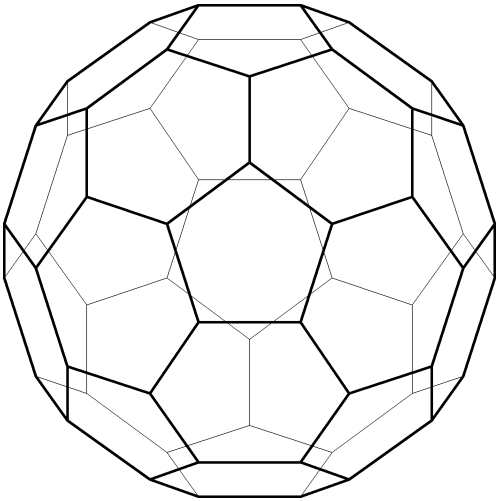
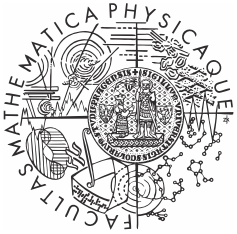
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XI. ročník – 1998/1999





Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XI. ročník – 1998/1999

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož jedenáctý 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 disciplíny 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 vůbec neznamená, že nemá 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 bratislavský). Rozhodně si však nekonkurujeme, ba 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 jak ohledně studia informatiky na naší fakultě, tak i stran 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@mff.cuni.cz*
www: *http://atrey.karlin.mff.cuni.cz/ksp/*

Zadání úloh

11-1-1 Cypherština — 10 bodů

Doktor Suess našel v troskách starověkého města Cypheru knihu, která byla psána písmeny podobnými písmenům latinské abecedy. Bohužel tato abeceda byla uspořádána jinak než abeceda latinská, tj. písmena nešla po sobě v pořadí `ABCD`... Doktora Suesse velmi mrzelo, že neví, jak je cypherská abeceda uspořádána, ale po chvíli listování knihou narazil na stránku, která byla pravděpodobně vytržena z jiné knihy, a do této pouze založena. Doktor Suess prozkoumal tuto stránku a zjistil, že je to stránka cypherského slovníku. Za předpokladu, že cypheřané třídili své slovníky podle své abecedy, pomozte doktorovi tím, že napíšete program, který pro seřazená slova ze stránky určí pořadí písmen v cypherské abecedě, případně odpoví, že to nejde. Předpokládejte, že slova na stránce slovníku jsou setříděna korektně.

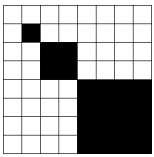
Příklad: Stránka obsahovala 4 slova v pořadí `CDA ABCD ACCD DAB`. Pak seřazená abeceda vypadá takto: `B` < `C` < `A` < `D`.

11-1-2 Quad rant — 10 bodů

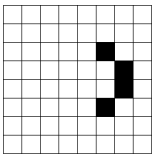
Mějme obrázek nakreslený do čtvercové sítě velikosti $2^n \times 2^n$, přičemž každý čtvereček je celý buďto černý nebo bílý. Obrázky tohoto druhu se často pro úsporu místa popisují takzvanými *kvadrantovými kódy* dle následujících pravidel:

- Kód čtverce, který je celý černý, je 0.
- Kód čtverce, který je celý bílý, je 1.
- V ostatních případech je kódem číslo $2k_1k_2k_3k_4$, kde k_i jsou kódy jednotlivých čtvrtin čtverce v pořadí levá horní, pravá horní, levá dolní, pravá dolní.

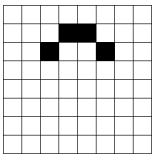
Příklad:



220001001001



20200201002001
002200012100000



22020001021000
220010020100000

Vaším úkolem je vymyslet algoritmus a napsat program, který dostane na vstupu kód obrázku a odpoví kódem obrázku vzniklého otočením zadání o 90° proti

směru hodinových ručiček (tedy například z druhého z našich příkladů by udělal třetí).

11-1-3 Magiášův Tfujtajxl — 10 bodů

Již od nepaměti žil na hoře Hejšovině černokněžník Magiáš. Ve dne kouzlil, a v noci, konec konců, také. Ale jak stárnul (přeci jen už mu na podzim bude 300 let), pomalu si uvědomoval, že ani jeho paměť už není, co bývala, a že si zrovna vůbec nemůže vzpomenout, jak zní magické zaříkadlo na hledání zapomenutých magických zaříkadel. Ale ještě si vybavoval, že určitě obsahuje slovo 'tfujtajxl' (i když možná s nějakou předponou či příponou), že počet písmen formule je dělitelný třemi, a že v něm není slovo začínající na 'q'.

Mistrovu naříkání již delší dobu tiše naslouchal jeho fámulus, sluha, kuchař, nosič, knihovník, donašeč, čistič laboratorního skla, dvorní krysař a v neposlední řadě hromosvod, to vše v jedné osobě jménem Vincek. A uvědomil si, že Magiáš přeci již dávno podlehl šílenství doby a své poznámky si přepsal do svého jeskynního počítače a že by tedy stálo za to napsat program, který zjistí, zda se mezi kouzelnými průpovídkami z Magiášových zápisků nějaký řádek splňující kouzelníkovy požadavky vyskytuje. Pokuste se sami takový program napsat, leč uvědomte si, že Magiášovo počitadlo má extrémně malou datovou paměť, a tedy čím méně proměnných použijete, tím lépe.

11-1-4 Sibylla 98 — 12 bodů

V tomto ročníku KSP bychom rádi v každé sérii uvedli jednu úlohu ražení spíše teoretického. Na zahřátí začneme něčím poměrně jednoduchým, ač poněkud netradičním: *nedeterministickým programováním.*

Programovací jazyk *Sibylla 98* se chová jako obyčejné C nebo Pascal, až na dva podstatné rozdíly:

1. Výsledkem programu je vždy buďto nula nebo jednička. Program tedy může odpovídat pouze na binární problémy typu „je toto číslo prvočíslem?“.
2. Má navíc jednu zajímavou funkci zvanou *oracle(n),* která poskytuje odpověď věštírny – celé číslo v rozsahu 0 až n. Tato odpověď je nedeterministická (to jest nevíte nic o tom, jak závisí na stavu výpočtu).

Pro každou posloupnost odpovědí věštírny tedy výpočet může probíhat úplně jinak (těmto variantám říkáme *větve výpočtu*) a potenciálně i vydat jiný výsledek. K čemu je to tedy dobré? Inu, nadefinujeme, že program na danou otázku odpoví *1 (true)* právě tehdy, když mohla věštírna poskytnout takovou posloupnost odpovědí, která by vedla výpočet k výsledku 1. V ostatních případech je odpovědí *0 (false).*

Časová složitost nedeterministického programu je v případě odpovědi 1 čas nejkratší větve výpočtu, která odpoví 1, jinak délka nejdelší ze všech větví výpočtu. (Tedy pokud žádná větev neodpoví 1 a některá větev se zacyklí, celý výpočet nikdy neskončí.)

Příklad: Nedeterministický program

```
begin
   readln(i);
   j := 2 + oracle(i-3);
   if (i>2) and (i mod j = 0) then writeln(1)
   else writeln(0)
end.
```

rozhoduje o tom, zda je zadané číslo i číslem složeným. Dělá to tak, že si nechá od věštírny poradit nějaké číslo $j \in \langle 2, i-1 \rangle$ a odpoví 1 právě tehdy, když je j dělitelem čísla i. Podle definice je celkovým výsledkem 1, právě když mohla věštírna vygenerovat j takové, aby program odpověděl jedničkou, což je přesně tehdy, když existuje vlastní dělitel čísla i, tedy když jde o číslo složené. Toto vše se děje v konstantním čase – $O(1)$, ježto všechny větve jsou prováděny konstantně rychle.

Úloha A: Napište nedeterministický program v jazyce Sibylla 98, který řeší co nejrychleji známý *problém batohu:* Máte batoh o nosnosti k kilogramů a předměty hmotností $m_1, \ldots, m_n$ kilogramů a otázka zní, zda je možno vybrat z těchto předmětů takové, abyste batoh přesně zaplnili.

Úloha B: Zkuste dokázat, že každá úloha, která je řešitelná nedeterministickým programem, je řešitelná i programem deterministickým, i když možná za cenu výrazného zpomalení výpočtu.

11-2-1 Kva-kva-kvadrant — 11 bodů

Již z minulé série všichni znáte princip kvadrantových kódů obrázků. Pro ty s pamětí zvící řešeta připomínáme: Obrázky nakreslené do čtvercové sítě velikosti $2^n \times 2^n$ se kódují kvadrantovými kódy takto:

- Kód čtverce, který je celý černý, je 0.
- Kód čtverce, který je celý bílý, je 1.
- V ostatních případech je kódem číslo $2k_1k_2k_3k_4$, kde k_i jsou kódy jednotlivých čtvrtin čtverce v pořadí levá horní, pravá horní, levá dolní, pravá dolní.

Vaším úkolem tentokráte je napsat program, který si přečte posloupnost transformací (potenciálně velmi dlouhý řetězec znaků, z nichž každý odpovídá jedné transformaci), a poté bude ze vstupu číst jednotlivé zakódované

obrázky Z_i a na každý z nich odpoví kódem obrázku vzniklého postupným aplikováním všech zadaných transformací v zadaném pořadí na Z_i.
V řetězci se mohou vyskytovat tyto transformace:

- `+` – otočení kolem středu o 90° proti směru hodinových ručiček.
- `-` – otočení kolem středu o 90° po směru hodinových ručiček.
- `X` – zrcadlení podle svislé osy.
- `Y` – zrcadlení podle vodorovné osy.
- `I` – inverze (prohození černé a bílé barvy).

Příklad: Posloupnost transformací `-XI+YI` převede obrázek s kódem `220001200 10201011` na tentýž obrázek.

11-2-2 Zákeřná posloupnost — 9 bodů

Na vstupu dostanete dlouhatánskou posloupnost přirozených čísel, ve které je každé číslo dvakrát ... až na jedno darebné, jež je tam pouze jednou. A jak přijít na to, které to je?

11-2-3 Případ scottského skota — 10 bodů

Sir Walter Scott byl archeolog amatér. Původním povoláním skotačící pasáček šlechtěného skotu (pochopitelně narozený v Oxfordu), nyní ovšem skotský šlechtic (na tom má zásluhu zejména to, že při svém amatérském archeologování vykopal něco jam a v nich učinil pár epochálních objevů – mimo jiné našel zlatou minci z doby železné, několik trilobajtů a blíže neurčené množství zakopaných psů). A sir Scott je hrozně bohatý. A ještě víc lakomý.

Jednoho dne náš skot Scott objevil na dně jedné bezedné jámy plánek dosud neobjeveného bludiště, a tak začal okamžitě plánovat, jak bludištěm projít až k pokladu, který v něm jistě bude skryt (že v bludištích bývají poklady, to ví přeci každý). Tedy ne, že by se začal shánět po tom, kde vlastně bludiště je – on si totiž ponejprv všiml, že některé chodby jsou přehrazeny dveřmi, od kterých mu jistě nikdo nedá klíč, a proto se s lakotností sobě vlastní jal vymýšlet, jak bludištěm projít, dveře „odemykat“ krumpáčem a přitom si krumpáč co nejméně otlouci, aby si nemusel ještě léta kupovat nový.
Úloha: Na vstupu je plánek bludiště (čtvercová matice velikosti $N \times N$, v každém políčku je buďto 0 (místnost), 1 (zeď), 2 (prokopnutelné dveře), 3 (poklad, vyskytuje se pouze jednou) nebo 4 (místnost, kde začínáte; rovněž unikátní). Vaším cílem je najít cestu (to jest posloupnost políček sousedících hranou [nikoliv pouze rohem]) začínající na políčku typu 4, končící políčkem typu 3, neobsahující políčko typu 1 a k tomu ještě takovou, která prochází nejmenším počtem políček typu 2. Pokud je takových cest více, vyberte libovolnou z nejkratších.

11-2-4 Alea iacta est — 10 bodů

V rovině je dána souřadná soustava (tak, jak už to bývá – dvě na sebe kolmé souřadné osy, ta první svislá, ta druhá vodorovná, na obou stejně velké dílky) a v souřadné soustavě n bodů (určených svými souřadnicemi) a výchozí místo M. Vaším úkolem je vymyslet program pro rozmisťovacího robota takový, aby začal v bodě M (počáteční směr pohybu si můžete zvolit), do každého z daných n bodů umístil kostičku (a nikam jinam!) a skončil opět v bodě M. Krásně jednoduché, že?

Robota ovšem zkonstruovala nejmenovaná firma nejmenovaného počítačového magnáta proslulá jednak nejmenovaným „operačním systémem“ a jednak tím, že se snaží ovládání všeho zjednodušovat. Robot má díky tomu velice jednoduchý programovací jazyk sestávající z příkazů $G(a)$ (jdi dopředu o a jednotek), L (umísti kostičku a otoč se o 90° doleva), R (umísti kostičku a otoč se o 90° doprava). Takže zase tak jednoduché to přeci jen nebude, že?

Úloha: Napište program, který dostane na vstupu souřadnice všech n požadovaných kostiček a počátečního (či koncového, jak chcete) bodu M a jeho výstupem bude program pro umisťovacího robota takový, aby robot splnil svůj úkol, případně zpráva o tom, že to nelze.

11-2-5 Mnoho CPŮ, složitosti smrt — 12 bodů

Minule jsme si zavedli nedeterministický programovací jazyk, tentokráte pro změnu zkusíme programovat deterministicky, zato však paralelně – představte si, že máte počítač, který umí provádět tolik programů najednou, kolik si řeknete. Takový paralelní počítač budeme programovat opět v trošku modifikovaném klasickém programovacím jazyku (řekněme Pascalu či Céčku); zato však zavedeme vedle normálních bloků (`begin` ... `end` či `{` ... `}`) ještě bloky, před něž budeme psát klíčové slovo `parallel`, což bude znamenat, že všechny příkazy takového bloku se budou provádět najednou (nesmějí ovšem zapisovat do těchže proměnných či jedna větev výpočtu do jedné proměnné zapisovat a druhá z ní číst). Dobu potřebnou k vykonání takového paralelního bloku stanovíme jednoduše jako maximum z časů provádění jednotlivých příkazů (pozor, mohou zahrnovat i volání funkcí!).

Příklad: Tento program v P-Pascalu (paralelním Pascalu) počítá najednou minimum a maximum ze zadaných čísel:

```
var x,n,min,max:integer;
begin
   read(n);
   min:=-maxint; max:=maxint;
   while n>0 do begin
      read(x);
```

```
        parallel begin
           if x<min then min:=x;
           if x>max then max:=x;
           n:=n-1;
        end;
     end;
     writeln(min, max);
  end.
```

Úlohy:

1. Napište paralelní funkci, která nalezne nejmenší prvek v daném poli v čase rychlejším než $O(N)$.
2. Dokažte, že vše, co se dá spočítat na paralelním počítači, lze spočítat i na počítači klasickém (sekvenčním), možná ovšem daleko, daleko pomaleji.
3. Dokažte, že cokoliv, co šlo naprogramovat na nedeterministickém počítači v čase omezeném nějakým polynomem, lze v polynomem omezeném čase (možná však jiným polynomem než tím předešlým) spočítat na počítači paralelním.

11-3-1 Telefouňové — 10 bodů

Nejmenovaná telefonní společnost vydala reklamní leták, v němž se chlubí tím, která všechna města telefonizovala. Pro každé z uvedených měst uvádí počet telefonních vedení vedoucích do jiných měst ze seznamu, přičemž mezi každými dvěma městy může vést pouze jedno vedení a toto je obousměrné. Jak už to u reklamních letáků, zejména pak pocházejí-li od nejmenovaných telefonních společností, bývá, vy jim nevěříte, a proto jste se rozhodli, že vytvoříte program, jenž si seznam přečte a rozhodne, zda existuje telefonní síť daných vlastností a pokud ano, tak nějakou takovou nalezne (nalezením telefonní sítě se myslí vypsání počátečního a koncového města každého vedení).

Příklad 1: Zadání: Praha 3, Brno 2, Horní Kotěhůrky 1, Oulehlice 2. Jeden z možných výsledků: Praha – H. K., Praha – Brno, Brno – Oulehlice, Praha – Oulehlice.

Příklad 2: Zadání: Praha 3, Metelice 1. Výsledek: podvod.

Předpokládejte, že města jsou očíslována od 1 do n a že vstupem je posloupnost n čísel, přičemž i-té číslo udává počet vedení pro i-té město.

11-3-2 Ffffaktoriál — 9 bodů

Je všeobecně známo, že $n!$ (faktoriál čísla n, definován jako $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$) bývá ohromné číslo končící na mnoho nul. O to překvapivější je ovšem

fakt, že je možné snadno zjistit, kolik je poslední nenulová číslice v desítkovém zápisu $n!$. Napište program, který tuto číslici pro dané n spočte.

Příklad: $6! = 720$, poslední nenulová číslice je 2.

11-3-3 Obdélníčky — 11 bodů

Jeník Příhoda, syn známého Strýčka Příhody, hrál neobvyklou hru: Měl obdélníkový stůl velikosti $a \times b$ a n papírových obdélníčků velikostí $a_i \times b_i$ a chtěl obdélníčky po stole rozložit tak, aby jejich hrany byly rovnoběžné s hranami stolu a celková plocha zakrytá obdélníčky byla co největší. Obdélníčky se mohly i překrývat, ale nesměly přečnívat přes okraj stolu.

Po krátkém hraní této obtížné hry Jeníka napadlo, že by si mohl napsat program, který pro dané polohy a velikosti obdélníčků spočte, jaká je celková plocha jejich sjednocení. Zkuste to i vy.

11-3-4 Machina Universalis — 12 bodů

V tomto pokračování našeho seriálu o tajích teoretické informatiky zkusíme dokázat, že existuje tzv. universální program, to jest program, kterému dodáte na vstupu nějaký program P a vstup pro tento program a on vám odpoví výstupem programu P a nebo nikdy neskončí, pokud program P nikdy neskončí.

Bylo by ovšem velice pracné psát takový program přímo pro Pascal resp. Céčko, a tak si nejdříve nadefinujeme velice jednoduchý assembler a dokážeme, že se v něm dá naprogramovat totéž, co v Pascalu a Céčku.

Mikroassembler μ_1 počítá v paměti, která je indexována přirozenými čísly a každá její buňka obsahuje opět nějaké přirozené číslo. K dispozici jsou čtyři instrukce (Jedna z instrukcí `DEC` a `CLR` je „přebytečná“ a dala by se nahradit, ale je to zbytečná komplikace; blíže viz úloha 9-3-3):

- `INC` x – zvýšení obsahu paměťové buňky číslo x o 1.
- `DEC` x – snížení obsahu paměťové buňky číslo x o 1. Pokud tato již byla nulová, nezmění se.
- `CLR` x – uložení nuly do dané paměťové buňky.
- `JEQ` x,y,z – pokud je v paměťových buňkách x a y uloženo totéž číslo, skočí o z instrukcí dopředu (z je libovolné celé číslo), v opačném případě neprovede nic.

Program začíná běh svojí první instrukcí a mimo pevně zvolené paměťové buňky obsahující vstup je obsah zbytku paměti nedefinován. Končí pak při pokusu o skok *před* svou první instrukci, přičemž v pevně zvolených (možno jiných než vstupních) paměťových buňkách je uložen výsledek výpočtu. Na ukázku uveďme program, který sečte čísla na adresách 1 a 2, načež výsledek uloží na adresu 0.

```
CLR    3
CLR    0
JEQ    1,3,4
DEC    1
INC    0
JEQ    3,3,-3
JEQ    2,3,-7
DEC    2
INC    0
JEQ    3,3,-3
```

A nyní slíbený důkaz ekvivalence μ_1 a Pascalu (pro Céčko analogicky). Kroky označené [*] uvádíme jen v náznacích, vaším úkolem je popsat je přesně. V průběhu našeho důkazu budeme postupně omezovat, jaké jazykové konstrukce používáme, a to tak, že na konci zbudou přesně obraty odpovídající našemu miniassembleru. Za to pochopitelně zaplatíme prudkým nárůstem velikosti programu a drastickým zpomalením, ale to je jedno, protože naším cílem je pouze dokázat, že se v obojím *dá spočítat totéž.*

1. Pro naše úvahy se omezíme na Pascal, v němž typ *word* pojme libovolné přirozené číslo bez omezení rozsahu. To nám důkaz značně zjednoduší, nicméně sílu jazyka taková úprava nikterak nemění, protože můžeme implementovat dynamicky alokovanou aritmetiku počítající s libovolně dlouhými čísly.
2. [*] Každý pascalský datový typ je možno reprezentovat jedním wordem (to platí nejen pro základní typy, ale i pro typy strukturované, jako je třeba pole [indexované čímkoliv, tedy vlastně wordem a obsahující libovolné prvky, tedy vlastně wordy, čímž se řeší i případ recordů]). Tím jsme si tedy převedli program na jiný program, který počítá pouze s wordy.
3. Speciálně také vstup a výstup programu je možno prohlásit za word.
4. V programu je možno eliminovat volání procedur za cenu zavedení vlastního zásobníku, v němž budou uloženy lokální proměnné procedur a návratové adresy (vše pochopitelně wordy) a objevení se velkého množství skoků.
5. Všechny výrazy můžeme zavedením vhodných pomocných proměnných rozložit na elementární přiřazení typu `x := y @ z`, (`@` je operátor).
6. [*] Všechny cykly je možné rozložit na podmíněné skoky, tedy konstrukce typu `if x then goto y`, kde `x` je proměnná (výrazů jsme se přeci v předchozím kroku zbavili) a `y` návěští.

7. Násobení, dělení, bitové operace a jiné vymoženosti můžeme nahradit pouhým sčítáním, odčítáním, testem na rovnost a nerovnost typu „je menší než“. (To je jednoduché, ale poměrně pracné, jelikož je nutno uvažovat každou operaci zvlášť.)
8. Zbylým proměnným přiřadíme pevná místa v paměti μ_1.
9. [*] Sčítání, odčítání a skoky podmíněné rovností a nerovností je možno převést na instrukce μ_1.

Již tedy víme, že libovolný program v Pascalu můžeme přeložit na ekvivalentní program v μ_1 (náš důkaz dokonce dává algoritmus, jak to provést). Nyní zkuste na základě tohoto tvrzení sami dokázat, že existuje universální program v našem mikroassembleru (pomněte, že programy a jejich vstupy i výstupy je také možné reprezentovat přirozenými čísly), jakož i universální program v Pascalu.

11-3-5 Pascal či Céčko? — 10 bodů

Napište program, který jde zkompilovat jak kompilátorem Pascalu, tak kompilátorem Céčka a po spuštění vypíše, čím byl vlastně zkompilován. Další rozpoznávané jazyky budou hodnoceny prémiovými body!

11-4-1 Výlet po řece — 12 bodů

Bill Gates po své prezentaci Windows 98 zchudnul. Poté, co exekutoři opustili jeho vilu, zjistil, že mu v pěněžence zbylo N dolarů. Uvědomil si, že za ty peníze může žít maximálně pár dní a pak bude muset žebrat. Protože je Bill neobyčejně hrdý, rozhodl se, že peníze utratí co nejpříjemněji, načež spáchá sebevraždu. A protože Bill už od dětství miluje řeky a vodu a má lítačku na parníky, rozhodl se, že pojede parníkem po řece Mississippi. Řeku Mississippi si můžeme představit jako strom (graf) – místa, kde se stékají dvě řeky jsou uzly (vrcholy grafu) a kořen tohoto grafu je v místě, kde se Mississippi vlévá do moře. Po každé hraně (část řeky mezi dvěmi sousedními soutoky) jedou vždy jednou za den v 10h ráno dva parníky, jeden dolů po řece a druhý nahoru.

Billova cesta po řece bude vypadat, tak, že nasedne u jednoho soutoku na parník, popojede k sousednímu, a protože ten den už mu nikam žádný parník nejede, přespí v tom městě v hotelu a druhý den popojede jiným parníkem (tj. nebude se přece vracet zpátky) na sousední soutok řek. Protože Billa parníková doprava nic nestojí, jediné jeho náklady budou na přespání v hotelech, jejichž ceny se mohou v různých městech lišit. Bill zná ceny hotelů v jednotlivých městech, a protože chce utratit všechny své peníze, zajímá ho, jestli existuje taková cesta po řece, u které součet cen za hotely v uzlech, kterými prochází, je roven přesně N. Jinými slovy, Bill po vás chce, abyste mu napsali program, který pro vrcholově ohodnocený strom zjistí, zda v něm existuje cesta mezi

dvěma vrcholy, u které suma hodnot vrcholů, jimiž prochází, je rovna danému číslu N.

11-4-2 Komplot — 10 bodů

Bylo–nebylo. V jednom z mnoha paralelních vesmírů, v jedné vzdálené galaxii za 10^9-atero supernovami, 10^9-atero černými děrami a 10^9-atero hnědými trpaslíky byla–nebyla jedna malá žlutá hvězda, kolem níž ve vzdálenosti asi tak 150 milionů kilometrů obíhala malá bezvýznamná modrobílá planeta. Na jednom z jejích kontinentů stála malá vesnice, v ní malý domeček, v něm malý stoleček, na stolečku mapa, na mapě jeden z mnoha paralelních vesmírů, v jedné ... ehm, takhle bychom se asi moc daleko nedostali...

Tak jinak: Kolem domečku stál plot, který ohraničoval okolní pozemky. A jelikož katastrální úředníci neměli toho času do čeho píchnout, byly hranice pozemku opravdu křivolaké – parcelu nezbývalo než považovat za obecný (tedy ne nutně konvexní) mnohoúhelník a plot za obecnou uzavřenou lomenou čáru, která sama sebe nikde neprotíná.

Napište program, který na vstupu dostane souřadnice jednotlivých zlomů plotu = vrcholů mnohoúhelníka (pozemek je malý, takže pro jednoduchost můžeme předpokládat, že země je plochá) a spočte obsah vytýčené plochy. Souřadnice budou zadány v pořadí obcházení plotu proti směru hodinových ručiček při pohledu shora.

11-4-3 V jámě — 10 bodů

O Praze se říká, že je to nejrozkopanější město na této planetě a názvy některých ulic (Jamská, Na Příkopě, V Jámě, ...) se snaží vypovědět, že tomu tak jest už od nepaměti.

Pan N. O'Body, zavilý pragocentrista se rozhodl, že svým přátelům předvede, že tomu tak není. Vytipoval si jednu pražskou ulici a během jedné ze svých dlouhých nedělních procházek si sestavil katalog všech rozkopanin v ní a pokoušel se najít co nejdelší úsek bez děr, kterým by své známé mohl provést, aniž by narazili na příkop, výkop, zákop či jinou silniční singularitu.

Zkuste nalézt algoritmus na řešení tohoto problému: Máte zadanou posloupnost $a_1, \ldots, a_n$ čísel [nesetříděných!] reprezentujících vzdálenosti jednotlivých děr od pevně zvoleného počátku ulice. Nalezněte v této posloupnosti dvě čísla a_i, a_j taková, že pro žádné k není $a_i < a_k < a_j$ a přitom $a_j - a_i$ je největší možné. [Nenápadná poznámka: existuje řešení v čase $O(n)$.]

11-4-4 Turingovy stroje — 10 bodů

Minule jsme si ukázali, že libovolný „rozumný“ programovací jazyk lze přepsat do jednoduchého assembleru, na který nečiní žádné problémy napsat interpreter. Tentokráte zkusíme nalézt ještě jednu možnost zápisu algoritmů,

a to jsou Turingovy stroje – „běžnému programátorskému uvažování“ jsou sice poněkud vzdálenější, ale o to více se hodí pro různé úvahy o výpočetních možnostech jednotlivých programovacích jazyků.

Definice Turingova stroje (vyskytla se v KSP již jednou v příkladu 10-2-3, takže ti, kdo řešili loňský ročník, ji klidně mohou přeskočit – slibujeme, že jsme nic nezměnili): Turingův stroj sestává z pásky a řídící jednotky. Páska Turingova stroje má jen jeden konec, a to levý (doprava je nekonečná) a je rozdělena na políčka. Na každém políčku se nachází právě jeden znak z abecedy Σ (to je nějaká konečná množina, o které navíc víme, že obsahuje znak Λ). Nad páskou se pohybuje hlava stroje, v každém okamžiku je nad právě jedním políčkem.

Řídící jednotka stroje je v každém okamžiku v jednom stavu ze stavové množiny Q (opět nějaká konečná množina) a rozhoduje se podle přechodové funkce $f(q, z)$, která pro každou kombinaci stavu q a znaku z, který je zrovna pod hlavou, dává uspořádanou trojici (q', z', m), přičemž q je stav, do kterého řídící jednotka přejde v dalším kroku, z' znak, kterým bude nahrazen znak z umístěný pod hlavou a konečně m je buďto L nebo R podle toho, zda se má hlava posunout doleva nebo doprava.

Výpočet Turingova stroje probíhá takto: na počátku je hlava nad nejlevějším políčkem, na počátku pásky jsou uložena vstupní data (zbytek pásky je vyplněn symboly Λ) a řídící jednotka je ve stavu q_0. V každém kroku výpočtu se Turingův stroj podívá, co říká funkce f o kombinaci aktuální stav + znak pod hlavou, načež znak nahradí, přejde do nového stavu a posune hlavu v udaném směru. Takto pracuje do té doby, než narazí hlavou do levého okraje pásky, čímž výpočet končí a páska obsahuje výstup stroje.

Příklad: Následující tabulka definuje Turingův stroj s abecedou $\Sigma = \{\mathbf{0}, \mathbf{1}, \Lambda\}$, který ze vstupního slova složeného ze znaků **0** a **1** odstraní všechny jedničky. Počátečním stavem je stav 0, políčka tabulky označená ‘—’ nemohou být strojem nikdy dosažena.

	0	**1**	Λ
0	$0, \mathbf{0}, R$	$0, \mathbf{1}, R$	$1, \Lambda, L$
1	$1, \mathbf{0}, L$	$2, \mathbf{0}, R$	—
2	$2, \mathbf{0}, R$	—	$3, \Lambda, L$
3	$1, \Lambda, L$	—	—

Úloha: Dokažte, že pro libovolný program v Pascalu je možno nalézt Turingův stroj, který počítá totéž (to jest zvolíme-li si nějaké kódování čísel na pásce stroje, vydá pro každý vstup stejný výstup jako původní program). Není nutné, abyste psali překládací program nebo přesně rozepisovali překlady jednotlivých operací – stačí dostatečně podrobně popsat myšlenku převodu. (Rada: Zkuste se zamyslet nad minulou úlohou, mohla by vám k tomu dopomoci.)

Vzorová řešení

11-1-1 Cypherština — Jan Kára

Řešitelé této úlohy se rozdělili vesměs na 2 skupiny. První skupina řešitelů si uvědomila, že ze získaných vztahů mezi písmeny si mohou postavit graf a problém uspořádání písmen se změní na problém topologického třídění grafu či hledání nejdelší cesty v grafu. Většina řešitelů v této skupině měla také úlohu víceméně správně. Řešitelé z druhé skupiny zpravidla získali vztahy mezi jednotlivými písmeny (nebo mezi skupinami písmen) a poté se pokoušeli nějak tyto informace spojovat, aby získali výslednou abecedu. V této skupině se úspěšných řešitelů vyskytlo skutečně pomálu. Jinak obecným nešvarem mnoha řešení byl nedostatečný popis algoritmu (z popisu algoritmu má být už jasné, jak program funguje bez nutnosti studia zdrojového textu), chybějící odhad časové a paměťové složitosti či nepřítomnost zdůvodnění správnosti algoritmu.

Teď už tedy ke správnému algoritmu. Je celkem zřejmé (z definice lexikografického uspořádání), že ze dvou sousedních slov získáme informaci maximálně o pořadí dvou písmen (pokud bude první slovo začátek druhého – třeba jako `auto`, `autobus` – nezjistíme nic). Budou to první odlišná písmena na stejné pozici (např. ze slov `list`, `limonáda` zjistíme, že `s` $<$ `m`). Tyto zjištěné nerovnosti si budeme v orientovaném grafu (písmena jsou vrcholy) representovaném maticí sousednosti zaznamenávat jako hrany z menšího písmene do většího. Po projití celého slovníku tedy budeme mít v tomto grafu všechny získatelné nerovnosti. Mimo to si také budeme pamatovat, která písmena se už v textu vyskytla, abychom mohli poznat, zda je slovník úplný. V dalším textu už vrcholy písmen, která se v textu nevyskytla nebudeme uvažovat. Také si u každého vrcholu budeme pamatovat, kolik do něj vede hran.

Teď nastává druhá fáze – samotné vytváření abecedy. Nalezneme všechny vrcholy, do kterých nevede hrana. Budou to tedy ta písmena, pro která neexistuje menší písmeno. Pokud je takové písmeno právě jedno, je zřejmě nejmenší a přidáme ho tedy jako další písmeno do abecedy (na počátku je abeceda prázdná). Pokud není takové písmeno žádné, neexistuje zjevně nejmenší písmeno a ve slovníku je chyba (Tuto alternativu jste ve svých řešeních nemuseli uvažovat, protože byla zaručena bezespornost slovníku.). Pokud je takových písmen více, nemáme možnost, jak písmena mezi sebou srovnat (nevedou mezi nimy žádné hrany), a proto neexistuje jednoznačné řešení. Pokud se nám podařilo úspěšně přidat do abecedy písmeno, odebereme odpovídající vrchol z grafu a hrany z něho vycházející. Tím se nám bez vstupních hran ocitne vrchol zastupující následující písmeno. Můžeme tedy zase zopakovat druhou fázi, čímž získáme druhé písmeno abecedy, pak

třetí, čtvrté a tak dále. Nakonec budeme mít takto seřazenu celou abecedu.

Algoritmus bude mít časovou složitost $O(N + A^2)$, kde N je velikost slovníku a A je počet písmen v abecedě. Potřebujeme totiž postupně přečíst celý slovník – $O(N)$ – pro každé písmeno výsledné abecedy najít vrchol bez vstupních hran – $O(A)$ (díky tomu, že si pamatujeme, kolik hran do vrcholu vede) – a vyřadit všechny hrany z tohoto vrcholu (těch bude maximálně A). Dohromady tedy $O(A^2)$. Paměťová složitost bude $O(A^2)$. Potřebujeme si totiž pamatovat matici sousednosti grafu. Správnost algoritmu byla zdůvodněna již v popisu.

Program je přímou implementací algoritmu, a proto není myslím třeba zvláštního popisu.

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

#define MAXABC 26                       /* Maximalni velikost abecedy */
#define MAXLEN 20                       /* Maximalni delka slova */

int Used[MAXABC];                       /* Oznaceni pouzitych pismen */
int Edges[MAXABC];                      /* Pocet vztahu znamych pro jednotliva pismena */
int Before[MAXABC][MAXABC];             /* Before[i][j] == 1 pokud i > j */
int Chars;                              /* Pocet znaku v abecede */

void InitStruct (void)                  /* Zinicializuje struktury */
{
  int i, j;

  for (i = 0; i < MAXABC; i++)
  {
    Used[i] = 0;
    Edges[i] = 0;
    for (j = 0; j < MAXABC; j++)
      Before[i][j] = 0;
  }
}

void GetRelation (char *S1, char *S2)   /* Zjisti z danych retezcu vztah dvou pismen a
                                           zaznamena ho */
{
  for (; *S1 == *S2 && *S1; S1++, S2++);    /* Najdeme prvni rozdil */

  if (!*S1 || !*S2)                     /* Vlastne jsme nic nezjistili? */
    return;
  Before[*S2 - 'a'][*S1 - 'a'] = 1;
  Edges[*S2 - 'a']++;
}

void Use (char *S)                      /* Zaznamename pouzita pismena */
{
  for (; *S; S++)
    if (!Used[*S - 'a'])
    {
      Chars++;
```

```
            Used[*S - 'a'] = 1;
        }
}
void ReadVocabulary (void)          /* Nacte slovnik a zaznamena do matice vztahy mezi
                                       pismeny */
{
    char SBuf[2][MAXLEN];           /* Jednotliva nactena slova */
    int ABuf = 0;

    puts ("Zadavejte slova:");
    gets (SBuf[ABuf]);
    Use (SBuf[ABuf]);               /* Zaznamename pouzita pismena */

    while (*SBuf[ABuf])             /* Dokud je neco nacteno... */
    {
        /* Ze soucasneho slova se stane predchozi, nacteme dalsi slovo */
        ABuf = !ABuf;
        gets (SBuf[ABuf]);
        Use (SBuf[ABuf]);           /* Zaznamename pouzita pismena */

        GetRelation (SBuf[!ABuf], SBuf[ABuf]);   /* Zjisti a zapise eventualni vztah */
    }
}
int CreateAlphabet (char *Alphabet) /* Vytvori abecedu */
{
    int Pos;                        /* Pocet znaku zaznamenanych v abecede */
    int Next = -1, Cur;             /* Dalsi pismeno k provedeni; Aktualni pismeno */
    int i;

    for (i = 0; i < MAXABC; i++)
        if (!Edges[i] && Used[i])   /* Kandidat na nejmensi pismeno? */
            if (Next == -1)         /* Nemame jeste zadneho? */
                Next = i;
            else
                return 0;

    for (Pos = 0; Pos < Chars; Pos++)
    {
        if (Next == -1)             /* Bonusove zjistime nekorektni slovnik... */
            return -1;

        Cur = Next;
        Next = -1;

        Alphabet[Pos] = Cur + 'a';
        for (i = 0; i < MAXABC; i++)
            if (Before[i][Cur])     /* Vede do vrcholu i hrana? */
            {
                Before[i][Cur] = 0; /* Odstranime ji */
                if (!--Edges[i])    /* Odstranili jsme posledni hranu? */
                    if (Next == -1) /* Jeste nemame dalsi pismeno? */
                        Next = i;
                    else            /* Nejednoznacna abeceda... */
                        return 0;
```

```
        }
    }
    Alphabet[Pos] = '\0';
    return 1;
}
int main (void)
{
    char Alphabet[MAXABC];
    int Ret;
    InitStruct ();
    ReadVocabulary ();                  /* Nacte slovnik a zjisti a zaznamena vztahy */
    Ret = CreateAlphabet (Alphabet);
    if (!Ret)
      puts ("Nemam dostatek informaci k sestaveni abecedy.");
    else if (Ret == -1)
      puts ("Nekorektni slovnik.");
    else
    {
      puts ("Abeceda je:");
      puts (Alphabet);
    }
    return 0;
}
```

11-1-2 Quad rant — Aleš Přívětivý

Zopakujme si pravidla kódování obrázku: je-li na vstupu 0 nebo 1, pak obrázek je celý černý nebo bílý, je-li tam 2, pak následují čtyři po sobě jdoucí kódy čtvrtin původního obrázku v pořadí levá horní, pravá horní, levá dolní a pravá dolní čtvrtina. Tyto čtvrtiny jsou popsány stejným kódem.

Naším úkolem je takto zadaný obrázek převést na obrázek otočený o 90° proti směru hodinových ručiček (tj. vypsat kód takto otočeného obrázku). Je zřejmé, že pokud je obrázek daný kódem 0 nebo 1 (jednolitý obrázek) je výsledkem po otočení také kód 0 nebo 1. V opačném případě je tvořen čtyřmi čtvrtinami a my je přesuneme o 90° proti směru hodinových ručiček. Tím se sice čtvrtiny dostanou na své místo, ale ouha jsou otočené o 90 stupňů po směru hodinových ručiček. Nezbývá nám než, otočit je o 90° proti směru, což provedeme zase použitím výše popsaného algoritmu.

Algoritmus je sice jednoduchý, ale lze ho implementovat více způsoby – efektivně a méně efektivně. Hlavní chybou, které se mnozí řešitelé dopouštěli, bylo, že orotovaný kód sestavovali z řetězců vrácených z rekurzivního volání funkce na jednotlivé čtvrtiny, přičemž spojování řetězců neprobíhalo v konstantním čase, ale v čase úměrným délce jednotlivých spojovaných řetězců. To

vedlo na algoritmy s časovou složitostí $O(n \cdot \log n)$ až $O(n^2)$, kde n je délka kódu k orotování.

Tomu se dalo vyhnout sestavením stromu popisujícího obrázek. Uzel stromu, který byl listem, měl hodnotu buď 0 nebo 1 (jednolitý čtvereček). Uzel, ve kterém se strom větvil, měl hodnotu 2 a obsahoval 4 ukazatele na uzly popisující jednotlivé čtvrtiny obrázku v daném pořadí. Vypsání orotovaného kódu se pak dá provést projitím tohoto stromu, kde v místě větvení probíráme větve v takovém pořadí, v jakém mají být vypsány přerovnané čtvrtiny obrázku.

Časová složitost tohoto algoritmu je lineární $O(n)$, v lineárním čase strom zkonstruujeme, v lineárním ho projdeme, protože všech uzlů je stejně jako znaků kódu, tedy n. Paměťová složitost je také lineární $O(n)$, tj. počet uzlů stromu. Lépe to už nejde, protože každý znak musíme alespoň jednou vypsat, což způsobí, že časová složitost libovolného algoritmu řešícího tuto úlohu je alespoň lineární. Správnost a konečnost algoritmu je evidentní.

```
#include <stdio.h>

typedef struct Node {
        char type;                      /* 0 = bily, 1 = cerny, 2 = subquad  */
        Node *subnode[4];               /* pokud type = 2, odkaz na subquady  */
} Node;

int trans[4] = { 1,  3,  0,  2 };       /* transformace quadu pro rotaci  */

Node * get_sub_tree ()                  /* vytvori strom popisujici quady  */
{
        Node *node = new Node;
        node->type = getchar ();
        if  (node->type=='2')
                for  (int i=0; i<4; i++) node->subnode[i] = get_sub_tree ();
        return node;
}

void put_rsub_tree (Node *node)         /* vypise transformovany kod podle stromu  */
{
        putchar (node->type);
        if  (node->type=='2')
                for  (int i=0; i<4; i++) put_rsub_tree (node->subnode[trans[i]]);
        delete node;
}

int main ()
{
        printf ("Kod:␣");
        put_rsub_tree (get_sub_tree ());
        puts ("");
        return 0;
}
```

11-1-3 Magiášův Tfujtajxl — Martin 'Mafi' Bělocký

Správných řešení této úlohy se nám sešlo pomálu – většina programů totiž nevyhledávala správně požadované slovo `TFUJTAJXL`. Když je v zadání uveden požadavek na minimální datovou paměť, neznamená to žádné omezení na délku programu. Téměř všichni řešitelé načítali Magiášův text ze souboru a potřebovali pro to proměnnou pro přístup k souboru a proměnné pro načtená písmena či slova. Ze souboru je možno načítat data, aniž bylo třeba přistupovat k souboru takto: V programu lze použít příkazy pro načítání ze standardního vstupu, tj. klávesnice a po přeložení programu program spustit příkazem `tfujtajxl.exe <vstup.txt`, který zařídí přesměrování standardního vstupu programu na soubor `vstup.txt`, v němž jsou napsané testovací věty. Tento systém přesměrování standardního vstupu (obdobně s `>` přesměrování std. výstupu) se dá použít snad na všech běžných operačních systémech (MS-DOS, Unix, ...). Nikdo také nechtěl výpisy čísel řádků, které splňovaly 3 zadaná kritéria. Stačilo pouze např. vypsat *nalezeno* a skončit program. Algoritmus, který řešitelé většinou použili, používal pole, do něhož načetli řádek, a pole s hledaným řetězcem `TFUJTAJXL`. Délka řádku však nebyla omezena, a tak načítání celého řádku nebylo možné. Asi nejčastější chybou bylo nenalezení řádků obsahujících slova jako `TFTFUJTAJXL` či `TFUJTFUJTAJXL`. Nikdy nelze hledat řetězce tak, že se testuje po jednotlivých písmenech vstup a první písmeno hledaného řetězce a poté, co se písmena shodují testovat na shodu druhé hledané písmeno a v případě neshody začínat znovu od prvního hledaného písmena.

Správným řešením úlohy je zkonstruovat tzv. vyhledávací stavový automat. To je (zjednodušeně) program, který obsahuje proměnnou *stav*, která spolu s právě načteným znakem vstupu určuje další chování programu a změnu obsahu proměnné *stav*. Proměnná *stav* může mít jednu z hodnot `0`,`T`,`F`,`U`,`J`,`T1`,`A`,`J1`,`X`, `L`,`XX`. Uvažujme teď pouze první kritérium, tj. vyhledat řádek obsahující text `TFUJTAJXL`. Musíme nadefinovat tzv. přechodovou funkci automatu, neboli nadefinovat pro každou možnou hodnotu proměnné *stav* a každý možný načtený znak, jak se má změnit proměnná *stav*.

Algoritmus automatu prohledávajícího jeden vstupní řádek pak vypadá takto:

```
K1: stav:=0;
K2: načti(znak);
K3: case stav of
     0: case znak of
          't': stav:=t;
          end of line: stav:=XX;
          else stav:=0;
     t: case znak of
```

```
          'f': stav:=f;
          't': stav:=t; { ...TTfujtajxl }
          end of line: stav:=XX;
          else stav:=0;
     f: case znak of
          'u': stav:=u;
          't': stav:=t; { ...TFTfujtajxl }
          end of line: stav:=XX;
          else stav:=0;
     u: case znak of
          'j': stav:=j;
          't': stav:=t; { ...TFUTfujtajxl }
          end of line: stav:=XX;
          else stav:=0;
     j: case znak of
          't': stav:=t;
          end of line: stav:=XX;
          else stav:=0;
    t1: case znak of
          'a': stav:=a;
          'f': stav:=f; { ...TFUJTFujtajxl }
          't': stav:=t; { ...TFUJTTfujtajxl }
          end of line: stav:=XX;
          else stav:=0;
     a: case znak of
          'j': stav:=j;
          't': stav:=t; { ...TFUJTATfujtajxl }
          end of line: stav:=XX;
          else stav:=0;
    ...
    end;
K4: goto K2;
```

Stavy `j1` a `x` jsou analogické např. stavu `a`; stav `l` je tzv. koncový stav, kdy automat např. vypíše hlášku o nalezení řetězce a přejde např. do stavu `0` nebo jeho činnost skončí. Stav `XX` je taktéž koncový stav automatu, kdy vypíšeme např. hlášku o nenalezení řetězce a skončíme.

Toto byl automat pro hledání řetězce `TFUJTAJXL` na jednom řádku. Chceme ho teď rozšířit na celý soubor řádek. To učiníme snadno přepsáním všech přechodů při načtení znaku *end of line* místo `XX` do stavu 0 a dopsáním přechodu ve všech stavech, které nejsou koncové (`l`, `XX`), při načtení znaku *konec*

souboru do stavu `XX`. Chceme-li nyní rozšířit automat pro 2. kritérium, tj. řádek nesmí obsahovat slovo začínající znakem `Q`, musíme doplnit do automatu ve všech nekoncových stavech přechod do stavu `M` při načtení mezery a dále doplnit stavy `M` a `YY`:

```
 M: case znak of
      end of line: stav:=0;
      end of file: stav:=XX;
      ' ': stav:=M;
      'q': stav:=YY;
      't': stav:=t;
      else stav:=0;
YY: case znak of
      end of line: stav:=0;
      end of file: stav:=XX;
      else stav:=YY;
```

Ve stavu `YY` čteme znaky až do konce řádky, neboť na řádce se slovem začínajícím `Q` určitě zaklínadlo není. Ještě musíme ošetřit, že za řetězcem `TFUJTAJXL` nebude na téže řádce slovo na `Q`, tedy dopíšeme ošetření do koncového stavu `l` takto:

```
 l: case znak of
      end of line: stav:=TRUE;
      end of file: stav:=TRUE;
      mezera: stav:=TM;
      else stav:=l;
TM: case znak of
      end of line stav:=TRUE;
      end of file stav:=TRUE;
      ' ': stav:=TM;
      'q': stav:=YY;
      else stav:=l;
```

Zbývá ošetřit první znak na řádce. Tomu nemůže předcházet mezera a může to být začátek slova na `Q`. Toto ošetření provedeme např. doplněním dalšího stavu, který prohlásíme za počáteční:

```
First: case znak of
      end of line: stav:=0;
      end of file: stav:=XX;
      Q: stav:=YY;
      t: stav:=t;
      else stav:=0;
```

Posledním kritériem, které musel splňovat nalezený řádek byl počet znaků na řádku dělitelný třemi. Pokud by nám nezáleželo na počtu proměnných, tak bychom to jednoduše zařídili tak, že bychom při každém načtení znaku zvýšili počitadlo znaků o 1. Ve stavu `First` bychom ho nulovali a ve stavu `TRUE` bychom otestovali, zda je toto číslo dělitelné třemi. Toto řešení ale vyžaduje další velikou proměnnou, neboť nevíme, jak dlouhé řádky jsou na vstupu. Abychom ušetřili datovou paměť, stačí nám pouze vědět, jaký je zbytek po dělení počtu znaků na řádce třema, tedy hodnoty 0, 1, 2. Nechceme-li zavádět další proměnnou, zařídíme to zvětšením počtu stavů na skoro trojnásobek: hodnotu si budeme pamatovat stavem. Stavy, v nichž ještě můžeme na řádce nalézt zaklínadlo, rozepíšeme podle následujícího příkladu pro stav `t`:

```
t_0: case znak of
      f: stav:=f_1;
      t: stav:=t_1;
      end of line: stav:= XX;
      else stav:=0_1;
t_1: case znak of
      f: stav:=f_2;
      t: stav:=t_2;
      end of line: stav:= XX;
      else stav:=0_2;
t_2: case znak of
      f: stav:=f_0;
      t: stav:=t_0;
      end of line: stav:= XX;
      else stav:=0_0;
```

Nyní už máme automat, který se zastaví ve stavu `TRUE`, pokud ve vstupním textu existuje řádka splňující všechna 3 kritéria. Vidíme, že jeho triviální implementace v Pascalu zabírá 2 proměnné: *znak* a *stav.* Proměnná *stav* není potřeba, neboť místo cyklu s příkazem `case`, můžeme použít příkazů `goto`, kterými skočíme přímo do cílového stavu, tedy každý příkaz `stav:=A` nahradíme `goto A`. V programu nám tedy zbyde poslední proměnná na načtení znaku ze vstupu. Když bychom chtěli a programovací jazyk nám to umožnil, v programu bychom se i této poslední proměnné zbavili nahrazením načtení znaku do proměnné a následných podmínek jedním složeným podmíněným příkazem, např v jazyce C by to byl `switch(getchar())`.

Popis programu: Vzhledem k tomu, ze jazyk Pascal nemá podporu maker, je program napsán v jazyce C, kde využíváme preprocesoru k výraznému zkrácení zdrojového kódu. Většina stavů, vyskytujících se v programu kvůli dělitelnosti třemi po trojicích, se pak napíše jedním makrem. Na ostatní přechody jsou použita makra `G` a `G1`, která se liší jen načítáním vstupu do proměnné *c.*

Chytré hlavy si jistě poradí s přepsáním programu tak, aby se proměnná c nepoužila.

Časová a paměťová složitost: Každý znak vstupu čteme jednou a přejdeme do jiného stavu po vyhodnocení maximálně čtyř podmínek, výsledná složitost je tedy $O(n)$. Spotřeba paměti je 1 znak (byte), bez níž se obecně neobejdeme, neboť musíme někam načíst vstup.

```
#include <stdio.h>

#define NL(x) if(c=='\n') goto NewLine; if(c==EOF) goto F; if(c=='t') goto T##x
#define B(x) if(c==' ') goto Blank##x
#define G(a,b) if(c==a) goto b
#define G1(A,B) if((scanf("%c",&c),c)==A) goto B
#define X1 goto One
#define X2 goto Two
#define X3 goto Three
#define Z(A,BB,CC) \
        A##1: if ((scanf("%c",&c),c)==BB) goto CC##2; if(c=='\n') goto NewLine; \
              if (c==EOF) goto F; if (c=='t') goto T2; \
              if (c==' ') goto Blank1; goto Two;  \
        A##2: if ((scanf("%c",&c),c)==BB) goto CC##3; if(c=='\n') goto NewLine; \
              if (c==EOF) goto F; if (c=='t') goto T3; \
              if (c==' ') goto Blank2; goto Three; \
        A##3: if ((scanf("%c",&c),c)==BB) goto CC##1; if(c=='\n') goto NewLine; \
              if (c==EOF) goto F; if (c=='t') goto T1; \
              if (c==' ') goto Blank3; goto One

int main(void) {
int c;                                              // vstupni znak
 NewLine:                                           // novy radek
 G1(EOF,F); G('q',Dumb); G('t',T2); NL(2); B(1); X2;

 One: G1('t',T2); NL(2); B(1); X2;                  // zacatek cteni
 Two: G1('t',T3); NL(3); B(2); X3;
 Three: G1('t',T1); NL(1); B(3); X1;

 Z(T,'f',Tf);                                       // precteno t
 Z(Tf,'u',Tfu);                                     // precteno f
 Z(Tfu,'j',Tfuj);                                   // precteno u
 Z(Tfuj,'t',Tfujt);                                 // precteno j
 Tfujt1: G1('a',Tfujta2); G('f',Tf2); NL(2);B(1);X2; // osetreni f
 Tfujt2: G1('a',Tfujta3); G('f',Tf3); NL(3);B(2);X3;
 Tfujt3: G1('a',Tfujta1); G('f',Tf1); NL(1);B(3);X1;
 Z(Tfujta,'j',Tfujtaj);                             // precteno j
 Z(Tfujtaj,'x',Tfujtajx);                           // precteno x
 Z(Tfujtajx,'l',Tfujtajxl);                         // precteno l

 Tfujtajxl1: G1('\n',TRUE); G(EOF,TRUE); G(' ',OK1); goto Tfujtajxl2;
 Tfujtajxl2: G1('\n',NewLine); G(' ',OK2); G('*',F); goto Tfujtajxl3;
 Tfujtajxl3: G1('\n',NewLine); G(' ',OK3); G('*',F); goto Tfujtajxl1;

 // precten tfujtajxl, cteni do konce radky
 OK1: G1('q',Dumb); if (c=='\n'||c==EOF) goto TRUE; G(' ',OK1); goto Tfujtajxl2;
 OK2: G1('q',Dumb); G('\n',NewLine); G(' ',OK2); G('*',F); goto Tfujtajxl3;
 OK3: G1('q',Dumb); G('\n',NewLine); G(' ',OK3); G('*',F); goto Tfujtajxl1;
```

```
 Blank1: G1('q',Dumb); NL(2);B(1); X2;              // nacteni mezery
 Blank2: G1('q',Dumb); NL(3);B(2); X3;
 Blank3: G1('q',Dumb); NL(1);B(3); X1;

 Dumb: G1('*',F); G('\n',NewLine); goto Dumb;       // cteni do konce radky
 TRUE: printf("Zaklinadlo nalezeno\n"); return 0;
 F: printf("Zaklinadlo nebylo nalezeno\n"); return 0;
}
```

11-1-4 Sibylla 98 — Martin Mareš

Řešení podúlohy **a)** je velice jednoduché: stačí si od věštírny pro každý předmět nechat nadiktovat, zda se má do batohu umístit či nikoliv a nakonec otestovat, jestli to dalo požadovanou hmotnost. To přesně činí tento program:

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

int main(void)
{
  int n,k,i;
  scanf("%d%d", &n, &k);
  while (n--)
    {
      scanf("%d", &i);
      k -= i * oracle(1);
    }
  return !k;
}
```

Důkaz správnosti: Posloupnosti odpovědí věštírny v tomto programu přesně odpovídají možnostem, jak předměty vybrat a akceptování této posloupnosti programem přesně odpovídá tomu, že předměty batoh zcela zaplní. Proto program odpoví ano (`1`) $\Leftrightarrow$ (podle definice Sibylly) existuje akceptovaná posloupnost odpovědí orákula $\Leftrightarrow$ (jak již víme) existuje množina předmětů, které batoh zaplní.

Časová složitost algoritmu je $O(n)$, jelikož každý prvek vstupu zpracujeme právě jednou. Paměťová složitost je konstantní, tedy $O(1)$.

S úlohou **b)** je to poněkud složitější. Většinu z vás sice napadlo, že stačí generovat všechny možnosti, jak mohlo orákulum odpovědět, a testovat, jestli je některá z nich správná, ale často jste přehlédli několik drobností:

- Existují programy, které jsou konečné, ale mají nekonečné větve výpočtu. Tak například

  ```
  if (oracle(1) == 1)
  ```

```
    for (;;) ;
return 1;
```

je program, který má jednu konečnou a jednu nekonečnou větev, ale podle definice časové složitosti nedeterministického programu je konečný.

- Větví výpočtu, jakož i volání funkce *oracle* v nich může být nekonečně mnoho. Třeba:

```
while (oracle(1) == 1) ;
return 1;
```

Tento prográmek má nekonečně mnoho větví výpočtu, přičemž každá větev má jiný počet volání orákulní funkce.

A jak na to tedy půjdeme? Inu, nejdříve si původní nedeterministický program $N(x)$ (x je posloupnost čísel na vstupu) krapet upravíme:

1. Z původního orákula uděláme orákulum binární, které vždy odpovídá buďto nulou nebo jedničkou. To lze, protože každé číslo si můžeme nechat poradit po bitech a v případě, že vyjde více než daná horní mez, prostě příslušnou větev výpočtu zrušit s výsledkem 0. A jak určíme počet bitů, které nám má orákulum prozradit? Třeba tak, že se za každým bitem ještě orákula zeptáme, zda máme pokračovat či nikoliv.
2. Do programu doplníme počítadlo příkazů a přidáme parametr T, který bude udávat, kolik příkazů smíme nejvýše provést. Pokud tento počet překročíme, vrátíme nulu.
3. Každý orákulový dotaz nahradíme prostým přečtením jednoho bitu z čísla R.

Tímto postupem program $N(x)$ upravíme na program $\check{N}(x, R, T)$ a tvrdíme, že $N(x) = 1$ právě tehdy, když existují nezáporná čísla R a T taková, že $\check{N}(x, R, T) = 1$. Důkaz: První úprava evidentně správnost programu nemění. Je-li $N(x) = 1$, pak podle definice nedeterministického výpočtu existuje větev výpočtu (to jest posloupnost odpovědí orákula, námi zakódovaná v čísle R), která vrátí jedničku. Tato větev ovšem také musí skončit, a to za nějakých T kroků, což však přesně odpovídá tomu, že $\check{N}(x, R, T) = 1$.

Nyní již zbývá jenom napsat program, který bude postupně procházet všechna T počínaje nulou, pro každé z nich vygeneruje všechna možná R ($0 \leq R < 2^T$, jelikož počet volání orákula je maximálně roven počtu kroků programu $\check{N}$) a otestuje, zda $\check{N}(x, R, T)$ je nenulové. Jestliže taková R, T nalezneme, pak dle předchozího tvrzení je i $N(x) = 1$. Opačně, pokud $N(x) = 1$, pak taková R, T existují, a proto je nalezneme (zkoušíme všechny možnosti a každou z nich zpracujeme v čase $O(T)$).

Tím jsme dokázali, že deterministickými algoritmy lze spočítat totéž, co nedeterministickými (odhlédneme-li ovšem od poločasu rozpadu našeho počítače).

[Jiná možnost je dívat se na jednotlivé stavy programu (to jest pozice v programu plus obsah všech proměnných) jako na vrcholy nějakého grafu, na přechody mezi stavy jako na hrany tohoto grafu a na možné průběhy výpočtu jako na cesty začínající v nějakém pevném počátečním vrcholu. Deterministickému výpočtu odpovídá, že z každého vrcholu vychází maximálně jedna hrana, v případě nedeterministických algoritmů jich může být i více. Ptáme se tedy, zda pro daný graf existuje cesta, která vede do vrcholu, v němž se vrací výsledek rovný jedné. Jenže tento graf může být nekonečný, takže nám nezbývá, než jej prohledávat do šířky a vrcholy i hrany generovat tak, jak jsou během prohledávání navštěvovány. Správnost plyne z těchže úvah jako správnost předchozího převodu.]

11-2-1 Kva-kva-kvadrant — Robert Špalek

Řešení této úlohy má dvě hlavní části: první je zjednodušování posloupnosti transformací a druhá je jejich provádění. Na každé z nich se dalo mnoho optimalizovat – při zjednodušování se ušetřil čas a při transformování paměť. Nechť počet transformací je M a délka kódu obrázku je N.

1. Zjednodušování permutací

Jak již bylo v zadání zdůrazněno, posloupnost transformací může být velice dlouhá, a není proto radno aplikovat je postupně všechny (M krát) na každý obrázek. Stačí si všimnout těchto základních vlastností transformací:

1) Inverze a ostatní transformace se neovlivňují. Černá a bílá se prohazují právě tehdy, pokud je počet inverzí lichý.
2) Všechny popsané transformace (X, Y, $+$, $-$) aplikované na 4 podoblasti jsou obyčejné permutace čtyřprvkové množiny. Jako takové se dají skládat. Místo postupného provádění dílčích permutací tedy můžeme provést všechny najednou. Při dalším zpracování samozřejmě nesmíme zapomenout, že stejnou permutaci musíme provést rekurzivně i na dalších podoblastech podoblastí.

Na počátku programu nadefinujeme tabulkou 4 základní transformace a uložíme si identitu jako počáteční stav. V proceduře `nacti_transformace` po každém načteném znaku složíme aktuální permutaci s permutací odpovídající načtenému znaku, resp. změníme paritu počtu výskytů inverze. Výsledkem je permutace ekvivalentní dané posloupnosti permutací. Permutace čtyřprvkové množiny skládáme v konstantním čase, zjednodušení nám zabere celkem $O(M)$ kroků. Paměti potřebujeme konstantně mnoho $O(1)$.

Pár řešitelů se složitě pokoušelo nalézt algebraické metody, jak posloupnost transformací zjednodušit. To bylo značně složité a použitelné pouze na tomto konkrétním zadání. Výše uvedená metoda může pracovat na libovolné sestavě základních permutací jakékoliv množiny. Kromě toho je jednodušší a pochopitelnější.

2. Transformace kódu obrázku

V druhé části úlohy většina z vás správně odhadla, že by se měl použít strom. Jeho struktura odpovídá zadání, tzn. pokud je oblast jednobarevná, uložíme si její barvu, v opačném případě zapíšeme 4 ukazatele na stromy 4 kvadrantů – podoblastí dané oblasti.

Při řešení této úlohy bylo nutné zvládnout 3 fáze: konverze vstupního kódu do stromu, transformace stromu podle zadání a výpis modifikovaného stromu. Zajímavé je, že 2. fázi nemusíme vůbec provádět, protože se jedná o pouhé přeházení pořadí kvadrantů. To můžeme přece udělat až při výstupu. Oba směry konverze mezi textovou a stromovou reprezentací lze nejjednodušeji naprogramovat rekurzivními procedurami.

Většina řešitelů celkem neefektivně alokovala dynamické datové struktury a propojovala je ukazateli. To je celkem pomalé a pamětí plýtvající řešení. Když už máme znaky jednou načtené v poli (vstupní řetězec), nejjednodušší je použít přímo toto pole. Jako ukazatel použijeme index do pole, hodnotou bude znak 0, 1, 2 přímo na daném místě. K uložení stromové struktury potřebujeme ještě u znaků 2 uložit pozice 4 synů.

Kdybychom použili takto naivní postup, potřebujeme na každý znak v poli uložit 4 čísla, což zabere až $4*4 = 16$ bajtů. Většinu takto naalokovaného místa přitom nepoužijeme, protože plno oblastí bude jednobarevných. Zkusme na to jít jinak. Existuje metoda, pomocí které můžeme pouhým binárním stromem zaznamenat libovolně rozvětvený strom s různými stupni vrcholů. Místo toho, abychom zaznamenávali všechny informace o synech v daném vrcholu, propojíme syny do spojového seznamu. Levý syn vrcholu bude nyní ukazatel na prvního původního syna, pravý syn vrcholu zase ukazatel na původního bratra nebo `NULL` pokud to byl poslední bratr. Vzhledem k tomu, že pozice prvního syna je v našem příkladě jasná (je-li na pozici k rozvětvovací znak 2, pak levý horní kvadrant, což je první syn, začíná na pozici $k + 1$), stačí ukládat pouze pozici pravého bratra.

Strom lze tedy snadno vytvořit pomocí rekurzivní funkce `dekoduj_obraz`, která pro jednobarevnou oblast 0 nebo 1 (triviální případ) neudělá nic, pro dělenou oblast 2 propojí syny spojovým seznamem a vrátí celkovou délku celé oblasti. Pomocí této vrácené délky může její volající zase vyplnit spojový seznam svých bratrů. Každá oblast je takto zapojena do seznamu maximálně jednou a každá dělená oblast se maximálně jednou takto zpracuje, takže celkový čas bude $O(N)$ s pamětí $O(N)$ pro kód obrazu.

Transformaci a zápis stromu zajistí funkce `zapis_ztransformovano`, která triviální jednobarevnou oblast rovnou vypíše (po případné inverzi) a složenou oblast zpracuje takto: pročtením spojového seznamu zjistí pozice 4 synů, načež rekurzivně vyvolá sama sebe ve zpermutovaném pořadí pro každou podoblast. Tím pádem nejenom převedeme strom na text, ale dokonce ho i ztransformujeme podle dané permutace. Opět zpracujeme každý znak právě jednou, což nám zabere čas $O(N)$.

Celkově program vystačí s pamětí $O(N_{max})$, kde N_{max} je délka nejdelšího kódu obrázku, a poběží $O(M + \sum N_i)$ kroků, kde N_i jsou délky jednotlivých kódů.

```
  /* Kva-kva-kvadrant, 10-2-1 */

#include <stdio.h>

#define POCET_TRANSF 4
#define POCET_POLI 4
#define ERROR (x...) {fprintf (stderr, x); exit (1); }
#define MAX_DELKA_KODU 10000

char zn_transf[POCET_TRANSF+1]="+-XY";
int typ_transf[POCET_TRANSF][POCET_POLI]={
            {1, 3, 0, 2},                   /* +  */
            {2, 0, 3, 1},                   /* -  */
            {1, 0, 3, 2},                   /* X  */
            {2, 3, 0, 1}                    /* Y  */
};
int transf[POCET_POLI]={0, 1, 2, 3};        /* na počátku je identita */
int inv=0;                                  /* a bez inverze */

  /* uloží do transf permutaci, která je ekvivalentní celému dlouhému řetězci vstupních
       transformací; v inv je uloženo, zda se invertují barvy */
void nacti_transformace (){
            char zn;
            int i, j;
            while ( (zn=getchar ())!='\n'){
                        if (zn=='I'){               /* inverze */
                                    inv=!inv;
                        }else{
                                    int nova[POCET_POLI];
                                    i=0;            /* hledej transformaci */
                                    while (i<POCET_TRANSF && zn_transf[i]!=zn)
                                                i++;
                                    if (i>=POCET_TRANSF)
                                                ERROR ("neznama transformace %c", zn);
                                    /* složení permutací */
                                    for (j=0; j<POCET_POLI; j++)
                                                nova[j]=transf[typ_transf[i][j]];
                                    memcpy (transf, nova, sizeof (nova));
                        }
            }
}
```

```
  /* pokud je kod[idx]=='2', pak rekurzivně zjistí indexy všech 4 podoblastí; pozice 1. syna
     je zřejmě o 1 znak dále, každý syn bude v poli bratr ukazovat na svého dalšího bratra
     až do počtu 4. funkce vrátí celkovou délku kódu této oblasti */
int dekoduj_obraz (char *kod, int *bratr, int idx){
        int sum=1;
        if (kod[idx]=='2'){                    /* děleno na podoblasti */
                int i;
                idx++;
                for (i=0; i<POCET_POLI; i++){
                        int len=dekoduj_obraz (kod, bratr, idx);
                        bratr[idx]=idx+len;
                        idx+=len;
                        sum+=len;
                }
        }
        return sum;
}
  /* rekurzivně vypíše kód celé podoblasti; barvy se vypisují triviálně (případně prohozené
     podle inv), při zpracování podoblastí nejprve načteme souřadnice 4 synů a posléze se
     ve vhodném pořadí rekurzivně vyvoláme */
void zapis_ztransformovano (char *kod, int *bratr, int idx){
        if (kod[idx]!='2'){                    /* vypsání základní barvy */
                putchar (kod[idx]^inv);    /* případná inverze */
        }else{
                int syn[POCET_POLI], i;
                syn[0]=idx+1;                  /* 1. syn následuje */
                putchar ('2');
                for (i=1; i<POCET_POLI; i++)   /* načti z link-listu synů */
                        syn[i]=bratr[syn[i-1]];
                for (i=0; i<POCET_POLI; i++)   /* vypiš vhodně 4 podoblasti */
                        zapis_ztransformovano (kod, bratr, syn[transf[i]]);
        }
}
int main (void){
        char kod[MAX_DELKA_KODU];
        nacti_transformace ();
        do{
                int bratr[MAX_DELKA_KODU];    /* číslo dalšího bratra */
                scanf ("%s", kod);
                dekoduj_obraz (kod, bratr, 0);    /* od kořene (0) očíslujeme */
                zapis_ztransformovano (kod, bratr, 0);    /* od kořene (0) */
                printf ("\n");
        }while (!feof (stdin));
        return 0;
}
```

11-2-2 Zákeřná posloupnost — Ája Jančaříková

Zákeřnost naší posloupnosti je v její délce. Je totiž dlooouhatánská, což znamená, že se nevejde do žádného pole ani souboru a dokonce ani na celý disk. Proto bylo nutné najít takový algoritmus, který není paměťově a ani časově náročný. Takový algoritmus existuje a využívá funkce XOR, která má následující vlastnosti:

1) Komutativnost: a xor $b = b$ xor a
2) Asociativnost: (a xor b) xor $c = a$ xor (b xor c)
3) $\forall a$ platí, že a xor $a = 0$
4) $\forall a$ platí, že a xor $0 = a$

Pokud postupně xorujeme čísla v dané posloupnosti, bude výsledkem číslo, které je v posloupnosti pouze jednou. Proč?

Každé číslo mimo b je v posloupnosti dvakrát. Díky vlastnosti 1) můžeme posloupnost xorů určitě přerovnat na: b xor a_1 xor a_1 xor ... xor a_n xor a_n. Díky vlastnostem 2) a 3) pak víme, že výsledek bude b xor 0, což je podle 4) rovno b.

Časová složitost bude lineární s délkou posloupnosti (lépe už to ani nemůže jít – musíme načíst celou posloupnost), paměťová bude konstantní.

```
int main (void)
{
    int m,  x = 0;
    while  (scanf ("%d",  &m) == 1) x ^= m;
    printf ("%d\n",  x);
    return 0;
}
```

11-2-3 Případ scottského skota — Jan Hubička

Valnou většinu řešitelů napadl známý Dijkstrův algoritmus. U tohoto algoritmu je ale problém se správou prioritní fronty, která je v optimální implementaci pomocí Fibonacciho hald sice docela rychlá, ale zato značně komplikovaná. Nicméně použití Dijkstrova algoritmu pro tento problém je vrhání atomových náloží na vrabce.

Úlohu lze řesit jednoduchou modifikací algoritmu vlny. To je algoritmus, který lze použít ke hledání nejkratší cesty v jednoduchém bludišti (bez dveří). Algoritmus postupně počítá velikosti nejkratších cest do jednotlivých políček bludiště. Nejprve nastaví délku cesty vstupního políčka na 0. Dále následují jednotlivé etapy (vlny), které vždy označí všechna ještě neoznačená políčka sousedící s označeným políčkem z minulé etapy. Jejich vzdálenost se snadno určí tak, že se přičte 1 ke vzdálenosti jeho označeného souseda.

V okamžiku, kdy algoritmus označí políčko s pokladem, je vyhráno. U každého políčka je dobré si poznamenat, od jakého souseda se na něj přišlo. Cestou zpět se pak stačí řídit těmito ukazateli a musíme dojít nejkratší cestou ke vchodu. Aby se cesta nevypisovala odzadu, lze použít jednoduchý trik a prohodit vchod a poklad v bludišti. Pokud algoritmus v nějaké etapě neoznačí žádná nová políčka, cesta k pokladu neexistuje.

Nejdelší čas algoritmus tráví při hledání sousedů označených políček z minulé etapy. Tato práce je ale naprosto zbytečná. Je možné si vytvořit frontu a už při zpracovávání políčka zařazovat jeho sousedy na konec fronty. Další políčko na zpracování se pak získá vybráním prvního prvku fronty. Tím zmizí i hranice etap. Políčka z jedné etapy se dostanou do fronty dříve, než políčka z následujících etap. Tím pádem se algoritmus o žádné etapy starat nemusí. Pokud cesta k pokladu nevede, algoritmus vyplní celé dostupné bludiště a pak vyprázdní frontu. Tak se snadno pozná, že cesta neexistuje.

Komplikaci přinášejí dveře. Pokud algoritmus bude brát dveře jako místnosti, najde sice cestu, ale chudák Scott o dveře omlátí nejeden krumpáč. Zkusíme tedy náš algoritmus poněkud vylepšit. Pro začátek budeme o dveřích uvažovat jako o zdech a nalezené dveře si pouze budeme řadit do nějaké pomocné fronty. Když už projdeme všechna políčka v dané oblasti (část bludiště dosažitelná bez prokopnutí dveří), začneme prokopávat dveře. Prokopneme tedy první dveře (zařadíme je do fronty ke zpracování), na které jsme narazili, a za nimy pokračujeme v šíření stejně jako na počátku, až na jednu malou úpravu. V případě, že je vzdálenost zařazovaných políček stejná jako vzdálenost prvních dveří v pomocné frontě (ty zároveň budou jistě také nejbližší), zařadíme tyto dveře do fronty jako normální políčko ke zpracování (samozřejmě z pomocné fronty dveře odebereme). Takto si zajistíme, že pokud jde mezi dvěma oblastmi projít různými dveřmi, vybere si Scott ty nejvhodnější (tzn. ne nutně ty nejbližší). Ještě si musíme zajistit, abychom takto nezařadili dveře, ke kterým jsme se dostali na větší počet prokopnutí. To můžeme udělat snadno – třeba tak, že si budeme udržovat 2 pomocné fronty na dveře. Jednu, ze které odebíráme dveře, a druhou, do které dveře přidáváme (ke dveřím z této fronty jsem se dostal až po prokopnutí dveří z předchozí fronty). Když se fronta, ze které odebírám, vyprázdní a nebude už kam jít – prošel jsem všechny oblasti dosažitelné jedním prokopnutím – začnu odebírat z druhé fronty a zařazovat zase do nějaké další…

Pro důkaz si poněkud předefinujeme význam slova vzdálenost. Pro nás bude vzdálenost políčka nějaké dvojsložkové číslo $X = (x_1, x_2)$, kde x_1 je počet prokopnutí nutný k dosažení políčka a x_2 pak počet políček, kterými k tomuto políčku musíme na daný počet prokopnutí projít. Řekneme, že $A < B$ – políčko A je blíže než políčko B, právě když $a_1 < b_1 \vee (a_1 = b_1 \wedge a_2 < b_2)$.

Nyní si ukážeme, že náš algoritmus v průběhu celého výpočtu dodrží následující podmínky:

1) V každé etapě vyřídí všechna políčka s nějakým ohodnocením w.
2) Na konci každé etapy budou mít všechna nevyřízená políčka větší vzdálenost, než w.

Z těchto podmínek už správnost algoritmu snadno vyplyne – když skončí etapa vyřizující políčka s ohodnocením w a nenalezneme poklad, víme, že jeho vzdálenost je jistě větší, než w (plyne z podmínky 2)). Pokud v nějaké etapě na poklad narazíme, víme, že všechna nevyřízená políčka mají stejnou nebo větší vzdálenost, než w, a tedy se k pokladu rozhodně nemůžeme dostat lépe.

To, že náš algoritmus bude dané podmínky splňovat je poměrně jasné. Pro počáteční etapu podmínky jistě platí (políčko je jen jedno a všechna nově dosažená políčka mají nenulovou vzdálenost). V další etapě vezmeme všechna „nedveřní“ políčka z předchozí etapy. Ta budou jistě mít stejnou vzdálenost. Stejnou vzdálenost navíc mohou mít také dveře zbylé z procházení předchozích oblastí. Ty si ale hlídáme a v případě, že skutečně nějaké dveře stejnou vzdálenost mají, tak je též zpracujeme. První podmínka tedy bude splněna. Druhá ale také, protože pro každou etapu vybíráme w co nejmenší (čtenář jistě sám snadno uváží, že naše fronta bude vlastně setříděna podle vzdálenosti a vybráním jejího prvního prvku tedy vezmeme minimum), a protože všechna nově dosažená políčka budou mít jistě větší vzdálenost než w. Pokud tedy algoritmus skončí, dá správný výsledek. Algoritmus ale jistě skončí po nějakém konečném počtu kroků, protože v každém kroku vyřídí právě jedno políčko a tím se již nikdy nebude zabývat. Tím jest ukázáno.

Časová náročnost tohoto algoritmu je $O(N)$, kde N je počet políček v bludišti, protože se každé políčko zpracovává v konstantním čase maximálně jednou. Paměťová náročnost je též lineární s velikostí bludiště.

Program je ve své podstatě přímou implementací algoritmu. Za zmínku snad stojí pouze to, že frontu na dveře si udržuji jen jednu a pouze si pamatuji, kde v ní končí dveře dosažitelné na N prokopnutí a začínají dveře dosažitelné na $N + 1$ prokopnutí…

```
#include <stdio.h>
#define MAXN 200                    /* Maximální velikost bludiště */
#define DELKA (MAXN*MAXN)           /* Maximální délka fronty */
struct pozice {
      int x, y;
};
  /* Fronty jsou kruhové buffery pevné velikosti. */
struct fronta {
      int zacatek, konec;
      struct pozice pozice[DELKA];
} mistnosti, dvere;
```

```
int matice[MAXN][MAXN];                 /* bludiště uložené takto: */
#define MISTNOST 0
#define ZED 1
#define DVERE 2
#define POKLAD 3
#define VCHOD 4
#define ZPRACOVANO 5
int velikost = 0;                       /* hrana bludiště */
struct pozice kam[MAXN][MAXN];          /* kudy z daného políčka vede nejkratší cesta k
                                           pokladu */
int vzdal[MAXN][MAXN];                  /* vzdálenost jednotlivých políček */
int dalsikop;                           /* index prvních dveří, na které je třeba o kop
                                           více */

  /* Přidání políčka do fronty */
void pridej (struct fronta *f, int x, int y)
{
    f->pozice[f->konec].x = x;
    f->pozice[f->konec].y = y;
    f->konec = (f->konec + 1) % DELKA;
}

  /* Vyzvedne políčko z fronty. Vrací 0, pokud je fronta prázdná */
int vyzvedni (struct fronta *f, int *x, int *y)
{
    if (f->konec == f->zacatek)
      return 0;
    *x = f->pozice[f->zacatek].x;
    *y = f->pozice[f->zacatek].y;
    f->zacatek = (f->zacatek + 1) % DELKA;
    return 1;
}

  /* Vrátí vzdálenost prvmího políčka ve frontě */
int vzdalenost (struct fronta *f)
{
    if (f->zacatek == f->konec)         /* Je fronta prázdná? */
      return -1;                        /* Vrátíme neexistující vzdálenost... */
    return vzdal[f->pozice[f->zacatek].x][f->pozice[f->zacatek].y];
}

  /* Vypíše nejkratší cestu */
void vypis (int x, int y)
{
    printf ("Nejsnadneji ziskas poklad takto:\n");
    while (x >= 0) {
      int x1;
      printf ("%d %d\n", x, y);
      x1 = x;
      x = kam[x][y].x;
      y = kam[x1][y].y;
    }
}
```

```
  /* Přidá dané políčko spolu se stejně vzdálenými dveřmi do fronty */
void pridej_mistnost (int x, int y)
{
    int dx, dy;
    pridej (&mistnosti, x, y);
    /* Je teď průchod dveřmi stejně výhodný jako momentální pozice? */
    while (vzdalenost (&dvere) == vzdal[x][y] && dalsikop > dvere.zacatek)
    {
      /* Zařadíme dveře do fronty k provedení */
      vyzvedni (&dvere, &dx, &dy);
      pridej (&mistnosti, dx, dy);
    }
}

  /* Zpracuje políčko - porozhlédne se po sousedech. Vrací 1, pokud cesta byla nalezena. */
int zpracuj (int x, int y)
{
    int x1, y1;

    for (x1 = x - 1; x1 <= x + 1; x1++)
      for (y1 = y - 1; y1 <= y + 1; y1++)
        if ( ( (x1 != x) ^ (y1 != y)) &&
        /* tato podmínka zajistí, že se nebudu zabývat políčky, která sousedí pouze
             vrcholem. Pozn. ^ znamená bitový xor */
          x1 >= 0 && y1 >= 0 && x1 < velikost && y1 < velikost &&
          matice[x1][y1] != ZPRACOVANO && matice[x1][y1] != ZED)
        {
          kam[x1][y1].x = x;
          kam[x1][y1].y = y;
          if (matice[x1][y1] == VCHOD) {   /* Jsme v cíli? */
            vypis (x1, y1);
            return 1;
          }
          matice[x1][y1] = ZPRACOVANO;
          vzdal[x1][y1] = vzdal[x][y] + 1;
          /* Zařadíme sousední políčko do příslušné fronty */
          if (matice[x1][y1] == DVERE)
            pridej (&dvere, x1, y1);
          else
            pridej_mistnost (x1, y1);
        }
    return 0;
}
  /* načtení matice */
void cti ()
{
    int i, j;

    scanf ("%d", &velikost);
    getchar ();                              /* Načteme konec řádky */
    for (i = 0; i < velikost; i++)
    {
      for (j = 0; j < velikost; j++)
```

```
            matice[j][i] = getchar () - '0';
        getchar ();                          /* Načteme \n */
    }
}
int main (void)
{
    int x, y;

    cti ();
    /* Poklad je třeba zařadit do fronty, aby se algoritmus rozjel. Musím jej přidat jako
       dveře, protože algoritmus začíná "dveřní" vlnou */
    for (x = 0; x < velikost; x++)
      for (y = 0; y < velikost; y++)
        if (matice[x][y] == POKLAD) {
          pridej (&dvere, x, y);
          matice[x][y] = ZPRACOVANO;
          kam[x][y].x = -1;
        }

    while (1) {
      if (vyzvedni (&dvere, &x, &y))   /* Vybere další dveře - jsou ještě? */
      {
        /* Přecházíme do oblasti, na kterou je třeba další kop? */
        if (dvere.zacatek > dalsikop)
          dalsikop = dvere.konec;
        pridej_mistnost (x, y);          /* Přidáme všechny stejně vzdálené dveře */
      }
      else
        break;

      /* Projdeme celou oblast... */
      while (vyzvedni (&mistnosti, &x, &y))
        if (zpracuj (x, y))
          return 0;
    }
    printf ("Cesta neexistuje. Smula, penize nebudou.\n");
    return 0;
}
```

11-2-4 Alea iacta est — Daniel Kráľ

Nejprve se zkusme zamyslet nad tím, jak vypadá pohyb robota. Uvažme nějaký korektní program pro robota – předpokládejme, že příkaz pro pohyb robota ($G(a)$) se pravidelně střídá s příkazy pro otočení (L, R); to můžeme předpokládat, neboť dva příkazy pro pohyb, které jsou v programu za sebou, lze sloučit beze změny funkčnosti programu v jeden a dva příkazy pro otočení za sebou být nemohou, protože by robot do bodu, kde právě je, položil dvě kostičky. Na tomto místě se sluší poznamenat, že nyní uvažujeme pouze ty případy, kdy zadaných n bodů je navzájem různých. Každý si jistě sám snadno rozmyslí, jak by se řešení a naše úvahy změnily, kdyby tomu tak nebylo.

Směr, do kterého vyrazí robot na začátku svého pohybu, si nazvěme vodorovný. Pohyb robota se skládá z pravidelně se střídajících vodorovných a svislých úseků. Robot dorazí, např. ze svislého směru, do bodu X, zde se otočí doprava nebo doleva o 90 stupňů, pokračuje vodorovně do bodu Y, v něm se znovu otočí a poté opustí vodorovnou přímku (řádek), na které leží body X a Y. Leží-li na tomto řádku ještě nějaké další body, např. A, pak robot někdy dorazí (opět ze svislého směru) do bodu A a tento řádek opět opustí v nějakém dalším bodě. Pohyb robota, tedy body na stejném řádku „spároval" – jedním na řádek „vstoupil" a druhým z řádku „odešel". Na každém řádku (analogicky i sloupci), pokud existuje korektní program pro robota, leží tedy sudý počet bodů. Výjimku může tvořit pouze řádek a sloupec s počátečním bodem. Pokud bude v tomto řádku i sloupci lichý počet bodů, program bude též existovat. V tomto případě totiž skončí robot natočen do svislého směru, čímž bod „ušetříme". Detailní rozbor tohoto případu necháváme na laskavém řešiteli.

Pokud existuje program pro robota, tak platí, že pro každý zadaný bod X existuje program, že se robot postupující podle něj přesune z počátečního bodu M do bodu X, přičemž se bude otáčet pouze v ostatních zadaných bodech. Stačí totiž uvážit tu počáteční část programu řešícího naši úlohu, která končí otočkou robota v bodě X.

Než přistoupíme k vyřešení zadané úlohy, zaměřme pozornost na dvě chyby, které se v řešeních často vyskytovaly. V zadání úlohy není řečeno, že se robot může pohybovat pouze rovnoběžně s osou x nebo y; naopak je tam zdůrazněno, že počáteční směr robota, lze určit libovolně a tedy je třeba za „vodorovný" směr z minulého odstavce postupně volit směry z bodu M do všech zadaných bodů. Řada z řešitelů poté konstatovala, že tato úloha (pro pevně zvolený „vodorovný" směr) je hledáním hamiltonovské kružnice v grafu s jistými omezujícími podmínkami – vrcholy jsou zadané body, hrany jsou mezi body na stejném řádku nebo sloupci. To, že pro hledání hamiltonovské kružnice v grafu není znám polynomiální algoritmus, však neznamená, že námi zadanou úlohu nelze vyřešit polynomiálně – výše naznačená úvaha pouze znamená, že polynomiální algoritmus pro hledání hamiltonovské kružnice v grafu, by nám (možná) pomohl vyřešit naši úlohu v polynomiálním čase; nikoliv však, že z polynomiálního algoritmu pro naši úlohu se nám podaří sestrojit algoritmus pro hledání hamiltonovské kružnice v obecném grafu, protože ty grafy, které odpovídají rozmístění bodů do roviny, určitě nejsou všechny grafy, které existují, a pouze pro tyto grafy jsme schopni nalézt hamiltonovskou kružnici v polynomiálním čase a to ještě s nějakými omezujícími podmínkami. Naši úlohu lze však převést na hledání eulerovského tahu v grafu, kde vrcholy jsou řádky a sloupce, a vrchol odpovídající řádku je spojen hranou s vrcholem odpovídajícím sloupci, pokud

existuje bod na průsečíku příslušného řádku a sloupce; vrcholy odpovídající řádkům, resp. sloupcům, nejsou nikdy spojeny navzájem hranou.

Nyní předpokládejme, že počáteční směr robota máme již zvolen, a chceme rozhodnout o existenci a případně i nalézt program pro robota, který vyhovuje zadání úlohy. Začněme sestrojovat program pro pohyb robota z bodu M „hladově“. Pokud ve směru, do kterého je robot natočen, leží nějaký nenavštívený bod, přidejme do programu příkaz, který přesune robota do tohoto bodu. Pak robota otočme (pokud je to možné) tak, aby se díval směrem k některému z bodů, který dosud nenavštívil. Takto postupujme, dokud je to možné. Skončí-li robot svou pouť v jiném bodě, než v bodě M, potom v řádku, resp. sloupci, ve kterém se měl robot nyní pohybovat, je lichý počet bodů a tedy korektní program pro robota neexistuje. Pokud robot doputoval až do bodu M, rozlišíme 2 případy. Prošel-li robot všemi body, které jsou zadány, nalezli jsme program řešící naši úlohu. V opačném případě, zkusme najít bod Y, který robot nenavštívil a je přitom ve stejném řádku nebo sloupci jako některý z navštívených bodů (ten si označme X). Pokud takový bod Y neexistuje, tak pro žádný z dosud nenavštívených bodů neexistuje program, který by dovedl robota z bodu M do tohoto bodu tak, jak je poznamenáno na konci předminulého odstavce. Tedy opět určitě neexistuje program pro robota, který vyhovuje podmínkám úlohy. V opačném případě existuje výše zmíněná dvojice bodů X a Y. Dosud vytvořený program P pro pohyb robota, pozměňme tak, že robot nejprve vykoná část programu P od návštěvy bodu X, přijde tedy do bodu M, jím projde, a vykoná část programu po návštěvu bodu X, nyní se natočí tak, aby mohl dále pokračovat přes bod Y a program prodlužujeme dále „hladově“. Pokud, již dále nelze postupovat a nevrátili jsme se do bodu Y jsme v řádku nebo sloupci s lichým počtem zadaných bodů a program tedy neexistuje. V opačném případě, nově vytvořenou část programu vložíme do programu P do místa, kde procházíme bodem X. Samozřejmě je potřeba si ještě rozmyslet, že programy půjdou napojit a jak tak učinit, ale to jistě každý zvládne sám.

Nyní přejděme k realizaci výše naznačeného algoritmu. Nejprve načteme souřadnice bodu M a bodů, kterými má robot projít. Body si v rovině posuneme tak, aby bod M byl v počátku souřadnic. Nyní postupně pro každý bod X ze zadaných bodů, otočíme body kolem počátku tak, aby bod X ležel v kladné části x-ové osy. Směr z bodu M do bodu X bude pro nás „vodorovný“. Přepočítané souřadnice bodů setřídíme zvlášť podle x-ové a podle y-ové složky – to nám umožní rychleji rozpoznat, zda dva body leží ve stejném řádku nebo sloupci. V programu ještě k zadaným bodům přidáme bod M, čímž se vyhneme ošetřování okrajových případů v proceduře `vytvorplan`. Nyní vytvoříme plán cesty robota – pro každý bod určíme, který bod robot navštíví po tomto bodě. Požadujeme navíc, aby první a poslední bod v našem plánu byl bod M. Nejprve sestrojíme nějaký plán z bodu M do bodu M. Pak půjdeme po jednotlivých

bodech v tomto plánu a budeme zkoumat, zda některý z nich leží ve stejném řádku nebo sloupci jako některý z dosud nenavštívených bodů. Pokud takový bod neexistuje a robot neprošel všemi zadanými body, program, jak již bylo výše zdůvodněno, nelze sestrojit. Pokud robot prošel všemi zadanými body, máme pro robota plán, jak projít všemi body, a sestavit z něj program je již jednoduché.

Nyní provedeme odhad časové a paměťové složitosti programu. Nechť je na vstupu zadáno n bodů. Program celkem n-krát zvolí vodorovný směr. Pro každou z těchto voleb, přepočítá souřadnice bodů, setřídí je dle jejich souřadnic a pokusí se sestavit program pro robota. K setřídení bodů je třeba čas $O(n \log n)$, ostatní části se provedou v lineárním čase vzhledem k n. Celková časová složitost je tedy $O(n^2 \log n)$, paměťová $O(n)$.

```
program robot;
const MAX=100;                  {maximální počet zadaných bodů}
      pi=3.1415926535;          {pi}
      chyba=1e-8;               {nepřesnost operací s typem real}
var n:word;
    n0:word;                    {počet zadaných bodů}
    mx,my:real;                 {souřadnice bodu M}
    x,y:array[1..MAX] of real;  {souřadnice "zadaných" bodů}
    setridx,setridy:array[0..MAX+1] of word;
                                {body setříděné dle souřadnic}
    navstivil:word;             {počet bodů, které robot navštívil}
    body:array[1..MAX] of
          record
           setrx,setry:word;    {pořadí podle x/y-ové souřadnice}
           nasl:word;           {další bod, který robot navštíví}
           navst:boolean;       {byl tu už robot?}
          end;
    i,j:word;                   {pomocné proměnné}
procedure qsortx(p,q:word);     {quicksort dle x-ové souřadnice}
var i,j,k:word;
    pivot:real;
begin
  i:=p; j:=q; pivot:=x[setridx[(p+q) div 2]];
  repeat
    while x[setridx[i]]<pivot do inc(i);
    while pivot<x[setridx[j]] do dec(j);
    if i<=j then
    begin
      k:=setridx[i]; setridx[i]:=setridx[j]; setridx[j]:=k;
      body[setridx[i]].setrx:=i; body[setridx[j]].setrx:=j;
      inc(i); dec(j);
    end
  until i>j;
  if p<j then qsortx(p,j);
  if i<q then qsortx(i,q)
end;
procedure qsorty(p,q:word);     {quicksort dle y-ové souřadnice}
var i,j,k:word;
    pivot:real;
begin
```

```
 i:=p; j:=q; pivot:=y[setridy[(p+q) div 2]];
 repeat
   while y[setridy[i]]<pivot do inc(i);
   while pivot<y[setridy[j]] do dec(j);
   if i<=j then
   begin
     k:=setridy[i]; setridy[i]:=setridy[j]; setridy[j]:=k;
     body[setridy[i]].setry:=i; body[setridy[j]].setry:=j;
     inc(i); dec(j);
   end
 until i>j;
 if p<j then qsorty(p,j);
 if i<q then qsorty(i,q)
end;
procedure natoc(p:word);        {natočí robota směrem k p-tému bodu}
var uhel,xnov,ynov:real;
    k:word;
begin
  if x[p]=0 then if y[p]<0 then uhel:=3/2*pi else uhel:=pi/2 else
    if x[p]>0 then uhel:=arctan(y[p]/x[p]) else uhel:=arctan(y[p]/x[p])+pi;
  {spočteme úhel, o který budeme otáčet}
  for k:=1 to n do               {A otočíme body}
  begin
    xnov:=x[k]*cos(uhel)-y[k]*sin(uhel);
    ynov:=y[k]*cos(uhel)+x[k]*sin(uhel);
    x[k]:=xnov;
    y[k]:=ynov
  end;
  for k:=1 to n do               {Nainicializujeme pole}
  begin
    body[k].setrx:=k; body[k].setry:=k;
    setridx[k]:=k; setridy[k]:=k;
  end;
  qsortx(1,n);                   {Setřídíme dle x-ové souřadnice}
  qsorty(1,n);                   {Setřídíme dle y-ové souřadnice}
end;
function soused(p:word):word;
begin
  if (p and 1)=1 then soused:=p+1 else soused:=p-1
end;
function nesoused(p:word):word;
begin
  if (p and 1)=1 then nesoused:=p-1 else nesoused:=p+1
end;
{Vytvoříme plán průchodu robota, přičemž právě vycházíme
 z p-tého bodu, je-li q true vodorovně, je-li false svisle}
procedure vytvorplan(p:word;q:boolean);
var dalsi,okoli:word;
begin
  body[p].navst:=true; inc(navstivil);
  if q then                      {další bod ve stejném řádku/sloupci}
    dalsi:=setridx[soused(body[p].setrx)]
  else
    dalsi:=setridy[soused(body[p].setry)];
  body[p].nasl:=dalsi;
  if not(body[dalsi].navst) then {dokud nejsi na začátku prodlužuj plán}
    vytvorplan(dalsi,not(q));
  if (p>n0) and q then exit;     {jsme na konci}
```

```
  if q then                      {zdalipak je tu ještě nenavštívený bod?}
    okoli:=setridx[nesoused(body[dalsi].setrx)]
  else
    okoli:=setridy[nesoused(body[dalsi].setry)];
  if (okoli>=1) and (okoli<=n) and
     (q or (abs(y[okoli]-y[p])<chyba)) and
     (not(q) or (abs(x[okoli]-x[p])<chyba)) and
      not(body[okoli].navst) then
  begin                          {pokud ano, tak ho zapojíme}
    vytvorplan(okoli,not(q));
    body[soused(okoli)].nasl:=body[p].nasl;
    body[p].nasl:=okoli;
  end;
  if q then                      {zdalipak je tu ještě nenavštívený bod?}
    okoli:=setridx[nesoused(body[p].setrx)]
  else
    okoli:=setridy[nesoused(body[p].setry)];
  if (okoli>=1) and (okoli<=n) and
     (q or (abs(y[okoli]-y[p])<chyba)) and
     (not(q) or (abs(x[okoli]-x[p])<chyba)) and
      not(body[okoli].navst) then
  begin                          {pokud ano, tak ho zapojíme}
    vytvorplan(okoli,not(q));
    body[soused(okoli)].nasl:=body[p].nasl;
    body[p].nasl:=okoli;
  end;
end;
function zparovano:boolean;      {kontrola, zda v řádcích/sloupcích}
var i:word;                      {je sudý počet bodů}
begin
  zparovano:=false;
  for i:=1 to n div 2 do
    if abs(x[setridx[2*i]]-x[setridx[2*i-1]])>chyba then exit;
  for i:=1 to n div 2 do
    if abs(y[setridy[2*i]]-y[setridy[2*i-1]])>chyba then exit;
  zparovano:=true
end;
function vzdal(p,q:word):real;  {vzdálenost p-tého a q-tého bodu}
begin
  vzdal:=sqrt(sqr(x[p]-x[q])+sqr(y[p]-y[q]))
end;
{vektorový součin vektoru z p-tého do q-tého a z q-tého do r-tého bodu}
function vektsoucin(p,q,r:word):real;
begin
  vektsoucin:=(x[q]-x[p])*(y[r]-y[q])-(x[r]-x[q])*(y[q]-y[p])
end;
begin
  readln(n0);                    {načtení vstupu}
  readln(mx,my);
  for i:=1 to n0 do
  begin
    readln(x[i],y[i]);
    x[i]:=x[i]-mx;               {bod M posuneme do počátku}
    y[i]:=y[i]-my;
  end;
  n:=n0+1;                       {z bodu M vytvoříme "zadaný" bod}
  x[n]:=0; y[n]:=0;
  if (n mod 2=1) then
```

```
  begin
    inc(n); x[n]:=0; y[n]:=0;
  end;
  setridx[0]:=0; setridy[0]:=0;  {nastavíme zarážky}
  setridx[n+1]:=0; setridy[n+1]:=0;
  for i:=1 to n0 do
  begin
    natoc(i);                    {natočíme si body}
    for j:=1 to n do body[j].navst:=false;
    navstivil:=0;
    if zparovano then            {v řádcích a sloupcích je sudý počet bodů}
    begin
      if n-n0=2 then             {dva přidané body umístíme vedle sebe}
      begin
        setridx[body[n0+2].setrx]:=setridx[soused(body[n0+1].setrx)];
        body[setridx[soused(body[n0+1].setrx)]].setrx:=body[n0+2].setrx;
        setridx[soused(body[n0+1].setrx)]:=n0+2;
        body[n0+2].setrx:=soused(body[n0+1].setrx);
        setridy[body[n0+2].setry]:=setridy[soused(body[n0+1].setry)];
        body[setridy[soused(body[n0+1].setry)]].setry:=body[n0+2].setry;
        setridy[soused(body[n0+1].setry)]:=n0+2;
        body[n0+2].setry:=soused(body[n0+1].setry);
      end;
      vytvorplan(n0+1,false)
    end;
    if navstivil=n then          {prošli jsme všemi body?}
    begin
      j:=n0+1;                   {začneme v bodě M}
      writeln('Robot je natočen k ',body[j].nasl,'-tému bodu.');
      writeln('Program pro robota:');
      write('G(',vzdal(j,body[j].nasl):0:2,')');
      while(body[j].nasl<=n0) do  {a skončíme v bodě M}
      begin
        if vektsoucin(j,body[j].nasl,body[body[j].nasl].nasl)<0 then
          write(',R')
        else
          write(',L');
        j:=body[j].nasl;
        write(',G(',vzdal(j,body[j].nasl):0:2,')');
      end;
      writeln;                   {odřádkujeme a skončíme}
      exit
    end
  end;
  writeln('Program pro robota nelze sestavit.');
end.
```

11-2-5 Mnoho CPŮ, složitosti smrt — Martin Mareš

1. Hledání minima

Tuto úlohu je možno vyřešit použitím metody Rozděl a panuj, to jest napsáním rekursivního paralelního programu. Myšlenka je velice jednoduchá: minimum v poli najdeme tak, že si pole rozdělíme na poloviny, v každé z nich rekursivním aplikováním téhož algoritmu (ovšem pro obě poloviny najednou!) nalezneme minimum a poté za globální minimum prohlásíme menší z minim částí.

Program zapsaný v paralelním Céčku jako funkce, která dostane pole, levou a pravou hranici úseku v tomto poli a spočte minimum tohoto úseku, vypadá takto:

```
int min(int x[], int l, int r)
{
  int a, b, m;
  if (l == r)     /* jednoprvkový úsek */
    return x[l];
  m = (l+r)/2;    /* střed úseku */
  parallel {  /* obě poloviny najednou */
    a = min(x, l, m);
    b = min(x, m+1, r);
  }
  return (a < b) ? a : b;
}
```

Časovou složitost algoritmu rozebereme pro jednoduchost pouze v případě, že n je mocninou dvojky (jinak bychom museli zohlednit to, že obě „poloviny“ nejsou stejně velké; uvědomte si, že to není na újmu obecnosti, protože pro libovolné jiné n jistě algoritmus vykoná méně práce než pro nejbližší vyšší mocninu dvou). Pro úsek délky jedna provede algoritmus konstantní počet operací, tedy $t(1) = c$, pro libovolný větší pak provede paralelně dva (stejně dlouhé) výpočty pro poloviny pole plus nějakou konstantní režii navíc: $t(n) = t(n/2) + C$, tedy také

$$t(n) = t(n/4) + 2C = t(n/8) + 3C = \ldots = t(n/2^k) + kC.$$

Pro $k = \log_2 n$ je tedy $t(n/2^k) = t(1) = c$, proto

$$t(n) = c + \log_2 n \cdot C = O(\log n).$$

Paměťovou složitost spočítáme obdobně: pro jednotkovou délku úseku je konstantní, $s(1) = s$, pro delší úsek potřebujeme pouze místo na zásobníku pro

vyvolání funkcí plus místo zabrané každou z nich (tentokráte ovšem dohromady – použitá paměť není paralelním zpracováním nijak ovlivněna):

$$\begin{aligned} s(n) &= 2s(n/2) + S = 2(2s(n/4) + S) + S = \\ &= 4s(n/4) + 3S = \ldots = 2^k s(n/2^k) + (2^k - 1)S, \end{aligned}$$

pro $k = \log_2 n$ dostaneme

$$s(n) = 2^k + (2^k - 1)S = n + (n-1)S = O(n).$$

2. Převod paralelní → sekvenční výpočet

Jelikož definice paralelního výpočtu nepovoluje vzájemné ovlivňování jednotlivých větví výpočtu, musí být výsledek nutně týž, jako když se provádí jedna po druhé. Proto nahradíme-li všechny bloky `parallel` bloky normálními, získáme ekvivalentní (i když mnohem pomalejší [až tolikrát, kolik bylo původních větví výpočtu, což může být až exponenciálně mnoho]) sekvenční program. [Analogicky bychom takto mohli z nedeterministického paralelního programu zkonstruovat nedeterministický sekvenční program, ale to po nás nikdo nechtěl.]

3. Převod nedeterministický → paralelní výpočet

Nejprve je třeba uvědomit si, jaké problémy vlastně při tomto převodu mohou nastat:

1. Časová složitost nedeterministického algoritmu je dána nejkratší úspěšnou větví, případně větví nejdelší, pokud žádná větev není úspěšná. To tedy také znamená, že v úspěšně dokončeném výpočtu mohly existovat nekonečné větve. Tohoto problému se zbavíme tak, že stejně jako v řešení úlohy 11-1-4 přídáme počítadlo omezující délku výpočtu a budeme úlohu opakovaně řešit pro stále rostoucí omezení – tím sice vše kvadraticky zpomalíme, ale polynom na druhou je opět polynom, pročež to nevadí.
2. Funkce `oracle` mohla za parametr dostat libovolně velké číslo. Ano, to je chyba v zadání, argumenty této funkce měly být také omezené nějakým polynomem, takže předpokládejme, že opravdu jsou. Navíc vzhledem k tomu, že libovolné polynomiálně velké číslo má polynomem omezený počet bitů, můžeme po vzoru minulé úlohy generovat odpovědi orákula po jednotlivých bitech.

Po vypořádání se s těmito problémy můžeme postupovat velice jednoduše: při každém volání orákula prostě spustíme paralelně dvě větve výpočtu pro obě možné odpovědi a vrátíme `true`, pokud alespoň jedna z nich vrátila `true`.

[Analogie s prohledáváním grafu z minulé úlohy opět platí, pouze místo prohledávání do šířky použijeme rekursivní prohledávání do hloubky omezené maximální hloubkou, v němž budeme rekursivní volání paralelizovat stejně jako při hledání maxima v poli.]

11-3-1 Telefouňové — Martin 'Mafi' Bělocký

Téměř všichni řešitelé při sestavování algoritmu zapomněli na věc naprosto základní – důkaz správnosti algoritmu. Také když je v zadání příklad výstupu, je dobré tento formát zachovat. Většina řešitelů nepsala na výstup názvy měst.

Úkolem bylo, vypsat libovolný graf se zadaným skóre, což je problém z teorie grafů. Pro podrobnější prostudování doporučujeme např. Úvod do teorie grafů od Jiřího Sedláčka, vydavatelství Academia. Pro neznalé začneme definicemi: (Neorientovaným) grafem $G(V, H)$ nazveme uspořádanou dvojici množiny vrcholů V a množiny hran H. Hrana je (neuspořádaná) dvojice vrcholů z množiny vrcholů grafu. Stupeň vrcholu X je číslo, jehož hodnota udává počet hran vycházejících z X. Posloupnost stupňů všech vrcholů grafu se nazývá skóre grafu. O posloupnosti kladných celých čísel $P = a_1, a_2, \ldots, a_n$ řekneme, že je grafová, pokud existuje alespoň jeden graf mající skóre P.

Na vstupu byly názvy měst (vrcholy grafu) a počty spojení z měst (stupně vrcholů). Stačilo tedy pouze zjistit, zda je vstupní posloupnost stupňů vrcholů grafová. To však některým řešitelům činilo potíže. Uváděli obvykle nepostačující podmínky pro existenci grafu jako např. součet stupňů vrcholů musí být sudé číslo, největší stupeň nesmí být větší než je počet vrcholů mínus jedna. Jak *spolehlivě* zjistit, zda je posloupnost grafová, říká přesně následující tvrzení.

Havlova věta: Posloupnost celých čísel

$$S_1 = s_1, s_2, \ldots, s_n,$$

kde $n - 1 \geq s_1 \geq s_2 \geq \ldots \geq s_n \geq 1, n \geq 2$ je grafová, právě když je grafová posloupnost $S_2 = s_2 - 1, s_3 - 1, \ldots, s_{s_1+1} - 1, s_{s_1+2}, s_{s_1+3}, \ldots, s_n$.

Důkaz:

i) Je-li posloupnost S_2 grafová, existuje graf G_2 na $n-1$ vrcholech $u_2, u_3, \ldots, u_n$ takový, že stupeň vrcholu u_i je $s_i - 1$ pro $2 \leq i \leq s_1 + 1$ a s_i pro $s_1 + 2 \leq i \leq n$. Potom můžeme sestrojit nový graf G_1 tím, že připojíme nový uzel u_1 a nové hrany $\{u_1, u_i\}$ (pro $2 \leq i \leq s_1 + 1$). Graf G_1 ukazuje, že též posloupnost S_1 je grafová.

ii) Nechť nyní S_1 je grafová posloupnost. Dokážeme, že existuje graf G_1 se skóre S_1 (stupeň u_1 je s_1, stupeň u_2 je $s_2 \ldots$) takový, že vrchol u_1 je spojen právě s vrcholy $u_2, u_3, \ldots, u_{s_1+1}$. Potom je S_2 zřejmě skórem grafu $G_1 - \{u_1\}$, tj. grafu, ze kterého odebereme vrchol u_1 a všechny hrany, které z něj vedou.

Pro spor předpokládejme, že G_1 neexistuje. Vezměme tedy graf G takový, že vrchol u_1 není spojen s vrcholem u_i a i je maximální (zřejmě $i \leq s_1 + 1$). Zřejmě musí existovat vrchol u_j, $j > s_1 + 1$

spojený s u_1 hranou. Protože $s_j \leq s_i$, musí existovat vrchol u_k spojený s u_i a ne s u_j. Když nyní spočteme skóre grafu $G' = G - \{u_1, u_j\} - \{u_i, u_k\} + \{u_1, u_i\} + \{u_j, u_k\}$, zjistíme, že je to S_1. Potom ale G nemohl být graf s maximálním i, protože pro graf G' je i větší. Spor.

Náš algoritmus vychází přímo z Havlovy věty. Opakovaným použitím věty se zachovává invariant – posloupnost je stále grafová a přitom se při každém použití věty zmenší počet členů posloupnosti stupňů alespoň o jedna. Tedy máli posloupnost na počátku N členů, pak algoritmus skončí nejpozději po $N-1$ použitích věty. Náš algoritmus je tedy konečný a korektní. Zde jsou jeho kroky podrobněji:

1. Načti posloupnost stupňů (alespoň 2 členy).
2. Odstraň nulové členy z posloupnosti a posloupnost sestupně setříď.
3. Pokud nemá posloupnost ani jeden člen, skonči.
4. Je-li první člen posloupnosti s_1 větší než počet členů posloupnosti minus jedna, skonči a oznam podvod. Jinak od následujících s_1 členů odečti jedna a při každém odečítání vypiš spojení: název města odpovídajícího vrcholu se stupněm $s_1 \longrightarrow$ název města odpovídajícího vrcholu, jehož stupeň snižujeme o 1. Nakonec odstraň 1. člen posloupnosti.
5. pokračuj krokem 2

První třídění provedeme algoritmem quicksort, se složitostí $O(N \cdot \log N)$, kde N je počet měst. Další třídění provádíme rychleji takto: Během kroku 4 se odečítáním jedničky od setříděných členů posloupnosti zachovává setřídění, problém může nastat jen na konci bloku, kde jsme odečítali jedničku. Je-li posloupnost např. $S_3 = 5, 4, 4, 3, 3, 3, 3, 3, \ldots$ mohla nastat nejhůř situace $S_4 = 3, 3, 2, 2, 2, 3, 3, \ldots$ Pokud odečítání jedničky skončilo uprostřed úseku stejných hodnot, dojde k situaci S_4, kdy není posloupnost setříděná. Stačí ale nalézt okraje úseku s původně stejnými hodnotami a celý úsek převrátit. Tím dostáváme opět setříděnou posloupnost. Okraje původního úseku nalezneme po odečtení jedniček snadno. Pravý okraj je nejpravější člen od posledního zmenšovaného členu, který má hodnotu o jedna větší než poslední zmenšovaný člen. Obdobně levý okraj úseku.

Převracený úsek může mít maximálně délku N. Tedy setřídění provedeme v čase $O(N)$. V kroku 4 algoritmu můžeme odečítat maximálně ode všech zbývajících členů. Složitost tohoto kroku je tedy také $O(N)$. Kroky 2 a 4 se opakují maximálně $N-1$ krát, což jsme zdůvodnili výše. Celková časová složitost algoritmu je tedy $O(N^2)$. Pokud algoritmus přímo přepíšeme do programu, nedostaneme ovšem přesný výstup podle příkladu, neboť algoritmus vypisuje

průběžně spojení mezi městy a když zjistí podvod, vypíše to. Zadání však vyžaduje vypsat buď obvinění z podvodu nebo seznam spojení mezi městy. To v programu vyřešíme např. tak, že celý postup necháme proběhnout dvakrát, což na asymptotické složitosti nic nemění. Jednou bez vypisování spojení a pokud se nevypíše „podvod!“, pak celý postup zopakujeme, ale s vypisováním spojení, neboť už je jasné, že telefonní síť existuje. Tato úprava nás bude stát dvojnásobné množství paměti na vstupní data. Celková paměťová náročnost je však stále lineární.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100

typedef struct {
    int stupen; char *jmeno;
} uzel;
uzel score[MAX], score1[MAX];        /* 2x pole pro uložení vstupu */
int n;

int cmp (const void *a,  const void *b)   /* porovnání pro quicksort */
{
    return  ( ( (uzel*)b)->stupen -  ( (uzel*)a)->stupen);
}

int test (uzel* pole,  int t)              /* t=indikátor výpisu */
{
    int i,  l,  stupen,  left,  right;
    uzel *k,  tmp;

    /* pro každý vrchol setříděné posloupnosti */
    for  (i=0; i < n; i++)
    {
       stupen= (pole+i)->stupen;       /* největší stupeň */
       if  (!stupen)
          return 1;                    /* graf v pořádku */
       else {
          k=pole+i;
          for  (l=1; l<=stupen; l++) {   /* od následující "stupen" vrcholů odečti hranu */
             if  (-- ( (k+l)->stupen) < 0) return 0;    /* podvod - došly vrcholy */
             if  (t) printf ("spojeni %s - %s\n", k->jmeno,   (k+l)->jmeno);
          }
          k=pole+i+stupen; left=right=0;    /* teď dotřídíme stupně */
          /* najdeme okraje */
          while ( (k+right != pole+n-1) && ( (k->stupen) == (k+right+1)->stupen-1))
             right++;
          while (k->stupen ==  (k+left-1)->stupen)
             left--;
          /* převrátíme blok */
          for  (; left < right; left++,  right--) {
              tmp.stupen= (k+left)->stupen;
              tmp.jmeno= (k+left)->jmeno;
```

```
                (k+left)->stupen= (k+right)->stupen;
                (k+left)->jmeno= (k+right)->jmeno;
                (k+right)->stupen=tmp.stupen;
                (k+right)->jmeno=tmp.jmeno;
            }
             (pole+i)->stupen=0;          /* zrušíme vrchol */
        }
    }
    return 1;
}
int main (void)
{
    char jmeno[100];
    int pom=0, konec=0, i;
    scanf ("%d\n", &n);
    for (i=0; i<n; i++) {
        scanf ("%s %d", jmeno, &score[i].stupen);
        score[i].jmeno= (char*)malloc (strlen (jmeno)+1);
        strcpy (score[i].jmeno, jmeno);
        pom += score[i].stupen;          /* součet stupňů musí být sudý */
        if (score[i].stupen >= n) konec=1;    /* stupeň nesmí být větší */
    }
    if ( (pom&1) || konec)               /* už teď je jasný podvod */
    {
        puts ("Podvodnici!");
        return 0;
    }
    qsort (score, n, sizeof (uzel), cmp);
    for (i=0; i<n; i++) score1[i]=score[i];
    if (test (score1, 0))
        test (score, 1);
    else
        puts ("Podvodnici!");
    return 0;
}
```

11-3-2 Ffffaktoriál — Dan Kráľ

Nejprve se zkusme nad zadanou úlohou trochu zamyslet. Nechť n je dáno. Počet koncových nul čísla $n!$ je roven největšímu k_{10} takovému, že $10^{k_{10}} \mid n!$, což je rovno $\min\{k_2, k_5\}$, kde k_2 je největší číslo takové, že $2^{k_2} \mid n!$, analogicky je definováno k_5. Tato čísla tedy udávají počet dvojek resp. pětek v prvočíselném rozkladu $n!$. Protože dvojka i pětka jsou prvočísla, tak platí:

$$k_2 = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \cdots = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{2^i} \right\rfloor$$

$$k_5 = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{5^2} \right\rfloor + \left\lfloor \frac{n}{5^3} \right\rfloor + \cdots = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{5^i} \right\rfloor$$

Správnost výše uvedených vztahů je zřejmá, neboť $\lfloor \frac{n}{2} \rfloor$ udává počet čísel menších nebo rovných n dělitelných dvěma, $\lfloor \frac{n}{2^2} \rfloor$ je počet čísel dělitelných 2^2 atd. Čísla dělitelná právě dvěma přispějí do prvočíselného rozkladu n jednou dvojkou, čísla dělitelá čtyřmi ještě jednou dvojkou (kromě té za dělitelnost dvěma) atd. Analogickou úvahu lze provést pro dělitelnost pěti.

Idea našeho algoritmu bude následující: Ze všech čísel dělitelných pěti, tyto pětky vytkneme, a za každou z nich, vytkneme z nějakého jiného čísla dvojku – tak obdržíme součin nějakých nových čísel a_i. Tedy $n!$ převedeme do tvaru $10^m \Pi_{i=1}^{n} a_i$, kde žádné z čísel a_i není dělitelné pětkou a tedy poslední číslice jejich součinu nemůže být nula. Jinými slovy: Poslední číslice jejich součinu je poslední nenulová číslice $n!$. Výše uvedený součin budeme počítat v několika fázích – nejprve vypočteme součin těch a_i, která vznikla z čísel nedělitelných pětkou, potom těch, která vznikla z čísel dělitelných právě pětkou, potom z čísel dělených právě dvaceti pěti atd. Kromě toho v první fázi některá z čísel podělíme dvojkami za čísla dělitelná jednou pětkou, v druhé fázi dělíme za čísla dělitelná dvaceti pěti atd. Dvojkou nebudeme dělit „náhodně vybraná“ čísla, ale za čísla, která končila pětkou, podělíme dvojkou čísla končící dvojkou (vznikne jako poslední číslice jednička nebo šestka), a za čísla, která končila nulou, podělíme dvojkou, čísla končící šestkou (vznikne jako poslední číslice trojka nebo osmička). Tímto přístupem si zajistíme, že v každé fázi budeme schopni v konstantním čase zjistit počet násobených čísel, která končí cifrou jedna, dvě, tři, čtyři, šest, sedm, osm nebo devět. Poslední číslici mohu nyní snadno spočítat modulo 10, což s pozorováním $i = i^5 \pmod{10}$ pro uvažovaná i zvládnu v konstantním čase. Každá z $O(\log n)$ fází bude trvat konstantní čas a celkový čas spotřebovaný naším algoritmem tedy bude $O(\log n)$.

Nyní se na chvíli ještě zamysleme nad následující úvahou (často vídanou v řešeních): Poslední nenulovou číslici mohu určit tak, že spočítám součin všech posledních nenulových číslic čísel 1 až n různých od pětek a potom za každou poslední číslici rovnou pěti odeberu z tohoto součinu jednu dvojku (tím vyeliminuji dvojky „spotřebované“ na koncové nuly) — což udělám např. tak, že z číslic osm v tomto součinu udělám čtyřky. Tato úvaha je však chybná! Neboť $28/2$ je sice 14, ale $38/2 = 19$ a tuto osmičku jsme tedy v žádném případě nahradit čtyřkou nemohli. Proto jsme také v našem řešení museli z některých dvojek dělat jedničky a z jiných šestky.

```
#include<stdio.h>
int n, i, k, l;
int cisl[10]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};          /* Počty jednotlivých koncových cifer */
int posl[10][4]={{0, 0, 0, 0}, {0, 0, 0, 0}, {6, 2, 4, 8}, {1, 3, 9, 7}, {6, 4, 6, 4},
                 {5, 5, 5, 5}, {6, 6, 6, 6}, {1, 7, 9, 3}, {6, 8, 4, 2}, {1, 9, 1, 9}};
```

```
/* Pro přehlednost indexace nejsou některé položky polí používány */
int main (void)
{
  scanf ("%d", &n);
  for (; n; n /= 5)
  {
    k = (n + 9) / 10;                    /* Četnost výskytu jednotlivých cifer */
    l = n % 10;
    for (i = 1; i <= 10 ; i++)           /* Pro všechna zakončení... */
    {
      switch (i)
      {
        case 5:
          /* Odebereme potřebné dvojky k pětkám */
          cisl[2] -= k;
          cisl[1] += (k+1)/2;
          cisl[6] += k/2;
          break;
        case 10:
          /* Odebereme potřebné dvojky k pětkám */
          cisl[6] -= k;
          cisl[3] += (k+1)/2;
          cisl[8] += k/2;
          break;
        default:
          cisl[i] += k;
      }
      if (l == i) k--;
    }
  }
  /* Nakonec si spočteme výsledný součin */
  k = 1;
  for (i = 2; i <= 9; i++)
    if (i != 5 && cisl[i])
      k *= posl[i][cisl[i]%4];
  printf ("%d\n", k%10);
  return 0;
}
```

11-3-3 Obdélníčky — Michal Píše

Algoritmus pro řešení této úlohy pracuje následujícím způsobem: Do pomocného pole P si uloží všechny hodnoty x-ových souřadnic, na kterých nějaký čtverec začíná nebo končí. Toto pole následně setřídí. Je vidět, že pás mezi přímkami $x = P[i]$ a $x = P[i+1]$ už neobsahuje žádnou svislou hranu, takže si náš stůl rozdělíme na takovéto pásy a spočítáme obsah pokryté části v každém pásu.

Budeme potřebovat pole čtverců, setříděné podle dolní y-ové souřadnice. V tomto poli postupujeme po jednom čtverci a u každého čtverce nejprve

zkontrolujeme, zda má s daným pásem neprázdný průnik (tedy $x_1 \leq px_1$ a $x_2 \geq px_2$, kde x_1 a x_2 jsou x-ové souřadnice čtverce a px_1 a px_2 hranice pásu). Dále otestujeme, zda čtverec není úplně obsažen v předchozím ($y_1 > maxy$, kde y_1 je horní y-ová souřadnice čtverce a $maxy$ maximální y-ová souřadnice z předchozích čtverců). Pokud není, připočteme jeho výšku k celkové výšce čtverců v tomto pásu a odečteme výšku průniku s předchozím čtvercem ($v = v + y_1 - y_2 - \max(maxy - y_2, 0)$, maximum zajišťuje, že nám průnik nevyjde záporný). Pak jen stačí celkovou výšku vynásobit šířkou pásu a přičíst k prozatimnímu obsahu. Toto se opakuje pro všechny pásy.

Časová složitost: vytvoření pomocného pole P spotřebuje $O(N)$ (každý čtverec přidá maximálně dva pásy), jeho setřídění $O(N \log N)$, setřídění čtverců podle dolní y-ové souřadnice $O(N \log N)$, spočítání celkové výšky čtverců v pásu $O(N)$. Celkem tedy $O(N^2)$.

Paměťová složitost: několik polí o délce $O(N)$ a pár pomocných proměnných. Celkem $O(N)$.

V popisu programu jsou vynechána některá triviální místa. Jejich implementaci byste jistě zvládli sami.

[Pozn.: Existuje i řešení v čase $O(N \log N)$, ale je poměrně složité. Idea: Opět si rovinu rozřežeme na svislé pásy, ale jejich zaplnění počítáme efektivněji. Na počátku si setřídíme všechny hodnoty y-ových souřadnic vrcholů obdélníků a sestavíme vyvážený binární strom, jehož listy budou odpovídat intervalům na y-ové ose, v nichž žádný obdélník nezačíná ani nekončí, a pro každý z těchto intervalů si budeme průběžně udržovat, kolik obdélníků v daném pásu tento interval pokrývá. Obsah průniku všech obdélníků v pásu je pak roven celkové délce všech pokrytých intervalů vynásobené šířkou pásu. Stačí pouze dořešit, jak do tohoto stromu vkládat a odstraňovat intervaly a aktualizovat si současně celkovou délku intervalů, to vše v čase $O(\log N)\ldots$ –M.M.]

```
{ Načtení n čtverců do pole c vynecháno. }
for i:=1 to n do begin
                    P[2*i-1]:=c[i].xdolni;
                    P[2*i]:=c[i].xhorni;
                  end;
{ Setřídění pole P vynecháno, hodnoty se třídí bez ohledu na to, zda jde o
horní či dolní hranici. Při takovémto třídění mohou vzniknout i pásy nulové
šířky, na správnosti algoritmu to ale nic nemění. }
{ Setřídění pole c podle dolní y-ové hranice vynecháno. }

obsah:=0;
for i:=1 to 2*n-1 do
begin
  vyska:=0;
  ypredchozi:=c[1].ydolni;
  for j:=1 to n do
    if (c[j].xleve <= P[i]) and (c[j].xprave >= P[i+1]) and
       (ypredchozi < c[j].yhorni) then
```

```
    begin
      vyska:=vyska+c[j].yhorni-c[j].ydolni - max(0,ypredchozi-c[j].ydolni);
      ypredchozi:=c[j].yhorni;
    end;
  obsah:=obsah+vyska*(P[i+1]-P[i]);
end;

{ V proměnné obsah je nyní obsah zaplněné plochy }
```

11-3-4 Machina Universalis — Martin Mareš

Nejdříve doplníme „bílá místa v zadání“:

3. Převod všech Pascalských datových typů na wordy:
   - Celočíselné typy bez znaménka můžeme přímo nahradit wordem.
   - Celočíselné typy se znaménkem nahradíme recordem, jehož první prvek bude absolutní hodnota a druhý znaménko.
   - Ostatní ordinální typy nahradíme ordinálními čísly.
   - Typ *real* můžeme ukládat jako record obsahující mantisu a exponent, případně čitatele a jmenovatele zlomku.
   - Record obsahující wordy (speciálně tedy cokoliv, co umíme zakódovat do wordu) nahradíme polem wordů.
   - Pole wordů indexované wordem (tedy vlastně pole čehokoliv indexované čímkoliv, za předpokladu že jsme všechno schopni wordem popsat) je jediné, co vám působilo větší problémy. Ukážeme si jeden z mnoha možných přístupů, který využívá toho, že každé kladné celé číslo má jednoznačný rozklad na prvočinitele.
     Buď $p_1, p_2, \ldots$ rostoucí posloupnost všech prvočísel (jistě si dovedete představit funkci $p(n)$, která spočte n-té prvočíslo). Pak pole hodnot
     $$x_1, x_2, \ldots, x_n$$
     zakódujeme číslem
     $$p_1^{x_1} \cdot p_2^{x_2} \cdot \ldots \cdot p_n^{x_n}.$$
     Indexovat takto reprezentované pole je velice jednoduché: spočteme si prvočíslo odpovídající indexu a zjistíme, kolikrát je kód pole možno beze zbytku tímto prvočíslem vydělit. Změna jedné hodnoty v poli je také snadná: opět nalezneme příslušné prvočíslo, dělíme jím kód, dokud to jde (tím jsme x_i vynulovali) a poté násobíme tolikrát, na kolik chceme x_i nastavit.
   - Dynamické proměnné a pointery: celou paměť lze reprezentovat jako
     `memory : array [word] of word`,

čímž je problém vyřešen.

7. Rozklad cyklů a podmínek na podmíněné skoky:
   - `if A then B else C`
     ⟶ `if A then goto 1; C; goto 2; 1: B; 2:`
   - `while A do B`
     ⟶ `goto 1; 2: B; 1: if A then goto 2;`
   - `repeat A until B`
     ⟶ `1: A; if not B then goto 1;`
   - Cyklus `for` snadno převedeme na `while`.

10. Převod triviálních operací:
    - Sčítání: viz příklad v zadání.
    - Odčítání: analogicky, pouze `INC 0` na předposledním řádku nahradíme instrukcí `DEC 0`. Navíc pro $x < y$ vyjde $x - y = 0$, což se nám bude hodit.
    - Skok podmíněný rovností: přímo instrukcí `JEQ`.
    - Skok podmíněný nerovností $x \leq y$: testujeme $x - y = 0$ (ostatní nerovnosti analogicky, případně negací již odvozené podmínky).
    - Ještě potřebujeme přiřazení, ale to vyřešíme jednoduchou úpravou sčítacího podprogramu: budeme najednou inkrementovat dvě paměťové buňky, čímž si výsledek „rozmnožíme“ a potom pro přiřazení `x:=y` provedeme `v:=y+0` (tím jsme přišli o obsah `y`) a `(x,y):=v+0` (opraveno a zkopírováno).

Nyní již stačí odvodit universální programy a budeme hotovi.

Sestrojíme program P v Pascalu, který bude interpretovat program v mikroassembleru zadaný jako pole čtveřic čísel (kódů instrukcí), samozřejmě reprezentované jedním wordem. Tento program je natolik přímočarý, že pokládáme za zhola zbytečné jej zde uvádět. Nyní spojíme-li v zadání popsaný překladač z μ_1 do Pascalu s tímto programem, vznikne nám universální program pro Pascal a naopak přeložíme-li program P tímto překladačem, vznikne universální program pro μ_1. Toť vše.

Zvídavého čtenáře samozřejmě dříve či později napadne záludná otázka „a k čemu je tohle všechno dobré?“

To, že Pascal je výpočetně stejně silný jako Mikroassembler (to znamená, že se v obou těchto výpočetních modelech dá spočítat to samé) a že jsme též dokázali, že totéž platí o Pascalu a nedeterministickém Pascalu (viz druhá série), naznačuje, že i na první pohled velice odlišné programovací prostředky umí vlastně to samé. Tak řečená *Churchova teze* říká, že toto platí pro *všechny* rozumné výpočetní modely (bohužel ji není možno dokázat, protože není jasné, co vlastně znamená formulace „rozumný výpočetní model“).

Existence universálního programu nám pro změnu naznačuje, že s programy je možné nakládat jako s přirozenými čísly, kterým právě universální program přiřazuje význam. Na universální program se tak můžeme dívat jako na nějakou funkci $U(p, x)$, která má za parametry číslo programu p, který má interpretovat a vstup x pro tento program a platí $U(p, x) = p(x)$ [přičemž za jednu z možných hodnot výstupu považujeme také stav „nedoběhne“].

Z tohoto pozorování můžeme vyvodit spoustu užitečných věcí, například dokázat, že neexistuje žádný algoritmus rozhodující *Halting Problem* (to jest zjišťující, zda se daný program pro daný vstup zastaví či nikoliv). Předpokládejme pro spor, že takový algoritmus existuje – mějme tedy funkci $H(p, x)$, která vrací jedničku v případě, že se $p(x)$ zacyklí a nulu, pokud se zastaví. Potom také můžeme sestrojit program $R(x)$ následujícího znění:

$$R(x) = \begin{cases} 0 & \text{pokud } H(x, x) = 1 \\ \text{zacyklí se} & \text{pokud } H(x, x) = 0. \end{cases}$$

Nyní si ovšem položme otázku, jak odpoví R na svůj vlastní kód, tedy kolik je $R(R)$. Pokud se $R(R)$ zastaví, znamená to, že $H(R, R) = 1$, tudíž že se $R(R)$ dle funkce H zastavit neměl. Naopak, pokud se nezastaví, bylo $H(R, R) = 0$, tedy se dle H zastavit měl. Takže se nemůže ani zastavit, ani nezastavit, což je spor. Tudíž žádný program H s touto vlastností neexistuje.

11-3-5 Pascal nebo C? — Robert Špalek

Většina z řešitelů správně pochopila, že ve zdrojáku je nutno nakombinovat vhodně komentáře, a tak jediným problémem zůstalo zahájení programu. Způsobů je hned několik:

1) Nejobvyklejší bylo klíčové slovo `const`. Pak je možno navázat např. `struct {`, které v Pascalu deklarovalo konstantu `struct` a otevíralo komentář, zatímco v C se tímto pouze deklaroval záznam. Další pokračování je zřejmé z přiloženého programu. Jinou možností je `enum {`. *(programy 1, 2)*
2) Za slovem `const` může následovat i hodnota konstanty `x={1`. Jazyk C toto povoluje. *(3)*
3) Jeden z nejkratších způsobů je `(*x);`. Toto v Pascalu otevírá komentář, zatímco v C deklaruje proměnnou typu ukazatel na int, nicméně zastaralým způsobem podle originální normy jazyka C; novější standardy to již jen mlčky připouštějí a návrh standardu `C9x` dokonce zakazuje. Tomu se dá samozřejmě předejít uvedením oblíbeného `const int (*x);`. *(4, 5)*
4) Existují i další neportabilní řešení, např. využívající proměnné `randseed` v Borland Pascalu. Toto začíná `begin(randseed){`.

V C takto začíná deklarace funkce, v Pascalu již samotný program. *(6)*

Většina vašich řešení nechala ignorovat text za závěrečným Pascalovským `end`. Standard toto myslím předepisuje, nicméně některé kompilátory (např. `gpc`) se podle toho neřídí. Čistší je zbylý text v Pascalu zakomentovat.

Další elegantní řešení má úloha zabývající se rozlišením `C` a `C++`. *Marek Sterzik* využil toho, že překladače `C` a `C++` se chovají jinak při práci s pojmenovanými strukturami (viz zdrojový text na konci řešení).

Výraz `sizeof(struct a)` v každém případě vrátí velikost struktury. Ale výraz `sizeof(a)` pochopí každý překladač jinak. Překladač `C` vypočítá velikost proměnné, zatímco překladač `C++` si automaticky doplní klíčové slovo `struct` a zabývá se opět strukturou.

Dále se dá snadno rozlišit, zda překladač `C`/`C++` podporuje `C++` komentáře `//`. Konstrukce `1//**/2` si totiž vyloží buď jako `1` nebo jako `1/2`.

Několik prográmků napsaných pod jinými programovacími jazyky je uvedeno na konci řešení.

Na *ftp://atrey.karlin.mff.cuni.cz/pub/ksp/polyglot.c* si můžete prohlédnout ještě jeden pěkný program nalezený na Internetu. Jeho autor o něm tvrdí, že je napsán v těchto jazycích: `COBOL`, Pascal, Fortran, `C`, PostScript, shellový skript a spustitelný program v `MS-DOS`u. V každém z těchto jazyků by měl program vypsat pozdrav '`Hello polyglots`'. Neměl jsem možnost otestovat Fortran ani `COBOL`, ale v Pascalu, `C` a PostScriptu program skutečně funguje. Při spuštění v shellu `bash` vypisuje tento množství chybových hlášek, ale nápis je rovněž vypsán. Když jsem se ho pokusil spustit pod `MS-DOS`em, tak spadl.
Zde jsou uvedeny zdrojové texty výše zmiňovaných programů:
Pascal vs. `C`:

1)
```
const struct {/*} =1; begin writeln('Pascal'); end.
(*/} a; int main() { printf("C\n"); return 0; /*) {*/}
```

2)
```
const enum {/*} =1; begin writeln('Pascal'); end.
(*/x} a; int main() { printf("C\n"); return 0; /*) {*/}
```

3)
```
const x={1/*}1; begin writeln('Pascal'); end.
(*/}; int main() { printf("C\n"); return 0; /*) {*/}
```

4)
```
const int (*a); int main(){ printf("C\n"); return 0;
/*) =1; begin writeln('Pascal'); end. {*/}
```

5)
```
(*x)(); int main(){ printf("C\n"); return 0;
/*) begin writeln('Pascal'); end. {*/}
```

6)
```
begin (randseed) {/*}:=0; writeln('Pascal'); end. (*/
return 0; } int main(){ printf("C\n"); return 0; /*) {*/}
```

`C` vs. `C++`:

1)
```
int a[2]; int main(){ struct a{int b;};
printf( sizeof(a)==sizeof(struct a) ? "C++\n" : "C\n"); return 0;}
```

2)
```
int main(){ printf("does%s support //\n",1//**/2
```

```
?"":" not"); return 0;}
```

Ostatní jazyky:

1) *Martin Zlomek*: jednoduché propojení assembleru `TASM` a `C`.

```
;/*
.model tiny
.code
org 100h
start:
        mov ah,9
        mov dx,offset text
        int 21h
        xor ah,ah
        int 21h
text    db 'ASM',13,10,'$'
end start
*/ int main(){printf("C\n"); return 0;}
```

2) *Zdeněk Dvořák*: celkem mazané propojení Prologu (odzkoušeno na LPA win Prolog) a `C`.

```
//// . ?-op(50,fx,int). a:-
int X=3%2; int main(){printf("C\n");return 0;}
; //// . ?-write('Prolog'),nl.
```

3) *Zdeněk Dvořák*: propojení Prologu (odzkoušeno na LPA win Prolog) a Basicu (Q-Basic).

```
'Basic-comment'. ?-op(50,xf,x). ?-op(50,xfy,:). a:-
X = 0: 'a'. a:-
PRINT X; "Basic": 'Basic-comment'. ?-write('Prolog'),nl.
```

11-4-1 Výlet po řece — Aleš Přívětivý

V této úloze se mělo zjistit, zda ve vrcholově ohodnoceném stromu existuje prostá cesta (tj. cesta, ve které jdeme každým vrcholem právě jednou), jejíž součet hodnot vrcholů dává zadané číslo N. Bylo dobré si uvědomit dva fakty. Mezi každými dvěma vrcholy existuje právě jedna prostá cesta – to plyne z toho, že graf je strom. To, že se jedná o binární strom (na soutoku se stékají vždy jen dvě řeky), není až zas tak důležité a nemění to podstatně charakter úlohy. Daleko důležitější je první fakt, který znamená, že stačí zkontrolovat pouze počet cest kvadraticky úměrný počtu vrcholů, tj. pro každou dvojici vrcholů cestu mezi nimi.

Algoritmus bude tedy vypadat následovně: Pro každý vrchol grafu v_i budeme graf procházet do hloubky. Přitom si budeme počítat vzdálenost vrcholu, ve kterém právě stojíme, od vrcholu v_i. To budeme provádět tak, že při průchodu vrcholem směrem dolů přičteme jeho hodnotu a při návratu ji zase odečteme. Takto budeme znát délku cesty z vrcholu v_i do všech vrcholů, kterými

jsme prošli při procházení do hloubky – tedy do všech vrcholů. Po provedení tohoto pro všechny vrcholy v_i jsme zkontrolovali ceny cest mezi všemi dvojicemi vrcholů. Pokud se při procházení v nějakém vrcholu rovná cena cesty ceně hledané, algoritmus nalezl řešení a skončí. V opačném případě po zkontrolování všech cest víme, že cesta požadované ceny neexistuje.

„Časová složitost bude někde mezi lineární a exponenciální," jak vtipně uvedl jeden řešitel ve svém řešení. Přesněji kvadratická vzhledem k počtu vrcholů, tj. $O(n^2)$. To je proto, že časová složitost procházení je lineární a to provedeme právě n-krát. Lépe to ani nejde, protože musíme vyšetřit počet cest kvadraticky úměrný počtu vrcholů. Paměťová složitost roste lineárně s počtem vrcholů, tj. $O(n)$. To je způsobeno prohledáváním do hloubky a pamětí potřebnou na reprezentaci grafu.

K programu bych dodal jen formát vstupních dat. Nejprve je načten počet vrcholů a požadovaná cena cesty. Pak pro každý vrchol načteme čtyři čísla – cenu hotelu a tři sousedy v grafu udané pořadovými čísly, pokud je jich méně (například list má pouze jednoho souseda) mají zbývající hodnotu -1.
[Úloha na prázdniny: Tento problém je řešitelný dokonce v čase $O(n \log n)$, zkuste přijít na to, jak. Pro vyvážené stromy to je jednoduché, pro nevyvážené poněkud obtížnější, ale přeci jen zvládnutelné. –M.M.]

```
#include <stdio.h>

#define MAX 100

int cena[MAX], m[MAX][3];                    /* pole cen hotelu a sousedu */
int aktualni, pozadovana, hotelu;            /* pocet hotelu a ceny */

void trace (int kde, int odkud)              /* projde do hloubky stromem */
{                                            /* pricemz pocita cenu cesty */
    aktualni += cena[kde];                   /* od pocatecniho vrcholu do */
    if (aktualni == pozadovana)              /* ostatnich */
        {
          printf ("Cesta ceny %d nalezena...\n", pozadovana);
          exit (0);
        }
    for (int i=0; i<3; i++)
        if (m[kde][i]>=0 && m[kde][i]!=odkud) trace (m[kde][i], kde);
    aktualni -= cena[kde];
}

int main (void)
{
    scanf ("%d%d", &hotelu, &pozadovana);
    for (int i=0; i<hotelu; i++) scanf ("%d%d%d%d", cena+i, m[i], m[i]+1, m[i]+2);
    for (int i=0; i<hotelu; i++) trace (i, -1);    /* pro kazdy vrchol hledej ceny */
    printf ("Cesta ceny %d neexistuje...\n", pozadovana);    /* cest do ostatnich */
    return 0;
}
```

11-4-2 Komplot — Jan Kára

Všechna došlá řešení byla více méně správně a je možné je rozdělit do dvou skupin: První skupina počítala obsah tak, že z mnohoúhelníka postupně „ořezávala“ trojúhelníky. Přístup je to sice funkční, leč pomalý (obvykle $O(N^2)$ až $O(N^3)$). Druhá skupina počítala obsah jako součet obsahů lichoběžníků nebo trojúhelníků. Do této kategorie spadají i ti řešitelé, kteří se odkázali do literatury. Proti odkazům do literatury v zásadě nic nenamítám, ale přeci jen celou úlohu zamést tím, že je to vyřešeno v té a té knize se mi nezdá ideální. Vzhledem k tomu, že většina těchto řešitelů neuvedla ani náznak důkazu správnosti, přišli tím o bod (tento byl totiž připsán zpravidla panu H. J. Bartschovi). Jinak i ostatní řešitelé často na důkaz správnosti zapomněli...

Teď už k samotnému řešení. Uvažme mnohoúhelník „rozřezaný“ přímkami vedenými každým z bodů kolmo k ose x. Každá část musí být protnuta sudým počtem hran (jistě když projdeme hranou na jednu stranu, tak se opět někdy musíme vrátit zpět). V mnohoúhelníku pak je vždy část mezi první a druhou hranou, mezi třetí a čtvrtou hranou atd. Pokud by to tak nebylo, nějaká hrana by neohraničovala mnohoúhelník ale vedla by uvnitř něj či vně. Abychom tedy zjistili příspěvek dané části k obsahu, stačí tedy přičíst plochy pod lichými hranami a odečíst plochy pod sudými hranami (pokud by některým matematičtěji založeným jedincům vadilo, že plochy pod některými hranami mohou být nekonečné, mohou si představovat, že přičítáme pouze plochu k nejspodněji položenému průsečíku). Ještě se nám bude hodit, když si uvědomíme, že liché hrany vždy vedou zleva doprava (pro zadání proti směru hod. ručiček) a sudé opačně. To snadno dokážeme sporem. Nechť jsou pod sebou dvě hrany vedoucí stejným směrem. Bez újmy na obecnosti zleva doprava. Od pravého konce horní hrany musí nějak vést lomená čára k levému konci dolní hrany. Od pravého konce spodní hrany pak musí nějak vést lomená čára k levému konci horní hrany. To ovšem nelze bez protnutí první čáry, nebo bez průchodu mezi horní a spodní hranou. Spor.

Když si teď představíme zase celý mnohoúhelník, můžeme si všimnout, že se vždy plocha pod jednou hranou celá buď přičítala, nebo odečítala. To totiž plyne z toho, že hrany se nikde nemohou protínat, a tedy parita počtu hran nad danou hranou je vždy stejná (pokud náhodou nad naší hranou nějaká hrana přibyla, tak jich musel určitě přibýt sudý počet – jedna hrana doleva a jedna doprava).

K dalšímu zjednodušení algoritmu přispěje poznání, že nemusíme počítat obsahy pod hranami, ale stačí počítat obsahy lichoběžníků určených koncovými body hran a libovolnou pevně zvolenou přímkou (třeba osou x). Čtenář si jistě sám rozmyslí, že i v případech, kdy přímka protíná hranu, či kdy je hrana pod přímkou, přičteme nakonec správný obsah. Navíc pokud k výpočtu obsahu mechanicky používáme vzorec $(a_x - b_x) * (a_y + b_y)/2$, kde A a B jsou konce hrany,

bude automaticky zajištěno přičtení obsahu u hran jdoucích jedním směrem a odečtení obsahu u hran jdoucích druhým směrem.

Algoritmus má lineární časovou a konstantní paměťovou složitost. Správnost algoritmu byla ukázána výše.

Program je přímou implementací algoritmu, jen dělení dvojkou je vytknuto před všechny výpočty, takže až do konce můžeme počítat v celých číslech. (Z toho mimo jiné plyne velice zajímavý fakt, a to že obsah každého mnohoúhelníka s celočíselnými souřadnicemi vrcholů je celý počet „půlčtverečků“, což dokonce platí i pro oblasti s dírami, ale to je potřeba dokazovat trošku složitěji.)

```
#include <stdio.h>

int main (void)
{
  int AX, AY, PX, PY;                    /* Souřadnice aktuálního a předchozího
                                            bodu */
  int FX, FY;                            /* Souřadnice prvního bodu */
  int S = 0, N, i;                       /* Obsah; Počet bodů */

  scanf ("%d", &N);
  scanf ("%d %d", &FX, &FY);
  PX = FX; PY = FY;
  for (i = 1; i < N; i++)
  {
    /* Načteme hranu a přičteme lichoběžník */
    scanf ("%d %d", &AX, &AY);
    S += (AX - PX) * (AY + PY);
    PX = AX; PY = AY;
  }
  S += (FX - PX) * (FY + PY);            /* Ještě poslední lichoběžník... */

  /* Obsah vyjde v jednom směru záporně a v druhém kladně, tak se pojistíme :-) */
  if (S < 0)
    S = -S;
  printf ("Obsah je %f.\n",  ( (float)S)/2);
  return 0;
}
```

11-4-3 V jámě — Michal Píše

Triviálním řešením této úlohy je všechny vzdálenosti setřídit a poté je jednu po druhé projít a zjistit tak, mezi kterými dvěma sousedními prvky je největší mezera. Ale to nám dává algoritmus se složitostí $O(n \cdot \log n)$. Pravda, mohli bychom použít příhrádkového třídění, ale to by selhalo, pokud by souřadnice nebyla malá celá čísla. Další možnost je rozkouskovat si ulici na jednotlivé centimetry a u každého si poznačit, zda tam je či není díra, ale to by paměťová i časová náročnost rostly s délkou ulice. Takže je potřeba jít na to jinak...

Nejdříve budeme předpokládat, že díry jsou rozloženy přibližně pravidelně a rozdělíme si ulici (přesněji řečeno část ulice mezi první a poslední dírou)

na $n-1$ stejných úseků a pro každý úsek si budeme pamatovat souřadnici první a poslední díry v daném úseku. Potom probereme všechny díry a určíme, do kterého úseku přísluší a upravíme zapamatované hodnoty pro tento úsek. Pak už stačí jen najít takové dva sousední úseky, u kterých je rozdíl souřadnice poslední díry prvního úseku a první díry druhého maximální, a to je řešení naší úlohy.

Proč však nemůže být hledané řešení někde uvnitř úseku? Délka všech úseků dohromady je $x_r - x_l$ (x_r je souřadnice poslední díry, x_l první), délka jednoho úseku tedy $(x_r - x_l)/(n-1)$. Pokud by hledané řešení leželo uvnitř nějakého úseku, pak by maximální vzdálenost dvou děr byla menší než délka úseku, a proto vzdálenost od první díry k poslední (tedy součet vzdáleností mezi sousedními dírami, $x_r - x_l$) menší než součet délek všech úseků, což je spor.

Odhad časové a paměťové složitosti: nalezení minima a maxima provedeme snadno v čase $O(n)$, přiřazení jedné jámy do příslušného úseku v $O(1)$, pro všechny jámy tedy dohromady $O(n)$. Závěrečné nalezení nejdelší mezery mezi krajními jámami úseků rovněž $O(n)$. Celkem tedy $O(n)$. Paměťová složitost je také $O(n)$, protože si stačí pamatovat pouze informace o úsecích.

```
const nenastaven = -1;
begin
{ nacteni dat do pole dira[1..n]
  nalezeni nejlevejsi a nejpravejsi diry (min a max)
  nastaveni poli levy[1..n] a pravy[1..n] na konstantu nenastaven }
for i:=1 to n do begin
                   usek := ( dira[i] - min ) / ( max - min ) * (n-1);
                   if ( levy[usek] > dira[i] ) or
                      ( levy[usek] = nenastaven ) then levy[usek]:=dira[i];
                   if ( pravy[usek] < dira[i] ) or
                      ( pravy[usek] = nenastaven ) then pravy[usek]:=dira[i];
                 end;
vysledek:=0;
zarazka:=pravy[1];
for i:=2 to n do if ( levy[i] <> nenastaven ) then
                   begin
                     if ( levy[i] - zarazka > vysledek ) then
                       vysledek:=levy[i]-zarazka;
                     zarazka:=pravy[i];
                   end;
{ Vysledek je v promenne vysledek }
end.
```

11-4-4 Turingovy stroje — Martin Mareš

Analýza všech možných i nemožných Pascalských konstrukcí v úloze 11-3-4 vás jistě znechutila natolik, že nebyvše lačni po zatracení na věky věkův, netroufáme si cokoliv podobného zopakovat. Využijeme proto toho, že již víme, že Pascal se dá překládat do Mikroassembleru a dokážeme proto pouze, že se pro libovolný mikroassemblerský program M dá nalézt příslušný Turingův stroj T.

Hledaný Turingův stroj bude pracovat s abecedou $\Sigma = \{\circ, \heartsuit, \Lambda\}$ a obsah jeho pásky bude representovat paměťové buňky μ_1 uložené v „jedničkové soustavě“, a k tomu ještě prokládaně (možno samozřejmě i jinak, ale tento postup vede k nejjednodušší realizaci instrukcí μ_1 na TS). Konkrétněji: Program M má konečnou délku, takže využívá konečný počet registrů, a proto existuje n takové, že registry mimo $0, \ldots, n-1$ využity nejsou. Nyní j-tý registr μ_1 uložíme na políčka pásky TS s indexy $ni + j$, kde i prochází všemi přirozenými čísly (to znamená na j-té políčko a pak vždy $n-1$ políček přeskočíme, protože v nich budou uloženy hodnoty ostatních registrů). Je-li v registru j hodnota r, je prvních r políček příslušících tomuto registru zaplněno znaky $\heartsuit$ a ostatní znaky Λ. Navíc si na samý začátek pásky umístíme symbol $\circ$.

Instrukce programu M nebudeme ukládat na pásku, nýbrž je rovnou representovat stavy řídící jednotky stroje T. i-té instrukci programu M bude odpovídat nějaká množina stavů S_{ij}, přičemž provádění každé instrukce začíná stavem S_{i0} a po jejím provedení stroj přejde do stavu S_{j0}, kde j je číslo následující instrukce ($j = i+1$, pokud se nejednalo o skok) či do stavu Q (koncového; do něj jdeme, pokud nám program předepsal zastavení skokem před první instrukci), v němž Turingův stroj zastavíme posunem přes levý okraj pásky. Navíc se domluvíme, že po vykonání každé instrukce hlavu stroje přesuneme na první „datové“ políčko pásky (to znamená nalezneme symbol $\circ$ a pak poskočíme doprava).

Nyní již k realizaci jednotlivých instrukcí:

- `INC` k: Nalezneme začátek registru k (uděláme k kroků doprava), načež nalezneme první Λ v zápisu hodnoty tohoto registru (postupnými skoky o n políček doprava) a přepíšeme ji na $\heartsuit$.
- `DEC` k: Analogicky jako `INC`, pouze až narazíme na Λ, vrátíme se na předchozí „číslici“ téhož registru (o n políček zpět) a přepíšeme ji na Λ (pokud tam již žádná taková nebyla, poznáme to podle toho, že potkáme symbol $\circ$, tudíž i dekrement nuly vyjde správně jako nula).
- `CLR` k: Nalezneme začátek registru k a přepisujeme všechny $\heartsuit$ v jeho zápisu na Λ.
- `JEQ` x, y, z: (bez újmy na obecnosti $x < y$) Nalezneme první políčko zápisu x a zapamatujeme si ve stavu stroje, zda to byla Λ či $\heartsuit$. Poté posuneme hlavu o $y - x$ políček doprava a porovnáme, zda zde se nacházející odpovídající číslice registru y má stejnou hodnotu. Pokud nikoliv, přejdeme na následující instrukci, ježto $r_x \neq r_y$. Jinak posunem o $n + x - y$ políček doprava přejdeme na další číslici registru x a pokračujeme v porovnávání. Pokud jsou obě porovnávané číslice Λ, jsme již na konci obou čísel a tato jsou si rovna, tudíž zvolíme následující stav podle cíle skoku.

Tím je důkaz dokončen.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Zdeněk Dvořák	G Nové Město na Moravě	4	18	193
2.	Pavel Šanda	G Klatovy	4	18	181
3.	Jiří Cvachovec	G Tř. kpt. Jaroše, Brno	3	18	169
4.	Pavel Šimeček	G Tř. kpt. Jaroše, Brno	3	18	158
5.	Petr Zika	G Voděradská, Praha	4	18	151
6.	Alexandr Kára	G Hellichova, Praha	4	18	135
7.	Martin Zlomek	G Strážnice	2	18	123
8.	Petr Vršovský	G F. X. Šaldy Liberec	4	16	111
9.	Stanislav Živný	G Soběslav	4	16	105
10.	Ivan Piliš	G Velká okružná, Žilina	4	14	100
11.	Daniel Fiala	G Sušice	3	15	97
12.	Branislav Katreniak	SPŠ Brezno	4	10	89
13.	František Němec	G Zborovská, Praha	2	17	85
14.	Ondřej Zajíček	SPŠ strojnická, Chrudim	2	13	81
15.	František Folber	G Bráfova, Třebíč	3	18	78
16.	Petr Adam	G Bráfova, Třebíč	3	17	72
17. – 18.	Zdeněk Bouchner	G Telč	3	18	71
	Jiří Svoboda	G Zborovská, Praha	1	13	71
19. – 20.	Jan Jakubův	G Vlašim	3	11	70
	Milan Vraný	G Mikulášské nám., Plzeň	2	16	70
21. – 22.	Robert Káldy	G Zborovská, Praha	4	9	69
	Jozef Tvarožek	G Jura Hronca, Bratislava	1	10	69
23.	Radim Tyleček	G Zborovská, Praha	2	16	68
24.	Lukáš Matějka	G Lanškroun	3	16	57
25. – 26.	Miroslav Bajtoš	G Jura Hronca, Bratislava	3	9	52
	Roman Krejčík	G Zborovská, Praha	2	17	52
27. – 28.	Tomáš Vyskočil	G Lanškroun	3	13	51
	Jakub Zemánek	G Hustopeče	4	12	51
29.	David Štěrba	OA Mladá Boleslav	4	9	46
30.	Milan Roubal	G Mikulášské nám., Plzeň	4	9	41
31.	Lukáš Horák	G Hradec Králové	2	8	40
32. – 33.	Jakub Bystroň	G Karviná	3	7	36
	Slavomír Katuščák	G Konštantínova, Prešov	3	7	36
34. – 35.	Petr Chovanec	G Terezy Novákové, Brno	4	4	35
	Jan Drchal	G Nad Štolou, Praha	4	5	35
36.	Dávid Pál	G Jura Hronca, Bratislava	4	4	34
37. – 39.	Michal Fašina	G Jihlava	4	8	33

	Martin Nečaský	G Semily	3	6	33
	Jaroslav Tykal	G Jihlava	2	9	33
40. – 42.	Jan Hladký	G Tř. kpt. Jaroše, Brno	0	7	32
	Martin Vejmelka	G Českolipská, Praha	4	9	32
	Ondřej Vošta	G Soběslav	4	8	32
43. – 45.	Pavel Celba	G Úpice	2	7	29
	Michal Forišek	G Poprad	4	4	29
	Robert Poch	SPŠST Panská, Praha	2	4	29
46.	Ondřej Nekola	G Kolín	4	3	28
47.	Adam Slavický	G Nad Alejí, Praha	4	5	27
48. – 50.	Eduard Bejček	G Nad Štolou, Praha	4	3	24
	Miloslav Brada	G Tábor	4	4	24
	Petr Němec	G Arabská, Praha	4	5	24
51. – 52.	David Holec	G Tř. kpt. Jaroše, Brno	4	4	23
	David Kovář	G Telč	3	8	23
53. – 54.	Zdeněk Nový	G Tanvald	4	6	22
	Juraj Suchár	G Dubnica nad Váhom	3	3	22
55. – 57.	Jan Holeček	G Tř. kpt. Jaroše, Brno	4	3	21
	Tomáš Hrubý	G Klatovy	4	6	21
	Miroslav Novotný	SPŠ Jičín	2	5	21
58. – 59.	Jiří Plachý	G Uherské Hradiště	2	4	19
	Marek Sterzik	ZŠ Nové Sedlo	0	5	19
60. – 62.	Václav Novák	G Poděbrady	4	4	18
	Jakub Seidl	G Rychnov nad Kněžnou	2	4	18
	Jaroslav Urban	G Jana Opletala, Litovel	3	8	18
63.	Pavel Čížek	SPŠ Jedovnice	2	3	17
64.	Jan Bilak	SPŠST Panská, Praha	2	4	16
65.	Tomáš Holubec	G Vsetín	4	4	15
66.	Josef Dušek	G Aloise Jiráska, Litomyšl	2	5	14
67.	Václav Blaha	G Uherský Brod	3	4	13
68. – 69.	Martin Doležal	G Hradec Králové	2	4	11
	Martin Kukačka	G Jindřichův Hradec	4	2	11
70. – 72.	Jan Gahura	SPŠ Zlín	2	2	10
	Martin Hejna	SPŠE Dobruška	2	2	10
	Milan Kryl	G Jana Opletala, Litovel	4	2	10
73. – 74.	Jan Bartoš	G Rychnov nad Kněžnou	2	3	9
	Oldřich Vořechovský	G Rožnov pod Radhoštěm	4	3	9
75. – 76.	Martin Jurica	G Klatovy	4	3	8
	Jiří Samek	G Semily	4	2	8
77.	Juraj Matuš	G Sabinov	2	2	6
78.	Antonín Faltýnek	G Lanškroun	4	1	3

Pořadí řešitelů

79. – 80.	Josef Jetmar	G Mikulášské nám., Plzeň	4	4	2
	Lukáš Lipavský	G Arabská, Praha	2	1	2
81. – 94.	Petr Částek	G Teplice	2	0	0
	Jakub Červený	G Česká Lípa	4	0	0
	Martin Děcký	G M. Koperníka, Bílovec	2	0	0
	Jana Fabriková	G Velké Meziříčí	-1	0	0
	Michal Filka	SPŠST Panská, Praha	4	0	0
	David Hartman	G Příbram	4	0	0
	Jan Horčík	G Brandýs nad Labem	0	0	0
	Jiří Krejsa	G Semily	3	0	0
	Pavel Kubát	G Ivančice	2	0	0
	Petr Mandys	G Trutnov	3	0	0
	Jan Novotný	G Čakovice, Praha	0	1	0
	Ondřej Plášil	G Chomutov	2	0	0
	Tomáš Richter	G Dobruška	2	0	0
	František Seifrt	G Cheb	4	0	0

Obsah

Martin Mareš a kolektiv

Korespondenční seminář v programování XI. ročník

Autoři a opravující úloh:
Martin Bělocký, Jirka Hanika, Jan Hubička,
Ája Jančaříková, Jan Kára, Daniel Kráľ,
Martin Mareš, Michal Píše, Aleš Přívětivý,
Robert Špalek

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 1999

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

72 stran, 1 obrázek
Písmem Computer Modern v programu TEX vysázel Jan Kára
Vydání první
Náklad 300 výtisků