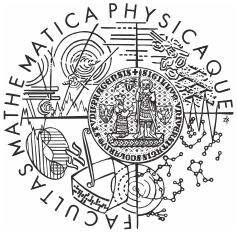
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XIII. ročník – 2000/2001



Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XIII. ročník – 2000/2001

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

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

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

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

Náš korespondenční seminář není ojedinělou aktivitou svého druhu – 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://atrey.karlin.mff.cuni.cz/ksp/*

Zadání úloh

13-1-1 Logické výrazy — 10 bodů

Firma Paleologic se rozhodla uvést na trh svůj nový kalkulátor pracující mimo jiné i s logickými výrazy. Protože doba uvedení na trh se blíží a modul pracující s výrazy dosud nebyl vytvořen, požádali vás vývojáři této renomované společnosti o pomoc.

Na vstupu váš program dostane logický výraz a nemá udělat nic složitějšího, než na výstup vypsat jeho výsledek. Zadaný logický výraz (pro jednoduchost předpokládejte, že je korektní) se skládá z čísel ve dvojkové soustavě, operátorů (`!`, `&`, `|`, `^`) a závorek (`(` a `)`). Operátor `!` je unární a má nejvyšší prioritu ze všech operátorů. Ostatní operátory jsou binární, přičemž `&` má vyšší prioritu než `|` a ten má vyšší prioritu než `^`.

Jednotlivé operátory jsou definované pro dvojice bitů, resp. pro jeden bit v případě `!` (na definiční tabulky jednotlivých operátorů se můžete podívat na konci tohoto zadání). Na čísla se pak aplikují po bitech. Tedy např. `1100 & 0101` je `0100`.

Příklad: `101|0010&!(111^011) = 111`

Tabulky operátorů:

A	B	&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

A	!
0	1
1	0

13-1-2 Centrum — 10 bodů

V Railetánii se rozhodli vybudovat nové ředitelství drah. Aby úředníci nemuseli jezdit moc daleko, je třeba ho vybudovat někde „uprostřed“ železniční sítě. Takové místo je ovšem těžké najít. Situaci naštěstí zjednodušuje fakt, že v Railetánii mezi každými dvěma zastávkami vede právě jedna cesta po kolejích.

Váš program dostane na vstupu počet zastávek N. Zastávky si budeme číslovat od jedné do N. Dále dostane seznam tratí mezi zastávkami. Každá trať je charakterizována čísly dvou zastávek, které propojuje. Předpokládáme, že trati se mezi zastávkami nekříží. Jako vzdálenost dvou zastávek pak vezmeme počet tratí, po kterých je nutné projet, abychom se dostali z jedné zastávky do druhé. Vaším úkolem je nalézt takovou zastávku, že k ní nejvzdálenější zastávka

je co nejblíže (tedy má minimální maximum ze vzdáleností z ní do ostatních zastávek). Pokud je takových zastávek více, stačí vrátit libovolnou z nich.

13-1-3 Stabilní úsek — 11 bodů

Strýček Skrblík se jednoho dne rozhodl, že budova plná mincí je již přeci jen trochu staromódní způsob uchovávání majetku, a že své peníze tedy investuje do akcií. Nechce pochopitelně o žádné peníze přijít, a tak potřebuje najít akcie, které jsou dostatečně důvěryhodné. Nakonec se rozhodl, že důvěryhodnost akcií bude posuzovat podle stability jejich ceny. Čím déle má (nebo měla) nějaká akcie stabilní cenu, tím je důvěryhodnější. Nyní, když už pan Skrblík oddřel všechno myšlení, obrátil se na vás s takovým triviálním problémem. Pro každou akcii spočítat, jak nejdéle byla stabilní.

Na vstupu váš program dostane dvě čísla D a N a pak posloupnost celých čísel $a_1, a_2 \ldots a_N$. Jeho úkolem je nalézt nejdelší úsek v posloupnosti takový, že žádné dva prvky v něm se neliší o více jak D (formálně tedy hledáme taková i, j, $1 \leq i \leq j \leq N$, že pro každé $k, l \in \{i \ldots j\}$ platí $-D \leq a_k - a_l \leq D$ a $j - i$ je maximální možné).

13-1-4 Stoky — 10 bodů

V jednom městě, které raději nebudeme jmenovat, měli složitý systém odvádění odpadních vod. Po letech přestaveb nakonec vedlo potrubí z každého sběrného místa do každého. Za ta léta ovšem dělníci také zapomněli, ve kterém sběrném místě je vlastně vývod z celého kanalizačního systému do blízké čističky odpadních vod, a nyní by to potřebovali zjistit. Vaším úkolem je napsat program, který jim ono místo pomůže nalézt.

Program na vstupu dostane počet sběrných míst N. Může pak klást dělníkům dotazy, zda teče voda mezi dvěma sběrnými místy daným směrem. Výtok ze systému je v takovém sběrném místě, kde ze všech ostatních míst teče voda do něj. Protože počítač správy kanalizací má dosti omezenou paměť a počet sběrných míst je velký, není možné si pamatovat nějakou informaci pro každé sběrné místo (odborně řečeno: vaše programy by měly mít lepší než lineární paměťovou složitost).

Příklad:

```
N = 5
Dotazy:
2->5: Ne
1->5: Ne
2->4: Ano
1->3: Ano
2->3: Ano
```

```
4->3: Ano
5->3: Ano
Odtok je v místě 3.
```

13-1-5 LISP — 10 bodů

V tomto ročníku jsme se rozhodli ukázat vám trochu netradiční přístup k programování, kterému se říká programování funkcionální (mějte s námi chvilku strpení, za chvíli dospějeme i k tomu, proč má takový podivný název), a tak v každé letošní sérii bude jedna úloha věnována jazyku Lisp.

Lisp je jazykem mnoha dialektů – existuje totiž ohromné množství různých interpreterů a aplikačních programů, které dávají uživateli možnost si dodefinovat své vlastní příkazy právě v Lispu (například známý Unixový textový editor Emacs), a jak už to tak bývá, každý, kdo si programoval svůj interpretr, se rozhodl, že si do jazyka přidá to, co mu tam chybí a že upraví věci, o kterých si myslí, že by se daly udělat lépe. Naštěstí všechny tyto dialekty mají mnoho společného, a tak si tu nadefinujeme náš vlastní velice jednoduchý dialekt, který bude obsahovat více méně jenom ono společné „jádro jazyka“, a nazveme ho Ksp-Lisp. Interpreter tohoto jazyka pod DOS či Linux si můžete stáhnout ze stránkek KSP na Internetu (adresa viz úvod).

Veškerá data jsou v Ksp-Lispu uložena jako *objekty.* Každý objekt má svůj typ a svoji hodnotu (což může být i odkaz na nějaký jiný objekt, jak uvidíme za chvíli). Typů existuje jen několik málo:

- *nil* – od tohoto typu existuje jen jediný objekt, a to objekt nazývaný `nil`.
- *integer* – pro každé celé číslo existuje právě jeden objekt typu integer, který jej reprezentuje. Tyto objekty je zvykem označovat příslušnými přirozenými čísly.
- *symbol* – symboly jsou vlastně pojmenované odkazy. Každému jménu (neprázdná posloupnost libovolných znaků, která neobsahuje znaky '`(`', '`)`', '`"`', '`.`' ani mezeru a neskládá se pouze z číslic) je přiřazen právě jeden objekt typu symbol, který obsahuje odkaz na jeden libovolný objekt (nebo speciální hodnotu *undefined,* pokud mu ještě nebyl žádný odkaz přiřazen). '`+`', '`+1`', '`define`' či '`nil`' jsou správně utvořená jména symbolů, symbol '`nil`' standardně ukazuje na objekt *nil.*
- *pár* – každý objekt typu pár obsahuje právě dvě položky, z nichž každá je odkazem na libovolný objekt. První položka se nazývá `a`, druhá `b` (v některých dialektech `car` a `cdr`).
- *primitivum* – to je zabudovaná funkce jazyka, kterou umí interpreter spočítat sám od sebe.

Symboly se často používají jako proměnné: místo toho, abyste si do proměnné (tedy vlastně do pojmenovaného místa v paměti) uložili nějakou hodnotu, v LISPu necháte nějaký symbol ukazovat na vaši hodnotu (jinými slovy nepojmenováváte místa v paměti, ale samotné hodnoty; díky tomu také nejsou přiřazeny typy proměnným, nýbrž objektům).

Páry a `nil` se používají k tvoření seznamů. Prázdný seznam se vždy vyjadřuje objektem `nil`, neprázdný seznam pak párem, jehož `a` ukazuje na první prvek seznamu a `b` na zbytek seznamu. Seznamy se obvykle zapisují jako posloupnosti objektů uzavřené v závorkách, přičemž jednotlivé objekty jsou odděleny libovolnou posloupností mezer, tabulátorů a konců řádků – tak například `(+ 1 (2))` je tříprvkový seznam, jehož prvním prvkem je symbol `+`, druhým číslo `1` a třetím seznam obsahující jako svůj jediný prvek číslo `2`. Tato struktura bude reprezentována párem, jehož `a` bude ukazovat na symbol `+` a `b` na druhý pár, jehož `a` bude ukazovat na integerový objekt `1`, `b` na třetí pár, který bude mít v `a` uložen odkaz na čtvrtý pár (`a` ukazuje na integer `2`, `b` na `nil`) a v `b` odkaz na `nil`. `()` je jen jiný název pro `nil`.

Mimo 'klasických' seznamů můžete vytvářet i složitější struktury – „nekonečné" cyklické seznamy (stačí, aby `b` posledního prvku ukazovalo na některý z prvků předchozích) nebo třeba binární stromy (listy reprezentujeme čísly nebo symboly, vnitřní vrcholy pak páry, jejichž `a` se bude odkazovat na levý podstrom a `b` na podstrom pravý).

Důležitým rozdílem oproti klasickým procedurálním jazykům je, že v LISPu nikdy přímo nealokujete paměť (a tím pádem ani neuvolňujete) – jakmile jakýkoliv objekt vznikne, interpreter pro něj sám nějaký kousek paměti přidělí a když už objekt nebude přístupný (to znamená, že se k němu nepůjde nijak dostat pomocí symbolů a odkazů mezi objekty), automaticky paměť uvolní k dalšímu použití (tomuto procesu se říká *garbage collection* [sbírání smetí]).

A jak se v LISPu programuje? Inu, je to funkcionální jazyk, takže se všechny programy skládají z funkcí. Každá funkce dostává své parametry (což jsou vlastně jen odkazy na objekty), vrací nějakou hodnotu (opět odkaz) a případně způsobí nějaký postranní efekt (side-effect), tedy založí, zruší či změní nějaké globálně viditelné objekty nebo třeba něco vypíše do výstupu. Každé volání funkce se zapíše jako seznam typu `(f 1 2 3)`, tedy prvním prvkem je funkce, která se má zavolat (obvykle symbol ukazující na nějaký seznam, jenž je definicí funkce) a všechny ostatní prvky se vyhodnotí jako parametry této funkce (mohou to být třeba opět volání funkcí popsaná seznamy).

Každý interpreter LISPu má dva módy – mód interaktivní (v tom přímo zadáváte funkce a on je vyhodnocuje a obratem vypisuje výsledky; velice šikovné pro ladění programů) a mód dávkový (tomu předáte nějaký soubor a on jej zpracuje, jako by byl řádek po řádku zadán interaktivně; tak se spouští již hotové programy). Nastartujete-li interpreter KSP-LISPu, objeví se vám prompt

oznamující, že se s vámi komunikuje interaktivně. Když pak zadáte `(+ 1 2)`, KSP-LISP to pochopí jako volání funkce `+` s parametry `1` a `2` a jelikož funkce `+`, jak již název napovídá, slouží ke sčítání celých čísel, vypíše vám obratem výsledek `3`. (Přesněji: nejprve se vyhodnotil symbol `+` a zjistilo se, že ukazuje na funkci, té se předaly odkazy na objekty `1` a `2`, funkce jako výsledek vrátila odkaz na objekt `3`, který byl vzápětí vypsán.) Seznam `(* (+ 1 2) (+ 3 4))` by způsobil zavolání již několika funkcí a výsledkem, jak asi čekáte, by bylo číslo `21`.

A takhle vypadají v LISPu všechny programy, i podmínky jsou totiž funkce (lépe řečeno speciální formy, jak uvidíme za chvilku, protože jejich argumenty se samočinně nevyhodnocují před tím, než jsou předány) – `(if` *p q r*`)` způsobí nejprve vyhodnocení podmínky *p* a poté, pokud je splněna (to znamená vyšlo-li něco jiného než `nil`), vyhodnotí se *q*, jinak *r* a vrátí se jeho hodnota. Místo cyklů se obvykle používají rekurzivně se volající funkce.

A teď již konečně prozradíme přesná pravidla pro vyhodnocování výrazů, tedy vlastně pro provádění programů. Vyhodnocení je proces, jehož vstupem je nějaký objekt x a výstupem nějaký jiný objekt y, který nazveme hodnotou objektu původního. Vyhodnocování probíhá takto:

- Je-li x `nil` nebo integer, je výsledkem tentýž objekt.
- Je-li x symbol a je-li mu přiřazen odkaz na nějaký objekt, je výsledkem tento objekt. Není-li mu přiřazeno nic, dojde k běhové chybě.
- Pokud je x pár, interpretujeme jej jako seznam kódující volání funkce. Nejprve (rekurzivně) zavoláme vyhodnocení na jeho první prvek, což nám má dát definici funkce, již voláme (taková definice je seznam, který začíná buďto symbolem `lambda` pro normální funkci nebo `special` pro speciální formu, jeho druhý prvek je seznam symbolů a, jimž se mají přiřazovat parametry a zbytek seznamu v je samotný vnitřek funkce). Nyní, nejedná-li se o speciální formu, rekurzivním voláním této vyhodnocovací procedury vyhodnotíme všechny zbylé prvky seznamu x, uložíme si na zásobník, kam se odkazují symboly zmíněné na seznamu a, a necháme je ukazovat na vyhodnocené argumenty, načež opět rekurzivním zavoláním vyhodnocujeme jednotlivé objekty seznamu v. Až toto vyhodnocování skončí, obnovíme uložené obsahy symbolů, v nichž jsme předávali parametry, a jako výsledek vrátíme výsledek posledního objektu ze seznamu v. Pokud by šlo o speciální formu, zachovali bychom se stejně, pouze bychom parametry předali nevyhodnocené. Pokud není první prvek x pár, jeho složka není požadovaný symbol a nebo nesouhlasí počet argumentů uvedených v a s tím, jaké byly skutečně předány v x, vyhlásíme běhovou chybu.
- Pakliže x je primitivum, chováme se k němu stejně jako v předchozím případě, pouze místo rekurzivního vyhodnocování podle definice

použijeme speciální znalosti o tomto primitivu, které nám umožní jej vyhodnotit „mimo lispovský svět“.

Vyhodnocování si ukážeme na jednoduchém příkladu: `(max 0 (+ 1 2))` zpracujeme tak, že nejprve zjistíme, že `max` je symbol, který ukazuje na definici funkce max, poté vyhodnotíme rekurzivně její argumenty: `0` se vyhodnotí na sebe samu, druhý parametr pak jako volání primitiva pojmenovaného symbolem `+` s parametry `1` a `2`. Nakonec zavoláme funkci `max`, předáme jí výsledky (objekty `0` a `3`) a výsledek posledního objektu v její definici vrátíme jako výsledek našeho vyhodnocování.

Mimo to ještě existují funkce s proměnným počtem argumentů – v jejich definici je místo úplného seznamu parametrů jen seznam končící symbolem `&rest` následovaným ještě jedním symbolem, kterému se přiřadí všechny zbývající argumenty jako seznam.

Následuje úplný seznam KSP-LISPových primitiv a jejich parametrů. (Uvědomte si ovšem, že primitiva samotná jsou objekty, takže i když to není dobrým zvykem, si je můžete přejmenovat, jak chcete. Pod zde uvedenými jmény jsou dostupná při startu interpreteru.) `Latinkou` budeme označovat jména primitiv a argumenty speciálních forem, které se samočinně nevyhodnocují, *kurzívou* pak argumenty funkcí; ‘. . . ’ značí, že funkce může mít argumentů libovolně mnoho.

`(+` *x* . . . `)` vrátí součet zadaných přirozených čísel. Analogicky `-` (s jedním parametrem se chová jako unární minus, jinak od prvního čísla odečte druhé, od výsledku třetí atd.), `*` a `/`.

`(=` *x y*`)` porovná dvě čísla, vrátí `t` pokud jsou stejná, `nil` pokud nejsou. Analogicky `<`, `>`, `<=`, `>=` a `<>`.

`(eq` *x y*`)` vrátí symbol `t`, pokud jak *x*, tak *y* ukazují na tentýž objekt, jinak `nil`. Pozor, to, že dva objekty stejně vypadají (například jsou-li to dva seznamy se stejným obsahem), ještě nemusí znamenat, že jsou opravdu stejné. Každému číslu odpovídá vždy právě jeden objekt, takže se pro čísla `eq` chová přesně jako `=`.

`(not` *x*`)` vrátí `t` pokud *x* je `nil`, jinak `nil`.

`(block` *x* . . . `)` vrátí poslední ze svých argumentů.

`(cons` *x y*`)` vrátí odkaz na nově vyrobený pár, jehož první složkou bude *x* a druhou *y*.

`(list` *x* . . . `)` vrátí odkaz na nově vyrobený seznam obsahující zadané prvky.

`(geta` *x*`)` vrátí složku `a` páru *x*, analogicky `getb`.

`(seta` *x y*`)` nastaví složku `a` páru *x* na *y*, analogicky `setb`.

`(get` *s*`)` vrátí, na co ukazuje symbol *s*. Pokud na nic neukazoval, dojde k běhové chybě.

`(defined?` *s*`)` vrátí `t`, pokud symbol *s* na něco ukazuje, jinak `nil`.

`(set` *s x*`)` nastaví symbol *s*, aby ukazoval na objekt *x*.

`(number?` x`)` vrátí `t`, pokud x je číslo, jinak `nil`. Obdobné funkce existují i pro ostatní typy: `pair?`, `symbol?`, `nil?` a `primitive?`.

`(eval` x`)` seznam x vyhodnotí podle popsaných vyhodnocovacích pravidel. Umožňuje za běhu programu vytvořit funkci a pak ji provést.

`(if` c `t f)` je speciální forma sloužící k větvení výpočtu. Nejprve vyhodnotí c; pokud vyjde `nil`, vyhodnotí `f` a vrátí jeho výsledek, jinak vyhodnotí `t` a vrátí jeho výsledek.

`(and x...)` je speciální forma sloužící jako booleovské and. Vyhodnocuje se tak, že se nejprve vyhodnotí první argument; je-li `nil`, skončí vyhodnocování celého `and`u s výsledkem `nil`, jinak se pokračuje dalším argumentem ve stejném duchu. Pokud ani poslední argument není `nil`, vrátí se jeho hodnota jako výsledek.

`(or x...)` je obdobná speciální forma, tentokráte fungující jako logický součet. Opět se postupně vyhodnocují argumenty a vrací se první výsledek, který není `nil`; pokud žádný takový není, vrátí se `nil`.

`(quote x)` je speciální forma, která vrátí svůj první argument. Často se používá k zamezení vyhodnocení nějakého argumentu (uvědomte si, že argumenty speciálních forem se nevyhodnocují, takže uvedete-li před nějaký argument funkce volání `quote`, jediným důsledkem je, že argument „propadne“ skrz `quote` nezměněný a nevyhodnocený). Například výsledkem `(quote (1 2))` je seznam `(1 2)`. Mezi `list` a `quote` je jeden podstatný rozdíl: `list` vám dá pokaždé nový seznam, kdežto `quote` vrací odkaz na stále tentýž. Výrazy typu `(quote (...))` lze rovněž zkracovat jako `'(...)`.

`(define (f a...) y...)` je zkratka, která ušetří spoustu práce při jinak zdlouhavém definování funkcí. Nadefinuje funkci `f` s parametry `a...` a tělem `y`, jinými slovy sestrojí seznam `(lambda (a...) y...)` a přiřadí symbolu `f` odkaz na tento seznam. Pokud použijete `define` se symbolem místo seznamu jako druhým parametrem, výsledkem bude pouze nastavení tohoto symbolu, aby ukazoval na objekt `y` (jinými slovy `define` pak bude fungovat stejně jako `set`, pouze bude své parametry automaticky quotovat).

`(let ((l1 v1) (l2 v2)...) x...)` je speciální forma, která nejprve uloží obsah symbolů `l1`, `l2` atd., načež začne vyhodnocovat výrazy `v1`, `v2` atd. a jejich výsledky přiřazovat těmto symbolům. Poté vyhodnotí posloupnost výrazů `x...` a nakonec obnoví původní význam symbolů a vrátí jako výsledek hodnotu posledního z uvedených výrazů. Tato lokální definice symbolů se často používá podobně jako lokální proměnné v procedurálních jazycích, pouze je nutné si dát pozor na to, že pomocí `let` symbolům přiřazené objekty jsou viditelné i ve funkcích zevnitř `let` zavolaných.

A nyní si na jednoduchém příkladu ukážeme, jak v KSP-LISPu něco jedno-

duchého naprogramovat – bude to funkce `length`, jejíž hodnotou bude délka zadaného seznamu:

```
(define (length x)
  (if x
    (+ 1 (length (getb x)))
    0)
)
```

Délku počítáme rekurzivně: nejprve otestujeme, zda je seznam neprázdný (podmínka u `if` je nesplněna jen tehdy, je-li `x` objekt `nil`); pokud ano, vrátíme o jedničku zvětšenou délku jeho zbytku (bez prvního prvku), jinak vrátíme nulu jakožto délku prázdného seznamu.

Naším druhým příkladem bude funkce `reverse`, která pro zadaný seznam vrátí jiný seznam, jenž bude obsahovat tytéž prvky v opačném pořadí (bylo by také možné použitím `seta` a `setb` změnit pořadí prvků v seznamu původním, ale to my nechceme):

```
(define (rev2 x y)
  (if x
    (rev2 (getb x)
          (cons (geta x) y))
    y)
)
(define (reverse x)
  (rev2 x nil)
)
```

V řešení jsme si nadefinovali obecnější funkci `rev2`, která vrátí seznam, jenž vznikne připojením seznamu `y` za obrácení seznamu `x`. Taková funkce se dá snadno naprogramovat rekurzivně a `reverse` je vlastně `rev2` s `y=nil`.

Soutěžní úlohy: Naprogramujte v KSP-LISPu:

- funkci `max`, která jako své parametry dostane n celých čísel a vrátí nejmenší z nich.
- funkci `equal`, která porovná dva objekty (čísla, seznamy, symboly atd.) na ekvivalenci podle struktury (jinými slovy na číslech a symbolech se bude chovat stejně jako `eq`, ale seznamy a podobné konstrukce z párů bude považovat za stejné i tehdy, pokud jsou vystavěny z různých objektů, ale obsahují totéž). Vrátí `t` při shodě, jinak `nil`. *Příklady:* `(equal '(1 2) '(3)) = nil`, `(equal 1 1) = t`, `(equal '(1 (2 3)) '(1 (2 3))) = t`.

13-2-1 Sirky — 10 bodů

Pat a Mat hrají spolu následující hru (kupodivu nemá se šachy nic společného): Na stole leží hromádka s N sirkami. Hráči se střídají v tazích. V každém tahu může hráč odebrat z hromádky libovolný počet sirek ve tvaru $2^i \cdot 3^j \cdot 5^k$, kde i, j a k jsou nezáporná celá čísla. Hráč, který odebere poslední sirku, vyhrává.

Mat pořád prohrával, a tak vás požádal, abyste mu napsali program, který by mu poradil, jak na Pata vyzrát. Váš program tedy dostane na vstupu počet sirek na hromádce. Po jeho načtení vypíše, jak má Mat táhnout a zeptá se, jak táhl Pat. Pak opět poradí tah Matovi...

Příklad:

```
Na hromádce je 8 sirek.
Rada Matovi: 1
Pat: 4
Rada Matovi: 3
Mat vyhral!
```

13-2-2 Přetlačovaná — 12 bodů

Při studiu starých knih narazil pan Filogam na hru Přetlačovaná. Ta se hraje na herním plánu ve tvaru čtvercové sítě. Hráči střídavě kladou své kameny (první hráč červené, druhý zelené) na dosud neobsazená křížení. Situace na hracím plánu se nazývá *vyrovnaná*, pokud v každém řádku a každém sloupci má jeden hráč nejvýše o jeden kámen více než hráč druhý. Pan Filogam nalezl v knize zakreslenou jednu pozici hry, o které je v knize napsáno, že je vyrovnaná. Pan Filogam tomu ovšem nevěří, a protože za dlouhá léta už barvy vybledly, není možné jednoduše přepočítat počty kamenů v jednotlivých řádcích a sloupcích. Obrátil se proto na vás, abyste mu pomohli.

Vaším úkolem je napsat program, který na vstupu dostane počet kamenů na herním plánu a jejich souřadnice a vypíše, zda pozice skutečně mohla být vyrovnaná – tzn. zda existuje obarvení kamenů takové, že v každém řádku a každém sloupci se bude počet červených a zelených kamenů lišit nejvýše o jeden. Pokud takové obarvení kamenů existuje, má ho váš program vypsat.

Příklad: Na herním plánu jsou 4 kameny na pozicích $(0,0)$, $(0,1)$, $(1,1)$ a $(2,1)$. Řešením je třeba obarvit kameny 1 a 3 zelenou a kameny 2 a 4 červenou.

13-2-3 Kamiony — 10 bodů

Firma Doleva & Doprava se zabývá kamionovou přepravou na větší vzdálenosti. Délku trasy kamionu budeme značit D. Kamion ujede na nádrž plnou nafty K kilometrů. Na trase je N čerpacích stanic. Pro každou stanici i je dána její vzdálenost od počátku trasy a_i a cena c_i nafty na jeden kilometr v této

stanici (kamion může samozřejmě načerpat pouze část nádrže). Vaším úkolem je navrhnout zastávky u čerpacích stanic a množství načerpané nafty tak, aby cesta vyšla firmu co nejlevněji. Předpokládáme, že na počátku kamion vyjíždí s plnou nádrží.

Příklad: Trasa má délku 800 km. Kamion ujede na plnou nádrž 300 km. Na trase jsou 4 čerpadla ve vzdálenostech 200, 300, 500 a 600 km od počátku trasy. Ceny nafty na čerpadlech jsou po řadě 10, 40, 20 a 10.

Kamion pojede k čerpadlu 1, kde načepuje na 200 km. Pak k čerpadlu 3, kde načepuje na 100 km a nakonec k čerpadlu 4, kde načepuje na 200 km.

13-2-4 Mraveniště — 10 bodů

V jednom mravenčím království u Jáchymova se dostala k moci osvícená mravenčí královna. Tato královna se rozhodla poněkud vylepšit dopravní situaci ve svém mraveništi.

V mraveništi jsou jednotlivé komůrky (očíslujeme si je od 1 do N), které jsou propojené chodbičkami. Každá chodbička je jednoznačně určena dvojicí komůrek, mezi kterými vede (předpokládáme, že chodbičky se protínají pouze v komůrkách). Královnu by pro začátek zajímalo, jak dlouhá je nejkratší cesta mezi dvěma komůrkami. Má to ovšem jeden háček. Mravenčí počítač má velmi omezenou paměť a mraveniště je veliké. Nemůžete si tedy pamatovat všechny komůrky, natož pak všechny chodbičky (tzn. paměťová složitost vašeho algoritmu by měla být lepší než lineární vzhledem k počtu komůrek). Abyste vůbec mohli zjistit, jak vlastně mraveniště vypadá, můžete se obsluhujícího mravence ptát, zda mezi danými dvěma komůrkami vede chodbička či nikoliv.

Příklad: Počet komůrek je 4, zajímá nás délka nejkratší cesty mezi komůrkami 1 a 3. Dotazy:

```
1 3: Ne
1 2: Ano
2 3: Ano
```

Nejkratší cesta mezi komůrkami 1 a 3 má délku 2.

13-2-5 LISP — 10 bodů

Vaším úkolem v druhém pokračování našeho seriálu bude napsat funkci `cut`, která dostane na vstupu vyhledávací strom a dvojici čísel určujících interval. Vaše funkce by měla vrátit vyhledávací strom, ze kterého budou vypuštěna všechna čísla mimo zadaný interval.

Binární strom je datová struktura složená z uzlů. V každém uzlu jsou uloženy odkazy na dva uzly – levého a pravého syna (oba odkazy či jeden z nich

mohou být nastavené na *nil*, pokud příslušný syn neexistuje). Binární strom pak můžeme nadefinovat následovně:

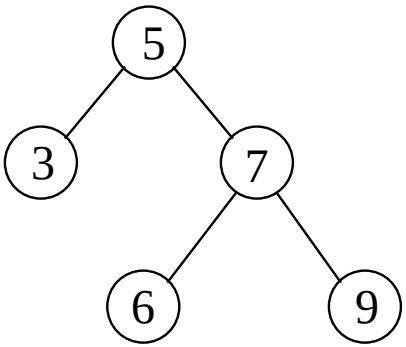
- Uzel a *nil* (tedy prázdná množina) jsou binární stromy.
- Jsou-li S a T binární stromy, pak struktura, která vznikne připojením S jako levého a T jako pravého syna pod nějaký nový uzel u, je binární strom.
- Všechny binární stromy lze získat konečným počtem aplikací předchozích dvou pravidel.

Vyhledávací stromy mají navíc v každém uzlu číslo. Pro toto číslo platí, že čísla ve všech uzlech v levém podstromu jsou menší a čísla ve všech uzlech v pravém podstromu jsou větší.

V LISPu budeme uzel vyhledávacího stromu reprezentovat ve tvaru

(*číslo* . (*levý syn* . *pravý syn*)).

Příklad: `(cut '(5 . ((3 . (() . ())) . (7 . ((6 . (() . ())) . (9 . (() . ()))))))) 4 6)` má vrátit `(5 . (() . (6 . (() . ()))))`. Strom vypadá takto:



13-3-1 Hip hop — 10 bodů

Malý Vašek dostal k Vánocům hru Hip hop. Na hracím plánu této hry je N významných bodů (budeme jim říkat „stanoviště“). Mezi těmito stanovišti vedou trasy rozdělené na políčka (z jednoho stanoviště mohou vést trasy do libovolně mnoha dalších stanovišť). Trasy se nikde mimo stanoviště nekříží.

Jednou z meziher hry Hip hop jsou takzvané „závody“. Ty probíhají tak, že se náhodně vybere počáteční a cílové stanoviště a hráči se pak musí co nejrychleji dostat z počátečního do cílového stanoviště. Přesun probíhá tak, že si v každém kole hráč hodí kostkou a může se posunout o tolik políček kupředu, kolik hodil.

Vaším úkolem je pomoci Vaškovi a napsat program, který dostane na vstupu popis herního plánu (tzn. počet stanovišť N, počet tras M a poté popis

jednotlivých tras, přičemž každá trasa je popsána čísly dvou stanovišť, která propojuje, a počtem políček na ní) a počáteční a cílové stanoviště. Na výstup pak vypíše stanoviště na nejkratší cestě (délku cesty měříme počtem políček na ní) z počátečního do cílového stanoviště. Při řešení úlohy by vám ještě mohlo pomoci, že žádná trasa neobsahuje více než 12 políček.

Příklad: Mějme plán s pěti stanovišti a následujícími šesti trasami: 1–2 (1), 1–3 (5), 2–4 (4), 1–4 (7), 3–4 (2), 4–5 (3) (čísla v závorkách udávají počet políček na trase). Nejkratší cesta ze stanoviště 1 na stanoviště 5 vede $1 \to 2 \to 4 \to 5$.

13-3-2 Fairies rights — 11 bodů

Společnost na ochranu tiskařských šotků se rozhodla zasáhnout proti diskriminaci skřítků pracujících v tiskařském průmyslu. Uvádíme úryvek z jejich prohlášení: „. . . kdjaký pisálek si klofá dp svého psacího strojw a každý druhý klof jde veddle. Pak se ani neobtěžuje po sobě text přečíst a chybh opravit. Když je již vše vytištěno a na chyby se přejde, tak se neprávem svádí na tiskřské šotky . . . “

Byla ustavena speciální komise na posuzování tiskařských chyb (SKoPoTiCH), která měla za úkol posoudit vliv šotků na množství chyb v textu. Komise vždy získala jednak výtisk nějakého textu a jednak autorův originál a měla zjistit množství chyb způsobených tiskařskými šotky. Tato práce se ukázala být značně namáhavá a pomalá. Proto se komise rozhodla využít výpočetní techniky a vymyslela důmyslný algoritmus automatického počítání výskytů chyb.

Algoritmus se vlastně snaží jednoduchými úpravami převést text originálu na text výtisku. Pro každou úpravu je stanovena její „cena“ (nějaké kladné celé číslo) a algoritmus hledá takový převod, který je nejlevnější (tzn. součet cen jednotlivých úprav je minimální). Povolené úpravy jsou následující:

- Vypuštění znaku
- Vložení znaku
- Záměna znaku

Protože však nikdo nedokázal algoritmus implementovat, byli jste požádáni o pomoc.

Vaším úkolem je napsat program, který dostane na vstupu ceny oněch tří povolených úprav a dva řetězce (originál a výtisk) a na výstup vypíše cenu nejlevnějšího převodu originálu na výtisk a poté samotný převod (tedy posloupnost úprav).

Příklad: Ceny:

- Vypuštění: 2
- Vložení: 5
- Záměna znaku: 3

Pro řetězce `abcdefghi` a `abbcdfghj` je cena nejlevnějšího převodu 10 a převod vypadá následovně: vypustit znak na pozici 3, vložit `e` za pozici 4, změnit znak na pozici 9 na `i`.

13-3-3 Konvertor — 10 bodů

Pan Brouček se rozhodl si pro své výlety postavit vesmírnou loď (vždyť přece cestoval až na Měsíc a prý má plány vyrazit ještě dále). Už si sehnal rozličné součástky, jako třeba plasmový hypermotor, emitor mionů nebo prediktor chyb. Stále mu ovšem chybí Konvertor a nemůže ho nikde sehnat. A tak požádal vás, jestli byste mu nepomohli.

Konvertor je zařízení, které převádí číslo zapsané v jedné číselné soustavě na číslo zapsané v jiné soustavě. Není to ovšem tak jednoduché, protože základ soustavy může být i záporný.

Nadefinujme si čísla v soustavě o základu z, kde z je celé číslo a $2 \leq |z| \leq 16$. Každá cifra čísla je znak z intervalu $0, 1, \ldots, |z|-1$, kde předpokládáme, že číslu deset odpovídá znak `a`, číslu jedenáct znak `b` až číslu patnáct znak `f`. Hodnota čísla $c_k c_{k-1} \ldots c_0$, kde c_i pro $0 \leq i \leq k$ je i-tá cifra zprava, je pak $\sum_{i=0}^{k} c_i z^i$.

Pro bližší osvětlení uvedeme jako příklad soustavu o základu -2. Hodnoty jednotlivých řádů v této soustavě jsou $(-2)^0 = 1, (-2)^1 = -2, (-2)^2 = 4, \ldots$ a cifry jsou buď `0` nebo `1`. Tedy například číslo 9 má v minusdvojkové soustavě zápis `11001`, protože $9 = 1 \cdot 16 + 1 \cdot (-8) + 0 \cdot 4 + 0 \cdot (-2) + 1 \cdot 1$.

Vaším úkolem je napsat program, který dostane na vstupu základ vstupní soustavy, základ výstupní soustavy a korektní číslo ve vstupní soustavě. Na výstup pak má vypsat dotyčné číslo převedené do výstupní soustavy (můžete předpokládat, že zadané číslo bude nezáporné, takže půjde vždy vyjádřit ve výstupní soustavě).

Příklad: Pro vstupní soustavu o základu -3, výstupní soustavu o základu 16 a číslo `11112` by měl program vypsat `3e`.

13-3-4 Šťastná čísla — 10 bodů

Pan Uno se zabývá numerologií. Při svých výzkumech přišel na to, že o šťastnosti čísla nerozhoduje ani tak jeho konkrétní hodnota, jako spíše jeho dělitelé. Po letech výzkumů pak došel k závěru, že šťastná jsou ta čísla, která jsou dělitelná pouze třemi, sedmi, jedenácti nebo libovolnými mocninami těchto čísel. Poté, co pan Uno došel k takto závažným výsledkům, je na vás, abyste je pomohli uvést do praxe (přeci jen zákazníci pana Una chtějí slyšet konkrétní čísla a ne jen nějakou teorii). Vaším úkolem je tedy pro dané N vypsat prvních N šťastných čísel.

Příklad: Pro $N = 8$ by program měl vypsat 3, 7, 9, 11, 21, 27, 33, 49.

13-3-5 LISP — 10 bodů

Vaším dnešním úkolem bude implementovat funkce *Map* a *BFS*. Funkce *Map* dostane jako argumenty funkci beroucí jeden argument a seznam. Vrátit by měla seznam, jehož i-tý prvek je výsledek funkce zadané jako první argument na i-tý prvek původního seznamu.

Funkce *BFS* dostane jako argument binární strom a vrátí seznam prvků vzestupně seřazený podle vzdálenosti od kořene (tzn. v pořadí, v jakém vrcholy prochází prohledávání do šířky). Uzel stromu bude mít strukturu `(hodnota . (levý syn . pravý syn))`, pokud levý nebo pravý syn neexistují, je na jejich místě hodnota *nil*.

Příklad: Volání `(Map '(lambda (x) (+ x 1)) (5 4 6 1))` by mělo vrátit seznam `(6 5 7 2)`, protože funkce daná jako první argument vrátí vždy dané číslo zvětšené o jedna.

Příklad: `(BFS '(1 . ((5 . (() . ())) . (3 . ((4 . (() . ())) . (6 . (() . ()))))))))` by mělo vrátit seznam `(1 5 3 4 6)`.

13-4-1 Fotografové — 12 bodů

Nudlová Lhota je vesnice se skutečně příhodným jménem. Domy v ní jsou totiž postaveny pouze podle silnice, která vsí prochází, a to ještě k tomu jenom z jedné strany. Protože Nudlová Lhota je skutečně raritou, rozhodli se ji na fotografiích zachytit hned dva umělci – Alfons Různý a Adalbert Rovný. Protože omítky na domech už jsou poněkud omšelé, rozhodl se starosta, že každý dům bude pro tuto významnou událost nově natřen. Problémem ovšem je, že fotograf Alfons Různý by chtěl, aby na každé jeho fotografii byly alespoň dva domy různé barvy. Adalbert Rovný by si naopak přál, aby alespoň dva domy na každé jeho fotografii měli stejnou barvu. A aby starostí nebylo málo, tak vesničtí obyvatelé by chtěli, aby bylo použito co nejvíce barev. Protože starostovi už z toho jde hlava kolem (a ještě k tomu kouká jako sůva z nudlí), rozhodl se vás požádat o pomoc.

Vaším úkolem je navrhnout algoritmus a napsat program, který na vstupu dostane počet domů ve vsi N. Pak také dostane záběry, které by rád vyfotil Alfons Různý a záběry, které by rád vyfotil Adalbert Rovný. Každý záběr je charakterizován číslem prvního a posledního domu na záběru – domy jsou očíslovány od 1 do N. Váš program by měl pro každý dům navrhnout barvu omítky tak, aby pro každý záběr byly splněny podmínky příslušného malíře, které jsou uvedeny výše. Navíc by mělo být použito co nejvíce barev.

Příklad: Ve Lhotě je 5 domů. Alfons by rád udělal snímky 1–3, 3–5 a Adalbert snímky 1–2, 1–4.

Domy je možno obarvit třeba po řadě barvami $1, 1, 2, 3, 4$ a více barev už použít nelze.

13-4-2 Okružní jízda — 10 bodů

Po zakázce pro firmu Doleva & Doprava vám byl předložen další problém z motoristického prostředí. Slavný cestovatel a závodník Mickey Louda se vsadil, že projede okružní trasu okolo své rodné země Autistánu. Protože Autistán nemá vybudovanou síť čerpacích stanic, může Mickey doplňovat benzín jen v určitých místech na trati, kde byly předtím uloženy kanystry. A protože benzín je velmi drahý, bylo na trati rozmístěno jen tolik benzínu, aby stačil právě na její projetí. Mickey si ale uvědomil, že se mu takto může snadno stát, že skončí někde uprostřed pustiny bez jediné kapky benzínu. Proto si najal vás, abyste mu poradili, kde trasu začít, aby se mu takováto nepříjemná nehoda nestala.

Vaším úkolem je navrhnout algoritmus a napsat program, který na vstupu dostane délku okruhu v kilometrech L, počet stanovišť s benzínem K, $p_1 \dots p_K$ vzdálenosti jednotlivých stanovišť od hlavního města, které též leží na trati, a $a_1 \dots a_K$ počet kilometrů, které lze ujet s benzínem uloženým na daném stanovišti (předpokládejte, že nádrž má neomezenou kapacitu). Vaším úkolem je říci, v jaké vzdálenosti od hlavního města má Mickey započít svou okružní jízdu, aby mu nikde na trase nedošel benzín.

Příklad: Trasa je dlouhá 15 kilometrů a jsou na ní 4 stanoviště ve vzdálenostech 2, 7, 9, 13 kilometrů od hlavního města. Na stanovištích je uložen benzín na 4, 5, 3 a 3 kilometry.

Mickey může začít svůj objezd třeba na sedmém kilometru. Množství benzínu v nádrži na jednotlivých stanovištích pak bude: 0 (5), 3 (6), 2 (5), 1 (5), 0. Čísla v závorkách udávají množství benzínu po dočerpání.

13-4-3 Fazolky — 9 bodů

„Fazolky“ je hra pro jednoho hráče. Hraje se tak, že na počátku hry je N kalíšků a v těchto kalíšcích je náhodně rozmístěno N fazolí (v i-tém kalíšku je jich a_i). Jediným povoleným tahem ve hře je vzít z nějakého kalíšku fazoli a přemístit ji do kalíšku sousedního. Cílem hry je, aby v každém kalíšku byla právě jedna fazole. Vaším úkolem je navrhnout algoritmus a napsat program, který pro dané N a dané počty fazolí v jednotlivých kalíšcích určí minimální počet tahů potřebný k dosažení cílového stavu.

Příklad: Počet kalíšků je 5 a počty fazolí v kalíšcích jsou 3, 0, 1, 1, 0. K dosažení cílového stavu je třeba 5 tahů ($1 \to 2$, $1 \to 2$, $2 \to 3$, $4 \to 5$, $3 \to 4$).

13-4-4 Vodárna — 11 bodů

Poté, co jste pomohli v jednom nejmenovaném městě vyřešit problém s kanalizací, obrátili se na vás představitelé vodárenské společnosti Velká voda,

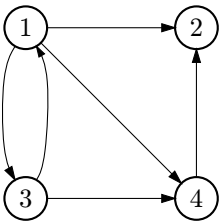
abyste jí pomohli s následujícím problémem: Rada města rozhodla, že s platností od 29. února 2002 bude muset téci z vodovodních kohoutků voda s malinovou příchutí. Proto je třeba urychleně vystavět továrnu na výrobu malinové šťávy a vyrobenou šťávu přidávat do rozváděné vody. A navíc je podnik třeba postavit na takovém místě, aby šťáva dotekla do všech míst v rozvodné síti. Vaším úkolem je rozhodnout, kde má být podnik postaven.

Váš program dostane na vstupu popis vodovodní sítě – počet uzlů v síti N a poté seznam propojení těchto uzlů. Každé propojení je charakterizováno čísly dvou uzlů, které propojuje, přičemž voda v síti teče od prvního ke druhému uvedenému uzlu. Vaším úkolem je nalézt takový uzel, ze kterého jsou po směru toku vody dosažitelné všechny ostatní uzly v síti (popřípadě říci, že takový uzel neexistuje).

Příklad: Počet uzlů v síti je 8. Propojení vedou takto: $1 \to 8$, $8 \to 5$, $5 \to 1$, $7 \to 5$, $8 \to 2$, $6 \to 2$, $2 \to 4$, $4 \to 3$, $3 \to 6$. Továrnu je možno postavit v uzlu 7.

13-4-5 LISP — 10 bodů

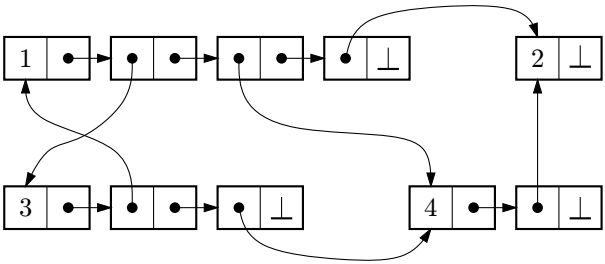
Náš Lispovský seriál pokračuje, tentokráte grafovou úlohou. Nejprve si ale ukážeme, jak orientované grafy (to znamená množiny vrcholů spolu s množinami šipek mezi nimi) v Lispu elegantně reprezentovat. Provedeme to takhle: ke každému vrcholu si vytvoříme seznam, jehož první prvek bude obsahovat číslo vrcholu (to proto, abychom mohli nějak vrcholy vypisovat a také aby tyto seznamy nemohly nikdy být prázdné) a ostatní prvky budou odpovídat vrcholům, do nichž vede z popisovaného vrcholu hrana, lépe řečeno tyto prvky budou rovnou obsahovat seznamy odpovídající příslušným vrcholům. (Uvědomte si, že v Lispu může být seznam klidně prvkem jiného seznamu a když na to přijde, tak i více takových.)
Ukážeme si to na příkladu. Graf



uložíme podle obrázku na následující stránce (obdélníčky odpovídají *cons*ům, jejich poloviny složkám `a` a `b`, šipky tomu, co v které složce je uloženo a „uzemnítka“ jsou *nil*y). Takovouto cyklickou strukturu bohužel nelze zapsat přímo, můžeme ji ale snadno vytvořit například takto:

```
(set 'v1 (list 1))
(set 'v2 (list 2))
```

```
(set 'v3 (list 3))
(set 'v4 (list 4))
(setb v1 (list v3 v4 v2))
(setb v3 (list v1 v4))
(setb v4 (list v2))
```



A nyní již slíbená úloha. Napište funkci `(path v w)`, která pro dané dva vrcholy `v` a `w` v nějakém orientovaném grafu nalezne nejkratší cestu z `v` do `w` a vrátí seznam čísel vrcholů na této cestě ležících, případně *nil,* pokud žádná taková cesta neexistuje. Pro náš ukázkový graf by tedy `(path v3 v2)` bylo `(3 4 2)`, zatímco `(path v2 v1)` vrátí *nil.*

13-5-1 Bláznivé hodiny — 11 bodů

Pan Johnson byl vynálezce. Ačkoliv někteří lidé o jeho genialitě pochybují, nelze mu upřít několik velmi zajímavých vynálezů. Patří mezi ně i jedny velmi speciální hodiny. Tyto hodiny vypadají přibližně jako svislá trubice s kovovými koulemi seřazenými za sebou. Koule jsou očíslované čísly od tří do N a na počátku jsou podle těchto čísel vzestupně seřazeny. Hodiny pracují tak, že v jednom tiku vezmou kouli ze spodu trubice a přesunou ji o tolik míst výše, kolik je její číslo (důmyslný mechanismus v průběhu přesunu přidržuje koule nad místem vložení). Pokud je koule přemístěna dále za poslední kouli, automaticky propadne na místo těsně za poslední koulí. Jistě chápete, že přes bezespornou zajímavost tyto hodiny vykazují i jistou míru nepraktičnosti. Vlastník těchto hodin by tedy po vás chtěl, abyste mu napsali program, který dostane počet koulí v trubici a spočte, po kolika ticích se poslední koule dostane na první místo.

Příklad: Pro $N = 8$ se poslední koule dostane na počátek po 7 krocích: 345678, 456378, 563748, 637485, 374856, 748356, 483567, 835647.

13-5-2 Holiči — 12 bodů

V jednom trpasličím městě svou živnost provozovali dva holiči. Protože již zítra bude svátek Blyštivého briliantu, rozhodlo se mnoho trpaslíků (a proslýchá se, že i trpaslic), že si nechají upravit svůj vous. Problém ale je, že trpaslíci jsou

dosti hašteřiví, a tak by se někteří u holiče mezi sebou pohádali. Tomu je třeba zabránit vhodným rozdělením trpaslíků mezi holiče. Zároveň je ale žádoucí, aby byli všichni zákazníci obslouženi co nejdříve.

Vaším úkolem je napsat program, který dostane na vstupu počet trpaslíků N a pak seznam dvojic trpaslíků, kteří se nesnáší. Na výstup by pak měl program vypsat pro každého trpaslíka, ke kterému holiči by měl jít, tak, aby žádná nesnášející se dvojice nešla ke stejnému holiči (pokud takové rozdělení trpaslíků neexistuje, stačí vypsat odpovídající zprávu). Navíc by měl být rozdíl mezi počty trpaslíků u jednotlivých holičů minimální.

Příklad 1: Oholit se chtějí 3 trpaslící. Nesnáší se 1–2, 2–3 a 1–3. V tomto případě rozdělení mezi holiče neexistuje.

Příklad 2: Oholit se chce 6 trpaslíků. Nesnáší se 1–2, 1–3, 2–4, 4–5. K prvnímu holiči mají jít trpaslíci 1, 4 a 6, k druhému 2, 3 a 5.

13-5-3 Optimální strom — 10 bodů

Jsou dány délky $d_1, d_2, \ldots, d_n$ setříděných posloupností v neklesajícím pořadí. To znamená $d_1 \leq d_2 \leq \ldots \leq d_n$. Na slití dvou posloupností délek x a y je potřeba $x + y$ porovnání. Navrhněte algoritmus a napište program, který určí minimální počet porovnání nutný ke slití všech posloupností. Nezapomeňte na důkaz správnosti vašeho algoritmu.

Příklad: Posloupnosti mají délky 2, 3, 5, 6 a 8. Slévání s nejmenším počtem porovnání slije nejdříve první a druhou posloupnost (5 porovnání), výsledek pak se třetí posloupností (10 porovnání). Potom slije čtvrtou a pátou posloupnost (14 porovnání) a nakonec výsledky předchozích dvou slévání (24 porovnání). Celkem bylo třeba 53 porovnání.

13-5-4 Generátor — 9 bodů

Poté, co jste pro pana Broučka sestrojili vytoužený Konvertor, jeho vesmírné lodi už mnoho nechybí. Podpalubí má vybavené celé a na palubě chybí již jen opravit pár prkýnek, která poškodili dělníci při instalaci tachyonového děla. Pan Brouček ovšem přišel na závažný problém. Při konstrukci své lodi úplně zapomněl na Generátor nepravděpodobnosti, který lodi zajišťuje ochranu proti meteoroidům a podobnému vesmírnému smetí. Generátor vypadá jako poměrně jednoduché zařízení – na zadané přirozené číslo odpoví nějakým jiným přirozeným číslem (na stejné přirozené číslo odpoví vždy stejně). Generátor má ovšem tu pozoruhodnou vlastnost, že posloupnost přirozených čísel $A_1, A_2, A_3, \ldots$, kde $A_1 = 1$ a A_i pro $i > 1$ je odpovědí generátoru na číslo A_{i-1}, se nikdy nezačne opakovat (tzn. každé číslo je v ní nejvýše jednou). Panu Broučkovi se nakonec podařilo jeden generátor sehnat v bazaru. Potřeboval by ale vyzkoušet, jestli notně opotřebovaný přístroj ještě funguje. A to už je úkol pro vás.

Napište program, který dostane na vstupu N a ověří, jestli se v prvních N prvcích generované posloupnosti vyskytne každé číslo nejvýše jednou. Váš program se může pouze dotazovat generátoru (v našem případě simulovaném uživatelem) na číslo následující po daném čísle. Dejte si v řešení pozor na to, že N může být značně velké a prvních N čísel posloupnosti se vám tedy zdaleka nemusí vejít to paměti.

13-5-5 LISP — 11 bodů

Naše Lispovská série pomalu končí a mnozí z vás si určitě již dávno řekli: „co to je za divný jazyk, ten Lisp, vždyť to nemá ani objekty!“ a v duchu si přáli, abychom je před takovým preglaciálním monstrem uchráníti ráčili. Ale omyl – naše pochoutková úloha na závěr totiž ukáže, že objekty Lisp ani mít nepotřebuje, neboť se v něm dají snadno naprogramovat.

Objekt je struktura v paměti, která obsahuje jednak proměnné (ty se obvykle nazývají *atributy* objektu), jednak funkce s tímto objektem pracující (těm se říkává *metody*). Objekt bude v naší Lispové implementaci vlastně speciální forma. Metoda m objektu o se pak bude snadno volat jako `(`*o m argumenty*`...)`. Hodnota atributu a objektu o se zjistí výrazem `(`*o a*`)` a do atributu se přiřadí nová hodnota h výrazem `(`*o a h*`)`. Na atributy se tedy můžeme dívat jako na metody s jedním nepovinným argumentem. Pokud argument není specifikován, metoda jen vrátí příslušnou hodnotu. Pokud je argument specifikován, metoda atribut na tuto hodnotu přenastaví a vrátí ji i jako výsledek.

Nové atributy a metody vytváříme voláním následujících *systémových metod,* které dostanou do vínku všechny vytvořené objekty. Každý objekt v naší implementaci má následující systémové metody:

- `def-attr 'a init` přidá do objektu nový atribut jménem `a` s počáteční hodnotou `init`, pokud objekt atribut `a` ještě nemá. V opačném případě pouze předefinuje hodnotu atributu `a` na `init`.
- `def-method 'm 'def` přidá do objektu novou metodu `m` s tělem `def` – to tedy musí být nějaký seznam začínající symbolem `lambda` nebo `special` a implementující metodu `m`. Pokud už objekt metodu `m` měl, `def-method` pouze předefinuje tuto metodu.
- `get-method 'm` vrátí definici metody `m`.
- `get-attrs` vrátí seznam všech metod a atributů objektu.

Nyní už jen zbývá popsat, jak vznikají nové objekty. Za tímto účelem je zde funkce `object` s jedním nepovinným parametrem. Volání `object` bez parametrů vytvoří nový objekt obsahující pouze systémové metody. Volání `object` s objektem o jako parametrem vytvoří kopii objektu o a tu vrátí, čímž snadno implementujeme objektovou dědičnost. Vaším úkolem je navrhnout funkci `object`, která bude generovat objekty se všemi popsanými vlastnostmi.

Vzorová řešení

13-1-1 Logické výrazy — Zdeněk Dvořák

V zadání této úlohy se vyskytlo několik nejasností:

- Nebylo zcela jasné, zda se zpracovávaná čísla vejdou do zabudovaného celočíselného typu. Jestliže ano, lze poměrně jednoduše dosáhnout lineární časové složitosti; jestliže je nutno s nimi zacházet jako s řetězci, je to podstatně složitější (a podařilo se to právě jednomu řešiteli). Vzhledem k tomuto jsem při hodnocení považoval za rovnocenné lineární řešení s omezenou velikostí čísel a kvadratické s neomezenou, v případě, že by toto kvadratické řešení přešlo na lineární, kdybychom operace s čísly prováděli v konstantním čase.
- Vyskytlo se asi 6 různých možností, jak provádět bitovou negaci čísla.

Pro autorské řešení jsem zvolil zřejmě nejobtížnější variantu – čísla neomezené velikosti a negace všech (∞) bitů (kterážto varianta negace se mi zdála nejlogičtější; její základní vadou je, že negace přirozeného čísla pak není přirozené číslo, nýbrž nějaká (nekonečná) posloupnost nul a jedniček).

Základní myšlenka algoritmu je tato: máme 2 zásobníky – jeden pro operátory, druhý pro čísla. Postupně načítáme jednotlivé tokeny (čísla nebo operátory). Načteme-li číslo, uložíme ho na zásobník čísel. Načteme-li operátor, podíváme se, zda předchozí operátor (uložený na vrcholu zásobníku operátorů) má vyšší prioritu; má-li, vyhodnotíme ho (odebereme ho ze zásobníku operátorů, ze zásobníku čísel odebereme operandy, provedeme příslušnou operaci a výsledek uložíme na zásobník čísel) a zkontrolujeme operátor pod ním. To opakujeme, dokud nenarazíme na operátor s nižší prioritou (nebo dno zásobníku); poté ho uložíme na zásobník operátorů. Při tomto postupu každý operátor vyhodnotíme po vyhodnocení předchozí a následující části výrazu ohraničené operátory nižší priority, což je přesně to, co chceme.
Zbývá ošetřit několik dalších záležitostí:

- Konec zadání považujeme za operátor s nejmenší prioritou, který způsobí vyhodnocení všech zbylých operátorů – výsledek pak již jenom odebereme ze zásobníku čísel.
- U prefixových operátorů neprovádíme vyhodnocování předchozích operátorů (ty by ještě nemusely mít svůj druhý operand), jen je uložíme na zásobník.
- Kdybychom měli nějaké postfixové operátory, neukládali bychom je po vyhodnocení předchozích operátorů na zásobník, nýbrž bychom je rovnou vyhodnocovali.

- Otevírací závorku považujeme za prefixový operátor velmi malé priority; k uzavírací závorce se chováme, jako by to byl postfixový operátor velmi malé priority (jen o málo větší než otevírací závorka), nicméně nevyhodnocujeme ji, ale odebereme místo toho ze zásobníku operátorů odpovídající otevírací závorku.

Tento algoritmus (až na vyhodnocování operací) má lineární časovou i paměťovou složitost, neboť na každý operátor sáhneme právě dvakrát – jednou při jeho vkládání do zásobníku, jednou při jeho vyhodnocování.

Nyní k samotné realizaci operací. Základním problémem je negace, kterou nechceme vyhodnocovat pokaždé znova (jinak bychom na výrazu typu `!!!...111...` nabrali kvadratickou časovou složitost). Proto si u čísla budeme pamatovat, je-li znegováno. Je nyní ovšem nutno dávat trochu pozor při implementaci zbytku operací. Jak po chvíli uvažování nahlédneme (detaily viz program), všechny ostatní operace lze provést v čase lineárním k délce kratšího z nich. Vzhledem k tomu, že kratší z operandů vždy po provedení operace zahodíme a výsledek ukládáme na místo, kde byl delší z nich, čas potřebný k provedení jedné operace je úměrný délce, o kterou se zkrátí součet délek všech operandů, tj. dohromady přes všechny operace nemůže být delší než $O(N)$. Tedy algoritmus má časovou i paměťovou složitost lineární.

Ještě poznámku k chybě, která se vyskytla v mnoha řešeních: většina operací s řetězci má časovou složitost úměrnou délce řetězců v ní užitých (to, že string v Pascalu vypadá stejně jako ostatní jednoduché typy, na tom nic nemění). Tedy např. cyklus tvaru

```
var a,b:string;
for i:=length(a)+1 to length(b) do
   a:='0'+a;
```

má časovou složitost $O((\text{rozdíl délek } \texttt{a} \text{ a } \texttt{b}) \times (\text{délka } \texttt{b}))$, tedy až kvadratickou!

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

#define MAXL 1000

#define PUSH (kam, co)  (* (kam)++= (co))
#define POP (odkud)  (*-- (odkud))
#define TOP (odkud)  ( (odkud)[-1])

#define swap (a, b) {typeof (a) p=a; a=b; b=p; }

typedef unsigned char uch;

#define UNUSED 0

#define ZACATEK  (uch) 1
#define KONEC  (uch) 0
#define AND  (uch) '&'
#define OR  (uch) '|'
#define XOR  (uch) '^'
```

```
#define NOT  (uch) '!'
#define LEFT  (uch) '('
#define RIGHT  (uch) ')'
#define NUMBER  (uch) 255

typedef struct {
    uch *od;
    int l,  ng;
} number;

number NIC;

void unnegate (number * co)
{
    if  (co->ng) {
        int i;

        co->ng = 0;
        for  (i = 0; i < co->l; i++)
            co->od[i] ^= 1;
    }
}

void negate (number * co)
{
    if  (!co->ng) {
        int i;

        co->ng = 1;
        for  (i = 0; i < co->l; i++)
            co->od[i] ^= 1;
    }
}

number enoth (number a,  number b)
{
    return a;
}

number enot (number a,  number b)
{
    a.ng ^= 1;
    return a;
}

number eand (number a,  number b)
{
    int ai,  bi;

    if  (a.l < b.l)
        swap (a,  b);
    if  (b.ng) {
        unnegate (&b);
        for  (ai = a.l - 1,  bi = b.l - 1; bi >= 0; ai--,  bi--)
            a.od[ai] = a.ng ^  ( (a.ng ^ a.od[ai]) & b.od[bi]);
    }
    else {
```

```
        a.od += a.l - b.l;
        a.l = b.l;
        unnegate (&a);
        for (ai = 0; ai < a.l; ai++)
            a.od[ai] &= b.od[ai];
    }
    return a;
}
number eor (number a, number b)
{
    int ai, bi;
    if (a.l < b.l)
        swap (a, b);
    if (b.ng) {
        a.od += a.l - b.l;
        a.l = b.l;
        negate (&a);
        unnegate (&b);
        for (ai = 0; ai < a.l; ai++)
            a.od[ai] = 1 ^ ( (1 ^ a.od[ai]) | b.od[ai]);
    }
    else {
        for (ai = a.l - 1, bi = b.l - 1; bi >= 0; ai--, bi--)
            a.od[ai] = a.ng ^ ( (a.ng ^ a.od[ai]) | b.od[bi]);
    }
    return a;
}
number exor (number a, number b)
{
    int ai, bi;
    if (a.l < b.l)
        swap (a, b);
    if (b.ng) {
        unnegate (&b);
        for (ai = a.l - 1, bi = b.l - 1; bi >= 0; ai--, bi--)
            a.od[ai] = 1 ^ a.od[ai] ^ b.od[bi];
        a.ng ^= 1;
    }
    else {
        for (ai = a.l - 1, bi = b.l - 1; bi >= 0; ai--, bi--)
            a.od[ai] = a.od[ai] ^ b.od[bi];
    }
    return a;
}
void vypis (number co)
{
    int l = 0;
    if (co.ng)
        printf ("...11");
```

```
    else {
        for (; l < co.l && !co.od[l]; l++);
        if (l == co.l)
            putchar ('0');
    }
    for (; l < co.l; l++)
        putchar ('0'+ (co.od[l] ^ co.ng));
    putchar ('\n');
}

typedef struct {
    uch typ;
    number value;
} token;

typedef enum { o_pref, o_post, o_cbra, o_inf, o_nul } typ;
typedef number (*akce) (number a, number b);

int prio[256];
typ top[256];
akce what[256];

#define operator (zn, tp, pr, ak)\
{\
prio[zn]=pr; \
top[zn]=tp; \
what[zn]=ak; \
}

void init_ops (void)
{
    operator (ZACATEK, o_pref, -3, enoth);
    operator (KONEC, o_post, -2, enoth);
    operator (AND, o_inf, 3, eand);
    operator (OR, o_inf, 2, eor);
    operator (XOR, o_inf, 1, exor);
    operator (NOT, o_pref, 4, enot);
    operator (LEFT, o_pref, -1, NULL);
    operator (RIGHT, o_cbra, 0, NULL);
    operator (NUMBER, o_nul, UNUSED, NULL);
}

uch *ostack, *otop;
number *nstack, *ntop;

uch buffer[MAXL];

void contr (int pr)
{
    while (prio[TOP (otop)] >= pr) {
        uch ao = POP (otop);
        number c1, c2;

        c2 = POP (ntop);
        if (top[ao] == o_inf) {
            c1 = POP (ntop);
            PUSH (ntop, what[ao] (c1, c2));
```

```
        }
        else
          PUSH (ntop, what[ao] (c2, NIC));
    }
}
void nexttoken (uch ** odkud, token * kam)
{
    switch (**odkud) {
        case 0:
        case '!':
        case '&':
        case '|':
        case '^':
        case ')':
        case '(':
          kam->typ = * (*odkud)++;
          break;
        case '0':
        case '1':
          kam->typ = NUMBER;
          kam->value.od = *odkud;
          kam->value.ng = 0;
          while (**odkud == '0' || **odkud == '1')
            * (*odkud)++ -= '0';
          kam->value.l = *odkud - kam->value.od;
          break;
    }
}
int main (void)
{
    uch *co = buffer;
    token akt;

    init_ops ();
    gets (buffer);
    ostack = otop = malloc (sizeof (uch) * (strlen (co) + 2));
    nstack = ntop = malloc (sizeof (number) * (strlen (co) + 2));
    PUSH (otop, ZACATEK);
    while (1) {
        nexttoken (&co, &akt);
        switch (top[akt.typ]) {
          case o_inf:
            contr (prio[akt.typ]);
            PUSH (otop, akt.typ);
            break;
          case o_post:
            contr (prio[akt.typ]);
            PUSH (ntop, what[akt.typ] (POP (ntop), NIC));
            break;
          case o_cbra:
            contr (prio[akt.typ]);
```

```
            POP (otop);
            break;
         case o_pref:
            PUSH (otop, akt.typ);
            break;
         case o_nul:
            PUSH (ntop, akt.value);
            break;
      }
      if (akt.typ == KONEC)
         break;
   }
   vypis (POP (ntop));
   return 0;
}
```

13-1-2 Centrum — Pavel Šanda

Sotvaže se po království roznesl problém, kam umístit ředitelství drah, rozpadla se tamější společnost na dvě nestejnoměrně velké části. Ta větší navrhovala předefinovat polohu centra a přesunout ředitelství spolu s managementem někam opodál, nejlépe na jiný kontinent, přičemž své návrhy doplňovala kupou vylepšovacích nápadů, jimiž se k mé převeliké lítosti nebudeme dále zabývat. Ta menší část se k problému postavila přeci jen realističtěji a počala se zabývat pouze vylepšovacími návrhy na naložení s ředitelstvím (ocet?). Nemohu smlčet, že se objevila i mikroskopická skupinka lidí, jež se začali problémem seriózněji zabývat.

Jedni (šťastnější) se snažili v grafu hledat nejdelší cestu, v jejímž středu nalezli hledané centrum (zdůvodni!) – byli-li to navíc staří kmeti, uchovávající paměť rodu, vzpomněli, že se kdes řešila obdobná úloha (12-1-2) a lineární čas jim zpravidla neutekl. V jiných se pro změnu přihlásil o slovo pravěký instinkt sběračů plodin a začali ze stromu v postupných vlnách odtrhávat listy až zbyly poslední dva (či jeden) vrcholy, které mohou být oba centrem grafu. Zbyla ještě nepočetná skupina šťastlivců, hledajících nejhlubší podstromy našeho stromu.

Následují méně šťastní – ti si např. nevšimli, že náš graf je strom (též zdroj mnoha chyb v odhadech časových složitostí) a bez skrupulí zavedli matici sousednosti, nedbajíce při tom nárůstu na minimálně kvadratickou složitost. O nic lépe nedopadli ani oddaní fandové Dijkstry, kteří v každém městě posílali poslíčky, by zjistili nejvzdálenější města, a tak grafem pobíhalo křížem a krážem stádo poslů, aby zjistili vzdálenost, kterou již před nimi desetkrát zjistil někdo jiný...

Ačkoliv nejsem vegetariánem, přidám se dnes na stranu požíračů listoví. Nejprve několik pozorování: Náš graf o n vrcholech je strom. Pro $n \geq 3$ centrum zřejmě není listem. Dále – odtrhnutím všech listů se nemění poloha centra (vzhledem k tomu, že max. cesty (rozuměj takové, jež nelze prodloužit)

musí končit v listech, všechny se zmenší o 1 a nerovnosti zůstanou zachovány). A konečně – pro $n \leq 2$ je centrum libovolný vrchol a protože pro $n \geq 3$ existuje vždy vrchol stromu, který není listem, vždy nám alespoň jeden vrchol zbyde.

Z toho plyne algoritmus: Odtrhnu všechny listy. Jedná se o strom, tedy vzniknou nové listy. Opakuji do okamžiku $n \leq 2$, kdy je nalezení centra triviální.

Implementace: Fronta, do které si postupně ukládám listí určené k utrhnutí. Při každém utržení vkládám do fronty nově vzniklé listy. (Rozmysli, proč nelze použít zásobník!)

Čas: Na každý vrchol sáhnu jednou $O(n)$ a u každého vrcholu hledám sousedy – stromeček má v *celém* grafu lineární počet sousedů, tedy $O(n)+O(n) = O(n)$. V každém kroku odstraním jeden lístek, takže je patrná i konečnost.

Paměťová složitost je $O(n)$ (je sice pravda, že v programu můžete vidět pole se sousedy velikosti $O(n^2)$, ale z něj se využívá pouze $O(n)$ prvků a nebyl by nejmenší problém toto pole upravit na n spojových seznamů sousedů, které by skutečně využívaly pouze paměti $O(n)$).

```
#include <stdio.h>
#define Max 10

int top[Max][Max];                      /* Sousedé vrcholu */
int deg[Max];                           /* Stupně vrcholu */
int newdeg[Max];                        /* Nové stupně vrcholu (po odebrání) */
int queue[Max];                         /* Odtržení = inkremenatace na nulu  */

int main (void) {
  int n, first=0, last=0, i, x, y;      /* Počet vrcholů, ukazatelé do fronty */
  scanf ("%d", &n);

  for (i=0; i<n-1; i++){                /* E=V-1, vrcholy číslovány od 0  */
    scanf ("%d %d", &x, &y);
    top[x][deg[x]++, newdeg[x]++]=y;    /* Nastav sousedy a stupně vrcholu */
    top[y][deg[y]++, newdeg[y]++]=x;
  }

  for (i=0; i<n; i++) if (deg[i]==1) queue[last++]=i;    /* Začneme všemi listy */
  /* Řádek výše přepsaný pro C-čkové labužníky :) */
  /* for (i=0; i[deg]==1 ? last++[queue]=i:1 , i¡n ; i++); */

  while (last>first) {
    x=queue[first++];                   /* Další list na utržení */
    for (i=0; i<deg[x]; i++)
      if (--newdeg[top[x][i]] == 1)     /* Neodtrženo? */
        queue[last++]=top[x][i];        /* Přidáme do fronty */
  }
  printf ("%d\n", queue[n-1]);
  return 0;
}
```

13-1-3 Stabilní úsek — Jakub Bystroň

Většina z vás by asi strýčka Skrblíka potěšila fungujícím programem, který by mu zřejmě pomohl k ještě většímu jmění. Někdy by se ale dlooouho načekal... Až na pár světlých výjimek poslali všichni více či méně elegantní řešení pracující s kvadratickou časovou a lineární paměťovou složitostí. Ta trocha ostatních úlohu vyřešila buď v čase $O(N \log N)$ a pamětí $O(N)$, nebo v čase $O(ND)$ a pamětí $O(D)$. Posledně jmenovaní dostali nejvíce bodů. Pochvalu si zaslouží Peter Krčah za nejelegantnější řešení v $O(N^2)$, které bylo asi na 20 řádků, podobně jako řešení Petera Belly v $O(ND)$. My zde ukážeme algoritmus s časovou složitostí $O(N \log D)$ a paměťovou $O(D)$. Ukážeme také, že si nepotřebujeme pamatovat celou posloupnost najednou v paměti. A jak tedy na to? Budeme se postupně ptát na členy posloupnosti a po načtení k-tého členu budeme schopni určit nejdelší stabilní úsek končící právě v k-tém členu. Takto snadno nalezneme i nejdelší stabilní úsek v celé posloupnosti. Všichni jste si správně uvědomili, že nějaká posloupnost celých čísel je stabilní, právě když je rozdíl maxima a minima ze členů této posloupnosti menší nebo roven D. Teď již k vlastnímu algoritmu. Předpokládejme, že známe nejdelší stabilní úsek končící k-tým členem dané posloupnosti. Začátek tohoto úseku si označme s. Nyní obdržíme další člen x a chceme určit nejdelší stabilní úsek v posloupnosti končící členem s indexem $k+1$, tj. aktualizovat s tak, aby úsek $a_s \dots a_{k+1}$ byl stabilní. Pro k si ještě pamatujeme dvě hodnoty a a b – minimum a maximum z hodnot prvků nejdelšího stabilního úseku končícího členem a_k. Zřejmě platí, že rozdíl čísel b a a je menší nebo roven D. Také si v nějaké prozatím blíže nespecifikované datové struktuře, označme ji T, pamatujme všechny různé hodnoty, které se v tomto stabilním úseku vyskytly, a u každé této hodnoty si ještě budeme pamatovat pozici jejího posledního výskytu. Uvědomme si, že počet prvků struktury T nepřekročí $D+1$. Prvky struktury T si také ještě pamatujme v nějakém seznamu (v programu je pro jednoduchost použit spojový seznam), jehož prvky jsou seřazeny podle indexu v posloupnosti. Každý prvek z T bude v dvojici s právě jedním prvkem ze seznamu (odborně se tomu říká bijekce). Proč seznam, ukážeme dále.

A jak teď toto množství informace aktualizujeme? Rozlišme tři případy:

1) $a \le x \le b$ – vše je v pořádku. Číslo x můžeme s klidným srdcem přidat do zatím nalezeného stabilního úseku, aniž bychom jeho stabilitu narušili. Hodnoty proměnných a, b, s se nemění.
2) $x < a$ – v tomto případě již musíme býti obezřetnější. Může se totiž stát, že přidáním prvku x můžeme stabilitu prozatím nalezeného úseku končícího na pozici k narušit (víme totiž, že $b - a$ může být až D). Nově tvořený stabilní úsek nemůže obsahovat prvky s hodnotami, které jsou větší než $x + D$. Všechny takové prvky proto z T

vyhodíme a podle toho samozřejmě upravíme i počátek nového úseku (proměnná s). Musíme také zrušit všechny prvky, které se v posloupnosti vyskytly dříve než nějaký právě zrušený prvek s největším indexem. Tyto prvky by sice svojí hodnotou obstát mohly, ale stabilní úsek musí být souvislý. S tímto úkolem nám pomůže právě již zmíněný seznam. Jednoduše sekvenčně smažeme všechny starší prvky, které se do této doby vyskytovaly v T. A kde bude počátek nového úseku? Inu, ten bude bezprostředně za zrušeným prvkem s největším indexem. Nakonec položíme $a := x$ a do b vložíme maximum z hodnot prvků posloupnosti tvořící nový stabilní úsek (tedy maximum z prvků struktury T). To je vše.

3) $x > a$ – tento bod je analogický jako bod 2) a snadno si ho rozmyslíte, stejně jako korektnost uvedeného postupu.

Nyní, nezávisle na hodnotě, x buď vložíme do T, pokud tam ještě není, nebo pouze aktualizujeme příslušnou hodnotu posledního výskutu x. Teď již jen zbývá popsat datovou strukturu T. Rozumné je T implementovat jako nějaký binární vyhledávací strom, který nám umožní vykonávat operace vložení a vyjmutí prvku stejně jako vyhledání minima resp. maxima v čase úměrném logaritmu počtu prvků struktury T neboli $O(\log D)$. Takovou vlastnost mají například AVL stromy, B-stromy a amortizovaně i splay stromy, na kterých se příkladně dá elegantně provést operace zrušení podstromu. Jelikož popisy těchto základních datových struktur včetně jejich implementací najdete v každé slušné učebnici programování a implementace jsou poměrně zdlouhavé (udržování základních vyvažovacích vlastností pomocí rotací apod.), jsou v programu uvedeny jen hlavičky těchto funkcí. Vše ostatní je uvedeno v plném rozsahu.

Jelikož vkládáme a rušíme každý prvek posloupnosti ve stromu T resp. v seznamu nejvýše jednou a obojí je v čase nejvýše $O(\log D)$ resp. $O(1)$, je časová složitost popsaného algoritmu opravdu $O(N \log D)$. Paměťová složitost je $O(D)$. Nakonec ještě poznamenejme, že se tato úloha dá řešit mnoha jinými algoritmy a jen s použitím statických datových struktur. Kvůli přehlednosti však byla v programu zvolena varianta s dynamickou alokací proměnných. Toť vše.

```
#include <stdio.h>
#include <stdlib.h>
struct node {                                   /* Vrchol BVS */
    int key;                                    /* Hodnota prvku posloupnosti */
    int pos;                                    /* Index v posloupnosti */
    struct item *f;                             /* Ukazatel do seznamu */
/* ... Ostatní tree–dependent věci */
};
struct item {                                   /* Prvek seznamu */
    struct node *n;                             /* Ukazatel na vrchol stromu */
```

```
    struct item *prev, *next;                    /* Předchozí, další ve spojáku */
};
struct node *root;                               /* Kořen stromu */
struct item *last;                               /* Poslední (nejnovější) prvek */

struct node *insert (struct node *T, int x, int pos, int *new)
{
    /* Pokud ve stromu T není prvek s key = x, pak jej vloží a vrací new = 1, jinak vrací
       new = 0 */
}

void delete (struct node *T, struct node *n)
{
    /* Ze stromu T smaže vrchol n */
}

struct node *find_bad (struct node *T, int x, int smer)
{
    /* Ve stromu T najde prvek s key > x pro smer = 1 resp. menší než x pro smer=0.
       Pokud takový prvek neexistuje, vrací NULL. */
}

int find_min (struct node *T)
{                                                /* ... */
}

int find_max (struct node *T)
{                                                /* ... */
}

int D;                                           /* Maximální diference */
int a, b;                                        /* Infimum a supremum */
int pocet;                                       /* Počet prvků aktuálního úseku */

void update (int smer)
{
    struct node *p;
    struct item *i, *j;

    for (; ; ) {
        p = find_bad (root, smer ? a + D : b - D, smer);   /* Najdeme nevyhovující
                                                              prvek... */
        if (!p)
            break;
        /* ...a rušíme všechny starší v aktuálním úseku */
        i = p->f;
        if (i->next)
            i->next->prev = NULL;
        while (i) {
            delete (root, i->n);                 /* Smažeme vrchol */
            j = i->prev;
            free (i);                            /* i prvek seznamu */
            i = j;
            pocet--;                             /* Ubyl nám jeden prvek */
        }
    }
```

```
    /* Nakonec ještě aktualizace druhé meze */
    if (smer)
        b = find_max (root);
    else
        a = find_min (root);
}
void add (int x, int pos)
{
    struct node *p;
    struct item *i;
    int new;

    p = insert (root, x, pos, &new);
    if (!new) {
        p->pos = pos;
        i = p->f;
        if (i->prev)
            i->prev->next = i->next;          /* vyjmutí */
        if (i->next)
            i->next->prev = i->prev;
        free (i);
    }
    i = (struct item *)malloc (sizeof (struct item));

    i->next = NULL;
    i->prev = last;
    i->n = p;
    if (last)
        last->next = i;
    last = i;
    p->f = i;
    pocet++;
}
int main (void)
{
    int N, x, pos;
    int kon, maxpocet;

    scanf ("%d %d", &D, &N);
    kon = 0;
    maxpocet = 1;

    pos = 0;
    scanf ("%d", &x);
    N--;
    a = b = x;
    add (x, 0);
    while (N--) {
        pos++;
        scanf ("%d", &x);
        if (x < a) {
            a = x;
            update (1);
```

```
        }
        else if (x > b) {
            b = x;
            update (0);
        }
        add (x, pos);
        if (pocet > maxpocet) {
            kon = pos;
            maxpocet = pocet;
        }
    }
    printf ("Krajni indexy nejdelsiho stabilniho useku jsou %d, %d.\n",
        kon - maxpocet + 2, kon + 1);
    return 0;
}
```

13-1-4 Stoky — Daniel Kráľ

Myšlenka algoritmu, kterým budeme zadanou úlohu řešit, je jednoduchá – budeme si udržovat množinu I těch sběrných míst, kde by mohl být vývod do čističky, a v každém kroku vyřadíme jedno sběrné místo z množiny I: Zvolíme dvě sběrná místa X a Y z množiny I a zeptáme se, zda teče voda z X do Y. Pokud teče, pak X můžeme vyřadit z množiny I, protože určitě není vývodem; pokud neteče, pak naopak můžeme z množiny I vyřadit Y, protože Y nemůže být v tomto případě vývodem. Aby náš algoritmus měl požadovanou konstantní paměťovou složitost, budeme postupovat tak, aby množina I byla vždy nějaký interval, tj. rovnala se všem celým číslům od A do B; v každém kroku se zeptáme, zda teče voda ze sběrného místa A do B a jedno z těchto míst z množiny I vyřadíme, tj. zvětšíme A o 1, pokud voda teče z A do B, nebo zmenšíme B o 1 v opačném případě. Po $N-1$ dotazech (N je počet sběrných míst), bude množina I jednoprvková a tedy jsme nalezli hledaný vývod z kanalizačního systému. Ve zdrojovém kódu algoritmu v Pascalu si všimněte, že používá pouze dvou proměnných, a to A a B.

Ze zadání úlohy plyne, že vývod ze systému existuje. Kdyby toto nebylo splněno, můžeme spustit náš algoritmus, jako kdybychom tuto skutečnost měli zaručenu, a pro sběrné místo, které náš algoritmus určí jako výtok z celého systému, tuto skutečnost jednoduše pomocí $N-1$ dotazů ověřit.

```
program vytok;
var A,B:word;
begin
  A:=1;
  writeln('Jaký je počet sběrných míst?');
  readln(B);
  while A<B do
    begin
      write(A,'->',B,': ');
      if readkey='A' then
```

```
      begin
        writeln('Ano'); inc(A)
      end
    else
      begin
        writeln('Ne'); dec(B)
      end;
  writeln('Odtok je v místě ',A,'.');
end.
```

13-1-5 LISP — Martin Mareš

a) Jelikož náš dialekt LISPu neobsahuje žádné konstrukce pro zápis cyklů, budeme si muset pomoci jinak, a to použitím rekurze. Myšlenka je snadná: minimum z jednoprvkového seznamu je jeho první prvek, minimum z libovolného delšího seznamu je buďto opět jeho první prvek nebo minimum ze zbytku seznamu, podle toho, co je menší (a na tom nic nezmění ani matoucí zadání vyžadující, aby se funkce jmenovala `max`). A také se snadno naprogramuje:

```
(define (max l)
  (if (and (getb l)
           (> (geta l) (max (getb l))))
    (max (getb l))
    (geta l)))
```

Tak s chutí do toho a půl je hotovo (ještě nám přeci zbývá část **b**) ... Jenže ouha, to, co jsme právě stvořili, je program z čeledi želvovitých, který už nad krátkým vstupem bude přemýšlet celé věky. Pokud mu totiž zadáte nalézt minimum z klesající posloupnosti délky n, bude minimum z posloupnosti délky $n-1$ počítat dvakrát, pro každou z nich pak dvakrát z délky $n-2$ atd., zkrátka a dobře na to celé spotřebuje čas minimálně 2^n.

Stačila by však maličkost, a to zapamatovat si někde, kolik vlastně minimum ze zbytku posloupnosti vyšlo, abychom ho v případě, že je menší než první prvek, mohli přímo vrátit jako výsledek a nemuseli ho počítat znovu. K tomu nám pomůže například primitivum `let`:

```
(define (max l)
  (if (getb l)
    (let ((m (max (getb l))))
      (if (< (geta l) m)
        (geta l)
        m))
    (geta l)))
```

A nebo bychom si mohli zvlášť nadefinovat minimum ze dvou čísel a poté minimum ze seznamu jako minimum z jeho prvního prvku a minima ze zbytku:

```
(define (max2 x y)
  (if (< x y) x y))
(define (max l)
  (if (getb l)
    (max2 (geta l) (max (getb l)))
    (geta l)))
```

Obě tato řešení mají lineární časovou a paměťovou složitost (každá úroveň rekurze přispěje konstantním časem a konstantní pamětí a zavolá následující úroveň nejvýše jednou). Ještě si ukážeme, jak snadno sestrojit funkci, která přímo počítá minimum ze všech svých parametrů, takže jí nemusíte předávat všechna čísla jako seznam, a navíc dodefinovává minimum z prázdného seznamu jako `nil`:

```
(define (pmax &rest l) (and l (max l)))
```

Pokud bychom chtěli výpočet maxima naprogramovat rovnou s `&rest`, musíme si dát pozor na to, jak naši funkci volat rekurzivně, abychom jí opravdu předali seznam parametrů a nikoliv celý seznam jako jeden parametr. K tomu nám pomůže funkce `(apply f '(...))` počítající `(f ...)`:

```
(define (apply f l) (eval (cons f l)))
```

(zkonstruuje si seznam odpovídající volání funkce `f` se správnými parametry a pak jej pomocí `eval` vyhodnotí). Pak bude naše funkce `max` vypadat takto:

```
(define (pmax a &rest l)
  (if l
    (max2 a (apply 'pmax l))
    a))
```

b) Porovnávání objektů podle struktury je o něco těžší, ale také se dá snadno popsat rekurzívně: dva objekty jsou `equal` tehdy, jsou-li `eq` nebo jsou-li oba z nich páry, jejichž první i druhé složky jsou `equal`. A to ověříme například následující funkcí:

```
(define (equal x y)
  (or (eq x y)
      (and (pair? x)
           (pair? y)
           (equal (geta x) (geta y))
           (equal (getb x) (getb y)))))
```

Časová i paměťová složitost jsou opět nejvýše lineární (vzhledem k velikosti porovnávaných objektů včetně všeho, na co se odkazují).

13-2-1 Sirky — Pavel Nejedlý

Stačilo si uvědomit, že za podmínek v zadání není možné odebrat z hromádky počet sirek dělitelný sedmi a naopak každé z čísel $1 \ldots 6$ odebrat lze. Protože nula je násobkem sedmi, bude nám stačit udržovat soupeře na násobcích sedmi – soupeř bude muset odebrat na $7 \cdot k + x, 1 \leq x \leq 6$ a my v následujícím tahu odebereme x, čímž na hromádce zbyde $7 \cdot k$ sirek. Pokud budeme mít tu smůlu, že počet sirek na začátku našeho tahu bude dělitelný sedmi (musel by být i na začátku hry), odebereme jednu sirku a budeme tajně doufat, že soupeř udělá chybu.

Program je teď již jednoduchý, časová složitost je lineární, paměťová konstantní.

```
#include <stdio.h>

int main (void) {
  int sirek, tah;

  printf ("Počet sirek:␣");
  scanf ("%d", &sirek);

  while (sirek) {                          /* Dokud jsou sirky  */
    if ( ! (tah=sirek%7) )                 /* Jsme v prohrávající pozici?  */
      tah=1;
    sirek-=tah;
    printf ("Beru %d, zbývá %d\n", tah, sirek);
    if (!sirek) {
      printf ("Prohrál jste, saláte\n");
      return 0;
    }
    printf ("tah :␣");
    scanf ("%d", &tah);
    sirek-=tah;
  }

  printf ("Příště jistě prohraješ\n");

  return 0;
}
```

13-2-2 Přetlačovaná — Pavel Machek

Na začátek jedno jednoduché pozorování: „To se nedá zkazit.“

Nejdříve zjistíme, které kameny leží ve stejných řádcích a sloupcích. To můžeme snadno udělat buď pomocí třídění či pomocí hashování, které je v průměrném případě rychlejší. V každém řádku a sloupci pak rozdělíme kameny do dvojic (libovolně). Kameny ve dvojicích si pak představíme jakoby propojené hranou. Hrany budou dvou druhů – vertikální (spojují kameny ve stejném sloupci) a horizontální (spojují kameny ve stejném řádku). Získáme tak graf, kde z každého z vrcholů (kamenů) vedou nejvýše dvě hrany (z každého vrcholu totiž vede nejvýše jedna horizontální a jedna vertikální hrana).

Kameny, které nejsou v žádné dvojici, můžeme obarvit libovolně (máme dovolený rozdíl jedna v počtu kamenů různých barev). Pokud se nám tedy podaří obarvit kameny tak, že kameny ve dvojici budou mít různé barvy, je toto obarvení zřejmě i řešením pro původní úlohu.

Barvíme následujícím způsobem: Vybereme libovolný kámen a obarvíme ho červeně. V jaké komponentě grafu může ležet? Může ležet na cestě nebo na cyklu sudé délky. Cykly liché délky se nevyskytují, protože se v grafu pravidelně střídají vertikální a horizontální hrany a žádné další komponenty existovat nemohou, protože z každého vrcholu vedou nejvýše dvě hrany. Jak cesty, tak cykly sudé délky je možné obarvit jednoduše hladově. Obarvení jednoho kamene jednoznačně určuje obarvení komponenty grafu, ve které leží, a nemá žádný vliv mimo tuto komponentu, takže když jsme s komponentou hotovi, můžeme vybrat libovolný jiný kámen, a začít od začátku.

Protože zjišťování, které kameny leží na stejném řádku a sloupci pomocí hashování, je lineární k počtu kamenů a samotné procházení grafu a barvení též, má výsledný algoritmus lineární časovou složitost k počtu kamenů. Paměťová složitost je též lineární k počtu kamenů.

Program je přímou implementací algoritmu. Snad jediná odlišnost spočívá v tom, že ve skutečnosti žádné dvojice ani graf nekonstruujeme. Prostě vždy vezmeme první neobarvený vrchol v horizontálním či vertikálním směru, na který narazíme.

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

struct kamen {
    struct kamen *next_horiz;
    struct kamen *next_vert;
    int i;
    int j;
    int cislo;
    int barva;
    int tisknuto;
};

struct kamen * hash_horiz[100], * hash_vert[100];
int cislo;                                          /* Počet kamenů */

void
tiskni (struct kamen *kam)
{
      printf ("Kamen cislo %d (na %d, %d) by mel byt %s.\n", kam->cislo,
              kam->i, kam->j, kam->barva==1 ?"cerveny":"zeleny");
}
/* Obarvíme daný vrchol a jdeme rekurzivně barvit souseda (podle proměnné faze buď
     vertikálního nebo horizontálního) */
void
oznac (struct kamen *kam, int faze, int barva)
```

```
{
    if (!kam)
        return;
    if (!faze) {
        int j;
        kam->barva = barva+1;
        tiskni (kam);
        j = kam->j;
        kam = hash_vert[j%100];
        while (kam) {
            if ( (j==kam->j) && !kam->barva)
                return oznac (kam, !faze, !barva);
            kam = kam->next_vert;
        }
        return;
    }
    if (faze) {
        int i;
        kam->barva = barva+1;
        tiskni (kam);
        i = kam->i;
        kam = hash_horiz[i%100];
        while (kam) {
            if ( (i==kam->i) && !kam->barva)
                return oznac (kam, !faze, !barva);
            kam = kam->next_horiz;
        }
        return;
    }
}
int main (void)
{
    int i;
    struct kamen *kam = malloc (sizeof (struct kamen));

    /* Načteme vstup */
    while (scanf ("%d %d\n", &kam->i, &kam->j)==2) {
        kam->cislo = ++cislo;
        kam->tisknuto = kam->barva = 0;
        kam->next_horiz = hash_horiz[kam->i % 100];
        hash_horiz[kam->i % 100] = kam;
        kam->next_vert = hash_vert[kam->j % 100];
        hash_vert[kam->j % 100] = kam;
        kam = malloc (sizeof (struct kamen));
    }

    /* Jdeme barvit */
    for (i=0; i<100; i++) {                    /* Projdeme celou hashovací tabulku */
        struct kamen *kamen = hash_horiz[i];

        while (kamen) {
            if (!kamen->barva) {               /* Kámen ještě nemá barvu? */
                /* Obarvíme komponentu grafu na jednu a na druhou stranu */
```

```
            oznac (kamen, 0, 0);
            oznac (kamen, 1, 0);
          }
          kamen = kamen->next_horiz;
        }
      }
    }
    return 0;
}
```

13-2-3 Kamióny — Aleš Přívětivý

Nechť N je počet pump, D délka trasy a K velikost nádrže auta. Posloupnost $a_1, a_2, \ldots, a_N$ určuje vzdálenost pump od startu a $c_1, c_2, \ldots, c_N$ jsou jim příslušné nezáporné ceny nafty. Pro zjednodušení si původní úlohu malinko pozměňme – auto nezačíná s plnou nádrží, ale na startu je pumpa s nulovou cenou nafty. Je zřejmé, že obě úlohy jsou si ekvivalentní.

Nyní už ale k řešení. Každého hned napadne, že by se úloha dala řešit za pomoci dynamického programování. To vede k algoritmu s časovou složitostí $O(NK^2)$ a paměťovou $O(K)$. Ukážeme si ale hladový algoritmus, který má mnohem lepší časovou složitost.

Je zřejmé, že k projetí celé trasy je třeba natankovat alespoň D litrů nafty. Na druhou stranu je ale zbytečné tankovat víc. Tedy optimální řešení tankuje dohromady přesně D litrů. Dále je očividné, že jedno optimální řešení je takové, při kterém u nejlevnější pumpy natankujeme co nejvíce nafty, ale jen tolik, kolik potřebujeme na dojetí do konce. U druhé nejlevnější zase co nejvíce nafty, ale zase jen tolik, kolik potřebujeme – protože něco už nám mohlo zbýt od nejlevnější pumpy, pokud je před, nebo pokud je za, tak bychom po příjezdu k ní museli naftu „vylít“). Podobně můžeme uvažovat pro třetí, čtvrtou až N-tou nejlevnější. Označme toto jako princip minimality.

Uvažme nyní následující algoritmus, podle kterého budeme čerpat benzín:

1) Pokud jsou v dojezdu pumpy s nižší cenou, nakupme jen tolik benzínu, abychom mohli dojet do nejbližší z nich.
2) Jinak doplňme celou nádrž a zastavme se v nejbližší další pumpě.
3) Pokud nejbližší další pumpa je dál, než je dojezd auta, úloha nemá řešení.

Tento algoritmus splňuje princip minimality, budeme-li výše uvedený postup používat postupně u všech pump v pořadí podle vzrůstající vzdálenosti od startu. Následuje zcela neformální náhled, proč tomu tak je: Pokud nám po příjezdu k i-té pumpě zbylo $t, t > 0$ litrů nafty v nádrži, musel nastat bod 2) u nějaké předchozí pumpy a je tedy logické, že jsme u té pumpy vzali více nafty (cena v ní je nižší). Podle bodů 1)–3) tedy v i-té pumpě nakoupíme co nejvíce nafty, ale pouze tolik, abychom neomezili nákupy u levnějších pump. Výše uvede-

ný hladový algoritmus tedy nalezne optimální řešení. (Nepovažujte v žádném případě předchozí řádky za důkaz – ten by byl poněkud komplikovanější.)

Při provádění algoritmu tedy procházíme pumpy podle vzrůstající vzdálenosti od startu a rozhodujeme se podle pravidel 1)–3). Body 2) a 3) umíme rozhodnout v konstantním čase, bod 1) vyžaduje projití několika následujících pump. To by mohlo trvat v nejhorším případě $O(N)$. Proto si pumpy, které jsou v dojezdu, vkládáme do haldy uspořádané podle cen nafty, a tedy vždy můžeme rychle ověřit bod 1). Vložení do haldy nám zabere čas nejvýše $O(\log N)$. Operace zjistění pumpy s minimální cenou čas $O(1)$. Před touto operací, ale musíme vždy odebrat pumpy s nižší cenou, kterými jsme již projeli. Odebrání jedné pumpy nám v nejhorším případě zabere čas $O(\log N)$. Celkově tedy algoritmus potřebuje čas $O(N \log N)$. Paměťové nároky jsou $O(N)$ – na implementaci haldy a uložení všech pump.

```
#include <stdio.h>
#define MAXPUMP 100

struct PUMPA {
            double c;
            int a;
          } pumpa[MAXPUMP];

int N, D, K;
int velh, halda[MAXPUMP];

void insert (int p)                          /* Vložení prvku do haldy */
{
  int i;

  halda[velh]=p; i=velh++;
  while (i && pumpa[halda[ (i+1)/2]].c > pumpa[halda[i]].c )
        { p=halda[i]; halda[i]=halda[ (i+1)/2]; halda[ (i+1)/2]=p; i= (i+1)/2; }
}

void extractmin (void)                       /* Vyjmutí prvku s minimální cenou */
{
  int i, j, p;

  if (velh==0) return;

  halda[0]=halda[--velh]; i=0;
  while ( i*2+1 < velh )
  {
    if (i*2+2 < velh && pumpa[halda[i*2+1]].c > pumpa[halda[i*2+2]].c )
      j=i*2+2; else j=i*2+1;
    if (pumpa[halda[i]].c <= pumpa[halda[j]].c) break;
    p=halda[i]; halda[i]=halda[j]; halda[j]=p; i=j;
  }

}

int cmp (const void *e1, const void *e2)     /* Pomocná proc. pro qsort */
{
   return ( (struct PUMPA *)e1)->a - ( (struct PUMPA *)e2)->a;
```

```
}
int main (void)
{
    int i, j, t, vloz, dotank;

    /* Načtení a inicializace údajů */
    scanf ("%d%d%d", &N, &D, &K);
    for (i=1; i<=N; i++) scanf ("%d%lf", &pumpa[i].a, &pumpa[i].c);
    pumpa[0].c=0; pumpa[0].a=0; velh=0; t=0; vloz=0;

    /* Pumpy si setřídíme podle rostoucí vzdálenosti od startu */
    qsort (pumpa, N+1, sizeof (struct PUMPA), cmp);

    for (i=0; i<=N; i++)
    {
        /* Spotřebujeme naftu za přejezd z i - 1-té na i-tou pumpu */
        if (i) t-=pumpa[i].a-pumpa[i-1].a;
        /* Vložíme do haldy pumpy v dojezdu */
        for (; vloz<=N && pumpa[vloz].a-pumpa[i].a<=K ; vloz++) insert (vloz);
        /* Odstraníme ty s minimální cenou, které jsme už přejeli */
        while ( velh>0 && pumpa[halda[0]].a <= pumpa[i].a ) extractmin ();

        if (!velh && i<N )                    /* Případ c) */
            { printf ("Velikost nadrze nedovoluje projet.\n"); return 0; }

        if (pumpa[halda[0]].c < pumpa[i].c)   /* Případ a) */
        {
            if (t>pumpa[halda[0]].a-pumpa[i].a) dotank=0;
                else dotank=pumpa[halda[0]].a-pumpa[i].a-t;
        } else                                /* Případ b) */
        {
            if (pumpa[i].a+K>D) dotank=D-pumpa[i].a-t; else dotank=K-t;
        }

        if (dotank>0)
            printf ("Na pumpe na %d km bereme %d nafty.\n", pumpa[i].a, dotank);
        t+=dotank;
    }

    return 0;
}
```

13-2-4 Mraveniště — Daniel Kráľ

Naším úkolem v této úloze je s malou pamětí nalézt nejkratší cestu mezi dvěma komůrkami v mraveništi. Vytvořme si funkci `cesta`, která bude ověřovat, zda mezi dvěma komůrkami vede cesta nejvýše zadané délky. Tato funkce obdrží 3 parametry (A,B a d) a vrátí `true`, pokud mezi A a B existuje cesta délky d. Jak bude tato funkce fungovat? Pokud je d rovné jedné, položí dotaz mravenci obsluhujícímu počítač. Pokud je d větší než jedna, tak zkusí, zda existuje komůrka C taková, že délka cesty z A do C je $\lfloor d/2 \rfloor$ a délka cesty z C do B je $\lceil d/2 \rceil$. Jinými slovy pro všechna C od 1 do n (n je počet komůrek) se

tato funkce rekurzivně zavolá nejprve s parametry A, C a $\lfloor \mathrm{d}/2 \rfloor$ a poté s parametry C, B a $\lceil \mathrm{d}/2 \rceil$ – pokud pro nějaké C uspěje, vrátí `true`, jinak `false`. Správnost návratové hodnoty funkce plyne z jejího popisu. Samotný program nejprve načte počet komůrek (n) a čísla komůrek (K,L), mezi kterými je třeba najít nejkratší cestu. Pak zavolá funkci `cesta` s parametry K, L a d, kde za d bude postupně dosazovat čísla od 1 do n. Nejmenší d, pro které funkce `cesta` vrátí `true`, je délka nejkratší cesty mezi K a L. Pokud pro žádné d tato funkce nevrátí `true`, cesta mezi K a L neexistuje.

Paměťová složitost algoritmu je $O(\log n)$, neboť při každém rekurzivním volání funkce `cesta` je velikost parametru d poloviční. Časová složitost našeho algoritmu je $O(n(2n)^{O(\log n)}) = n^{O(\log n)}$, neboť každé zavolání funkce `cesta` s d alespoň 2 vyústí v $2n$ dalších zavolání této funkce a hloubka rekurze je $O(\log n)$. Samotný program pak funkci `cesta` volá celkem nejvýše n-krát. Zde by bylo možné dosáhnout (zanedbatelného) zrychlení, pokud bychom nejmenší d, pro které funkce `cesta` uspěje, nehledali sekvenčně, ale metodou půlení intervalu.

Ještě drobný komentář k řešením, která jsme obdrželi: Bohužel většina z nich nesplňovala zadání, neboť jejich paměťová složitost (měřeno nejhorším případem) byla lineární. V zásadě se objevily dva nesprávné postupy řešení: Postup založený na prohledávání do hloubky má paměťovou složitost úměrnou délce hledané cesty, ale její délka může být až počet komůrek! Navíc takovéto řešení je úděsně pomalé (časová složitost $O(n^n)$). Postup založený na prohledávání do šířky (algoritmus vlny) může mít také paměťovou složitost lineární (např. každá komůrka je spojena s každou), ale jeho časová složitost je $O(n^2)$ a je tedy lepší než prohledávání do hloubky; zadání úlohy však též nesplňuje.

```
program mraveniste;
  var n:word;
        { počet komůrek v mraveništi. }
      K,L:word;
        { komůrky mezi kterými hledáme nejkratší cestu }
      i:word;
function chodba(a,b:word):boolean;
  { Je mezi komůrkami a b chodbička? }
  var s:string;
  begin
    write(a,' ',b,': ');
    readln(s);
    chodba:=s='Ano';
  end;
function cesta(a,b:word; delka:word):boolean;
  { Má nějaká cesta mezi a b danou délku? }
  var i:word;
  begin
    if delka=1 then
      if a=b then
        cesta:=false
      else
```

```
        cesta:=chodba(a,b)
    else
      begin
        cesta:=true;
        for i:=1 to n do
          if cesta(a,i,delka div 2) and cesta(i,b,(delka+1) div 2) then
            exit;
        cesta:=false;
      end
  end;
begin
  readln(n,K,L);
  for i:=1 to n do
    if cesta(K,L,i) then
      begin
        writeln('Nejkratší cesta mezi komůrkami ',K,' a ',L,' má délku ',i,'.');
        halt
      end;
  writeln('Mezi ',K,' a ',L,' neexistuje cesta - mraveniště jsou alespoň dvě!');
end.
```

13-2-5 LISP — Martin Mareš

Zadání této úlohy si s námi původně trochu zažertovalo. Zde v ročence je již opravená verze, ale přesto nastíníme, na jaké problémy lze narazit: Příklad uvedený v zadání sice ukazoval správnou činnost funkce `cut`, ale samotné volání funkce muselo být zapsáno trochu jinak: strukturu s kódem stromu je potřeba ochránit před vyhodnocováním (parametry každé funkce se automaticky vyhodnocují, vzpomínáte?) použitím `'` či `quote`. Pak se nám ovšem nebudou vyhodnocovat `nil`y a zůstanou ve struktuře v podobě symbolů, což také není dobré. Proto každý `nil` nahradíme prázdným seznamem `()`, který se přečte přímo jako objekt *nil,* nikoliv symbol `nil`. Výsledek ořezávání mohou některé interpretery (konec konců i ten náš ukázkový) vypsat trochu jinak, protože se budou snažit vyložit si páry jako části seznamu. To ale vůbec není na škodu, oba zápisy totiž popisují tytéž LISPovské objekty (viz ostatně funkce `equal` v první sérii).

Samotné oříznutí binárního stromu je velice jednoduché, těžší je již provést to efektivně. My si nejprve vyřešíme snazší problém, a to ořezávání stromu zleva, tedy odstranění všech čísel menších než zadaná hodnota l. Na tuto cestu je ideálním, i když trochu rozmarným společníkem rekurze:

- Pokud je strom prázdný, pak je výsledkem opět prázdný strom. (Aneb kde nic není, ani kat nebere...)
- Pokud hodnota x uložená v kořeni stromu je menší než l, jak kořen, tak celý levý podstrom jsou mimo rozsah a mají být odříznuty. Výsledkem je tedy pravý podstrom zleva oříznutý hodnotou l.
- Pokud je $x \geq l$, je jak kořen, tak celý pravý podstrom zaručeně v rozsahu, takže stačí vyměnit levý podstrom za jeho oříznutou verzi.

Zapsáno v LISPu:

```
(define (cutl tree l)
  (if (pair? tree)
    (let (
      (root (geta tree))          ; kořen
      (left (geta (getb tree)))   ; levý syn
      (right (getb (getb tree)))) ; pravý
      (if (< root l)
        (cutl right l)
        (cons root (cons (cutl left l) right))))
    nil))                         ; prázdný strom
```

(mimochodem, jelikož netestujeme, zda je strom prázdný, nýbrž zda jeho kořen je či není pár, bude nám tento program fungovat i pokud se ve stromě omylem objeví symboly `nil`).

Analogicky naprogramujeme oříznutí zprava `cutr` tak, že v celém programu prohodíme levou a pravou stranu a menšítko nahradíme většítkem. Požadovanou funkci `cut` pak získáme snadným složením obou jednostranných ořezání:

```
(define (cut tree l r) (cutr (cutl tree l) r))
```

Jelikož na každé hladině stromu provedeme pouze konstantní počet operací, má naše funkce časovou složitost lineární s hloubkou stromu d (ovšem pozor, to nemusí znamenat logaritmickou s počtem vrcholů, protože ne každý vyhledávací strom je vyvážený). V lepší ani doufat nemůžeme, snadno lze totiž sestrojit příklady, v nichž je nutné změnit některý z vrcholů na nejnižší hladině a od kořene se k takovému vrcholu nemůžeme rychleji dostat. Paměťová složitost je rovněž $O(d)$ (pokud do ní nepočítáme vstup), i když výsledkem naší funkce může být strom s daleko větším počtem vrcholů – uvědomte si, že mnohé části zkonstruovaného stromu jsou sdílené se stromem původním a že jsme opravdu přidali pouze $O(d)$ nových objektů (mimochodem, v LISPu můžeme v každém kroku výpočtu vytvořit nejvýše jeden objekt, takže paměťová složitost našich programů nikdy nemůže být vyšší než časová (z toho se na první pohled zdá vymykat primitivum `list` (poznámky v tomto spisku už také začínají být poněkud LISPovské, není-liž pravda (dočetl vůbec někdo až sem (do páté úrovně vnoření?)?)?), které ovšem není ničím než zkratkou za opakované volání `cons`, takže nepracuje v konstantním čase)).

13-3-1 Hip hop — Tomáš Vyskočil

A jak to všechno dopadlo? Vašek i s jeho hrou by byl asi spokojen, protože většina z vás napsala funkční program. Samozřejmě se našli i tací, jejichž řešení by mu spíše zamotala hlavu a nebo kterých by se už pro malé hrací plány ve svém životě nedočkal.

Většina z vás tento problém řešila tzv. Dijkstrovým algoritmem, který je v tomto případě ve své standarní podobě spíše pověstným kanónem na vrabce.

A teď již k našemu algoritmu. Nejprve načteme na vstupu N vrcholů a M hran. Protože víme, že délka těchto hran je přirozené číslo menší nebo rovné dvanácti, můžeme každou hranu rozdělit na tolik jednotkových hran, kolik je její délka. Tak nám vznikne neohodnocený graf. Na tento graf již můžeme použít algoritmus prohledávání do šířky, který všichni jistě dobře znáte.

Algoritmus má časovou i paměťovou složitost $O(N + M)$.

Poznámka pro chytré hlavy: Pokud by konstanta omezující délku hrany byla větší, není toto řešení moc praktické, protože pro omezení d se paměťová složitost zvětší d-krát. Na tento případ lze s úspěchem použít modifikaci Dijkstrova algoritmu, kde se místo obvyklé haldy na dosud nezpracované vrcholy vhodně použije pole velikosti d (všimněte si, že při omezení délky hrany na d se bude vzdálenost libovolných dvou dosažených ale dosud nezpracovaných vrcholů lišit nejvýše o d).

```
#include <stdio.h>

#define MAX_FRONT 1000
#define MAX_VRCH 100

int f[MAX_FRONT][2], zacf, konf;            /* Takto je deklarovaná fronta */
int h[12*MAX_VRCH][12*MAX_VRCH], ps[12*MAX_VRCH];
                                            /* Zde jsou uloženy vstupy */
int v[12*MAX_VRCH];                         /* Zde jsou uloženy mezivýsledky */
int i, j, n, m, a, b, delka, poc, pos, start, cil, ted;   /* a pomocné proměnné */

int main (void)
{
    for (i=0; i<12*MAX_VRCH; i++)
        v[i]=-1;
    /* Načteme vstup */
    scanf ("%d %d", &n, &m);
    poc=n;
    for (i=0; i<m; i++){
        scanf ("%d %d %d", &a, &b, &delka);
        a--; b--;
        pos=a;
        for (j=1; j<delka; j++){
            h[pos][ps[pos]++]=poc; h[poc][ps[poc]++]=pos;
            pos=poc;
            poc++;
        }
        h[pos][ps[pos]++]=b; h[b][ps[b]++]=pos;
    }
    scanf ("%d %d", &start, &cil);
    start--; cil--;
    /* Prohledávání do šířky */
    zacf=0; konf=1; f[0][0]=cil; f[0][1]=0;
    do{
```

```
        ted=f[zacf][0];
        v[ted]=f[zacf][1];
        for (i=0; i<ps[ted]; i++)
            if (v[h[ted][i]]==-1){
                f[konf][0]=h[ted][i]; f[konf][1]=v[ted]+1;
                konf= (konf+1)%MAX_FRONT;
                if (zacf==konf){
                    printf ("Error: Prilis mala fronta!\n");
                    exit (0);
                }
            }
        zacf= (zacf+1)%MAX_FRONT;
    }while (zacf!=konf && v[start]==-1);
    /* Vypíšeme výsledek */
    if (v[start]!=-1){
        i=start; delka=v[start];
        while (i!=cil){
            if (i<n)
                printf ("%d->", i+1);
            j=0;
            while (v[h[i][j]]!=v[i]-1)
                j++;
            i=h[i][j];
        }
        printf ("%d\n", cil+1);
    }else
        printf ("Cesta mezi %d a %d neexistuje!\n", start+1, cil+1);
    return 0;
}
```

13-3-2 Fairies rights — Zdeněk Dvořák

Jak jeden z našich řešitelů konstatoval: je to jednoduchá dynamika. Budeme počítat cenu převodu prvních k písmen originálu na prvních l písmen výtisku (označme si ji $c[k,l]$). Je-li k-té písmeno originálu shodné s l-tým písmenem výtisku, je tato cena $c[k-1,l-1]$ (k-té písmeno necháme beze změny). Jinak můžeme tento převod provést jedním z těchto způsobů:

- k-té písmeno originálu odstraníme. Pak musíme převést prvních $k-1$ písmen originálu na l písmen výtisku, tedy cena bude $c[k-1,l]+$cena odstranění.
- k-té písmeno originálu změníme na l-té písmeno výtisku. Cena bude $c[k-1,l-1]+$ cena změny.
- K originálu přidáme l-té písmeno výtisku. Cena bude $c[k,l-1]+$cena přidání.

Z těchto možností si vybereme tu nejlepší (tj. s minimální cenou). Pro výpočet $c[k,l]$ tedy potřebujeme znát pouze $c[k-1,l]$, $c[k-1,l-1]$ a $c[k,l-1]$. Budeme-li při výpočtu postupovat ve vhodném pořadí (např. po řádcích matice c), budeme

je už ve chvíli zjišťování $c[k,l]$ znát. Hledanou cenu na konci snadno zjistíme jako $c[M,N]$, kde M je délka originálu, N délka výtisku. Správnost algoritmu je zřejmá z popisu. Algoritmus je konečný, neboť nepoužívá žádné netriviální cykly.

Kdybychom chtěli znát pouze cenu převodu, stačilo by nám ukládat si pouze aktuální řádek a dosáhli bychom lineární paměťové složitosti. Chceme však také vědět, jakým způsobem jsme této ceny dosáhli. Budeme si tedy navíc ukládat, kterou z možností jsme si zvolili, a paměťová složitost tedy bude $O(MN)$.

Na výpočet jednoho prvku c potřebujeme konstantní počet akcí, tedy časová složitost bude dohromady také $O(MN)$.

```
#include <stdio.h>

#define MAXLEN 100

char orig[MAXLEN+1], pozm[MAXLEN+1];
int vaha_prid, vaha_ods, vaha_zmen;

int c[MAXLEN+1][MAXLEN+1];
int dirs[MAXLEN+1][MAXLEN+1];

int main (void)
  {
  int i, j, lorig, lpozm, adir;
  scanf ("%d%d%d%s%s", &vaha_prid, &vaha_ods, &vaha_zmen, orig, pozm);
  lorig=strlen (orig);
  lpozm=strlen (pozm);
  for (i=0; i<=lpozm; i++)
    {
    c[0][i]=vaha_prid*i;
    dirs[0][i]=2;
    }
  for (j=0; j<=lorig; j++)
    {
    c[j][0]=vaha_ods*j;
    dirs[j][0]=1;
    }
  for (j=1; j<=lorig; j++)
    for (i=1; i<=lpozm; i++)
      {
      if (orig[j-1]==pozm[i-1])
        {
        c[j][i]=c[j-1][i-1];
        dirs[j][i]=7;
        }
      else
        {
        int costd[3]={c[j-1][i]+vaha_ods, c[j][i-1]+vaha_prid, c[j-1][i-1]+vaha_zmen};
        int amin=0;
        if (costd[1]<costd[0]) amin=1;
```

```
      if (costd[2]<costd[amin]) amin=2;
      c[j][i]=costd[amin];
      dirs[j][i]=amin+1;
      }
    }
  printf ("Vaha zmen je %d\n", c[lorig][lpozm]);
  for (j=lorig, i=lpozm, adir=-1; i||j; )
   {
   int dir=dirs[j][i]; dirs[j][i]=adir; adir=dir;
   j-=adir&1; i-=!! (adir&2);
   }
  for (; adir>=0; adir=dirs[j][i])
   {
   switch (adir)
    {
    case 3:
     printf ("Zmenit '%c' na pozici %d na '%c'\n", orig[j], i+1, pozm[i]);
     break;
    case 1:
     printf ("Odstranit '%c' na pozici %d\n", orig[j], i+1);
     break;
    case 2:
     printf ("Pridat '%c' za pozici %d\n", pozm[i], i);
     break;
    }
   j+=adir&1; i+=!! (adir&2);
   }
  return 0;
  }
```

13-3-3 Konvertor — Dan Kráľ

Tato úloha se řadí bezesporu mezi ty techničtější v našem semináři. V zadání úlohy nikde nebylo řečeno, že převáděné číslo je natolik malé, že ho lze uložit do standartního číselného typu v pascalu či céčku. Proto jsem řešení, která byla založena na tom, že si převáděné číslo uložila do proměnné, ohodnotil nejvýše 9 body.

Pokud přijmeme omezení, že převáděné číslo nelze uložit do proměnné, nezbývá nic jiného, než si naprogramovat několik procedur pro práci s dlouhými čísly. Zkusme si rozmyslet, jak by vypadala procedura, která sečte dvě čísla v soustavě o základu z. Algoritmus pro $z > 0$ zná jistě většina z vás: Nechť $A = a_n \dots a_1 a_0$ a $B = b_n \dots b_1 b_0$ jsou sčítaná čísla. První cifra výsledku bude $v_0 = (a_0 + b_0) \bmod z$ a přenos (kolik musíme přičíst k další cifře) bude $c_0 = (a_0 + b_0) \operatorname{div} z$. Obecně $v_i = (a_i + b_i + c_{i-1}) \bmod z$ a $c_i = (a_i + b_i + c_{i-1}) \operatorname{div} z$. Jak se situace změní, když bude $z < 0$? Přenos c_i se nebude k součtu následujících cifer přičítat, ale odčítat (a to pro i sudé i liché). Tedy $v_i = (a_i + b_i - c_{i-1}) \bmod |z|$ a $c_i = (a_i + b_i + c_{i-1}) \operatorname{div} |z|$;

zde by stálo zato podotknout, že a mod b v předchozím vzorečku je vždy číslo mezi 0 a $b-1$ (a to i pro a záporné) a a div b je největší celé číslo menší než a/b. Podrobný důkaz těchto formulek je snadné cvičení na matematickou indukci. Povšimněte si, že naše vzorečky jdou snadno zobecnit na přičtení čísla $\alpha \cdot B$ (α je nezáporné celé číslo uložitelné do proměnné) k číslu A:

Pro $z > 0$:

$$v_i = (a_i + \alpha b_i + c_{i-1}) \bmod z$$

$$c_i = (a_i + \alpha b_i + c_{i-1}) \operatorname{div} z$$

Pro $z < 0$:

$$v_i = (a_i + \alpha b_i - c_{i-1}) \bmod |z|$$

$$c_i = (a_i + \alpha b_i - c_{i-1}) \operatorname{div} |z|$$

Časová složitost přičtení (malého) násobku jednoho čísla k druhému je pak lineární v délce výsledného součtu.

Zbývá tedy dořešit problém, jak využít právě popsaný algoritmus k převodu mezi různými soustavami. Nechť máme číslo $A = a_n \ldots a_0$ v soustavě o základu z a chceme ho převést na A' do soustavy o základu z'. Nechť Z_i je číslo z^i vyjádřené v soustavě o základu z'. Pak $A' = (a_n z^2 + a_{n-1} z) Z_{n-2} + \ldots + (a_2 z^2 + a_1 z) Z_0 + a_0 Z_0$. Pokud by všechny výrazy $(a_n z^2 + a_{n-1} z), \ldots, (a_2 z^2 + a_1 z)$ byly nezáporné, byli bychom hotovi: Nejprve bychom k nule přičetli a_0-násobek čísla Z_0, potom k takto získanému výsledku $(a_2 z^2 + a_1 z)$-násobek Z_0, k tomuto součtu $(a_4 z^2 + a_3 z)$-násobek Z_2 atd. Pokud známe Z_i, pak Z_{2i} snadno spočteme jako z^2-násobek Z_i. Časová složitost takto navrženého algoritmu je kvadratická v délce zadaných čísel – počet kroků je lineární a časová složitost každého kroku je též lineární. Bohužel svět není takto jednoduchý a může se stát, že například $d = (a_2 z^2 + a_1 z)$ je záporné číslo. Potom místo d-násobku přičteme $(|z|^3 + a_2 z^2 + a_1 z)$ (což bude jistě nezáporné číslo) a v následujícím kroku přičteme místo $(a_4 z^2 + a_3 z)$-násobku Z_2 jeho $(a_4 z^2 + a_3 z + z)$-násobek. Vzhledem k tomu, že A je kladné, nebude toto přesouvání zápornosti nekonečné. Tím je popis našeho algoritmu ukončen.

Program se drží výše uvedeného postupu. Procedura `nacti`, resp. `vypis`, načte, resp. vypíše, číslo uložené v typu `cislo`. Procedura `pricti_nasobek` implementuje přičítání násobku tak, jak je popsané v druhém odstavci. Procedura `convert` provádí konverzi tak, jak je popsaná ve třetím odstavci. V proměnné `rad` si pamatujeme postupně hodnoty $Z_0, Z_2, \ldots$, v proměnné `nove` pak součty po jednotlivých krocích.

```
program convertor;
const MAX=100;
type cislo=record
            zaklad:integer;
```

```
            delka: word;
            cislice: array[1..MAX] of integer;
          end;

{Vypíše dané číslo}
procedure vypis(c:cislo);
var i:word;
begin
  for i:=c.delka downto 1 do
    if c.cislice[i]<10 then
      write(c.cislice[i])
    else
      write(chr(c.cislice[i]+55));
  if c.delka=0 then write(0);
  writeln;
end;

{Načte číslo od uživatele}
procedure nacti(var c:cislo);
var i:word;
    s:string;
begin
  readln(c.zaklad);
  readln(s);
  c.delka:=length(s);
  for i:=c.delka downto 1 do
    if s[c.delka+1-i]<='9' then
      c.cislice[i]:=ord(s[c.delka+1-i])-48
    else
      c.cislice[i]:=ord(upcase(s[c.delka+1-i]))-55;
end;

{Přičte 'nasobek' násobek čísla 'c' k 'd'}
procedure pricti_nasobek(var c:cislo; nasobek:word; d:cislo);
var i:word;
    prenos:integer;
begin
  prenos:=0; i:=1;
  while (prenos<>0) or (i<=d.delka) do begin
    if (c.delka<i) then begin
      inc(c.delka);
      c.cislice[c.delka]:=0;
    end;
    c.cislice[i]:=c.cislice[i]+nasobek*d.cislice[i]+prenos;
    if (c.cislice[i]>0) then
      prenos:=c.cislice[i] div c.zaklad
    else
      prenos:=1+(c.cislice[i]+1) div c.zaklad;
    c.cislice[i]:=c.cislice[i]-prenos*c.zaklad;
    inc(i);
  end;
end;

{Hlavní převáděcí procedura}
procedure convert(nova:integer; var c:cislo);
var rad:cislo;
    nove:cislo;
    i:word;
```

```
    kolik:integer;
begin
  rad.zaklad:=nova; rad.delka:=1; rad.cislice[1]:=1;
  nove.zaklad:=nova; nove.delka:=0;
  pricti_nasobek(nove,c.cislice[1],rad);
  i:=2;
  while (i<=c.delka) do begin
    if c.delka=i then c.cislice[c.delka+1]:=0;
    kolik:=c.zaklad*c.zaklad*c.cislice[i+1]+c.zaklad*c.cislice[i];
    if kolik<0 then begin
      kolik:=kolik-c.zaklad*c.zaklad*c.zaklad;
      inc(c.cislice[i+2]);
    end;
    pricti_nasobek(nove,kolik,rad);
    inc(i,2);
    pricti_nasobek(rad,c.zaklad*c.zaklad-1,rad);
  end;
  c:=nove;
end;

var c:cislo;
    nova:integer;
begin
  nacti(c);
  readln(nova);
  convert(nova,c);
  vypis(c);
end.
```

13-3-4 Šťastná čísla — Pavel Šanda

Pro tentokrát byli zlí skřítkové radílkové obzvláště potměšilí a rozdělili řešitele na dva zcela odlišné typy.

Zatímco prví se šťourají pouze ve svých „šč“, druzí studují ve svém entuziasmu každé číslo na šťasnost – potměšilost spočívá v tom, že „šč“ jsou rozmístěna na číselné ose velmi řídce a my musíme zkoumat exponenciální počet číslíček (a to věčně věkův), což nadšení nejednoho človíčka pochroumá, stejně tak jako pana Una, kterýžto ve své zběsilé honbě za čísly nemá pro podobné výstřednosti pochopení.

Vydáme se proto ve stopách prvých.

Dohoda: Píšu-li číslo, myslím tím šťastné číslo.

Šťastná čísla budeme postupně ukládat do fronty. Uvažujme první verzi algoritmu: Jsme ve frontě u čísla k (stanovme první $k = 1$). Na konec fronty dáme všechna další čísla, která vzniknou jako $k \cdot 3, k \cdot 7, k \cdot 11$ (def. „šč“). Přesuneme se k dalšímu číslu ve frontě a opakujeme tentýž postup. Nejprve nahlédněme, že tímto postupem nagenerujeme libovolné číslo v konečném počtu kroků (sporem). Problém zde spočívá v tom, že budeme zbytečně generovat mnoho vysokých čísel, která nebudou ve výsledku vůbec třeba, krom toho nebudeme vědět kdy přestat, atp. Tomu se vyhneme vzestupným generováním čísel – tj. v každém kroku dostaneme vždy nejmenší možné číslo, které jsme

dosud nenalezli (po n krocích budeme proto mít požadovaných n šťastných čísel). Tím odstraníme náš problém, který vznikl např. tím, že jsme číslo $11 \cdot k$ zařadili do fronty dříve než $3 \cdot 3 \cdot k$.

Elegantní způsob, jak čísla vzestupně generovat, využívá tří ukazatelů p_3, p_7, p_{11}. Ty ukazují na nejmenší taková čísla, pro která platí $3 \cdot \mathit{fronta}[p_3] > k$, $7 \cdot \mathit{fronta}[p_7] > k$, $11 \cdot \mathit{fronta}[p_{11}] > k$. Na konec fronty pak připojím nejmenší z čísel $i \cdot \mathit{fronta}[p_i]$, $i \in \{3, 7, 11\}$.

Poslední detail – mohlo by se stát, že nějaké číslo vygenerujeme dvakrát ($\mathit{fronta}[p_3] = 3^{x-1} \cdot 7^y \cdot 11^z$, $\mathit{fronta}[p_7] = 3^x \cdot 7^{y-1} \cdot 11^z$). Musíme tedy při generování každého čísla zkontrolovat, jestli nevznikne i při vynásobení čísel, na která ukazují ostatní ukazatelé a v tom případě inkrementovat i tato p_i.

Odhad složitostí: Časová složitost bude lineární vzhledem k počtu hledaných šťastných čísel. Paměťová složitost našeho programu je taktéž lineární.

Opravdová legračina začíná v okamžiku, kdy si všimneme, že čísla, která byla vynásobena 3, 7 i 11, už nikdy během výpočtu nebudou třeba – mohli bychom je zapomínat (implementace např. dynamickým seznamem).

Otázka, jaká by byla paměťová složitost nyní, pana Una velmi rozrušila (zjistil totiž mystickou souvislost s jeho teorií šťastných čísel – víc mu žel vědomo není). Rozhodl se proto vykonat k Vám návštěvu a vyškemrat nové rady. Za důkaz paměťové složitosti (zajímal by nás její co nejlepší jak horní, tak dolní odhad) je možno získat až 4 body.

```
#include <stdio.h>
#define min (a, b, c)  (a<b  ?  (a<c ?a:c) :  (b<c ?b:c))
#define q (a)  (  (d[p##a]*a==d[i]  ? p##a++ : 0) )

void main (){
  int n, p3, p7, p11, i, d[1000];                  /* počet, ukazatele, fronta */
  scanf ("%d", &n);

for (p3=p7=p11=0, d[0]=i=1; i<=n; i++)
    d[i]=min (d[p3]*3, d[p7]*7, d[p11]*11),  q (3), q (7), q (11),  printf ("%d␣", d[i]);
}
```

13-3-5 LISP — Martin Mareš

a) Na funkci *Map* nebylo opravdu nic těžkého – stačilo si uvědomit, že v LISPu lze funkce předávat jako cokoliv jiného a místo pevného jména volané funkce drze napsat proměnnou. A pak použít už několikrát popisované rekurzivní procházení seznamů:

```
(define (Map f l)
  (if l
    (cons (f (geta l)) (Map f (getb l)))
    nil))
```

Tento program má lineární časovou i paměťovou složitost (ovšem nepočítaje v to funkci `f`), a to je evidentně nejlepší možné.

b) Prohledávání stromu do šířky se ukázalo být poněkud zapeklitějším. Náš vzorový řešitel mohl uvažovat třeba takhle: Prohledávání do šířky, to se přeci odjakživa dělalo pomocí fronty, ve které se schovávaly už nalezené, ale ještě nezpracované vrcholy – prostě se začalo s kořenem a v každém kroku se odebral první vrchol z fronty a na konec se přihodili jeho synové. Hmmm, fronta ... to je přeci seznam, takže věc dočista lispovská, že lispovštější si snad už ani nejde představit. Aha, ale jak přidávat na konec seznamu bez toho, abychom ho pokaždé museli znovu hledat? Co na tom, tak si budeme pamatovat, kde končí. A tak se zrodilo následující řešení:

```
(define (append last v)
  (if v
    (block
       (setb last (list v))
       (getb last))
    last))

(define (BFS2 last first)
  (if first           ; fronta není prázdná
    (let ((v (geta first))      ; vrchol
          (w (geta v))          ; hodnota
          (l (geta (getb v)))   ; levý syn
          (r (getb (getb v))))  ; pravý syn
      (cons w
        (BFS2
           (append (append last l) r)
           (getb first))))
    nil))

(define (BFS t)
  (let ((q (list t))) (BFS2 q q)))
```

Funkce *BFS* založí jednoprvkovou frontu obsahující kořen stromu t a předá řízení funkci *BFS2*, která provádí samotné prohledávání a předává si přitom dva parametry: *first* ukazující na počátek fronty a *last* ukazující na konec. Pokud fronta ještě není prázdná, odebere z ní první vrchol v a vrátí seznam obsahující hodnotu tohoto vrcholu připojenou před výsledek téže funkce rekurzivně zavolané na frontu, do níž jsme zavoláním funkce *append* přidali syny vrcholu v a samotný vrchol v odebrali (na tomto místě se dopouštíme drobné podlosti: využíváme toho, že se argumenty funkce vyhodnocují zleva doprava a schválně předáváme nejdříve *last* a pak teprve *first*, takže nejdříve vrcholy přidáváme

a pak teprve odebíráme, tudíž se nám fronta nemůže před koncem výpočtu nikdy ani na chvilku vyprázdnit, a tak v *append* nemusíme ošetřovat přidávání k prázdné frontě).

Co dodat? Snad jen to, že časová i paměťová složitost jsou lineární, že rekurze je nejlepším přítelem programátora a že venku nádherně svítí měsíc.

13-4-1 Fotografové — Dan Kráľ

Již samotné bodové hodnocení této úlohy naznačuje, že tato úloha patří mezi ty obtížnější v našem semináři. Celkem přišlo šest řešení, z nichž pouze dvě obsahovala správný (tedy funkční) polynomiální algoritmus, byť ne s optimální časovou složitostí a navíc bez kompletního důkazu správnosti.

Program řešící tuto úlohu samozřejmě nejprve načte počet domů v Nudlové Lhotě a popis snímků Alfonse a Adalberta. Nejprve si povšimněme, že pokud Adalbert chce mít na svém snímku pouze dva domy, pak je nutné tyto dva domy natřít stejnou barvou. Abychom se takovýchto speciálních případů zbavili, postavíme mezi takové dva domy zídku a spojíme je v jeden (v našem programu to bude odpovídat vhodnému přečíslování domů v Nudlové Lhotě). Tím Adalbert bude spokojený se všemi svými snímky, které původně obsahovaly oba takto spojené domy (a tedy tyto Adalbertovy snímky z dalšího zpracovávání vyřadíme). Pokud se nám stane, že na některém z Alfonsových snímků bude nyní již jen jediný dům (vzniklý z několika původních), pak zřejmě domy v Nudlové Lhotě nelze obarvit dle požadavků obou umělců a náš program o tom vypíše vhodnou zprávu.

Předpokládejme tedy nadále, že každý Adalbertův snímek obsahuje alespoň 3 domy. Domy ve Lhotě si očíslujeme od jedné podél cesty. Obarvení prvních k domů (tj. těch s čísly 1 až k) nazveme optimální, pokud:

- používá maximální možný počet barev
- podmínky vyplývající ze všech snímků Adalberta nebo Alfonse, které obsahují pouze domy s čísly 1 až k, jsou splněny
- mezi všemi obarveními splňujícími první dvě pravidla má toto největší „index poslední dvojice“

Index poslední dvojice obarvení je největší číslo i takové, že existuje j takové, že $i < j \leq k$ a domy i a j jsou obarveny stejnou barvou. Pokud jsou všechny domy obarveny různými barvami, je index poslední dvojice roven nule. Označme dále b_k počet barev a i_k index poslední dvojice optimálního obarvení prvních k domů. Zřejmě platí $b_{k+1} \leq b_k + 1$. Uvažujme nějaké optimální obarvení prvních k domů a dále označme x a y barvu $(k-1)$-ního a k-tého domu podle tohoto obarvení. Potom platí:

- Pokud $x = y$, pak $b_{k+1} = b_k + 1$ a $i_{k+1} = i_k$. Dům s číslem $k+1$ lze totiž v tomto případě obarvit úplně novou barvou. Naopak kaž-

dé optimální obarvení prvních $k+1$ domů, používá pro obarvení $(k+1)$-ního domu úplně novou barvu, neboť jinak by existovalo obarvení prvních k domů používající b_k+1 barev.

- Pokud $x \neq y$ a obarvením $(k+1)$-ního domu úplně novou barvou získáme dobré obarvení, pak $b_{k+1} = b_k+1$ a $i_{k+1} = i_k$. Každé optimální obarvení prvních $k+1$ domů totiž musí pro obarvení $(k+1)$-ního domu použít úplně novou barvu (jinak by existovalo obarvení prvních k domů používající b_k+1 barev) a z toho ihned plyne $i_{k+1} = i_k$.
- Pokud $x \neq y$ a obarvením $(k+1)$-ního domu úplně novou barvou získáme špatné obarvení, musí být $b_k = b_{k+1}$. Kdyby totiž $b_{k+1} = b_k+1$, pak optimální obarvení prvních $k+1$ by obarvilo prvních k domů sice jen b_k barvami, ale s větším indexem poslední dvojice než námi uvažované optimální obarvení prvních k domů.
- Pokud $b_k = b_{k+1}$ (a tedy $x \neq y$) a Alfons *nechce* pořídit snímek pouze s domy s čísly k a $k+1$, pak $i_{k+1} = k$, neboť $(k+1)$-ní dům můžeme obarvit barvou y.
- Pokud $b_k = b_{k+1}$ (a tedy $x \neq y$) a Alfons *chce* pořídit snímek pouze s domy s čísly k a $k+1$, pak $i_{k+1} < k$. Protože $(k+1)$-ní dům lze obarvit barvou x, platí $i_{k+1} = k-1$.

Právě provedený rozbor případů nám dává návod na konstrukci optimálního obarvení všech domů v Nudlové Lhotě.

Zkusme se nyní zamyslet nad samotnou implementací výše popsaných myšlenek. Náš program se skládá z procedur `nacti`, `spoj` a `spocitej`. První z procedur, `nacti`, načte počet domů v Nudlové Lhotě a informace o snímcích, které chtějí umělci pořídit. Druhá procedura „staví“ zdi mezi ty dvojice domků, které chce samostatně vyfotografovat Adalbert: Hodnota `spojit[i]` v poli `spojit` je `true`, pokud Adalbert chce pořídit fotografii s domy $i-1$ a i. Hodnota `spojen_na[i]` pole `spojen_na` udává číslo domu s původním číslem i po postavení zídek a spojení domů dohromady. Povšimněte si, jak je v této proceduře vyřešen test, zda došlo ke spojení dvou domů v rámci jedné větší Adalbertovy fotografie, či zda došlo ke spojení všech domů z jedné Alfonsovy fotografie. Do pole `spojeno` uloží tato procedura pro každý dům, z kolika původních domů se skládá.

Obraťme nyní naši pozornost k proceduře `spocitej`. Budeme postupovat dle výše uvedeného postupu a do pole `barev` budeme ukládat počty barev, které používá optimální obarvení prvních k domů, do pole `barva` barvu k-tého domu v optimálním obarvení a do pole `posledni` index poslední dvojice optimálního obarvení. Aby náš algoritmus byl co nejrychlejší, tak si před samotným výpočtem uložíme do pole `ruznost`, zda Alfons chce pořídit fotografii dané dvojice domů (`ruznost[i]` je `true`, pokud Alfons chce vyfotografovat samostatně dvojici domů s čísly $i-1$ a i). Do pole `ruznost` ukládáme informace

o Adalbertových snímcích; číslo `ruznost[i]` je rovno největšímu j takovému, že Adalbert chce pořídit snímek právě domů s čísly j až i.

Časová a paměťová složitost naší implementace popsaného algoritmu je lineární, tedy $O(N + M)$, kde N je počet domů Nudlové Lhoty a M je počet snímků, které chtějí oba umělci pořídit.

```
program nudlova_lhota;
const MAX=100;
var
  ruzny, rovny : array[1..MAX, 1..2] of word;
  spojeno : array[1..MAX] of word;
  n, mruz, mrov: word; { počet domů a počet snímků }

procedure nacti;
var
  i : word;
begin
  readln(n,mruz,mrov);
  for i := 1 to mruz do readln(ruzny[i, 1], ruzny[i, 2]);
  for i := 1 to mrov do readln(rovny[i, 1], rovny[i, 2]);
end;

procedure spoj;
var
  spojen_na : array[1..MAX] of word;
  spojit : array[1..MAX] of boolean;
  i, j : word;
begin
  for i := 1 to n do
    spojit[i] := false;
  for i := 1 to mrov do
    if rovny[i, 1]+1=rovny[i, 2] then
      spojit[rovny[i, 2]] := true;
  j := 0;
  for i := 1 to n do
    if spojit[i] then begin
      spojen_na[i] := j;
      inc(spojeno[j]);
    end
    else begin
      inc(j);
      spojen_na[i] := j;
      spojeno[j] := 1;
    end;
  n := j;
  for i := 1 to mruz do begin
    ruzny[i, 1] := spojen_na[ruzny[i, 1]];
    ruzny[i, 2] := spojen_na[ruzny[i, 2]];
    if ruzny[i,1]=ruzny[i,2] then begin
      writeln('Domy nelze obarvit.');
      exit
    end
  end;
  j := 0;
  for i := 1 to mrov do begin
    if rovny[i,2]-rovny[i,1]>spojen_na[rovny[i,2]]-spojen_na[rovny[i,1]] then
```

```
      continue;
    inc(j);
    rovny[j,1]:=spojen_na[rovny[i,1]];
    rovny[j,2]:=spojen_na[rovny[i,2]];
  end;
  mrov := j;
end;

procedure spocitej;
var
  ruznost : array[1..MAX] of boolean;
  nejvetsi : array[1..MAX] of word;
  barev : array[1..MAX] of word;
  barva : array[1..MAX] of word;
  posledni : array[1..MAX] of word;
  i, j:word;
begin
  for i := 1 to n do
    ruznost[i] := false;
  for i := 1 to mruz do
    if ruzny[i,1]+1=ruzny[i,2] then
      ruznost[ruzny[i,2]] := true;
  for i := 1 to n do
    nejvetsi[i]:=0;
  for i := 1 to mrov do
    if nejvetsi[rovny[i,2]]<rovny[i,1] then
      nejvetsi[rovny[i,2]]:=rovny[i,1];
  barev[1] := 1; barva[1] := 1; posledni[1] := 0;
  barev[2] := 2; barva[2] := 2; posledni[2] := 0;
  for i := 3 to n do
    if nejvetsi[i] <= posledni[i-1] then begin
      barev[i] := barev[i-1]+1;
      barva[i] := barev[i];
      posledni[i] := posledni[i-1];
    end
    else
      if ruznost[i] then begin
        barev[i] := barev[i-1];
        barva[i] := barva[i-2];
        posledni[i] := i-2;
      end
      else begin
        barev[i] := barev[i-1];
        barva[i] := barva[i-1];
        posledni[i] := i-1;
      end;
  writeln('Maximální počet barev: ', barev[n]);
  for i := 1 to n do
    for j := 1 to spojeno[i] do
      write(barva[i], ' ');
    writeln;
end;

begin
  nacti;
  spoj;
  spocitej;
end.
```

13-4-2 Okružní jízda — Robert Špalek

Na úvod si zopakujeme značení ze zadání: L budiž délka závodní trasy, K počet stanovišť na ní a $p_1, \ldots, p_K$ vzdálenosti jednotlivých stanovišť od hlavního města ležícího na trati. $a_1, \ldots, a_K$ pak jsou počty kilometrů, které lze ujet na benzín načerpaný na daném stanovišti.

A nyní již samotné řešení. Předpokládejme na chvíli, že auto může jet na dluh – to znamená, že objem benzínu v nádrži může být dočasně záporný. Uvědomíme-li si, že množství benzínu rozestavěné podél trati stačí přesně na projetí okruhu, je zřejmé, že auto bude mít jak na startu tak v cíli prázdnou nádrž.

Nechme auto vyjet z prvního stanoviště. Pro účely snazšího zápisu si přidáme stanoviště $K+1$ ve vzdálenosti $L + p_1$, které bude v podstatě shodné s prvním stanovištěm, pouze nám umožní vyhnout se rotaci. Označme si b_i objem benzínu v okamžiku, kdy auto přijíždí do města i, ještě než natankuje. Je zřejmé, že

$$b_i = \left(\sum_{j=1}^{i-1} a_i \right) - (p_i - p_1), \quad b_1 = b_{K+1} = 0.$$

Označme si jako m číslo stanoviště, ve kterém nabývá posloupnost $\{b_i\}_{i=1}^{l}$ minimum. Ukážeme, že toto stanoviště (anebo jakékoliv jiné, ve kterém má b_i stejnou hodnotu) je jediné správné startovní stanoviště.

Stačí uvážit, co by se stalo, kdybychom vyjeli z jiného stanoviště. Posloupnost b_i by se zrotovala o příslušný počet stanovišť doleva a hodnoty b_i by se posunuly tak, že v počátečním stanovišti by byly 0. Pokud se tedy v počátečním stanovišti nenabývalo minimum, bude mít b_m hodnotu menší než 0, a tedy autu někde dojde benzín.

Pokud chceme, aby objem benzínu nebyl nikde záporný, musíme tedy startovat ze stanoviště m.

Program je přímou implementací uvedeného postupu. Oproti tomuto textu je zjednodušen ve smyslu, že auto necháváme vyjíždět z nultého kilometru místo prvního stanoviště, což nám objemy benzínu zmenší o konstantu. Jeho časová složitost $O(K)$ je nejlepší možná, protože každý vstup musíme alespoň přečíst, paměťová složitost $O(1)$ také nejde vylepšit.

```
#include <stdio.h>

int main (void)
{
    int len, count;
    int oil_total = 0, oil_min = 0x7fffffff, oil_pos;
    int i;
```

```
    scanf ("%d %d\n", &len, &count);
    for (i=0; i<count; i++) {
      int pos, oil;
      scanf ("%d %d\n", &pos, &oil);
      if (oil_total - pos < oil_min) {
        oil_min = oil_total - pos;
        oil_pos = pos;
      }
      oil_total += oil;
    }
    printf ("Starting from %d-th kilometer.\n", oil_pos);
    return 0;
}
```

13-4-3 Fazolky — Jan Kára

Tato úloha byla dosti jednoduchá, a tak se značné části řešitelů podařilo nalézt optimální algoritmus pro tuto úlohu. Algoritmus postupně prochází kalíšky od prvního k poslednímu. Průběžně si udržuje počet fazolek m_i, které je nutné přesunout mezi kalíšky i a $i+1$. Pokud je tento počet kladný, je třeba přesouvat doprava, pokud je počet záporný, je třeba přesouvat doleva. Jak tento počet udržovat? Mezi kalíšky nula (ten vlastně neexistuje) a jedna je to zřejmě nula (tedy $m_0 = 0$). Mezi kalíšky i a $i+1$ je to pak $m_{i-1} + p_i - 1$, kde p_i je počet fazolí v i-tém kalíšku. Tento vztah snadno ověříme:

- Pokud je $m_{i-1} \geq 0$, tak bylo do kalíšku i z kalíšku $i-1$ třeba přesunout m_{i-1} fazolí. V kalíšku tedy budeme mít $m_{i-1} + p_i$ fazolí. Jednu fazoli musíme v kalíšku nechat a zbytek odsunout dále doprava (vlevo již z indukčního předpokladu nemají žádné uplatnění – m_{i-1} by totiž v tom případě mohlo být menší a nebyl by to tedy nutný počet fazolí).
- Pokud je $m_{i-1} < 0$, tak bylo z kalíšku i do kalíšku $i-1$ třeba přesunout $-m_{i-1}$ fazolí. V kalíšku i již máme $p_i - 1$ volných fazolí. Pokud tyto fazole nestačily na pokrytí přesunu, je třeba ještě z kalíšku $i+1$ do kalíšku i dodat $-m_{i-1} - p_i + 1$ fazolí (tedy m_i je rovno dokazovanému výrazu). Pokud fazole stačily na pokrytí, tak je jejich zbytek nutno odsunout doprava, a tedy dokazovaný výraz pro m_i též platí.

Nyní, když máme dokázaný vztah pro nutný počet přesunů mezi kalíšky i a $i+1$, je již zřejmé, že celkový počet tahů musí být $\sum_{i=1}^{n-1} |m_i|$. Algoritmus tedy pouze průběžně přičítá absolutní hodnoty ze spočteného nutného počtu přesunů k celkovému počtu tahů.

Algoritmus má lineární časovou složitost k počtu kelímků a konstantní paměťovou složitost (nikdy si nepotřebujeme pamatovat více než počet fazolí v aktuálním kelímku, nutný počet přesunů a celkový počet tahů). Správnost algoritmu byla ukázána v popisu.

Program je přímou implementací algoritmu. Za drobnou poznámku snad stojí pouze to, že k celkovému počtu tahů je v programu přičteno i m_n. To je ale pro korektní vstup vždy nula, takže nezpůsobí žádnou chybu.

```
#include <stdio.h>

int main (void)
{
    int n, p;                    /* Počet kalíšků; Počet fazolí v aktuálním kalíšku */
    int tahu = 0, m = 0;         /* Celkový počet tahů; Počet přesouvaných fazolí */

    scanf ("%d", &n);
    while (n--) {
      scanf ("%d", &p);
      m += p-1;
      tahu += (m > 0) ? m : -m;
    }
    printf ("Na uvedeni hry do ciloveho stavu je treba %d tahu.\n", tahu);
    return 0;
}
```

13-4-4 Vodárna — Pavel Šanda

Naštěstí se po přijetí konečného plánu „Co s tunami (shnilých) malin stávkujících zemědělců“ neozval jekot žádného Ancijáše (kormutlivě poukazuje na nesmyslnost data termínu platnosti) a celá věc se tedy mohla předat skupině odborníků (která spokojeně valíc svoji kuličku poznání, nemá pokdy rozvažovat o finalitě svého konání), jež se začala problémem obírat.

Jako obvykle existovalo několikero různých variant řešení – mohlo se speciálním způsobem využít barvení grafu (rozuměj obšlehnout loňskou olympiádu), různě komplikované varianty prohledávání do šířky v opačném směru, nežli jsou orientovány hrany, a konečně se nechalo vystačit i s klasickým prohledáváním do šířky – toho se přidržíme i my.

Vezmeme si libovolný bod grafu x a označíme si všechny vrcholy v grafu, do kterých se lze z x dostat (včetně x samotného). Buďto jsou označeny všechny vrcholy grafu (x je pak kandidát na hledaný vrchol), anebo nám zůstal alespoň jeden neoznačený vrchol y – v tom případě zvolíme $x = y$ a jdeme opět od začátku (Nerušíme však předchozí označení u vrcholů!).

Nechť z je poslední zvolený x. Pak pokud má naše úloha řešení, tak z je jedním z nich. Důkaz tohoto tvrzení je jednoduchý: Nechť pro spor nějaký vrchol r je řešením a vrchol z řešením není. Zřejmě ze žádného z předtím volených x nemohla vést cesta do r, protože potom by z x vedla cesta přes r do všech vrcholů a x by bylo řešením. Tedy v okamžiku počátku prohledávání ze z bylo r ještě „netknuté“. Po konci prohledávání už ale byly projité všechny vrcholy. Tedy existovala cesta ze z do r a ze z přes r se šlo dostat všude. Spor.

Stačí tedy po nalezení vrcholu z (1. fáze) zjistit, zdali se z něj dostanu do všech vrcholů grafu (2. fáze).

Na zjišťování dosažitelnosti vrcholů mi v obou částech algoritmu stačí klasické prohledávání do šířky. Dále si povšimněme, že pokud v 1. fázi hledám dosažitelné vrcholy z x, mohu ignorovat ty větve výpočtu, kde narazím na již označený vrchol (nic nového z toho nevytěžím).

Stran složitostí: Časová složitost je $O(m + n)$, kde n je počet vrcholů a m je počet hran. V 1. fázi z každého vrcholu hledám maximálně jednou (pak je již označen jako prohledaný). Samotné hledání sice nemusí trvat konstatní čas (vzhledem k počtu vrcholů), ale uvědomte si, že všechny vrcholy, které naleznu při hledání, označím, a z označených vrcholů již nehledám. Nalezení všech sousedů jednotlivých vrcholů pak v sumě odpovídá počtu hran v celém grafu. Ve 2. fázi je klasický průchod do šířky, tedy opět $O(m + n)$.

Paměťová složitost je $O(n^2)$, přičemž by šlo modifikací programu živořiti (z čeho to slovo jen vzniklo?) i na $O(m + n)$. Vzhledem k tomu, že v každém kroku obarvím alespoň jeden vrchol, je zřejmá i konečnost.

```
#include <stdio.h>

#define Max 50
int g[Max][Max];                        /* graf – seznam sousedů jednotlivých
                                           vrcholů */
int used[Max];                          /* které vrcholy jsou prošlé */
int uzlu, cest, last;                   /* poslední vrchol použitý jako zdroj */

Prohledávání do šířky z vrcholu x

void wave (int x)
{
  int F[Max], BF, i, I;                 /* fronta, její začátek a aktuální prvek */

  for (BF = 1, F[i=0] = x; i<BF; i++) {  /* vlna */
    if (used[F[i]])
      continue;
    else
      used[F[i]]=1;
    for (I=1; I<=g[F[i]][0]; I++)
      if (!used[g[F[i]][I]])
        F[BF++]=g[F[i]][I];             /* nepoužití sousedé */
  }
}
int main (void)
{
  int a, b, i;

  scanf ("%d%d", &uzlu, &cest);
  for (i = 0; i < cest; i++) {
    scanf ("%d%d", &a, &b);
    g[a][++g[a][0]]=b;
  }
```

```
    for (i = 0; i < uzlu; i++)
      if (!used[i])
        wave (last=i);
    for (i = 0; i < uzlu; i++)
      used[i]=0;
    wave (last);                                  /* projdu celou sítí ? */
    for (i = 0; i < uzlu; i++)
      if (!used[i]) {
        printf ("Nelze\n");
        return 0;
      }
    printf ("Hledany vrchol: d\n", last);
    return 0;
}
```

13-4-5 LISP — Martin Mareš

Stejně jako pro úlohu z minulé série, i zde výtečně poslouží prohledávání do šířky, tentokráte ale bude krapet složitější, neboť si budeme muset pamatovat, ve kterých vrcholech jsme již byli, abychom se nezacyklili. Hledání cesty z v do w tedy bude vypadat asi takto:

1. Fronta Q na začátku obsahuje v, předchůdce p_v nastavíme na $\emptyset$.
2. Dokud je Q neprázdná, odeber z ní vrchol x, všechny jeho dosud neoznačené sousedy označ, přidej je na konec fronty a nastav jejich p_y na x.
3. Pokud je vrchol w označený, nejkratší cestu nalezneme „přeskákáním“ po p_x z w zpět do v. V opačném případě neexistuje z v do w žádná cesta.

Takový algoritmus by si jistě zasloužil důkaz správnosti: Algoritmus se určitě zastaví, a to po nejvýše lineárním počtu kroků, protože každý vrchol a každou hranu grafu projde maximálně jednou. Navíc vrcholy prochází v pořadí podle rostoucí vzdálenosti od v: nejdříve vrchol v samotný, pak všechny vrcholy ve vzdálenosti 1, pak všechny ve vzdálenosti 2 atd., přesněji vždy, když z fronty vybírá vrcholy vzdálené k, přidává na její konec vrcholy vzdálené $k+1$. Přitom p_x vždy ukazuje na předposlední vrchol na nejkratší cestě z v do x (a z toho přímo plyne, že nalezená cesta je opravdu nejkratší a že pokud nějaká cesta existuje, tak ji nalezneme). To celé snadno dokážeme indukcí: pro počáteční vrchol v tvrzení jistě platí. Víme-li, že platí pro všechny vrcholy ve vzdálenosti $< k$, podívejme se na situaci v okamžiku, kdy byl do fronty zařazen poslední vrchol vzdálený $k-1$ (podle indukčního předpokladu byly tento vrchol i všechny vrcholy bližší k v určitě ohodnoceny správně, mají správná p_x a první vrchol ve vzdálenosti $k-1$ nebyl dosud z fronty odebrán). Algoritmus

nyní přidá do fronty právě vrcholy vzdálené k: cokoliv přidá, je dosud neoznačený soused nějakého vrcholu vzdáleného $k-1$ (a ten nemůže mít vzdálenosti větší než k, jenže menší také ne, neboť to by byl již označený); naopak každý vrchol vzdálený k přidá, neboť má nějakého souseda vzdáleného $k-1$ a toho zaručeně máme ve frontě; p_x přitom vždy nastaví právě na tohoto souseda, což je přesně předchůdce x na nejkratší cestě z v. A tím je celé tvrzení dokázáno. (Mimochodem, celé prohledávání jsme mohli ukončit už v okamžiku, kdy jsme dorazili do w, ostatní hodnoty totiž nalezení nejkratší cesty ani její tvar nemohou ovlivlit. Toho také v našem programu využijeme.)

Stačí již maličkost: naprogramovat náš algoritmus v KSP-LISPu tak, aby všechny grafové operace i operace s frontou pracovaly v konstantním čase, a tím pádem celý program běžel v čase $O(m+n)$ (lineární v počtu vrcholů a počtu hran grafu).

Frontu reprezentujeme stejně jako v minulé sérii, tj. jako seznam, u nějž si budeme pamatovat místo pro čtení a pro zápis. Značkování a uchovávání p_x vyřešíme tak, že číslo c_x vrcholu x uvedené na začátku seznamu, kterým je vrchol reprezentován, nahradíme u označkovaných vrcholů dvojicí (c_x, p_x), což můžeme snadno rozlišit od původních čísel pomocí `pair?`. Nesmíme ale zapomenout na konci uvést graf do původního stavu, k tomu nám poslouží zapamatovaný začátek původní fronty – všimněme si, že operace s frontou jsou nedestruktivní, tedy že z ní nikdy prvky neodebíráme, jen si posouváme ukazatel na první nezpracované políčko, takže na konci výpočtu bude seznam reprezentující frontu obsahovat právě všechny označkované vrcholy.

A teď již program:

```
; Přidá všechny sousedy na seznamu next na konec
; fronty tail, označkuje je, nastaví předchůdce na from
; a vrátí nový konec fronty.

(define (add-neigh tail next from)
   (if next
      (let ((v (geta next)))
         (if (pair? (geta v))
            (add-neigh tail (getb next) from)
            (block
               (setb tail (list v))
               (seta v (cons (geta v) from))
               (add-neigh (getb tail)
                          (getb next)
                          from))))
      tail))
```

; Prohledá graf do šířky: *head* je začátek fronty,
; *tail* její konec, *target* vrchol, na kterém se má
; prohledávání zastavit. Vrátí *t* nebo *nil* podle toho,
; zda cesta do *target* existuje.

```
(define (bfs tail head target)
   (if head
      (let ((v (geta head)))
         (if (eq v target)
            't
            (bfs (add-neigh tail (getb v) v)
                 (getb head) target)))
      nil))
```

; Rekonstruuje nejkratší cestu z v do w podle zpětných
; ukazatelů p_w a přidá na její konec seznam l.

```
(define (trace-back w l)
   (if w
      (trace-back (getb (geta w))
                  (cons (geta (geta w)) l))
      l))
```

; Odstraní značky ze všech vrcholů ve frontě *head.*

```
(define (cleanup head)
   (if head
      (block
         (seta (geta head)
               (geta (geta (geta head))))
         (cleanup (getb head)))
      nil))
```

; Hledání cesty osobně: založí frontu obsahující
; počáteční vrchol, prohledá do šířky, zrekonstruuje
; cestu a vyčistí graf.

```
(define (path x y)
   (let ((queue (list x))
         (z (seta x (list (geta x))))
         (result (bfs queue queue y))
         (p (if result
                (trace-back y nil)
                nil)))
      (cleanup queue)
      p))
```

13-5-1 Bláznivé hodiny — Jan Kára

Řešitelé této úlohy se rozdělili na tři kategorie. První a bezkonkurečně největší byla kategorie simulantů. Její příslušníci se rozhodli hodiny prostě simulovat, což je sice funkční, leč značně pomalý (jak si také někteří uvědomovali) přístup. Tomu odpovídalo i bodové hodnocení těchto řešení. Druhá kategorie věštců obsahovala pouze Lukáše Turka, který rychlé řešení uhodl, nicméně již nebyl schopen ho dokázat. Třetí kategorie pamětníků obsahovala dva řešitele, kteří si vzpomněli, že tato úloha již byla kdysi uvedena ve slovenském KSP. Tito řešitelé dodali optimální řešení.

A nyní k samotnému řešení. Budeme postupně počítat tiky způsobené kuličkami s číslem větším nebo rovným i. Pro $i = n$ (n je celkový počet kuliček) je zřejmě počet tiků $t_i = 0$. Když máme počet tiků spočítaný pro i, můžeme ho spočítat pro $i - 1$ podle následující úvahy: Na počátku je pořadí kuliček $\ldots, i-1, i, i+1, i+2, \ldots$. Když se $i - 1$ dostane na první pozici a přesouvá se dozadu, tak přeskočí všechny kuličky s čísly $3 \ldots i - 2$ (ty se totiž mohou vyskytovat pouze na pozicích $1 \ldots i-1$). Protože kuliček s těmito čísly je pouze $i - 4$, přeskočí ještě další tři kuličky s čísly vyššími než $i - 1$. Protože každá kulička s číslem větším než $i - 1$ přeskočí při přesunu dozadu kuličku s číslem $i - 1$, dostane se tato kulička na počátek vždy po třech „počítaných“ ticích. Tedy počet tiků způsobených kuličkami s čísly většími nebo rovnými $i - 1$ je $t_{i-1} = \lfloor t_i/3 \rfloor + 1$. Celkový počet tiků je zřejmě t_3. Sestavit algoritmus využívající výše uvedeného vzorce je již triviální záležitostí.

Správnost algoritmu byla ukázána v popisu, algoritmus bude mít časovou složitost $O(n)$ a paměťovou $O(1)$.

Poznámka pro zvídavé: Výše uvedená tvrzení o složitosti nejsou úplně přesná. Jak si totiž snadno spočtete, hodnota výsledku roste exponenciálně k n (řádově jako $(4/3)^n$). Proto pro velká n již není zcela přijatelná představa, že všechny operace probíhají v konstantním čase a potřebují konstantní paměť – skutečný počet potřebných bitů bude řádově lineární k n a třeba čas na jedno sčítání bude přinejlepším $O(n)$. Celkově tedy složitost pro náš algoritmus v případě času vyjde $O(n^2)$ a v případě paměti $O(n)$. Takovéto detailní úvahy o složitosti po vás samozřejmě ve většině úloh nechceme, spíše je zde uvádíme jako ukázku toho, že ne vždy je u složitosti vše tak zřejmé, jak to na první pohled vypadá...

```
#include <stdio.h>

int main (void)
{
    int n,  ret = 0;

    scanf ("%d",  &n);
    while  (n-- > 3)
      ret += ret/3 + 1;
```

```
    printf ("Tiku bude %d\n", ret);
    return 0;
}
```

13-5-2 Holiči — Dan Kráľ

Naše řešení bude mít dvě části: V první prozkoumáme, kteří trpaslíci určitě nemohou a kteří naopak musí navštívit stejného holiče, ve druhé pak zkonstruujeme optimální rozdělení trpaslíků. Téměř všechna řešení, která jsme obdrželi, se skládala z těchto dvou částí – první z nich nečinila nikomu větší problémy, naopak druhou správně vyřešil jen málokdo. Naše řešení by mělo být pochopitelné i bez znalosti teorie grafů, nicméně pro ty sečtělejší z vás bude řeč teorie grafů na několika místech zmíněna. Trpaslíky lze totiž reprezentovat vrcholy grafu, přičemž dva vrcholy budou spojeny hranou, pokud se tito dva trpaslíci nesnášejí. My pak chceme nalézt nezávislé množiny vrcholů A a B, tak aby se velikosti A a B lišily co nejméně.

Na chvíli si představme, že trpaslík s číslem 1 půjde k prvnímu holiči. Potom všichni trpaslíci, se kterými se nesnáší musí jít k druhému holiči, všichni trplaslíci, kteří nesnášejí některého z trpaslíků, co musí jít k druhému holiči, musí jít k prvnímu holiči atd. Náš program tedy pošle prvního trpaslíka k prvnímu holiči, ty, se kterými se nesnáší, k druhému atd. Pokud nastane situace, že některý trpaslík nemůže jít k ani jednomu z holičů, pak trpaslíky zřejmě nelze k holičům rozdělit. V opačném případě, jsme mohli zjistit, že k trpaslíků musí jít k prvnímu holičí, l k druhému a zbylí mohou jít k libovolnému z holičů, protože vychází dobře se všemi těmito $k + l$ trpaslíky. Na těchto $k + l$ trpaslíků zapomeneme a vybereme ze zbylých trpaslíků jednoho a zopakujeme stejný postup. Takto získáme několik dvojic skupin trpaslíků o velikostech $k_1, k_2, \ldots$ a $l_1, l_2, \ldots$, přičemž trpaslíci z různých skupin v každé této dvojici musí jít k různým holičům, ale na tom, kterého holiče navštíví trpaslíci ze skupin z různých dvojic, nezáleží. V řeči teorie grafů: Rozložíme graf na komponenty souvislosti, zkontrolujeme, že všechny tyto komponenty tvoří bipartitní graf, a nalezneme jejich partity.

Nyní zbývá rozřadit trpaslíky v jednotlivých skupinách k holičům. K tomu použijeme metodu zvanou dynamické programování. *Deficitem* budeme nazývat rozdíl počtu trpaslíků přiřazených prvnímu a druhému holiči. Pro každé i od 0 do počtu dvojic skupin si spočítáme, jakých všech různých deficitů lze dosáhnout (rozdělením prvních i dvojic skupin trpaslíků mezi holiče) za předpokladu, že trpaslíci od $(i + 1)$-ní dvojice dále jsou rozděleni mezi holiče tak, že trpaslíci z první skupiny z dvojice jdou k prvnímu holiči a trpaslíci z druhé skupiny z dvojice jdou k druhému holiči. Hodnoty si budeme udržovat v poli velikosti N (N je počet trpaslíků) – vždy si budeme pamatovat, zda lze daného deficitu dosáhnout a v kladném případě která dvojice skupin je poslední inver-

tována (tj. trpaslíci z první skupiny jdou k druhému holiči a naopak). Pro $i = 0$ je inicializace pole triviální. V i-té iteraci pole projdeme a ke všem hodnotám, které obsahuje, přičteme dvojnásobek rozdílu počtu trpaslíků v druhé a první skupině z i-té dvojice – tak získáme nový možný deficit a pokud již není v poli uložen, tak ho tam uložíme.

Správnost algoritmu je zřejmá z popisu. Program je pouhým přepisem výše popsaného postupu. Časová složitost algoritmu je $O(N^2)$ a paměťová je $O(N + M)$, kde N je počet trpaslíků a M je počet dvojic nesnášejících se trpaslíků. Paměťovou složitost by bylo možné zlepšit na $O(N)$ lepší implementací první části algoritmu (vstup bychom zpracovávali rovnou při načítání), nicméně toto zlepšení zde nebudeme popisovat.

```
program trpaslici;
const MAXN=100;
var nesnasi_kolik: array[1..MAXN] of word;
      { Kolik trpaslíků nesnáší i-tý trpaslík? }
    nesnasi_koho: array[1..MAXN,1..MAXN] of word;
      { Které trpaslíky nesnáší i-tý trpaslík? }
    skupina: array[1..MAXN] of word;
      { Do které skupiny patří i-tý trpaslík? }
    skupin_rozdily: array[1..MAXN] of integer;
      { Velikosti rozdílů velikostí skupin v jednotlivých dvojicích }
    deficit: array[-MAXN..MAXN] of integer;
      { Kterých deficitů lze dosáhnout?
        = -1 ... nelze
        =  0 ... lze dosáhnout bez invertování
        >  0 ... pořadí poslední invertované dvojice skupin trpaslíků }
    holic: array[1..MAXN] of byte;
      { Ke kterému holiči půjde i-tý trpaslík? }
    N: word;
      { Počet trpaslíků }
    L: word;
      { Počet dvojic skupin }
procedure nacti_vstup;
  var M:word;
      i,j,k:word;
  begin
    readln(N,M);
    for k:=1 to N do nesnasi_kolik[k]:=0;
    for k:=1 to M do
      begin
        readln(i,j);
        inc(nesnasi_kolik[i]); nesnasi_koho[i][nesnasi_kolik[i]]:=j;
        inc(nesnasi_kolik[j]); nesnasi_koho[j][nesnasi_kolik[j]]:=i;
      end
  end;
function urci_skupiny: boolean;
  { Vrací false, pokud rozdělení neexistuje. }
  function zarad(trpaslik, do_skupiny: word; k_holici: byte): boolean;
    { Vrací false, pokud rozdělení neexistuje. }
    var i: word;
    begin
      skupina[trpaslik]:=do_skupiny;
      holic[trpaslik]:=k_holici;
```

```
      zarad:=false; { Default }
      for i:=1 to nesnasi_kolik[trpaslik] do
        begin
          if skupina[nesnasi_koho[trpaslik][i]]=0 then
            if not zarad(nesnasi_koho[trpaslik][i],do_skupiny,3-k_holici) then
              exit;
          if skupina[trpaslik]<>skupina[nesnasi_koho[trpaslik][i]] then
            halt(1); { Chyba v programu ;( }
          if holic[trpaslik]=holic[nesnasi_koho[trpaslik][i]] then
            exit; { Nelze rozdělit }
        end;
      zarad:=true
    end;
  var k :word;
  begin
    for k:=1 to N do skupina[k]:=0;
    L:=0; { Číslo poslední vytvořené skupiny }
    urci_skupiny:=false; { Default }
    for k:=1 to N do
      if skupina[k] = 0 then
        begin
          inc(L);
          if not zarad(k,L,1) then exit;
        end;
    urci_skupiny:=true;
    for k:=1 to L do skupin_rozdily[k]:=0;
    for k:=1 to N do
      if holic[k]=1 then
        inc(skupin_rozdily[skupina[k]])
      else
        dec(skupin_rozdily[skupina[k]])
  end;
procedure rozdel_trpasliky;
  var i,j:integer;
  begin
    for i:=-N to N do deficit[i]:=-1;
    j:=0;
    for i:=1 to L do j:=j+skupin_rozdily[i];
    deficit[j]:=0;
    for i:=1 to L do
      for j:=-N to N do
        if (deficit[j]<>-1) and (deficit[j]<>i) and
           (deficit[j-2*skupin_rozdily[i]]=-1) then
          deficit[j-2*skupin_rozdily[i]]:=i;
    j:=0;
    for i:=N downto 0 do
      begin
        if deficit[i]<>-1 then j:=i;
        if deficit[-i]<>-1 then j:=-i;
      end;
    while deficit[j]<>0 do
      begin
        for i:=1 to N do
          if skupina[i]=deficit[j] then
            holic[i]:=3-holic[i];
        j:=j+2*skupin_rozdily[deficit[j]];
      end;
  end;
```

```
procedure vypis_vystup;
  var i:word;
  begin
    writeln('K prvnímu holiči:');
    for i:=1 to N do if holic[i]=1 then write(i,' ');
    writeln;
    writeln('K druhému holiči:');
    for i:=1 to N do if holic[i]=2 then write(i,' ');
    writeln;
  end;
begin
  nacti_vstup;
  if not urci_skupiny then
    writeln('Trpaslíky nelze rozdělit mezi holiče!');
  rozdel_trpasliky;
  vypis_vystup;
end.
```

13-5-3 Optimální strom — Tomáš Vyskočil

Většina z vás tento problém řešila pomocí jednoduchého algoritmu používajícího haldu. Ten má ale složitost $O(N \log N)$ a naši úlohu šlo řešit i lépe. Optimálních řešení bylo opravdu pomálu (přesněji dvě), a to s časovou složitostí $O(N)$.

A teď již k popisu algoritmu. Na vstupu podle zadání dostaneme setříděnou posloupnost délek. V každém kroku se rozhodneme sloučit dvě nejkratší posloupnosti. Jejich délky tedy můžeme z posloupnosti vyřadit a místo nich do posloupnosti vložit součet jejich délek. Abychom mohli rychle nacházet nejmenší délky, potřebujeme posloupnost udržovat setříděnou. Na to použijeme drobný trik – přidáme si pomocnou frontu, do které budeme vždy na konec ukládat právě vzniklé součty. Tato fronta bude vždy utříděna, protože minulé součty, které do této fronty byly již přidány, byly součty dvou v té době nejmenších čísel, a tedy i jejich součet bude nejmenší. Když pak hledáme dvě nejmenší čísla z posloupnosti, tak budeme hledat jak v číslech z původní posloupnosti, tak v nově vytvořené frontě součtů. A to je vše. Algoritmus má časovou i paměťovou složitost $O(N)$.

Nyní ještě důkaz správnosti algoritmu. Postup slévání si můžeme představit jako binární zakořeněný strom. V jeho listech budou délky vstupních posloupností, u vnitřního vrcholu bude délka posloupnosti vzniklé slitím dvou posloupností odpovídajících synům vrcholu. Pokud máme vstupní posloupnost délky d v n-té hladině, potom nám tato posloupnost přidá do celkového počtu porovnání $n \cdot d$. Nadále zkoumejme strom s nejmenším počtem porovnání. V nejhlubší hladině určitě musí mít slity dvě nejkratší posloupnosti (a ty my přesně navrhujeme slévat nejdříve), protože pokud bychom se na počátku rozhodli sloučit jiné než dvě nejkratší posloupnosti, byla by v nejhlubší (n-té) hladině posloupnost s délkou větší než d a jak si snadno spočtete, celkový počet porovnání by se zvýšil. V dalších krocích jsme pak vlastně v přesně stejné

situaci jako v prvním kroku – stačí, když zapomeneme na listy odpovídající sloučeným posloupnostem. Tím je správnost tohoto algoritmu dokázána.

13-5-4 Generátor — Aleš Přívětivý

Úloha byla jednoduchá a většina z vás si s ní hravě poradila. Stačí si uvědomit fakt, že pokud se nějaké číslo v posloupnosti vyskytuje alespoň dvakrát, pak se v ní vyskytuje alespoň dvakrát i poslední číslo A_N. Nechť A_i je první takové číslo, které se v posloupnosti vyskytuje vícekrát a A_{i+k} je jeho druhý výskyt. Pak zřejmě platí i $A_i = A_{i+k} = A_{i+2k} = ...$, a tedy i pro $j \geq i$ platí $A_j = A_{j+k} = A_{j+2k} = ...$ Položme $j = N - k$, vzhledem k $i + k \leq N$ platí $j \geq i$, a tedy $A_{N-k} = A_N$.

Nyní už je algoritmus nasnadě. Zjistíme si A_N a pak porovnáním s prvky $A_1, ...A_{N-1}$ zjistíme, zda se s některým z těchto čísel neshoduje. Pokud ano, generátor je vadný, pokud ne, generátor je v pořádku. To vše zvládneme otestovat s konstantní paměťovou složitostí (což byla podmínka zadání) a budeme k tomu potřebovat $2N-3$ dotazů na následující náhodné číslo. Vzhledem k tomu, že posloupnost si musíme alespoň jednou projít celou, to asymptoticky lépe nejde.

```
#include <stdio.h>

int N, i, aN, ai;

int next (int number)
{
        printf ("Zadej cislo po cislu %d:", number);
        scanf ("%d", &number);
        return number;
}

int main (void)
{
        printf ("Zadej N:␣");
        scanf ("%d", &N);

        for (i=0, aN=1; i<N-1; i++) aN=next (aN);

        for (i=0, ai=1; ai!=aN && i<N-2; i++) ai=next (ai);

        if (i==1||i==N-2) printf ("Generator funguje.\n");
            else printf ("Generator nefunguje.\n");

        return 0;
}
```

13-5-5 ObjectLISP — Martin Mareš

Tato úloha byla, pravda, poněkud zběsilá, ale nebojte se, řešení nezůstane pozadu. Před startem se raději ještě jednou podívejte do popisu KSP-LISPu, jak funguje *eval* a *quote* a už se držte, jedeme z kopceeeeee!

Každý objekt bude ve skutečnosti jednoduchá speciální forma, která neudělá nic jiného než že zavolá funkci *objcall* a předá jí jednak, co se přesně má s objektem udělat (to jest jméno metody a seznam všech jejích parametrů) a jednak seznam metod tohoto objektu (unikátní pro každý objekt, bude se totiž při přidávání nových metod modifikovat). Funkce *object* tedy pouze vytvoří takovou speciální formu a připojí k ní seznam metod, který bude buďto kopií seznamu metod předka, dědíme-li, nebo kopií seznamu systémových metod:

```
(define (object &rest parent)
  (list
    'special
    '(op &rest args)
    (list 'objcall
          (list quote
            (copy-list
               (if parent
                  ((geta parent) get-attrs)
                  default-attrs)))
          'op
          'args)))
(define (copy-list l)
  (if l
    (cons (geta l) (copy-list (getb l)))
    nil))
```

Dále si nadefinujeme seznam systémových metod, všechny z nich budou ukazovat na funkce téhož jména s hvězdičkou na začátku:

```
(set 'default-attrs (list
  (cons 'get-attrs   *get-attrs)
  (cons 'get-method  *get-method)
  (cons 'def-method  *def-method)
  (cons 'def-attr    *def-attr)
))
```

Pokud chceme zavolat metodu, stačí pomocí *get-method* vyhledat, která funkce této metodě odpovídá, a zavolat ji s patřičnými parametry; jako první parametr opět předáváme seznam metod našeho objektu.

```
(define (objcall obj op args)
  (eval
    (cons
      (list quote (*get-method obj op))
      (cons 'obj args))))
```

Metoda *get-attrs* je ze všech nejjednodušší: má vrátit seznam všech metod a atributů, a to je přesně její argument:

```
(define (*get-attrs obj)
   obj)
```

Metoda *get-method* vyhledá v seznamu metodu daného jména a vrátí její definici, pokud žádná taková neexistuje, vrátí odkaz na chybovou metodu *undefined-method*:

```
(define (*get-method obj op)
  (if obj
    (if (eq (geta (geta obj)) op)
      (getb (geta obj))
      (*get-method (getb obj) op))
    *undefined-method))
(define (*undefined-method obj &rest ignored)
  (print 'Undefined 'method 'called))
```

Definice nové metody je také poměrně snadná: stačí do seznamu, který jsme dostali jako parametr, připsat novou položku, případně pokud již metoda tohoto jména existovala, tak její položku přepsat. K tomu nám dopomůže, že víme, že seznam je zaručeně neprázdný, protože vždy obsahuje alespoň systémové metody.

```
(define (*def-method obj op val)
  (if (eq (geta (geta obj)) op)
    (cons (cons op val) (getb obj))
    (setb obj
      (if (getb obj)
        (*def-method (getb obj) op val)
        (cons (cons op val) nil)))))
```

A nakonec přidání nového atributu: každý atribut je vlastně také metoda, která si pamatuje nějakou konstantu a pokud je zavolána bez parametru, vrátí tuto konstantu, pokud s parametrem, pak předefinuje sebe sama na verzi s konstantou nahrazenou tímto parametrem (to je jednodušší než definici funkce nějak přetvářet):

```
(define (*def-attr obj attr val)
  (*def-method obj attr
    (list 'lambda '(obj &rest x)
      (list if 'x
        (list '*def-attr 'obj
            (list quote attr)
            (list geta 'x))
```

```
                (list quote val))))
          val)
```

Tak, a střecha nad náš objektový domeček právě dosedla, ještě pročistit komín a namontovat anténu pro připojení k Internetu `:-)` Pěkné prázdniny a `(see you (+ 1 this-year))`.

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Miloslav Trmač	Biskupské G, Brno	4	24	260
2.	Lukáš Turek	G Zborovská, Praha	2	24	228
3.	Peter Bella	G Jura Hronca, Bratislava	3	21	179
4.	Jiří Štěpánek	G Tř. kpt. Jaroše, Brno	2	23	160
5.	Ondřej Zajíček	SPŠ strojnická, Chrudim	4	17	156
6.	Martin Hamrle	G Pelhřimov	3	18	137
7.	Martin Lopatář	G Tř. kpt. Jaroše, Brno	1	23	136
8.	Jozef Tvarožek	G Jura Hronca, Bratislava	3	15	134
9.	Jiří Fink	SPŠ Ječná, Praha	4	16	131
10.	Jiří Paleček	G Kladno	2	17	116
11.	Ondřej Hrabal	SPŠ Uherské Hradiště	4	14	107
12.	Martin Zlomek	Purkyňovo G, Strážnice	4	12	104
13.	Alexandr Kazda	G Nad alejí, Praha	1	18	103
14.	David Matoušek	G Arcus, Praha	1	18	98
15.–16.	Marek Ludha	G Bánská Bystrica	1	16	89
	Zuzana Vlčková	G Alejová, Košice	4	18	89
17.	Marek Sterzik	SPŠ Ostrov	2	9	87
18.–19.	Jaroslav Havlín	G Sedlčany	1	16	82
	Milan Straka	G Strakonice	2	11	82
20.	Miroslav Rudišin	G Šrobárova, Košice	4	9	76
21.	Martin Macko	G Spišská Nová Ves	4	9	75
22.	Jiří Svoboda	G Zborovská, Praha	3	6	72
23.	Jiří Šofka	G Sedlčany	4	13	71
24.	Jakub Galognek	G P. Bezruče, Frýdek-Místek	3	11	67
25.	Pavel Bazika	G Nad Kavalírkou, Praha	3	10	66
26.	Pavel Čížek	G Kralupy nad Vltavou	2	10	61
27.	Jan Matějek	G Kladno	1	7	59
28.	Josef Hala	G Uherské Hradiště	4	7	58
29.	Dávid Haraga	G Bánská Bystrica	4	8	55
30.	Pavel Celba	G Úpice	4	7	54
31.	Pavel Srb	G Karlovy Vary	1	11	51
32.	Tomáš Záležák	G M. Lercha, Brno	2	10	46
33.	Martin Dobroucký	G Moravská Třebová	0	10	44
34.	Roman Krejčík	G Zborovská, Praha	4	5	41
35.	Ján Oravec	G Bánská Bystrica	4	5	40
36.	Pavel Kůs	G Zborovská, Praha	4	5	39
37.	Michal Lichvár	G Trenčín	3	6	37

Pořadí řešitelů

38.	Boris Burdiliak	G Jura Hronca, Bratislava	4	5	36
39.	Radovan Bauer	G Poštová, Košice	3	4	35
40.–41.	Jiří Koula	G U Libeňského zámku, Praha	4	3	29
	Marek Sulovský	G Tř. kpt. Jaroše, Brno	4	5	29
42.	Peter Šufliarsky	G Nové Zámky	1	5	28
43.	Peter Krčah	G Nitra	4	5	27
44.	Martin Maňák	Purkyňovo G, Strážnice	3	5	26
45.–46.	Jiří Plachý	G Uherské Hradiště	4	4	25
	Michal Pokorný	G Grosslingová, Bratislava	4	4	25
47.–48.	Matej Dubový	G Trenčín	3	4	23
	Petr Los	G Hranice	2	6	23
49.	Pavel Troubil	G Tř. kpt. Jaroše, Brno	1	4	20
50.	Jiří Musil	Havlíčkovo G, Havl. Brod	3	4	17
51.–52.	Stanislav Hotmar	?	4	7	14
	Pavel Košař	G T. G. M., Frýdek-Místek	4	3	14
53.	Cyril Strejc	G B. Balbína, Hradec Králové	2	2	12
54.–55.	Marián Dvorský	G Šrobárova, Košice	4	1	10
	Michal Rjaško	G Vranov	2	3	10
56.	Aleš Kučík	G Trutnov	4	2	6
57.–58.	Jakub Mareček	G Vídeňská, Brno	3	1	4
	Ondrej Sňahničan	G Púchov	2	1	4
59.	Jaromír Šimek	G Kojetín	2	2	3
60.–61.	Pavel Kolář	SPŠ stojnická, Chrudim	4	1	1
	Jiří Koudelka	SPŠ Jihlava	2	1	1
62.–75.	Pavel Baránek	G J. Opletala, Litovel	-1	0	0
	Jiří Bláha	G Český Krumlov	2	0	0
	Jana Cimrmanová	OA Vlašim	4	0	0
	Otakar Dokoupil	G Přerov	3	0	0
	Štěpán Dušek	G Na vítěz. pláni, Praha	3	0	0
	Jiří Janák	G Frenštát p. Radhoštěm	4	0	0
	Vít Jedlička	G B. Balbína, Hradec Králové	2	0	0
	Jakub Kotowski	G Trutnov	4	0	0
	Marek Kyselý	G Kojetín	2	0	0
	Mikoláš Panský	?	?	0	0
	Oto Petřík	G Vrchlabí	0	0	0
	Radka Picková	G Mladá Boleslav	2	0	0
	Alexej Sidorenko	SPŠ Holešov	3	0	0
	Václav Zajíc	G Kralupy nad Vltavou	2	0	0

Experti v Linuxu

Společnost SuSE Linux AG, se sídlem v Norimberku, je jedním z předních světových dodavatelů komplexních řešení na základě open source operačního systému Linux. Vedle operačního systému a aplikací pro soukromé osoby nabízí SuSE Linux AG softwarová řešení a komplexní systémy pro instalaci Linuxu v podnicích. Svým obchodním partnerům nabízí SuSE Linux AG širokou škálu kvalifikovaných poradenských a školících služeb včetně podpory na všech úrovních. Společnost zaměstnává celosvětově největší tým vývojářů pro řešení s otevřeným zdrojovým kódem, jejichž jedinečné projektové a instalační know-how zveřejnila v nejobsáhlejší linuxové databázi na Internetu. V současné době čítá SuSE Linux AG přes 500 zaměstnanců a má pobočky v šesti zemích světa.

V České republice je SuSE Linux AG zastoupena dceřinou společností SuSE CR, s.r.o., která nabízí komplexní linuxová řešení, konzultační služby, školení, vývoj, instalační podporu, distribuci produktů SuSE a dalších linuxových aplikací.

SuSE značně přispívá k linuxovým vývojovým projektům jako Linux kernel, glibc, XFree86^{TM}, KDE, ISDN4Linux, ALSA (Advanced Linux Sound Architecture) a USB (Universal Serial Bus).

Více informací o celé společnosti získáte na *http://www.suse.cz/*.

Od roku 2000 je společnost SuSE CR, s.r.o. sponzorem MFF.

Obsah

Úvod 5
Zadání úloh 6
  První série 6
  Druhá série 14
  Třetí série 16
  Čtvrtá série 19
  Pátá série 22
Vzorová řešení 25
  První série 25
  Druhá série 40
  Třetí série 48
  Čtvrtá série 58
  Pátá série 69
Pořadí řešitelů 78
Reklama 80
Obsah 81

Martin Mareš a kolektiv

Korespondenční seminář z programování XIII. ročník

Autoři a opravující úloh:
Martin Mareš, Zdeněk Dvořák, Pavel Machek,
Pavel Nejedlý, Jan Kára, Robert Špalek,
Daniel Kráľ, Aleš Přívětivý, Pavel Šanda,
Tomáš Vyskočil, Jakub Bystroň

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 2001

Písmem Computer Modern v programu TEX vysázel Martin Mareš

Korektury provedla Karolína Šimková

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

84 stran, 4 obrázky
Vydání první
Náklad 300 výtisků