovnáváme každý uzel maximálně N-krát, stejně tak jej maximálně N-krát přesuneme. Přitom průměrný počet přístupu k uzlu bude $\log N$. Uzlů je $O(N)$, proto časová složitost je průměrně $O(N \cdot \log N)$, maximálně $O(N^2)$. Uvědomme si, že závěrečné porovnání stromů je lineární.

```
#include <stdio.h>

typedef char Ttree[100];

char cmptree (char* t1, char* t2) {      /* Porovná stromy, vrací -1 pro t1 < t2, 0 pro
                                            t1 = t2, 1 pro t1 > t2 */
    char lev;
    lev=0;
    do {
      if (*t1 < *t2) return -1; else
      if (*t1 > *t2) return 1; else
      if (*t1 == '(') lev++;
      if (*t1 == ')') lev--;
      t1++; t2++;
    } while (lev != 0);
    return 0;
}

void copytree (char* t1, char* t2) {     /* Zkopíruje strom t2 na pozici t1 */
    char lev;
    lev=0;
    do {
      if (*t2 == '(') lev++;
      if (*t2 == ')') lev--;
      *t1=*t2;
      t1++; t2++;
    } while (lev != 0);
}

Ttree tmptree;

char* normalize (char* t) {              /* Normalizuje strom t a po skončení ukazuje za
                                            jeho konec */
    char *t1, *t2;
    if (*t=='0') return t+1;
```

```
    t1=t+1;
    t2=normalize (t1)+1;
    t=normalize (t2)+1;
    if (cmptree (t1, t2)==-1) {          /* je-li t1 < t2, prohodíme je */
      copytree (tmptree, t1);
      copytree (t1, t2);
      copytree (t-t2+t1, tmptree);
      * (t-t2+t1-1)=',';
    }
    return t;
}
int main (void) {
    Ttree t1, t2;
    gets (t1); gets (t2);
    normalize (t1);
    normalize (t2);
    if (cmptree (t1, t2)) printf ("Nejsou isomorfní.\n");
      else printf ("Jsou isomorfní.\n");
    return 0;
}
```

8-5-4 Čekej, Snad Dojede

Jan Kotas

Analogie železniční sítě s grafem, ve kterém stanicím odpovídají vrcholy a jízdám vlaku mezi stanicemi hrany, je nabíledni. Následující řešení je založené na mírně modifikovaném Dijkstrově algoritmu.

Popis algoritmu:

Ohodnocení každého vrcholu určuje čas, ve kterém se umíme do tohoto vrcholu dostat. Ohodnocení může být buď trvalé nebo dočasné.

Na začátku dočasně ohodnotíme vrchol, ze kterého vyjíždíme, časem, kdy v něm jsme. Ostatní vrcholy jsou zatím bez ohodnocení, resp. ohodnoceny na dočasné nekonečno. Dále vždy vybereme z dočasně ohodnocených vrholů ten s nejmenším ohodnocením (v prvním kroku to bude ten, ze kterého vyjíždíme), ohodnotíme ho tou samou hodnotou trvale, a u všech vrcholů, do kterých se můžeme přímo dostat z tohoto vrcholu, zkontrolujeme, zda se tam nemůžeme dostat dříve – v tomto případě dočasné ohodnocení patřičně upravíme.

Jakmile trvale ohodnotíme vrchol, do kterého se máme dostat, končíme, a zrekonstruujeme cestu, po které jsme se do něj dostali.

Bližší popis Dijkstrova algoritmu je možné najít téměř v každé knize, která se zabývá algoritmy (např. v [1]). To, co činí tuto úlohu zajímavou, jsou datové struktury:

Jména stanic nám na vstup přicházejí jako řetězce. Abychom mohli se stanicemi v programu rozumně zacházet, potřebujeme datovou strukturu, která by nám převáděla jména stanic na souvislý interval přirozených čísel a naopak. Jinými slovy: potřebujeme černou skříňku, která by uměla operace:

`int Insert(char *s);` ..... přidá řetězec s a vrátí jeho index, jestliže řetězec již existuje, vrátí -1.
`int Find(char *s);` ..... najde řetězec s a vrátí jeho index, jestliže řetězec neexistuje, vrátí -1.
`char *Fetch(int i);` ..... vrátí řetězec s indexem i.

Pro operaci `Fetch` je asi nejlepší udržovat normální pole s odkazy na řetězce. Problém je s operacemi `Insert` a `Find`. Někteří řešitelé si jména udržovali v seznamu, který pak prohledávali. Vyskytla se řešení, která používala vyhledávací stromy – to je dobré řešení odpovídající standardu semináře. V praxi by se s největší pravděpodobností použilo hashování. Průměrné časové složitosti jednotlivých operací (n je počet stanic, l délka jména stanice):

	Insert	*Find*	*Fetch*
Seznam	$O(1)$	$O(n \cdot l)$	$O(1)$
Setříděný seznam	$O(n + l \log n)$	$O(l \log n)$	$O(1)$
Bin. vyhl. strom	$O(\log n)$	$O(\log n)$	$O(1)$
Hashování	$O(l)$	$O(l)$	$O(1)$

Je dobré si uvědomit, že úloha má vlastně dvě části: načtení jízdního řádu a vlastní vyhledávání. Měli bychom optimalizovat rychlost vyhledávání optimálního spojení, které se bude asi provádět mnohokrát, proti rychlosti načtení jízdního řádu, které probíhá pouze jednou při startu programu.

Dále potřebujeme nějakou datovou strukturu pro uložení jízdního řádu. Jízdní řád není rozumné ukládat do normální matice $n \times n$, kde n je počet stanic, jak je tomu v grafových úlohách zvykem. V jednom políčku takové matice může být více záznamů, protože mezi dvěma stanicemi může jezdit více vlaků v různých časech. Navíc by, vzhledem k realitě, byla tato matice řídká (obsahovala by mnoho prázdných prvků).

Od datové struktury pro uložení jízdního řádu potřebujeme pouze, aby nám k dané stanici vrátila všechny vlaky, které z ní vyjíždějí. To se dá realizovat jako pole ukazatelů na pole následujících struktur:

```
struct ScheduleItem {
  Train train;             // číslo vlaku
  Time departure, arrive;  // čas odjezdu, čas příjezdu do další stanice
  Station next; }          // číslo další stanice
```

Pro každou stanici nás pak zajímá seznam těchto struktur ukončený fiktivním vlakem s číslem nula.

Pro samotné vyhledávání potřebujeme datovou strukturu, která by nám umožnila rychle vybrat vrchol s minimálním ohodnocením z dočasně ohodnocených vrcholů. Můžeme použít pole, které budeme při hledání minima celé

prohledávat. Lepší řešení je použít vyhledávací strom, v němž budeme mít minimum vždy v nejlevějším listu.

Nejlepší je použít datovou strukturu na daný problém specializovanou – haldu:

`void Insert(void *a);` ..... vloží do haldy prvek a
`void *DeleteMin();` ..... vrátí prvek s minimální hodnotou a z haldy ho vyřadí, vrátí *nil*, když je halda prázdná.
`void Decrease(void *a);` ... znovu zatřídí do haldy prvek a po snížení jeho ohodnocení.

Nejznámnější je asi jednostromová binární halda, kterou používá algoritmus HeapSort. Tuto haldu ale nemůžeme použít, protože neumožňuje provádět operaci Decrease v rozumném čase. Řešením je použít Fibonacciho haldu, jejíž popis publikovali v roce 1984 pánové Fredman a Tarjan.
Průměrné časové složitosti jednotlivých operací:

	Insert	*DeleteMin*	*Decrease*
Pole	$O(1)$	$O(n)$	$O(1)$
Bin. vyhl. strom	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacciho halda	$O(1)$	$O(\log n)$	$O(1)$

Odhadneme-li počty provádění jednotlivých operací na datových strukturách, dostaneme při použití chytrých datových struktur lineární časovou složitost vzhledem k "velikosti" jízdního řádu (za předpokladu, že jízdní řád je velký alespoň $n \cdot \log n$, kde n je počet stanic). Řádově stejnou, ale ve skutečnosti konstanta-krát větší časovou složitost dostaneme i když budeme místo chytrých datových struktur používat pouze stromy.

Jelikož datové struktury jsou zapotřebí docela často, programují je lidé jako samostatné moduly. Po světě běhá mnoho více či méně povedených knihoven pro práci s datovými strukturami. Nejznámější z nich je asi Standard Template Library (STL), kterou naprogramovali pánové Alexander Stepanov a Meng Lee z Hewlett-Packard Laboratories v roce 1994.

Literatura:

[1] Töpfer Pavel: Algoritmy a programovací techniky, Prometheus Praha, 1995

[2] Kurt Mehlhorn: Data Structures and Algorithms

8-5-5 Automata finita *Michal Koucký*

Jelikož se jednalo o poslední sérii, došlých řešení bylo vskutku pomálu. Asi se na tom podepsaly probíhající maturity. Většina došlých řešení byla za to správně.

Jediné drobné chyby, kterých jste se hojně dopouštěli v první části, tj. v automatu pro nevnořený komentář, bylo, že po načtení "pravděpodobně ukončo-

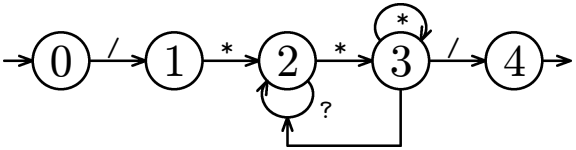
vací hvězdičky" jste se nesprávně vraceli do předchozího stavu pokud přišla další hvězdička, místo abyste zůstali v tomto stavu.

Jinak byl automat extrémně jednoduchý a opět k němu není potřeba nic dodávat.

$$\begin{aligned} KA &= (\Sigma, Q, \delta, D_0, F) \\ \Sigma &= \text{ASCII} \\ Q &= \{D_0, D_1, D_2, D_3, D_4\} \\ F &= \{D_4\} \end{aligned}$$

$$\begin{aligned} \delta(D_0, /) &= D_1 & \delta(D_3, /) &= D_4 \\ \delta(D_1, *) &= D_2 & \delta(D_3, *) &= D_3 \\ \delta(D_2, *) &= D_3 & \delta(D_3, ?) &= D_2 \\ \delta(D_2, ?) &= D_3 \end{aligned}$$

Přičemž '?' jest libovolný jinak neošetřený znak.



Konečný automat pro vnořené komentáře, jak vetšina z řešitelů správně zjistila, nemůže samozřejmě existovat. Dá se to opět ukázat způsobem, jakým jsme ukazovali, že jazyk slov 0^i1^i není přijímán žádným konečným automatem.

Předpokládejme, že bychom měli automat pro vnořené komentáře. Ten má nějakých n stavů. Podívejme se, kam se automat dostane po přečtení následujících slov:

```
/*
/*/*
 .........
/*/*/*/*.../*        (n + 1-krát)
```

Nutně některá dvě slova z výše uvedených ho převedou do stejného stavu (stavů je n, slov je $n+1$). Tato dvě slova mají délku řekněme $2a$ a $2b$. A teď pozor, přichází zlatý hřeb programu!

Slova $(/*)^a(*/)^a$ a $(/*)^b(*/)^a$ převedou tedy automat nutně též do stejného stavu, tedy buď obě akceptuje, nebo obě odmítne. V každém případě se pro jedno z nich zachová špatně, neboť jedno je vnořený komentář a druhé není.

Náš automat tedy pracuje špatně a není to tedy automat pro požadovaný jazyk. Žádný takový automat tudíž neexistuje.

Pořadí nejlepších řešitelů

Pořadí	*Jméno*	*Škola*	*Roč.*	*Bodů*
max.				334
1.	Daniel Kráľ	Gym. Zlín	4	281
2.	Jan Kára	Gym. U Libeňského zámku, Praha	2	270
3.	Tomáš Tichý	Gym. Dašická, Pardubice	4	243
4.	Aleš Přívětivý	Gym. Dašická, Pardubice	3	233
5.	Vlastimil Janda	Gym. Humpolec	2	219
6.	Martin Klíma	Gym. Havlíčkův Brod	2	211
7.	Martin Dráb	Gym. U Libeňského zámku, Praha	3	184
8.	Petr Plavjaník	Gym. Třebíč	1	175
9.	Stanislav Mikeš	Gym. Jírovcova, Č. Budějovice	4	149
10.	Jan Hubička	Gym. Jírovcova, Č. Budějovice	4	139
11. – 12.	Jakub Ouhrabka	Gym. Jeronýmova, Liberec	2	127
	Pavel Jelínek	Gym. Mikulášské nám., Plzeň	4	127
13.	Jan Kratochvíl	Gym. U Libeňského zámku, Praha	3	119
14.	Roman Ženka	Gym. Jírovcova, Č. Budějovice	4	116
15.	Tomáš Horáček	SPŠ Resslova, Praha	2	115
16.	Jan Tožička	Gym. Jateční, Ústí nad Labem	3	110
17.	Peter Kehl	Gym. Prešov	4	106
18.	Ondřej Nečas	Gym. Seifertova, Blansko	4	105
19.	Kamil Staufčík	Gym. Jeseník	4	103
20.	Věroslav Kaplan	Gym. Tř. kpt. Jaroše, Brno	3	99
21.	Martin Boldyš	Gym. Ostrov	4	96
22. – 23.	Petr Kadeřábek	Masarykovo Gym., Plzeň	3	95
	Václav Habáň	Slovanské Gym., Olomouc	4	95
24.	Jan Vodička	Gym. Zborovská, Praha	4	94
25.	Pavel Šanda	Gym. Klatovy	1	92
26.	Jindřich Makovička	Gym. Telč	3	90
27.	Tomáš Kozel	Gym. Český Těšín	4	87
28.	Jiří Vyskočil	Gym. Lanškroun	1	81
29. – 30.	Jan Štola	Gym. Zborovská, Praha	3	76
	Tomáš Šalamon	Gym. Nad Alejí, Praha	4	76
31. – 32.	Ladislav Šobr	Gym. Jeseník	2	75
	Antonín Hildebrand	Gym. Jeseník	2	75
33.	Helena Kupková	Gym. Tř. kpt. Jaroše, Brno	4	65
34.	Luboš Nový	Gym. Klatovy	3	64
35. – 38.	Jan Verner	Gym. Sladkovského nám., Praha	4	60
	Matouš Jirák	Gym. Říčany	4	60

	Tomáš Holubec	Gym. Vsetín	1	60
	Ondřej Adamovský	Gym. Jos. Jungmanna, Litoměřice	4	60
39. – 41.	Miroslav Jarošík	Gym. Klatovy	4	58
	Vlastislav Hynek	Gym. Žatec	3	58
	Pavel Šedivý	Gym. Ml. Boleslav	3	58
42.	Šárka Štěpánová	Gym. Rychnov nad Kněžnou	2	57
43.	Michal Píše	Gym. J. K. Tyla, Hradec Králové	3	56
44.	David Holec	Gym. Tř. kpt. Jaroše, Brno	1	55
45.	Roman Kozubík	Gym. Tř. kpt. Jaroše, Brno	4	52
46.	Martin Vojta	Gym. J. K. Tyla, Hradec Králové	4	51
47.	Robert Haken	Gym. Karlovy Vary	4	50
48.	Karel Volný	Gym. Bráfova, Třebíč	2	47
49.	David Laštovička	Gym. Třebíč	3	46
50. – 52.	Igor Szöke	Gym. Tř. kpt. Jaroše, Brno	2	44
	Lenka Kocmanová	Gym. Tř. kpt. Jaroše, Brno	3	44
	Miroslav Hebký	Gym. Český Těšín	4	44
53. – 54.	Vít Žďára	Gym. Polička	4	43
	Peter Vasil	Gym. Michalovce	3	43
55.	Zdeněk Šebl	Gym. Klatovy	4	41
56.	Michal Finěk	Gym. Strakonice	3	38
57.	Jiří Pik	Gym. Na Vítězné pláni, Praha	4	37
58. – 59..	Robert Špalek	Gym. Tř. kpt. Jaroše, Brno	4	32
	Pavel Zvolský	Gym. Hellichova, Praha	3	32
60.	Milan Gottvald	Gym. Dašická, Pardubice	4	31
61. – 62.	Petr Matoulek	Gym. Jírovcova, Č. Budějovice	4	29
	Tomáš Kalibera	Gym. Zborovská, Praha	4	29
63.	Jozef Mika	Gym. Žilina	4	28
64. – 66.	Jan Vítek	Gym. Slaný	2	27
	Radan Baše	Gym. Benešov	4	27
	Ondřej Kalný	Gym. Zborovská, Praha	3	27
67.	Roman Kašpar	Gym. Trutnov	3	25
68. – 69.	Michal Pták	Gym. Jičín	3	23
	Miroslav Krhounek	Gym. Kralupy nad Vltavou	2	23
70.	Jiří Semecký	Gym. Nad Alejí, Praha	4	22
71.	Denisa Denglerová	Gym. Tř. kpt. Jaroše, Brno	3	19
72. – 73.	Josef Jelínek	????	?	18
	Dušan Hlaváč	Gym. Nad Alejí, Praha	?	18
74. – 75.	Jan Antoš	Gym. Ml. Boleslav	3	16
	Bedřich Svoboda	Gym. Nad Alejí, Praha	4	16
76.	Jiří Sedláček	Gym. Teplice	3	14
77.	Michal Novotný	Gym. Teplice	4	13

Obsah

Martin Mareš a kolektiv

Korespondenční seminář v programování – VIII. ročník

Autoři a opravující úloh:
Martin Bělocký, Jan Kotas, Michal Koucký
Pavel Machek, Martin Mareš, Vít Novák

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty University Karlovy
Ke Karlovu 3, 121 16 Praha 2
jako svoji XXXX. publikaci.
Praha 1996

Vytisklo Reprografické středisko MFF UK
Malostranské nám. 25, 118 00 Praha 1

76 stran, 13 obrázků
Sazba písmem Computer Modern v programu mjTEX
Vydání první
Náklad 600 výtisků

ISBN-00-00000-00-0