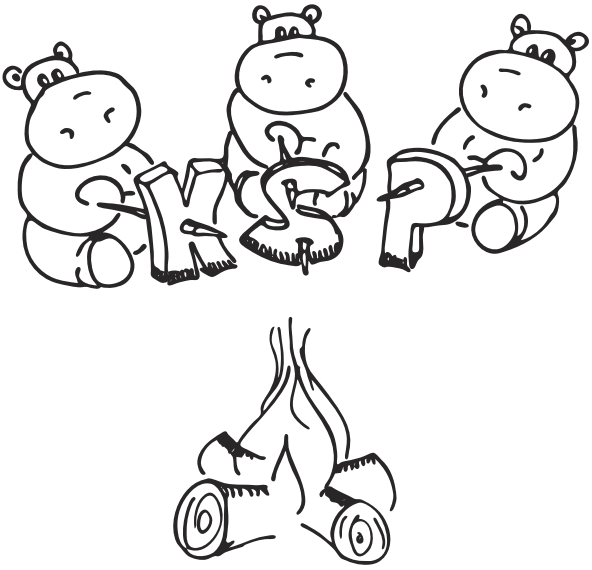
MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVII. ročník – 2004/2005



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVII. ročník – 2004/2005

matfyzpress

Praha 2005

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji!

ISBN 80-86732-00-8

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož sedmnáctý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešením úloh našeho semináře získávají středoškoláci 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 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 (ať už fyzickou či elektronickou). My je poté 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 – existují korespondenční semináře z fyziky a matematiky při MFF, 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
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Zadání úloh

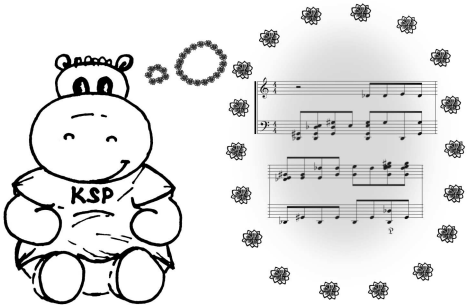
17-1-1 Výdělek bratří Součků — 10 bodů

Bratři Součkovi, Ain a Kábel, potomci známého velikého Suka, byli odmala talentovaní hudebníci. Jejich vzájemný vztah bohužel byl, jak jejich jména kážou, dosti špatný. I v dospělém věku si oba konkurovali jako hudební kritici.

Při vzácné příležitosti vystoupení známého zpěváka Miguela J. X. Sona byli oba bratři Součkovi firmou Granny najati, aby se pokusili odposlechnout J. X. Sonův největší hit. Oba bratři – každý sám – koncert navštívili a když se do Sonova hitu zaposlouchali, zjistili, že se neustále opakuje. Tak si oba poznamenali jeho začátek až do chvíle, kdy si byli jisti, že celý hit je jen opakování jimi zaznamenaného začátku.

Při odevzdání svých záznamů ale zjistili, že jsou různě dlouhé! Oba však ale trvají na tom, že zaznamenali skladbu správně, a obviňují toho druhého. Vedoucí firmy Granny, paní Babičková, si však myslí, že ačkoliv jsou jejich zápisy různě dlouhé, mohly by představovat jedinou skladbu. A vás poprosila, jestli byste jejich zápisy mohli porovnat.

Na vstupu dostanete Ainův i Kábelův záznam. Každý se skládá z délky a pak z jednotlivých not, které budeme pro jednoduchost zapisovat přirozenými čísly. Úkolem vašeho programu je říci, zda posloupnost, která vznikne jako nekonečné opakování Ainova zápisu je *stejná* jako ta, která vznikne jako nekonečné opakování zápisu od Kábela.



Příklad: Pokud je Ainův záznam $1, 2, 1, 2, 1, 2$ a Kábelův $1, 2, 1, 2$, zaznamenali oba bratři skladbu stejně. Pokud by Ain zaznamenal $1, 2, 1, 2, 3, 2$, nebylo by tomu tak.

17-1-2 Bůhdhova odměna — 10 bodů

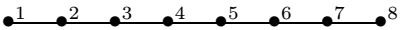
Když známý tigamský kupec Semtodaj Čornulaj Apadaj, věrný reprezentant svého národa, prodal další kus „své“ Tigamské plošiny, rozhodl se Bůhdha, že už se na to nemůže dál koukat. Ovšem jeho hlasité „Budiž černočerná tma!“ se minulo účinkem a vrátilo ozvěnou. (Přeci jenom Bůhdha nemůže být všemocný; kdyby mohl, dokázal by vytvořit neřešitelný problém, který by nevyřešil ani on sám – ale pak by nebyl všemocný. *QED.*)

A tak si usmyslel, že Tigamany alespoň odmění – obmění jejich jazyky. A to tak, aby si žádní obyvatelé dvou sousedních vesnic nerozuměli. Sousední vesnice jsou takové, mezi kterými vede (samozřejmě horská) pěšina. A protože jsme v horách, žádné dvě pěšiny se nekříží.

Ubohý Bůhdha ale dokázal vymyslet jen 6 odlišných jazyků. Zklamán dosavadními neúspěchy se raději obrátil na vás, abyste zjistili, zda je možné jeho ďá... božský plán provést.

Máte napsat program, který dostane na vstupu popis Tigamské říše – počet vesnic N a dále M pěšin, každá z nich spojuje právě dvě vesnice. Každá pěšina je obousměrná a navíc platí, že žádně dvě pěšiny se mimo vesnice nekříží (ani nadjezdem, natož tunelem). Úkolem programu je zjistit, zda je možno přiřadit každé vesnici jeden z šesti jazyků tak, aby si žádní obyvatelé sousedních vesnic nerozuměli. Pokud to jde, má vypsat jedno takové přiřazení.

Příklad: Pro následující situaci



stačí Bůhdhovi dokonce jen dva jazyky – rozdá je střídavě.

17-1-3 Chmatákův lup — 10 bodů

Cecil Hromdotruhlice, Mistr Antibankovních Technologií Álias Kraďas byl zářným potomkem svého otce. Zdědil po něm vše dobré, co měl a co se tak za nehet vešlo, ale také všechno špatné. Včetně svého povolání. A ne ledasjakého povolání. Cecil je totiž profesionální antibankovní činitel – to znamená, že bohatým bere a chudým koneckonců taky. Sice už nezbyl nikdo, komu by mohl dávat, než on sám, ale s touto nepříjemností se už všichni Hromdotruhlíkové dávno smířili.

Jednoho dne se Cecil vydal na prohlídku jedné obzvláště bohaté banky v přestrojení za hygienika telefonních sluchátek. Uvnitř ke svému Hromovému překvapení zjistil, že není schopen všechny cenné věci odnést! Chtěl by ale dostát své antibankovní cti a obrat banku o co nejvíc peněz.

Cecil dokáže unést nanejvýš (spíše nanejtíž) N kg lupu. V bance je P cenných věcí a u každé odhadl Cecil její hmotnost na m_i celých kg a cenu na c_i zlaťáků. Cena, na rozdíl od váhy, může být i desetinné číslo.

Napište program, který poradí Cecilovi, jaké předměty vzít, aby je ještě unesl a přitom jejich celková cena byla *největší možná*.

Příklad: Pokud dokáže Cecil unést $N = 8$ kg a v bance jsou tyto $P = 4$

i	1	2	3	4
m_i	5	4	3	2
c_i	12.5	10	6	7.5

cennosti, je pro Cecila nejlepší odnést věci 1 a 4. Pokud by ale byla jeho nosnost o kilogram větší ($N = 9$), bylo by nejlepší odnést předměty 2, 3 a 4.

17-1-4 Paloučkova výhra — 10 bodů

Ludvík Palouček, známý to milovník přírody, byl svým přítelem Pepou Běhavým vyzván k běžeckému závodu, který se má odehrát v Běhavého rodném městě. Ludvík se závodu nebojí, protože jeho přítel dostal jméno spíš po svých zažívacích potížích než kvůli rychlým nohám, ale nechce se mu trávit mnoho času jinde než na svém paloučku:„A jak dlouho to bude, Pepo, trvat?“ „Ale, stačí jedno kolečko,“ odpověděl mu vítězství chtivý kamarád.

Ludvík se této odpovědi chytl a rozhodl se naplánovat trasu závodu sám. Závod má začínat a končit na jednom místě (Pepa chtěl kolečko) a přitom má být co nejkratší, aby mohl být Ludvík co nejdřív doma. Když ale uviděl mapu města, zhrozil se a raději vás požádal o pomoc.

Na vstupu dostanete popis Běhavého města: N, což je počet křižovatek, a dále M ulic. Každá ulice je obousměrná, má nějakou délku a spojuje dvě křižovatky. Ačkoliv se ulice mimo křižovatky nekříží, ve městě může být mnoho nadjezdů a tunelů.

Vaším úkolem je zjistit, zda ve městě existuje nějaký *okruh*, a pokud ano, máte najít a vypsat libovolný *nejkratší* z nich i s jeho délkou. Okruh je posloupnost alespoň dvou neopakujících se ulic, přičemž po sobě následující ulice okruhu začínají a končí na stejné křižovatce – včetně první a poslední ulice okruhu. Délkou okruhu rozumíme součet délek všech jeho ulic.

Příklad: Pokud je v městě $N = 5$ křižovatek a ulice

odkud	kam	délka
1	2	2
2	3	3
1	3	9
3	4	1
4	5	3
1	5	2

tak nejkratší je okruh $1 \to 2 \to 3 \to 4 \to 5 \to 1$ délky 11. Všimněte si, že $1 \to 2 \to 1$ není okruh, protože je skládá z jediné opakující se ulice.

17-1-5 Jazykozpytcův poklad — 10 bodů

Co mají společného překladače programovacích jazyků, vyhledávání v textu, komprese dat nebo třeba také rozdělování slov? Na první pohled nepříliš, ale teoretickým informatikům se přesto podařilo najít teorii, která shrnuje základní věci z těchto oblastí (a mnohých jiných) a říká o nich mnoho zajímavého. Je to teorie automatů a formálních jazyků a právě té jsme se rozhodli věnovat náš letošní seriál.

Začneme nejprve názvoslovím:

- *Abeceda* je libovolná konečná množina znaků.
- *Slovo* α nad abecedou A je uspořádaná konečná posloupnost znaků abecedy A. Prázdné slovo značíme λ. Množinu všech možných slov nad abecedou A značíme A^*.
- *Jazyk* L nad abecedou A je nějaká podmnožina (klidně nekonečná) množiny A^*. Nenechte se zmást názvem jazyk, nemáme tím na mysli nějaký specifický programovací či dokonce přirozený jazyk (i když i tyto do naší definice spadají), jedná se zkrátka o nějakou množinu slov.
- Jsou-li α a β dvě slova, pak zápisem $\alpha\beta$ rozumíme jejich zřetězení za sebe.
- Zápisem α^i rozumíme i-násobné opakování slova α (tj. třeba $(ab)^2 = abab$).

Příklad: nad abecedou $\{a, b, c\}$ lze vybudovat třeba jazyky $\{baba, abba, bac\}$ (ten je konečný) či $\{a^i b^i;\ \forall i \in \mathbf{N}\}$ (ten je nekonečný a patří do něj třeba slova ab či $aaabbb$, nepatří tam abb ani $bbbaaa$).

U každého jazyka lze studovat například tyto dvě věci: jak daný jazyk *rozpoznávat* (rozhodnout o zadaném slovu, zda patří do jazyka) a jak *generovat* všechna slova daného jazyka. K prvnímu úkolu slouží „stroje“ čili *automaty,* s jejichž nejběžnějšími typy se v seriálu seznámíme. To druhé mají na starost *gramatiky.* Gramatika je formální popis pravidel, pomoci kterých se vytvářejí všechna slova daného jazyka. Původně je vymyslel lingvista pan Chomsky pro popis přirozených jazyků – z hodin českého jazyka jistě znáte větné rozbory, tj. pravidla typu [věta] → [podmětná část][přísudková část], kde podmětná a přísudková část se opět rozpadají na podčásti, atd. Gramatika se tedy skládá ze sady přepisovacích pravidel $\alpha \to \beta$, kde na obou stranách vystupují slova sestávající se jednak z pomocných symbolů (těm se říká *neterminální*) a jednak ze symbolů *terminálních* (po domácku *terminálů*), které už se dále neexpandují (čili už se na ně dále nepoužívají přepisovací pravidla). Terminály se vlastně dají chápat jako jednotlivé znaky použité abecedy.

Formální definice: Gramatikou nazveme čtveřici (V_N, V_T, S, P), kde:

- V_N je konečná množina neterminálních symbolů,

- V_T je konečná množina terminálních symbolů,
- $S \in V_N$ je počáteční neterminální symbol,
- P je konečný systém přepisovacích pravidel $\alpha \to \beta$, kde $\alpha, \beta \in (V_N \cup V_T)^*$ a α obsahuje alespoň jeden neterminální symbol. Dvě pravidla $\alpha \to \beta$ a $\alpha \to \gamma$ obvykle zkráceně zapisujeme jako $\alpha \to \beta \mid \gamma$.

Gramatika vezme počáteční symbol a začne ho expandovat (nahrazovat) podle některého z uvedených pravidel. Typicky bývá několik možností, jak expandovat, tehdy můžeme použít libovolné vhodné pravidlo. Expanze končí, když z expandovaného řetězce vymizí všechny neterminální symboly. Všechna možná slova, která pomocí jedné gramatiky G můžeme různými posloupnostmi expanzí dostat, tvoří *jazyk gramatiky,* ten budeme značit $L(G)$.

Jako příklad si uvedeme gramatiku, která popisuje jazyk všech aritmetických výrazů s čísly 1 a 2 používajících operace + a ∗ a závorky. Použijeme neterminální symboly $V_N = \{V, T, F\}$, terminální symboly $V_T = \{1, 2, +, *, (,)\}$, počáteční symbol je V a pravidla:

$$V \to T + V \mid T$$
$$T \to F * T \mid F$$
$$F \to (V) \mid 1 \mid 2.$$

Například výraz $1 + 2 * 2$ je generován posloupností přepisů $V \to T + V \to F + V \to 1 + V \to 1 + T \to 1 + F * T \to 1 + F * F \to 1 + 2 * F \to 1 + 2 * 2$. Slovo 22++1 zjevně pomocí sady našich pravidel nevytvoříme.

V prvním dílu seriálu se seznámíme s nejjednodušší rodinou jazyků, s takzvanými *regulárními* jazyky. Regulární jazyk je takový jazyk, ke kterému existuje *konečný automat,* který ho rozpoznává.

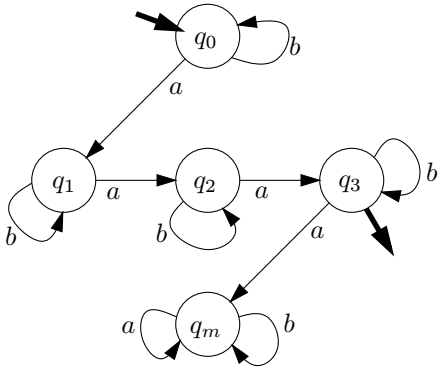
Co že to ten konečný automat (též zkratkou KA) vlastně je? Matematici mají rádi nejrůznější uspořádané k-tice, formálně si proto konečný automat zavedeme jako pětici (Q, A, δ, q_0, F), kde:

- Q je konečná množina stavů stroje,
- A abeceda, nad kterou stroj pracuje,
- $\delta : Q \times A \to Q$ je tzv. *přechodová funkce*, která ke každé kombinaci stavu a načteného znaku určuje nový stav, do kterého automat přejde,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových (přijímajících) stavů.

A nyní lidsky: konečný automat je stroj, který dostane na vstupu nějaké slovo a má se o něm rozhodnout, zda ho přijme či nikoliv. Automat se může nacházet v konečné a předem dané množině stavů Q, na začátku dejme tomu ve stavu q_0. V každém kroku své činnosti načte jeden znak ze vstupu a podle tohoto znaku se rozhodne, do jakého stavu přejde. To je dáno přechodovou funkcí, která k aktuálnímu stavu q a znaku a vrátí nový stav q', tedy

$\delta(q, a) = q'$. Pokud po přečtení všech znaků slova automat skončil v některém z přijímacích stavů z množiny F, říkáme, že slovo bylo přijato, jinak bylo odmítnuto. Všechna slova, která daný automat A přijímá, tzv. *jazyk automatu,* značíme $L(A)$.

Příklad: automat nad abecedou $\{a, b\}$, přijímající všechna slova s právě třemi výskyty znaku a a libovolným počtem výskytů znaku b. Automaty je nejpřehlednější zapisovat obrázkem:



Automat má 5 stavů, stav q_0 je počáteční, stav q_3 je jediný přijímací. Stav q_i nám vlastně značí, že doposud jsme načetli i znaků a, stav q_m je záchytný a znamená, že a-ček už jsme přečetli moc.

V následujících dílech seriálu si představíme více jazykových rodin, ukážeme si jak jejich příslušné rozpoznávající stroje (tzv. akceptory), tak také odpovídající typy gramatik. Například regulárním jazykům odpovídají gramatiky obsahující pouze pravidla ve tvaru $X \to \alpha Y$, $X \to \alpha$, kde $X, Y \in V_N$ a $\alpha \in V_T^*$.

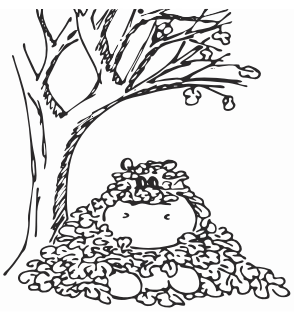
Ale nyní již **soutěžní úlohy:**

1. Uvažme abecedu $A = \{0, 1\}$. Slovo nad touto abecedou bude kódovat číslo zapsané v dvojkové soustavě, s obvyklou konvencí, tj. nejvýznamnější bit nalevo, nejméně významný napravo. Sestrojte konečný automat nad A rozpoznávající všechna čísla dělitelná třemi a nedělitelná dvojkou (tj. jeho jazykem budou všechna slova kódující číslo dělitelné 3 a nedělitelné 2). *[5 bodů]*

2. Sestrojte gramatiku se stejným jazykem jako v první úloze – tj. generující právě čísla v binárním zápisu, která jsou dělitelná třemi a nejsou dělitelná dvojkou. *[5 bodů]*

Kromě zkonstruovaného automatu a gramatiky by měl být součástí řešení i stručný slovní popis toho, proč daný automat resp. gramatika dělá to, co má, případně důkaz, že hledáme marně a to, co chceme, neexistuje.

17-2-1 Prasátko Květ(ák)omil — 10 bodů



Květomil byl úplně normální prasátko. Již v útlém dětství ničím nevynikal mezi svými vrstevníky, své rodiče nepřekvapoval svou předčasnou duševní vyspělostí. Ani později nijak nezastiňoval své přátele a známé v žádné činnosti, kterou prováděl, snad s jedinou výjimkou, a tou bylo jídlo (proto si vysloužil přezdívku Pašík Kvašík). Byl prostě úplně normální obyčejné prasátko.

Když vyrostl, zvolil si úplně normální obyčejné povolání a stal se programátorem u firmy *Ptáček Sáček, práce všeho druhu.* Jeho práce u této firmy (konkurující známému příteli Ferdy Mravence) byla také úplně normální a obyčejná. Proto, když Sáček kontroloval práci svých zaměstnanců, aby zjistil, proč jeho konkurence dodává rychlejší programy, zjistil, že programy Pašíka Kvašíka jsou pomalé až hrůza. Prostě úplně normální a obyčejné.

A tak si Sáček najal vás, abyste mu pomohli Kvašíkovy programy zrychlit. Kvašíkovy programy jsou posloupnosti přiřazení do proměnných, což jsou řetězce znaků složené z malých písmen, velkých písmen a podtržítek. Na pravé straně přiřazení může být buď proměnná nebo operace „+“ nebo „∗“ aplikovaná na dvě proměnné. Tyto operace jsou komutativní, neboli $a + b = b + a$ a $a * b = b * a$.

Vaším úkolem je napsat program, který dostane Kvašíkův program skládající se z N přiřazení a má říci, jak moc ho lze zrychlit, čili říci, kolik nejméně operací „+“ a „∗“ stačí k tomu, aby nový program přiřadil do všech proměnných stejnou hodnotu jako Kvašíkův. Formálně pro každých i prvních řádků Kvašíkova programu musí v novém programu existovat místo, kdy jsou hodnoty všech proměnných z Kvašíkova programu v obou programech shodné. Můžete využívat toho, že operace „+“ a „∗“ jsou komutativní, ale jejich asociativita a distributivita se neberou v úvahu, čili $a + (b + c) \neq (a + b) + c$ a také $(a + b) * c \neq a * c + b * c$.

Příklad: Vlevo je Kvašíkův program, vpravo náš.

```
                          t = b + c;
a = b + c;                a = t;
d = a + b;                d = a + b;
e = c + b;                e = t;
                          s = a * e;
f = a * e;                f = s;
a = d;                    a = d;
g = e * e;                g = s;
```

Zatímco Kvašíkův program potřeboval operací pět, náš si vystačí se třemi, takže výstup programu by měl být „3“.

17-2-2 Bobr Béďa — 10 bodů

Béďa byl hodný bobr, který poslouchal svou maminku. A ta ho, jako každá jiná maminka, naučila čistit si zoubky. Když Béďa vyrostl, začal používat zubní pastu *Bělosup*, po které, jak bylo na jejím obalu napsáno, „zuby nádherně vypadají."

A byla to pravda. Hodnému Béďovi vypadaly všechny zuby. Dostal sice samozřejmě umělé, ale protože bobři vyrábějí vše ze dřeva, byly celé dřevěné. Leč s dřevěnými zuby Béďa nemohl chodit do normální bobří práce, protože by sotva přehryzal dřevěný strom. A tak začal pracovat u firmy Ptáčka Sáčka.

Přestože byly Béďovy zuby dřevěné, stále dokázal skvěle pracovat se dřevem, a tak byl zaměstnán jako výrobce integrovaných odvodů. Integrovaný odvod je součástka na rozvod vody. Představit si ji můžete jako dřevěnou desku, na které je pravidelná čtvercová síť bodů, některé sousední body jsou spojeny vydlabanou spojnicí. Za sousední body se považují takové, které se liší v jedné souřadnici o jedničku (čili vnitřní body mají každý čtyři sousedy). Béďův úkol je dodělat na odvod nějaké spojnice sousedních bodů, aby bylo možné dostat se z každého bodu do jiného.

Protože je integrovaný odvod ze dřeva, dokáže Béďa vytvořit svislé spojnice rychleji než vodorovné (jdou „po letech"). Změřil si, že udělat jednu vodorovnou nebo dvě svislé spojnice mu zabere stejně času. A protože je Béďa hodný, chce mít každý odvod co nejrychleji hotový.

Napište Béďovi program, který dostane na vstupu N a M (rozměry mřížky bodů na integrovaném odvodu), S (počet již hotových spojnic), a popis jednotlivých spojnic (souřadnice dvou bodů, které spojuje). Výstupem by měl být seznam spojnic takových, že po jejich přidání do integrovaného odvodu se půjde dostat z každého bodu do každého a navíc doba na vytvoření těchto spojnic bude co nejmenší (čili neexistuje jiná množina spojnic, která by se dala vyrobit v kratším čase a přitom by splnila popsanou podmínku).

Příklad: Pro $N = 4, M = 4$ a odvod

je pro Béďu nejlepší vytvořit spojnice nakreslené dvojitě.

17-2-3 Krkavec Kryšpín — 8 bodů

Krkavec Kryšpín byl velmi známý a uznávaný básník, snad každý se obdivoval jeho poezii. Což ale znamená, že to byl básník velmi zaneprázdněný, protože každé zvířátko po něm chtělo jinou básničku. Jako každý básník i Kryšpín potřeboval inspiraci – zpěv ptáků. Ovšem po čase se mu všichni ptáci začali

vyhýbat, nebavilo je věčně stát před krkavcem, který je neustále napomínal, ať nekrákají.

To se Kryšpínovi ani trochu nelíbilo a usmyslel si, že zpěvavé ptáky nějak naláká. Rozhodl se, že jim postaví fontánu roztodivného tvaru – ptáci budou obdivovat fontánu (hned ji překřtil na Fontárnu, když u ní bude psát básně), on ptačí zpěv a ostatní jeho poezii.

Hned vyrobil zkušební Fontárničku, ale zjistil, že potřebuje najít její těžiště, aby mu nepadala ze stojánku. Vypravil se za Ptáčkem Sáčkem, zda by mu jeho firma mohla pomoci, ale Sáček se mu jenom vysmál, protože nechtěl zradit své ptačí přátele. Dokážete Kryšpínovi poradit vy?

Na vstupu dostanete N bodů zadaných svými souřadnicemi, které představují Fontárnu. Tu si můžete představit jako mnohoúhelník, který vznikne, spojíme-li vždy dva po sobě jdoucí zadané body (a ještě první s posledním). Tento mnohoúhelník je navíc konvexní (všechny jeho vnitřní úhly jsou menší než 180°). Vaším úkolem je najít *těžiště* zadaného mnohoúhelníka.

Příklad: Pro $N = 4$ a body $[0, 0], [12, 6], [12, 12], [0, 18]$ by měl váš program odpovědět, že těžiště se nachází na souřadnicích $[5, 9]$.

Pro náročnější: Pokud bude váš program fungovat i pro nekonvexní mnohoúhelníky, můžete dostat další 3 body.

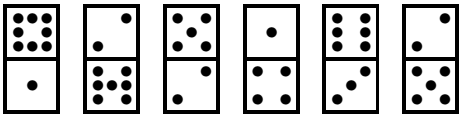
17-2-4 Mravenec Ferda — 11 bodů

Ferdova tajná láska, Beruška, přišla jednou za ním a jeho přítelem Pytlíkem na návštěvu. K Ferdově veliké lítosti ale odmítla jeho návrh najít stonožku Šmajdulu a svázat jí nohy dohromady, a místo toho se podivovala Pytlíkově kvetoucí firmě. Zklamaný Ferda se rozhodl Berušce předvést, že co dokáže jeho přítel, dokáže taky. Aby ale nemusel začínat s prázdnými tykadly, rozhodl se, že se stane vedoucím firmy *Ptáček Sáček, práce všeho druhu*.

Příští den zašel do Sáčkovy firmy a spustil: „Podívejte se na něj, na ptáčka. Zaměstnává naprosto neschopné programátory, chudáčkovi Bobrovi vyrazil zuby, aby pro něj vyráběl odvody a je to takový trumbera, že ani nedokáže spočítat těžiště. A takové zvíře vám má šéfovat?“ Zvířátka uznala, že na tom je něco pravdy, a rozhodla se dát Ferdovi šanci. Když vyřeší jejich úlohu, stane se jejich šéfem.

Zvířátka položila před Ferdu N kostiček domina, na každém jsou nahoře a dole dvě celá čísla od 1 do K. Každou kostičku domina může Ferda obrátit, čili její horní číslo se dostane dolů a naopak. Jeho úkolem je dosáhnout toho, aby rozdíl součtu horní a dolní řady čísel na kostičkách byl co nejmenší. A navíc toho má dosáhnout přehozením co nejmenšího počtu kostiček. Ferda ale zjistil, že je to i nad jeho mravenčí síly. Pomůžete mu?

Příklad: Pro $K = 8$ a $N = 6$ a dominové kostky



musí Ferda otočit druhou, třetí a pátou dominovou kostku.

Pozor: I sám Ferda si po chvíli přemýšlení uvědomil, že nestačí najít si kostičku s největším rozdílem čísel, která by mu pomohla snížit celkový rozdíl, otočit ji a podle stejného postupu pokračovat dál (to nezabere ani pro uvedený příklad). Kdyby to bylo takhle jednoduché, určitě by vás o pomoc nepožádal.

17-2-5 Jazykozpytcova pomsta — 10 bodů

V druhém dílu seriálu o formálních jazycích se ještě stále budeme věnovat nejjednodušší jazykové rodině, regulárním jazykům. Minule jsme si řekli, co jsou to gramatiky, konečné automaty a regulární jazyky, a nyní si zavedeme věc, která se může na první pohled jevit jako naprostý nesmysl – *nedeterminismus*. Pokud pracuje nějaký proces (stroj, algoritmus, ...) deterministicky, pak pokud známe jeho vstupní data, jsme schopni dopředu předpovědět, jak se bude chovat. Ovšem u nedeterministického procesu nic takového určit nemůžeme.

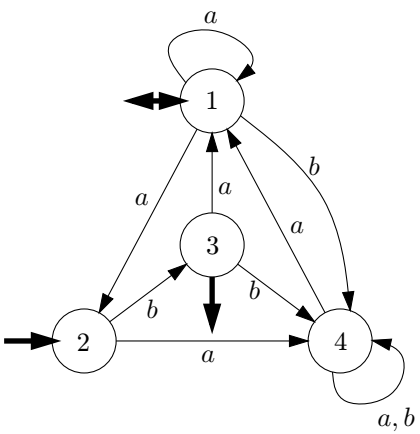
Pokud je konečný automat ve stavu q a načte nové písmeno p, je pomocí přechodové funkce přesně definováno, jaký bude nový stav, do kterého stroj přejde. Ovšem nedeterministický konečný automat má k jedné kombinaci stavu a písmena na výběr hned několik možných stavů, do kterých může přejít, a z těch si jeden naprosto libovolně vybere.

Formálně je nedeterministický konečný automat (odteď ho budeme značit NKA) pětice (Q, A, S, δ, F), kde

- Q je konečná množina stavů,
- A je konečná abeceda, nad kterou automat pracuje,
- S je množina počátečních stavů,
- $\delta : Q \times A \to P(Q)$ je přechodová funkce, která ke každé kombinaci aktuálního stavu a nově načteného písmenka vrací neprázdnou množinu stavů (tedy nějakou podmnožinu Q), do kterých automat může přejít,
- $F \subseteq Q$ je množina přijímacích stavů.

Zbývá nadefinovat, kdy je či není dané slovo nedeterministickým konečným automatem přijato. Výpočtem NKA nad slovem w rozumíme konkrétní průběh činnosti stroje při postupném čtení písmen slova w. Díky nedeterminismu je možných výpočtů pro jediné slovo více. Slovo w je tedy přijato, jestliže mezi všemi možnými výpočty nad slovem w existuje alespoň jediný, který končí v přijímacím stavu.

Příklad NKA nad abecedou $A = \{a, b\}$:



Stroj používá stavy $Q = \{1, 2, 3, 4\}$, počáteční stavy jsou $S = \{1, 2\}$ a koncové $F = \{1, 3\}$. Ve stavech 1 a 4 jsou dvě možnosti, jak se při načtení písmena a může stroj zachovat.

Ačkoli to tak na první pohled rozhodně nevypadá, přidáním nedeterminismu do konečných automatů jsme ve skutečnosti nijak nezvedli „výpočetní sílu“ stroje.

Tvrzení. *Množina jazyků přijímaných deterministickými KA je stejná jako množina jazyků přijímaných nedeterministickými KA.*

Jinými slovy, ke každému NKA jsme schopni sestrojit ekvivalentní (přijímající stejný jazyk) DKA a naopak. Tato skutečnost rozhodně stojí za to být dokázána. První převod je jednoduchý, každý DKA je totiž pouze případem takového NKA, jehož přechodová funkce vrací vždy jednoprvkovou množinu stavů. Převod druhý je však už o poznání těžší.

Mějme libovolný nedeterministický konečný automat $M = (Q, A, S, \delta, F)$ a hledejme k němu ekvivalentní DKA $M' = (Q', A, q'_0, \delta', F')$. Nejprve se zamysleme, jak bychom asi M simulovali „programátorsky“. Nejspíše bychom si napsali program, který pro vstupní slovo probacktrackuje všechny možné výpočty. Další metodou je v každé situaci, kdy se M rozhoduje z několika možností, spustit pro každou takovou možnost paralelně proces, který ji dále simuluje. Právě na této myšlence je založena naše konstrukce. Jak ovšem takový postup namodelovat omezenými prostředky konečných automatů?

V novém konečném automatu M' především podstatně rozšíříme množinu stavů, položíme $Q' = P(Q)$, kde $P(Q)$ značí množinu všech podmnožin množiny stavů Q původního stroje M. Jeden stav stroje M' tedy bude kódován hned několika stavy stroje M. Počáteční stav bude $q'_0 = S$, čili množina obsahující všechny počáteční stavy stroje M. Přechodovou funkci $\delta'(q', a)$ pro

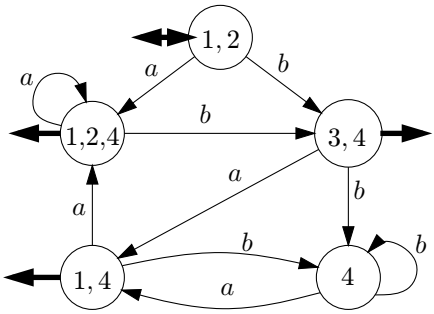
$q' = \{q_1, q_2, \ldots, q_k\} \in Q'$ a $a \in A$ definujeme takto:

$$\delta'(\{q_1, q_2, \ldots, q_k\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \cdots \cup \delta(q_k, a)$$

Všimněte si, že v jednom stavu z Q' jsme schopni najednou simulovat hned několik stavů původního stroje. Stejně tak nová přechodová funkce δ' provede paralelní výpočet hned pro několik stavů původního stroje najednou. Zbývá položit $F' = \{q' \in Q';\ \exists q \in q' :\ q \in F\}$, čili přijímací stav stroje M' je taková množina stavů stroje M, ve které se vyskytuje alespoň jeden přijímací stav stroje M.

Právě jsme si ale uvědomili, že v jednom stavu z Q' paralelně simulujeme několik různých výpočtů původního stroje. Podle definice přijímání nedeterministickým konečným automatem je přijímací stav z F' takový, že alespoň jeden výpočet z odpovídající množiny výpočtů stroje M je přijímací. Oba stroje tedy přijímají stejný jazyk, čímž je důkaz hotov. Počet stavů nového stroje M' nám sice oproti M exponenciálně narostl, ale je stále konečný a splňuje tak definici konečného automatu.

Nás ukázkový stroj tedy po převodu bude vypadat takto (nikdy nedosažitelné stavy z obrázku vynecháme):



Zavedením nedeterminismu jsme tedy nijak nerozšířili možnosti konečných automatů, nicméně právě předvedená věta se dá třebas využít k zjednodušení nejrůznějších důkazů. O strojích, jejichž nedeterministická verze má větší výpočetní sílu než verze deterministická, si povíme v příštích dílech seriálu.

Ale nyní už asi netrpělivě čekáte na další **soutěžní úlohy:**

- Každý stroj má svá omezení a nejinak tomu je v i případě konečného automatu. Nechť U je jazyk nad dvoupísmennou abecedou $\{(,)\}$, který je tvořen všemi správně uzávorkovanými výrazy. Např. slovo ()(()) patří do U, slovo))(do U nepatří. Formálně dokažte, že nemůže existovat konečný automat, který by rozpoznával jazyk U. *[6 bodů]*
- Nechť L_1 a L_2 jsou libovolné regulární jazyky. Zřetězením regulárních jazyků L_1 a L_2 nazveme jazyk $L_1.L_2 = \{uv;\ u \in L_1, v \in L_2\}$. Tedy např. pro $L_1 = \{a, ab\}$ a $L_2 = \{c^i;\ i \in N\}$ je $L_1.L_2 = \{ac, abc, acc, abcc, \ldots\}$. Dokažte, že $L_1.L_2$ je také regulární jazyk. *[5 bodů]*

17-3-1 Spisovatel Vilík — 10 bodů

Spisovatel Vilík Šekjrspír (v originále Shaker's Pear, neboli šejkařova hruška, oblíbený to alkoholický nápoj) byl velmi známý autor. Nicméně měl pocit, že jeho knihám se nedostává tolik pozornosti, kolik by si jí podle něj zasloužily. Nakonec se mu dokonce povedlo přijít na to, čím by to mohlo být. (A vzhledem k tomu, že ho dnes zná skoro každý, měl asi pravdu.)

Zjistil, že jeho knihy jsou poněkud nudné, protože se v nich často opakují celé kusy textu. A tak si řekl, že všechna opakování nějakého textu ze svých knih smaže, nebudou potom tak nudné a navíc budou mít otevřený konec.

Když takto upravil první knihu, zjistil, že z ní zbyla ještě celá třetina. Známý myslitel Cibulka mu tedy poradil, že za stejná slova může považovat i dva kusy textu, které mají stejnou délku a skládají se ze stejných znaků. (Nezáleží tedy na pořadí znaků v obou slovech.)

Vilík teď už ale sám nedokáže najít stejná slova, Cibulka mu sám také nechce pomoci (prý řeší zajímavější problém), a tak to zbyde na vás.

Napište program, který dostane na vstupu číslo k a text délky N znaků skládající se z písmen '`a`'..'`z`', '`A`'..'`Z`', mezer, teček, otazníků a vykřičníků. Má spočítat délku nejdelšího začátečního úseku tohoto textu, ve kterém se ještě *nevyskytují* dvě *shodná* slova. Dvě slova jsou shodná, pokud to jsou souvislé podřetězce zadaného textu a obě se skládají z právě k stejných písmen, i když pořadí těchto písmen může být různé. Velká a malá písmena nerozlišujte, čili '`a`' = '`A`'.

Příklad: Pro $k = 3$, $N = 14$ a text '`Den␣bude␣hned.`' je správný výsledek 12 (v prvních dvanácti písmenech nejsou žádná shodná slova). Pro text '`Den␣je␣tu␣hned`' je správná odpověď 6 (protože '`␣je`'='`je␣`').

17-3-2 Popleta Truhlík — 10 bodů

Pan Truhlík byl veliký popleta. Nedokázal si zapamatovat, jak se jmenuje, kde pracuje, a často se mu stalo, že nepoznal svou vlastní dceru a ptal se jí: „Holčičko, nevíš, kde bydlím?“ (Zaměstnáním by se nejlépe hodil na matematika.)

Dokud žil s maminkou, bylo vše v pořádku, ale jakmile se odstěhoval z domu, začal mít se svou popleteností veliké problémy. A tak si řekl, že když by mamince občas zavolal, určitě by mu pomohla. Problém byl ale v tom, že i když si vzpomněl, že má maminku, nedokázal si vzpomenout na její telefonní číslo. Jednou se mu povedlo si celý problém uvědomit a svěřil se s ním prvnímu kolemjdoucímu, kterým byl zrovna myslitel Cibulka.

Pan Cibulka po chvíli hovoru zjistil, že Truhlík je sice veliký popleta (nebo pan Popletal je veliký truhlík?), ale pamatuje si všechny večerníčkové postavy. A tak vymyslel následující zlepšovák: místo telefonního čísla si bude Truhlík

pamatovat větu, která se bude skládat ze jmen večerníčkových postav takovou, že když se napíše na klávesnici telefonu, vznikne chtěné číslo. Klávesnice telefonu vypadá následovně:

1	2	3
abc	de	fgh
4	5	6
ij	klm	no
7	8	9
pqr	st	uvw
	0	
	xyz	

(Číslo 7951140151651 jde zapsat jako „Rumcajz a Manka“.)

Jenomže Truhlík si sám takovou větu nedokáže vymyslet, Cibulku už o pomoc kvůli panu Popletalovi požádat nestihl, a tak zbýváte jen vy.

Na vstupu dostanete seznam slov, která si pan Truhlík dokáže zapamatovat. Tato slova se skládají pouze ze znaků ‘`a`’..‘`z`’ a celková velikost slovníku je P písmen. Dále dostanete seznam N telefonních čísel, každé o délce C_i. Vaším úkolem je pro každé telefonní číslo najít posloupnost slov ze slovníku takovou, že pokud se napíše na klávesnici, vznikne kýžené telefonní číslo, případně říci, že to není možné.

Příklad: Jsou-li ve slovníku slova „brok, kuba, je“, číslo 5911425911 můžeme nahradit větou „kuba je kuba“, ale číslo 1765911 není možné žádnou větou složenou ze slovníkových slov nahradit.

Bonus: Pokud dokážete navíc vypsat pro každé telefonní číslo takovou větu, která má ze všech možných správných vět nejmenší počet slov, Truhlík se vám určitě bohatě (bodově) odmění za to, že si ji zapamatuje brzy.

17-3-3 Starosta Hafák — 10 bodů

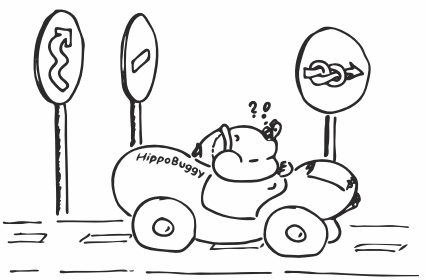
Pan Hafák se stal navzdory svému nevhodnému jménu starostou Kocourkova a jako starosta dostal za úkol starat se o bezpečnost chodců na silnicích. Nechal provést několik nezávislých odborných průzkumů a zjistil, že k největšímu počtu nehod dochází, když chodci přecházejí na zelenou. Rozhodl se tedy, že přikáže chodcům přecházet na červenou.

Slavný myslitel Cibulka mu ovšem vysvětlil, že takhle by rozhodně dopravních nehod neubylo, a poradil mu jiný způsob: pokud by všechny ulice v Kocourkově byly jednosměrné, bude šance, že nějakého chodce přejede auto, jenom poloviční.

To se Hafákovi velmi zalíbilo a ihned nechal udělal ze všech silnic jednosměrky. Při cestě domů ale zjistil, že to nebyl úplně dobrý nápad, protože už

z první křižovatky, na kterou dojel, nevedla žádná silnice v jeho směru. A protože se Cibulka mezitím, mumlaje si nějaké nuly a jedničky, ztratil, budete muset Hafákovi poradit vy.

Váš program dostane na vstupu popis silniční sítě v Kocourkově. Ta se skládá z N křižovatek a M silnic, každá silnice je obousměrná a spojuje dvě různé křižovatky. Žádné dvě silnice se mimo křižovatky nestýkají, mohou se ale mimoúrovňově křížit. Vaším úkolem je zjistit, kolik nejvýše silnic jde zjednosměrnit tak, aby bylo možné se ve výsledné zjednosměrněné síti dostat z nějaké křižovatky na jinou právě tehdy, když to šlo i v původní síti. A kromě počtu by měl váš program vypsat, jak má zjednosměrněná síť vypadat.



Příklad: V síti $N = 3$, $M = 3$ se silnicemi spojujícími každé dvě křižovatky lze zjednosměrnit všechny tři silnice, výsledná síť bude vypadat například takto: $(1 \to 2), (2 \to 3), (3 \to 1)$. Pokud by v síti byla ještě čtvrtá křižovatka, která by byla spojena silnicí s první křižovatkou, silnice mezi touto čtvrtou a první křižovatkou by zjednosměrnit nešla.

17-3-4 Myslitel Cibulka — 10 bodů

Poté, co jste snadno vyřešili problémy, které vám myslitel Cibulka (vlastním jménem Filip Bonifác Narcis Cibulka) vymyslil, získali jste si jeho respekt, a tak se rozhodl obrátit se na vás se svým vlastním problémem.

Cibulka si vymyslel zvláštní posloupnost čísel, kterou nazval po sobě Filipova Bonifácova Narcisova Cibulkova posloupnost. (Protože si to ale nikdo nemohl zamapatovat, zkrátil to na FiBoNaCiho.) Její první dva členy jsou 1 a 2 a každý další člen jest roven součtu dvou členů předchozích. Matematicky máme tedy posloupnost $\{F_n\}_{i=0}^{\infty}$, kde $F_0 = 1$, $F_1 = 2$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$.

Tato posloupnost se Cibulkovi tolik zalíbila, že se rozhodl zapisovat pomocí ní veškerá čísla, se kterými bude pracovat. Každé takto zapsané číslo je posloupnost nul a jedniček $a_n a_{n-1} \ldots a_1 a_0$ a jeho hodnota je $\sum_{i=0}^{n} a_i \cdot F_i$. Po krátké úvaze si uvědomil, že takový zápis nebude jednoznačný (třeba $011 = 100$), a proto vymyslil ještě *normalizovaný* zápis, který je stejný jako právě popsaný,

jen se v něm nesmí vyskytnout dvě jedničky vedle sebe a nesmí začínat nulou (čili 11 ani 0100 není normalizované, ale 100 je).

Poté, co se dostatečně vychválil za svou genialitu, zjistil, že není schopen s takovými čísly vůbec pracovat. Už jenom je sečíst je veliký problém. A tak se nyní obrátil na vás, zda byste mu nemohli vypomoci.

Zkuste napsat Cibulkovi program, který dostane na vstupu dvě čísla v nenormalizovaném FiBoNaCiho zápisu (tyto zápisy mají délky N a M) a vypíše zadaná čísla a jejich součet v normalizovaném tvaru. Není snad nutno dodávat, že skutečná hodnota zadaných čísel bude tak velká, že se nemůže vejít do žádného celočíselného typu (může jít o tisíce cifer ve FiBoNaCiho zápisu).

Příklad: Pro vstup 11101 a 1101 by měl váš program vypsat $100101 + 10001 = 1001000$.

Hintík: Cibulka vám ještě prozradil, že každé nezáporné číslo v normalizovaném tvaru zapsat jde.

17-3-5 Jazykozpytcova naděje — 9 bodů

Na jisté nejmenované univerzitě vědecky působil a samozřejmě též vyučoval nadšený lingvista, pan Choam Nomsky. Svým studentům často zadával domácí úkoly a nejčastějším úkolem bylo sestrojení konečného automatu, který má rozpoznávat nějaký zadaný jazyk. (Pro nezasvěcené: pokud netušíte, o čem je řeč, nahlédněte do zadání první série, kde jsou potřebné pojmy definovány.)

Jenže jeho studenti nepatřili zrovna k nejbystřejším a často nosili automaty, které používaly obrovské množství stavů, i když jazyk šlo rozpoznávat automatem s podstatně méně stavy. Kontrola správnosti obrovských automatů způsobovala panu Nomskému četné vrásky a bolesti hlavy, rozhodl se tedy, že před kontrolou správnosti si musí automat zjednodušit. Za zjednodušený automat, říkejme mu odborně *redukovaný,* považoval takový automat (Q, A, δ, q_0, F) ekvivalentní s původním, který neměl žádné nedosažitelné stavy a žádné dva stavy nebyly ekvivalentní. Co to znamená:

- Stav q je *nedosažitelný,* pokud neexistuje slovo u nad abecedou A, že by pro něj výpočet skončil ve stavu q.
- Dva různé stavy p a q jsou *ekvivalentní,* pokud pro každé slovo u nad abecedou A platí následující: výpočet nad slovem u startující ze stavu p skončí přijetím slova u, právě když výpočet nad slovem u startující ze stavu q skončí přijetím slova u.

O redukovaných automatech se dají dokázat některé zajímavé skutečnosti, například že libovolné dva ekvivalentní automaty se zredukují na stejně velký a „stejně vypadající“ automat, ale tím vás tentokrát zatěžovat nebudeme. Nyní po vás nechceme nic snazšího, než vymyslet co nejefektivnější algoritmus a posléze napsat program, který panu Choamu Nomskému usnadní jeho úděl,

čili ke konečnému automatu zadanému na vstupu najde příslušný redukovaný automat (což vlastně není nic jiného než ekvivalentní automat s co nejmenším počtem stavů).

Program nejprve načte z první řádky vstupu počet stavů n (očíslujme si je tedy 1 až n), počet symbolů abecedy a (taktéž si je očíslujme 1 až a), číslo počátečního stavu p a počet přijímacích stavů f. Následuje řádek s f čísly, které udávají přijímací stavy. Pak je na vstupu n řádků, každý s a čísly. Číslo c umístěné v i-tém řádku a j-tém sloupci znamená, že $\delta(i, j) = c$. Program by měl na výstup vypsat redukovaný automat v podobném formátu.

Příklad: vstup je vlevo, vzorový výstup vpravo.

```
5 2 1 2          2 2 1 1
1 3              1
1 2              1 2
2 3              2 1
3 4
4 1
3 5
```

17-4-1 Mandarinková zeď — 10 bodů

Veliký císař *Čching Ňamňam No-san* byl osvíceným vládcem Mandarínie. A jako osvícený vládce znal i zvyky a svátky jiných zemí. Ze všeho nejvíce se mu líbil jakýsi křesťanský svátek – Vánoce. Ten den totiž všichni dostávají mnoho dárků a on jako veliký osvícený panovník by jich určitě dostal opravdu mnoho. A tak se rozhodl, že se v Mandarínii budou Vánoce slavit také.

Prostí Mandaríni a Mandarínky nebyli ovšem jeho nápadem moc nadšeni, protože hlavní postava Vánoc *Santa Hood-san* měl podle *No-sana* chudým brát a jemu dávat. A proto se císař rozhodl, že si raději zkontroluje, jestli budou jeho poddaní Vánoce radostně slavit.

Kolem celé Mandarínie je postavena Velká Mandarinková zeď. Na této zdi jsou v pravidelných rozestupech strážní věže a v každé je jeden strážce. A *No-san* chce, aby právě tito strážci kontrolovali dodržování Vánoc. Práce strážců je ovšem velmi nudná, a tak každý strážce požaduje jistý počet *různých* medailí, aby byl se svou prací spokojen.

Císař chce všem strážcům vyhovět, ovšem rád by ušetřil, a tak se rozhodl použít co nejméně druhů medailí a rozdat je strážcům tak, aby každý dostal právě tolik různých medailí, o kolik si řekl, a navíc žádní dva sousední strážci neměli medaili stejného druhu (to, že nějaký strážce vidí, že jeho levý a pravý soused mají stejné druhy medailí, už císařovi nevadí). Poradíte?

Na vstupu dostanete počet strážních věží N a dále čísla a_1 až a_N, kde a_i reprezentuje počet medailí vyžadovaných strážcem číslo i. Vaším úkolem je zjistit, kolik nejméně druhů medailí je potřeba, aby každý strážce dostal, kolik

chce různých druhů medailí, a aby žádní dva sousední strážci (sousedí spolu strážci i a $(i+1)$ a navíc ještě N a 1) nedostali ani jednu medaili stejného druhu.

Příklad: Pro 5 strážců a požadavky $2, 2, 2, 2, 2$ je minimální počet medailí 5.

17-4-2 Válicie — 10 bodů

Poté, co byl *Santa Hood-san* několikrát, zrovna když se jako na potvoru nedíval žádný strážce, zboulován, rozhodl se *No-san*, že bude muset prosazovat Vánoce ještě o něco důrazněji. A tak se rozhodl zavést v Mandarínii Vánoční policii, zvanou Válicie. (I když zlí jazykové tvrdí, že její název pochází spíš od toho, že si Válicisti stále válí šunky.)

Mandarínie je vlastně jedno velké město (obehnané zdí). A aby v něm císař udržel pořádek, rozhodl se postavit na některých křižovatkách stanice Válicie, a to tak, aby na konci každé ulice byla alespoň jedna stanice. Ale aby se nestalo, že na sebe v temných a strašidelných uličkách Mandarínie zaútočí Válicisté ze dvou stanic, které jsou na koncích jedné ulice, je třeba postavit stanice tak, aby na koncích každé ulice ve městě byla postavena právě jedna stanice. Císař je (jako obvykle) velký škudlil a minimalista, a tak by chtěl, aby stanic musel postavil co nejméně.

Na vstupu dostanete graf o N křižovatkách a M ulicích. Každá ulice je obousměrná a spojuje právě dvě křižovatky a žádné dvě ulice se mimo křižovatky nekříží (mimoúrovňově mohou). Vaším úkolem je zjistit, na kolika nejméně křižovatkách je třeba postavit Válicejní stanice tak, aby na koncích jedné ulice byla stanice právě jedna. Kromě počtu těchto křižovatek vypište i křižovatky, kde mají stanice stát. Pokud je řešení více, stačí vypsat libovolné řešení, pokud není řešení žádné, vypište odpovídající zprávu.

Příklad: Pro 6 křižovatek a 4 ulice spojující křižovatky $(1, 2)$, $(1, 5)$, $(3, 4)$ a $(1, 6)$ jsou potřeba dvě stanice na křižovatkách 1 a 3. Pro 3 křižovatky a 3 ulice $(1, 2)$, $(2, 3)$ a $(3, 1)$ stanice postavit nejde.

17-4-3 Phirma — 10 bodů

Poté, co dokázal *No-san* udržet v Mandarínii klid, se jeho pozornost přesunula k tomu, aby, když už Vánoce tak horko těžko zavedl, dostal odpovídající množství dárků. A protože mezi chudými už mnoho dárků hodných Velkého *No-sana* nebylo, rozhodl se je hledat jinde.

Snad nejznámější firmu v Mandarínii založili paní *Čestná* a pan *Jakobi*. A protože firma dělala čest svému jménu, byla také nejbohatší. Ledva to císař zvěděl, rozhodl, že mu Vánoční dárek zaplatí právě ona.

Firma *Jakobi-Čestná* musí dodat časově seřazený seznam výdajů a příjmů a císař určí daně podle toho, jak dlouhé bylo nejdelší časové období, kdy firma vykazovala zisk (takzvaný *kradit*). Ovšem účetní této firmy dokážou falšovat zisky opravdu bleskově, proto by *No-san* potřeboval zjistit požadované údaje co nejrychleji. A tak se obrátil na vás.

Napište císaři program, který dostane na vstupu číslo N a dále posloupnost N celých čísel. Vaším úkolem je najít a vypsat nejdelší úsek (to je souvislá podposloupnost) takový, že součet čísel v tomto úseku je větší než nula. Pokud je takových úseků více, vypište libovolný s největším součtem.

Příklad: Pro posloupnost $(1, -2, -5, 1, 1, 1, 1, 1, -1)$ má hledaný nejdelší úsek délku 7 a je to úsek $(1, 1, 1, 1, 1, 1, -1)$.

17-4-4 Antifrňákovník — 10 bodů

Mandarínům se nakonec *No-sanovy* klidné Vánoční svátky natolik znelíbily, že se rozhodli císaři utéct. Ovšem Mandarinková zeď obsazená strážemi s jejich záměry moc nesouhlasila. A tak si Mandaríni, aby se na útěk mohli pořádně připravit, založili Sportovní Klub Utek' & Utekl.

Po dlouhé debatě se členové SK Utek' & Utekl dohodli, že si postaví stroj *antifrňákovník,* který Mandarinkovou zeď rozbije, a oni budou moci utéct. Ale aby jejich práce nemohla být *No-sanovi* nikým z nich prozrazena, rozhodli se, že každý bude znát pouze část antifrňákovníku.

Jaké bylo nakonec jejich (ale ne naše) překvapení, když po sestavení celého přístroje zjistili, že nikdo neví, jak propojit jeho elektrické obvody. Dokonce ani neví, jaké konce drátů na jednom konci přístroje odpovídají koncům na straně druhé. A protože si *No-san* usmyslel, že právě vy budete dalším sponzorem jeho vánočních dárků, rozhodli jste se s dokončením antifrňákovníku pomoci.

Na obou koncích přístroje je N konců drátů očíslovaných 1 až N a dále zemnění, což je drát, o kterém jako jediném víte, jak je propojen. Můžete vlevo dráty na zemnění napojovat a odpojovat a na pravé straně můžete měřit, zda mezi koncem drátu a zemněním teče elektrický proud. Vaším úkolem je říci, jaký konec drátu na straně levé je spojen s jakým koncem na straně pravé. Bohužel se může stát i to, že některé dráty jsou přerušeny a nevedou nikam.

Máte tedy napsat program, který dostane na vstupu počet konců drátu N, vypisuje příkazy `+X` pro připojení levého konce X-tého drátu na zemnění, `-X` pro odpojení levého konce X-tého drátu od zemnění a `?X` pro změření napětí na pravém konci X-tého drátu a zemnění (na tyto dotazy odpovídá uživatel). Nakonec má vypsat, které levé konce drátů jsou připojeny na jaké pravé (nevodivé dráty nevypisujte vůbec, vodivé vypište všechny). S levým i pravým koncem drátu může být spojen nanejvýš jeden opačný konec. Na začátku není na zemnění připojen žádný konec levého drátu.

Příklad: Pro $N = 3$ a následující „rozhovor“

výstup programu	*uživatelský vstup*
`+1`	
`?1`	*Ano*
`-1`	
`+2`	
`?3`	*Ano*
`-2`	
`+3`	
`?2`	*Ne*

je správná odpověď $1 \to 1$, $2 \to 3$.

17-4-5 Jazykozpytec vrací úder — 15 bodů

Minule jsme si praktičtější úlohou odpočinuli od rozmanité teorie formálních jazyků, což nyní opět napravíme. Nastal čas, abychom od jednoduchých regulárních jazyků pokročili k složitějším *bezkontextovým jazykům.* Název těchto jazyků plyne ze souvislosti s gramatikami, kterou si ovšem ukážeme až v příštím díle seriálu. (Nezasvěceným doporučujeme, aby si prostudovali seriálové úlohy předchozích sérií.)

Zavedeme si podstatně mocnější výpočetní prostředek než byl konečný automat, tzv. *zásobníkový automat.* Ten vznikne tak, že starý známý konečný automat vybavíme zásobníkem, což je paměť potenciálně neomezené kapacity, ve které jsou naskládané symboly z nějaké pevné abecedy, ale je možno přistupovat vždy jen k symbolu, který je na vrcholu a buďto tento symbol odebrat a nebo přidat další nad něj.

Většina vlastností KA zůstane zachována, jen přechodová funkce se nyní bude počítat z kombinace aktuálního stavu, písmene na vstupu a symbolu na vrcholu zásobníku, tedy trojice (q, p, z). Funkce potom vrátí nový stav, do kterého má stroj přejít, a také posloupnost symbolů, kterými se nahradí dosavadní vrchol zásobníku. Oproti konečnému automatu, který v každém kroku musel ze vstupu přečíst právě jedno písmeno a z něj počítat přechodovou funkci, umí zásobníkový automat také načtení písmene vynechat, což si můžeme představovat jako načtení prázdného znaku λ (a přechodovou funkci tudíž počítat z trojice (q, λ, z)). Vše si zavedeme formálně.

Formálně definujme *deterministický zásobníkový automat (DZA)* jako sedmici $M = (Q, A, Z, \delta, q_0, z_0, F)$, kde

- Q je konečná množina stavů,
- A je konečná vstupní abeceda,
- Z je konečná zásobníková abeceda (tedy symboly, které lze ukládat na zásobník),
- $\delta : Q \times (A \cup \{\lambda\}) \times Z \to Q \times Z^*$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $z_0 \in Z$ je počáteční zásobníkový symbol (čili symbol, který je při spuštění automatu uložen na zásobníku)
- $F \subseteq Q$ je množina přijímacích stavů.

Jeden výpočet přechodové funkce $\delta(q, a, z) = (q', w)$, neboli vykonání instrukce $(q, a, z) \to (q', w)$ pro $q, q' \in Q$, $a \in A \cup \{\lambda\}$, $z \in Z$ a $w \in Z^*$ znamená, že aktuální stav q se změní na q', ze vstupu se přečte písmeno a (anebo také nepřečte, v případě že $a = \lambda$) a aktuální symbol na vrcholu zásobníku z se nahradí posloupností w (třeba prázdnou či jednoprvkovou) zásobníkových symbolů (symboly zapsané vlevo se do zásobníku umístí níže než symboly vpravo). Pokud by bylo možné provést jak instrukci s $a = \lambda$, tak s konkrétním znakem, automat si vybere možnost s $a = \lambda$.

U zásobníkového automatu na rozdíl od KA nepožadujeme, aby byla přechodová funkce δ definována pro všechny možné kombinace stavu, písmene a zásobníkového symbolu. Výpočet stroje se zastaví při dvou příležitostech: pro (q, a, z) není definována žádná instrukce nebo došlo k odstranění všech symbolů ze zásobníku. Všimněte si, že zásobníkový automat s (ne)vhodnou přechodovou funkcí (tedy vlastně programem) se již může zacyklit v nekonečné smyčce.

Zbývá si přesně říci, kdy je dané slovo přijato. Na rozdíl od konečných automatů, u zásobníkových automatů můžeme stanovit hned dvě možnosti přijetí.

- Slovo $u \in A^*$ je přijímáno DZA M *koncovým stavem,* pokud se stroj M spuštěný na slovo u po konečném počtu kroků zastaví, celé slovo u je přečteno a M se nachází v přijímacím stavu. Jazykem DZA M přijímajícího stavem nazveme množinu všech slov, která M přijímá, značíme ji $L(M)$. Množině všech jazyků, které lze rozpoznávat DZA koncovým stavem (tedy všech takových jazyků L, že pro L existuje nějaký DZA M přijímající stavem takový, že $L = L(M)$), se říká *deterministické bezkontextové jazyky,* budeme ji značit *BKS*.
- Slovo u je přijímáno DZA M *prázdným zásobníkem,* pokud se stroj M spuštěný na slovo u po konečném počtu kroků zastaví, celé slovo u je přečteno a zásobník stroje M je vyprázdněný. Jazykem DZA M přijímajícího zásobníkem nazveme množinu všech slov, která M přijímá,

značíme ji $N(M)$. Množině všech jazyků, které lze rozpoznávat DZA prázdným zásobníkem (tedy všech takových jazyků L, že pro L existuje nějaký DZA M přijímající zásobníkem takový, že $L = N(M)$), se říká *bezprefixové bezkontextové jazyky,* budeme ji značit BKZ.

Příklad: Přijímat jazyk $L = \{0^n 1^n;\ n \in N\}$ zásobníkovému automatu nečiní potíže, narozdíl od konečného automatu (což jsme si dokázali v seriálové úloze druhé série). Setrojíme si tedy DZA M, který bude přijímat prázdným zásobníkem. Vstupní abeceda M bude $A = \{0, 1\}$, množina stavů $Q = \{l, p\}$, zásobníkové symboly $Z = \{z, 0\}$, počáteční stav bude l a počáteční zásobníkový symbol z, přijímací stavy F nejsou podstatné. Sadu instrukcí (čili přechodovou funkci δ) sestrojíme takto:

$$\begin{aligned}
\delta(l, 0, z) &= (l, 0) && \ldots \text{čte první symbol 0}\\
\delta(l, 0, 0) &= (l, 00) && \ldots \text{čte další symbol 0}\\
\delta(l, 1, 0) &= (p, \lambda) && \ldots \text{čte první symbol 1}\\
\delta(p, 1, 0) &= (p, \lambda) && \ldots \text{čte další symbol 1}
\end{aligned}$$

Pokud bychom chtěli raději přijímat koncovým stavem $F = \{q_F\}$, pak první instrukci změníme na $\delta(l, 0, z) = (l, z0)$ (čili neodstraníme počáteční symbol hned na začátku) a přidáme ještě navíc jednu instrukci:

$$\delta(p, \lambda, z) = (q_F, \lambda) \quad \ldots \text{detekuje úspěšný konec}$$

Uvědomíme si, že každé DZA M přijímající prázdným zásobníkem lze převést na DZA přijímající koncovým stavem, jinými slovy tedy $BKZ \subseteq BKS$. Nový stroj bude mít jiný počáteční stav, řekněme q_0', a na začátku výpočtu původní počáteční symbol na zásobníku z podloží ještě jedním pomocným symbolem, řekněme z'. To se udělá například instrukcí $\delta(q_0', \lambda, z) = (q_0, z'z)$. Dále se pokračuje v původním programu, ale dodáme ještě speciální instrukce $\delta(q, \lambda, z') = (q_F, \lambda)$ pro každý $q \in Q$, které když uvidí na zásobníku z' (neboli zásobník původního stroje se vyprázdnil), přejdou do přijímacího stavu a vyprázdní zásobník (čímž skončí). Opačný převod však provést nelze.

Soutěžní úloha 1: Ukažte, že jazyk $L = \{0^n 1^m;\ 0 < n \leq m\}$ lze rozpoznávat DZA koncovým stavem, ale neexistuje DZA přijímající prázdným zásobníkem, který by L rozpoznával. Najděte příklad regulárního jazyka (tedy rozpoznatelného konečným automatem), který nelze rozpoznávat DZA prázdným zásobníkem (a pochopitelně zdůvodněte proč). *[6 bodů]*

Podobně jako u konečných automatů i u zásobníkových automatů můžeme velmi podobně zavést nedeterministickou verzi stroje (viz zadání druhé série). *Nedeterministický zásobníkový automat (NZA)* se od DZA liší tím, že přechodová funkce $\delta : Q \times (A \cup \{\lambda\}) \times Z \to P(Q \times Z^*)$, kde značením $P(X)$ rozumíme

množinu všech podmnožin X, nyní vrací hned několik možností, jak může stroj zareagovat. Z nich si stroj jednu libovolnou vybere. Stejně tak pokud má stroj na výběr mezi čtením písmene a nebo jeho nečtením ($a = \lambda$), může si vybrat libovolnou možnost. Dané slovo u je přijato NZA M koncovým stavem resp. prázdným zásobníkem, pokud mezi všemi možnými výpočty nad u existuje alespoň jediný, po jehož konci je celé u přečteno a M se nachází v přijímacím stavu, resp. celé u je přečteno a zásobník je prázdný. DZA je tedy zjevně pouze speciálním případem takového NZA, kde přechodová funkce vrací vždy jednoprvkovou množinu.

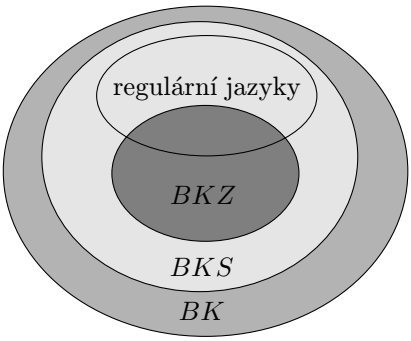
Soutěžní úloha 2: Narozdíl od DZA, přijímání koncovým stavem a přijímání prázdným zásobníkem je u NZA ekvivalentní. Tedy ke každému NZA přijímajícímu prázdným zásobníkem lze zkonstruovat ekvivalentní NZA přijímající koncovým stavem a naopak. Ukažte. *[3 body]*

Obě množiny jazyků přijímaných NZA stavem i NZA zásobníkem jsou tedy stejné. Těmto jazykům se říká *bezkontextové jazyky* a označíme si je BK.

Soutěžní úloha 3: Sestrojte NZA, který umí rozpoznávat jazyk L všech palindromů nad abecedou $\{a, b\}$. (Palindrom je slovo, které se čte pozpátku stejně jako zepředu.) Také ukažte, že jazyk L nelze rozpoznávat DZA prázdným zásobníkem. (Lze dokonce dokázat, že L se nedá rozpoznávat DZA koncovým stavem. To je ale o dost obtížnější a po vás to nechceme. Pokud ovšem někdo zašle *správný* důkaz i této varianty, štědrý bodový bonus ho jistě nemine.) NZA jsou tedy výpočetně silnější stroje než DZA. *[6 bodů]*

Připomínáme, že vaše řešení by měla obsahovat matematicky správné a pokud možno i formální argumenty. Nicméně jelikož zásobníkové automaty jsou už poměrně složité stroje, nebudeme již takoví puntičkáři co se týká požadavků na matematický formalismus.

Po vyřešení všech soutěžních úloh vlastně sami ukážete tento vztah mezi bezkontextovými jazyky, deterministickými bezkontextovými jazyky a bezprefixovými bezkontextovými jazyky:



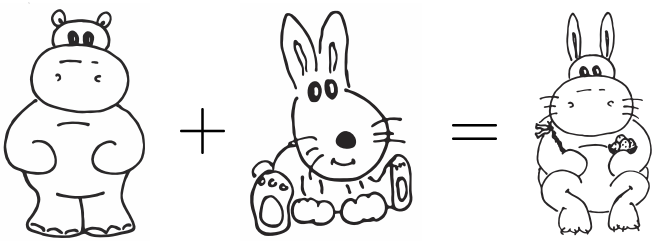
17-5-1 Velkovezír — 10 bodů

Když šel malý hrošík koledovat, srazil se na cestě se zajíčkem. Zajíček, který velmi pečlivě dodržoval velikonoční zvyky, také se mu říkalo **Vel**mi **ko**merční **ve**likonoční **z**aječí **r**aubíř, mu povídá: „Dávej přece pozor, když jde koledovat ten nejlepší koledník z okolí!“

„Vůbec nejsi nejlepší, Velkovezíre. Já jsem minulý rok vykoledoval čtyřicet dva vajíček!“ „No, to je sice pravda, ale za poslední tři roky jsem jich já vykoledoval sto dvanáct!“ „Ale před pěti lety jsem jich já vykoledoval šedesát čtyři!“

Když ani po notné chvíli nepřestali, rozhodli jste se jim pomoci a spravedlivě určit, který z nich je lepší koledník. Nakonec jste se dohodli, že lepší koledník je ten, jehož průměr vykoledovaných vajíček za souvislé období alespoň K roků je největší.

Napište zvířátkům program, který dostane na vstupu přirozená čísla N a K taková, že $1 \leq K \leq N$. Dále dostane posloupnost N celých čísel a vaším cílem je najít takovou souvislou podposloupnost této posloupnosti délky alespoň K, že má největší *aritmetický průměr*, což je součet jejích prvků vydělený jejich počtem. Pokud je takových podposloupností víc, vypište libovolnou z nich. Ale protože se mezitím hádka přiostřila, měl by váš program pracovat co nejrychleji.



Příklad: Pro $N = 5$, $K = 2$ a posloupnost $(4, 8, -2, 15, -5)$ by měl váš program najít $(8, -2, 15)$ s průměrem 7.

17-5-2 Ranní hroše — 9 bodů

Poté, co jste rozhodli při hrošíka s Velkovezírem (bohužel v hrošíkův neprospěch), se malý hrošík vrátil domů a stěžoval si tatínkovi, že Velkovezír je lepší koledník než on. „Tatínku, proč je lepší než já?“ „A kdypak jsi dneska vstával?“ „Časně ráno, sluníčko ještě nezapadlo.“ „A vida ho, našeho lenocha. Nevíš, že ranní hroše dál doduše? Zajíček určitě vstává brzo a každý mu dá spoustu vajíček.“

To malého hrošíka nadchlo. Pokud je to pravda, určitě by dokázal vstát jednou v roce už před polednem. Ale aby zjistil, jestli je to opravdu tak, jak tatínek říká, běžel se zeptat zajíčka, odkdy dokdy chodil o Velikonocích koledovat.

Ale když se vrátil, musel jít špinit nádobí (hroši mají čisté nádobí neradi) a proto vám dal následující úkol.

Dostanete dvě množiny H a V, každá obsahuje nějaké intervaly tvaru (od, do). Jednotlivé intervaly znamenají odkdy dokdy chodila zvířátka koledovat, v množině H jsou časy hrošíka a v množině V časy Velkovezíra. Máte zjistit, zda hrošík někdy začal a skončil s koledováním dřív než zajíček. Matematicky řečeno zjišťujete, zda existuje nějaký interval h z množiny H a v z množiny V, že h začíná dříve než začíná v a h končí dříve než končí v, čili že $h_{od} < v_{od}$ a $h_{do} < v_{do}$.

Příklad: Pro množiny $H = \{(10, 100), (5, 200)\}$ a $V = \{(8, 110), (20, 105)\}$ je hledané $h = (10, 100)$ a $v = (20, 105)$, protože $10 < 20$ a $100 < 105$. Pokud by bylo $V = \{(8, 110), (9, 105)\}$, hledané intervaly by neexistovaly.

17-5-3 Nouze V-dáli-hrocha — 8 bodů

Hrošík teď už věděl, proč je zajíček lepší koledník než on, a rozhodl se, že ho příští rok překoná. Ale aby toho dosáhl, musí si své koledování podrobně naplánovat. Rozhodl se, že bude věren přísloví Nouze naučila V-dáli-hrocha houstnouti, bude jíst jen šestkrát denně a celý rok plánovat své koledování.

Rád by si vybral nejrychlejší trasu, podél které bude koledovat. A aby byla opravdu nejrychlejší, rozhodl se projít všechny možnosti, které má, a vybrat tu nejlepší.

Hrošík bude koledovat u N svých sousedů. Jeho trasa je vlastně pořadí, v jakém bude své sousedy navštěvovat, takže je to posloupnost čísel $1, 2, \ldots, N$, ve které se každé vyskytuje právě jednou. Hrošík by po vás chtěl, abyste mu vypsali všechny možné trasy, které má, každou právě jednou. Navíc by si ale přál, aby se dvě trasy vypsané hned po sobě lišily jenom prohozením jedné dvojice sousedů (čili aby se posloupnosti reprezentující trasy shodovaly ve všech prvcích kromě dvou, které jsou prohozené). První a poslední vypsané trasy se mohou libovolně lišit.

Bonus: Pokud se bude i první a poslední trasa lišit právě prohozením jedné dvojice prvků, dostanete bonus *3 body*.

Příklad: Pro $N = 3$ je jedním ze správných výstupů (i pro bonusovou úlohu):

1 2 3
2 1 3
3 1 2
3 2 1
2 3 1
1 3 2

17-5-4 Kudy tudy cestička — 11 bodů

Zatímco si hrošík vybíral nejkratší trasu, uběhl skoro celý rok. A tak den před Velikonocemi hrošík zjistil, že už půjde zítra koledovat, a přitom neví, jakou trasou vlastně půjde.

Protože je hrošík váš kamarád (nebo snad proto, že nemáte rádi zajíčky?), určitě mu rádi poradíte, tentokrát trochu konkrétněji než v minulé úloze.

Hrošík chce opět navštívit N svých sousedů. Tyto sousedy si můžete představit jako body v rovině. Navíc pokud si všechny tyto body představíte jako vrcholy N-úhelníku, platí, že tento N-úhelník je konvexní (to znamená, že všechny jeho vnitřní úhly mají velikost menší než 180°).

Mezi některými dvojicemi sousedů vedou pěšinky, každá je nějak dlouhá. Všechny pěšinky jsou úsečky, každá spojuje dané dva body odpovídající sousedům, mezi kterými vede. Různé úsečky se samozřejmě mohou křížit.

Hrošík by chtěl takovou trasu, která začíná u libovolného souseda, bude se skládat jenom z daných pěšinek a navštíví každého souseda právě jednou. (To znamená, že na své trase použije právě $N - 1$ pěšinek.) Navíc se žádné dvě pěšinky, které jsou v trase použity, nesmí křížit. A aby mohl být hrošík lepší koledník než Velkovezír, vámi nalezená trasa by měla být nejkratší možná.

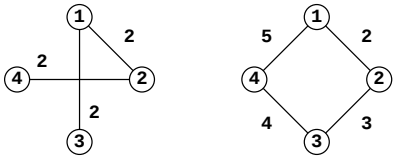
Váš program tedy dostane na vstupu N, dále souřadnice N bodů v rovině, které tvoří konvexní mnohoúhelník, a dále seznam M pěšinek, každá vede mezi jinou dvojicí sousedů a má nějakou délku. Všechny pěšinky jsou obousměrné a jsou to úsečky spojující body odpovídající sousedům, mezi kterými pěšina vede. Vaším úkolem je najít takovou nejkratší trasu, která navštíví každého souseda právě jednou a žádné dvě z pěšinek této trasy se nekříží. Pokud je jich víc, vypište libovolnou z nich.

Příklad: Pro 4 body $(0,1),(1,0),(0,-1),(-1,0)$ a pěšinky $P1$ žádná hledaná trasa neexistuje. Pro pěšinky $P2$ existuje, je to trasa $4,3,2,1$.

$P1$		
odkud	*kam*	*délka*
1	2	2
1	3	2
2	4	2

$P2$		
odkud	*kam*	*délka*
1	2	2
2	3	3
3	4	4
4	1	5

Pro představu následuje obrázek obou popsaných případů:



17-5-5 Jazykozpytec se loučí — 10 bodů

V posledním díle našeho automatově-gramatického seriálu jsme si slíbili povědět něco o dalších typech gramatik. (Asi se bude hodit připomenout si, co je to gramatika. Přesnou definici, komentáře a příklady čtenář najde v prvním díle seriálu.)

Bezkontextová gramatika je gramatika skládající se pouze z pravidel tvaru

$$X \to \alpha,$$

kde

- $X \in V_N$ je neterminál,
- $\alpha \in (V_T \cup V_N)^*$ je nějaká konečná posloupnost terminálních či neterminálních symbolů.

Jak vidíme, oproti gramatikám popisujícím regulární jazyky (připomeňme, že ty obsahují pouze pravidla $X \to wY$ nebo $X \to w$) je pravá strana pravidel poněkud volnější.

Příklad: Jazyk všech palindromů (množinu všech slov, co se čtou pozpátku stejně jako zepředu) nad abecedou $\{a, b\}$ popisuje následující jednoduchá bezkontextová gramatika (V_N, V_T, S, P). Jediný neterminální symbol $V_N = \{S\}$ je zároveň počáteční, terminální symboly jsou $V_T = \{a, b\}$ a přepisovací pravidla P jsou

$$S \to \lambda \mid a \mid b \mid aSa \mid bSb.$$

Například slovo *babab* dostaneme posloupností přepisů

$$S \to bSb \to baSab \to babab.$$

Pokud vás mate název „bezkontextové gramatiky“, pak věžte, že existují ještě tzv. *kontextové gramatiky,* které obsahují pouze pravidla tvaru

$$\alpha X \beta \to \alpha \gamma \beta,$$

kde

- $X \in V_N$ je neterminál,
- $\alpha, \beta \in (V_T \cup V_N)^*$ jsou libovolné konečné posloupnosti, klidně prázdné, terminálních či neterminálních symbolů,
- $\gamma \in (V_T \cup V_N)^+$ je libovolná posloupnost terminálních či neterminálních symbolů s výjimkou prázdného slova λ.

Navíc ještě povolíme pravidlo $S \to \lambda$, pokud se S nevyskytuje na pravé straně žádného pravidla. Ačkoliv na první pohled vypadá docela chaoticky, že jsme zakázali všechna zkracující pravidla a právě toto jedno povolili, věžte, že je to opravdu potřeba, protože jinak by vznikla daleko obecnější třída jazyků, než chceme, nebo by naopak kontextové jazyky nebyly rozšířením bezkontextových nebo regulárních. O podrobnostech pro tentokrát pomlčíme.

Lidsky řečeno, kontextové gramatiky mohou pro rozexpandování symbolu X využít ještě informaci o tom, jakými symboly je právě obalen, tedy v jakém se nachází „kontextu“, a na základě tohoto kontextu provádět odlišné expanze.

Příklad: Ukážeme si gramatiku popisující jazyk $L = \{a^n b^n c^n;\ \forall n \in N, n > 0\}$. Gramatika $G = (V_N, V_T, S, P)$ bude používat neterminální symboly $V_N = \{S, B, C, X\}$, terminální symboly $V_T = \{a, b, c\}$ a množina přepisovacích pravidel P bude následující:

$$\begin{aligned} S &\to aSBC \mid abC && \ldots \text{namnož pomocné symboly} \\ CB &\to BC && \ldots \text{setřiď je (Pozor, podvod!)} \\ bB &\to bb && \ldots \text{a zruš všechny pomocné symboly} \\ bC &\to bc \\ cC &\to cc \end{aligned}$$

Všimněte si řádku, kde upozorňujeme na podvod. Toto pravidlo samozřejmě není kontextové. Namísto něj ve skutečnosti použijeme tři pravidla

$$\begin{aligned} CB &\to XB \\ XB &\to XC \\ XC &\to BC, \end{aligned}$$

o kterých už každý snadno vidí, že jsou kontextová a dělají to samé, co původní pravidlo. Slovo $aabbcc$ dostaneme například touto posloupností přepisů:

$$\begin{aligned} S \to aSBC \to aabCBC \to aabXBC \to aabXCC \to \\ \to aabBCC \to aabbCC \to aabbcC \to aabbcc \end{aligned}$$

S naší sadou pravidel zjevně bude všech symbolů a, b, c stejný počet a navíc jediná možnost, kdy se expandování gramatiky může zastavit, je, když jsou symboly utříděné. Gramatika G je tedy schopná vytvořit libovolné slovo z jazyka L a už nic dalšího navíc.

Soutěžní úloha 1: Sestrojte kontextovou gramatiku, která popisuje jazyk $L = \{a^i b^j c^k;\ 1 \le i \le j \le k\}$. Například slova $aabbcc$ či $abbccc$ do L patří, slova $aaabbc$ či cba do L nepatří. *[5 bodů]*

Soutěžní úloha 2: Sestrojte kontextovou gramatiku, která popisuje jazyk $L = \{a^{2^n};\ \forall n \in N\}$, čili jazyk všech slov ze symbolů a, jejichž počet je mocninou dvojky. Tedy například slova a, aa a $aaaa$ do L patří, avšak slovo aaa do L nepatří. *[5 bodů]*

Dejte si zejména pozor na to, aby vámi sestrojená gramatika byla skutečně kontextová a nepoužívala nepovolená pravidla. Vaše řešení by měla obsahovat zdůvodnění, že gramatika dělá to, co má, případně důkaz, že hledáme marně a příslušná gramatika neexistuje.

O nedeterministických zásobníkových automatech, kterým byl věnován minulý díl seriálu, se dá dokázat, že rozpoznávají právě jazyky popsatelné bezkontextovou gramatikou. Důkaz je však poněkud obtížnější a my si ho předvádět nebudeme. Dají se zavést i podstatně mocnější výpočetní prostředky, než jsou zásobníkové automaty, například různé varianty tzv. *Turingových strojů*. U mnoha z nich se potom ukazuje souvislost s nejrůznějšími typy gramatik. Konkrétně kontextové gramatiky popisují právě jazyky, které jsou rozpoznatelné nedeterministickými Turingovými stroji v paměťovém prostoru omezeném lineárně vzhledem k délce vstupu.

V našem seriálu však již nezbývá dosti časoprostoru na to, abychom si tyto další stroje a gramatiky popsali, natož o nich ukázali něco zajímavého. Ještě bychom však rádi dodali, že teorie formálních jazyků je podkladovou teorií nejdůležitější informatické vědy – teorie složitosti. Rozhodovací algoritmické problémy se totiž dají převést na problém rozpoznávání určitého jazyka. Že se teorie gramatik hodí například při psaní překladačů programovacích jazyků, si čtenář jistě domyslí sám.

Tímto se tedy s vámi loučíme a děkujeme za pozornost, kterou jste formálním jazykům věnovali.

Programátorské kuchařky

16-1-K Kuchařka první série – dynamické programování

I v následujícím ročníku KSP vám kromě úloh budeme servírovat recepty z programátorské kuchařky. V první kuchařce nového ročníku si povíme něco o jedné z nejpoužívanějších programátorských technik, tzv. *dynamickém programování.* Dynamickým programováním rozumíme takový postup, kdy vyřešíme zadanou úlohu nejprve pro zadání menší velikosti a pak nalezená řešení zkombinujeme dohromady, abychom získali řešení původní úlohy. Techniku dynamického programování si předvedeme na dvou (učebnicových) příkladech.

První z našich dvou příkladů je úloha známá jako *problém batohu.* Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ a dále je dáno číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale zároveň nepřekročil M. My si popíšeme algoritmus, který tento problém řeší, s časovou složitostí $O(N \cdot M)$.

Náš algoritmus bude používat pomocné pole $A[0 \ldots M]$ a jeho činnost bude rozdělena do N kroků. Na konci k-tého kroku budou nenulové hodnoty v poli A právě na těch pozicích, které odpovídají součtu hmotností předmětů z nějaké podmnožiny prvních k předmětů. Před prvním krokem (po nultém kroku), jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1. Všimněme si, že kroky algoritmu odpovídají podúlohám, které řešíme: nejdříve vyřešíme podúlohu tvořenou jen prvním předmětem, pak prvními dvěma předměty, prvními třemi předměty, atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$ po $i = m_k$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k. Rozmysleme si, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů, pokud před jeho provedením nenulové hodnoty odpovídaly hmotnostem podmnožin z prvních $k-1$ předmětů. Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů) anebo se stala nenulovou v k-tém kroku. Potom ale existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$, ke které stačí přidat k-tý předmět, abychom našli podmnožinu hmotnosti přesně i. Naopak, pokud lze vytvořit

podmnožinu I hmotnosti m, pak I je buď tvořena jen prvními $k-1$ předměty, a tedy hodnota $A[m]$ je nenulová již před k-tým krokem, anebo $k \in I$. Potom ale hodnota $A[m-m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny $I \setminus \{k\}$ je $m-m_k$) a hodnota $A[m]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M. Nalézt jednu množinu této hmotnosti také není obtížné: v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové podmnožiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Samotný kód našeho algoritmu lze nalézt níže.

Časová složitost algoritmu je $O(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $O(M)$. Paměťová složitost činí $O(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

```
var N: word; {   počet předmětů   }
    M: word; { hmotnostní omezení }
    hmotnost: array[1..N] of word;
              { hmotnosti daných předmětů }
    A: array[0..M] of integer;
    i, k: word;
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then
        A[i]:=k;
  i:=M;
  while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
  write('Předměty v množině:');
  while A[i]<>-1 do
    begin
      write(' ',A[i]);
      i:=i-hmotnost[A[i]];
    end;
  writeln;
end.
```

Náš druhý příklad je z oblasti grafových algoritmů, tzv. Floyd-Warshallův algoritmus pro nalezení nejkratších cest mezi všemi vrcholy grafu. My se však pokusíme bez definice grafu jak v zadání, tak v řešení tohoto příkladu obejít.

Vstupem algoritmu jest N měst. Mezi některými dvojicemi měst vedou (obousměrné) silnice, jejichž délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky

nejkratší cest mezi všemi dvojicemi měst. Cestou rozumíme posloupnost měst spojených silnicemi a délkou cesty součet délek silnic, které spojují po sobě následující města.

Jsou sice známy i trošičku rychlejší způsoby řešící popsaný problém (umí se $O(N^2 \log N + N \cdot M)$), ale výhoda popisovaného algoritmu je v tom, že je velmi krátký a jednoduchý.

Na začátku jsou uloženy vzdálenosti mezi městy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$, v praxi tedy nějaké dostatečně velké číslo. V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty.

Samotný algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která může procházet skrz libovolné z měst $1, \ldots, k$. V průběhu k-té fáze tedy stačí vyzkoušet, zda mezi městy i a j je kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo nová cesta přes město k. Pokud nejkratší cesta prochází přes město k, pak její část do města k je nejkratší cesta z i do k přes města $1, \ldots, k-1$ a její část z města k je nejkratší cesta z k do j přes města $1, \ldots, k-1$. Délka takové cesty je tedy rovna $D[i][k]+D[k][j]$. Pokud je tedy součet $D[i][k]+D[k][j]$ menší než stávající hodnota $D[i][j]$, nahradíme hodnotu na pozici $D[i][j]$ tímto součtem.

Z popisu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j. Každá z N fází algoritmu vyžaduje čas $O(N^2)$, takže celková časová složitost bude $O(N^3)$. Paměťová složitost algoritmu je $O(N^2)$. Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i samotné nejkratší cesty. To lze jednoduše vyřešit například tak, že si budeme udržovat další pomocné pole $E[i][j]$, do kterého při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k). Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

```
var N:word; { počet měst }
    D:array[1..N] of array[1..N] of longint;
      { délky silnic mezi městy, D[i][i]=0,
        místo neexistujících je "nekonečno" }
    i,j,k:word;
begin
  for k:=1 to N do
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
        D[i][j]:=D[i][k] + D[k][j];
end.
```

Na rozmyšlenou:

- Jak byste algoritmus modifikovali, kdyby silnice byly jednosměrné?
- Nastavit ∞ na `maxint` je sice lákavé, ale špatně, protože $\infty + \infty$ by pak mohlo přetéci. Pomůže `maxint div 2`.
- Hodnoty v poli si sice přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně.
- Popis algoritmu vysloveně svádí k „rejpnutí“: „Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)?“ Inu, to samozřejmě nevíme, ale všimněte si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá ... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. (Což, pokud bychom chtěli být přesní, musíme přidat do předpokladů našeho algoritmu.)
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní ... jenže pak samozřejmě nebude fungovat.

16-2-K Kuchařka druhé série – hešování

V tomto dílu programátorské kuchařky si povíme něco o hešování. (V literatuře se také často setkáme s jinými přepisy tohoto anglicko-českého patvaru (hashování), či více či méně zdařilými pokusy se tomuto slovu zcela vyhnout a místo „heš“ používat například termín asociativní pole.) Na heš se můžeme dívat jako na pole, které ale neindexujeme po sobě následujícími přirozenými čísly, ale hodnotami nějakého jiného typu (řetězci, velkými čísly, apod.). Hodnotě, kterou heš indexujeme, budeme říkat *klíč*. K čemu nám takové pole může být dobré?

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.

- Potřebujeme najít v seznamu objekty, které jsou „stejné“ podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

Potřebovali bychom tedy umět do heše přidávat nové hodnoty, najít hodnotu pro zadaný klíč a případně také umět z heše nějakou hodnotu smazat.

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

Ideální případ by nastal, kdyby se nám podařilo nalézt funkci, která by každým dvěma klíčům přiřazovala různou hodnotu (i to se může podařit, pokud množinu klíčů, které v heši budou, známe dopředu – viz třeba příklad s rozpoznáváním klíčových slov v překladačích). Pak nám stačí použít jednoduché pole velikosti K, jehož prvky budou obsahovat jednak hodnotu klíče, jednak jemu přiřazená data:

```
struct položka_heše
{
  int obsazeno;
  typ_klíče klíč;
  typ_hodnoty hodnota;
} heš[K];
```

A operace naprogramujeme zřejmým způsobem:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Kolize nejsou, čili heš[index].obsazeno=0.
  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}
```

```
int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  // Nic tu není nebo je tu něco jiného.
  if (!heš[index].obsazeno ||
      !stejný (klíč, heš[index].hodnota))
    return 0;

  // Našel jsem.
  *hodnota = heš[index].hodnota;
  return 1;
}
```

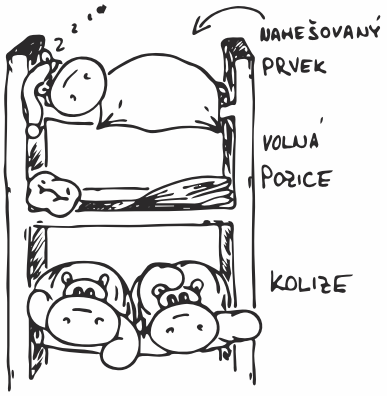
Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

Jiná možnost je v případě kolize uložit kolidující hodnotu na první následující volné místo v poli (cyklicky, tj. dojdeme-li ke konci pole, pokračujeme na začátku). Samozřejmě pak musíme i příslušně upravit hledání – snadno si rozmyslíme, že musíme projít všechny položky od pozice, kterou nám poradí hešovací funkce, až po první nepoužitou položku. Tento přístup se obvykle nazývá *hešování se srůstajícími řetězci* (protože seznamy hodnot odpovídající různým hodnotám hešovací funkce se nám mohou spojit). Implementace pak vypadá takto:

```
void přidej (typ_klíče klíč,typ_hodnoty hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    index++;
    if (index == K)
      index = 0;
  }

  heš[index].obsazeno = 1;
  heš[index].klíč = klíč;
  heš[index].hodnota = hodnota;
}
```

```
int najdi (typ_klíče klíč, typ_hodnoty *hodnota)
{
  unsigned index = hešovací_funkce (klíč);

  while (heš[index].obsazeno)
  {
    if (stejný (klíč, heš[index].klíč))
    {
      *hodnota = heš[index].hodnota;
      return 1;
    }

    // Něco tu je, ale ne to, co hledám.
    index++;
    if (index == K)
      index = 0;
  }

  // Nic tu není.
  return 0;
}
```



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Pak hledání může přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase).

Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T+H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

Mazání prvků může působit menší problémy (rozmyslete si, proč nelze prostě nastavit u mazaného prvku „obsazeno“ na 0). Pokud to potřebujeme dělat, buď musíme použít separované řetězce (což se může hodit i z jiných důvodů, ale je o trošku pracnější), nebo použijeme následující fígl: když budeme nějaký prvek mazat, najdeme ho a označíme jako smazaný. Nicméně při hledání nějakého jiného prvku se nemůžeme zastavit na tomto smazaném prvku, ale musíme hledat i za ním. Ovšem pokud nějaký prvek přidáváme, můžeme jím smazaný prvek přepsat.

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory

pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slijí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

```
#define mix(a,b,c) {                          \
  a-=b; a-=c; a^=(c>>13);                     \
  b-=c; b-=a; b^=(a<< 8);                     \
  c-=a; c-=b; c^=((b&0xffffffff)>>13);        \
  a-=b; a-=c; a^=((c&0xffffffff)>>12);        \
  b-=c; b-=a; b =(b ^ (a<<16)) & 0xffffffff;  \
  c-=a; c-=b; c =(c ^ (b>> 5)) & 0xffffffff;  \
  a-=b; a-=c; a =(a ^ (c>> 3)) & 0xffffffff;  \
  b-=c; b-=a; b =(b ^ (a<<10)) & 0xffffffff;  \
  c-=a; c-=b; c =(c ^ (b>>15)) & 0xffffffff;  \
}
```

My se ale spokojíme s málem a ukážeme si jednoduchý způsob, jak hešovat čísla a řetězce. Pro čísla stačí zvolit za velikost tabulky vhodné prvočíslo a klíč vymodulit tímto prvočíslem. (S hledáním prvočísel si samozřejmě nemusíme dělat starosti, v praxi dobře poslouží tabulka několika prvočísel přímo uvedená v programu.)

Rozumná funkce pro hešování řetězců je třeba

```
unsigned hash_string (unsigned char *str)
{
  unsigned r = 0;
  unsigned char c;

  while ((c = *str++) != 0)
    r = r * 67 + c - 113;

  return r;
}
```

Zde můžeme použít vcelku libovolnou velikost tabulky, která nebude dělitelná čísly 67 a 113. Šikovné je vybrat si například mocninu dvojky (což v příštím odstavci oceníme), ta bude s prvočísly 67 a 113 zaručeně nesoudělná. Jen si musíme dávat pozor, abychom nepoužili tak velkou hešovací tabulku, že by 67 umocněno na obvyklou délku řetězce bylo menší než velikost tabulky (čili by hešovací funkce častěji volila začátek heše než konec). Tehdy ale stačí místo našich čísel použít jiná, větší prvočísla.

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš

dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

To na první pohled vypadá velice neefektivně, ale protože se po každém nafouknutí heš zvětší na dvojnásobek, musí mezi přehešováním na N prvků a na $2N$ přibýt alespoň N prvků, čili průměrně provádíme jedno přehešování na každý vložený prvek.

Pokud navíc používáme mazání prvků popsané výše (u prvku si pamatujeme, že je smazaný, ale stále zabírá místo v heši), nemůžeme při mazání takového prvku snížit počet prvků v heši, ale na druhou stranu při nafukování můžeme takové prvky opravdu smazat (a konečně je odečíst z počtu obsazených prvků).

Pár poznámek na závěr:

- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé přihrádce (přihrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.
- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nový heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

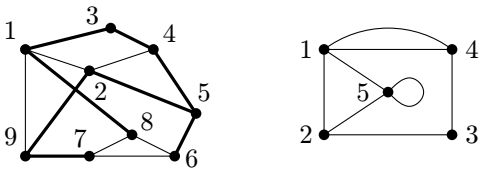
16-3-K Kuchařka třetí série – grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované, ba i rovinné. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran, což jsou neuspořádané dvojice vrcholů. Hrana $e = x, y$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám

říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a jeho kostra; multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* je posloupnost vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, obsahuje nějaký vrchol u dvakrát, nechť $u = v_i = v_j$, $i < j$. Z takového sledu ale můžeme vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

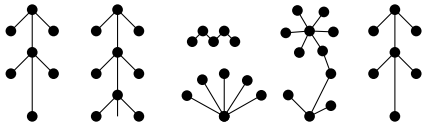
Kružnicí nazýváme cestu délky alespoň 3, ve které oproti definici platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé*. Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné hrany. Takové podgrafy nazýváme *komponentami souvislosti*.

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



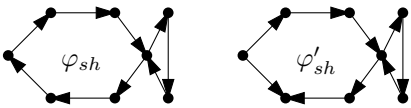
Les, jak ho vidí matematici

Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale grafoví teoretici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu je strom, který spojuje všechny vrcholy. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je kostra levého grafu znázorněna silnými hranami.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf.* Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany (i když se mohou vyskytovat v grafu obě najednou). Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, platí i pro grafy orientované, jen si musíme dát pozor na směr hran. Kružnici v orientovaném grafu často nazýváme *cyklem*.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme *slabou a silnou souvislost.* Slabě souvislý je graf tehdy, pokud když zapomeneme na orientaci hran, dostaneme souvislý orientovaný graf. Silně souvislým ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y ($x \neq y$) orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. Nejčastější jsou tyto dva způsoby:

- *matice sousednosti* – to je pole A velikosti $N \times N$ (kde N je počet vrcholů). Na pozici $A[i,j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

```
011000001
100110001
100100000
011010000
010101000
000010110
000001011
100001100
110000100
```

- *seznam sousedů* se obvykle zapisuje dvěma poli: polem hran E, do kterého uložíme všechny hrany tak, aby hrany vedoucí z jednoho vrcholu tvořily souvislý úsek, a polem vrcholů V, které pro každý vrchol udává začátek odpovídajícího úseku v poli E. Pokud do $V[N+1]$ uložíme $M+1$, kde M je počet hran, platí, že hrany vycházející z vrcholu i jsou uloženy v $E[V[i]], \ldots, E[V[i+1]-1]$. Tato reprezentace má tu výhodu, že má velikost pouze $O(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$E[i].a$	1	1	1	1	2	2	2	2	3	3	4	4	4	5	5	5	6	6	6	7	7	7	8	8	8	9	9	9
$E[i].b$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$V[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

Recepty

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale prvky přidáváme a odebíráme z konce zásobníku. Anglický název je (překvapivě) *last in, last out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme na zásobník a označíme.
4. Body 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož v bodě 2 pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. V bodu 3 probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární v počtu vrcholů N a počtu hran M – $O(N + M)$. Paměťová složitost je stejná, protože si musíme hrany a vrcholy pamatovat.

Nejjednodušší implementace prohledávání do hloubky je rekurzivní funkcí. Jako zásobník v tom případě požíváme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Oznacen: array[1..MaxN] of Boolean;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V]:= True;
  for I:= Hrany[V] to Hrany[V + 1]-1 do
    if not Oznacen[Sousedi[I]] then
      Rekurze(Sousedi[I]);
end;
```

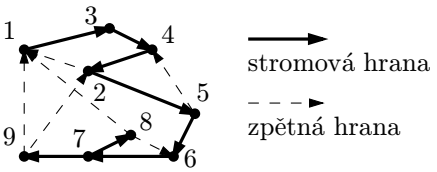
Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $O(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady složitost $O(N + M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost také $O(N + M)$.

```
var Komponenta: array[1..MaxN] of Integer;
  Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  NovaKomponenta: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V]:= NovaKomponenta;
  for I:= Hrany[V] to Hrany[V + 1]-1 do
    if (Komponenta[Sousedi[I]] = -1) then
      Rekurze(Sousedi[I]);
end;
```

```
var I: Integer;
begin
   for I:= 1 to N do Komponenta[I]:= -1;
   NovaKomponenta:= 1;
   for I:= 1 to N do
      if Komponenta[I] = -1 then
         begin
            Projdi(I);
            Inc(NovaKomponenta);
         end;
end.
```

Průběh prohledávání grafu do hloubky můžeme znázornit stromem. Z počátečního vrchol w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslujeme jako syny vrcholů, ze kterých jsme přišli. Hranám mezi těmito vrcholy budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, vznikne nám strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, jsou tzv. *zpětné hrany. Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla „doprava“, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; „doleva“ rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale o té už víme, že neexistuje.



Strom prohledávání do hloubky

Prohledávání do hloubky lze tedy využít na hledání kostry neorientovaného grafu, což je strom, který jsme prošli, a rovnou při tom také zjistíme, zda graf neobsahuje cyklus, což je v případě, kdy nalezneme zpětnou hranu různou od té stromové, kterou jsme do vrcholu přišli.

Pro orientované grafy je opět situace složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou „zprava doleva“, tedy pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).

Prohledávání do šířky

Prohledávání do šířky je založené na trochu jiné myšlence a na rozdíl od prohledávání do hloubky používá jinou datovou strukturu, a to frontu.

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Body 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n+1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do x. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a předposlední vrchol z této cesty. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, což je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $O(N + M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli, které nám bude představovat frontu.

```
var Fronta, Delka: array[1..MaxN] of Integer;
  Oznacen: array[1..MaxN] of Boolean;
  Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  I, Prvni, Posledni, PocatecniVrchol: Integer;

begin
  Prvni:= 1;
  Posledni:= 1;
  Fronta[Prvni]:= PocatecniVrchol;
  Delka[Prvni]:= 0;

  repeat
  for I:= Hrany[Fronta[Prvni]] to
                Hrany[Fronta[Prvni]+1]-1 do
```

```
      if not Oznacen[Sousedi[I]] then begin
        Oznacen[Sousedi[I]]:= True;
        Delka[Sousedi[I]]:=Delka[Fronta[Prvni]]+1;
        Inc(Posledni);
        Fronta[Posledni]:= Sousedi[I];
      end;
    Inc(Prvni);
    until Prvni > Posledni;  {Fronta je prázdna}
  end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je to *topologické uspořádání*. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy pro každou hranu $e = (v_i, v_j)$ platí $i > j$. Představme si ho jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy tohoto cyklu v_1, $\ldots$, v_n, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než $v_2, \ldots, v_n$ než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což je spor.

Pokud graf cyklus neobsahuje, lze ho vždycky topologicky uspořádat. Zde je algoritmus:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana. Pokud žádný takový není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Nejprve si ukážeme, že neprázdný graf, který neobsahuje cyklus, vždy obsahuje vrchol, ze kterého nevede žádná hrana. Pro spor předpokládejme, že žádný takový vrchol neexistuje. Pak si vyberme libovolný vrchol v_1. Z něj vede hrana do dalších vrcholů, vybereme jeden z nich a označme ho v_2. Z v_2 vybereme další hranu a takto pokračujeme. Protože je vrcholů konečný počet, dospějeme k jednomu z těchto případů:

- Z některého vrcholu v_i nevede žádná hrana.
- Některé dva vrcholy v_i, v_j jsou stejné, a graf tedy obsahuje cyklus.

Což je spor s našimi předpoklady. Graf G má konečně mnoho vrcholů a protože v bodě 3 pokaždé odebereme další vrchol grafu, musí algoritmus skončit. Z vrcholu v, který přidáváme do posloupnosti, nevedou žádné hrany, a proto může mít nižší číslo než zbývající vrcholy grafu. To platí pro každý takový vrchol v, a proto je uspořádání korektní.

Algoritmus můžeme snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě, a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolujeme si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidáme takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $O(N + M)$.

Také můžeme graf prohledat do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud právě opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili vyšší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $O(N + M)$.

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Ocislovani: array[1..MaxN] of Integer;
  Posledni: Integer;
  I: Integer;
procedure Projdi(V: Integer);
var I: Integer;
begin
  for I:= Hrany[V] to Hrany[V+1]-1 do
    if Ocislovani[Sousedi[I]] = 0 then
      Projdi(Sousedi[I]);

  Inc(Posledni);
  Ocislovani[Vrchol]:= Posledni;
end;

begin
  ...
  for I:= 1 to N do
  Ocislovani[I]:= 0;

  Posledni:= 0;
  for I:= 1 to N do
  if Ocislovani[I] = 0 then Projdi(I);
  ...
end.
```

Hranová a vrcholová souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má 3 a více vrcholů,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most.*

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky. Pokud by v grafu nebyly žádné zpětné hrany, byla by mostem každá hrana – rozdělila by graf na část obsahující kořen a podstrom „visící“ pod touto hranou. Aby nevznikly dvě komponenty souvislosti, musí mezi těmito částmi vést další hrana (a může to být jedině zpětná hrana).

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejnižší hladiny vedou hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $O(N + M)$. Zde jsou důležité části programu:

```
var Hrany: array[1..MaxN + 1] of Integer;
  Sousedi: array[1..MaxM] of Integer;
  Hladina, Spojeno: array[1..MaxN] of Integer;
  DvaSouvisle: Boolean;
  I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I: Integer;
begin
  Hladina[V]:= NovaHladina;
  Spojeno[V]:= Hladina[V];

  for I:= Hrany[V] to Hrany[V + 1]-1 do
    if Hladina[Sousedi[I]] = -1 then
    begin
      Projdi(Sousedi[I], NovaHladina + 1);
      if Spojeno[Sousedi[I]] < Spojeno[V] then
        Spojeno[V]:= Spojeno[Sousedi[I]];
```

```
      if Spojeno[Sousedi[I]] > Hladina[V] then
        DvaSouvisle:= False;
    end else
    if Hladina[Sousedi[I]] < Spojeno[V] then
      Spojeno[V]:= Hladina[Sousedi[I]];
end;

begin
  for I:= 1 to N do Hladina[I]:= -1;

  DvaSouvisle:= True;
  Projdi(1, 0);
end.
```

Další formou souvislosti je *vrcholová souvislost*. Řekneme, že graf je *vrcholově 2-souvislý*, právě když:

- má 3 a více vrcholů,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

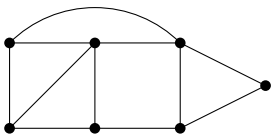
Artikulace je takový vrchol, který když odebereme, zvýší se nám počet komponent souvislosti.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

O rovinných grafech

Rovinný graf je graf, který můžeme nakreslit do roviny tak, že vrcholům přiřadíme vhodné body a hrany nakreslíme jako křivky spojující příslušné body, a to tak, že se žádné dvě křivky neprotínají mimo své krajní body. Ne každý graf takto nakreslit můžeme – sami si rozmyslete, že například graf K_5, což je 5 vrcholů spojených každý s každým, žádné rovinné nakreslení nemá. Na druhou stranu například každý strom určitě rovinný je.

Vezměme si tedy nějaký graf a jeho rovinné nakreslení, například tento:



Hrany nakreslení dělí rovinu na několik oblastí, těm budeme říkat *stěny.* Náš graf má 6 stěn: jednu čtvercovou, čtyři „trojúhelníkové“ (tedy ohraničené třemi hranami, byť to nejsou vždy úsečky) a jednu 6-úhelníkovou (to je celý zbytek roviny okolo grafu, tzv. *vnější stěna*). Například libovolné rovinné nakreslení stromu by mělo pouze jednu stěnu, a to tu vnější. Všimněte si, že pokud v grafu nejsou mosty ani artikulace, je každá stěna ohraničena nějakou kružnicí. [Pozor, to, jak vypadají stěny, závisí na konkrétním nakreslení do roviny!]

O rovinných grafech platí několik důležitých vět, které se často hodí při vytváření grafových algoritmů:

Věta: *(o počtu hran stromu)* Pro každý strom platí, že $e = v - 1$, kde v je počet vrcholů a e počet hran.

Důkaz: Indukcí podle počtu vrcholů. Pro strom s jedním vrcholem formulka určitě platí. Strom s $v > 1$ vrcholy má jistě list, tak jej odtrhneme [poněkud vandalské, nicméně účinné], čímž získáme strom s menším počtem vrcholů, pro který podle indukčního předpokladu formulka platí, a opětovným přidáním listu platit nepřestane, protože k oběma stranám přičteme jedničku.

Věta: *(Eulerova formule)* Pro každý souvislý graf nakreslený do roviny platí, že $v + f = e + 2$, kde v je počet vrcholů, e počet hran a f počet stěn.

Důkaz: Opět indukcí, tentokráte podle počtu hran. Každý souvislý graf má alespoň $v - 1$ hran a pokud jich má právě tolik, je to strom. (Kdyby ne, stačí se podívat na kostru grafu, což musí být strom a ty, jak už víme, mají právě tolik hran a náš graf měl hran více.) Jenže každé rovinné nakreslení stromu má právě jednu stěnu, takže Eulerova formule platí.

Pokud máme nakreslení grafu, který je souvislý a není to strom, znamená to, že obsahuje alespoň jednu kružnici. A každá hrana na kružnici jistě odděluje nějaké dvě stěny. Zvolme si tedy nějakou takovou hranu h a z grafu ji odeberme. Tím získáme graf s menším počtem hran (opět nakreslený do roviny), použijeme indukční předpoklad, Eulerova formule pro něj tedy již platí, a vrátíme hranu zpět. Levá strana rovnosti se tím zvětší o 1 (přidali jsme stěnu), pravá také (přidali jsme hranu), tedy rovnost stále platí.

Věta: *(o hustotě rovinných grafů)* O každém rovinném grafu platí, že $e \leq 3v - 6$.

Důkaz: Zvolme si libovolné nakreslení grafu do roviny. Nejprve předpokládejme, že je to triangulace, čili že každá stěna je trojúhelník. V takovém grafu patří každá hrana k právě dvěma trojúhelníkovým stěnám, takže $e = f \cdot 3/2$, čili $f = e \cdot 2/3$. Dosazením do Eulerovy formule získáme $v + (2/3)e = e + 2$, tedy $e = 3v - 6$.

Není-li náš graf triangulace, může to mít několik důvodů. Buďto není souvislý (pak ale stačí větu dokázat pro jednotlivé komponenty a nerovnosti sečíst), nebo je moc malý (má nejvýše dva vrcholy, proto to musí být jedna samotná

hrana a pro tu naše věta určitě platí) a nebo obsahuje nějakou stěnu ohraničenou více než třemi hranami. Dovnitř takové stěny ovšem můžeme dokreslit další hrany a tím ji rozdělit na trojúhelníčky. Tím tedy dokážeme graf doplnit hranami na triangulaci, pro tu, jak už víme, platí dokonce rovnost, a když přidané hrany opět odebereme, snížíme pouze počet hran a uděláme tak z rovnosti nerovnost.

Věta: *(o vrcholu nízkého stupně)* V každém rovinném grafu existuje vrchol stupně maximálně 5. (Stupeň vrcholu je počet hran, které s vrcholem sousedí.)

Důkaz: Sporem. Kdyby všechny vrcholy měly stupeň alespoň 6, byl by součet stupňů alespoň $6v$. Jenže součet stupňů je přesně dvojnásobek počtu hran (každá hrana má dva konce), takže $e \geq 3v$, což je spor s předchozí větou.

Poznámky na okraj:

- K čemu je to všechno dobré, zjistíte třeba v řešení úlohy 17-1-2.
- Kdybychom definici rovinného nakreslení změnili a dovolili hrany kreslit pouze jako úsečky místo libovolných křivek, překvapivě se nic nezmění: každý rovinný graf má rovinné nakreslení, v němž jsou všechny hrany úsečky. Ale není to zrovna jednoduché dokázat.
- Stejně jako do roviny bychom mohli grafy kreslit třeba na povrch koule. Tím se také nic nezmění, zkuste sami vymyslet, jak z rovinného nakreslení udělat „kulové“ a naopak. Ale třeba anuloid (povrch pneumatiky) se už chová jinak, například zmíněný nerovinný graf K_5 se na anuloid dá nakreslit bez křížení hran.
- Rovinné grafy, jejichž všechny vrcholy mají stupeň právě 5, opravdu existují, je to například graf odpovídající pravidelnému dvacetistěnu [má 12 vrcholů stupně 5 a 20 trojúhelníkových stěn]. V jistém smyslu je tedy naše poslední věta nejlepší možná.
- Více informací o teorii (nejen rovinných) grafů najdete třeba v knížce pánů Matouška a Nešetřila Kapitoly z diskrétní matematiky.

16-4-K Kuchařka čtvrté série – rozděl a panuj

Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj*. Myšlenka této metody je následující: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme počítat opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce“ v druhé sérii 16. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám od ní pak snadno budou odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
type Pole=array[1..MaxN] of Integer;      {budeme třídit takováto pole}

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin
                                {pivotem se stane poslední prvek úseku}
  x:=a[r];                                              {hodnota pivota}
  i:=l-1;                          {a[i] bude vždy poslední <= pivotovi}

  for j:=l to r-1 do                        {samotné přerovnávání      }
    if a[j]<=x then                         {právě probíraný prvek     }
    begin                                   {menší/rovný hodnotě pivota}
      Inc(i);                               {pak zvyš ukazatel         }
      q:=a[j];                              {a proveď přerovnání prvku }
      a[j]:=a[i];
      a[i]:=q;
    end;

  q:=a[r];                     {nakonec přesuneme pivota za poslední <=}
  a[r]:=a[i+1];
  a[i+1]:=q;
  prer:=i+1;                                {vrátíme novou pozici pivota}
end;

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
  if l<r then                                        {máme ještě co dělat?}
```

```
  begin
    m:=prer(l,r);                          {přerovnej, m pozice pivota}
    QuickSort(l,m-1);                            {setřiď prvky nalevo}
    QuickSort(m+1,r);                           {setřiď prvky napravo}
  end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které se to bude dít pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1 atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $O(N + (N-1) + (N-2) + \ldots + 1) = O(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $O(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $O(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\log_2 N$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\log_2 N$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $O(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak? Haf?
- *Spokojit se se „lžimediánem“:* kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti v prostřední polovině (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $O(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4) \cdot N$, takže na k-té hladině budou úseky délek $\leq (1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = O(\log N)$. Místo $1/4$ by dokonce fungovala libovolná jiná konstanta, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.

- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se to tak často dělá.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do loňského seriálu o pravděpodobnostních algoritmech). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $O(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián je to pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $O(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k - 1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, hledaný prvek se nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme

hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $O(N+N/2+N/4+\ldots+1)=O(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
  x:=prer(a,l,r);                      {přerovnej, x je pozice pivota}
  z:=x-l+1;                           {pozice pivota vzhledem k [l..r]}
  if k=z then
    kty:=a[x]                                  {k-tý nejmenší je pivot}
  else if k<z then
    kty:=kty(a,l,x-1,k)                       {k-tý nejmenší je nalevo}
  else
    kty:=kty(a,x+1,r,k-z);                                   {napravo}
end;
```

Hledání k-tého nejmenšího podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme InsertSortem (opět viz třídící kuchařka) a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku InsertSortem. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím

algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.

- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

Řečeno s panem Pascalem:

```
{potřebujeme přerovnávací funkci, která dostane}
{pozici pivota jako parametr}
function prerp(var a:Pole;l,r,m:Integer):Integer;
var q:Integer;
begin
  q:=a[m]; a[m]:=a[r]; a[r]:=q;   {pivota prohodíme s posledním prvkem}
  prerp := prer(a,l,r);           {a zavoláme původní přerovnávací fci}
end;

{hledání k-tého nejmenšího prvku z a[l..r],}
{vracíme pozici prvku, nikoliv jeho hodnotu}
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;                                  {pole pro mediány pětic}
    i,j,q,x,pocet,m,z:Integer;
begin
  pocet:=r-l+1;                               {s kolika prvky pracujeme}

  if pocet<=1 then                                  {pouze jeden prvek?}
    kth:=l                                {výsledek ani nemůže být jiný}
  else if pocet<6 then begin                          {méně než 6 prvků}
    for j:=l+1 to r do begin                             {=> InsertSort}
      q:=a[j];
      i:=j-1;
      while (i>=l) and (a[i]>q) do begin
        a[i+1]:=a[i];
        Dec(i);
      end;
      a[i+1]:=q;
    end;
  kth:=l+k;
  end
  else begin                                {mnoho prvků, jde to tuhého}
    {rozdělíme prvky do pětic}
    q:=1;                                      {zatím máme jednu pětici}
    i:=l;                                      {levý okraj první pětice}
    j:=i+4;                                   {pravý okraj první pětice}
    while j<=r do begin                         {procházíme celé pětice}
      medp[q]:=kth(a,i,j,2);                              {medián pětice}
      Inc(q);                                           {zvyš počet pětic}
      Inc(i,5);                                {nastav levý okraj pětice}
```

```
      Inc(j,5);                                {nastav pravý okraj pětice}
    end;
  if i<=r then begin                              {zbyla neúplná pětice}
    medp[q]:=kth(a,i,r,(r-i+2) div 2);
    Inc(q);
  end;

  {najdeme medián mediánů pětic, je na pozici m}
  m:=kth(medp,1,q-1,q div 2);

  {přerovnej a zjisti, kde skončil pivot}
  x:=prer(a,l,r,m);
  z:=x-l+1;                                  {pozice vzhledem k [l..r]}
  if k=z then
    kth:=m                                     {k-tý nejmenší je pivot}
  else if k<z then
    kth:=kth(a,l,x-1,k)                          {k-tý nejmenší nalevo}
  else
    kth:=kth(a,x+1,r,k-z);                                   {napravo}
 end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze opravdu má lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za mediánem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí například pro $d = 10c$, takže opravdu $t(N) = O(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – my volíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo můžeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A + B) \cdot (C + D) = AC + AD + BC + BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) =$ nějaká konstanta d. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = O((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$. Konstanta d se nám „schová do O-čka“, takže algoritmus má časovou složitost přibližně $O(n^{1.58})$. Umí se to i lépe – $O(n \log n)$, ale to je mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou pěti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $O(\log N)$. Žádný div: stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

16-5-K Kuchařka páté série – rekurze, dynamické programování II

V dnešní kuchařce se budeme zabývat převážně rekurzí a dynamickým programováním. Čemu tedy říkáme rekurze? Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe. Pojďme se na jednoduchém příkladě podívat, jak může taková funkce vypadat.

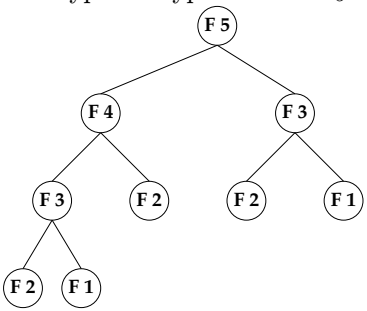
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

$$1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad \ldots$$

Pro nalezení n-tého členu (ten budeme značit F_n) si napíšeme rekurzivní funkci `Fibonacci(n)`, která bude postupovat přesně podle definice: zeptá se sama sebe rekurzivně, jaká jsou dvě předchozí čísla, a pak je sečte. Možná více řekne program:

```
function Fibonacci(n: Integer): Integer;
begin
  if n <= 2 then
    Fibonacci := 1
  else
    Fibonacci := Fibonacci(n-1) + Fibonacci(n-2)
end;
```

Podívejme se, jak bude vypadat výpočet čísla F_5:



Vidíme, že volání funkce se rozvětvuje a tvoří strom volání. Všimněme si také, že některé podstromy jsou shodné. Zřejmě to budou ty části, které reprezentují výpočet stejného Fibonacciho čísla – v našem příkladě třeba třetího.

Pokusme se odhadnout časovou složitost T_n naší funkce. Pro $n = 1$ a $n = 2$ funkce skončí hned, tedy v konstantním (řekněme jednotkovém) čase. Pro vyšší n zavolá sama sebe pro dva předchozí členy plus ještě spotřebuje konstantní čas na sčítání:

$$T_n \geq T_{n-1} + T_{n-2} + const, \text{ a proto } T_n \geq F_n.$$

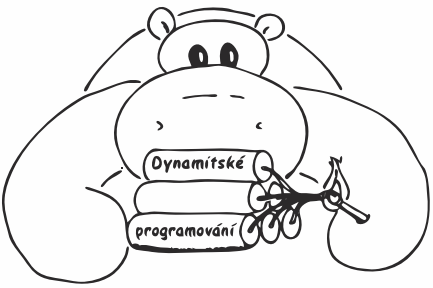
Tedy na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme využít toho, že:

$$F_n = F_{n-1} + F_{n-2} \geq 2 \cdot F_{n-2},$$

z čehož plyne:

$$F_n \geq 2^{n/2}.$$

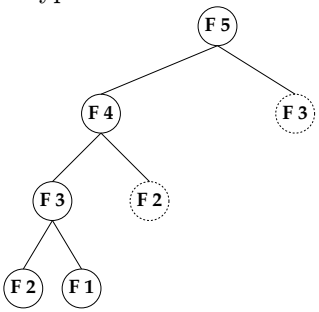
Funkce `Fibonacci` má tedy exponenciální časovou složitost, což není nic vítaného. Ovšem jak už jsme řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka (Velkovezíra?) z klobouku s minimem námahy.



Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

```
var P: array[1..MaxN] of Integer;
function Fibonacci(n: Integer): Integer;
begin
  if P[n] = 0 then
  begin
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```

Podívejme se, jak nyní vypadá strom volání:



Na každý člen posloupnosti se tentokrát ptáme maximálně dvakrát – k výpočtu ho potřebují dva následující členy. To ale znamená, že funkci `Fibonacci` zavoláme maximálně $2n$-krát, čili jsme touto jednoduchou úpravou změnili exponenciální složitost na lineární.

Zdálo by se, že abychom získali čas, museli jsme obětovat paměť, ale to není tak úplně pravda. V prvním příkladu sice nepoužíváme žádné pole, ovšem při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, na což spotřebujeme paměť lineární s hloubkou vnoření, v našem případě tedy lineární s n.

Určitě vás už také napadlo, že n-té Fibonacciho číslo se dá snadno spočítat i bez rekurze. Stačí prvky našeho pole P plnit od začátku – kdykoliv známe $P[1] = F_1, \ldots, F_k = P[k]$, dokážeme snadno spočítat i $P[k+1] = F_{k+1}$:

```
function Fibonacci(n: Integer): Integer;
var
  P: array[1..MaxN] of Integer;
  I: Integer;
```

```
begin
  P[1] := 1;
  P[2] := 1;
  for I := 3 to n do
    P[I] := P[I-1] + P[I-2];
  Fibonacci := P[n];
end;
```

Zopakujme si, co jsme postupně udělali: nejprve jsme vymysleli pomalou rekurzivní funkci, tu jsme zrychlili zapamatováváním si mezivýsledků a nakonec jsme celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

V případě Fibonacciho čísel je samozřejmě snadné přijít rovnou na nerekurzivní řešení (a dokonce si všimnout, že si stačí pamatovat jen poslední dvě hodnoty a paměťovou složitost tak zredukovat na konstantní), ale zmíněný obecný postup – obvykle se mu říká *dynamické programování* – funguje i pro řadu složitějších úloh. Třeba na tuto:

Mějme dvě posloupnosti čísel A a B. Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$\begin{array}{lllllllllllllll} A & = & 2 & 3 & 3 & 1 & 2 & 3 & 2 & 2 & 3 & 1 & 1 & 2 & \\ B & = & 3 & 2 & 2 & 1 & 3 & 1 & 2 & 2 & 3 & 3 & 1 & 2 & 2 & 3 \end{array}$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C \;=\; 2 \;\; 3 \;\; 1 \;\; 2 \;\; 2 \;\; 3 \;\; 1 \;\; 2$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat. Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká. Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem. Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost, takže pokud

známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich pamatovat pouze P, jelikož v libovolném rozšíření Q-čka můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě, a dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l+1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný: Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a+1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B. Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti. Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněte si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP). Ukážeme si to na našem příkladu: jelikož

poslední nenulové číslo na posledním řádku je v 8. sloupci, má hledaná NSP délku 8. $D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[11, 7]$, třetí z $D[9, 6]$, atd. Jednou z hledaných podposloupností je:

poslupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, T: Integer;
begin
  ...
  if LA > LB then  { A bude kratší z obou }
  begin
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
  end;

  for I := 1 to LA do
    D[0, I] := LB;

  L := 0;
  for I := 1 to LA do
  begin
    for J := 1 to LA do
      D[I, J] := D[I-1, J];

    L := 1;
    for J := 0 to LB-1 do
      if B[J] = A[I-1] then
      begin
        while D[I-1, L] < J do Inc(L);
        if D[I, L] >= J then
          D[I, L] := J;
      end;
  end;

  LC := L;
  J := LA;
```

```
    for I := LC downto 1 do
    begin
      while D[J-1, I] = D[J, I] do Dec(J);
      C[I-1] := A[J-1];
      Dec(J);
    end;
    ...
end.
```

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $L(A)$ a $L(B)$, což jsou délky posloupností A a B. Vnořený cyklus while proběhne celkem maximálně $L(A)$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $O(L(A) \cdot L(B))$. Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti a tedy i velikost pole s daty je kvadrát této délky. Paměťovou složitost odhadneme $O(N^2 + M)$, kde N je délka kratší posloupnosti.

Vzorová řešení

17-1-1 Výdělek bratří Součků — Pavel Čížek

Řešení této úlohy by se dala rozdělit do tří skupin. Lineární, kvadratické (vůči délce vstupu) a nefunkční. Kromě toho několik lidí předpokládalo, že řetězce mohou být vůči sobě posunuté. Nechápu proč. Bratři byli vyslání na koncert společně a začali zapisovat oba hned na začátku.

No a jak mělo řešení vypadat? Nejprve je třeba si uvědomit, že pokud mají být řetězce záznamem toho samého, pak musí obsahovat stejný podřetězec, jehož opakováním vytvoříme Ainův i Kábelův záznam. No a jak dlouhý tento podřetězec může být? Jelikož jeho několikanásobným zopakováním musíme dostat jak Ainův, tak Kábelův záznam, musí jeho délka být největší společný dělitel délek záznamů Aina a Kábela, resp. NSD musí být jeho celočíselným násobkem.

Pokud bude tedy Ainův i Kábelův záznam složen z opakujícího se podřetězce délky NSD a tyto podřetězce budou u Aina i Kábela stejné, víme, že zápisy budou stejné. A pokud podmínka splněná nebude, zaznamenali oba něco jiného.

Algoritmus řešící úlohu je jenom přímočarou implementací popsaného postupu, jeho časová i paměťová složitost je lineární vůči délce zápisů obou bratří.

Toto mělo být řešení této úlohy, ale při pročítání řešení účastníků jsem nalezl jedno výrazně elegantnější:

```
var Ain,Kabel:string;
begin
    write('Ain:');readln(Ain);
    write('Kabel:');readln(Kabel);
    if (Ain+Kabel = Kabel+Ain) then
      writeln('Záznamy jsou stejné.')
   else
      writeln('Záznamy jsou různé.');
end.
```

A proč toto funguje? BÚNO (bez újmy na obecnosti) předpokládejme, že Kábelův záznam (označme jej K a jeho délku L_K) je delší než Ainův (A, L_A). Záznamy budeme indexovat od nuly. Zřejmě ono porovnání vypadá takto:

$$\begin{aligned} &A[0]\ A[1]\ \ldots\ A[L_A-1] \quad K[0]\ \ldots\ K[L_K-L_A-1]\ K[L_K-L_A]\ \ldots\ K[L_K-1]= \\ =&K[0]\ K[1]\ \ldots\ K[L_A-1]\ K[L_A]\ \ldots\ K[L_K-1] \qquad A[0] \qquad\qquad \ldots\ A[L_A-1] \end{aligned}$$

Z porovnání těchto řádků dostaneme jednoduše:

$$K[i] = K[i \bmod L_A] \tag{1}$$

$$A[i] = K[i] \tag{2}$$

$$A[i] = K[L_K - L_A + i] \tag{3}$$

Zkombinováním (1) a (3):

$$A[i] = K[(L_K + i) \bmod L_A]$$

A po uvažování (2):

$$A[i] = A[(i + (L_K \bmod L_A)) \bmod L_A]$$

Iterováním této rovnosti pro libovolné celé c:

$$A[i] = A[(i + (c * (L_K \bmod L_A))) \bmod L_A]$$

Z toho je již vidět (po chvilce zamyšlení), že:

$$A[i] = A[i \bmod \mathrm{NSD}(L_A, L_K)] = K[i]$$

a tedy rovnost je splněna, právě když jsou záznamy stejné.

```
var Ain,Kabel : string;
    Perioda   : longint;
    index     : longint;
    Shodne    : boolean;

function NSD(a,b:longint):longint;
begin
   while (a<>0) and (b<>0) do
      if (a>b) then
         a:=a mod b
      else
         b:=b mod a;
   NSD:=a+b;
end;

begin
   write('Ain:');readln(Ain);
   write('Kabel:');readln(Kabel);
   Perioda:=NSD(length(Ain),length(Kabel));
   Shodne:=true;

   for index:=1 to length(Ain) do        { generuje podřetězec délky NSD oba? }
      Shodne:=Shodne and (Ain[index]=Ain[((index-1) mod Perioda)+1]);
   for index:=1 to length(Kabel) do
      Shodne:=Shodne and (Kabel[index]=Kabel[((index-1) mod Perioda)+1]);
   for index:=1 to Perioda do            { jsou generující podřetězce shodné? }
      Shodne:=Shodne and (Ain[index]=Kabel[index]);
   if Shodne then
      writeln('Záznamy jsou stejné.')
   else
      writeln('Záznamy jsou různé.');
end.
```

17-1-2 Bůhdhova odměna — Martin Mareš

V této úloze nám nejde o nic jiného než o tak řečené *barvení grafu,* čili o přiřazení nějakých čísel (barev) $1, \ldots, k$ vrcholům grafu tak, aby žádné dva vrcholy spojené hranou nedostaly stejnou barvu. Zjistit, zda je graf nějakými k barvami obarvitelný, je obecně velmi těžký problém (pro obecné grafy a $k > 2$ barev ho nikdo neumí vyřešit v polynomiálním čase; případy $k = 1$ a $k = 2$ si rozmyslete sami, ty jsou pro změnu triviální). Ale my o našem grafu naštěstí víme, že je rovinný (viz povídání v tomto vydání naši kuchařky), což situaci značně mění.

Každý rovinný graf lze obarvit čtyřmi barvami (to ale vůbec není triviální dokázat, matematikové se s tím trápili více než 100 let a nejkratší známý důkaz má přes sto stránek a rozebírá 633 různých případů). My si dokážeme, že vždy stačí 6 barev a že Bůhdha má tedy vždy šanci svou odměnu rozdělit:

Věta: *(o šesti barvách)* Každý rovinný graf je možno obarvit šesti barvami.

Důkaz: Indukcí podle počtu vrcholů. Pokud má graf nejvýše 6 vrcholů, obarvit ho dozajista můžeme. Pokud je graf větší, věta dokázaná v kuchařce nám říká, že v něm vždy existuje nějaký vrchol v stupně maximálně 5. Když takový vrchol odstraníme, dostaneme menší graf a ten je již podle indukčního předpokladu obarvitelný. Následně do obarveného grafu vrchol v vrátíme, a jelikož má nejvýše 5 sousedů, vždy je pro v alespoň jedna barva volná.

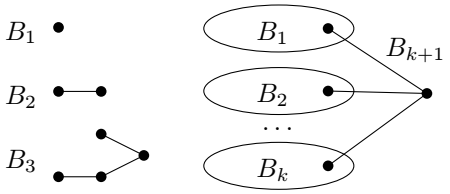
Již tento důkaz nám dává jednoduchý algoritmus pro Bůhdhův problém, ale ještě si musíme rozmyslet, jak neztrácet příliš mnoho času hledáním vrcholů nízkého stupně. Předem si spočítáme stupně všech vrcholů a do fronty uložíme ty vrcholy, které měly už na počátku stupeň ≤ 5. Poté postupně čteme vrcholy z fronty, snižujeme stupně jejich sousedům a jakmile některému sousedovi klesne stupeň pod 6, přidáme jej na konec fronty. Tím jsme vlastně sestrojili takové uspořádání vrcholů, že z každého vrcholu vede „doprava“ nejvýše 5 hran. Stačí tedy frontu projít pozpátku a postupně přidělovat volné barvy.

Tento algoritmus má lineární časovou i prostorovou složitost $O(N)$ (všimněte si, že jelikož je graf rovinný, má $O(N)$ hran). V programu stojí za zmínku snad jedině způsob reprezentace grafu: hrany máme uloženy v poli (každou dvakrát – v obou směrech), každý vrchol si pamatuje číslo první hrany z něj vycházející a každá hrana číslo následující hrany vycházející z téhož vrcholu. Při odtrhávání vrcholů hrany neodstraňujeme, pouze snižujeme stupně.

[P.S.: Po chvíli přemýšlení můžete náš důkaz upravit tak, aby ukazoval, že stačí 5 barev. Bůhdha se ale spokojí s šesti, takže mu nebudeme komplikovat život.]

Na závěr ještě ukážeme několik variant hladového barvícího algoritmu, který se často objevoval ve vašich řešeních a bohužel pro rovinné grafy nemůže fungovat.

Hladové barvení funguje takto: probíráme vrcholy jeden po druhém a každému přidělíme nejnižší barvu, která není použitá již obarvenými sousedy tohoto vrcholu. Zkusme si rozmyslet, jak obarví následující grafy:



Zde B_1 je graf z jednoho vrcholu a B_{k+1} zkonstruujeme tak, že vezmeme $B_1, \ldots, B_k$ a přidáme nový koncový vrchol v_k spojený s koncovými vrcholy všech B_i hranou. Vrcholy nového grafu uspořádáme tak, že nejdříve půjdou vrcholy grafů B_1, pak $B_2, \ldots, B_k$ a na konec umístíme nově vytvořený vrchol.

Všimneme si, že zadáme-li hladovému barvícímu algoritmu graf B_k, spotřebuje k barev a vrchol v_k obarví barvou k. I tentokrát nám k důkazu pomůže indukce: B_1 obarvíme jednou barvou, B_2 dvěma; pokud spustíme algoritmus na graf B_k, nejdříve obarví B_1 až B_{k-1}, a jelikož jsou v tom správném pořadí a nevedou mezi nimi žádné hrany, dopadne to stejně, jako bychom barvili každý zvlášť, čili podle indukčního předpokladu budou jejich koncové vrcholy obarveny barvami 1 až $k-1$, proto na zbývající vrchol v_k zbude barva k.

Jenže B_k je určitě rovinný graf, takže se dá obarvit šesti barvami (on je to dokonce strom, a tak stačí barvy dvě). Proto na něm pro $k > 6$ nemůže hladový barvící algoritmus fungovat.

Hladové barvení se ještě můžeme pokusit zachránit tím, že si zvolíme nějaké šikovné pořadí vrcholů (konec konců i náš správný barvící algoritmus je vlastně toho druhu). Ale jen tak ledajaké pořadí neposlouží:

- Pořadí podle rostoucích stupňů: v našem grafu pouze zparalelizuje barvení podgrafů $B_1, \ldots, B_k$ – jenže mezi nimi nevedou žádné hrany, takže výsledek musí vyjít stejný.
- Podle klesajících stupňů: stačí k vrcholům přivěsit spoustu listů (vrcholů stupně 1) a tím algoritmus donutit k takovému pořadí, jaké chceme (možná bude potřebovat ještě jednu barvu na listy, ale tím hůř pro něj).
- Podle prohledávání do šířky: přidáme každému B_i ještě počáteční vrchol w_i a při vytváření B_{k+1} připojíme nový počáteční vrchol w_{k+1} k počátečním vrcholům $w_1, \ldots, w_k$ cestami délek zvolených tak, aby koncové vrcholy všech B_i byly ve stejné vzdálenosti od w_{k+1}. Tehdy bude nový koncový vrchol v_{k+1} obarven až po všech ostatních vrcholech, čehož jsme přesně potřebovali dosáhnout. Graf přitom zůstane rovinný.

- Podle prohledávání do hloubky: tentokrát budeme přidávat hrany z v_i do w_{i+1} a zařadíme je tak, aby je prohledávací algoritmus objevil vždy před hranou do v_{k+1}. Takové cesty donutí prohledávání zpracovat nejdřív B_1 až B_k a teprve pak barvit vrchol v_{k+1}. Přitom vrchol w_i má blokovánu pouze barvu $i-1$, takže ho určitě obarvíme jedničkou. Protipříklad opět zachráněn, graf opět zůstane rovinný. (Jediný problém je s hranou v_1w_2, na tu musíme přidat ještě jeden vrchol, jinak bude w_2 obarven barvou 2, což nechceme.)

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

#define MAXN 100
#define MAXM (6*MAXN)

int N;                                  /* Počet vrcholů */
int first[MAXN];                        /* První hrana z vrcholu */
int dest[MAXM], next[MAXM];             /* Pro hranu: kam vede a další hrana */
int deg[MAXN];                          /* Stupně vrcholů */
int queue[MAXN];                        /* Fronta vrcholů nízkého stupně */
int color[MAXN];                        /* Přiřazené barvy */

void melolontha (void) {                /* Schroustá vstup */
  int i, j, M=1;
  scanf ("%d", &N);
  while (scanf ("%d%d", &i, &j) == 2) {
    /* Zařadíme novou hranu */
    deg[i]++, deg[j]++;
    dest[M]=j; next[M]=first[i]; first[i]=M;
    M++;
    dest[M]=i; next[M]=first[j]; first[j]=M;
    M++;
  }
}

void scan (void) {                      /* Projde graf podle stupňů */
  int r=0, w=0;                         /* Čtecí a zápisový index fronty */
  int i, j;
  for (i=0; i<N; i++)                   /* Na počátku nízké stupně */
    if (deg[i] < 6)
      queue[w++] = i;
  while (r < w) {                       /* Procházíme frontu */
    i = queue[r++];
    for (j=first[i]; j; j=next[j])      /* ... všechny hrany z vrcholu i */
      if (--deg[dest[j]] == 5)          /* Nový vrchol nízkého stupně */
        queue[w++] = dest[j];
  }
  if (r != N) {
    puts ("Graf nebyl rovinný, to není fair!");
    exit (1);
  }
}
```

```
void paint (void) {                          /* Obarví vrcholy */
    int r, i, j;
    int avail[7];                            /* Které barvy jsou zatím volné */
    for (r=N-1; r>=0; --r) {                 /* Projdeme frontu pozpátku */
        i = queue[r];
        for (j=1; j<=6; j++)                 /* Zatím jsou všechny barvy volné */
            avail[j] = 1;
        for (j=first[i]; j; j=next[j])       /* Mimo barev použitých sousedy */
            avail[color[dest[j]]] = 0;
        j = 1;                               /* Zvolíme první nepoužitou */
        while (!avail[j])
            j++;
        color[i] = j;
    }
    for (i=0; i<N; i++)
        printf ("Ve městě %d se bude mluvit jazykem %d.\n", i, color[i]);
}
int main (void)
{
    melolontha ();
    scan ();
    paint ();
    return 0;
}
```

17-1-3 Chmatákův lup — Miroslav „miEro“ Rudišín

Při řešení tohoto příkladu se mnoho z vás inspirovalo příkladem v kuchařce. Avšak asi jenom polovina řešení byla správně. Hlavním problémem nefunkčních řešení bylo zejména nesprávné ošetření vícenásobného započtení jednoho předmětu.

Klíčové pozorování, které vede k pěknému řešení: Máme-li maximální dosažitelné ceny lupu pro všechny celočíselné kapacity batohu pomocí prvních k předmětů, snadno spočtěme maximální ceny pro prvních $k+1$ předmětů. A to tak, že zkusíme přidat $k+1$-ní předmět do batohu s k předměty a kapacitou nižší o hmotnost tohoto předmětu. Předmět nepřidáme, pokud cena takového lupu není vyšší od ceny lupu z prvních k předmětů v batohu se stejnou kapacitou. První předmět je možné vložit do batohu s kapacitou rovnou alespoň hmotnosti předmětu a cena lupu je maximální možná. Pro další předměty to plyne z indukce.

Budeme postupně zaplňovat tabulku podle popisu v pozorování. Z této tabulky snadno zrekonstruujeme vložené předměty. Stačí si uvědomit, že předmět i byl použit, pokud vylepšil cenu lupu v batohu se stejnou nosností bez tohoto předmětu.

Časová i paměťová složitost je $O(P \cdot N)$.

```
#include <stdio.h>

#define maxP 100
#define maxN 100
#define MAX (a, b)  ( ( (a)> (b)) ? (a): (b))

int main () {
    int N;                                          /* kolik zloděj unese */
    int P;                                          /* počet předmětů */
    int m[maxP];                                    /* hmotnosti předmětů */
    float c[maxP];                                  /* ceny předmětů */

    /* maximální hodnota lupu z prvních p předmětů v batohu s kapacitou n */
    float batoh[maxP][maxN];

    scanf ("%d %d",  &N,  &P);
    for  (int i=1; i<=P; i++)
       scanf ("%d %f",  &m[i],  &c[i]);

    for  (int j=0; j<=N; j++)
       batoh[0][j] = 0;

    /* postupně přidáváme předměty */
    for  (int i=1; i<=P; i++) {
       for  (int j=0; j<m[i]; j++)                  /* nízké hmotnosti jenom zkopírujeme */
          batoh[i][j] = batoh[i-1][j];
       for  (int j=m[i]; j<=N; j++)                 /* pokud lze, zlepšíme i-tým předmětem */
          batoh[i][j] = MAX (batoh[i-1][j],  batoh[i-1][j-m[i]]+c[i]);
    }

    printf ("cena: %f\n",  batoh[P][N]);
    printf ("předměty:");
    for  (int i=P,  j=N; i>0; i--)
       if  (batoh[i][j] > batoh[i-1][j]) {          /* předmět i bereme, vylepšil cenu */
          printf ("␣%d",  i);
          j -= m[i];
       }

    return 0;
}
```

17-1-4 Paloučkova Výhra — Tomáš Gavenčiak

V této úloze jde o nalezení nejkratší kružnice v ohodnoceném grafu. Celkem často se vyskytovalo jedno z popsaných řešení, ale našla se i originální řešení v $O(N^4)$, $O(N^5)$.

Jednodušší řešení je dynamické, upravím vlastně Floyd-Warshalla tak, že postupně rozšiřuji množinu S už prozkoumaných vrcholů. Na začátku do ní vložím libovolné 2 vrcholy a do matice vzdáleností D si uložím délky hran nebo příp. nekonečna (jako v kuchařce). Potom vždy vezmu vrchol k, který ještě není v S, a pro všechny jeho sousedy i,j, kteří už jsou v S (jsou-li takoví), prozkoumám cestu $i \cdots j$. Pokud ji totiž už znám, určitě nevede přes k a $k - i \cdots j - k$ je tedy kružnice. Ze všech takto postupně nacházených kružnic

si vyberu nejmenší a tu si zapíšu (pamatuji si vždy jen tu nejlepší) – to udělám tak, že si (stejně jako v kuchařce) pamatuji město s největším číslem na každé cestě $E[i,j]$, updatuji ho při změně cesty, vypisuji rekurzí. Nemůžu nejmenší kružnici nijak přeskočit, protože jakmile právě přidávám do S její poslední vrchol, potom i a j se jednou trefí do jejich hran a pokud by její součástí měla být delší cesta než mnou nalezená $i \cdots j$, nebyl by výsledek nejmenší kružnice. Už zbývá jen projet všechna $i, j \in S$ a zjistit, zda nejsou $i \cdots k \cdots j$ nebo $i \cdots j - k$ kratší než původní $i \cdots j$ nebo $i \cdots k$ a případně je vylepšit. Časová složitost jednodušší verze je $O(N^3)$, paměťová složitost $O(N^2)$.

Další možnost řešení byla vybrat postupně každý vrchol jako start a spustit z něj Dijkstrův algoritmus (kterým najdu nejkratší cesty do všech ostatních vrcholů) a potom pro každou hranu nevedoucí z/do startu prozkoumat nejkratší cesty vedoucí z jejich konců na start. Pokud ani jedna z těchto cest není podmnožinou té druhé (na to se stačí podívat na jejich první hrany, jestli některá není ta prozkoumávaná), určitě tvoří buď kružnici nebo pseudokružnici (s „ocáskem“). Tak jako tak si z těchto všech mohu zapamatovat nejkratší (pseudo)kružnici, protože platí, že stejně musím (pro nějaký jiný start) objevit i kružnici bez toho ocásku, a ta bude určitě kratší. Časová složitost je zde (podle implementace Dijkstrova algoritmu) $O(N^3)$ při nejjednodušší implementaci, $O(MN \log N)$ při použití haldy a $O(MN + N^2 \log N)$ při použití Fibonacciho haldy. Rozdíl těchto časových složitostí není sice velký (ty jsou v nejhorším vždy $O(N^3)$ až $O(N^3 \log N)$), ale některé z těchto implementací jsou mnohem lepší pro řídké grafy. Paměťová složitost je $O(N^2)$.

```
var N:integer;                                    {počet měst}
    D,E:array[1..N] of array[1..N] of integer; {D - délky silnic mezi městy,
                                                D[i][i]="nekonečno",
                                                neexistující="nekonečno",
                                                E - nastaveny na 0}
    P:array[1..N] of integer;              {křižovatky v min kružnici, končí 0}
    i,j,k,min_i,min_j,min:integer;
begin
  Nacti_a_inicializuj(E,D,N);
  min:=maxint;                                    {zatím žádná kružnice}
  for k:=3 to N do begin                          {S=(1..k-1)}
    for i:=1 to k-2 do                            {hledám min kružnice obsah. k}
      for j:=i+1 to k-1 do
        if (D[i][k]+D[k][j]+D[i][j]<min) then  {i-k a j-k musí být přímo hrany}
        begin
          min:=D[i][k]+D[k][j]+D[i][j];
          min_i:=i;
          min_j:=j;
        end;

    {do P si zapamatuj si novou min. kružnici z i do j s k}
    Pamatuj(P,E,i,j,k);
```

```
    {podle upraveného F-W zjisti nové lepší cesty}
    for i:=1 to k-1 do
      for j:=1 to k-1 do begin
        if D[i][k]+D[k][j]<D[i][j] then begin
           D[i][j]:=D[i][k]+D[k][j]; E[i][j]:=k;
        end;
        if D[i][j]+D[j][k]<D[i][k] then begin
           D[i][k]:=D[i][j]+D[j][k]; E[i][k]:=MAX(E[i][j],j);
        end;
      end;
  end;
  Vypis(P,min);              {pokud min="nekonečno", žádná kružnice neexistuje}
end.
```

17-1-5 Jazykozpytcův poklad — Tomáš Valla

Obě zadané úlohy patřily spíše k těm snazším a řešitelé, kteří úlohy odeslali, byli převážně dvou druhů. První, malá skupina těch, kteří si nejspíše pořádně nepřečetli zadání, řešila povětšinou něco úplně jiného. Skupina druhá, naštěstí podstatně větší, která se prokousala povídáním o jazycích a správně pochopila definici konečného automatu a gramatiky, po přečtení zadání bez problémů obě úlohy vyřešila.

Jak tedy na konstrukci konečného automatu, který rozpoznává binární čísla dělitelná třemi a nedělitelná dvěma? Použijeme velice jednoduchý trik. Budeme automatem postupně cifru po cifře načítat číslo a průběžně si budeme pamatovat nikoli jeho hodnotu, nýbrž pouze zbytek po dělení šesti. Zjevně nám potom budou vyhovovat pouze čísla tvaru $6k + 3$ pro nějaké k, ostatní jsou buďto dělitelná dvojkou nebo nedělitelná trojkou. Když máme načtené číslo s aktuálním zbytkem z (tedy tvaru $6k + z$), po načtení 0 dostaneme číslo $2(6k + z)$, po načtení 1 číslo $2(6k + z) + 1$. Zbývá si tedy pouze spočítat, jak se pro každé z a načtenou cifru zbytek změní.

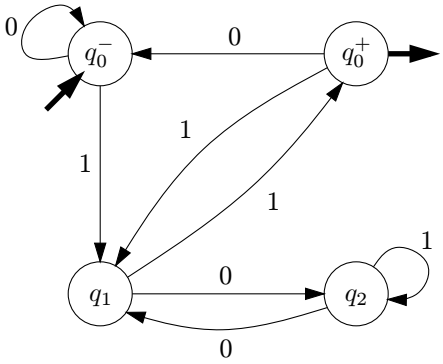
Náš stroj tedy bude používat stavy $Q = \{q_0, \ldots, q_5\}$, kde stav q_i značí, že aktuální zbytek je i. Počáteční stav bude q_0 a jediný přijímací stav bude $F = \{q_3\}$. Přechodová funkce δ bude vypadat takto:

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_3
q_2	q_4	q_5
q_3	q_0	q_1
q_4	q_2	q_3
q_5	q_4	q_5

Každý sám už si asi domyslí, že podobný postup by fungoval, pokud bychom měli zadanou libovolnou konečnou množinu M čísel, které musí, resp.

nesmí dělit vstup. Stačí vzít tolik stavů, kolik je nejmenší společný násobek čísel z M, přepočítávat při načítání zbytek a vhodně zvolit přijímací stavy.

Nicméně náš konkrétní automat lze navrhnout ještě jednodušší. Dělitelnost dvojkou je totiž ekvivalentní tomu, že poslední načtená cifra je 0. Stačí si tedy průběžně počítat zbytek při dělení třemi a to, zda poslední načtená cifra byla 0 nebo 1, nás zajímá jen v případě, že bychom se pomocí ní dostali do stavu reprezentujícího zbytek nula. To lze vyřešit tak, že tento stav rozštěpíme na dva, q_0^- a q_0^+, podle toho, zda jsme zbytku nula dosáhli načtením 0 či 1. Přijímací stav pak bude pouze q_0^+.



Když už máme hotový automat (Q, A, δ, q_0, F), je, jak si také většina řešitelů správně všimla, velmi jednoduché podle něj zkonstruovat ekvivalentní gramatiku (V_N, V_T, S, P). Pro stavy automatu $Q = \{q_0, \ldots, q_k\}$, zavedeme do gramatiky neterminální symboly $V_N = \{Q_0, \ldots, Q_k\}$, terminální symboly $V_T = A$ budou písmena abecedy, počáteční symbol $S = Q_0$ bude neterminál odpovídající počátečnímu stavu automatu.

Pro každý stav q_i a každé písmeno p se podíváme, jaký nový stav $q_j = \delta(q_i, p)$ vrací přechodová funkce. Do množiny přepisovacích pravidel P potom dáme příslušné pravidlo $Q_i \to pQ_j$. Pokud je navíc stav q_j přijímací, přidáme ještě pravidlo $Q_i \to p$. Každá expanze gramatiky zjevně následuje výpočet automatu, a tudíž generuje stejný jazyk.

Konkrétně v našem případě dostaneme gramatiku tvaru $(V_N, \{0,1\}, Q_0^-, P)$ s neterminály $V_N = \{Q_0^-, Q_0^+, Q_1, Q_2\}$ a pravidly P:

$$\begin{aligned} Q_0^- &\to 0Q_0^- \mid 1Q_1 \\ Q_0^+ &\to 0Q_0^- \mid 1Q_1 \\ Q_1 &\to 0Q_2 \mid 1Q_0^+ \mid 1 \\ Q_2 &\to 0Q_1 \mid 1Q_2 \end{aligned}$$

První dva řádky jsou víceméně stejné, lze je tedy dokonce nahradit jediným řádkem $Q_0 \to 0Q_0 \mid 1Q_1$, příslušně modifikovat pravidlo pro Q_1 a ušetřit tak jeden neterminální symbol.

V zadání jsme tvrdili, že regulárním jazykům odpovídají právě gramatiky s pravidly tvaru $N \to uM$ a $N \to u$, kde N, M jsou neterminály a u je terminál. První část tohoto tvrzení, konstrukce ekvivalentní gramatiky ze zadaného automatu, jsme právě ukázali. Zbývá popsat konstrukci ekvivalentního automatu ze zadané gramatiky výše uvedeného tvaru. Ta se provede opačným postupem než při převodu automat → gramatika a detaily si jistě rozmyslí každý sám.

17-2-1 Prasátko Květomil — Zdeněk Dvořák

Jak si mnozí řešitelé správně povšimli, tato úložka byla zaměřena převážně na procvičení hešování – to se nám bude hodit dokonce dvakrát.

Nejprve si povšimněme, že se po nás chce pouze spočítat počet výrazů v programu, jejichž hodnota je různá – výrazy se stejnou hodnotou bychom nevyhodnocovali dvakrát, ale poprvé uložili do pomocné proměnné a podruhé použili tuto uloženou hodnotu. Upřesněme si ještě, co to znamená „mít stejnou hodnotu“. Představme si, že bychom za proměnné ve výrazech postupně dosazovali jejich definice tak dlouho, dokud by alespoň jedna proměnná neměla svou počáteční hodnotu. Pak dva výrazy E_1 a E_2 jsou si rovny, pokud

- $E_1 = E_2 = v$, kde v je nějaká proměnná, nebo
- $E_1 = E_1' \; op \; E_1''$, $E_2 = E_2' \; op \; E_2''$, kde op je buď $+$ nebo $*$ a buď
  - E_1' je rovno E_2' a E_1'' je rovno E_2'', nebo
  - E_1' je rovno E_2'' a E_1'' je rovno E_2'.

Samozřejmě ověřovat rovnost přímo podle této definice je nevhodné (už proto, že takto rozexpandované výrazy mohou mít i exponenciální velikost). Místo toho každému výrazu přiřadíme číslo, které bude reprezentovat jeho hodnotu – tj. dva výrazy dostanou stejné číslo právě tehdy, pokud jsou si rovny, jinak dostanou různá čísla.

První hešovací tabulka A bude jménu proměnné přiřazovat číslo hodnoty, která je aktuálně v této proměnné uložena. Ve druhé hešovací tabulce B si pak budeme pamatovat čísla hodnot výrazů, které se v programu vyskytují – klíčem této tabulky budou trojice (operátor, číslo hodnoty levého operandu, číslo hodnoty pravého operandu), a jim bude přiřazeno číslo hodnoty tohoto výrazu. Na konci stačí vypsat počet různých čísel hodnot v tabulce B, protože to bude právě počet různých hodnot výrazů v programu.

Čísla hodnot výrazů určujeme takto:

- Když zpracováváme nějakou proměnnou poprvé, přiřadíme jí nové číslo hodnoty.

- Když zpracováváme přiřazení $var_1 = var_2$, pak proměnné var_1 přiřadíme stejné číslo hodnoty, jaké má proměnná var_2.
- Když zpracováváme přiřazení $var_1 = var_2\ op\ var_3$, pak si nejprve zjistíme čísla hodnot v proměnných var_2 a var_3 – nechť to jsou n_2 a n_3. Pak se podíváme do hešovací tabulky B, zda v ní je uložen výraz (op, n_2, n_3). Je-li tomu tak, pak jeho číslo hodnoty přiřadíme proměnné var_1. Jinak tento výraz přidáme do tabulky B s novým číslem hodnoty a toto číslo přiřadíme proměnné var_1.

Zbývá si rozmyslet, jak ošetřit komutativitu operací. To je ale snadné – před prací s tabulkou B stačí čísla hodnot v trojici seřadit tak, aby druhé z nich bylo menší nebo rovno třetímu.

Časová složitost na operaci s tabulkou A je v průměrném případě $O(k)$, kde k je délka názvu proměnné. Protože pro každý výskyt proměnné v programu provedeme právě jednu operaci s touto tabulkou, dohromady bude časová složitost pro práci s ní $O(n)$, kde n je délka vstupu. Časová složitost pro práci s tabulkou B je $O(1)$ na operaci, a počet operací s ní je roven počtu přiřazení ve vstupu, tj. celková časová složitost je $O(n)$ – toto je složitost v průměrném případě, v nejhorším případě, kdy by docházelo ke všem možným kolizím, by časová složitost byla $O(n^2)$. Paměťová složitost je zřejmě $O(n)$.

Poznámka na závěr – zde popsaná metoda identifikace redundantních výpočtů se s mírnými vylepšeními skutečně používá v kompilátorech. Anglický název je Value Numbering.

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

#define MAX_VARS 107
#define MAX_EQS 107
#define MAX_LINE_LENGTH 100

struct var_hash_elt {
    char *var_name;
    unsigned var_value;
};

struct expr_hash_elt {
    char operator;
    unsigned left_val, right_val, expr_value;
};

struct var_hash_elt var_hash[MAX_VARS];
struct expr_hash_elt expr_hash[MAX_EQS];

unsigned n_vars, n_values;

static struct var_hash_elt* get_var (char *var_name) {
    unsigned hash = 0;
    char *t;

    for (t = var_name; *t; t++) hash = (76 * hash + *t) % MAX_VARS;
```

```
    while (var_hash[hash].var_name && strcmp (var_hash[hash].var_name, var_name)) {
        hash++;
        if (hash == MAX_VARS) hash = 0;
    }
    if (!var_hash[hash].var_name) {
        var_hash[hash].var_name = strdup (var_name);
        var_hash[hash].var_value = n_values++;
        n_vars++;
    }
    return var_hash + hash;
}
static struct expr_hash_elt* get_expr (unsigned left_val, char op, unsigned right_val) {
    unsigned hash;
    if (left_val > right_val) left_val^=right_val^=left_val^=right_val;
    hash = (76 * left_val + 777 * op + right_val) % MAX_EQS;
    while (expr_hash[hash].operator && (expr_hash[hash].operator != op
                                        || expr_hash[hash].left_val != left_val
                                        || expr_hash[hash].right_val != right_val)) {
        hash++;
        if (hash == MAX_EQS) hash = 0;
    }
    if (!expr_hash[hash].operator) {
        expr_hash[hash].operator = op;
        expr_hash[hash].left_val = left_val;
        expr_hash[hash].right_val = right_val;
        expr_hash[hash].expr_value = n_values++;
    }
    return expr_hash + hash;
}
static int id_char (char ch) {
    return ( ('a'<= ch && ch <= 'z') || ('A'<= ch && ch <= 'Z') || ch == '_');
}
static void skip_blanks (char **buffer) {
    while (**buffer == '␣') (*buffer)++;
}
static struct var_hash_elt* parse_var (char **buffer) {
    char *var_end, ech;
    struct var_hash_elt *ret;
    skip_blanks (buffer);
    for (var_end = *buffer; id_char (*var_end); var_end++) continue;
    ech = *var_end; *var_end = 0;
    ret = get_var (*buffer);
    *var_end = ech; *buffer = var_end;
    skip_blanks (buffer);
    return ret;
}
```

```
static char parse_eq (char *buffer, struct var_hash_elt **tgt, unsigned *lv, unsigned *rv) {
  char ret;

  skip_blanks (&buffer);
  if (!*buffer) return '␣';

  *tgt = parse_var (&buffer);
  if (*buffer++ != '=') abort ();

  *lv = parse_var (&buffer)->var_value;
  if (*buffer == ';') ret = '=';
  else {
    ret = *buffer++;
    if (ret != '+' && ret != '*') abort ();
    *rv = parse_var (&buffer)->var_value;
  }

  if (*buffer++ != ';') abort ();
  skip_blanks (&buffer);
  if (*buffer) abort ();

  return ret;
}

int main (void) {
  char buffer[MAX_LINE_LENGTH];

  while (gets (buffer)) {
    unsigned left_val, right_val;
    struct var_hash_elt *tgt;
    char eq_type = parse_eq (buffer, &tgt, &left_val, &right_val);

    switch (eq_type) {
      case '*':
      case '+':
        tgt->var_value = get_expr (left_val, eq_type, right_val)->expr_value;
        break;
      case '=':
        tgt->var_value = left_val;
        break;
    }
  }
  printf ("%d\n", n_values - n_vars);
  return 0;
}
```

17-2-2 Bobr Béďa — Tomáš Gavenčiak

Všichni správně uhodli, že jde o hledaní minimální kostry grafu a že předem dané hrany tvořící cykly nijak nevadí. Mnoho z vás ale pak zvolilo buď zbytečně pomalý algoritmus (no comment :–)) nebo použili více či méně rychlou implementaci DFU. Ta sice běží v čase $O(N \cdot \alpha(N))$, ale pro tuto úlohu je zbytečně složitá.

Naše řešení bude založeno na Jarníkově algoritmu pro hledání kostry. Budeme postupně budovat kostru tak, že začneme s jedním libovolným vrcholem a vybereme si jeho souseda takového, že ještě v kostře není, a přitom je k současné kostře nejblíže. Takto pokračujeme, dokud nepřidáme vrcholy všechny.

Popsaný algoritmus naimplementujeme pomocí haldy tak, že na začátku začneme s libovolným vrcholem a do haldy přidáme všechny hrany, které z tohoto vrcholu vedou. V každém kroku pak z haldy vezmeme hranu s nejmenším ohodnocením a podíváme se, jestli nějaký její konec ještě není v kostře. Pokud ne, přidáme ho do ní (je momentálně nejblíž vytvářené kostře) a do haldy vložíme všechny hrany z tohoto vrcholu vedoucí. Pokud už oba konce hrany v kostře jsou, neděláme nic. Tento celý postup se opakuje, dokud je v haldě nějaká hrana.

Jaká bude časová složitost? Každou hranu přidáme do haldy maximálně dvakrát a na každý vrchol sáhneme nanejvýš čtyřikrát (tolik z něj vede hran). Pokud by naše operace s haldou byly v konstantním čase, bude celková časová složitost $O(N \cdot M)$.

V haldě budeme mít naštěstí jenom hrany s ohodnocením 0 (dopředu vyryté kanálky), 1 (svislé kanálky) a 2 (vodorovné kanálky). Můžeme si tedy pamatovat hrany ve třech polích podle jejich ohodnocení. Pokud hledáme hranu s nejmenším ohodnocením, zkusíme pole s ohodnocením 0, a pokud je prázdné, tak 1 nebo 2. Každopádně všechny tyto operace (přidat hranu a odebrat hranu s nejmenším ohodnocením) zvládneme v konstantním čase.

Můj program bude mít v poli $h[i]$ vrcholy, ke kterým vede z už propojené části hrana s ohodnocením i. Vrchol v nich může tak být až 4× (přidán z různých stran), ale to mi nijak nevadí. Do polí fv a fs si na začátku načtu již hotové hrany.

```
var hp:array[0..2] of integer;              {počty hran s ohodnocením 0..2}
    hx:array[0..2,1..2*M*N] of integer;     {počáteční souřadnice}
    hy:array[0..2,1..2*M*N] of integer;
   hxo:array[0..2,1..2*M*N] of integer;     {cílové souřadnice}
   hyo:array[0..2,1..2*M*N] of integer;
     f:array[1..M,1..N] of boolean;         {značky navštívení}
    fv:array[1..M,1..N] of boolean;         {je kanálek z [i,j] doprava už hotov}
    fs:array[1..M,1..N] of boolean;         {je kanálek z [i,j] dolů už hotov}

procedure pridej(xo,yo,x,y:integer);        {přidá do haldy hranu odkud-kam}
var v:integer;
begin
  if xo>x and fv[x,y] then v:=0             {zjistím cenu}
    else if xo<x and fv[xo,y] then v:=0
      else if yo>y and fs[x,y] then v:=0
        else if yo<y and fs[x,yo] then v:=0
          else if yo!=y then v:=1
            else v:=2;
```

```
  inc(hp[v]);
  hx[v,hp[v]]:=x;
  hy[v,hp[v]]:=y;
  hxo[v,hp[v]]:=xo;
  hyo[v,hp[v]]:=yo;
end;

begin
  {nactu uz hotové do fv,fs}

  for i:=0 to N do
     for j:=1 to M do
      f[i,j]:=false;
  hp[0]:=1;
  hp[1]:=0;   hp[2]:=0;
  hx[0,1]=1; hy[0,1]:=1;
  while (hp[0]>0) or (hp[1]>0) or (hp[2]>0) do begin
    if hp[0]>0 then begin                          { vyberu hranu z haldy }
      x:=hx[0,hp[0]]; y:=hy[0,hp[0]];
      xo:=hx[0,hp[0]]; yo:=hy[0,hp[0]]; dec(hp[0]);
    end else if hp[1]>0 then begin
      x:=hx[1,hp[1]]; y:=hy[1,hp[1]];
      xo:=hx[1,hp[1]]; yo:=hy[1,hp[1]]; dec(hp[1]);
    end else begin
      x:=hx[2,hp[2]]; y:=hy[2,hp[2]];
      xo:=hx[2,hp[2]]; yo:=hy[2,hp[2]]; dec(hp[2]);
    end;
    if not f[x,y] then begin                      {přidám [x,y] do kostry?}
      f[x,y]:=true;
      writeln("(",x,",",y,")-(",xo,",",yo,")");
      if y>1 then pridej(x,y,x,y-1);
      if x>1 then pridej(x,y,x-1,y);
      if y<N then pridej(x,y,x,y+1);
      if x<M then pridej(x,y,x+1,y);
    end;
  end;
end.
```

17-2-3 Krkavec Kryšpín — Pavel Machek

Těžiště trojúhelníku je zároveň středem kružnice opsané, je to bod, který je od všech vrcholů stejně vzdálen. Souřadnice těžiště trojúhelníku lze tedy můžeme spočítat podle vzorce $x = (x_1 + x_2 + x_3)/3$ a $y = (y_1 + y_2 + y_3)/3$.

Další vzorec, který budeme potřebovat, je z fyziky: těžiště soustavy hmotných bodů lze spočítat jako vážený průměr souřadnic těch bodů. Tedy

$$x = (x_1 m_1 + x_2 m_2 + \ldots + x_n m_n)/(m_1 + m_2 + \ldots + m_n)$$
$$y = (y_1 m_1 + y_2 m_2 + \ldots + y_n m_n)/(m_1 + m_2 + \ldots + m_n)$$

kde x_i, y_i jsou souřadnice bodů a m_i je jeho hmotnost. Pro naše účely na jednotce hmotnosti nezáleží. Vzorec bude fungovat i pro záporné „hmotnosti“, což se nám bude hodit pro nekonvexní útvary.

Ještě potřebujeme znát plochu jednoho trojúhelníku, tu lze získat jednoduše jako $S = 1/2 \cdot |AB \times AC|$ neboli $S = 1/2 \cdot |(B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y)|$.

Pro konvexní útvary by stačilo rozdělit mnohoúhelník na trojúhelníky A_1, A_i, $A_i + 1$, a těžiště získat pomocí tří vzorců nahoře.

Pokud ale při výpočtu plochy použijeme výše uvedený vzorec bez absolutní hodnoty, tedy $S = 1/2 \cdot ((B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y))$, budeme dostávat kladné a záporné výsledky podle toho, jestli trojúhelník ABC je orientován po nebo proti směru hodinových ručiček. Toho lze obratně využít: vybereme libovolný bod O (třeba počátek systému souřadnic) a použijeme vzorce nahoře pro trojúhelníky O, A_i, A_i+1. Každý bod, který je uvnitř mnohoúhelníku, bude součástí lichého počtu dílčích trojúhelníků a díky tomu se bude počítat do celkového výsledku. Body, které jsou mimo mnohoúhelník, budou součástí sudého počtu trojúhelníků a díky opačným orientacím se navzájem odečtou a celkový výsledek neovlivní.

Časová složitost je lineární vzhledem k počtu vrcholů a lépe to nejde, protože výsledek záleží na všech bodech. Paměťová složitost je konstantní, body jsou zpracovávány hned, jak přichází ze vstupu.

S díky Janu Pelcovi.

```
#include <stdio.h>

int main (void) {
    int n, i;
    double x, y;                          /* Aktuální bod */
    double px, py;                        /* Předchozí bod */
    double lx, ly;                        /* Poslední bod */
    double tx = 0, ty = 0;                /* Sumy čitatelů */
    double tm = 0;                        /* Suma jmenovatele */
    double m;                             /* Váha aktualního trojúhelníku */

    printf ("Počet bodů mnohoúhelniku:");
    scanf ("%d", &n);

    printf ("Bod 1:\n");
    scanf ("%f %f", &lx, &ly);
    px = lx; py = ly;

    for (i=1; i<=n; i++) {
      if (i != n) {
        printf ("Bod %d:\n", i+1);
        scanf ("%f %f", &x, &y);
      } else {
        x = lx; y = ly;
      }
```

```
        m = x * py - px * y;                    /* Váha trojúhelníku */
        tx += (x + px) * m;
        ty += (y + py) * m;
        tm += m;

        px = x; py = y;
    }
    tx /= tm * 3.0;
    ty /= tm * 3.0;
    printf ("\nTežiště má souřadnice [%f, %f].\n", tx, ty);
    return 0;
}
```

17-2-4 Mravenec Ferda — Petr Škoda

Ferda se dlouho probíral vašimi programy, ale naštěstí pro něj a naneštěstí pro Ptáčka Sáčka nalezl mezi řešeními kýžený algoritmus.

Hlavní myšlenkou algoritmu je zpracovávat úlohu postupně. Bude nás zajímat součet horních čísel na kostkách. Pokud otočíme kostku, změní se horní číslo na kostce, a tedy i součet horních čísel. Naším cílem je najít takové otočení kostek, aby součet horních čísel byl co nejbližší součtu dolních čísel. Víme, že součet čísel je menší než $K \cdot N$. Zajímáme se tedy, pro které součty čísel s existuje otočení kostek, jehož součet horních čísel na kostkách je právě s. Dalším parametrem součtu čísel s je minimální počet otočených kostek $O[s]$, který je potřeba, abychom dosáhli součtu s. Pole O budeme budovat postupně. Označme O_i pole O vytvořené z i prvních kostek. Zřejmě $O_0[0] = 0$ a pro ostatní součty s má hodnotu *Null*, která znamená, že tohoto součtu nelze dosáhnout. Jakmile máme vytvořené pole O_i, pak pole O_{i+1} naplníme následovně: Označme čísla na $i+1$-ní kostce h a d. Projdeme všechny možné součty s od 0 do $K \cdot N$. Pokud $O_i[s]$ není *Null*, zkusíme k součtu přidat kostku. Pokud je $O_i[s] < O_{i+1}[s+h]$, nastavíme $O_{i+1}[s+h] := O_i[s]$ a podobně pokud $O_i[s]+1 < O_{i+1}[s+d]$, nastavíme $O_{i+1}[s+d] := O_i[s]+1$. Samozřejmě v případě hodnoty *Null* uložíme novou hodnotu. Tímto způsobem sice ztratíme některé možnosti otočení kostek, ale určitě si uchováme ty součty, které potřebují nejmenší otočení kostek. Jakmile naplníme pole O_n, jsme hotovi. Stačí už jen vybrat nejbližší součet a najít kostky, které musíme otočit. Otočení jednotlivých kostek si uložíme už při vytváření pole O – do pole $T_i[s]$ si poznamenáme, zda jsme při vytváření součtu s z prvních i kostek otočili i-tou kostku. Pak už stačí jen vystopovat všechny otočené kostky z finálního součtu prostým odečítáním horních či dolních čísel kostek.

Algoritmus používá k uložení součtů N polí o velikosti $K \cdot N$, takže jeho paměťová složitost je $O(N^2 \cdot K)$. Časově nejnáročnější operací je právě naplnění těchto polí, tedy časová složitost je také $O(N^2 \cdot K)$. V algoritmu jsme použili myšlenku spočítat si řešení pro část vstupu, uložit si ho a pak ho znovu použít pro další výpočet. Tomuto způsobu řešení se říká dynamické programování.

```
program Ferda;
const
  MaxN = 1000;
  MaxK = 10;
  Null = -1;
var
  O: array[0..MaxN, 0..MaxN * MaxK] of Integer;
  T: array[0..MaxN, 0..MaxN * MaxK] of Boolean;
  H, D: array[0..MaxN] of Integer;
  K, N, R: Integer;
  I, S: Integer;
begin
  Readln(N, K);
  for I:= 1 to N do Readln(H[I], D[I]);

  for I:= 0 to N do
    for S:= 0 to N * K do
      O[I, S]:= Null;

  O[0, 0]:= 0;
  for I:= 0 to N - 1 do
    for S:= 0 to K * N do
      if O[I, S] <> Null then begin
        if (O[I + 1, S + H[I + 1]] = Null) or
           (O[I, S] < O[I + 1, S + H[I + 1]]) then begin
           O[I + 1, S + H[I + 1]]:= O[I, S];
           T[I + 1, S + H[I + 1]]:= False;
        end;
        if (O[I + 1, S + D[I + 1]] = Null) or
           (O[I, S] + 1 < O[I + 1, S + D[I + 1]]) then begin
          O[I + 1, S + D[I + 1]]:= O[I, S] + 1;
          T[I + 1, S + D[I + 1]]:= True;
        end;
      end;

  R:= 0;
  for I:= 1 to N do R:= R + H[I] + D[I];

  for I:= R div 2 downto 0 do
    if (O[N, I] <> Null) or (O[N, R - I] <> Null) then
      if (O[N, R - I] = Null) or (O[N, I] <= O[N, R - I]) then
        R:= I else R:= R - I;

  for I:= N downto 1 do
    if T[N, R] then  begin
      Writeln(I);
      R:= R - D[I];
    end else
      R:= R - H[I];
end;
```

17-2-5 Jazykozpytcova pomsta — Tomáš Valla

Správných řešení první úlohy, ke kterým jsem neměl žádnou připomínku, tentokrát došlo poskrovnu. Nejběžnější chyba byla následující: V podstatě všichni řešitelé přišli na správnou myšlenku, že automat nutně nějak musí umět rozlišovat hloubku vnoření závorek. Bohužel už málokdo to uměl i správně dokázat. To přeci vůbec není na první pohled zřejmé! Stejně tak bychom mohli tvrdit, že když rozpoznáváme jazyk $\{a^i;\ i \in N\}$, tak je nutné si počítat ono i, ve skutečnosti to samozřejmě potřeba není. Proč by třebas nemohl existovat automat, který používá nějakou úplně jinou metodu? Řešitelům jsem uděloval body podle toho, nakolik myšlenku důkazu dotáhli do konce.

Ale teď už si předveďme jeden z možných správných důkazů. Jeho myšlenka je jednoduchá: automat se nemůže obejít bez rozlišování úrovně vnoření závorek. Jak to ovšem ukázat formálně?

Budeme postupovat sporem. Nechť tedy existuje konečný automat $M = (Q, A, q_0, \delta, F)$, který má k stavů a rozpoznává jazyk U správně uzávorkovaných výrazů. Vezmeme vhodné správně uzávorkované slovo a ukážeme, že z něj lze vynechat úsek tak, že slovo přestane být dobře uzávorkované, ale automat to vůbec nepozná a prohlásí ho za správné. Tím ukážeme, že žádný konečný automat M, který by měl umět rozpoznávat U, nemůže nikdy dobře pracovat.

Když má tedy automat k stavů, uvažme slovo $(^{k+1})^{k+1}$. Při načítání levých závorek automat nějak mění stavy, ale protože levých závorek je $k+1$, alespoň jedním stavem se musí projít dvakrát. Existují tedy dva indexy i a j, $i < j$, že po přečtení j-té levé závorky se automat ocitl ve stejném stavu q jako po přečtení i-té levé závorky. Jinými slovy, automat ve své „paměti“ považuje pozice i a j za nerozlišitelné. V obou případech se totiž stroj nachází ve stavu q, a následný výpočet tudíž musí mít úplně stejný průběh. A co se tedy stane, když automatu podstrčíme slovo, kde vynecháme levé závorky na pozicích $i+1$ až j? Automat to vůbec nepozná a prohlásí, že slovo je správné!

Dokonce bychom mohli tento úsek ne vynechat, nýbrž zdvojit, ztrojit, zkrátka libovolně mnohokrát znásobit. Když se nad tím zamyslíme hlouběji, podobnou vlastnost musí mít všechny nekonečné regulární jazyky. Stačí vzít dostatečné dlouhé slovo, a pak už se v něm nutně musí vyskytovat úsek, který lze beztrestně odmazat či libovolněkrát „nafouknout“. Tato skutečnost se dá v literatuře najít pod názvem *Pumping lemma.*

$\rightarrow * \leftarrow$

Druhá úloha byla myšlenkově jednodušší, zato bylo potřeba být pečlivější a dát si pozor na některé zrady, které mohly nastat. V podstatě všichni, kdo úlohu odeslali, správně přišli na to, že stačí vzít automaty $M_1 = (Q_1, A_1, q_1^0, \delta_1, F_1)$ a $M_2 = (Q_2, A_2, q_2^0, \delta_2, F_2)$, které jsou dle definice regulárního jazyka schopné rozpoznávat jazyky L_1 a L_2, a vhodně je sériově zapojit do nového stroje M, který bude rozpoznávat jazyk $L_1.L_2$. Například můžeme na každý přijímací

stav f_i stroje M_1 „přivěsit“ kopii stroje M_2 tak, že ztotožníme stav f_i s počátečním stavem stroje M_2. Počátečním stavem zvolíme počáteční stav q_1^0 stroje M_1 a jako koncové stavy zvolíme koncové stavy F_2 strojů M_2.

Z přijímacích stavů M_1 se však ještě výpočet může vrátit zpět do vnitřních stavů M_1 a tyto zpětné šipky nemůžeme vynechat. Po napojení stroje M_2 tudíž v propojovacích stavech vznikne více šipek pro jediné písmenko. To je ale přesně NKA. Jenže definice regulárního jazyka je taková, že pro něj musí existovat *deterministický* konečný automat. Tehdy však stačí použít větu dokázanou v zadání, podle které umíme k NKA M sestrojit ekvivalentní DKA.

Ještě je potřeba vyřešit pár důležitých technických detailů. Pokud bychom spojení obou automatů realizovali ztotožněním přijímacího a počátečního stavu, mohlo by se ještě stát, že se výpočet, který již přešel do M_2 přes propojovací stav, vrátí zpět do stroje M_1, což my určitě nechceme. Napojení se tudíž musí řešit rafinovaněji: Vezmeme stroj M_1 a pouze jednu kopii stroje M_2. Z každého stavu stroje M_1, ze kterého vede šipka pro písmeno a do některého přijímacího stavu z F_1, natáhneme ještě jednu šipku pro písmeno a navíc do počátečního stavu q_2^0 stroje M_2. Také je třeba ošetřit, když při práci stroje M_1 přijde znak z abecedy stroje M_2 a naopak. Zavedeme proto „odpadní“ stav q_{err}, ze kterého už nebude úniku a směřovat do něj budou šipky pro všechny špatné znaky.

Následujícím poněkud odpudivým formálním zápisem ještě výsledný nedeterministický stroj $M = (Q_1 \cup Q_2 \cup \{q_{err}\}, A_1 \cup A_2, P, \delta, F_2)$ přesně definujeme. Pokud je stav $q_1^0 \in F_1$, bude množina počátečních stavů $P = \{q_1^0, q_2^0\}$, v opačném případě $P = \{q_1^0\}$. Přechodová funkce δ bude

$$\delta(q,a) = \begin{cases} \{\delta_1(q,a)\} & q \in Q_1,\ a \in A_1,\ \delta_1(q,a) \notin F_1 \\ \{\delta_1(q,a), q_2^0\} & q \in Q_1,\ a \in A_1,\ \delta_1(q,a) \in F_1 \\ \{\delta_2(q,a)\} & q \in Q_2,\ a \in A_2. \\ \{q_{err}\} & q = q_{err},\ a \in A_1 \cup A_2 \\ \{q_{err}\} & q \in Q_1,\ a \in A_2 \setminus A_1 \text{ nebo} \\ & q \in Q_2,\ a \in A_1 \setminus A_2 \end{cases}$$

Na tento NKA M nyní aplikujeme větu o převodu na DKA a důkaz je hotov.

Ještě bychom měli věnovat pár slov několika málo řešitelům, který svůj důkaz založili na zřetězení gramatik speciálního tvaru $X \to aY$, $X \to a$ pro a terminální a X, Y neterminální, jež ke každému regulárnímu jazyku existují. Myšlenka je to samozřejmě dobrá, ale trpí stejnými neduhy při spojování jako automaty. Je opět třeba zajistit, aby se expanze gramatiky z pravidel pro L_2 nevrátila zpět do pravidel pro L_1. Navíc my jsme si ekvivalenci automatu a gramatik výše uvedeného tvaru celou nedokazovali. Druhý směr, tedy konstrukci deterministického automatu z gramatiky, jsme schválně ve vzorovém řešení první série odbyli, neboť ony „detaily, které si každý rozmyslí sám“ znamenají právě konstrukci nedeterministického konečného automatu a jeho následný převod na deterministický například pomocí věty o ekvivalenci DKA a NKA.

17-3-1 Spisovatel Vilík — Zdeněk Dvořák

Nejprve si uvědomíme, že dvě slova (úseky textu) jsou shodná, pokud obsahují stejné počty jednotlivých písmen. Tj. například „ABCAB“ a „AABBC“ jsou stejná, protože obsahují dvakrát A, dvakrát B a jednou C. Nejdříve si tedy pro každé slovo délky k v textu spočítáme, kolik kterých písmen se v něm vyskytuje – tj. každé pozici p v textu přiřadíme 30-tici čísel $T(p)$, udávající počet příslušných písmen v následujících k znacích textu. Přímočaré řešení by všechna $T(p)$ určilo v čase $O(kN)$, kde N je délka textu. Tuto složitost však můžeme snadno zlepšit na $O(N)$, pokud si povšimneme, že $T(p+1)$ se od $T(p)$ liší pouze ve dvou číslech – přibude jeden výskyt písmena na pozici $p+k$ a ubude výskyt písmena na pozici p. Čili můžeme $T(p)$ spočítat postupně od začátku do konce a na vytvoření každé 30-tice spotřebujeme pouze konstantní množství času.

Nyní zbývá pouze najít první opakování nějaké 30-tice. Jednou z možností je použít hešování (viz kuchařka druhé série). Nevýhodou je to, že lineární časovou složitost nemáme zaručenu, ale dosáhneme jí pouze v průměrném případě, nebo pokud jsme mocní mágové (tj. umíme zvolit správnou hešovací funkci), randomizovaně.

Řešení, které tuto nevýhodu nemá, je lexikograficky si 30-tice setřídit. Pak jsme schopni jedním průchodem najít opakující se 30-tice (v setříděné posloupnosti budou následovat za sebou), a pokud si navíc pamatujeme, kde se v zadaném textu vyskytovaly, je snadné určit první z nich.

K třídění použijeme RadixSort. Podrobně je popsán v kuchařce druhé série minulého ročníku, zde jen zopakujeme základní myšlenku. RadixSort funguje tak, že nejprve setřídíme posloupnost podle poslední složky 30-tice, pak podle předposlední, ..., a nakonec podle první, přičemž si dáváme pozor, abychom nezměnili pořadí prvků, které se v dané složce shodují. Není těžké si rozmyslet, že výsledná posloupnost pak bude opravdu setříděná – protože poslední třídění proběhlo podle nejdůležitější složky, a mezi slovy, která se v ní shodují, pak rozhoduje pořadí podle druhé nejdůležitější, atd. Třídění podle i-té složky v lineárním čase zvládneme snadno – prvky rozložíme do $k+1$ přihrádek podle hodnoty i-té složky a pak je vybereme od nejmenší k největší. Budeme tedy vybírat k přihrádek a samotné kopírování hodnot nám zabere čas N, dohromady bude časová složitost na jeden průchod $O(N+k) = O(N)$ a průchodů je konstantně mnoho – 30.

Takto dosáhneme časové složitosti $O(N)$ i v nejhorším případě. Paměťová složitost bude také $O(N)$.

```
#include <stdio.h>

#define MAXN 1000
#define MAXK 1000
```

```
typedef struct {
    int pocty[30];
} tice;

int stejne_pocty (tice *a, tice *b) {          /* Vrátí 1 pokud a a b jsou stejné. */
    unsigned i;
    for (i = 0; i < 30; i++) if (a->pocty[i] != b->pocty[i]) return 0;
    return 1;
}

void casesort (tice *to_sort[], unsigned l, unsigned k, unsigned slozka)
{                                              /* Setřídí L 30-tic v to_sort dle složky. */
    tice *tmp[MAXN];
    unsigned case_size[MAXK + 1], case_begin[MAXK + 1], i;

    for (i = 0; i <= k; i++) case_size[i] = 0;
    for (i = 0; i < l; i++) case_size[to_sort[i]->pocty[slozka]]++;
    case_begin[0] = 0;
    for (i = 1; i <= k; i++) case_begin[i] = case_begin[i - 1] + case_size[i - 1];
    for (i = 0; i < l; i++) tmp[case_begin[to_sort[i]->pocty[slozka]]++] = to_sort[i];
    for (i = 0; i < l; i++) to_sort[i] = tmp[i];
}

void radixsort (tice *to_sort[], unsigned l, unsigned k)
{                                              /* Lexgraf. setřídí l 30-tic v to_sort. */
    int i;
    for (i = 29; i >= 0; i--) casesort (to_sort, l, k, i);
}

unsigned kod (char ch) {                       /* Vrátí kód znaku ch. */
    if ('a'<= ch && ch <= 'z') return ch - 'a';
    if ('A'<= ch && ch <= 'Z') return ch - 'A';
    if (ch == '_') return 'z'- 'a'+ 1;
    if (ch == '.') return 'z'- 'a'+ 2;
    if (ch == '?') return 'z'- 'a'+ 3;
    if (ch == '!') return 'z'- 'a'+ 4;
    abort ();
}

int main (void) {
    char vstup[MAXN + 1];
    tice T[MAXN], *T_sorted[MAXN];
    unsigned n, k, i, p, min_opak;

    scanf ("%d%d%s", &n, &k, vstup);
    memset (&T[0], 0, sizeof (tice));

    for (i = 0; i < k; i++)                    /* Spočítáme četnosti písmen v k-ticích. */
      T[0].pocty[kod (vstup[i])]++;

    for (; i < n; i++) {
      p = i - k + 1;
      T[p] = T[p-1];
      T[p].pocty[kod (vstup[i])]++;
      T[p].pocty[kod (vstup[p - 1])]--;
    }
```

```
    for (i = 0; i < n − k + 1; i++)          /* Setřídíme 30-tice. */
      T_sorted[i] = &T[i];
    radixsort (T_sorted, n − k + 1, k);

    min_opak = n;
    for (i = 1; i < n − k + 1; i++)          /* A najdeme první opakující se. */
      if (stejne_pocty (T_sorted[i], T_sorted[i − 1])) {
        p = T_sorted[i] − T;
        if (p < min_opak) min_opak = p;
      }
    if (min_opak == n) printf ("Žádné opakování.\n");
    else printf ("Žádné opakování do pozice %d.\n", min_opak + k − 1);

    return 0;
}
```

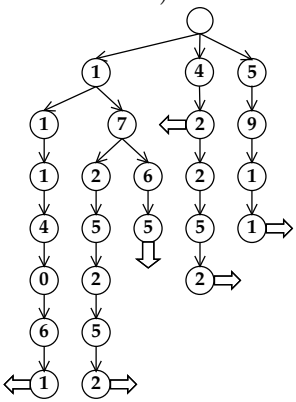
17-3-2 Popleta Truhlík — David Matoušek

Problém pana Truhlíka popletl i řadu zkušených řešitelů. Odevzdaná řešení tak byla plná roztodivného zvířectva. Za zmínku určitě stojí housenky složené z příkazů **if** (**then**) **else**, které dosahovaly délky až šestadvaceti řádků. Rovněž šestadvacetihlavá saň **switch/case** se často snažila řešit problém převodu vstupního slovníku na slovník číselný. Řešení obsahující tyto implementační neduhy však (kromě upozornění v podobě velkého FUJ) nebyla nikterak potrestána, přesto bych takovýmto řešitelům doporučil zhlédnout kód vzorového řešení. Co se již však neobešlo bez bodových ztrát, byla řešení s exponenciální složitostí, která po nalezení všech možných vět hledala větu nejkratší. Přitom leckde chybělo málo k tomu, aby časová složitost byla optimální!

Nutno podotknouti, že cest ke zdárnému vyřešení úlohy bylo poměrně mnoho. Stalo se tak hlavně proto, že úlohu bylo možné chápat z několika různých pohledů, z nichž ani jeden se po zralé úvaze nedá označit jako nesmyslný či špatný. Někteří řešitelé se tak zaměřili na provokativně zvolené N (počet telefonních čísel, ke kterým si chce Truhlík zapamatovat větu), jiní se snažili rychlost algoritmu poměřovat především s velikostí vstupního slovníku P.

Ukážeme si řešení, které pracuje v čase $O(P + \sum_{i=1}^{n} C_i^2)$. Nejprve si převedeme slova vstupního slovníku do číselné podoby. Takový převod je zřejmě jednoznačný, avšak opačný převod již ne. Číselné řetězce si uložíme do struktury zvané *trie*. Naše trie je strom, ve kterém každý vrchol představuje jednu konkrétní cifru a každý vrchol má právě deset ukazatelů na své potomky. Pokud v trii půjdeme od kořene k listům, pak vrcholy na této cestě tvoří číselný řetězec, který jsme v trii uchovali. Některé vrcholy a všechny listy jsou označené tak, že v nich číselný řetězec končí, u těchto si pamatujeme i ukazatel na slovo do původního slovníku. Příklad takové trie pro vstupní slovník „brok, kuba, je, brekeke, babizna, jedle“, kterému odpovídá slovník číselných řetězců „1765, 5911, 42, 1725252, 1114061, 42252“ je na obrázku (pro přehlednost zobrazuje-

me jen ty vrcholy, u nichž existuje nějaký ukazatel do slovníku, čili ty, které tvoří začátek nějakého slova ve slovníku):



Když už máme takovou trii postavenou, můžeme se zabývat skládáním věty pro telefonní číslo. K tomu budeme ještě potřebovat dvě pole wc a wi, obě délky telefonního čísla. V průběhu algoritmu bude prvek $wc[k]$, pro nějaké $k \in \{1, \ldots, C_i\}$, označovat minimální počet slov, které jsme doposud potřebovali ke složení cifer $i_1, \ldots, i_k$ telefonního čísla i. Prvek $wi[k]$ pak bude ukazatel na slovo v původním slovníku délky l takové, že číselný řetězec $i_{k-l}, i_{k-l+1}, \ldots, i_k$ odpovídá tomuto slovu. Na počátku inicializujeme obě pole nulami, postavíme si před sebe telefonní číslo i a začneme v kořeni trie. Načteme první cifru telefonního čísla. Pokud má kořen trie potomka, který odpovídá cifře i_1, přejdeme v trii do tohoto potomka. V případě, že u tohoto potomka je nastaven příznak konce slova, nastavíme $wc[1] = 1$ a $wi[1]$ položíme rovno ukazateli na slovo do původního slovníku, který v tomto vrcholu máme uložen. Poté načítáme další cifry a procházíme trii tak dlouho, dokud to jde, nebo až do chvíle, kdy vyčerpáme celé telefonní číslo. Zároveň pro každý navštívený vrchol trie, který označuje konec slova, nastavujeme hodnoty polí wc a wi. Udělali jsme tedy jeden průchod a všimněme si, že pole wc nyní obsahuje jedničky na těch místech, které odpovídají délkám slov původního slovníku takovým, že začátek telefonního čísla se shoduje s jejich číselnou reprezentací. Nyní přejděme k první nenulové hodnotě $wc[j]$, postavme se opět do vrcholu trie a spusťme celý průchod znovu s tím, že již pracujeme pouze s ciframi $j, \ldots, C_i$ telefonního čísla. V tomto a dalších průchodech již měníme pole wc a wi pouze tehdy, nebylo-li dané pozice v telefonním čísle ještě dosaženo žádnou posloupností slov. Po C_i takovýchto průchodech je zřejmě $wc[C_i] = 0$ právě tehdy, když telefonní číslo nelze složit pomocí slov ze slovníku. Pro jiné hodnoty číslo složit lze a pro složení dobře poslouží právě pole wi. Stačí na konec věty vypsat slovo, na které ukazuje $wi[C_i]$ a posunout se v poli wi na pozici o délku tohoto slova doleva, tam se nalézá předposlední slovo hledané věty atd.

Jak již bylo řečeno, je časová složitost takového algoritmu rovna $O(P + \sum_{i=1}^{n} C_i^2)$. $O(P)$ je čas potřebný k sestavení trie a pak pro každé telefonní číslo délky C_i děláme až C_i průchodů. Paměťová složitost je ovlivněná hlavně nutností zapamatování vstupního slovníku, když budeme uvažovat, že libovolné telefonní číslo má zanedbatelnou délku oproti velikosti slovníku, a je tedy $O(P)$. Na závěr poznamenejme, že existuje řešení ještě rychlejší. Pokud bude telefonních čísel opravdu mnoho, pak jejich umístění do druhé trie ušetří až logaritmicky mnoho průchodů vůči počtu čísel. Zvýšily by se nám však nároky na paměť, protože bychom si museli pamatovat všechna telefonní čísla.

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

#define MAX_WORDS 100                    /* maximální počet slov ve slovníku */
#define MAX_WORD_LEN 100                 /* maximální délka slova */
#define MAX_DICT_SIZE 1000               /* maximální velikost slovníku */
#define MAX_NUM_LEN 100                  /* maximální délka telefonního čísla */

const char conv[26]={'1', '1', '1',  '2', '2',  '3', '3', '3',  '4', '4',    /* konvertovací tabulka */
                     '5', '5', '5',  '6', '6',  '7', '7', '7',  '8', '8',
                     '9', '9', '9',    /*  */   '0', '0', '0'};
char dict[MAX_WORDS][MAX_WORD_LEN];    /* slovník */

typedef struct _TRIE {                   /* struktura trie */
    int succ[10];                        /* potomci */
    int word;                            /* ukazatel do slovníku nebo 0 */
} TRIE, *PTRIE;

int n, w;                                /* počet čísel, počet slov */
TRIE trie[MAX_DICT_SIZE];                /* trie jako pole vrcholů */
int trie_count;                          /* počet vrcholů trie */
int word_count[MAX_NUM_LEN];             /* počet slov na dosažení cisla */
int word_idxs[MAX_NUM_LEN];              /* indexy slov ve větě */

void trie_add (char *s, int idx) {       /* číselný řetězec idx-tého slova → trie */
    PTRIE t=&trie[0];                    /* začneme v kořeni trie */
    int len=strlen (s);                  /* délka řetězce */
    for (int i=0; i<len; i++) {
        char num=s[i]-'0';               /* konverze znaku na číslo */
        int next=t->succ[num];           /* index potomka v trii */
        if (!next) {                     /* existuje potomek? */
            trie_count++;                /* pokud není potomek, vytvoříme ho */
            t->succ[num]=trie_count;     /* označ potomka */
            next=trie_count;             /* půjdeme do nového vrcholu */
        }
        t=&trie[next];                   /* zanoření o level níž */
        if (i==len-1) t->word=idx;       /* jsme již na konci slova */
    }
    return;
}
int make_sentence (char *numstr) {       /* utvoří větu pro telefonní číslo do
                                            word_count, word_indxs. Vrací 0, pokud
                                            větu pro dané číslo nelze stvořit. */
```

```
    int len=strlen (numstr);                          /* počet cifer telefonního čísla */
    for (int i=0; i<=len; i++) word_count[i]=0;    /* inicializujeme na 0 = nedosazeno */

    for (int i=-1; i<len; i++) {                      /* začneme "před" slovem */
      PTRIE t=&trie[0];                               /* začneme v kořeni */
      int wc=0;                                       /* word_count pro i od 0 jinak nula */
      if (i+1) wc=word_count[i];
      if (! (i+1) || wc)
        for (int j=i+1; j<len; j++) {
          char num=numstr[j]-'0';                     /* konverze znaku na číslo */
          int next=t->succ[num];                      /* index potomka v trii */
          if (!next) break;                           /* není-li potomek, končíme průchod */
          t=&trie[next];                              /* zanořme se do potomka */
          if (t->word && (!word_count[j])) {   /* končí slovo a pozice nedosazená? */
            word_count[j]=wc+1;                       /* zapiš nový počet slov */
            word_idxs[j]=t->word;                     /* index slova ve slovníku  */
          }
        }
    } return (word_count[len-1]);
}
int main (int argc, char **argv) {
    scanf ("%d", &w);                                 /* načteme slova do slovníku  */
    for (int i=1; i<=w; i++) scanf ("%s", dict[i]);

    for (int i=1; i<=w; i++) {                        /* převedeme slova do číselné podoby */
      char wordnum[MAX_WORD_LEN];                     /* číselný řetězec */
      int len=strlen (dict[i]);                       /* délka slova */
      for (int j=0; j<len; j++) wordnum[j]=conv[dict[i][j]-'a'];
      wordnum[len]=0;                                 /* ukončení slova */
      trie_add (wordnum, i);                          /* přidej slovo do trie */
    }

    scanf ("%d", &n);                                 /* zpracujeme jednotlivá čísla */
    for (int i=0; i<n; i++) {
      char numstr[MAX_WORD_LEN];                      /* telefonní číslo */
      scanf ("%s", numstr);                           /* načti číslo */
      if (make_sentence (numstr)) {                   /* zkusime utvořit větu */
        char sentence[2*MAX_WORD_LEN];    /* pole pro větu (i s mezerami) */
        int numlen=strlen (numstr);                   /* délka telefonního čísla */
        int j=word_count[numlen-1]-1+numlen;     /* kam zapisujeme do věty */
        sentence[j]=0;

        while (numlen>0) {                            /* pro celé telefonní číslo */
          int idx=word_idxs[numlen-1];                /* index slova ve slovníku */
          int len=strlen (dict[idx]);                 /* délka slova ve slovníku */
          j-=len;                                     /* posuneme se o délku slova vlevo */
          strncpy (&sentence[j], dict[idx], len);     /* zkopírujeme slovo */
          if (--j) sentence[j]=' ';                   /* uděláme mezeru mezi slovy */
          numlen-=len;                                /* posuneme se na další slovo */
        }
        printf ("%s -> %s\n", numstr, sentence);      /* vypíšeme větu */
      } else printf ("%s nelze složit\n", numstr);
    } return 0; }
```

17-3-3 Starosta Hafák — Jana Kravalová

Snadno odhalíme, že hranu nesmíme zjednosměrnit právě tehdy, je-li mostem. Most je totiž hrana, jejímž odebráním se graf rozpadne na dvě komponenty souvislosti. Je tedy jediným spojením mezi těmito dvěma komponentami, a když ho učiním propustným pouze pro jeden směr, tak se dostanu z první komponenty do druhé, ale ne opačně.

Tady nám poslouží algoritmus na hledání mostů z kuchařky, který dá lehce upravit pro naše účely.

Pro každý vrchol v si stejně jako v kuchařce budeme pamatovat, v jaké hloubce vůči kořeni se nachází (kořen v hloubce 0, synové 1, synové synů 2, ...) a do jaké nejnižší hladiny se umím dostat z podstromu s kořenem v. Zde se ovšem v kuchařce vyskytla chyba, kterou naštěstí mnozí hravě odhalili. Při hledání spojení do nižší hladiny nesmíme vůbec uvažovat hranu mezi otcem vrcholu v a vrcholem v. Ta totiž způsobí, že cestu do nižší hladiny najdeme vždy (každý vrchol má otce), ale není to kýžená kružnice, nýbrž dvakrát započítaná hrana, po které jsme do vrcholu v přišli. Zrada! Takhle totiž nikdy nenajdeme žádný most.

Pokud se z podstromu s kořenem v (s otcem u) neumíme dostat do hladiny nižší, než je hladina vrcholu v, pak do tohoto podstromu vede jedna jediná hrana, a to hrana (u,v). Ta je tedy mostem, v zájmu propustnosti v obou směrech ji ponecháme obousměrnou.

Pokud se z podstromu s kořenem v (s otcem u) umíme dostat do hladiny nižší, než je hladina vrcholu v, bez použití hrany (u,v), tak jsme právě našli kružnici „$u \to v \to$ nějaké vrcholy v podstromu $v \to$ (vrcholy v nižší hladině než vrchol v, ne nutně) $\to u$“. A kružnici můžeme směle zjednosměrnit, snadno vidíme, že zjednosměrnění kružnice neublíží dosažitelnosti vrcholům na této kružnici, prostě budeme „kroužit“ dokola.

Také si můžeme všimnout, že na této kružnici jsou všechny hrany dopředné, kromě té jediné, která se z hlubší hladiny vrací do nižší, a ta je zpětná.

Nakonec ještě musíme vymyslet, jak sjednotit zjednosměrňování více kružnic. Ale k tomu se stačí dohodnout, že dopředné hrany budeme zjednosměrňovat „dopředu“, čili otec $\to$ syn, a zpětné ve směru od vrcholu na hlubší hladině k vrcholu na nižší hladině („dozadu“).

Algoritmus našel všechny mosty a ponechal je obousměrné, našel i všechny kružnice a zorientoval je ve shodném směru. Takže jsme nezjednosměrnili nic závadného a naopak jsme zjednosměrnili maximum. Při reprezentaci grafu seznamem následníků získáváme časovou i paměťovou složitost $O(N + M)$.

Škoda, že většinu řešitelů kuchařka svádělа k řešení „pomocí algoritmu z kuchařky odstraním mosty, pak v dalším průchodu zjednosměrním graf a pak ještě vypíšu všechny zjednosměrněné hrany“. Ve skutečnosti všechno můžu udělat přímo v jednom jediném průchodu grafem, stačí si uvědomit, že algoritmus

na hledání mostů nejen že hledá mosty, ale i detekuje dopředné a zpětné hrany, a tak můžeme výsledky ihned vypisovat.

Za funkční řešení v čase O(N+M) bylo možno získat 9 bodů a kdo to všechno zvládl v jednom průchodu grafem, dostal 10 bodů.

```
Program Mosty;
const MaxN = 100; MaxN=10000;                  {maximální počet vrcholů a hran}
var   Sousedi:array[1..MaxM] of 1..MaxN;                  {následnici vrcholů}
      V:array[1..MaxN+1] of 1..MaxM+1;         {indexy určující, kde v Sousedi}
                                            {začínají následnici daného vrcholu}
      N,jedn,i,j:integer;              {počet vrcholů, počet jednosměrek, čítač}
      Hladina,Spojeno:array[1..MaxN] of integer;              {jako v kuchařce}

{maximálni zjednosměrnění neorientovaného grafu}
procedure Projdi(otec,x,NovaHladina:integer);
var i:integer;
begin
  Hladina[x]:=NovaHladina;                    {hladina nově nalezeného vrcholu}
  Spojeno[x]:=Hladina[x];                 {zatím víme, že z něj vede spojení do}
                                            {něj samého, tj. do té samé hladiny}
  for i:=V[x] to V[x+1]-1 do                  {projdi všechny sousedy vrcholu V}
    if Hladina[Sousedi[i]] = -1 then begin    {pokud sousední vrchol neobjeven}
      Projdi(x,Sousedi[i], NovaHladina+1);           {zkus z něj další pátrání}
      if Spojeno[Sousedi[i]] < Spojeno[x] then       {spojení do nižší hladiny}
        Spojeno[x] := Spojeno[Sousedi[i]];
      if Spojeno[Sousedi[i]] <= Hladina[x] then begin
        writeln('(',x,',',Sousedi[i],')');       {proto je toto DOPŘEDNÁ hrana}
        inc(jedn);
      end;
    end else     {vrchol Sousedi[i] již byl při průchodu navštíven a není otec}
      if (Hladina[Sousedi[i]] < Spojeno[x]) and (Sousedi[i] <> otec) then begin
        Spojeno[x]:=Hladina[Sousedi[i]];                          {ZPĚTNÁ hrana}
        writeln('(',x,',',Sousedi[i],')');
        inc(jedn);
      end;
end;

begin
  readln(N);
  jedn:=0; V[1]:=1; j:=1;
  for i:=1 to N do begin
    while not eoln do begin                              {následnici vrcholu i}
      read(Sousedi[j]); inc(j);
    end;
    readln();
    V[i+1]:=j;
  end;
  for i:=1 to N do Hladina[i]:=-1;
  for i:=1 to N do if Hladina[i] = -1 then Projdi(0,i,0);
  writeln('Počet ulic k zjednosměrnění: ',jedn);
end.
```

17-3-4 Myslitel Cibulka — Pavel Čížek

Myslitel Cibulka by z vás asi radost neměl. Došlé řešení se totiž dala rozdělit do tří skupin. V první, největší, byla řešení kvadratická. V druhé ta, u nichž autoři ani nenaznačili, proč by měla skončit (a nebylo to, jako u většiny programů zřejmé z toho, že tento cyklus proběhne $3\times$, jiný $N\times$, ...). Ta, ačkoliv byla, jak dále ukážeme, lineární, jsem hodnotil o něco hůř, jelikož byla opravdu triviální, a dokázat o nich, že jsou opravdu rychlá, je mnohem obtížnější, než vymyslet kvadratické řešení. Navíc vymyšlení kvadratického řešení bylo také obtížnější, než vytvoření tohoto triviálního.

No a konečně ve třetí skupině (dá-li se to tak nazvat, skoro by se dalo hovořit o výjimkách potvrzujících pravidlo) byla řešení lineární.

Tak ono „triviální" řešení. Vezmeme FiBoNaCiHo čísla a sečteme je „po bitech", tj. po jednotlivých cifrách. Na některých místech se vyskytnou dvojky, kterých se potřebujeme zbavit, a také číslo normalizovat. Jak na to?

Zavedeme kurzor. Budeme předpokládat, že číslo je vpravo od kurzoru normalizované, tj. kdybychom zapomněli (nebo je nastavili na nulu) na všechny cifry nižší, než je pozice kurzoru, tak dostaneme normalizované číslo. A program bude opakovat několik operací, které zřejmě nemění hodnotu čísla a zachovávají výše zmíněný invariant, dokud kurzor nenarazí na konec čísla. Zřejmě když dojde kurzor na „nultou" cifru (cifry čísla indexujeme od jedničky) tak je celé číslo normalizované a je hotovo.

Pro pohodlnější práci zavedeme ještě nultou a mínus první cifru a zadefinujeme $F_0 = 1$ a $F_{-1} = 0$. Na začátku nastavíme cifry na nula a na konci se jich opět zbavíme. To nám umožní psát pravidlo $2 \cdot F_n = F_{n+1} + F_{n-2}$ pro každé $n \geq 1$ (nemám rád okrajové podmínky) a zjednoduší dále některé úvahy.

Teď tělo cyklu. Označme C_i i-tou cifru zpracovávaného čísla a k kurzor (resp. index cifry, na kterou ukazuje).

1) Pokud $C_k \geq 1$ a $C_{k+1} = 1$, tak hodnotu cifer k a $k+1$ snížíme o 1, hodnotu C_{k+2} nastavíme na 1 (musela být 0, jelikož číslo vpravo od kurzoru je, jak předpokládáme, normalizované) a kurzor o 2 zvětšíme (mohly se nám vedle sebe dostat dvě jedničky).
2) Jinak pokud je $C_k \geq 2$ (a $C_{k+1} = 0$, jinak by se provedla předchozí větev), pak C_{k+1} nastavíme na 1, C_{k-2} zvětšíme o 1, C_k o 2 snížíme a kurzor posuneme o jedna doprava (opět možnost dvojice jedniček).
3) A pokud jsme se ještě na nějaké podmínce nezachytili, pak je na pozici kurzoru nula, nebo jednička které nula předchází, a proto se můžeme „beztrestně" kurzorem posunout o jednu pozici doleva, aniž bychom porušili náš základní invariant.

Až tento cyklus doběhne, musíme se zbavit C_0 a C_{-1}. S C_{-1} není problém, jelikož F_{-1} je 0 a cokoliv krát nula je stále nula, takže tuhle cifru můžeme jedno-

duše vynulovat. Nyní co s C_0. Jak ukážeme později, může nabývat jen hodnot nula a jedna, stačí vyšetřit případ, kdy $C_0 = 1$. Pokud je C_1 nula, tak prohodíme nultou a mínus první cifru (protože $F_0 = F_1$) a jsme hotovi. Pokud C_0 i C_1 jsou jedna, tak použijeme pravidlo o dvou jedničkách za sebou a převedeme je na jedničku na druhé cifře. Samozřejmě musíme si dát pozor, jelikož tyto operace mohou narušit normalizaci tím, že se vyskytnou dvě jedničky vedle sebe. Nicméně tato „porucha“ se vyskytuje u čerstvě zapsané jedničky a protože každá redukce dvou jedniček na jednu sníží ciferný součet, vyřešíme to v lineárním čase.

Je v celku zřejmé, že tělo cyklu nemění hodnotu čísla a že neporuší výše definovaný invariant. Horší je to ale s tím, za jak dlouho doběhne, doběhne-li vůbec. Ještě než se do toho pustíme, dokažme si jedno pomocné tvrzení.

Lemma: Každou cifru, před tím, než se na ní dostaneme kurzorem, můžeme zvýšit nejvýše o jedna. (Pokud považujeme všechny nulové cifry vpravo od čísla za již prošlé)

Důkaz: Nejdříve takové drobné pozorování. Označme L nejnižší cifru, na kterou se kurzor při běhu programu zatím dostal. Potom všechny cifry vpravo od L jsou menší než 2. To se dokáže indukcí. Nejdříve si všimněme, že jediná operace, která v těle cyklu je schopná zvětšit jedničku na dvojku, je (2), a ta to dělá vlevo od kurzoru. Pak se podíváme na to, že změnu L je schopná provést jen (3), a ta to udělá jen tehdy, je-li C_L menší než 2. A jelikož vždy platí $k \geq L$, jsme hotovi (alespoň s tímto pozorováním).

Nyní k samotnému lemmatu. Dokážeme ho indukcí dle čísla cifry (označme ho N). Na počátku je $k = L$ rovno délce čísla a pro všechna $N \geq$ (délka čísla) to zřejmě platí z předpokladů. Nyní předpokládejme, že to platí pro $N + 1$ a větší. Všimněme si opět, že jediná operace, která mění hodnotu cifry vlevo od kurzoru, je (2), a ta to dělá právě o 2 pozice vlevo. Uvažujme L stejné jako v pozorování, na začátku důkazu. Mohou nastat 4 případy:

- $L > N + 2$. Protože tělo cyklu pracuje s ciframi nejvýše o 2 pozice vlevo, tak je tato cifra ještě „nedotčená“ a má svou původní hodnotu.
- $L \leq N$. Potom jsme již přes tuto cifru přešli a není co řešit.
- $L = N + 1$. Potom z úvodního pozorování plyne, že C_{N+2} je menší než 2 a proto se (2) na pozici $N + 2$ již během programu neprovede, a proto se hodnota C_N, dokud kurzor nedojde na N, nezmění.
- $L = N + 2$. Z indukčního předpokladu víme, že hodnota C_{N+2} se zvýšila nejvýše o 1 a proto je menší, nebo rovna 3 (po sečtení bitů je maximálně 2). Pokud provedeme (2), pak hodnota C_{N+2} klesne na 1 nebo 0. Protože pro všechny cifry vpravo od L platí, že jsou nejvýše 1, a jediná operace, která je schopna zvýšit hodnotu C_{N+2} nad jedna je (2) a to jen tehdy, je-li kurzor na pozici $N + 4$ (což je vpravo od L), provede se (2) na pozici $N + 2$ nejvýše jednou.

Tím je lemma dokázáno.

Důsledek 1: Hodnota jakékoliv cifry je během výpočtu nejvýše 3, protože na začátku je nejvýše 2, dokud na ní nedojde kurzor. Zvýšit se může maximálně o 1, ve chvíli, kdy L stojí na této cifře, se tato cifra nemůže zvětšovat (důkaz analogicky jako 4. případ v důkazu lemmatu) a pokud L je vlevo od cifry, pak je tato cifra 1 nebo 0.

Důsledek 2: Hodnota C_0 a C_{-1} je maximálně 1, protože začínaly na nule a kurzor na nich při provádění cyklu nemůže stát, zvýší se tedy nejvíc na 1.

No a nyní konečně k (ne)konečnosti algoritmu a jeho časové složitosti. Použijeme k tomu techniku, která se nazývá metoda potenciálu. Původně je určená k dokazování konečnosti algoritmu, ale v některých případech ji lze použít i ke stanovení složitosti algoritmu. A co to tedy je?

Odvodíme, uhodneme, nebo jinak stanovíme funkci φ (nějaké parametry, kterými charakterizujeme stav výpočtu (ne nutně jednoznačně)), která je:

a) *klesající,* čili po provedení nějaké části výpočtu, u které je zřejmé, že doběhne za $O(\text{cosi})$ (u konečnosti stačí jen to, že doběhne) se její hodnota ostře sníží.

b) *klesá „dost“ rychle,* to znamená, že existuje konstanta $\varepsilon > 0$ taková, že při poklesu v `a)` poklesne φ alespoň o ε. (Pozn. je-li funkce φ celočíselná, což většinou je, pak je `b)` splněno automaticky.)

Pak pokud ke každým vstupním datům dovedeme shora i zdola omezit (tj. pro každá vstupní data existují konstanty K a L takové, že po celou dobu výpočtu platí $K \leq \varphi \leq L$), výpočet je konečný. (ε se „vejde“ do intervalu $[K; L]$ jen konečněkrát). Proto jsme také požadovali „podivnou“ podmínku `b)`. Vyhnuli jsme se asymptotickému blížení φ ke K a podobným patologickým případům).

Ale vraťme se k určení složitosti našeho programu. Označme k pozici kurzoru, σ součet cifer FiBoNaCiHo čísla a X pozici nejpravějšího výskytu cifry, která je větší než 1 (pokud jsou všechny cifry menší než 2, pak zadefinujeme $X = 0$). Vezměme tento potenciál: $\varphi(k, \sigma, X) = k + 2X + 3\sigma$. Pro parametry zřejmě platí ,že $k \geq 0$, $X \geq 0$ a $\sigma \geq 1$ a tedy po celou dobu běhu programu je $\varphi \geq 3$.

Na druhou stranu, označme N délku delšího čísla. Na začátku výpočtu zřejmě platí, že $X \leq N$ a $\sigma \leq 2N$ a $k = N$. Protože, jak ukážeme v dalším odstavci, je φ klesající, platí během výpočtu $\varphi \leq 9N$.

Nyní, co provede s k, X a σ tělo cyklu. Rozebereme jednotlivé větve zvlášť. Čárkované proměnné budou proměnné po provedení těla cyklu, nečárkované před tím.

1) Zřejmě nevytváří žádné číslo větší než 2, ale možná nějaké snižuje. Proto $X' \leq X$. Ciferný součet se sníží o 1, proto $\sigma' = \sigma + 1$. A kurzor posuneme o 2 doprava, tedy $k' = k + 2$. Z toho plyne, že $\varphi' \leq \varphi - 1$, tedy $\varphi' < \varphi$.

2) S ciferným součtem se nic neděje (jen ty dvě jedničky přesuneme na jiné pozice). Protože momentálně pracujeme na nejpravější cifře, která je větší než 1 (viz. pozorování v důkazu lemmatu) a protože každá cifra je během výpočtu nejvýše 3 (viz. důsledek 1), klesá cifra provedením (2) na nejvýše 1 a proto platí $X' \leq X - 1$. A kurzor se posune o jedno místo doprava. Proto $\varphi' \leq \varphi - 1$.
3) Poslední případ jen hne s kurzorem a s číslem nic nedělá. Proto φ pro změnu poklesne o 1.

Protože φ je zřejmě celočíselná, splnili jsme všechny požadavky na potenciál a výpočet tedy skončí. Nyní se na potenciál podívejme podrobněji. Během celého provádění cyklu může klesnout nejvýše o $9N$ a klesá vždy alespoň o 1. Tedy celý cyklus se provede $O(N)$ krát. Protože provedení těla cyklu stihneme v konstantním čase, můžeme tvrdit, že celý cyklus doběhne v lineárním čase. Vzhledem k tomu, že ostatní části (výpis, načtení, a zbavení se jedničky na pozici 0) zvládneme v lineárním čase, běhá celý program v lineárním čase. Paměťová složitost je zřejmě lineární.

A je to. Uffff...

[Poznámka na okraj: ačkoliv je opravdu pozoruhodné, že o tak triviálním řešení se dá dokázat, že běží v lineárním čase, zděšení ze složitosti důkazu není na místě: existují i jiná lineární řešení, která jsou trochu pracnější na naprogramování, ale obejdou se bez složitého dokazování. Například můžeme zkusit přičítat druhé číslo k prvnímu po číslicích a po každé číslici normalizovat. To sice pro některá čísla bude kvadratické, ale stačí si všimnout, že kvadratické chování nastává pouze, objeví-li se blok číslic typu 010101...01. To můžeme napravit „kompresí“ zpracovávaného čísla – v případě, že za kurzorem následuje takovýto blok, si budeme udržovat pouze jeho délku a ne pokaždé celý blok zkoumat. Hezký algoritmus založený na této myšlence najdete například na *http://www.ucw.cz/~mj/papers/fibonacci/*. –M.M.]

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

#define MaxN 1024

int main () {
  char Cislo1[MaxN+1], Cislo2[MaxN+1], Cislo3[MaxN+6];
  int Fib1[MaxN+7], Fib2[MaxN+7], Fib3[MaxN+7];   /* první 2 jsou pro 0 a -1 */
  int index, i, Delka1, Delka2, Delka;
  printf ("Zadej číslo 1:"); scanf ("%s", Cislo1);
  printf ("Zadej číslo 2:"); scanf ("%s", Cislo2);
  Delka1 = strlen (Cislo1); Delka2 = strlen (Cislo2);
  Delka = ( (Delka1<Delka2) ?Delka2:Delka1) + 7;
  for (index = 0; index < Delka; index++)
    Fib1[index] = Fib2[index] = 0;
  for (index = 0; index < Delka1; index++)
    Fib1[Delka1 - index + 1] = (Cislo1[index] == '1');
```

```
    for (index = 0; index < Delka2; index++)
      Fib2[Delka2 - index + 1] = (Cislo2[index] == '1');    /* a je převedeno na pole */
    for (index = 0; index < Delka; index ++)
      Fib3[index] = Fib1[index] + Fib2[index];    /* sečteme po bitech */
    index = Delka - 6;                            /* poslední zajímavá cifra */
    while (index > 1)                             /* index je pozice kurzoru */
      if ( (Fib3[index] >= 1) && (Fib3[index + 1] == 1)) {
        Fib3[index] -= 1;
        Fib3[++index] = 0;
        Fib3[++index] = 1;
      } else if (Fib3[index] >= 2) {
          Fib3[index] -= 2;
          Fib3[index - 2] += 1;
          Fib3[++index] = 1;
      } else index--;
    if (Fib3[1] && !Fib3[2]) {                    /* sečteno, už se jen zbavit cifer 0 a -1 */
      Fib3[1] = 0;
      Fib3[2] = 1;                                /* prohodím cifry 0 a 1 */
      index = 2;
    } else index = 1;
    while (Fib3[index] && Fib3[index + 1]) {      /* dokud to jde... */
      Fib3[index] = 0;                            /* ...vyhazujeme dvojice jedniček */
      Fib3[++index] = 0;
      Fib3[++index] = 1;
    }
    index = Delka-1;                              /* poslední definovaná cifra */
    while ( (index > 2) && (Fib3[index] == 0)) index--;    /* odbouráme nuly */
    i = 0;
    for (; index > 1; index--) Cislo3[i++] = (Fib3[index]) ? '1': '0';
    Cislo3[i] = '\0';                             /* konec řetězce */
    printf ("Součet je: %s\n", Cislo3);
    return 0;
}
```

17-3-5 Jazykozpytcova naděje — Petr Škoda

Přestože je to úloha na automaty, k jejímu řešení se dalo využít znalosti grafových algoritmů. Automat si představíme jako orientovaný graf, ve kterém je pro každý stav jeden vrchol. Z každého vrcholu vede a orientovaných hran, každá pro jedno písmeno abecedy. Řešení rozdělíme do dvou částí.

Odstranění nedostupných stavů automatu znamená odstranit ty vrcholu grafu, do kterých se nedá dostat z počátečního vrcholu p – vstupního stavu. Projdeme graf do hloubky z počátečního vrcholu a označíme si, kam všude jsme se dostali. Ostatní vrcholy jsou nedostupné. Jediná věc, na kterou si musíme dávat pozor, je, abychom označili každý vrchol pouze jednou.

Složitějším problémem je nalezení ekvivalentních stavů automatu. Ekvivalentní stavy jsou ty, které stejně odmítají a přijímají každé slovo. Nechť máme p, q ekvivalentní stavy a slovo u začínající na $x \in A$. Pak $p' = \delta(p, x)$

a $q' = \delta(q, x)$ jsou také ekvivalentní. V opačném případě bychom nalezli slovo v, na které automat ve stavu p' a q' odpoví jinak, a přidali před něj x.

Vytvoříme si pole velikosti $n \cdot n$, do kterého si uložíme, zda jsou stavy i a j ekvivalentní. Pole naplníme hodnotami a pak z něj zkonstruujeme redukovaný automat. Na začátku označíme každý vrchol ekvivalentní sám se sebou a každou dvojici, kde jeden ze stavů je přijímací a druhý nikoli, označíme jako neekvivalentní. Ekvivalenci ostatních dvojic vrcholů budeme zjišťovat rekurzivní funkcí, která pro stavy p a q(parametry) provede:

- Prozatímně je označí jako ekvivalentní.
- Zeptá se pomocí rekurzivního volání, zda jsou ekvivalentní stavy $\delta(p, x)$ a $\delta(q, x)$ pro každé písmeno v abecedě A.
- Pokud ne, přeznačí stavy jako neekvivalentní.

Teď už můžeme vytvořit redukovaný automat. Místo každé skupiny ekvivalentních stavů vytvoříme jeden stav a tyto stavy pospojujeme.

V algoritmu jsme použili pole, kde jsme uchovávali přechodovou funkci a pole ekvivalencí. Paměťová složitost je tedy $O(n^2 + n \cdot a)$. Časově nejnáročnější operací v algoritmu je výpočet ekvivalence. Pro každou dvojici stavů se ekvivalence počítá právě jednou a zahrnuje a dotazů na ekvivalenci dalších stavů. Časová složitost je $O(n^2 \cdot a)$.

```
const MaxN = 100; MaxA = 5;
var   N, A, P, F, NN, NF: Integer;
      Edges: array[0..MaxN - 1, 0..MaxA - 1] of Integer;
      Final, Reach: array[0..MaxN - 1] of Boolean;
      Equiv, Done: array[0..MaxN - 1, 0..MaxN - 1] of Boolean;
      First, Renamed: array[0..MaxN - 1] of Integer;

procedure MarkReachable(X: Integer);
var I: Integer;
begin
   if Reach[X] then Exit;
   Reach[X]:= True;
   for I:= 0 to A - 1 do MarkReachable(Edges[X, I]);
end;

function Equivalent(X, Y: Integer): Boolean;
   function AllEquiv: Boolean;
   var I: Integer;
   begin
      AllEquiv:= False;
      for I:= 0 to A - 1 do
         if not Equivalent(Edges[X, I], Edges[Y, I]) then Exit;
      AllEquiv:= True;
   end;
```

```
begin
   if not Done[X, Y] then begin
      Done[X, Y]:= True;      Done[Y, X]:= True;
      Equiv[X, Y]:= True;     Equiv[Y, X]:= True;
      Equiv[X, Y]:= AllEquiv;
   end;
   Equiv[Y, X]:= Equiv[X, Y];
   Equivalent:= Equiv[X, Y];
end;

var I, J, X: Integer;
begin
   Readln(N, A, P, F);  Dec(P);
   for I:= 0 to F - 1 do Final[I]:= False;
   for I:= 1 to F do begin Read(X);  Final[X - 1]:= True; end;
   for I:= 0 to N - 1 do
      for J:= 0 to A - 1 do begin
         Read(X);  Edges[I, J]:= X - 1;
      end;

   for I:= 0 to N - 1 do Reach[I]:= False;
   MarkReachable(P);
   for I:= 0 to N - 1 do
      for J:= 0 to N - 1 do
         if I = J then begin
            Equiv[I, I]:= True;
            Done[I, I]:= True;
         end else if Final[I] <> Final[J] then begin
            Equiv[I, J]:= False;
            Done[I, J]:= True;
         end else Done[I, J]:= False;

   for I:= 0 to N - 1 do
      for J:= I + 1 to N - 1 do
         if Reach[I] and Reach[J] and not Done[I, J] then Equivalent(I, J);

   for I:= 0 to N - 1 do
      for J:= 0 to N - 1 do
         if Equiv[I, J] then begin
            First[I]:= J;
            Break;
         end;

   NN:= 0;
   NF:= 0;
   for I:= 0 to N - 1 do
      if Reach[I] and (First[I] = I) then begin
         Inc(NN);
         Renamed[I]:= NN;
         if Final[I] then Inc(NF);
      end;
```

```
   Writeln(NN, ' ', A, ' ', First[P] + 1, ' ', NF);
   for I:= 0 to N - 1 do
      if Reach[I] and (First[I] = I) and Final[I] then Write(Renamed[I], ' ');
   Writeln;
   for I:= 0 to N - 1 do
      if Reach[I] and (First[I] = I) then begin
         for J:= 0 to A - 1 do Write(Renamed[First[Edges[I, J]]], ' ');
         Writeln;
      end;
end.
```

17-4-1 Mandarinková zeď — Petr Škoda

V některých paralelních vesmírech císař *No-san* zkrachoval, či nepřežil povstání svých nevěrných poddaných, ale jinde jeho Mandarínie dále prosperovala díky vašim radám.

Problém můžeme rozdělit na dva případy. Pokud je strážců sudý počet, je řešení jednoduché. Označme si $P(i)$ počet medailí, které vyžaduje i-tý strážce. Celkem nám bude stačit maximum z požadavků libovolných dvou sousedů – $p_m = \max(P(i) + P(i+1))$, přičemž indexy bereme cyklicky, takže $n+1 = 1$. Druhy medailí budeme označovat čísly $1..p$. Medaile rozdělíme takto: Strážci na liché pozici dáme medaile $1..P(i)$, strážci na sudé pozici dáme medaile $p_m - P(i)+1..p_m$. Každí dva sousedi se liší paritou pozice, a proto mají dohromady medaile $1..P(i), p_m - P(i)+1..p_m$ a určitě nemají žádnou oba dva, protože pak by jich měli dohromady více než p_m.

Podívejme se na lichá n. Určitě potřebujeme alespoň p_m medailí, ale můžeme jich potřebovat i více. Například třem strážcům musíme dát tolik medailí, kolik je součet jejich požadavků. Označme si $S(i) = \sum_{k=1}^{i} P(i)$ součet prvních i požadavků. Protože jeden druh medaile můžeme dát maximálně pouze m strážcům, kde $n = 2m+1$, budeme určitě potřebovat alespoň $p_s = \lceil S(n)/m \rceil$ medailí. Ukážeme, že nám bude vždy stačit $p = \max(p_s, p_m)$ medailí.

Myšlenka je asi taková, že máme na začátku množinu medailí L, které má strážce na liché pozici, a medaile P, které má strážce na sudé pozici. Protože ale poslední strážce je na liché pozici, měly by se množiny v průběhu rozdělování prohodit tak, aby poslední strážce měl jiné medaile než ten první. Medaile budeme rozdělovat speciálním způsobem. Půjdeme od prvního strážce k poslednímu a přitom jim budeme dávat medaile. Zapíšeme si medaile do nekonečné cyklické posloupnosti $1..p, 1..p, \ldots$ a budeme je přiřazovat strážcům popořadě. Označíme tuto posloupnost a, $a[i] = ((i-1) \bmod p) + 1$. Můžeme tedy explicitně zapsat, jaké medaile dostane i-tý strážce – $a[S(i-1)+1]..a[S(i)]$. Protože $p \geq p_m$, nemohou dostat žádní dva sousedé stejné medaile.

Takto rozdělujeme medaile, ale jen do té doby, dokud mají strážci na liché pozici alespoň 1 z medailí $1..P(1)$ prvního strážce. Hledáme tedy *nejmenší* k takové, že součet požadavků do k-tého lichého strážce $S(2k+1)$ je menší

nebo roven $k \cdot p$. Že takové k existuje, se můžeme například přesvědčit tak, že zvolíme $k = m$. Pak víme, že $mp \geq m \cdot p_s = m \cdot \lceil S(n)/m \rceil \geq S(n)$, takže víme, že pro $k = m$ je předpoklad splněn a vždy takové k existuje. Nyní si ukážeme, že pokud rozdělíme výše popsaným způsobem medaile prvním $2k$ strážcům, můžeme strážcům $2k + 1..n$ dávat medaile už podle parity jako pro n sudé. Podívejme se na to, jaké medaile dostane strážce na pozici $2k$. Protože k je minimální, $S(2k-1) > (k-1) \cdot p$, proto strážce na pozici $2k$ nemá žádnou z barev $p..p - P(2k + 1) + 1$ (nakreslete si obrázek). Jsme tedy schopni najít rozdělení pro p druhů medailí, ale nám stačí jenom tento počet.

Algoritmus bude velmi jednoduchý, spočteme p_m a pro n sudé vypíšeme tuto hodnotu, pro n liché si ještě spočítáme p_s a vrátíme tu větší z nich. To vše určitě zvládneme s lineárním časem i pamětí – tedy $O(n)$.

```
program MandarinkovaZed;
const
   MaxN = 1000;
var
   P: array[1..MaxN] of Integer;
   N: Integer;

I, Pm, Ps, S: Integer;
begin
   Readln(N);
   for I:= 1 to N do
      Read(P[I]);

   if N = 1 then
      Writeln(P[1])
   else
   begin
      Pm:= P[1] + P[N];
      for I:= 1 to N - 1 do
         if P[I] + P[I + 1] > Pm then
            Pm:= P[I] + P[I + 1];

      if N mod 2 = 0 then
         Writeln(Pm)
      else
      begin
         S:= 0;
         for I:= 1 to N do
            S:= S + P[I];

         Ps:= (2*S + N - 2) div (N - 1);
         if Ps > Pm then Writeln(Ps) else Writeln(Pm);
      end;
   end;
end.
```

17-4-2 Válicie — Miroslav „miEro“ Rudišín

Úlohu rozmístit Válicie na křižovatkách Mandarínie tak, aby na každé křižovatce byla právě jedna, převedeme na úlohu barvení vrcholů grafu dvěma barvami. Lze snadno nahlédnut, že obarvení prvního vrcholu jednou ze dvou barev (postavit nebo nepostavit stanici) určuje barvu ostatních vrcholů, které jsou s ním spojeny cestou. Vrcholy vzdálené o lichý počet jednotek nesmí být obarvené stejnou barvou, na rozdíl od těch na sudých pozicích, které ji musí mít stejnou. Z této úvahy hned plyne, že dvěma barvami nelze obarvit graf, který obsahuje cyklus liché délky (například trojúhelník). Protože potřebujeme minimalizovat počet stanic, vybereme si v každé komponentě souvislosti grafu tu barvu, kterou je obarveno méně vrcholů.

Algoritmus řešící úlohu může být následující. Vezmeme vrchol, obarvíme ho a všechny jeho dosud neobarvené sousedy obarvíme týmž algoritmem druhou barvou. Úloha nemá řešení, pokud nějaký soused zpracovávaného vrcholu má již stejnou barvu. V tom případě totiž existuje v grafu cyklus liché délky.

Po dokončení obarvování komponenty zkontrolujeme, jestli je barvy značící „postavit stanici“ méně než barvy druhé. Pokud ne, barvy v komponentě zinvertujeme.

Vrcholy jsou obarvované rekurzivní funkcí pracující se seznamem sousedů. Každý vrchol je zpracován nanejvýš dvakrát. Proto je časová i paměťová složitost $O(N + M)$.

```
#include <stdio.h>
#define MAX_N 1000
#define ABS (a)  ( ( (a) < 0)  ? - (a) :  (a))

int sousedi[MAX_N+1][MAX_N+1];          /* mělo by se dynamicky alokovat */
int sousedi_len[MAX_N+1];               /* ale to by každý zvládl */
int barvy[MAX_N+1];                     /* barva vrcholů */
int nvrcholu[2],  nstanic;              /* počet vrcholů dané barvy, stanic */

int obarvi (int v,  int barva) {
    barvy[v] = barva;                   /* >0 – se stanicí, <0 – bez stanice */
    nvrcholu[  (barva > 0)  ? 0 : 1 ]++;

    for  (int i=0; i<sousedi_len[v]; i++) {
      if  ( barvy[ sousedi[v][i] ] == barva ) return 0;    /* lichý cyklus */

      if  ( ABS (barvy[ sousedi[v][i] ]) < ABS (barva) )   /* pokud ještě nebyl obarven */
        if  (!obarvi (sousedi[v][i],  -barva)) return 0;   /* v tomto kole, přebarvi */
    }
    return 1;
}

int main () {
    int i,  n,  m,  v[2];

    printf ("Zadejne n a m:");
    scanf ("%d %d", &n, &m);            /* počet měst a cest */
```

```
    for (i=0; i<m; i++) {
      printf ("Zadejte %d. hranu:", i+1);
      scanf ("%d %d", v+0, v+1);
      sousedi[v[0]][sousedi_len[v[0]]++]=v[1];
      sousedi[v[1]][sousedi_len[v[1]]++]=v[0];
    }
    for (i=1; i<=n; i++)
      if (barvy[i] == 0) {                    /* chceme pouze neobarvené */
        nvrcholu[0] = nvrcholu[1] = 0;        /* počet barev v komponentě */
        if (!obarvi (i, 1))                   /* zkusime začít 'kladnou' barvou */
          break;
        nstanic += nvrcholu[ (nvrcholu[0] < nvrcholu[1]) ? 0 : 1 ];
        if (nvrcholu[0] > nvrcholu[1]) obarvi (i, -2);    /* prohodíme barvy, zlepšení */
      }
    if (i > n) {                              /* povedlo se obarvit všechny vrcholy */
      printf ("Stačí postavit %d stanic.\n", nstanic);
      for (i=1; i<=n; i++) if (barvy[i] > 0) printf ("%d\n", i);
    } else printf ("Stanice postavit nelze!\n");
    return 0;
}
```

17-4-3 Phirma — Jana Kravalová

Nejveleváženější císaři No-sane!

Dle Tvého hlubokomyslného rozkazu Ti phirma Jakobi-Čestná zasílá opravdu skvostné dary. Na stanovení jejich ceny se podíleli nejpovolanější učenci, slovutní programátoři a osvícení teoretici, kteří za použití nejďábelštějších, nejfantasknějších a nejdůmyslnějších konstrukcí, mohutného binárního stromoví roztodivných názvů, jakož i větvoví intervalového a AVL, namnoze pak půlení binárního, sestavili vzletné programy lepých tvarů.

S nejuctivějšími pozdravy
Skutečně-Nečestná, účetní

Vážená paní Skutečně-Nečestná,

již jsme se chystali Váš velkolepě vyhlížející dar přijmout, když tu jakýsi účetní nevelkých znalostí povšiml si výpočtu mnohem jednoduššího, nemnoha struktur vyžadujícího, prabídně prostého, ba až hanebně rychlého.

Poslyšte návod:

Většinu řešitelů napadla jednoduchá myšlenka – vytvořit k zadané posloupnosti $a_1, \ldots, a_N$ posloupnost částečných součtů $s_1, \ldots, s_N$, kde $s_i = \sum_{k=1}^{i} a_k$, a řešení pak hledat prostým prozkoumáním všech možných dvojic. Takový postup je sice průzračný a zaručeně vede k výsledku, ale trvá $O(N^2)$. Někteří ostřílení řešitelé objevili, že různými hrátkami se stromy můžeme vyzískat řešení v čase $O(N \log N)$, ale my si ukážeme řešení v čase $O(N)$.

Chceme tedy najít dvojici indexů i a j ($i \leq j$) takovou, aby $s_j - s_{i-1} = a_i + \ldots + a_j > 0$ a $j - i$ bylo nejvyšší možné. Navíc máme vybrat úsek s největším součtem. Využijeme nápadu s posloupností částečných součtů s_i. Dále si připravíme pomocnou strukturku – posloupnost $m_1, \ldots, m_N$, kde $m_i = max(s_i, \ldots, s_N)$ a indexy i a j, které nastavíme na začátek posloupnosti.

V každém kroku se snažíme najít nejdelší kladný úsek, který začíná prvkem i, ale děláme to jenom tehdy, když máme jistotu, že může být výhodnější než zatím nejdelší nalezený úsek. Index j tedy posouváme tak dlouho, dokud platí, že $s_i < m_{j+1}$. Dále už nesmíme j zvyšovat, protože m_j je maximem z prvků $s_j, \ldots, s_N$, takže za indexem j se částečné součty už jenom snižují (to bychom si k zatím nalezenému kladnému úseku přičítali záporné prvky).

Jakmile nalezneme poslední j, pro které ještě platí $m_j > s_i$, posuneme index i na $i + 1$ a zkusíme najít nový kladný úsek. Klíčovým pozorováním je fakt, že s indexem j se nemusíme vracet, zlepšení může přinést jedině posun dále. Kdybychom se s j vrátili zpět, můžeme už získat jenom kratší úsek než ten už dříve nalezený.

Kdykoliv nalezneme nový úsek s kladným součtem prvků, porovnáme ho se zatím nejlepším nalezeným úsekem a zapamatujeme si samozřejmě ten lepší z nich. Porovnáváme nejprve podle délky, v případě shody ještě podle součtu prvků (ten můžeme počítat v konstantním čase, protože $a_i + \ldots + a_j = \sum_{k=1}^{j} a_k - \sum_{k=1}^{i-1} a_k = s_j - s_{i-1}$).

Jak i, tak j projdou posloupnost nejhůř jednou od začátku do konce, a protože načíst a vytvořit všechna pole umíme v čase $O(N)$, má algoritmus lineární časovou složitost.

```
#include <stdio.h>
#define MAX (a, b) ( ( (a) > (b)) ? (a) : (b))
#define MAX_N 10000

int a[MAX_N+1];                          /* zadaná posloupnost */
int s[MAX_N+1];                          /* posloupnost částečných součtů */
int m[MAX_N+1];                          /* pomocná posloupnost */
int N;

int main (void) {
  int i, j;
  int left, right;                       /* hranice zatím nejlepšího úseku */

  printf ("N:"); scanf ("%d", &N);

  a[0]=s[0]=0;
  for (i=1; i<=N; i++) {
    printf ("%d. člen:", i); scanf ("%d", a+i);   /* načtení zadané posloupnosti */
    s[i]=s[i-1]+a[i];                    /* vytvoření posloupnosti částeč. součtů */
  }

  m[N]=s[N];
  for (i=N-1; i>=1; i--) m[i]=MAX (m[i+1], s[i]);   /* pomocná posloupnost */
```

```
    left=0; right=0;
    for (i=j=0; j<N; i++) {
      while (j<N && m[j+1]>s[i]) j++;      /* najdeme novou pravou hranici */
      if (j−i < right−left || s[j]−s[i] < s[right]−s[left]) continue;
                                                  /* kratší nebo s menším součtem */
      left=i;
      right=j;
    }
    if (left−right) {
      printf ("Hledaný úsek délky %d je␣", right−left);
      for (i=left+1; i<=right; i++) printf ("%d␣", a[i]);
      putchar ('\n');
    } else printf ("Žádný hledaný úsek neexistuje.\n");
    return 0;
}
```

17-4-4 Antifrňákovník — Tomáš Gavenčiak

Celkem jednoduché řešení této úlohy bylo v čase $O(n^2)$ vyzkoušet všechny dvojice kabelů vlevo a vpravo. Pro trochu složitější řešení si všimnu, že můžu položit dotaz „které pravé kabely jsou připojeny k těmto levým“ v čase $O(n)$. V takovém čase si stihnu nalevo připojit k zemnění ty kabely, které potřebuji, a zjistit, které z pravých jsou uzemněny.

Nyní si stačí vybrat vhodné podmnožiny kabelů nalevo. Kabely si očísluji $1..n$ a budu zkoumat, jaké kabely pasují ke kabelům vpravo – R_i je číslo toho kabelu nalevo, který je spojen s kabelem i vpravo. Vyberu-li nalevo nejprve kabely s číslem nedělitelným dvěma, budou mít všechny odpovídající kabely vpravo určitě R_i liché, zatímco ostatní jsou buď nezapojené nebo mají R_i sudé. Takto jsem vlastně zjistil, jak bude vypadat 0. bit čísel $R[i]$. A stejně mohu zjistit i 1., 2., ..., $(\log n)$-tý bit: zapojím vždy kabely s i-tým bitem nenulovým a nastavím tento bit odpovídajícím kabelům v R_i, čili kabelům, jejichž levý konec je připojen na zemnění a má i-tý bit nenulový. Pokud bude mít nakonec nějaký kabel $R_i = 0$, pak je určitě nezapojený, jinak bude v R_i číslo odpovídajícího levému kabelu.

Toto řešení má časovou složitost $O(n \log n)$. Navíc je to nejmenší možná složitost, což můžeme dokázat takto: i kdyby byly všechny kabely zaručeně propojeny, potřebuji zjistit, kterou z permutací mám před sebou. Těch je $n!$, potřebuji tedy získat řádově $\log(n!) \approx n \log n$ bitů, přičemž jednou odpovědí získám právě 1 bit. Toto je tedy dolní odhad slabší verze našeho problému, s nezapojenými kabely je to určitě jen složitější.

```
function testuj(i:integer):boolean; {zjistí je-li kabel i vpravo právě zapojen}
var c:char;
begin write('?',i,' ');readln(c);testuj:=c='a'; end;
```

```
var
   i,j,N:integer;
   r:array[1..N] of integer;
begin
   readln(N);
   for i:=0 to trunc(log2(N)) do begin
      for j:=1 to N do if (j and (1 shl i))<>0 then writeln('+',j);
      for j:=1 to N do if testuj(j) then r[j]:=r[j] or (1 shl i);
      for j:=1 to N do if (j and (1 shl i))<>0 then writeln('-',j);
   end;
   for j:=1 to N do if r[j]<>0 then writeln(r[j],'->',j);
end.
```

17-4-5 Jazykozpytec vrací úder — Tomáš Valla

Odvážnému štěstí přeje, praví se, a velmi podobně tomu bylo i ve čtvrté seriálové úloze. Kdo v sobě našel dosti odvahy přečíst si dlouhé a hrozivě vypadající zadání a pochopit, co se po něm vlastně chce, zjistil, že všechny úlohy jsou velmi snadné. O tom koneckonců svědčí i bodové zisky. Účelem tentokrát nebylo vymýšlet komplikované algoritmy na komplikované problémy, jako si spíše přesně uvědomit, jak spolu souvisí různé druhy dosud převedených automatů. Ale teď už k správným řešením.

Úloha 1: Chceme-li ukázat, že jazyk $L = \{0^n1^m;\ 1 \leq n \leq m\}$ lze rozpoznávat deterministickým zásobníkovým automatem koncovým stavem, zkrátka takový automat sestrojíme. DZA bude používat stavy $Q = \{l, p, f\}$, zásobníkové symboly $Z = \{z, 0\}$, počáteční stav bude l a počáteční zásobníkový symbol z, jediný přijímací stav bude f. Sada instrukcí bude následující:

$$\begin{aligned}
\delta(l,0,z) &= (l,z0) && \ldots \text{načti první 0}\\
\delta(l,0,0) &= (l,00) && \ldots \text{čti další 0}\\
\delta(l,1,0) &= (p,\lambda) && \ldots \text{začni odmazávat 0 ze zásobníku}\\
\delta(p,1,0) &= (p,\lambda) && \ldots \text{maž další 0}\\
\delta(p,\lambda,z) &= (f,z) && \ldots \text{už máme } 0^n1^n\\
\delta(f,1,z) &= (f,z) && \ldots \text{dočítej zbylé 1}
\end{aligned}$$

Princip je stejný jako u příkladu v zadání s tím rozdílem, že při načtení 0^n1^n se ještě načítá libovolný počet 1. Kdyby se při dočítání vyskytla 0, stroj se zastaví na nedefinovanou instrukci, a jelikož se nenačetlo slovo celé, bude odmítnuto.

Chceme-li ukázat, že DZA přijímajícím prázdným zásobníkem jazyk L nemůže nikdy rozpoznávat, budeme argumentovat takto: Kdybychom takový automat měli, slovo 0^n1^m by bylo přijato, tedy automat by vyprázdnil zásobník a zastavil se tak. Ale slovo 0^n1^{m+1} tím pádem už nikdy nemůže být rozpoznáno, protože automat se zastavil už o krok dříve. Proto žádný takový stroj

nemůže vůbec existovat. Podobně lze najít i regulární jazyk nerozpoznatelný DZAPZ. Bude to třeba jazyk $\{a^i;\ i \in N\}$ (proč je regulární snad již nemusíme zdůvodňovat) a použijeme téměř stejný argument.

Úloha 2: U jednoho směru převodu, tedy NZA přijímající prázdným zásobníkem na ekvivalentní NZA přijímající koncovým stavem, projde naprosto stejný postup jako u deterministických ZA, který jsme si předvedli v zadání. Druhý směr je zajímavější a již nutně musí nějak využívat nedeterminismu stroje. Mějme tedy nějaký NZAKS $M = (Q, A, Z, \delta, q_0, z_0, F)$. Do M přidáme speciální stav q_f a instrukce $\delta(q_f, \lambda, z) = (q_f, \lambda)$ pro každý zásobníkový symbol z. Kdykoli se vstoupí do stavu q_f, zásobník se vymaže a stroj se zastaví. Z každého přijímacího stavu f potom natáhneme „odbočku“ do stavu q_f pomocí instrukcí $\delta(f, \lambda, z) = (q_f, z)$ pro každý $z \in Z$ a $f \in F$. V každém přijímacím stavu se pak stroj nedeterministicky rozhodne, jestli už je to ten opravdu poslední stav (neboli slovo je celé načteno), v tom případě odbočí a přijme slovo prázdným zásobníkem. Pokud by výpočet odbočil dříve než je celé slovo přečteno, podle naší definice zásobníkového automatu bude slovo odmítnuto a nebudou tak přijímány nesmysly. Zjevně jsme tak tedy sestrojili ekvivalentní automat přijímající prázdným zásobníkem.

Úloha 3: Rozmysleme si ještě pro pořádek, jak se dá u automatů pohlížet na nedeterminismus. První úhel pohledu je takový, že stroj, může-li se v některých situacích nedeterministicky rozhodnout, je veden neomylnou intuicí, a vždy si vybere tu možnost, která vede k přijetí slova. Pokud žádná cesta k přijetí neexistuje, pak ani neomylná intuice nepomůže a slovo bude odmítnuto. Druhý pohled je ten, že nedeterministický automat je schopen paralelně provádět mnoho větví výpočtu, a tak najít cestu k přijetí, pokud taková ovšem vůbec existuje. Oba pohledy vystihují naši původní definici nedeterminismu.

Myšlenka nedeterministického přijímání jazyka všech palindromů nad abecedou $A = \{a, b\}$ je poměrně jednoduchá: budeme načítat symboly na zásobník, nedeterministicky uhodneme, jestli právě přišel střed palindromu, načež začneme zásobník vyprazdňovat a kontrolovat, jestli jsou obě poloviny symetrické. Sepišme si to přesně.

Zkonstruujeme NZA přijímající prázdným zásobníkem, jehož množina stavů bude $Q = \{l, p\}$, zásobníkové symboly $Z = \{z_0, a, b\}$, počáteční stav bude l a počáteční zásobníkový symbol z_0. Sada instrukcí bude tato:

$$\begin{aligned}
\delta(l, x, z) &= (l, zx)\ \forall x \in A, z \in Z && \ldots \text{načti levou půlku}\\
\delta(l, x, z) &= (p, zx)\ \forall x \in A, z \in Z && \ldots \text{uhodni } \textit{sudý} \text{ střed}\\
\delta(l, x, z) &= (p, z)\ \forall x \in A, z \in Z && \ldots \text{uhodni } \textit{lichý} \text{ střed}\\
\delta(p, x, x) &= (p, \lambda)\ \forall x \in A && \ldots \text{ověř pravou půlku}\\
\delta(p, \lambda, z_0) &= (p, \lambda) && \ldots \text{vyprázdni zásobník a skonči}
\end{aligned}$$

Zjevně pokud automat uhodl střed na správné pozici a obě půlky byly symetrické, slovo bude přijato. Pokud symetrické nebyly, stroj se zastaví před vyprázdněním zásobníku na nedefinovanou instrukci. Pokud stroj uhodl střed palindromu příliš brzo či příliš pozdě, nebude načteno celé slovo nebo se zastaví se na nedefinovanou instrukci před vyprázdněním zásobníku. Tyto větve výpočtu tedy nevydají špatný výsledek a automat tak skutečně přijímá právě jazyk všech palindromů.

Že na tento jazyk nestačí DZA přijímající prázdným zásobníkem, se dá odůvodnit téměř stejně jako v první úloze. Kdyby takový automat přijal palindrom a^i, nemohl by již přijmout palindrom a^{i+1}.

17-5-1 Velkovezír — Pavel Čížek a Milan Straka

Došlá řešení se dala rozdělit na tři skupiny, a to na jednak na triviální kvadratická (k nim není co dodat, pomocí částečných součtů řady se zkusily všechny možné úseky a vybral se ten s nejlepším průměrem), na řešení se složitostí $O(KN)$ (také se zkouší všechny možnosti, ale uvědomíme si, že mezi všemi posloupnostmi s maximálním průměrem existuje alespoň jedna, která má délku menší než $2K$, viz druhé pozorování) a na lineární. Jak na to?

Začněme *pozorováním*: Máme-li posloupnost s průměrem d a rozdělíme ji na dvě části, alespoň jedna z nich musí mít stejný nebo větší průměr než původní posloupnost. Nechť má jedna část délku l_1, druhá l_2. Kdyby tvrzení neplatilo (průměr první části $d_1 < d$ a také $d_2 < d$), byl by průměr celé posloupnosti:

$$\frac{l_1 \cdot d_1 + l_2 \cdot d_2}{l_1 + l_2} < \frac{l_1 \cdot d + l_2 \cdot d}{l_1 + l_2} = d$$

ostře menší než d, což není možné, protože d je její průměr. Navíc stejné pozorování se dá provést i pro opačnou nerovnost, že tedy průměr jedné z částí musí být menší nebo roven průměru celé posloupnosti.

Přimíchejme ještě toto *pozorování*: Označme $\varphi_{a\ldots b}$ průměr čísel s indexy a až b a $\varphi_{\max}$ největší průměr nějaké posloupnosti. Předpokládejme, že posloupnost s maximálním průměrem končí prvkem s indexem $L > i + K$ a že průměr všech posloupností, které končí prvkem i, není maximální (matematik by řekl, že pro každé $1 \le j < i$ platí $\varphi_{j\ldots i} < \varphi_{\max}$). Potom posloupnost s maximálním průměrem začíná na čísle s indexem větším než i.

Rozepišme, jak dopadne průměr posloupnosti začínající prvkem $j \le i$:

$$\begin{aligned}\varphi_{j\ldots L} &= \frac{\varphi_{j\ldots i} \cdot (i-j+1) + \varphi_{i+1\ldots L} \cdot (L-i)}{L-j+1} \le \\ &\le \frac{\varphi_{j\ldots i} \cdot (i-j+1) + \varphi_{\max} \cdot (L-i)}{L-j+1} < \\ &< \frac{\varphi_{\max} \cdot (i-j+1) + \varphi_{\max} \cdot (L-i)}{L-j+1} = \varphi_{\max}.\end{aligned}$$

Tedy žádné z čísel s indexem $1 \dots i$ nemůže být obsaženo v posloupnosti s maximálním průměrem.

Dost už bylo přihazování pozorování, pojďme z nich nyní vařit algoritmus. Na začátku vezmeme prvních K prvků, které budou tvořit zpracovávanou podposloupnost. V každém kroku algoritmu posuneme pravý konec zpracovávané posloupnosti o jeden prvek doprava. Její levý konec už nemusíme posouvat zpátky doleva, můžeme ho nechat tam, kde je (a druhé pozorování nám říká, že nepřijdeme o žádnou posloupnost s maximálním průměrem). Levý konec tedy nemusíme posouvat doleva, ale může se nám stát, že ho budeme muset posouvat doprava, abychom zvětšili průměr zpracovávané posloupnosti. Jak, to vyřešíme za chvilku. Takto v tomto kroku najdeme posloupnost s největším průměrem, která má pevný pravý konec a vznikla zkracováním posloupnosti z minulého kroku. (Netvrdíme, že tato posloupnost má největší průměr ze všech, které končí tímto pravým okrajem, ale z druhého pozorování víme, že jsme nezapomněli na posloupnost s maximálním průměrem, a to nám stačí.) Když si z těchto posloupností (pro každý pravý okraj máme jednu) vybereme tu s největším průměrem, najdeme určitě posloupnost s maximálním průměrem.

Jak tedy přesně vypadá krok algoritmu? Na začátku $L := 1$ a $P := K$, zpracovávaná posloupnost je prvních K prvků. V každém dalším kroku uděláme

- $P := P + 1$
- dokud existuje $L' > L$, že $P - L' + 1 \geq K$ (nezkrátíme moc) a $\varphi_{L' \dots P} > \varphi_{L \dots P}$ (zlepšíme průměr), pokládáme $L := L'$. Navíc pokud použijeme naše první pozorování a trochu se zamyslíme, zjistíme, že podmínka $\varphi_{L' \dots P} > \varphi_{L \dots P}$ (část posloupnosti má větší průměr) je stejná jako podmínka $\varphi_{L \dots L'-1} < \varphi_{L \dots P}$ (druhá část posloupnosti má menší průměr). Pro nás bude druhá varianta lepší.

Zbývá tedy vyřešit, jak zjistit, když máme L a P, jestli existuje prvek L', aby průměr posloupnosti prvků $L \dots L' - 1$ byl menší než průměr prvků $L \dots P$ ($\varphi_{L \dots L'-1} < \varphi_{L \dots P}$). Použijeme k tomu datovou strukturu, která bude kombinací fronty a zásobníku (bude umět data přidávat na jeden konec a odebírat je z konců obou), říkejme jí *frobník*. Ve frobníku budeme mít uloženou informaci o tom, jak vypadají průměry posloupností mezi prvky L a $P - K$. Přesněji řečeno v něm budou uloženy průměry posloupností $X_0 \dots X_1 - 1, X_1 \dots X_2 - 1, \dots, X_{S-1} \dots X_S - 1$ s tím, že $X_0 = L$ (začínáme vždy v L) a $X_S = P - K + 1$ (končíme vždy v $P - K$). Navíc bude vždy platit, že průměr jedné posloupnosti je menší než průměry všech posloupností, které se nacházejí ve frobníku (a tedy i v původní posloupnosti) *za* ní, matematicky řečeno $\varphi_{X_{i-1} \dots X_i - 1} < \varphi_{X_i \dots X_{i+1} - 1}$.

Data budeme ve frobníku udržovat následujícím způsobem. Na začátku v něm není nic, protože $P - K = 0$. Pak vždy, když zvětšujeme P, přidáme na konec frobníku průměr jednoprvkové posloupnosti s prvkem na inde-

xu $P - K$ (když přidáváme prvek do zkoumané posloupnosti, přidáme do frobníku prvek, který ještě může být v posloupnosti končící prvkem P). To nám ale mohlo pokazit vlastnost zvětšujících se průměrů. Pokud se tak stalo ($\varphi_{X_{S-1}\ldots X_S-1} \geq \varphi_{P-K\ldots P-K} = P - K$), budeme slučovat dvě poslední posloupnosti ve frobníku do jedné, dokud nebude platit, že průměr poslední posloupnosti ve frobníku je větší než průměr posloupnosti předposlední, případně dokud nesloučíme všechny posloupnosti do jedné.

Když jsme tedy takto upravili frobník, můžeme zkusit najít hledané L', aby průměr prvků $L \ldots L' - 1$ byl menší než průměr prvků $L \ldots P = \varphi_{L\ldots P}$. Vezmeme první posloupnost z frobníku (je to posloupnost $L \ldots L' - 1$) a pokud má průměr menší než $\varphi_{L\ldots P}$, je to naše hledaná posloupnost s menším průměrem, tedy položíme $L = L'$, a tuto posloupnost z frobníku odebereme. Takto pokračujeme, dokud je průměr první posloupnost ve frobníku menší než $\varphi_{L\ldots P}$ nebo dokud frobník nevyprázdníme.

Nyní už víme, jak algoritmus pracuje, zbývá zjistit složitost. Program se skládá z N kroků, v každém zvětšujeme P, upravujeme L (frobník) a testujeme, zda je nalezená posloupnost lepší než dosud nalezená. Kromě úprav frobníku jsou všechny tyto operace konstantní, tedy časová složitost algoritmu bez úprav frobníku je lineární. Do frobníku vložíme nanejvýš N posloupností, každá se může nanejvýš jednou sloučit a jednou vyndat, takže všechny operace s frobníkem trvají dohromady také jenom lineárně dlouho. (V jednom kroku algoritmu sice můžeme ve frobníku sloučit až $O(N)$ posloupností, ale uvědomte si, že sloučit mohu jenom to, co jsem do frobníku dal, takže slučování je celkem nanejvýš $N - K$.) Tedy časová složitost celého algoritmu je lineární, paměťová taktéž.

```
#include <stdio.h>
#define MaxN 1000

struct PolozkaFrobniku {
    int Delka;
    int Soucet;
};

int N, K, Cisla[MaxN];
int AktualniDelka, AktualniSoucet;
int Levy, Pravy;                          /* aktuálně zpracovávané řady */
struct PolozkaFrobniku Frobnik[MaxN];     /* je vysvětlen v popisu řešení */
int Vrchol, Dno;                          /* vrchol a dno zásobníku */
double NejPrumer;                         /* nejlepší úsek - průměr */
int NejLevy, NejPravy;                    /* a levý + pravý konec */

int main (void) {
    int index;
    printf ("Zadej počet čísel:"); scanf ("%d", &N);
    printf ("Zadej K:"); scanf ("%d:", &K);
    printf ("Zadej čísla:");
    for (index = 0; index < N; index++) scanf ("%d", &Cisla[index]);
```

```
    Vrchol = 0; Dno = 0;                          /* inicializace frobníku */
    AktualniDelka = K;
    AktualniSoucet = 0;
    for (index = 0; index < K; index++) AktualniSoucet += Cisla[index];
    /* Aktuální kus řady je prvních K čísel */
    Levy = 0; Pravy = K-1;                        /* konce zkoumané posloupnosti */
    NejLevy = 0; NejPravy = K-1; NejPrumer = AktualniSoucet / (double) K;
                                                  /* a je to zatím nejlepší úsek */
    for (Pravy = K; Pravy < N; Pravy++) {   /* projdeme všechny pravé konce */
      AktualniSoucet += Cisla[Pravy];
      AktualniDelka++;                            /* přidáme číslo na konec posloupnosti */
      Frobnik[Vrchol].Delka = 1;
      Frobnik[Vrchol].Soucet = Cisla[Pravy - K];    /* přidáme číslo na frobník */
      Vrchol++;
      /* a teď frobník opravíme */
      while ( (Vrchol - Dno > 1) &&               /* dokud tam jsou 2 prvky a chyba */
              (Frobnik[Vrchol-1].Soucet * Frobnik[Vrchol-2].Delka <
               Frobnik[Vrchol-2].Soucet * Frobnik[Vrchol-1].Delka))
        {
          Vrchol--;                               /* slučujeme */
          Frobnik[Vrchol-1].Soucet += Frobnik[Vrchol].Soucet;
          Frobnik[Vrchol-1].Delka += Frobnik[Vrchol].Delka;
        }
      /* nyní jdeme opravit kandidáta na maximum */
      while ( (Vrchol != Dno) &&                  /* dokud je něco ve frobníku */
              (Frobnik[Dno].Soucet * AktualniDelka <   /* a vylepšujeme průměr */
               AktualniSoucet * Frobnik[Dno].Delka))
        {
          AktualniDelka -= Frobnik[Dno].Delka;
          Levy += Frobnik[Dno].Delka;
          AktualniSoucet -= Frobnik[Dno].Soucet;
          Dno++;
        }
      if (AktualniSoucet / (double) AktualniDelka > NejPrumer) {
        /* našli jsme něco lepšího */
        NejLevy = Levy;
        NejPravy = Pravy;
        NejPrumer = AktualniSoucet / (double) AktualniDelka;
      }
    }
    printf ("Nejvyšší průměr %g má úsek od %d do %d.",
            NejPrumer, NejLevy+1, NejPravy+1);
    return 0;
}
```

17-5-2 Ranní hroše — David Matoušek

Při řešení každé úlohy je nejprve nutno pochopit zadání. To se mnohým řešitelům této úložky nepovedlo i přesto, že úloha byla zadána poměrně srozumitelně. Úkolem bylo zjistit, zda existuje nějaký interval h z množiny H a v z V, že h začíná i končí dříve než v. Tedy jedná se o úlohu zjišťovací, na kterou stačilo odpovědět ano nebo ne. Vypsání hledané dvojice intervalů navíc samozřejmě není nijak na škodu, ale nebylo potřeba. Co však již vadí, je situace, ve které řešitel vypisuje všechny vyhovující dvojice intervalů, to vede na triviální kvadratický algoritmus. Takováto řešení nechť do budoucna odradí ne víc než dva body za funkčnost.

Tato úloha se dá optimálně pořešit rozličnými přístupy. Zkusme nejprve setřídit obě množiny dohromady podle počátků intervalů. Nyní procházejme po setříděné posloupnosti. Budeme si přitom pamatovat jeden interval z množiny H, říkejme mu kandidát, který má takovou vlastnost, že pokud existuje nějaká dvojice vyhovujících intervalů, pak existuje i vyhovující dvojice, ve které se nachází náš kandidát. Na počátku si poznačme, že kandidáta ještě nemáme. Pokud tedy při průchodu narazíme na interval z množiny H, pak ho porovnáme s kandidátem. V případě, že nově nalezený interval je „lepší“, označíme ho za nového kandidáta. Nově nalezený interval A je „lepší“ než interval B právě, když jeho konec je menší.

Nyní rozeberme případ, kdy narazíme na interval z množiny V. Pokud ještě nemáme kandidáta, znamená to, že neexistuje žádný interval z množiny H takový, který má menší začátek, tehdy jdeme v posloupnosti dále. Pokud však již máme nějakého kandidáta, pak jeho začátek je díky setříděnosti menší než začátek nově nalezeného intervalu. Stačí tedy porovnat konec kandidáta s koncem nového intervalu a v případě, že kandidát má konec intervalu menší, skončíme, protože jsme právě nalezli vyhovující dvojici.

Linearita průchodu po setřídění i jeho konečnost je zřejmá. Celková časová složitost algoritmu je tedy ovlivněna tříděním, které pro neceločíselné intervaly umíme při nejlepším v čase $O(N \log N)$, kde N budiž součet velikostí množin H a V. Paměťová složitost je vůči stejnému N lineární.

Zbývá ukázat, že pokud řešení existuje, náš algoritmus ho najde. Ať tedy dvojice intervalů splňující zadání existuje, označme ji h a v, kde h je z H, v z V, pak při průchodu posloupností narazíme na h dříve než na v. Tedy jistě dojde k porovnání h s kandidátem, v případě, že je kandidát „lepší“, pak dvojice kandidát a v také splňuje zadané podmínky. V případě, že kandidát není „lepší“, pak dojde k nahrazení kandidáta intervalem h. Tedy po průchodu přes h splňuje kandidát podmínky pro hledané intervaly. Zřejmě pokud v budoucnu se kandidát zlepší, stále bude kandidát splňovat podmínky. A konečně až narazíme při průchodu na v, porovnáme jej s kandidátem a program skončí.

Na závěr zmíním, že řešení, která měla po setřídění již jen lineární průchody, tedy taková, která by se v případě celočíselných intervalů dala napsat i se setříděním v lineárním čase, byla o maličko lépe ohodnocena než ostatní řešení.

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

typedef enum { HARE, HIPPI, NOBODY } OWNER;    /* vlastník intervalu */
typedef struct { OWNER owner; int start, end; } INTERVAL;

int compare (const void *a, const void *b)      /* porovnávání intervalů do qsortu */
{ return ( ( (INTERVAL *)a)->start -  ( (INTERVAL *)b)->start); }

int main (void) {
  int h_count, v_count, hv_count;               /* # intervalů hrošíka, zajíce a obou */
  scanf ("%d%d", &h_count, &v_count);           /* načteme vstup do pole hv a setřídíme */
  hv_count=h_count+v_count;

  INTERVAL hv[hv_count];                        /* množiny H a V dohromady */

  for (int i=0; i<h_count; i++) {               /* hrošíkovy intervaly */
    scanf ("%d%d", &hv[i].start, &hv[i].end);
    hv[i].owner=HIPPI;
  }
  for (int i=0; i<v_count; i++) {               /* zajícovy intervaly */
    scanf ("%d%d", &hv[h_count+i].start, &hv[h_count+i].end);
    hv[h_count+i].owner=HARE;
  }
  qsort (hv, hv_count, sizeof (INTERVAL), compare);    /* setřídíme */

  /* projdeme a hledáme vhodné intervaly */
  INTERVAL hippi_min={.owner=NOBODY};    /* hrošíkův kandidát zatím žádný */
  for (int i=0; i<hv_count; i++) {
    /* pokud ještě nemáme kandidáta, pak kadidujeme první hrošíkův interval */
    /* nebo pokud jsme našli lepší interval, změníme kandidáta */
    if ( (hv[i].owner==HIPPI) &&
         ( (hippi_min.owner==NOBODY) || (hippi_min.end>hv[i].end))) {
      hippi_min.owner=hv[i].owner;          /* nastavime hodnoty nového kandidáta */
      hippi_min.start=hv[i].start;
      hippi_min.end=hv[i].end;
      continue;
    }

    /* v případě zajícova intervalu porovnáme s kandidátem, pokud máme */
    if ( (hv[i].owner==HARE) &&  (hippi_min.owner!=NOBODY) &&
         ( (hippi_min.start<hv[i].start) &&  (hippi_min.end<hv[i].end))) {
      printf ("Hledané intervaly existují:\n");
      printf ("Zajíček: [%d,%d]\n", hv[i].start, hv[i].end);
      printf ("Hrošík: [%d,%d]\n", hippi_min.start, hippi_min.end);
      return 0;
    }
  }
  printf ("Zadané podmínky nesplňují žádné dva intervaly.\n");
  return 0;
}
```

17-5-3 Nouze V-dáli-hrocha — Martin Mareš

Poslyšte příběh kratochvilný o tom, jak hrošík v nouzi poprosil o pomoc kmotřičku Rekurzi (ťuky ťuk!) a jak to dopadlo.

Maličký hrošík to asi ještě neví, ale to, co potřebuje, je vypsat všechny *permutace* množiny $\{1, \ldots, N\}$ (čili všechna možná uspořádání těchto čísel) tak, aby se sousední permutace lišily prohozením právě jedné dvojice prvků (máte-li rádi cizí slovíčka, tak jednou *transpozicí*). A my vyřešíme rovnou těžší variantu úlohy, která po nás žádá, aby se jedinou transpozicí lišila i první a poslední permutace.

Jak na to? Všimněme si, že všechny permutace N prvků získáme z permutací $N-1$ prvků tak, že do každé původní permutace vložíme prvek N postupně na všechna možná místa. Čili pokud už víme, že pro $N = 2$ existují permutace 12 a 21, pak pro $N = 3$ můžeme z 12 získat 312, 132 a 123, zatímco z 21 získáme 321, 231 a 213. Když si uvědomíme, že míst, kam vložit nový prvek, je vždy N, dostaneme ihned vzoreček, který nám řekne, že permutací N prvků je $p(N) = N \cdot p(N-1)$, čili $p(N) = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1$ (obvykle značíme $N!$, tedy faktoriál z N).

Vyzbrojeni tímto pozorováním získáme ihned jednoduchý algoritmus, který nám všechny permutace vygeneruje. Pokud je $N = 1$, vrátíme ihned jedinou permutaci, a to jedničku samu. Pokud je $N > 1$, rekurzivním zavoláním našeho algoritmu vyřešíme úlohu pro $N-1$, čímž získáme nějaké permutace p_1 až $p_{(N-1)!}$. Nyní do nich budeme vkládat N-tý prvek, přičemž do p_1 ho vložíme nejdříve na poslední pozici, pak na předposlední, atd. až na první, zatímco u p_2 budeme postupovat popředu, u p_3 opět pozpátku atd.

Nejlépe to asi bude vidět na příkladu:

Pro $N = 1$: 1
Pro $N = 2$: 12 21
Pro $N = 3$: 123 132 312 321 231 213

Tak dosáhneme, že se sousední permutace liší jen jednou transpozicí: mezi dvěma sousedními permutacemi vzniklými z jedné p_i se přesunulo pouze N na sousední pozici, zatímco mezi poslední vzniklou z p_i a první z p_{i+1} zůstalo N na místě a změnily se pouze ostatní prvky, ovšem podle stejného algoritmu, takže také s jedinou transpozicí [ejhle, důkaz indukcí].

První vygenerovaná permutace bude určitě $12 \ldots N$ (každý prvek startuje vpravo). Jak ale bude vypadat ta poslední? Jelikož pro $N > 2$ je $(N-1)!$ sudé číslo, budeme v $p_{(N-1)!}$ posouvat N-kem zleva doprava, takže N skončí na poslední pozici. Indukcí nahlédneme, že tak dopadnou i všechny ostatní prvky až na jedničku a dvojku, které zůstanou prohozené. Proto poslední permutace bude $2134 \ldots N$, přesně, jak potřebujeme.

Náš algoritmus má ale jeden velký háček: potřebuje z rekurze vracet celý vygenerovaný seznam permutací, takže má paměťovou složitost $O(N \cdot N!)$.

To věru není dobré, ovšem můžeme to snadno napravit: připravíme si už na začátku celou počáteční permutaci $12\ldots N$ a použijeme rekurzivní proceduru, která pro dané i proskáče aktuální permutaci prvkem i a mezi každými dvěma skoky zavolá sama sebe pro $i+1$. Jen si pro každý prvek potřebujeme pamatovat, jestli jsme jím naposledy skákali doleva nebo doprava, abychom správně střídali směry, a také se nám bude hodit udržovat si *inverzi* permutace, čili pole, které nám pro každý prvek řekne, kde se zrovna v permutaci nachází.

Všimněte si, že při proskakování i-čkem nás prvky větší než i nebudou rušit, protože jsou bezpečně uklizeny na jednom či druhém kraji permutace, takže opravdu stačí i-čkem posouvat na sousední pozici ve správném směru. Tím jsme vlastně mimoděk splnili mnohem víc, než po nás zadání chtělo: používáme jen transpozice *sousedních* prvků.

Zbývá rozebrat složitost: Naše rekurzivní procedura potřebuje jen konstantní čas na vymyšlení jedné permutace, leč na její vypsání je potřeba čas lineární. Proto pokud chceme vypisovat celé permutace, časová složitost nutně dosáhne $O(N \cdot N!)$ a lépe to ani nejde, protože je to čas lineární ve velikosti výstupu; kdyby nám stačilo vypsat posloupnost prohazování, zvládli bychom to v čase $O(N!)$ a opět to lépe nemůže jít. Paměti spotřebováváme lineární množství (lineárně velká globální pole a konstantně na každou z N úrovní rekurze).

Pozorný čtenář si asi povšimne, že argument s velikostí výstupu na jednu nohu pokulhává, protože výstupem je přeci *desítkový* zápis permutace, ve kterém na každý prvek spotřebujeme řádově $\log N$ číslic, takže celý zápis musí měřit $O(N \log N)$ namísto $O(N)$. To je i není pravda, tato potíž je totiž důsledkem toho, že jsme si nikdy paměťovou složitost (ani velikost vstupu a výstupu) nezavedli precizně. Dá se zavést dvěma způsoby: buďto můžeme počítat složitosti na chlup přesně a měřit velikost vstupu a výstupu i zabranou paměť v bitech (což „opravdová“ teorie složitosti skutečně dělá a zde jí vyjde $N \cdot \log N$), nebo si zvolíme jako základní jednotku jeden integer (rozumné velikosti, řekněme polynomiální v N, abychom předešli trikům à la naskládání celého vstupu do jediného integeru) a vše měříme v integerech. V KSP obvykle používáme ten druhý, výrazně jednodušší (i když někdy zbytečně hrubozrnný) přístup, a ten nám v tomto případě říká, že výstup je velký jenom $O(N)$. Podobně je to s časovou složitostí: v druhém případě považujeme každou aritmetickou operaci za konstantně rychlou, v prvním její složitost závisí na počtu bitů čísel, se kterými operace počítá. Toto téma ještě nakousneme v úvodu k dalšímu ročníku KSP.

```
program NouzeJezNaucilaVDaliHrochaRekurzi;
type index=1..16;                                        { Rozsah čísel prvků }
var N:index;                                                   { Počet prvků }
    a,b:array [index] of index; { Právě zpracovávaná permutace a její inverze }
    dir:array [index] of -1..1;            { Kterým směrem putuje který prvek }
    i:index;                                { Pomocná proměnná prchavého významu }
```

```
procedure show;                                    { Vypíše aktuální pořadí }
var k:index;
begin
   for k:=1 to N do write(a[k], ' ');
   writeln;
end;

procedure swap(i,j:index);            { Prohodí dva prvky, vypíše nové pořadí }
var k:index;
begin
   k:=a[i]; a[i]:=a[j]; a[j]:=k;
   b[a[j]]:=j; b[a[i]]:=i;
   show;
end;

procedure Vdalihroch(i:index);   { The core of the poodle: posouvá i-tý prvek }
var j:index;
begin
   if i>N then exit;                          { Ale vždyť tolik jich nemáme! }
   for j:=1 to i-1 do begin            { Posouvat ho budeme celkem (i-1)-krát }
      Vdalihroch(i+1);        { Mezitím vždy prostrkáme prvky s většími čísly }
      swap(b[i], b[i]+dir[i]);                { A posuneme ve správném směru }
    end;
   Vdalihroch(i+1);                         { Ještě jednou pro poslední pozici }
   dir[i] := -dir[i];                         { Pamatuj, příště půjdeme opačně }
end;

begin
   read(N);                                                { Velikost městečka }
   for i:=1 to N do begin                            { Prostoduchá inicializace }
      a[i] := i;  b[i] := i;  dir[i] := -1;
    end;
   show;                                             { Ukážeme, kde jsme začali }
   Vdalihroch(1);                                              { Go! Go! Go! }
end.
```

17-5-4 Kudy tudy cestička — Zdeněk Dvořák

Začneme tím, že si určíme, v jakém pořadí po sobě následují zadané body na obvodu mnohoúhelníka (řekněme proti směru hodinových ručiček). Jeden ze způsobů, jak to udělat, je vybrat si nejlevější bod bod A a ostatní body setřídit sestupně podle úhlu, který svírá jejich spojnice s bodem A s osou x. Možná o něco jednodušší než si pro každý takový vektor počítat tento úhel, je pouze umět pro libovolné dva takové vektory rozhodnout, který z nich je má tento úhel větší. To poznáme podle znaménka jejich vektorového součinu: Pokud je tento součin kladný, leží druhý vektor v levé polorovině určené prvním vektorem, a tedy je úhel druhého vektoru větší. Povšimněte si, že pokud vektory porovnáváme tímto způsobem, můžeme si bod A vybrat libovolně, tj. nemusíme ani hledat nejlevější zadaný bod.

Nyní uvažme počáteční úsek P' libovolné nekřížící se cesty P, která projde všechny vrcholy. Vrcholy navštívené P' tvoří souvislý interval I na obvodu mnohoúhelníka (bráno cyklicky, tedy za N-tým vrcholem následuje první). Kdyby tomu tak nebylo a P' by nějaký vrchol w přeskočil, cesta P by se bez křížení nemohla dostat zároveň do w a do ostatních vrcholů. Z toho plyne, že poslední vrchol u úseku P' musí být jeden z krajních bodů intervalu I a následující vrchol v cesty P je jeden z maximálně dvou vrcholů, které sousedí s krajními body I na obvodu mnohoúhelníka (samozřejmě u a v musí být také spojeny pěšinkou).

Nabízí se řešení zvolit si nějaký vrchol a poté procházet všechny cesty, které v něm začínají. Nicméně podle pozorování z předchozího odstavce může být možné každý počáteční úsek rozšířit dvěma způsoby, tedy všech takových cest může být řádově až 2^N. To je příliš mnoho a přímočarý program založený na této myšlence by byl neúnosně pomalý.

Označme si $\ell(y, z)$ délku pěšinky mezi dvěma vrcholy y a z (tato hodnota je ∞, pokud mezi vrcholy y a z pěšinka není). Procházet všechny cesty je beznadějné, ale všimli jsme si, že počáteční úsek libovolné nekřížící se cesty odpovídá nějakému intervalu. Intervalů není mnoho (jen řádově N^2), zkusme toho využít. Je pro nás celkem nepodstatné, jak přesně cesta vypadá uvnitř intervalu, zajímá nás pouze její délka. Budeme si tedy pro každý interval mezi vrcholy s čísly u a v počítat délku nejkratší cesty, která pokryje interval u a v a navíc skončí v předepsaném vrcholu x (kde x je buď u nebo v). Označme si délku nejkratší takové cesty $L(u, v, x)$. Nechť bez újmy na obecnosti $x = v$. Předposlední vrchol této cesty potom je buď u nebo $v - 1$, a z těchto možností si chceme vybrat tu lepší. Takto dostáváme, že

$$L(u, v, v) = \min(L(u, v-1, u) + \ell(u, v), \\ L(u, v-1, v-1) + \ell(v-1, v)).$$

Z tohoto vztahu již snadno všechny hodnoty $L(u, v, x)$ spočítáme – k jejich výpočtu potřebujeme znát hodnoty L pro kratší intervaly, uděláme si tedy tabulku, do níž budeme počítat tyto hodnoty postupně podle rostoucí délky intervalu. Ještě zmiňme, že pro intervaly délky 0 (tj. jednotlivé vrcholy) si nastavíme $L(v, v, v) = 0$. Tato tabulka bude mít velikost $2N^2$ a každé její políčko spočítáme z předchozích hodnot v konstantním čase. Délku nejkratší cesty, která projde všechny vrcholy, pak dostaneme jako minimum z hodnot $L(v, v-1, v)$ pro všechny vrcholy v.

Zbývá přijít na to, jak najít tuto cestu, ne jen její délku. Při výpočtu hodnoty $L(u, v, x)$ si můžeme zapamatovat, který vrchol má být předposlední. Pak snadno cestu zkonstruujeme odzadu – začneme posledním vrcholem v_N, k němu si najdeme předposlední v_{N-1}, k tomu pak v_{N-2}, atd., až dokud nedojdeme k prvnímu vrcholu.

Třídění vrcholů na obvodu mnohoúhelníka nám zabere čas $O(N \log N)$, délku cesty si určíme vyplněním tabulky v čase $O(N^2)$ a cestu samotnou nalezneme v čase $O(N)$, výsledná časová složitost je tedy $O(N^2)$. Paměťová složitost je dána velikostí tabulek L a ℓ a je tedy $O(N^2)$. Povšimněte si, že pokud bychom chtěli znát pouze délku nejkratší cesty P a nepotřebovali bychom P vypsat, stačilo by nám pole L velikosti $O(N)$ – počítali bychom si hodnoty $L(u, v, x)$ podle vzrůstající délky k intervalu mezi vrcholy u a v a pamatovali bychom si pouze dva řádky tabulky – $(k-1)$-ní a k-tý.

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

#define MAXN 100
#define NEKONECNO 100000

struct bod {
    int x, y;                                   /* Souřadnice bodu. */
    int cislo;                                  /* Číslo bodu v pořadí */
};                                              /* podél obvodu mnohoúhelníka. */

static int N;
static struct bod body[MAXN];
static int vzdalenost[MAXN][MAXN];

static int L[MAXN][MAXN][2];                    /* Pole L(u,v,x): L[u][v][0] pokud x = u
                                                   L[u][v][1] pokud x = v. */
static int predposledni[MAXN][MAXN][2];         /* Předposl. vrchol cesty, index jako L. */

static int porovnej_smery (const void *a, const void *b) {  /* Porovná směry vektoru */
    const struct bod *ba = a;                   /* od bodu a k body[0] */
    const struct bod *bb = b;                   /* a od bodu b k body[0]. */
    int dxa, dya, dxb, dyb;

    dxa = ba->x - body[0].x;
    dya = ba->y - body[0].y;
    dxb = bb->x - body[0].x;
    dyb = bb->y - body[0].y;

    return -(dxa * dyb - dya * dxb);
}

static void nacti (void) {                      /* Načte vzdálenosti a body a ty setřídí */
    int u, v, l;                                /* dle pořadí na obvodu mnohoúhelníka. */

    scanf ("%d", &N);
    for (v = 0; v < N; body[v].cislo = v, v++)
      scanf ("%d%d", &body[v].x, &body[v].y);

    qsort (body + 1, N - 1, sizeof (struct bod), porovnej_smery);

    for (u = 0; u < N; u++)
      for (v = 0; v < N; v++)
        vzdalenost[u][v] = NEKONECNO;

    while (scanf ("%d%d%d", &u, &v, &l) == 3)
      vzdalenost[u - 1][v - 1] = vzdalenost[v - 1][u - 1] = l;
}
```

```
  /* Rozšíří nejlepším způsobem interval ⟨u, v⟩ přidáním dalšího vrcholu cesty dalsi. */
  /* Délku rozšířené cesty uloží do delka a index předposledního vrcholu do predp. */
static void rozsir_usek (int u, int v, int dalsi, int *delka, int *predp) {
    int cislo_u = body[u].cislo;
    int cislo_v = body[v].cislo;
    int cislo_dalsi = body[dalsi].cislo;
    int lu, lv;

    if (L[u][v][0] == NEKONECNO || vzdalenost[cislo_u][cislo_dalsi] == NEKONECNO)
      lu = NEKONECNO;
    else
      lu = L[u][v][0] + vzdalenost[cislo_u][cislo_dalsi];

    if (L[u][v][1] == NEKONECNO || vzdalenost[cislo_v][cislo_dalsi] == NEKONECNO)
      lv = NEKONECNO;
    else
      lv = L[u][v][1] + vzdalenost[cislo_v][cislo_dalsi];

    if (lu < lv) {
      *delka = lu;
      *predp = u;
    } else {
      *delka = lv;
      *predp = v;
    }
}

static void vypln_polozku (int u, int v, int z) {   /* Vyplní L[u][v][z]. */
    int u1, v1;

    u1 = (u + 1) % N;
    v1 = (v + N - 1) % N;

    if (z)                                          /* Určujeme L(u, v, v). */
      rozsir_usek (u, v1, v, &L[u][v][z], &predposledni[u][v][z]);
    else                                            /* Určujeme L(u, v, u). */
      rozsir_usek (u1, v, u, &L[u][v][z], &predposledni[u][v][z]);
}

static void vypln_L (void) {                        /* Vyplní tabulku L. */
    int u, v, l, z;

    for (v = 0; v < N; v++) {
      L[v][v][0] = 0;
      L[v][v][1] = 0;
      predposledni[v][v][0] = -1;
      predposledni[v][v][1] = -1;
    }

    for (l = 1; l < N; l++)
      for (u = 0; u < N; u++) {
        v = (u + l) % N;
        for (z = 0; z < 2; z++)
          vypln_polozku (u, v, z);
      }
}
```

```
  /* Vypíše cestu pokrývající interval ⟨u, v⟩ a končící vrcholem x (x je buď u nebo v). */
static void vypis_cestu (int u, int v, int x) {
    int z, predp;
    int u1, v1;

    u1 = (u + 1) % N;
    v1 = (v + N - 1) % N;

    z = (v == x);
    predp = predposledni[u][v][z];
    if (predp == -1) {
       printf ("%d", body[x].cislo + 1);
       return;
    }

    if (z) vypis_cestu (u, v1, predp);
    else vypis_cestu (u1, v, predp);

    printf (" %d", body[x].cislo + 1);
}

int main (void) {                                   /* Hlavní program. */
    int v, minv, min;

    nacti ();
    vypln_L ();

    min = L[0][N - 1][0];
    minv = 0;
    for (v = 1; v < N; v++)
       if (L[v][v - 1][0] < min) {
          min = L[v][v - 1][0];
          minv = v;
       }

    if (min == NEKONECNO)
       printf ("Cesta neexistuje.\n");
    else {
       printf ("Cesta mající délku %d:\n", min);
       vypis_cestu (minv, (minv + N - 1) % N, minv);
       printf ("\n");
    }

    return 0;
}
```

17-5-5 Jazykozpytec se loučí — Tomáš Valla

Nebude zřejmě na škodu, když si ještě jednou pořádně rozmyslíme, co jsou to vlastně gramatiky. Gramatika je nástroj, který se používá pro přesný formální popis jistého jazyka. Abychom mohli o určité gramatice diskutovat, jaký jazyk vlastně popisuje, hodí se na chvíli na ni pohlížet jako na stroj, který postupně generuje slova podle přepisovacích pravidel. Takový výpočet je v principu nedeterministický, typicky totiž bývá na výběr několik pravidel, které se

v daném okamžiku mohou použít. Některé větve výpočtu jsou slepé – pokud se v nich expandované slovo ocitne, nikdy z něj již nevymizí neterminální symboly a výpočet se nezastaví. Všechny možné větve výpočtu, které naopak úspěšně skončí odstraněním všech neterminálů, potom svým výsledným slovem tvoří jazyk gramatiky. Chceme-li sestrojit gramatiku popisující nějaký jazyk L, musíme jednak zajistit, aby se sadou přepisovacích pravidel bylo možno vytvořit všechna slova jazyka L, ale hlavně ukázat, že všechny ostatní větve výpočtu jsou slepé a gramatika tak netvoří slova nepatřící do L.

Úloha 1: První úloha je snadná, gramatiku pro jazyk $\{a^i b^j c^k;\ 1 \leq i \leq j \leq k\}$ vytvoříme úpravou příkladu ze zadání. Nosná myšlenka bude následující: kromě původních pravidel z příkladu, které zajišťovaly namnožení stejného počtu symbolů a, b a c, dodáme ještě pravidlo na přimnožení libovolného množství symbolů b a c, od obou ovšem stejný počet, a konečně ještě jedno pravidlo pro přimnožení libovolného počtu samotných symbolů c. Zjevně tak bude v každém okamžiku platit $1 \leq i \leq j \leq k$, a každá platná kombinace počtů i, j, k bude naší gramatikou pokryta.

Tedy přesně, sestrojíme gramatiku $G = (V_N, V_T, S, P)$, kde množina neterminálů bude $V_T = \{a, b, c\}$, množina neterminálů $V_N = \{S, B, C, X\}$, startovní symbol S a sada přepisovacích pravidel tato:

$$\begin{aligned} S &\to aSBC \mid abC \\ CB &\to BC \quad (CB \to XB,\ XB \to XC,\ XC \to BC) \\ bB &\to bb \\ bC &\to bc \mid bbCC \mid bCC \quad \ldots \text{množíme } bc \text{ a } c \\ cC &\to cc \end{aligned}$$

Abychom byli poctiví, bylo by ještě třeba si přesně zdůvodnit, že gramatika nevydá nepatřičná slova a nechtěné větve výpočtu tedy skončí jako slepé. Věříme však, že máte již dostatek zkušeností a znalostí, abyste si to po chvíli dívání na sadu pravidel bez problémů uvědomili sami.

Úloha 2: Druhá úloha se ukázala být pro některé řešitele oříškem, a občas tvrdili, že gramatiku popisující jazyk $\{a^{2^n};\ n \in N\}$ nelze sestrojit, kupodivu i s „důkazem“. To samozřejmě není pravda, příslušná gramatika existuje a my si jednu takovou ukážeme.

V první fázi nejdříve vytvoříme jeden symbol A. V každé další fázi potom rozmnožíme všechny symboly a na dvojnásobný počet. Problém ovšem je, jak toho v rámci jedné fáze dosáhnout. Jediné pravidlo typu $A \to AA$ by zjevně násobným použitím produkovalo i jiné počty, než 2^n.

Pomůžeme si speciálním neterminálním symbolem K, „kurzorem“, který bude běhat po slovu, vždy zleva doprava, přeskočí každé A a zdvojí ho při tom.

Budeme při tom potřebovat poznačit si, kde aktuální slovo začíná a končí, obalíme si ho tedy na začátku symboly L a R, které se mohou změnit na A. Zbývá chytře navrhnout sadu přepisovacích pravidel P tak, aby gramatika nevydávala špatná slova.

$$\begin{aligned}
S &\to a && \ldots \text{ošetři jednopísmenné slovo}\\
S &\to LR && \ldots \text{obal slovo mezemi}\\
L &\to A && \ldots \text{meze jsou v podstatě skryté } A\\
R &\to A\\
L &\to LAK && \ldots \text{vytvoř vlevo nový kurzor}\\
KA &\to AAK && \ldots \text{přeskoč } A \text{ a zdvoj ho; podvádíme}\\
KR &\to AR && \ldots \text{kurzor dorazil na konec, zruš ho}\\
A &\to a && \ldots \text{a nahraď neterminály terminály}
\end{aligned}$$

Všimněme si, že kurzorů může být v jednom okamžiku ve slově i více, ale protože zanikají až na konci slova, ničemu to nepřekáží a správně zdvojnásobí všechna A. Zdvojovací pravidlo není kontextové, ve skutečnosti tedy použijeme známý trik:

$$\begin{aligned}
KA &\to XA\\
XA &\to XK\\
XK &\to AAK
\end{aligned}$$

Formálně bude naše gramatika G čtveřice (V_N, V_T, S, P), kde neterminální symboly jsou $V_N = \{S, A, L, R, K, X\}$ a terminální jsou $V_T = \{a\}$.

Protože kurzor může zaniknout pouze až když dorazí úplně napravo, dojde tak ke správnému počtu zdvojnásobení všech symbolů A, gramatika tedy negeneruje nežádoucí slova. Naopak, pro slovo délky 2^n stačí gramatice vytvořit $n-1$ kurzorů, které už se postarají o umocnění.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	25	251
1.	Miroslav Klimoš	G Lanškr	0	25	234
2.	Josef Pihera	G Strakon	2	25	212
3.	Ondřej Bílka	G Zlín	3	24	185
4.	Jan Pelc	G UBrod	3	22	164
5.	Pavel Klavík	G Chrudim	2	25	154
6. – 7.	Zbyněk Konečný	GKptJaroš	2	24	149
	Peter Perešíni	GJGTajov	3	17	149
8.	Adam Zivner	G UBrod	3	24	148
9.	Peter Černo	GĽŠtúra	4	14	130
10.	Miroslav Cicko	GJGTajov	4	13	123
11.	Jan Bulánek	G Klatovy	4	14	107
12.	Roman Smrž	GOhradní	1	15	102
13.	Jakub Kaplan	GJKTyla	1	19	100
14.	Martin Koníček	G UBrod	4	12	93
15.	Jan Hrnčíř	GFXŠaldy	3	19	89
16.	Lukáš Lánský	GJKTyla	1	19	81
17.	Petr Kratochvíl	G SvětláNS	2	20	77
18.	Cyril Hrubiš	G Bílovec	3	13	60
19.	Eva Schlosáriková	G Piešťany	4	14	59
20.	Martin Čech	G UBrod	4	7	52
21.	Tomáš Herceg	G Třebíč	2	16	47
22.	Stanislav Basovník	G Kroměříž	4	5	43
23.	Tereza Klimošová	G Lanškr	3	4	42
24.	Daniel Marek	GZborov	3	5	34
25.	Martin Kupec	GMendel	3	10	33
26.	Zbyněk Falt	GNeumannov	4	5	32
27.	Michal Pavelčík	G UBrod	2	7	31
28.	Adam Ráž	GBudějo	2	7	28
29.	Josef Špak	GJírovco	2	5	25
30.	Lukáš Špalek	G Čadca	4	6	24
31.	Ondřej Bouda	GKptJaroš	2	3	18
32.	Marian Kaluža	GHavlíčkov	2	8	16
33. – 35.	Jiří Cabal	SPŠ DvKrál	2	8	15
	Ondřej Garncarz	G Příbor	4	5	15
	Martin Kahoun	GJNerudy	2	3	15
36.	Jan Palenčar	G Martin	2	2	14

37.	Martin Podloucký	G Strážnic	4	3	12
38. – 41.	Jakub Jenis	GsvCyrMet	1	2	11
	Hana Klempová	GUBalvanJN	4	2	11
	Jakub Porod	G Týn nV	2	5	11
	Ján Zahornadský	GZborov	4	3	11
42. – 44.	Lukáš Beleš	G Čadca	4	1	10
	Jakub Benda	GJNerudy	2	3	10
	Michal Vaner	G Turnov	3	1	10
45. – 47.	Kateřina Böhmová	G Rožnov	3	3	8
	Jiří Machálek	G Holešov	3	2	8
	Petr Soběslavský	GJHeyrovs	4	2	8
48. – 49.	Daniel Sedláček	SPŠE Hav	1	2	7
	Filip Šauer	G Klatovy	4	2	7
50.	Jiří Nohavec	G Domažl	4	3	6
51. – 55.	Dalibor Adamčík	SPŠE Preš	2	2	5
	Tomáš Ehrlich	G Holešov	2	4	5
	Petr Musil	G MBuděj	3	3	5
	Jan Staněk	GKptJaroš	3	3	5
	Zdeněk Vilušínský	G Turnov	4	4	5
56. – 57.	Florián Danko	SPŠEtech	2	2	2
	Martin Vařák	G Bílovec	2	3	2
58. – 59.	Tamara Kuštárová	GBiling	0	2	1
	Petr Zimčík	G UBrod	1	1	1
60. – 62.	Miroslav Hovorka	GJateční	4	1	0
	Adrián Lachata	G Svidník	3	3	0
	Michal Onderko	SPŠ Karviná	3	1	0

Obsah

Milan Straka a kolektiv

Korespondenční seminář z programování
XVII. ročník

Autoři a opravující úloh:
Pavel Čížek, Zdeněk Dvořák, Tomáš Gavenčiak,
Jana Kravalová, Pavel Machek, Martin Mareš,
David Matoušek, Marek Sulovský, Milan Straka,
Petr Škoda, Tomáš Valla

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 158. publikaci.

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.

S jejich pomocí ročenku vysázel Milan Straka.

Korektury provedla Jana Kravalová.

Ilustrace (včetně té na obálce) vytvořil Martin Kruliš.

Sazba byla provedena písmem Computer Modern v progamu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 132 stran
Náklad 300 výtisků
Praha 2005

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji!

ISBN 80-86732-00-8

ISBN 80-86732-00-8