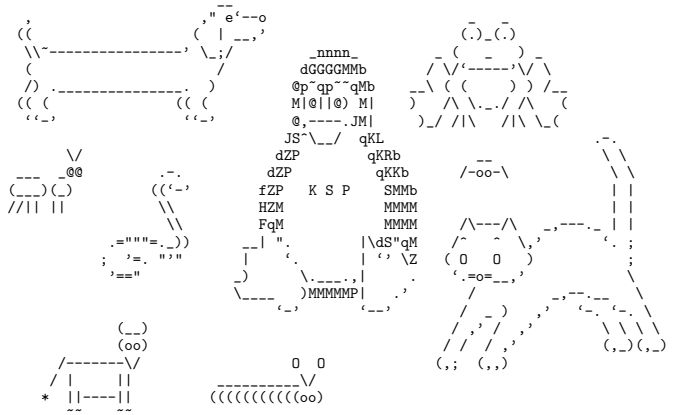
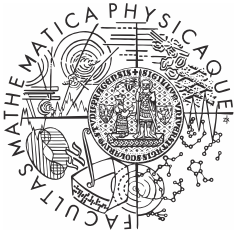
MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XV. ročník – 2002/2003





Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MARTIN MAREŠ A KOLEKTIV

Korespondenční seminář z programování

XV. ročník – 2002/2003

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož patnáctý 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://ksp.mff.cuni.cz/*

Zadání úloh

15-1-1 Narozeninový dort — 11 bodů

Manželé Terciovi měli trojčata – Alfíka, Betynku a Gamíka. Narozeniny byly pro rodiče vždy velmi náročné. Přeci jen shánět dárky hned pro tři děti naráz dá dost práce, a tak se rodiče jednoho roku rozhodli, že si alespoň trochu ulehčí práci a dají dětem dohromady jeden dort. Dort to nebyl ledajaký – měl tvar rovnostranného trojúhelníka a na dortu bylo tolik svíček, kolik měly děti dohromady let. Problém ale nastal, když se dort měl dělit. Každé dítě totiž chtělo mít tolik svíček, kolik mu právem náleží, a svíčky na dortu přitom byly rozmístěny náhodně. Tatínek se pokoušel dort rozdělit, ale brzy zjistil, že splnit požadavek dětí není tak jednoduché, a tak o pomoc požádal vás.

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu délku hrany dortu d (rohy dortu mají souřadice $[0,0]$, $[d,0]$, $[d/2, d \cdot \sqrt{3}/2]$), počet svíček na dortu N (pochopitelně dělitelný třemi) a souřadnice jednotlivých svíček a nalezne na dortu bod takový, že řezy z jednotlivých rohů do nalezeného bodu rozdělí dort na tři části obsahující stejný počet svíček.

Příklad: Pro dort o délce hrany 10 a souřadnicích 3 svíček $(1,1)$, $(9,1)$ a $(5,1)$ je hledaným bodem například bod o souřadnicích $(5,3)$.

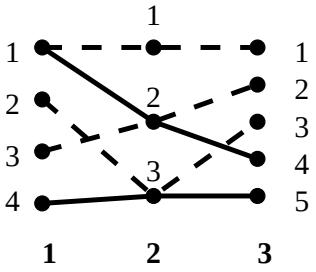
15-1-2 Vláčky — 11 bodů

Vlastík měl doma postaveno velké kolejiště. Na kolejišti vedlo vedle sebe několik kolejí, které se mezi sebou složitě proplétaly (Vlastík při stavbě nešetřil ani nadjezdy a tunely), spojovaly se a zase se dělily. Jednoho dne Vlastíka napadlo spočítat, kolik vlaků může na kolejiště vypustit současně. Po chvilce zjistil, že problém to není vůbec jednoduchý, a tak se rozhodl si při řešení úlohy pomoci programem. Pomůžete mu s ním?

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu popis kolejiště a na výstup vypíše maximální možný počet tras, které se navzájem neprotínají (tzn. nemají žádný společný bod), spolu s jejich popisem. Kolejiště se skládá z N kontrolních bodů. i-tým kontrolním bodem vede k_i kolejí. Dále je dán seznam kolejí mezi jednotlivými kontrolními body – j-té propojení mezi dvěma kolejemi v kontrolních bodech je popsáno dvěma dvojicemi (a_j, b_j), (c_j, d_j), kde a_j a c_j jsou čísla kontrolních bodů, mezi kterými kolej vede (předpokládejte, že $c_j = a_j + 1$) a b_j a d_j jsou čísla kolejí v jednotlivých kontrolních bodech. Protože v kontrolních bodech se mohou vyskytovat výhybky, může být na jednu kolej v kontrolním bodě napojeno hned několik kolejí v předcházejícím a následujícím kontrolním bodě (ale nemusí být napoje-

na kolej žádná). Předpokládejte, že koleje se stýkají či dělí pouze v kontrolních bodech a mezi body se koleje neprotínají (to ovšem neznamená, že jedna kolej nepřejíždí druhou po nadjezdu).

Příklad: Pro kolejiště se třemi kontrolními stanovištěmi se 4, 3 a 5 body a s propojeními $(1,1) \to (2,1)$, $(1,1) \to (2,2)$, $(1,2) \to (2,3)$, $(1,3) \to (2,2)$, $(1,4) \to (2,3)$, $(2,1) \to (3,1)$, $(2,2) \to (3,2)$, $(2,2) \to (3,4)$, $(2,3) \to (3,3)$ a $(2,3) \to (3,5)$ existují tři neprotínající se trasy. Jsou to například $(1,1) \to (2,1) \to (3,1)$, $(1,2) \to (2,3) \to (3,3)$ a $(1,3) \to (2,2) \to (3,2)$.



Kolejiště s vyznačenými trasami

15-1-3 Kulový blesk — 10 bodů

Stěhovací firma Popruh a spol. se rozhodla uspořádat propagační akci Kulový blesk. Akce spočívá v tom, že N rodin si vymění byty takovým způsobem, že pokud rodina bydlela v bytě i, tak po stěhování bude bydlet v bytě $i+1$ (pokud rodina bydlela v bytě N, tak bude po stěhování bydlet v bytě 1). Zorganizovat takovou směnu je samozřejmě velmi náročné, a tak se firma rozhodla ji trochu zjednodušit. Výměna bude probíhat tak, že se dohodnou vždy dvojice rodin a ty si vymění byty (současně může probíhat libovolně takovýchto výměn najednou, ale každá rodina se v daném okamžiku může účastnit nejvýše jedné směny). Tento způsob má ale tu nevýhodu, že některé rodiny se budou muset stěhovat vícekrát, než se dostanou do kýženého cílového bytu. Protože celou výměnu by bylo vhodné provést v jeden den, chce pan Popruh v rámci zachování dobrého jména firmy minimalizovat dobu nutnou na stěhování. Nakonec se rozhodl obrátit na vás, abyste mu se svou znalostí programování pomohli naplánovat výměny bytů.

Váš program dostane na vstupu počet bytů N. Stěhování, které má program navrhnout, má probíhat tak, že v jednom okamžiku si vždy nějaké dvojice rodin vymění byty (jeden takovýto okamžik záměny nazveme „fází“). Váš program by tedy měl vypsat počet fází a pro každou fázi seznam dvojic bytů, které jsou v této fázi směněny. Navržené stěhování by mělo mít nejmenší možný počet fází.

Příklad: Pro čtyři rodiny lze stěhování provést na dvě fáze. V první fázi se vymění rodiny v bytech $(1, 2)$ a $(3, 4)$ a ve druhé fázi rodiny v bytech $(1, 3)$.

15-1-4 Soustavy — 8 bodů

Vaším úkolem v této úloze je navrhnout algoritmus a napsat program, který dostane na vstupu tři čísla (či spíše řetězce složené ze znaků `0` až `9` a `a` až `z`) A, B a C a určí základ soustavy, ve které platí rovnice $A + B = C$. Pokud základ soustavy není jednoznačně určen, vypište soustavu s nejmenším základem, která rovnici vyhovuje.

Příklad: Pro $A =$`1`, $B =$`1a` a $C =$ `20` je základ soustavy 11.

15-1-5 Haskell — 10 bodů

V letošní sérii se budeme zabývat funkcionálním programováním. Cílem je zejména vyvrátit obecně rozšířenou představu, že funkcionální programování = LISP = umění správně spočítat závorky. Pro účely výkladu (a také úloh pro vás) budeme používat programovací jazyk Haskell. Jedná se o poměrně komplikovaný jazyk a není v našich možnostech probrat ho celý; v následujícím textu si vyložíme alespoň jeho základy.

Pro řešení semináře by vám měl postačovat interpreter Hugs, který najdete na adrese *http://www.haskell.org/hugs/*. Pokud byste chtěli v Haskellu sepsat nějaký seriózní program, existuje kompilátor GHC, bližší informace viz *http://www.haskell.org/ghc/*. Kompletní specifikaci jazyka si můžete přečíst například na *http://www.haskell.org/definition/*. Jsme ovšem v pokušení vypsat prémiové body za její pochopení.

Haskell je čistě funkcionální typovaný líně vyhodnocovaný jazyk. Co to všechno znamená?

- „Čistě funkcionální“ znamená, že celý program je skutečná funkce; tedy že nemají žádné vedlejší efekty (za vedlejší efekty považujeme například vstupy a výstupy, zápisy do globálních proměnných a podobně – tedy všechno to, co způsobuje, že `f(x) + f(x)` se nemusí chovat stejně jako `2*f(x)`). To vypadá dost podivně – k čemu by byl programovací jazyk, ve kterém nejde například cokoliv vypsat na obrazovku? Jak je toto vyřešeno, si povíme v některé z následujících sérií. Výhoda tohoto přístupu je v tom, že je mnohem snazší uvažovat o tom, co vlastně program dělá – víme, že to nemůže být ovlivněno ničím jiným, než parametry funkce. Navíc to překladači umožňuje provádět optimalizace, které by v jiných jazycích byly neproveditelné bez obtížné meziprocedurální analýzy.
- „Typovaný“ znamená, že všechny výrazy a funkce mají přesně daný typ, který se kontroluje v době překladu. To má své výhody (odchytí

se tím mnoho chyb a není potřeba kontrolovat typy za běhu programu – nemůže se nám stát, že bychom se omylem pokusili sečíst číslo a řetězec), ale i své nedostatky (v čistém Haskellu nelze právě z důvodu typování vytvořit heterogenní seznam – tj. takový, který by obsahoval například řetězce a čísla zároveň). Typový systém Haskellu je velmi flexibilní, jak uvidíme dále, ale je i velmi bezpečný vzhledem k tomu, že neexistuje přetypování (dokonce nelze přímo provést ani takovou „samozřejmost“, jako sečíst reálné a celé číslo).

- „Líný“ se vztahuje k tomu, že vyhodnocovány jsou jen ty výrazy, které jsou potřeba, a to až tehdy, když jsou potřeba (dá se říci, že všechny funkce v Haskellu se chovají jako special formy v LISPu). Mějme například následující definici (definující funkci `const` se dvěma argumenty, která vrací první z nich – dá se to ovšem také chápat tak, že je to funkce s jedním argumentem `x`, která vrací konstantní funkci (s jedním argumentem `y`, který ignoruje), jejíž hodnota je vždy `x`):

  ```
  const x y = x
  ```

  Pak `const 5 (error "Chyba")` vrátí `5` – druhý argument, který by ukončil program s chybou, se vůbec neprovede! Dále to znamená, že můžeme při výpočtu používat jeho výsledek, pokud tím nedojde k „zacyklení“ – viz následující příklad:

  ```
  natural = 1 : map (+1) natural
  ```

  Toto je definice posloupnosti 1, 2, 3, ... – její první člen je 1 a za ním následuje tatáž posloupnost zvětšená o 1. Tento program skutečně funguje, neboť první člen posloupnosti známe, a abychom zjistili hodnotu n-tého členu posloupnosti, potřebujeme znát pouze hodnotu $n-1$. členu.

Nyní se podívejme podrobněji na systém typů. Asi nejjednodušší bude začít příkladem:

```
data List x = Cons x (List x) |
              Nil
```

Tato definice deklaruje typ `List x`, což je seznam prvků typu `x`. Seznam je tedy buď prázdný (konstanta `Nil`), nebo je to dvojice, jejíž první prvek je typu `x` (tj. první prvek seznamu) a druhý prvek je seznam prvků typu `x` (zbytek seznamu). Například seznam 1, 2, 3 je vyjádřen jako `Cons 1 (Cons 2 (Cons 3 Nil))`. `Cons` a `Nil` jsou konstruktory; tedy funkce, které vytváří hodnoty daného typu. `Cons` má za argumenty hlavu a tělo seznamu, zatímco `Nil` je funkce bez argumentů (tedy konstanta).

Věci, které stojí za povšimnutí:

- Jména typů (v našem případě `List`) a konstruktorů (`Cons` a `Nil`) začínají velkým písmenem.

- Jména typů a konstruktorů nemusí být různá (nemůže dojít k záměně, protože typy se nemohou nikdy vyskytovat ve výrazech). To se často využívá, zejména tehdy, má-li typ jen jeden konstruktor.
- Typy mohou být parametrizovány. V našem případě jsme nadefinovali typ `List x`, což je seznam prvků typu `x`. Za `x` lze dosadit libovolný typ. Například uvedený seznam čísel by měl typ `List Int`. Lze ovšem také vytvořit funkce, které budou pracovat nad takto obecným typem (například u funkce vracející délku seznamu je jedno, jaký typ mají prvky tohoto seznamu).
- Typy mohou být rekurzivní – v definici seznamu jsme použili právě definovaný typ.

Haskell má některé základní typy:

- `Int` je typ pro celá čísla; konstanty tohoto typu se zapisují běžným způsobem (čísla v desítkové soustavě).
- `Char` je typ pro znaky; znakové konstanty se uváději v apostrofech: `'a'`.
- `Bool` je typ pro logické hodnoty; může nabývat hodnot `True` nebo `False`.
- `()` je jednotkový typ – obsahuje právě jen hodnotu `()` (používá se tehdy, jestliže chceme například deklarovat funkci, která nic nevrací (tedy proceduru))
- `(a,b)` je typ pro dvojici (jejíž první prvek má typ `a` a druhý `b`). Analogicky `(a,b,c)` pro trojice, atd. `(5,'a')` je příklad konstanty typu `(Int,Char)`.
- `[a]` je typ pro seznam prvků typu `a`. Jeho definice přesně odpovídá výše uvedenému typu `List`, pouze místo `Nil` se používá `[]` a místo `Cons` dvojtečka. Dále lze použít i seznam prvků v hranatých závorkách (tedy seznam 1, 2, 3 lze zapsat jako `1:2:3:[]` nebo jako `[1,2,3]`)
- `String` je typ pro řetězce (je identický s `[Char]`). Konstanty tohoto typu lze uvádět v uvozovkách (tedy `"abc"` = `['a','b','c']` = `'a':'b':'c':[]`).

Zbývá ukázat, jak vypadají typy funkcí. Funkce, která má jeden parametr typu `a` a vrací hodnotu typu `b`, má typ `a->b`. Logická otázka je, co s funkcemi více parametrů. Odpověd je, že si vystačíme s funkcemi s jedním argumentem. Například `plus` má typ `Int->(Int->Int)` – je to tedy funkce, která vrací funkci, která vrací `Int`. `plus 5` je tedy funkce typu `Int->Int` (která vrací argument zvětšený o 5), a `(plus 5) 10` je 15. Typy funkcí se závorkují vždy zprava, zatímco jejich aplikace zleva – tedy závorky v uvedeném příkladu jsou zbytečné, `plus` má typ `Int->Int->Int` a pro jeho vyhodnocení stačí napsat `plus 5 10`.

Pozor – funkce jsou hodnoty jako každé jiné (je možné je dávat jako parametry jiným funkcím, je možné mít typ `[a->b]` – seznam funkcí, atd.)

Typy funkcí není potřeba uvádět – překladač si je domyslí. Je ovšem rozumné to dělat (snáze se pak odhalí chyby). Deklarace typu funkce vypadá takto:

```
length :: [a]->Int
```

Jak jsme již viděli na příkladech, program v Haskellu je množina rovností definujících funkce. Na konci zadání si můžete prohlédnou rozsáhlejší příklad demonstrující podmnožinu Haskellu nutnou pro rozumnou práci.

Několik poznámek k programům:

- `--` začínají komentář. Ten končí na konci řádky.
- Jména funkcí a parametrů začínají malým písmenem. Jejich součástí může být apostrof; pozor, pojmenovat si cokoliv `x''''''''` nezvýší čitelnost programu.
- Aplikace funkcí mají vyšší prioritu než operátory; tj. `delka t + 1` se překládá jako `(delka t) + 1`.
- Klauzule `let` a `where` nám umožňují zavádět lokální definice; rozdíl mezi nimi je ten, že `let něco in výsledek` je výraz, zatímco `where` lze uvést pouze za definicí funkce (tohle není tak úplně pravda, ale pro naše účely to stačí). Lze mít víc lokálních definic uvnitř jednoho takového příkazu (u `where` je to vidět i na příkladu), tyto definice mohou být rekurzivní.
- K „rozkládání“ strukturovaných typů je možné použít tzv. „pattern matching“; tj. všude tam, kde se něco přiřazuje (uvnitř `let` nebo `where`, u parametrů funkcí, ...) lze uvést „vzor“ toho, co si představujeme, že dostaneme. Uvnitř tohoto vzoru můžeme mít proměnné, jim jsou pak přiřazeny hodnoty. Pokud to, co přiřazujeme, neodpovídá vzoru, u definice funkce se přechází na následující alternativní definici (je-li nějaká taková), jinak dojde k chybě. Pro znalce Prologu: pozor, nejedná se o unifikaci; pokud ve vzoru použijeme vícekrát jednu proměnnou, je to chyba!
- Pro „rozkládání“ struktur navíc slouží příkaz `case`; ten má seznam alternativ a vrátí první z nich, jejíž struktura odpovídá.
- U příkazu `if` vždy musí být `else` větev – je to výraz, tedy něco vrátit musí!

Nyní si uvedeme ještě několik užitečných funkcí definovaných ve standardní knihovně. U funkcí uvádíme i jejich typy (pro zjednodušení většinou jinak, než jak jsou skutečně definované):

- `==, /=, <, >, <=, >= :: Int->Int->Bool` – porovnávání čísel

- `+, -, *, div, mod :: Int->Int->Int` – aritmetika
- `min, max :: Int->Int->Int` – minimum a maximum dvou čísel
- `minimum, maximum :: [Int]->Int` – minimum a maximum seznamu (neprázdného)
- `&&,|| :: Bool->Bool->Bool`, `not::Bool->Bool` – logické operace and, or, not
- `fst :: (a,b)->a, snd :: (a,b) -> b` – první a druhý prvek dvojice
- `. :: (b->c)->(a->b)->(a->c)` – skládání funkcí
- `$ :: (a->b)->(a->b)` – nedělá nic; umožnuje nám psát například `min 1 $ min 2 $ min 3 4` místo `min 1 (min 2 (min 3 4))` – viz pravidla pro závorkování
- `error :: String -> a` – vypíše chybu a pak ukončí program
- `head`, `last :: [a]->a`, `tail::[a]->[a]` – první a poslední prvek seznamu, seznam po odtržení prvního prvku
- `++ :: [a]->[a]->[a]` – zřetězení dvou seznamů
- `map::(a->b)->[a]->[b]` – aplikuje funkci na každý prvek seznamu

A nyní již *úlohy* pro vás. Mějme typ pro strom definovaný takto:

```
data Tree a = Tree a [Tree a] deriving (Show)
```

Tedy strom prvků typu `a` obsahuje hodnotu typu `a` a seznam svých synů, což jsou opět stromy téhož typu (klauzule `deriving Show` zajistí, že překladač Haskellu bude umět hodnoty tohoto typu zobrazovat).

1) Máme dán strom celých čísel (typu `Tree Int`). Napište funkci, která zkonstruuje stejný strom, v němž budou všechny hodnoty nahrazeny minimálním prvkem zadaného stromu.

Příklad:

Vstup:

```
Tree 6 [Tree 2 [],
        Tree 3 [
         Tree 4 [],
         Tree 5 []]]
```

Výstup:

```
Tree 2 [Tree 2 [],
        Tree 2 [
         Tree 2 [],
         Tree 2 []]]
```

2) Vaše řešení předchozí úlohy zřejmě obsahuje 2 průchody stromem (jeden na nalezení minima, druhý na konstrukci výsledku). Zkuste vymyslet řešení, které řeší tutéž úlohu jediným průchodem. Rozmyslete si, jak bude počítač toto řešení vyhodnocovat.

15-2-1 Zatížení — 10 bodů

Bylo, nebylo. Za devatero dveřmi, devatero stoly a devatero stohy papíru leželo mezi razítky království Byrostánie. A nebylo to jen tak ledajaké království, bylo to království úředníků. Všichni úředníci každé ráno přišli do práce, svě-

domitě celou pracovní dobu bouchali razítky, podepisovali, schvalovali a pak večer odešli domů.

Jednoho dne si začali úředníci stěžovat, že jejich platy jsou příliš nízké, neboť nejen že stráví celou pracovní dobu razítkováním a jinou bohulibou činností, ale ještě musí dlouho do práce dojíždět. Když začali horkokrevnější úředníci volat po všeúřednické stávce, spočítal si rychle vrchní úředník a král v jedné osobě, že je zle (brzy by totiž zemřel zavalen listinami, které by psal, protože by je nikdo neodebíral). Rozhodl se proto zkrátit pracovní dobu o čas potřebný na dojíždění. K tomuto státotvornému kroku by ovšem potřeboval vědět, jak dlouho jezdí takový průměrný úředník do práce.

Vaším úkolem tedy je napsat program, který dostane na vstupu počet měst v Byrostánii N (města si očíslujeme od 1 do N), dále počet silnic M a pak popis jednotlivých silnic (každá silnice je popsána čísly dvou měst, mezi kterými vede, a časem, který je třeba k jejímu projetí). Dále dostane program počet úředníků K a K dvojic čísel měst, mezi kterými příslušný úředník musí jezdit (mezi dvěma městy může jezdit i více úředníků). Na výstup má váš program vypsat nejmenší možnou (bereme všechny možnosti, kudy mohou úředníci do práce jezdit) průměrnou dobu, jakou úředníci stráví cestou do práce.

Příklad: Pro 5 měst, 5 silnic (1, 2, 1) (z města 1 do města 2 vede silnice s dobou jízdy 1 hodina), (1, 3, 2), (2, 4, 4), (3, 4, 3), (4, 5, 3) a 3 úředníky jezdící mezi městy (1, 5), (4, 1) a (3, 2) je nejmenší možná průměrná doba jízdy 16/3.

15-2-2 Závorky — 8 bodů

Váš úkol v této úloze je velice jednoduchý. Napište program, který ověří, zda řetězec na vstupu složený ze znaků `(` a `)` tvoří korektní uzávorkování. Korektní uzávorkování je posloupnost znaků `(` a `)`, ve které jdou znaky `(` spárovat se znaky `)` do dvojic takovým způsobem, že znak `(` předchází znak `)`. Navíc pokud se první znak dvojice A vyskytuje před prvním znakem dvojice B, tak se druhý znak dvojice A vyskytuje před prvním nebo za druhým znakem dvojice B.

Příklady: Řetězce `(()(()))` a `()()` tvoří správné uzávorkování. Řetězce `(()` a `())(` správné uzávorkování netvoří.

15-2-3 Defragmentace — 10 bodů

Poté, co jste v minulé sérii pomohli stěhovací firmě Popruh a spol., vás nyní oslovila firma CCD (Corporation for Continuous Data) zabývající se stěhováním dat. Tato firma se mimo jiné zabývá i defragmentací disků. Poslední studie, které si firma nechala zpracovat, ukázaly, že optimální rozložení N souborů délek $B_1, \ldots, B_N$ je takové, že první soubor leží na blocích $1, \ldots, B_1$, druhý soubor na blocích $B_1 + 1, \ldots, B_1 + B_2$ atd. Navíc bloky jednoho souboru jsou

seřazené podle svého pořadí v souboru. Protože CCD nyní vyvíjí nový software pro defragmentaci, požádala vás, abyste jí s ním pomohli.

Vaším úkolem je napsat program, který dostane na vstupu velikost disku D, počet souborů na disku N a pak popis N souborů. Popis i-tého souboru se skládá z počtu bloků B_i, které soubor obsahuje, a dále ze seznamu čísel B_i bloků (i-tý prvek seznamu odpovídá i-tému bloku souboru). Na výstup váš program musí vypsat libovolnou nejkratší posloupnost přesunů bloků, která uvede disk do výše popsaného stavu (přesun bloku je operace, která zkopíruje blok A na blok B).

Poznámka 1: Pro jednoduchost můžete předpokládat, že na disku je alespoň jeden blok volný.

Poznámka 2: Zkuste se při programování této úlohy vyhnout poli velikosti D – počet bloků na disku může být hodně velký, ačkoliv skutečný počet obsazených bloků je velmi malý.

Příklad: Pro disk velikosti 16 bloků a 2 soubory velikostí 3 a 5 na blocích $3, 8, 2$ a $1, 4, 7, 16, 15$ je posloupnost přesunů třeba $4 \to 5, 1 \to 4, 3 \to 1, 7 \to 6, 15 \to 8, 2 \to 3, 16 \to 7, 8 \to 2$.

15-2-4 Žabky — 11 bodů

Veleváženě dámy, velectění pánové! Vítejte na představení přeslavného cirkusu Žamberto. Uvidíte nevídané artistické kousky, uslyšíte neslýchané vtipy našich klaunů a jako hvězda dnešního večera vystoupí sám principál Žamberto se svou drezurou stáda žab.

A skutečně. Po artistech a klaunech vešel do manéže bodrý chlapík a za ním skákalo několik žab oblečených do dresů. Poté co se všichni uklonili, seřadily se žabky na značkách do řady podle čísel na svých dresech. Principál pak řekl: „Hop!“ a jedna z žabek přeskočila na volnou značku. Když takhle žabky chvíli skákaly, zjistili jste, že žabky jsou teď seřazeny v opačném pořadí. Úžasné!

Tak takhle nějak by mohlo vypadat představení slavného cirkusu. Jenže žabky pana Žamberta jsou častým vystupováním už unaveny a nechtějí skákat. Pan Žamberto se tedy rozhodl, že zkusí vymyslet, jak naučit žabky skákat tak, aby se naskákaly co nejméně. A protože principál je přeci jen pouze cirkusák, měli byste mu s tímto úkolem pomoci vy.

Žabky mají na zemi vyznačeno $N + 1$ značek. Na počátku stojí žabky na značkách $1, \ldots, N$ a značka $N + 1$ je prázdná. V každém kroku může na volnou značku přeskočit žabka z nějaké značky s volnou sousedící, nebo ze značky ob jedno. Vaším úkolem je napsat program, který dostane na vstupu N – počet žabek – a vypíše postup (k popisu jednoho skoku žabky stačí vypsat značku, ze které žabka skáče na volné místo), jak mají žabky skákat, aby na konci stály v opačném pořadí, volná byla buď první nebo poslední značka a přitom

bylo provedeno co nejméně skoků. Body se udělují i za zdůvodnění, proč vaším programem navržený způsob vyžaduje nejmenší počet skoků.

Příklad: 4 žabky můžou skákat třeba následovně:

`S (1234␣), 3 (12␣43), 1 (␣2143), 2 (2␣143),`
`4 (241␣3), 5 (2413␣), 3 (24␣31), 1 (␣4231),`
`2 (4␣231), 4 (432␣1), 5 (4321␣).`

15-2-5 Haskell — 12 bodů

Jedním z rysů funkcionálních jazyků, který umožňuje značnou úsporu práce, je možnost psát funkce obecně, tj. tak, aby byly schopné bez modifikace pracovat nad různými druhy dat. Tuto schopnost jsme si demonstrovali již v minulé sérii na příkladech – funkce `delka` zjišťuje délku libovolného seznamu, bez ohledu na typ jeho prvků, `obrat` ho zase dokáže obrátit.

Podívejme se nyní na tuto ideu podrobněji. Zkusme napsat funkci, která quicksortem setřídí seznam. Propagační verze programu vypadá takto:

```
qsort [] = []
qsort (h:t) = qsort [x | x <- t, x <= h] ++ h : qsort [x | x <- t, x > h]
```

Když se budu držet prostředků, které jsem si zadefinoval, bude program o něco delší:

```
qsort::[a]->[a]
qsort [] = []                       -- třídit prázdný seznam jde snadno...
qsort (h:t) = s                     -- jinak zvol první prvek jako pivot,
 where
  (m,v) = split h t                 -- rozděl seznam na prvky menší a větší než pivot,
  ms = qsort m                      -- obě části rekurzivně setřiď
  vs = qsort v
  s = ms ++ h : vs                  -- a spoj je v odpovídajícím pořadí

  split p [] = ([],[])              -- rozdělení prázdného seznamu
  split p (x:r) =
    let (rm,rv) = split p r         -- nejprve rozděl tělo seznamu
     in if p < x then (rm,x:rv)     -- a pak přidej hlavu kam patří
                 else (x:rm,rv)
```

Drobný háček je v tom, že to nebude fungovat. Problém je v tom, že této funkci mohu podstrčit jakýkoliv seznam – a na jeho prvcích nemusí být žádným rozumným způsobem zavedeno porovnávání! Toto lze řešit několika způsoby:

- Mohli bychom to kontrolovat při běhu programu. Když bychom dostali takový seznam neporovnatelných prvků, skončili bychom s chybou. Toto řešení má několik nevýhod – jednak by zpomalovalo běh programu spoustou kontrol, jednak je daleko pohodlnější, když jsou takové chyby odhaleny už při kompilaci.
- Mohli bychom `qsort`u přidat další parametr, což by byla funkce, která by porovnávala dva prvky seznamu (tj. `qsort` by měl typ

`(a->a->Bool) -> [a] -> [a]`). To je však nepohodlné – kromě seznamu bychom všude po programu museli přenášet funkci, která porovnává jeho prvky.

- Místo toho přidáme do jazyka možnost „slíbit“, že prvky budou porovnatelné. V Haskellu to uděláme snadno – změníme typ `qsortu` na `qsort::(Ord a) => [a] -> [a]`

Takových standardních „slibů“ je definováno více (`Eq` pro typy, u kterých lze testovat na rovnost, `Show` pro typy, které lze zobrazit, `Num` pro číselné typy, `Bounded` pro typy s omezeným rozsahem, ...). Po parametrech funkcí můžeme požadovat i více vlastností, například u následující funkce provádějící porovnání dvou funkcí s omezeným definičním oborem:

```
sameFunction::(Bounded a, Num a, Eq b) => (a->b) -> (a->b) -> Bool
sameFunction f1 f2 =
  sameFromTo minBound maxBound  -- dvě fce jsou stejné, když jsou stejné
                                -- na celém definičním oboru
 where
  sameFromTo from to =
    if f1 from /= f2 from                   -- pokud se hodnoty neshodují
     then False                             -- funkce nejsou stejné
     else if from == to                     -- pokud už jsme na konci intervalu
            then True                       -- pak stejné jsou
            else sameFromTo (from + 1) to   -- jinak ještě musí být stejné
                                            -- na zbytku def. oboru
```

Samozřejmě žádný konečný seznam slibů není dostačující – pokaždé by se našla další užitečná vlastnost. Proto existuje způsob, jak si nadefinovat vlastní; a dokonce ani tyto standardní vlastnosti nejsou nijak výjimečné. Deklarace pro výše použité `Ord` vypadá takto:

```
data Ordering = LT | EQ | GT deriving (Eq)

class (Eq a) => Ord a where
  (>),(<),(<=),(>=) :: a->a->Bool
  compare :: a->a->Ordering

  p1 < p2 = compare p1 p2 == LT
  p1 > p2 = compare p1 p2 == GT
  p1 <= p2 = compare p1 p2 /= GT
  p1 >= p2 = compare p1 p2 /= LT
```

Tato deklarace říká: aby typ měl vlastnost `Ord`, musí mít vlastnost `Eq` a musí pro něj být definována funkce `compare` pro porovnávání; dále pro takový typ existují funkce `<`, `>`, `<=`, `>=` definované uvedeným způsobem. Klauzule `deriving (Eq)` u definice typu `Ordering` ukazuje, že standardní vlastnosti jsou přeci jen trochu zvláštní – je možné tímto způsobem říci, že tato vlastnost má být definována zřejmým (co to přesně znamená, viz specifikace) způsobem.

Zbývá ukázat, jak zajistit, že typ má danou vlastnost. K tomu je potřeba nadefinovat všechny potřebné funkce. Například:

```
data D = D Int Int deriving (Eq)
```

```
instance Ord D where
  compare (D x1 y1) (D x2 y2) =
    if x1 == x2
      then compare y1 y2
      else compare x1 x2
```

Taková definice může samozřejmě příslušnou vlastnost definovat i pro parametrizovaný typ; například toto je definice vlastnosti `Eq` pro funkce:

```
instance (Bounded a, Num a, Eq b) => Eq (a->b) where f1 == f2 = sameFunction f1 f2
```

A nyní *úložka:* Výše popsaný mechanizmus nám zjevně umožňuje psát „přetížené“ funkce, tj. funkce, které jsou definovány rozdílně pro různé typy parametrů. Co je již méně zřejmé, je, že nám to též umožňuje obejít nemožnost psát funkce s proměnným počtem parametrů (jaký by taková funkce měla typ?). Zkuste napsat funkci, která se bude chovat podobně jako Céčkovská funkce `printf`; bude se tedy dát napsat

```
(printf "Muzeme vypisovat retezce: %s, cisla: %d a zase retezce: %s"
   "ret1" (8::Int) "ret2")::String
```

a výsledkem je řetězec

```
"Muzeme vypisovat retezce: ret1, cisla: 8 a zase retezce: ret2"
```

Poznámky a rady:

- Specifikovat typ u čísla 8 je nutné, neboť čísla mají v Haskellu typ `(Num a) => a` (aby se dala používat jako hodnoty libovolného číselného typu) a překladač by jinak nebyl schopen zjistit, o jaký typ se vlastně jedná.
- Specifikovat typ výsledku je také nutné (pokud nevyplývá z dalších operací s ním) – proč pochopíte, vyřešíte-li tuto ulohu (opačná implikace zřejmě platí také).
- K převodu čehokoliv na řetězec slouží funkce `show::(Show a) => a -> String`.

15-3-1 Potrubí — 11 bodů

Za patero stohy dopisů, třemi poštovními schránkami a známkovým mořem leží Postánie. V Postánii si velmi libují v zasílání různých dopisů, pohledů či korespondenčních lístků, a protože standardní způsob přepravy zásilek je příliš pomalý, často se k doručování využívá potrubní pošta. Zásilka potrubní poštou putuje tak, že je na jedné poště vložena do potrubí a vypadne na druhém konci potrubí na jiné poště. Tam je opět vložena do nějakého potrubí (to se samozřejmě zvolí podle cílové pošty) a putuje takto dále, dokud nedojde na cílovou poštu. Udržování systému potrubí je ale poměrně nákladná záležitost, a tak vrchní poštmistr přemýšlel, jak zminimalizovat potřebný počet potrubí.

Přitom by ale nechtěl snížit úroveň služeb zákazníkům. Po nějaké době zjistil, že to není vůbec jednoduché, a proto vás požádal o pomoc.

Vaším úkolem je napsat program, který dostane na vstupu počet poštovních stanic N a počet potrubí mezi stanicemi M. Dále dostane popis oněch M potrubí – každé potrubí je jednoznačně určeno číslem poštovní stanice, ze které vychází, a číslem poštovní stanice, do které vede (stanice číslujeme od 1 do N). Potrubí jsou jednosměrná, ale mezi dvěma stanicemi mohou vést klidně dvě potrubí – každým směrem jedno. Na výstup má váš program vypsat nejkratší možný seznam potrubí, která je třeba zachovat (nová potrubí nesmíte vytvářet) pro zajištění služeb zákazníkům. Tedy pokud dříve mohla dojít zásilka z pošty A na poštu B, tak tam může dojít i po redukci potrubí. Pokud je možných seznamů více, stačí vypsat libovolný z nich.

Příklad: Pro 6 poštovních stanic a 8 potrubí $(2,1)$, $(2,3)$, $(3,4)$, $(4,2)$, $(4,1)$, $(6,5)$, $(5,4)$, $(6,4)$ je jedním z možných řešení zachovat potrubí $(2,3)$, $(3,4)$, $(4,2)$, $(2,1)$, $(6,5)$, $(5,4)$ (místo potrubí $(2,1)$ bychom též mohli zachovat potrubí $(4,1)$).

15-3-2 Permutace — 9 bodů

Permutace P čísel $1,\ldots,N$ je libovolné prosté zobrazení množiny přirozených čísel $\{1,\ldots,N\}$ na sebe. Permutace tedy přiřazuje každému $i \in \{1,\ldots,N\}$ nějaké číslo $P[i] \in \{1,\ldots,N\}$ a tato čísla jsou pro různá i různá. k-té složení permutace P je permutace P^k s vlastností:

$$P^k[i] = \underbrace{P[P[\ \ldots P[i]]]}_{k\text{-krát}},\ \forall i \in \{1,\ldots,N\}.$$

Napište program, který dostane na vstupu N, k, a dále N čísel popisujících permutaci P (i-té číslo je hodnota $P[i]$) a na výstup vypíše k-té složení permutace (ve stejném formátu jako na vstupu). Snažte se, aby program pracoval co nejrychleji a předpokládejte, že k může být o hodně větší než N.

Příklad: Pátým složením permutace na číslech $1,\ldots,5$ tvaru $(2,1,4,5,3)$ je permutace $(2,1,5,3,4)$.

15-3-3 Tajemný obraz — 10 bodů

V Chile objevili archeologové tajemný obraz z předkolumbovské doby. Obraz vypadá jako několik bodů rozmístěných na pomyslné kružnici, přičemž každé dva body jsou spojeny rovnou čarou. Každá z čar je buď žlutá nebo červená. Archeologové dlouho hledali smysl této malby, ale žádný nemohli nalézt. Až amatérský archeolog a dobrodruh Erik von Kätzinken po dlouhém pátrání vyslovil hypotézu, že obrazec je poselstvím dávných Inků. Počet jednobarevných trojúhelníků s vrcholy na pomyslné kružnici prý udává počet dní od na-

malování obrazce, po nichž mají na Zemi opět přistát mimozemšťané. Protože obrazec je poměrně velký, rozhodl se Erik určit počet těchto jednobarevných trojúhelníků pomocí počítače.

Vaším úkolem je napsat program, který dostane na vstupu počet bodů nakreslených na obrazci N, $3 \leq N$ a dále seznam dvojic čísel bodů (body si očíslujeme od jedné do N), které jsou propojeny červenou čarou (zbylé dvojice bodů jsou tedy propojeny žlutou). Na výstup program vypíše počet jednobarevných trojúhelníků v zadaném obrazci (tzn. takových trojúhelníků, jejichž všechny tři vrcholy leží v bodech vyznačených na kružnici a jsou spojeny čarami téže barvy).

Příklad: V obrazci s pěti body a červenými čarami mezi body $(1, 2)$, $(2, 3)$, $(3, 4)$ a $(2, 4)$ jsou tři jednobarevné trojúhelníky (jeden červený a dva žluté). Červený trojúhelník má vrcholy v bodech $2, 3, 4$ a žluté trojúhelníky v bodech $1, 3, 5$ a $1, 4, 5$.

15-3-4 Výsadek — 10 bodů

Krtci se jednoho krásného dne rozhodli provést výsadek (nebo možná přesněji výhrabek) na zahradě strýčka Pompa. Zahrada strýčka Pompa je ale dosti členitá a rozlehlá a když se takový krtek někde vyhrabe ze země, má velké problémy zjistit, zda je v zahradě či nikoliv (obzvláště proto, že zrak nepatří k zrovna silným vlastnostem krtků). Centrální velitelství krtčích výhrabkových sil (CVKVS) proto každého krtka vybavilo navigačním systémem GPS, takže když se krtek vyhrabe, zná přesně svou polohu (do té doby ji nezná, protože GPS pochopitelně nefunguje pod zemí). Nyní by ale CVKVS ještě potřebovalo program pro svůj počítač, který z polohy krtka určí, zda je v zahradě či nikoliv. A to je již úkol pro vás.

Napište program, který dostane na vstupu popis zahrady strýčka Pompa (zahrada strýčka Pompa je nekonvexní N-úhelník) – tedy počet sloupků v plotu zahrady N a dále jejich souřadnice (tedy N dvojic čísel). Souřadnice sloupků jsou zadávány v tom pořadí, v jakém jsou sloupky na obvodu zahrady. Po předzpracování popisu zahrady má váš program co nejrychleji odpovídat na dotazy, zda zadané souřadnice leží nebo neleží uvnitř zahrady. Při řešení se snažte, aby doba předzpracování byla rozumná (polynomiální), ale důležitá je hlavně rychlost odpovědi na dotaz.

Příklad: Pro zahradu o pěti sloupcích se souřadnicemi $(0, 5)$, $(3.2, 5)$, $(2.2, 3.1)$, $(4.2, 1)$, $(0, 0)$ by měl program odpovědět na dotaz $(1, 1)$ kladně (krtek je v zahradě) a na bod $(5, 4)$ záporně (krtek není v zahradě).

15-3-5 Haskell — 10 bodů

V minulé sérii jsme se zabývali psaním obecných funkcí; nicméně jejich obecnost byla poměrně omezeného druhu – víceméně šlo pouze o omezení se na (pro danou úlohu) relevantní vlastnosti objektu. V této sérii se podíváme ještě o úroveň výše, na prostředky umožňující nám vyjádřit ještě abstraktnější obecně používané postupy.

Typickým použitím je vyjádření sekvencionality nějakých procesů. Podívejme se například na následující situaci: nechť se naše úloha skládá z nějakých akcí, které mohou selhat, a selže-li jedna z nich, chceme přestat provádět ostatní akce a vrátit chybu. Navíc některé z těchto akcí mohou záviset na předchozích výsledcích. Konkrétně se může jednat třeba o načtení několika hodnot z databáze a zápis výsledku na nich založeného výpočtu; kterákoliv akce s databází může kvůli nějaké chybě selhat. Jak toto budeme realizovat?

Samotné funkce pro práci s databází budou mít tyto typy:

```
cti :: Databaze -> Klic -> Maybe Hodnota
pis :: Databaze -> Klic -> Hodnota -> Maybe Databaze
  -- pis musí vracet novou databázi, protože v Haskellu nelze změnit hodnotu
  -- proměnné (nemáme přesně definováno pořadí vyhodnocování operací, takže
  -- bychom jinak nebyli schopni rozhodnout, zda přistupujeme k nové či staré
  -- databázi)
```

Připomeňme definici typu `Maybe`:

```
data Maybe a = Just a |
               Nothing
```

Hodnotu `Nothing` budeme používat pro návrat chyby. Náš program by tedy vypadal asi takto:

```
uprava :: Databaze -> Maybe Databaze
uprava databaze =
  case cti databaze klic1 of
    Nothing -> Nothing                              -- chyba
    Just hodnota1 ->
      case cti databaze klic2 of
        Nothing -> Nothing                          -- chyba
       Just hodnota2 ->
         let vysledek = vypocet hodnota1 hodnota2 in
         case pis databaze klic3 vysledek of
            Nothing -> Nothing                      -- chyba
           Just databaze' -> Just databaze'         -- hurá, všechno je OK
```

Tento program je samozřejmě velmi nepřehledný, při jeho psaní jsme se mohli snadno splést a zbytečně opakujeme pořád stejnou práci – to nás vede k myšlence společné úseky schovat do nějaké funkce, která by vypadala třeba takto:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  -- kvuli jednoduššímu používání si nadefinujeme operátor (funguje to úplně
  -- stejně jako definice funkce)
Nothing >>= f = Nothing
  -- předchozí výpočet selhal --> konec
```

```
Just r >>= f =
  -- máme výsledek přechozího výpočtu, provedeme ten následující:
  let rf = f r in
  case rf of
    Nothing -> Nothing    -- chyba
    Just r1 -> Just r1    -- OK
```

Přidáme si ještě jednu konstrukci běžnou pro funkcionální jazyky – anonymní funkce. Výraz `\x -> x + x` je funkce s jedním parametrem, která provede naznačený výpočet (zdvojnásobení `x`). To je užitečné například pokud někde potřebujeme jednoduchou funkci, kterou nepoužíváme nikde jinde; třeba `map (\x -> x + 1) s` zvětší hodnotu všech prvků v seznamu `s` o 1. Pravá strana `->` sahá vždy tak daleko, jak je to jen možné; v následujícím příkladu tedy až ke konci definice funkce `uprava`.

Nyní můžeme výše uvedenou funkci napsat mnohem elegantněji:

```
uprava :: Databaze -> Maybe Databaze
uprava databaze =
  cti databaze klic1 >>= \hodnota1 ->
  cti databaze klic2 >>= \hodnota2 ->
  let vysledek = vypocet hodnota1 hodnota2 in
  pis databaze klic3 vysledek >>= \databaze' ->
  Just databaze'
```

Povšimněte si podobnosti s následujícím imperativním programem (ve skutečnosti je toto také program v Haskellu; zde popisovaná technika je natolik důležitá, že pro ni byla zavedena tato zkrácená forma zápisu):

```
uprava databaze =
  do
   hodnota1 <- cti databaze klic1
   hodnota2 <- cti databaze klic2
   let vysledek = vypocet hodnota1 hodnota2
   databaze' <- pis databaze klic3 vysledek
   return databaze'
```

Podobnost samozřejmě není náhodná – naším záměrem bylo popsat sekvenční vyhodnocování, což je základní rys imperativních jazyků.

Tohle by ještě nebylo až tak zajímavé; nicméně existuje mnoho dalších aplikací podobné myšlenky (uveďme třeba konstrukci parseru, přenášení stavu mezi výpočty, distribuci náhodných čísel, simulaci nedeterministických výpočtů, vstup a výstup, ...). Jejich společné vlastnosti zachycuje následující standardní třída `Monad`:

```
class Monad m where
 (>>=) :: m a -> (a -> m b) -> m b
 return :: a -> m a
 fail :: String -> m a
```

Povšimněme si toho, že `m` v této definici je „funkce“ nad typy – instancemi této třídy jsou typové konstruktory, jako například `Maybe`, `[]`, nikoliv typy `Maybe a`, `[a]`. Konkrétní příklad instance:

```
instance Monad Maybe where
 (>>=) = ...                  -- viz výše
 return x = Just x
 fail s~= Nothing
```

Samotný typ `Monad m => m a` je potřeba chápat jako „akci (program, proces, výpočet), určený typem `m`, vracející hodnotu typu `a`“. `>>=` je pak zřetězení takových dvou akcí (přičemž ta druhá má k dispozici výsledek první), `return a` je prázdná akce, vracející `a`, a `fail` označuje chybu (parametr je chybová hláška, v našem případě je prostě ignorována – nemáme ji kam uložit).

Proč jsou monády tak důležité?

- Mnoho zajímavých úloh v sobě zahrnuje nějakým způsobem sekvenčnost, a přirozeným způsobem jim odpovídají monády.
- Když už se nám podařilo nějakou úlohu zformulovat jako monádu, můžeme využít již napsaného kódu pro práci s nimi, až už jsou to standardní knihovny nebo speciální syntaxe zabudovaná přímo v jazyce.
- Právě monády jsou použity jako čistý a přitom jednoduchý a k používání příjemný způsob řešení vstupu a výstupu v Haskellu.

Na poslední bod se nyní podívejme trochu blíže. Vstup a výstup v čistém funkcionálním jazyce je problém – vše mají být funkce, tedy nesmí mít žádné vedlejší efekty, což interakce s okolím rozhodně je. I když bychom dovolili výjimky, máme problém:

- Haskell nemá předepsané pořadí vyhodnocování, na druhou stranu u vstupu a výstupu musíme dodržovat přesné pořadí operací (náhodně proházené řádky na výstupu by vypadaly divně, a to je nejmenší z problémů).
- V Haskellu není rozdíl mezi výpočtem a jeho hodnotou; mohlo by se nám stát, že jednu vstupní či výstupní akci bychom vyhodnocovali vícekrát.

Řešením je monáda `IO`. `IO a` je akce zahrnující případné vstupy a výstupy a jako výsledek vracející hodnotu typu `a`. K dispozici máme funkce jako `getChar::IO Char` (akce, která načte a vrátí znak) a `putChar::Char->IO ()` (funkce, která pro každý znak vrátí akci, která tento znak vypíše). Co je důležité, v Haskellu neexistuje funkce, která by takovou akci vyhodnotila. Jediný způsob, jak příslušnou akci provést, je přiřadit ji jako definici konstanty (tedy funkce bez paramentru) `main::IO ()`.

Samotný typ `IO a` by mohl vypadat (ve skutečnosti je to abstraktní typ, tj. jeho vnitřní struktura může být jakákoliv a nemůžeme s ním přímo pracovat) zhruba takto:

```
data IO a = IO (Svet -> (Svet, a))
```

To je funkce, která vezme stav světa, aplikuje na něj nějaké vstupně-výstupní operace a vrátí nový upravený svět a výsledek. (`Svet` je typ, který (teoreticky) obsahuje stav celého světa. To řeší problém vedlejších efektů – žádné nejsou, protože není žádné „vedle“. Práce `IO` monády spočívá v tom, že nám se `Svet`em znemožňuje dělat věci typu kopírování, vrácení se ke starší verzi a podobně). Operace `>>=` by pak byla definována zřejmým způsobem:

```
IO akce1 >>= f = IO akce
 where
   akce svet =
     let (svet1,vysledek1) = akce1 svet
         IO akce2          = f vysledek1
      in akce2 svet1
```

Tolik teorie. Samotný program pak vypadá třeba takto:

```
main :: IO ()
main =
 do
  l <- getLine              -- načti řádku
  let rev = reverse l       -- obrať ji
  putLine rev               -- a vypiš
  return ()
```

Hezké je, že předchozí program může napsat kdokoliv, bez jakýchkoliv vědomostí o tom, jak věci skutečně fungují.

Úloha: Mějme funkci

```
page :: String -> String -> String
page stav dotaz = "Stav: " ++ stav ++ "; " ++ dotaz ++ "?"
```

Napište monádu `CGI` a dále dvě funkce

```
runCGI :: CGI () -> String -> String -> String,
|cgiPage :: String -> CGI String|,
```

pro které program

```
cgi :: CGI ()
cgi =
 do
   r1 <- cgiPage "Dotaz 1"
   r2 <- cgiPage "Dotaz 2"
   r3 <- cgiPage ("Dotaz 3 (vase predchozi odpoved byla " ++ r2)
   return ()
```

bude fungovat takto:

```
> runCgi cgi "" ""
"Stav: q=0; Dotaz 1?"
> runCgi cgi "q=0" "odpoved1"
"Stav: q=1,o1=odpoved1; Dotaz 2?"
> runCgi cgi "q=1,o1=odpoved1" "odpoved2"
"Stav: q=2,o1=odpoved1,o2=odpoved2; Dotaz 3 (vase predchozi odpoved byla odpoved2)?"
> runCgi cgi "q=2,o1=odpoved1,o2=odpoved2" "odpoved3"
""
```

15-4-1 Archiv — 10 bodů

Pan Sips pracuje jako úředník v archivu, kde se ukládají různé dokumenty. Dokumenty jsou v archivu uloženy vzestupně podle jejich evidenčního čísla. Jednoho dne ale do archivu přišel nový úředník a neznalý zdejšího pořádku, uložil některé dokumenty na jiná místa, než kam patří, a porušil tím setřídění. Pan Sips je nyní nešťastný, protože setřídit všechny dokumenty je obrovská práce. Požádal vás proto o pomoc s tímto problémem.

Vaším úkolem je napsat program, který na vstupu dostane počet spisů v archívu N a dále posloupnost N evidenčních čísel dokumentů (předpokládejte, že se vejdou do typu `long` respektive `Longint`). Na výstup má váš program vypsat setříděnou posloupnost evidenčních čísel. Snažte se, aby třídění proběhlo co nejrychleji, a předpokládejte, že počet špatně zařazených dokumentů je řádově menší než celkový počet dokumentů.

Příklad: Pro posloupnost 1,3,5,2,7,6,10,12,14,11 by váš program měl vypsat posloupnost 1,2,3,5,6,7,10,11,12,14.

15-4-2 Permutace podruhé — 11 bodů

Permutace P čísel $1, \ldots, N$ je libovolné prosté zobrazení množiny přirozených čísel $\{1, \ldots, N\}$ na sebe. Permutace tedy přiřazuje každému $i \in \{1, \ldots, N\}$ nějaké číslo $P[i] \in \{1, \ldots, N\}$ a tato čísla jsou pro různá i různá. Dvojité složení permutace P je permutace P^2 definovaná jako:

$$P^2[i] = P[P[i]], \ \forall i \in \{1, \ldots, N\}.$$

Napište program, který dostane na vstupu N a dále N čísel popisujících permutaci P (i-té číslo je hodnota $P[i]$) a na výstup vypíše počet permutací Q takových, že jejich druhým složením vznikne permutace P (tedy takových, že $Q^2 = P$).

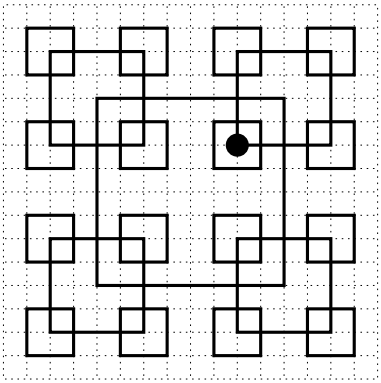
Příklad: Permutaci na číslech $1, \ldots, 7$ tvaru (2, 1, 5, 6, 7, 4, 3) lze získat jako druhou mocninu dvou permutací – (4, 6, 7, 2, 3, 1, 5) a (6, 4, 7, 1, 3, 2, 5).

15-4-3 Koberec — 9 bodů

Pan Naisrep tkal v daleké Persii koberce. A nebyly to koberce ledajaké, protože Naisrep se specializoval na čtvercové vzory. Jednoho dne ho napadl následující vzor: Jeho základem byl čtverec se středem uprostřed koberce o straně délky $2k$. Ve vrcholech tohoto čtverce ležely středy čtverců o stranách délky $2(k \text{ div } 2)$ a ve vrcholech těchto čtverců ležely opět středy menších čtverců a tak dále, až dokud čtverce neměly stranu délky 2. Aby vzor nebyl tak jednoduchý, rozhodl se Naisrep jednotlivé oblasti na koberci vyšít různými barvami podle toho, v kolika čtvercích leží. Ale ouha, zjistit, v kolika čtvercích leží nějaké místo, není žádná hračka. A tak se k vám Naisrep obrátil o pomoc.

Vaším úkolem je napsat program, který dostane na vstupu délku strany čtvercového koberce (sudé přirozené číslo, může být i poměrně velké), dále délku strany největšího čtverce (též sudé přirozené číslo) a souřadnice bodu na koberci ($[0,0]$ je levý horní roh). Na výstup má váš program vypsat uvnitř kolika čtverců daný bod leží (pokud bod leží na hraně nějakého čtverce, počítá se, že je uvnitř).

Příklad: Pro koberec o straně délky 16 a největší čtverec o straně délky 8 leží bod se souřadnicemi $[6,10]$ ve třech čtvercích (viz obrázek).



15-4-4 Písaři — 11 bodů

Před vynálezem knihtisku se knihy nemohly tisknout, ale musely se ručně opisovat. To byla dosti zdlouhavá práce. Například pokud herci chtěli sehrát nějakou divadelní hru, musel režisér nejdříve zajistit opis scénáře pro každého herce. Protože ale písaři stojí peníze, bylo zvykem dát každému herci pouze tu část scénáře, kterou skutečně potřebuje. Rozdělit opisování kusů scénáře mezi jednotlivé najaté písaře se tím ovšem zkomplikovalo. Proto jste byli přes propast času požádáni, abyste režisérovi pomohli.

Vaším úkolem je napsat program, který dostane na vstupu počet herců N, počet písařů K a dále pro každého herce počet stránek, které je pro něj třeba opsat. Vaším úkolem je každému písaři přiřadit herce, pro které má opsat jejich části scénáře tak, aby celkově opisování trvalo co nejméně (tedy aby maximum přes počty stránek pro jednoho písaře bylo co nejmenší). Navíc písaři píší různě pěkně a váženější herci by měli dostat hezčeji napsané scénáře. Proto musí první písař opisovat scénář pro herce $1, \ldots, h_1$, druhý písař pro herce $h_1 + 1, \ldots, h_2$ atd.

Příklad: Pro 7 herců, 3 písaře a počty stránek 8, 4, 3, 5, 6, 2, 9 je nejrychlejší, aby první písař opisoval scénař pro první dva herce, druhý písař pro další tři herce a poslední písař pro poslední dva herce.

15-4-5 Haskell — 10 bodů

Prozatím jsme se zabývali tím, jak se Haskell (a mnohé další alespoň přibližně čisté funkcionalní programovací jazyky) chovají navenek. Nyní se podíváme, co se děje uvnitř – tj. jak se tyto jazyky kompilují.

Už na úvod si řekněme, že těžko. Nejlepší současné překladače produkují kód, který bývá 5× – 10× pomalejší než odpovídající program v C, a také s výrazně vyššími paměťovými nároky. Toto je samozřejmě nefér srovnání – C je jazyk navržený pro zcela jiné účely než Haskell (a bohužel také pro jiné účely, než na které je v praxi používán). Nicméně ani srovnání s jinými imperativními jazyky nevypadá příznivě. Z větší části je to ovšem způsobeno tím, že současné počítače jsou navrženy tak, aby vyhovovaly imperativním, kódem řízeným výpočtům. Pro deklarativní jazyky by byly mnohem užitečnější daty řízené počítače, které se ovšem příliš nevyskytují (a v dohledné době zřejmě nebudou). Zaměřme se tedy na problémy, které se nám při kompilaci objevují, a způsoby jejich řešení.

V líně vyhodnocovaném jazyce nelze dopředu rozhodnout, které výpočty se pro daný vstup vlastně provedou. Víceméně nemáme jinou možnost, než provádět vždy jen ty výpočty, které právě potřebujeme – z toho ovšem plyne nutnost „odkládat“ si rozpracované výpočty stranou. Podívejme se na konkrétní příklad:

```
{- 1 -} jedničky = 1 : jedničky
{- 2 -} take 0 x = []
{- 3 -} take n x =
          case x of
            (h:t) -> h : take (n-1) t
```

Vyhodnocujme `take 3 jedničky`. Nejprve se provede 3. řádek; je tedy potřeba vyhodnotit `case jedničky of (h:t) -> h : take (n-1) t`. Toto vyžaduje zjistit, zda seznam `jedničky` není prázdný. Kvůli tomu provedeme jeden krok řádku 1. Nyní opět vyhodnocujeme řádek 3 – `take 2 t`, kvůli tomu musíme provést další krok na řádku 1, atd. – ve vyhodnocování se střídají `jedničky` a `take`. Všimněme si několika faktů:

- Nemůžeme si dovolit vyhodnotit celý seznam `jedničky`, prostě proto, že je nekonečný.
- Dokonce si ani nemůžeme dovolit vyhodnotit byť i jen jeden prvek navíc; kdybychom si nadefinovali `jedničky = 1 : 1 : 1 : error "Chyba"`, pak `take 3 jedničky` má vrátit `[1,1,1]`, kdybychom ale vyhodnotili prvek navíc, došlo by k chybě (ve skutečnosti takové „spekulativní vyhodnocování“ provádět lze, ale je potřeba značná dávka šikovnosti, abychom se vyhnuli popsaným problémům a přitom vyhodnocování ještě více nezpomalili).

- `take` vlastně nevyhodnocuje prvky seznamu; kdybychom zadali seznam `jedničky` obsahující `error "E1" : error "E2" : error "E3" : []`, pak `take 3 jedničky` vrátí seznam třech volání funkce `error`; pokud si však někdo nevynutí vyhodnocení prvků tohoto seznamu, k chybě nedojde.

Další věcí, kterou je třeba mít na paměti, je sdílení. Když napíšeme

```
let x = výpočet in x + x
```

chceme, aby se `výpočet` provedl jen jednou. To můžeme (ve spojení s líným vyhodnocováním) realizovat tak, že na místo, kde bude uložena hodnota `x`, si nejprve uložíme odložený `výpočet`, který navíc rozšíříme tak, že po svém dokončení tuto pozici přepíše hodnotou výsledku. Kvůli tomu ovšem musíme při každém přístupu k proměnné testovat, zda už byla vyhodnocena, což nám samozřejmě na rychlosti nepřidá. Asi nejelegantnější způsob, jak to jde v praxi realizovat, je ten, že vždy bude na místě proměnné uložen nějaký výpočet. Nejprve to bude ten, který vyhodnotí `výpočet`, a následně se přepíše triviálním výpočtem, který pouze vrací spočítaný výsledek. Tím se vyhneme testování, zda již byl výpočet proveden (na úkor toho, že vždy musíme provést zavolání funkce).

Další zdroj potíží je v předávání funkcí. Samotné funkce příliš velký problém nepředstavují (když už stejně musíme umět odkládat výpočty), nicméně musíme nějak zajistit předávání parametrů. Ty si musíme ukládat u příslušného „odloženého“ výpočtu. Parametry se nám mohou hromadit postupně, a navíc se nám jich může nakupit víc, než kolik potřebuje daná funkce – v případě, když vrací jinou funkci. Například

```
plus x y = x + y
id x = x
vysledek = map (id plus 5) [1, 2, 3]   -- = map (plus 5) [1, 2, 3] = [6, 7, 8]
```

se bude vyhodnocovat tak, že nejprve vytvoříme odložený výpočet (anglicky „thunk“, nejsem si jistý českým ekvivalentem) pro funkci `id` s parametry `plus` a 5. Později při vyhodnocování seznamu se z něj vyrobí nový thunk s parametry `plus`, 5, 1; pak se vyhodnotí `id`, spolkne první parametr a vrátí thunk pro funkci `plus` se zbývajícími parametry. Povšimněte si, že thunk nejde jen přepsat na místě, protože ho ještě budeme potřebovat při vyhodnocování dalšího prvku v seznamu. To se dá řešit buď zkopírováním starého thunku, nebo tím, že si parametry ukládáme ve spojovém seznamu. To může být rychlejší a úspornější – není potřeba kopírovat parametry, nicméně práce se spojovým seznamem je pomalejší, takže tyto výhody se projeví jen u funkcí s mnoha parametry. Navíc je složitější provést vyhodnocení thunku a také správa paměti (garbage collecting) se komplikuje.

Jak jste již zajisté uhodli, úkolem v této sérii bude napsat jednoduchý kompilátor. Nebudeme samozřejmě překládat celý Haskell, jen jeho malou podmno-

žinu (nicméně dostatečně velkou, aby do ní šla relativně snadno podstatná část Haskellu přeložit), ani se nebudeme zabývat takovými detaily (ve skutečnosti hodně zajímavými a důležitými) jako parsing, kontrola a odvozování typů a podobně. Také stroj, pro nějž budeme programy překládat, bude zvládat operace o něco vyšší úrovně, než je běžné – některé z nich jsou přímo navrženy tak, aby zjednodušily řešení výše uvedených problémů.
Program dostaneme zadaný jako seznam prvků typu `Equation`:

```
data Equation = Equation String [String] Expression
              -- Equation jméno_funkce parametry definice
```

Každá funkce je definována nanejvýš jednou. Parametry jsou prostě jména (tj. nedělá se tu pattern matching). Definice (typu `Expression`) je výraz, kterým je funkce definována. Hodnoty všech výrazů jsou celá čísla nebo funkce. Typ `Expression` je definován takto:

```
data Expression = Let [Equation] Expression |   -- Let definice výraz dělá totéž
                                                -- jako let definice in výraz
                  Apply Expression Expression | -- Apply funkce parametr
                                                -- aplikace funkce na daný parametr
                  Number Int |                  -- číslo
                  Variable String |             -- odkaz na proměnnou
                  Plus Expression Expression |  -- sečte dva výrazy
                  Mul Expression Expression |   -- násobí dva výrazy
                  Minus Expression Expression | -- odečte dva výrazy
                  Div Expression Expression |   -- dělí dva výrazy
                  Case Expression [(Int, Expression)] Expression
                                                -- Case výraz alternativy default
                                                -- podrobnější popis viz níže
```

Sémantika většiny příkazů je jasná – `Let` zavádí lokální definice, `Apply` je aplikace funkce, `Number` je celočíselná konstanta, `Variable` je hodnota, která je aktuálně uložena v dané proměnné, `Plus/Minus/Mul/Div` jsou aritmetické funkce. `Case` je výběr z více alternativ na základě hodnoty výrazu – zvolí se ta z alternativ, jejíž hodnota odpovídá, a vrátí se vybraný výraz; pokud neodpovídá žádná, vrátí se hodnota `default`.

Poznámka stranou – všimněte si, že se nezabýváme složitějšími typy. To je jednak pro zjednodušení, jednak proto, že se bez nich dá obejít. Například seznamy lze naprogramovat tímto způsobem:

```
cons hlava tělo = let h = hlava
                      t = tělo
                  in \sel -> if sel then h else t
head list = list True
tail list = list False
```

Tohle samozřejmě není korektní program v Haskellu – nejde otypovat. Nicméně v našem jazyce už typy nejsou, takže to nevadí. `Let` v definici funkce `cons` je nutný, abychom zajistili sdílení jednou spočítaných hodnot.

Příklad kompilovaného programu:

```
[
  Equation "mod" ["x", "y"]
     (Let [
            Equation "p" [] (Div (Variable "x") (Variable "y")),
            Equation "s" [] (Mul (Variable "p") (Variable "y"))
          ]
         (Minus (Variable "x") (Variable "s"))),
    -- mod x y = let p = x `div` y
    --               s = p * y
    --           in x -
s Equation "nsd" ["x", "y"]
     (Case (Variable "x")
        [(0, Variable "y")]
        (Let [
               Equation "p" [] (Apply
                                  (Apply (Variable "mod") (Variable "y"))
                                  (Variable "x"))
             ]
           (Apply (Apply (Variable "nsd") (Variable "p")) (Variable "x"))))
    -- nsd x y = case x of
    --             0 -> y
    --             _ -> let p = mod y x
    --                  in nsd p x
]
```

Výsledkem kompilace by měl být program pro stroj, který vypadá takto: Má několik oddílů paměti; všechny se skládají z více položek, které jsou očíslovány přirozenými čísly.

- Paměť pro kód má v každé z položek jednak kus kódu, což je seznam tvořený níže uvedenými instrukcemi, jednak číslo udávající počet parametrů, které tento kód používá.
- Paměť pro data obsahuje položky několika typů:
  - celé číslo
  - thunk – ten obsahuje odkaz do paměti pro kód (odpovídá funkci, kterou počítá) a seznam adres v datové paměti (parametry)
- Registry obsahují adresy položek v datové paměti; jsou používány při provádění instrukcí.

Kromě toho obsahuje seznam, v němž se nachází aktuálně vykonávaný kód. Stroj funguje tak, že vždy z tohoto seznamu odebere první instrukci, provede ji a celý postup opakuje. Stroj se zastaví, jestliže tento seznam bude prázdný. Dále má stroj ještě zásobník, do nějž si ukládá registry při volání funkce; ten je na počátku vždy prázdný.
Počáteční stav stroje je uložen v proměnné typu `InitialState`:

```
data InitialState =
       InitialState [(Int,Int,[Instruction])]   -- paměť pro kód
                    [(Int,Value)]               -- paměť pro data
                    [(Int,Int)]                 -- registry
```

```
                    [Instruction]              -- aktuální kód
data Value = VNumber Int |
             VThunk Int [Int]
```

„Program“ pro tento stroj je vlastně definice jeho počátečního stavu. Instrukce jsou tyto (pod `[x]` rozumíme hodnotu uloženou v paměti na adrese, obsažené v registru `x`).

```
data Instruction =
       IMove Int Int |
         -- IMove from to
         -- přesuň hodnotu z registru from do registru to
       IPlus Int Int Int |
         -- IPlus x y ret
         -- naalokuj novou položku v datové paměti, vlož do ní součet [x] a [y]
         --  ([x] a [y] musí být čísla, tj. ne thunky) a adresu této položky
         --  vlož do registru ret
       IMinus Int Int Int |
       IMul Int Int Int |
       IDiv Int Int Int |
         -- analogicky pro ostatní operace
       IExecute Int |
         -- IExecute w
         -- pokud [w] je číslo: vlož w do registru 0
         -- pokud [w] je thunk, obsahující adresu k do kódové paměti a seznam
         --  parametrů l: buď n počet parametrů položky k z kódové paměti,
         --  c kód tam uložený. Je-li n menší než délka l, vlož w do registru 0
         --  jinak
         --    odlož obsah registrů na zásobník
         --    smaž registry
         --    do registrů 1 až n vlož prvních n prvků l
         --    zbytek l ulož na zásobník
         --    c připoj na začátek aktuálního kódu
       IReturn |
         -- smaž všechny registry až na 0
         -- ostatní registry obnov ze zásobníku
         -- jestliže je seznam parametrů uložený na zásobníku neprázdný, přidej
         --  je na konec seznamu parametrů thunku [0] ([0] musí být thunk)
         -- na začátek aktuálního kódu připojí instrukci IExecute 0
       ICExecute Int Int Int |
         -- ICExecute x y w
         -- pokud [x] == [y] ([x], [y] musí být čísla), vlož instrukci IExecute w
       ICall Int Int |
         -- ICall k reg
         -- naalokuj novou položku v paměti, vytvoř na ní thunk obsahující
         --  odkaz k do paměti pro kód a prázdný seznam parametrů. Adresu
         --  této položky vlož do registru reg
       ILoad Int Int |
         -- ILoad val reg
         -- naalokuj novou položku v paměti, vytvoř na ní číslo val a adresu
         --  této položky vlož do registru reg
       IApply Int Int Int |
         -- IApply t p reg
         -- vezmi thunk [t], zkopíruj ho na nově vytvořené místo v datové paměti,
         --  na konec seznamu parametrů kopie přidej adresu z registru p a
         --  adresu této kopie vlož do registru reg
       IRewrite Int Int
         -- IRewrite what with
```

```
-- položku v datové paměti s adresou uloženou v registru what přepiš
--  hodnotou [with]
```

Vaším úkolem je napsat funkci

```
compile::[Equation]->Expression->InitialState
```

která dostane na vstup program a výraz ve výše uvedeném tvaru a výsledkem bude počáteční stav stroje takového, že hodnota výrazu bude na konci jeho výpočtu v `[0]`.

15-5-1 Komprimovaný obrázek — 10 bodů

Firma Quartz Graphics řešila v jednom ze svých programů následující problém: Měla obrázek o rozměrech $r \times s$ bodů. Každý bod obrázku měl určen svou barvu – nějaké celé číslo z intervalu $0, \ldots, 255$. Problém spočíval v nalezení co největší souvislé jednobarevné oblasti (dva body pokládáme za sousední, pokud se jejich poloha liší o jedna buď v x-ové nebo y-ové souřadnici). Programátoři firmy ale zjistili, že obrázky jsou příliš velké, a proto se je rozhodli komprimovat pomocí metody RLE. Tato metoda spočívá v tom, že si jednotlivé řádky obrázku naskládáme za sebe a vzniklou posloupnost pak zakomprimujeme tak, že celý souvislý úsek tvořený jediným číslem zakódujeme jako dvojici *(délka úseku, číslo v úseku)*. Tedy například posloupnost 1112233313333222 zakomprimujeme na $(3,1),(2,2),(3,3),(1,1),(4,3),(3,2)$. Řešit ovšem problém největší souvislé jednobarevné oblasti v takto zakomprimovaném obrázku již není jednoduché, a proto se firma obrátila na vás.

Vaším úkolem je navrhnout algoritmus a napsat program, který na vstupu dostane rozměry obrázku a zakomprimovaný obrázek (tedy nějakou posloupnost dvojic) a na výstup vypíše velikost největší souvislé jednobarevné oblasti. Protože po dekompresi může být obrázek značně velký, zkuste se ve svých řešeních vyhnout jeho dekompresi, nebo ho alespoň nedekomprimovat celý najednou.

Příklad: Pro obrázek o rozměrech 4×4 a výše uvedenou posloupnost dvojic by měl váš program vypsat 6 – oblast této velikosti je tvořená trojkami na druhém a třetím řádku.

15-5-2 Odečtolam — 12 bodů

Pepíček k vánocům dostal novou počítačovou logickou hru Odečtolam. Hra se hraje na pravidelném n-úhelníku, v jehož vrcholech jsou umístěna celá čísla. Navíc pro hru vždy platí, že součet všech čísel je nezáporný. Povoleným tahem ve hře je zvolení nějakého záporného čísla, jeho přičtení k oběma jeho sousedům na n-úhelníku a otočení jeho znaménka. Pokud tedy ve třech následujících vrcholech byla čísla a, b, c, kde $b < 0$, tak po tahu do vrcholu s číslem b tam budou čísla $a+b, -b, c+b$. Cílem hry je dosáhnout situace, kdy jsou všechna čísla

v n-úhelníku nezáporná. Protože Pepíček hru pořád nemohl vyhrát, rozhodl se, že si na ni napíše program. Pomůžete mu s ním?

Vaším úkolem je napsat program, který dostane na vstupu počet vrcholů mnohoúhelníka n a dále n čísel, která jsou na počátku v jeho vrcholech. Na výstup má váš program vypsat posloupnost tahů, která vede do cílové pozice, popřípadě zprávu, že taková posloupnost neexistuje. Nezapomeňte, že součástí řešení by měl být nejen váš algoritmus, ale i důkaz, že váš algoritmus pracuje správně. Tedy že najde posloupnost tahů právě tehdy, když existuje.

Příklad: Pro $n = 4$ a čísla $-1, -2, 3, 1$ je možná posloupnost tahů například 1 (tím získáme čísla $(1, -3, 3, 0)$), 2 $(-2, 3, 0, 0)$, 1 $(2, 1, 0, -2)$, 4 $(0, 1, -2, 2)$, 3 $(0, -1, 2, 0)$, 2 $(-1, 1, 1, 0)$, 1 $(1, 0, 1, -1)$, 4 $(0, 0, 0, 1)$.

15-5-3 Manhattan — 10 bodů

Ulice na Manhattanu mají jak známo tvar čtvercové sítě. Řekněme, že na Manhattanu vede m ulic severojižním směrem a n ulic východozápadním. Bill vyjíždí každé ráno z křižovatky o souřadnicích $1, 1$ (severozápadní roh) a jede do práce na křižovatku o souřadnicích m, n (jihovýchodní roh). Některé křižovatky se ovšem opravují, a tak přes ně Bill nemůže jet. Billa by zajímalo, kolika způsoby se může do práce dostat, pokud chce vždy jet pouze ze severu na jih nebo ze západu na východ. A to je úkol pro vás.

Napište program, který dostane na vstupu rozměry města m a n, počet opravovaných křižovatek k a souřadnice oněch opravovaných křižovatek (předpokládejte, že křižovatka $(1, 1)$ ani křižovatka (m, n) nejsou opravované). Na výstup má pak váš program vypsat počet cest, kterými Bill může jet.

Příklad: Pro město o rozměrech 3×4 a 2 rozkopané křižovatky na souřadnicích $(2, 2)$ a $(3, 3)$ má Bill 2 možnosti, jak do práce dojet.

15-5-4 Továrna — 12 bodů

Továrna na tvarůžky (TNT) získala velkou zakázku na výrobu svých jedinečných tvarůžků výbušné chuti. Problémem ale je, že továrna vlastní pouze N_1 strojů na lisování tvarůžků a N_2 strojů na jejich balení. Protože stroje jsou různě staré, je i jejich rychlost lisování a balení různá. Je proto poměrně obtížné vymyslet, jak strojům práci rozdělit, aby zakázka byla co nejrychleji splněna. Vedení podniku se tedy obrátilo na vás, abyste pomohli svými znalostmi.

Vaším úkolem je napsat program, který dostane na vstupu počet tvarůžků T, které je třeba vyrobit. Dále dostane počet lisovacích strojů N_1 a časy, za které stroje vylisují jeden tvarůžek – nějaká posloupnost N_1 čísel $t^1_1, \ldots, t^1_{N_1}$. Nakonec dostane ještě počet balících strojů N_2 a časy, za které stroje zabalí jeden tvarůžek – posloupnost N_2 čísel $t^2_1, \ldots, t^2_{N_2}$. Na výstup má váš program vypsat nejkratší čas, za který je možno zakázku splnit.

Příklad: Pro 10 tvarůžků, 3 lisovací stroje s časy $2, 2, 5$ a 2 balící stroje s časy $6, 3$ je nejkratší možný čas výroby 21.

15-5-5 Haskell — 10 bodů

Vzhledem k tomu, že praktické úložky v Haskellu byly pro vás zřejmě příliš triviální a bylo pod vaši úroveň je řešit, podíváme se nyní na funkcionální jazyky z teoretické stránky.

Nejspíš už jste slyšeli, že teoretickým základem pro funkcionální programovací jazyky je λ-kalkulus. Co to vlastně je?

Všechny objekty, se kterými v λ-kalkulu pracujeme, jsou funkce. Pokud bychom opravdu toužili po tom mít i nějaké hodnoty, mohli bychom je chápat jako funkce z jednobodové množiny (nicméně se bez toho obejdeme). Tyto funkce budou mít vždy právě jeden parametr – nicméně vzhledem k tomu, že to, co vrací, je zase funkce, můžeme je chápat jako funkce s libovolným množstvím parametrů (viz podobný trik použitý pro funkce s více parametry v Haskellu). Termy (tedy výrazy) λ-kalkulu jsou definovány takto:

- Proměnná je term. Proměnné budeme značit malými písmeny $x, y, \ldots$ (všechny proměnné jsou pochopitelně funkce).
- Jsou-li S, T termy, je $(S)(T)$ také term odpovídající tomu, že zavoláme funkci S s parametrem T. Tato operace se nazývá aplikace.
- Je-li T term, je i $\lambda x(T)$ term odpovídající tomu, že vytvoříme novou funkci s parametrem x takovou, že při jejím zavolání se dosadí parametr do T za x a vyhodnotí se výsledný term. Této operaci se říká abstrakce.

Výskyt proměnné x v termu T je vázaný, pokud je uvnitř podtermu tvaru $\lambda x(S)$, jinak je volný. Výrazem $T_{x:=S}$ budeme rozumět term T, kam za všechny volné výskyty x dosadíme S, tj. formálně (pro $x \neq y$):

- $x_{x:=S} = S$
- $y_{x:=S} = y$
- $[(U)(V)]_{x:=S} = (U_{x:=S})(V_{x:=S})$
- $[\lambda x(U)]_{x:=S} = \lambda y(U)$
- $[\lambda y(U)]_{x:=S} = \lambda y(U_{x:=S})$

Pro aplikaci a abstrakci platí následující pravidlo, které formalizuje výše popsanou sémantiku: $(\lambda x(T))(S) = T_{x:=S}$.

Aby se nám zbytečně nekupily závorky, zavedeme si následující konvence (podobné řešení užíval i Haskell): Pokud neuvedeme závorky, aplikace se závorkují doleva, abstrakce doprava. Závorky kolem proměnných neuvádíme. Tj. například:

$$\lambda x \lambda y \lambda z\ x\ y\ z = (\lambda x\ (\lambda y\ (\lambda z\ (x)\ )\ )\ (y)\ )\ (z).$$

Výraz $\lambda x\ .\ T$ znamená „zazávorkuj T tak, aby pravá závorka byla co nejvíc vpravo“, tj.:

$$\lambda x\ .\ \lambda y\ .\ \lambda z\ .\ x\ y\ z = \lambda x\ (\lambda y\ (\lambda z\ (x\ y\ z)\ )\ ).$$

Povšimněte si, že nemáme žádný způsob, jak si funkce pojmenovat – zdá se, že o něčem takovém jako rekurze nemůže být řeč. Naštěstí máme následující term (budeme ho označovat Y):

$$\lambda f\ .\ (\lambda x\ .\ f\ (x\ x)\ )\ (\lambda x\ .\ f\ (x\ x)\ ).$$

Y splňuje následující identitu:

$$\begin{aligned} Y\ g &= (\lambda x\ .\ g\ (x\ x)\ )\ (\lambda x\ .\ g\ (x\ x)\ ) = \\ &= g\ ((\lambda x\ .\ g\ (x\ x)\ )\ (\lambda x\ .\ g\ (x\ x)\ )\ ) = \\ &= g\ (Y\ g), \end{aligned}$$

tj. pro libovolnou funkci g vrátí její pevný bod (tj. y takové, že $y = g\ y$. Y se proto nazývá operátor pevného bodu. K čemu je to dobré? Mějme například rekurzivní funkci, která vrací délku seznamu:

$$\begin{aligned} \mathit{length} = \lambda s\ .\ &\texttt{if}\ \mathit{empty}\ s\ \texttt{then}\ 0 \\ &\texttt{else}\ 1 + \mathit{length}\ (\mathit{tail}\ s) \end{aligned}$$

S použitím operátoru pevného bodu ji můžeme napsat bez použití rekurze takto:

$$\begin{aligned} \mathit{length} = Y\ (\lambda g\ .\ \lambda s\ .\ &\texttt{if}\ \mathit{empty}\ s\ \texttt{then}\ 0 \\ &\texttt{else}\ 1 + g\ (\mathit{tail}\ s)\ ) \end{aligned}$$

Skutečně, označíme-li si term vpravo od Y jako T, dostáváme

$$\begin{aligned} \mathit{length}\ (h : t) = Y\ T\ (h : t) = T\ (Y\ T)\ (h : t) = \\ = 1 + Y\ T\ t = 1 + \mathit{length}\ t, \end{aligned}$$

což je přesně to, co chceme.

Nyní si můžeme již snadno zavést aritmetiku. Označme si například $\mathit{Nula} = \lambda x.\lambda y.x$ a $\mathit{Succ} = \lambda s.\lambda x.\lambda y.y\ s$. Pak za nulu považujeme term *Nula*, za jedničku term *Succ Nula*, za dvojku *Succ* (*Succ Nula*), atd. Každé takové číslo je pak funkce se dvěma parametry; *Nula* vrátí první z nich, zatímco jakékoliv jiné číslo zavolá druhý z nich s parametrem o jedna menším. Například funkce *Pred*, která od kladného čísla odečte jedničku a nulu nezmění, vypadá takto:

$$\mathit{Pred} = \lambda n.n\ \mathit{Nula}\ (x.x).$$

Nyní asi již není těžké uvěřit, že v λ-kalkulu dokážeme napsat libovolný program, který lze napsat v Haskellu (nebo jakémkoliv jiném rozumném programovacím jazyce).

Úlohy:

1. Napište term *Sum*, který dostane dvě čísla ve výše popsané reprezentaci a sečte je.
2. Napište termy Y_1 a Y_2 – operátory dvojného pevného bodu, splňující následující identity:

$$Y_1\ g\ h = g\ (Y_1\ g\ h)\ (Y_2\ g\ h),$$
$$Y_2\ g\ h = h\ (Y_1\ g\ h)\ (Y_2\ g\ h).$$

Vzorová řešení

15-1-1 Narozeninový dort — Marek Sulovský

Úloha patřila mezi těžší, takže řešení přišlo opravdu poskrovnu. Většina z nich fungovala v čase $O(N^2)$, a byly tu nějaké (nepříliš úspěšné) pokusy o řešení v čase $O(N \log N)$. Ukážeme si vzorové řešení, které běhá v $O(N^2)$ a dá se (pravda trochu pracně) upravit na řešení běžící v $O(N \log N)$.

Nejprve si trochu přeformulujeme zadání – snažíme se najít bod X v trojúhelníku ABC takový, že v trojúhelnících AXB, BXC i CXA je stejný počet svíček ($N/3$). Takovému bodu X budeme říkat řešení. Dále si nadefinujeme – pro zjednodušení následujícího textu – jeden pojem: úhlem bodu D trojhelníka ABC vzhledem k nějakému z jeho vrcholů budeme rozumnět úhel XAC v případě vrcholu A, XBA v případě vrcholu B a XCB v případě vrcholu C. Úhel budeme značit $U(\mathit{Vrchol}, \mathit{Bod})$. Dále je dobré si uvědomit, že trojúhelník lze rozdělit na oblasti, ve kterých jsou buď všechny body řešeními nebo není žádný bod řešením. Pokud totiž povedu přímky ze všech vrcholů trojúhelníku všemi svíčkami, budu mít trojúhelník rozdělen na několik konvexních mnohoúhelníků. Jestliže jeden bod vnitřku je řešením, pak jsou řešením všechny body vnitřku, protože mají stejnou polohu vůči přímkám z vrcholů trojúhelníku a svíčkám, a tedy jsou stejné počty svíček v oblastech, na které se řezem trojúhelník rozdělí. Pořád ale nevíme nic o okrajích těchto konvexních oblastí – o některých z nich můžeme říct, že na nich řešení nemůže ležet, protože by řezalo přes nějakou svíčku, ale mohou nám pořád zbýt nějaké, pro které řešení svíčku řezat nebude. Zbývá nám tedy otázka, co to pro nás znamená. Je však zřejmé, že takové řešení se dá velice nepatrně posunout, aby na takové přímce neleželo (formální důkaz tohoto tvrzení by byl docela zdlouhavý a musel by se patrně dělat rozborem případů – podle počtu těchto „nežádoucích“ přímek, na kterých naše řešení leží – a následným výpočetním vyjádřením vektoru, o který můžeme řešení posunout, aby se nám rozdělení svíček nezměnilo, ale přitom nové řešení už neleželo na žádné „nežádoucí“ přímce). Naše úloha je tedy ekvivalentní s hledáním řešení v jednotlivých mnohoúhelnících (resp. otestování jednoho libovolného bodu z každého mnohoúhelníku).

Ideově nejjednodušší řešení by tedy mohlo pracovat tak, že si spočítá všechny tyto oblasti a pro každou otestuje např. její těžiště. Dá se dokázat, že oblastí je $O(N^2)$ (každá přímka je rozdělena na max. $2N$ místech, každé takové místo je vrcholem nějakého mnohoúhelníka a je společné maximálně šesti mnohoúhelníkům, každý má alespoň jeden vrchol, odtud získáme, že počet mnohoúhelníků je maximálně $3N \cdot 2N \cdot 6 = 36N^2 = O(N^2)$). Teoreticky by tedy mohlo být možné vymyslet na základě této myšlenky kvadratický algoritmus, avšak je-

ho naprogramování by bylo patrně velice komplikované – takže my se raději uchýlíme ke komplikovanějšímu, ale zato na naprogramování ne příliš složitému kvadratickému algoritmu, který se dá zrychlit na časovou složitost $O(N \log N)$.

Náš algoritmus pracuje následovně: Nejprve si pro každý vrhol trojúhelníku setřídíme svíčky podle úhlu vzhledem ke každému z jeho vrcholů (dostaneme tedy tři setříděné posloupnosti svíček podle jednotlivých úhlů). Pro vrchol A získáme K_A různých úhlů, kde $K_A \leq N$. Označme si $\{UA\}_{i=1}^{K_A}$ rostoucí posloupnost těchto různých úhlů (podobně definujeme UB a UC). Postupně (pro $N/3 \leq i \leq 2D/3$) budeme zkoušet hledat řešení X takové, že $UA_i < U(A,X) < UA_{i+1}$ (v mnohých z vašich řešení jste situaci, když K_A, K_B nebo K_C je menší než N vůbec nebrali v úvahu). Nyní budu hledat krajní body intervalu, ve kterém musí ležet úhel X vzhledem k B (resp. k C), aby v oblasti ABX (resp. ACX) bylo $N/3$ bodů. To můžu udělat tak, že projdu všechny dvojice po sobě jdoucích úhlů v posloupnosti UB (resp. UC) a budu si postupně počítat, kolik svíček mám v oblasti AXB (resp. AXC). Nejmenší a největší úhel, pro který budu mít napočítáno $N/3$ bodů, budou krajní body hledaného otevřeného intervalu. Nyní mám tedy intervaly úhlů X vzhledem k A, B a C, kde mám zajištěno, že v AXB a AXC je $N/3$ svíček a musím zjistit, zda mi tyto intervaly určují nějaké řešení – tedy jestli výseče jimi určené mají společný průsečík. Intervaly úhlů z B a C určují čtyřúhelník (bez krajových úseček). Ten má s výsečí z vrcholu A průsečík právě když má jeho delší úhlopříčka (ve smyslu pohledu z vrcholu A) bez krajních bodů průsečík s výsečí z vrcholu A. To už je jednoduché ověřit pomocí nějakých algebraických výpočtů. Teď ověříme, že tento algoritmus skutečně funguje. Pokud má úloha řešení, existuje nějaký úhel z A, ve kterém se řešení „nachází“. Potom také musí existovat interval úhlů $U(B,X)$ a $U(C,X)$, které mají neprázdný průnik, a tedy náš algoritmus nějaké řešení najde. Pokud úloha řešení nemá, žádné výseče které najdu nemohou mít průnik (jinak by muselo existovat řešení...), a tedy můj algoritmus nic nenajde.

Nyní ještě něco k řešením. Všechna aspoň trochu správná řešení byla založena na docela podobném principu, ale mnohdy jste zapomínali na nějaké „drobnosti“, jako třeba že může být více svíček se stejným úhlem vzhledem k jednomu z vrcholů trojúhelníka apod. – což znamená, že vaše řešení v takovém provedení, jak jste napsali, nebude vždy fungovat, i když by (v některých případech) nebylo těžké je upravit na funkční řešení, takže jste většinou nějaký bod dostali. Dále jste často zapomínali na to, že floaty se nedají přímo porovnávat na rovnost – mělo by se zjišťovat, zda se neliší pouze o nějaké malé předem dané ϵ, které vám v zásadě pokrývá odchylky při výpočtech.

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

```
#define CHILDREN 3
#define MAXAGE 1024
#define MAXN (CHILDREN*MAXAGE)
#define EPS 0.00001

struct vector {
    double x, y;
} vertices[CHILDREN];

struct candle {
    int id, angleid;
    double angle;
} candles[CHILDREN][MAXN];

int N;
double d;

double
angle (double x, double y, int v)
{
    /* kosinus uhlu je skalarni soucin podeleny soucinem velikosti */
    double nx = x - vertices[v].x, ny = y - vertices[v].y, cosa;
    cosa = (nx* (-vertices[v].x+vertices[ (v+CHILDREN-1)%CHILDREN].x) +
            ny* (-vertices[v].y+vertices[ (v+CHILDREN-1)%CHILDREN].y))
      / (d*hypot (nx, ny));
    return sqrt (1-cosa*cosa);
}

int
angle_cmp (struct candle *a, struct candle *b)
{
    /* kdyz se uhly lisi jen o velmi malo (EPS), jsou stejne */
    if (a->angle - b->angle > EPS)
      return 1;
    if (b->angle - a->angle > EPS)
      return -1;
    return 0;
}

int
intersection (struct vector s0, struct vector e0, struct vector s1, struct vector e1,
              struct vector s2, struct vector e2, struct vector *sv)
{
    /* v poradi Amin, Amax, Bmin, Bmax, Cmin, Cmax, a tady ulozime vysledek */
    /* nejsou to uhly, ale vektory nejakych svicek */
    struct vector u, v, stp, endp;
    double t, sti, endi;

    u.x = s1.x - vertices[1].x; u.y = s1.y - vertices[1].y;
    v.x = s2.x - vertices[2].x; v.y = s2.y - vertices[2].y;
    t = ( ( (vertices[1].x-vertices[2].x)/u.x)- ( (vertices[1].y-vertices[2].y)/u.y))/
        (v.x/u.x - v.y/u.y);
    stp.x = vertices[2].x+t*v.x; stp.y = vertices[2].y+t*v.y;
    u.x = e1.x - vertices[1].x; u.y = e1.y - vertices[1].y;
    v.x = e2.x - vertices[2].x; v.y = e2.y - vertices[2].y;
```

```
    t = ( ( (vertices[1].x−vertices[2].x)/u.x)− ( (vertices[1].y−vertices[2].y)/u.y))/
        (v.x/u.x − v.y/u.y);
    endp.x = vertices[2].x+t∗v.x; endp.y = vertices[2].y+t∗v.y;
    v.x = endp.x − stp.x; v.y = endp.y − stp.y;
    u.x = s0.x; u.y = s0.y;
    t = (stp.y / u.y − stp.x / u.x) / (v.x / u.x − v.y / u.y);
    if (t < 0) sti = 0;
    else if (t > 1) sti = 1;
    else sti = t;
    u.x = e0.x; u.y = e0.y;
    t = (stp.y / u.y − stp.x / u.x) / (v.x / u.x − v.y / u.y);
    if (t < 0) endi = 0;
    else if (t > 1) endi = 1;
    else endi = t;
    if (sti > endi) {
       t = sti;
       sti = endi;
       endi = t;
    }
    if (sti + EPS > endi)
       return 0;
    t = (sti + endi) / 2;
    sv−>x = stp.x + t∗v.x;
    sv−>y = stp.y + t∗v.y;
    return 1;
}
int
main (void)
{
    int i, j, k;
    struct vector vcandles[MAXN];
    int dangles[CHILDREN], angles[CHILDREN][MAXN];
    int position[CHILDREN][MAXN], anglespart[CHILDREN−1][MAXN];
    int st[CHILDREN−1], end[CHILDREN−1], cnt[CHILDREN − 1];
    struct vector sv;
    /∗ Inicializace, nacteni vstupu ∗/
    scanf ("%lf%d", &d, &N);
    vertices[0].x = 0; vertices[0].y = 0;
    vertices[1].x = d; vertices[1].y = 0;
    vertices[2].x = d/2; vertices[2].y = sqrt (3)∗d/2;
    for (i = 0; i < N; i++)
       {
          scanf ("%lf%lf", &vcandles[i].x, &vcandles[i].y);
          for (j = 0; j < CHILDREN; j++)
             {
                candles[j][i].id = i;
                candles[j][i].angle = angle (vcandles[i].x, vcandles[i].y, j);
             }
       }
    /∗ Setrizeni podle uhlu z jednotlivych vrcholu ∗/
    for (j = 0; j < CHILDREN; j++)
```

```
    qsort (candles[j], N, sizeof (struct candle),
           (int (*) (const void *, const void *))angle_cmp);
  /* Pole ruznych uhlu z ednotlivych vrcholu, useky stejnych uhlu, ... */
  for (j = 0; j < CHILDREN; j++)
    dangles[j] = 1 + (*angles[j] = 0);
  for (j = 0; j < CHILDREN; j++)
    for (i = 1; i < N; i++)
      {
        if (angle_cmp (&candles[j][i], &candles[j][i-1]))
          angles[j][dangles[j]++] = i;
        candles[j][i].angleid = dangles[j]-1;
      }
  for (j = 0; j < CHILDREN; j++)
    angles[j][dangles[j]] = N;
  /* Zapamatovani si pozic vicek v setrizenych polich */
  for (j = 0; j < CHILDREN; j++)
    for (i = 0; i < N; i++)
      position[j][candles[j][i].id] = i;
  /* Pocatecni nastaveni poctu svicek v jednotlivych castech (podle deleni z A) */
  for (j = 1; j < CHILDREN; j++)
    for (i = 0; i < dangles[j]; i++)
      anglespart[j-1][i] = (j - 1) ?0: (angles[j][i+1] - angles[j][i]);
  /* Vypocet */
  for (i = 0; i < dangles[0] - 1; i++)
    {
      /* Prepocitani poctu svicek s jednotlivymi uhly v jednotlivych castech */
      for (j = angles[0][i]; j < angles[0][i+1]; j++)
        {
          anglespart[0][candles[1][position[1][candles[0][j].id]].angleid]--;
          anglespart[1][candles[2][position[2][candles[0][j].id]].angleid]++;
        }
      /* Hledani zacatku a konce intervalu pro B a C */
      /* Vrchol B */
      cnt[0] = k = 0;
      while (k < dangles[1] && cnt[0] < N/3)
        cnt[0] += anglespart[0][k++];
      if (cnt[0] != N/3) goto nextloop;
      st[0] = k-1;
      while (k < dangles[1] && !anglespart[0][k]) k++;
      end[0] = k < dangles[1] ? k: k-1;
      /* Vrchol C */
      cnt[1] = 0;
      k = dangles[2] - 1;
      while (k >= 0 && cnt[1] < N/3)
        cnt[1] += anglespart[1][k--];
      if (cnt[1] != N/3) goto nextloop;
      end[1] = k+1;
      while (k >= 0 && !anglespart[1][k]) k--;
      st[1] = k >= 0 ? k: k+1;
      if (intersection (vcandles[candles[0][angles[0][i]].id],
                                vcandles[candles[0][angles[0][i+1]].id],
```

```
                                vcandles[candles[1][angles[1][st[0]]].id],
                                vcandles[candles[1][angles[1][end[0]]].id],
                                vcandles[candles[2][angles[2][st[1]]].id],
                                vcandles[candles[2][angles[2][end[1]]].id],  &sv)) {
            printf ("%lf %lf\n",  sv.x,  sv.y);
            return 0;
          }
        nextloop:
        }
    printf ("No solution\n");
    return 0;
}
```

15-1-2 Vláčky — Martin Mareš

„Ejhle, grafová úloha!“ vykřikl železniční černokněžník pan Zababa a vyklepal to samou radostí do drážního telegrafu tak mocně, až vrabci na telegrafních drátech nadskakovali. A hned začal kout pikle, že celý problém šikovně převede na hledání maximálního toku v síti, a to je přeci standardní úloha, kterou ho kdysi dávno na vysoké škole pro dopravní černokněžníky učili. My si z něj ovšem příklad brát nebudeme a necháme už beztak vystrašené opeřence být, nestřílejíce po nich kanónem. A toky v sítích si sice v literatuře najdeme a prostudujeme (určitě se někdy budou hodit), ale raději si zvolíme prostředky poněkud přízemnější, i když teorii toků neupíráme jistou dávku inspirace.

Kontrolním bodům budeme říkat *nádraží*, jednotlivým kolejím v kontrolních bodech *nástupiště* a kolejím mezi kontrolními body *trati.* Celé kolejiště tedy můžeme popsat orientovaným grafem, jehož *vrcholy* budou odpovídat nástupištím, budou seskupeny do *vrstev* odpovídajících nádražím a podle tratí mezi nimi povedou *hrany*, a to z každé vrstvy jen do té bezprostředně následující (tento směr nazveme *zleva doprava*). Zkrátka jako na obrázku v zadání. První vrstvě budeme říkat *počáteční* (do té nevedou žádné hrany), poslední vrstvě *koncová* (naopak nevedou žádné ven), ostatním vrstvám *vnitřní.*

Naším cílem tedy bude v tomto grafu najít co nejvíce *vrcholově disjunktních cest,* tj. cest, které vždy vedou z nějakého vrcholu počáteční vrstvy do nějakého vrcholu koncové vrstvy a žádné dvě cesty nemají společný vrchol. Lépe se nám ovšem budou hledat cesty *hranově disjunktní,* tak si pomůžeme prostým trikem: Každý vrchol v nahradíme dvěma vrcholy v_1 a v_2, přičemž do v_1 povedou všechny hrany, které vedly do v, z v_2 povedou ty, co vedly z v, a navíc přidáme hranu $v_1 \to v_2$. Počet vrstev se tím zdvojnásobí. Když nalezneme největší množinu hranově disjunktních cest v tomto novém grafu a vrcholy „stáhneme zpět“, dostaneme největší možnou množinu vrcholově disjunktních cest v původním grafu (rozmyslete si, proč to funguje).

Ještě si všimněme, že nám stačí místo cest nalézt takovou množinu M hran, která je *vyvážená,* tj. s každým vrcholem ve vnitřních vrstvách sousedí

stejný počet hran z M zleva jako zprava. Snadno ověříme, že tehdy je počet hran z M mezi každými dvěma sousedními vrstvami stejný. Pak už M můžeme snadno rozložit na disjunktní cesty *hladovým algoritmem:* vyjdeme z počáteční vrstvy po libovolné hraně z M, přijdeme do nějakého vrcholu, vydáme se libovolnou další hranou z M doprava (ta musí vždy existovat, neboť M je vyvážená), až dospějeme do koncové vrstvy. Nalezenou cestu z grafu odstraníme, čímž vyváženost neporušíme, a tak můžeme stejný postup opakovat, dokud M nevyprázdníme. A jelikož všechny cesty jsou stejně dlouhé, největší množině disj. cest odpovídá největší vyvážená množina a opačně.

[Odbočka k tokům v sítích: Nyní bychom mohli přidat zdroj z připojený hranami ke všem vrcholům v počáteční vrstvě a spotřebič s, do nějž povedou hrany z vrcholů koncové vrstvy, všem hranám nastavit jednotkovou kapacitu a nalézt (třeba metodou Tří Indů v čase $O(n^3)$) maximální celočíselný tok ze z do s v této síti. Množina hran sítě, po kterých teče nenulové (tj. jedničkové) množství, je právě největší vyvážená.]

A jak na to půjdeme my? Začneme s libovolnou vyváženou množinou hran M (třeba prázdnou, ta je po ruce vždycky) a budeme ji zlepšovat tak dlouho, dokud to půjde, a až to nepůjde, prohlásíme ji za největší možnou. A vymyslíme si ono zlepšování tak šikovně, aby opravdu největší byla.

Definička: Hrany grafu si obarvíme dvěma barvami: *Červené* budou hrany ležící v M, *modré* všechny ostatní.

Pro lepší představu: s vrcholy ve vnitřních vrstvách sousedí buďto samé modré hrany nebo dvě červené (jedna zleva a jedna zprava) a všechny ostatní jsou modré (jinak nelze, neboť každý vrchol má z alespoň jedné strany stupeň 1, totiž tam, kde vede hrana přidaná při dělení vrcholů, a cesty v M jsou hranově disjunktní). Analogicky pro hrany v počáteční a koncové vrstvě, až na to, že červená hrana může být jen jedna.

Definice: *Zlepšující cestou* budeme nazývat takovou cestu, která povede z vrstvy počáteční do koncové, přičemž může používat buďto modré hrany zleva doprava nebo červené hrany zprava doleva. (Není to tedy poctivá orientovaná cesta, ale to nám nebude vadit.)

Pokud existuje nějaká zlepšující cesta, můžeme množinu M určitě vylepšit: stačí, když všechny červené hrany ležící na zlepšující cestě z M odebereme a ty modré naopak přidáme. M tak zůstane vyvážená a její velikost vzroste. Zajímavější ale je opačná implikace:

Věta: Pokud žádná zlepšující cesta neexistuje, M je největší vyvážená množina.

Pokud tato věta platí (to za chvilku dokážeme), již máme hotový algoritmus:

1. Rozdělíme vrcholy
2. $M \leftarrow \emptyset$

3. Nalezneme prohledáváním grafu do šířky (červené hrany v protisměru) zlepšující cestu C. Dokud existuje, M podle ní vylepšíme a hledáme další zlepšující cestu.
4. Pokud už žádná zlepšující cesta neexistuje, M je největší možná, takže ji hladovým algoritmem převedeme na hranově disjunktní cesty.
5. Vrcholy opět sloučíme a získáme tak hledané vrcholově disjunktní cesty v původním grafu.

Program bude fungovat přesně podle tohoto algoritmu. Graf si budeme pamatovat pomocí seznamů hran incidentních s každým vrcholem, každou hranu budeme mít uloženu v obou směrech a budeme si u ní pamatovat její aktuální barvu. Drobný trik: kroky 4 a 5 lze provádět najednou, abychom si zbytečně nemuseli pamatovat mezivýsledek.

Jak je to s časovou složitostí? Kroky 1, 2, 4 a 5 a každá iterace kroku 3 zaberou čas $O(m+n)$. Zlepšujících cest můžeme nalézt maximálně k (to je velikost nejmenší vrstvy – každé zlepšení nám totiž zvýší počet využitých vrcholů v každé vrstvě), takže celkový čas je $O(k \cdot (m+n))$. Paměti spotřebujeme $O(m+n)$.

A to už je všechno. Aha, ještě slíbený

Důkaz: Povedeme sporem. Nechť M je vyvážená množina, pro kterou neexistuje žádná zlepšující cesta a N je nějaká větší vyvážená množina. Uvažme $P = (M \backslash N) \cup (N \backslash M)$, tedy symetrickou diferenci (xor) těchto dvou množin: hrana e je v P právě tehdy, když e leží v M nebo v N, ale ne v obou současně. Jak množina P vypadá? Jak už víme, s každým vrcholem v z vnitřních vrstev sousedí buďto 0 nebo 2 hrany z M (každá z jedné strany) a totéž platí pro N. Takže mohou nastat následující případy:

a) v nesousedí ani s hranami z M, ani z $N \Rightarrow$ ani z P
b) v sousedí pouze s hranami z $M \Rightarrow$ sousedí z každé strany s jednou hranou z P
c) v sousedí pouze s hranami z $N \Rightarrow$ analogicky
d) v sousedí jak s hranami z M, tak i z $N \Rightarrow$ jedna z těchto hran musí být společná M i N (v má přeci z jedné strany stupeň 1), takže opět sousedí s právě dvěma hranami z P, ovšem tentokráte vedou obě z jedné strany.

Podobně to dopadne v první a poslední vrstvě, pouze „jednostranně“, takže v případech b) a c) sousedí P pouze jednou hranou a v případě d) buďto jednou nebo žádnou. A jelikož $|N| > |M|$, případů c) určitě v počáteční vrstvě nastane více než b).

Tedy víme, že graf určený hranami z P má všechny vrcholy stupně 0, 1 nebo 2. Takže to musí být disjunktní (vrcholově) sjednocení cest a kružnic. A jelikož v první vrstvě existuje alespoň jeden vrchol typu c), vede z něj nějaká

cesta začínající hranou z $N \backslash M$ (tedy vzhledem k M modrou). Následujme ji: buďto půjdeme stále po modrých hranách nebo narazíme na případ d) a začneme se vracet po červené, v dalším vrcholu typu d) můžeme opět změnit směr atd., až dorazíme do dalšího vrcholu stupně 1, kde cesta skončí. Vrcholy stupně 1 jsou ovšem pouze v počáteční a koncové vrstvě. Jenže pokud jsme se vrátili zpět do počáteční, určitě existuje další, dosud nepoužitý vrchol typu c) a v něm znovu začneme, takže jednou v koncové vrstvě skončit musíme. Pozorný čtenář však dávno ví, že taková cesta je přímo učebnicovým příkladem cesty zlepšující, o které jsme předpokládali, že neexistuje, což je kýžený spor. EPA.

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

int N;                                  /* Počet nádraží */
struct vertex {                         /* Jeden vrchol, tedy půlka nástupiště */
    int x, y;                           /* Nádraží a nástupiště */
    int h;                              /* Která polovina to je: 0=levá, 1=pravá */
    struct edge *left, *right;          /* První hrana doleva a doprava */
    struct vertex *next;                /* Další vrchol ve frontě při hledání do šířky */
    struct edge *back;                  /* Zpětný odkaz při hledání do šířky */
    int flag;                           /* Univerzální značka, v klidu nulová */
};
int *N2V;                               /* Číslo vrcholu prvního nástupiště pro každé
                                           nádraží */
struct vertex *V;                       /* Pole vrcholů */
struct edge {                           /* Jedna hrana, tedy trať */
    struct edge *next;                  /* Následující pro totéž nástupiště */
    struct edge *twin;                  /* Sestřička v protisměru */
    struct vertex *dest;                /* Cílový vrchol */
    int red;                            /* 0=modrá, 1=červená */
};

void *malloc_zero (unsigned int size)
{                                       /* Alokuje a vynuluje blok paměti */
    void *x = malloc (size);
    if (!size) exit (1);
    memset (x, 0, size);
    return x;
}

void add_edge (struct vertex *x, struct vertex *y)
{                                       /* Přidá jednu hranu v obou směrech */
    struct edge *e = malloc_zero (2*sizeof (struct edge));
    struct edge *f = e+1;
    e->next = x->right;
    x->right = e;
    e->twin = f;
    e->dest = y;
    f->next = y->left;
    y->left = f;
```

```
    f->twin = e;
    f->dest = x;
}
void read_input (void)
{                                       /* Načte vstup a vytvoří graf */
    scanf ("%d", &N);                   /* Počet nádraží */
    N2V = malloc_zero (sizeof (int) * (N+2));
    for (int i=1; i<=N; i++)            /* Počty nástupišť */
      {
        int k;
        scanf ("%d", &k);
        N2V[i+1] = N2V[i] + k;
      }
    int n = N2V[N+1];                   /* Prvních n jsou levé poloviny, dalších n pravé */
    V = malloc_zero (sizeof (struct vertex) * 2 * n);
    for (int i=1; i<=N; i++)            /* Dělíme nástupiště na vrcholy */
      for (int j=N2V[i]; j<N2V[i+1]; j++)
        {
          V[j+n].x = V[j].x = i;
          V[j+n].y = V[j].y = j-N2V[i]+1;
          V[j+n].h = 1;
          add_edge (&V[j], &V[j+n]);
        }
    int T;                              /* Počet tratí */
    scanf ("%d", &T);
    while (T--)                         /* Trati */
      {
        int x1, y1, x2, y2;
        scanf ("%d%d%d%d", &x1, &y1, &x2, &y2);
        add_edge (&V[N2V[x1]+y1-1+n], &V[N2V[x2]+y2-1]);
      }
}
void add_to_queue (struct vertex ***qlastp, struct vertex *v, struct edge *back)
{
    if (!v->flag)
      {
        **qlastp = v;
        v->next = NULL;
        *qlastp = &v->next;
        v->back = back;
        v->flag = 1;
      }
}
struct vertex *find_improving_path (void)
{                                       /* Nalezne zlepšující cestu a vrátí, kde skončila */
    struct vertex *queue = NULL, **qlast = &queue;
    for (int i=N2V[1]; i<N2V[2]; i++)
      add_to_queue (&qlast, &V[i], NULL);   /* Naplníme frontu vrcholy z vrstvy 0 */
    struct vertex *qfirst = queue;      /* Schováme si, kde fronta začínala */
    while (queue && (queue->x != N || !queue->h))
```

```
      {
        for (struct edge *e = queue->right; e; e=e->next)
          if (!e->red)                    /* Po modrých hranách doprava */
            add_to_queue (&qlast, e->dest, e);
        for (struct edge *e = queue->left; e; e=e->next)
          if (e->red)                     /* Po červených hranách doleva */
            add_to_queue (&qlast, e->dest, e);
        queue = queue->next;
      }
    /* Ještě potřebujeme smazat všechny značky */
    while (qfirst)
      {
        qfirst->flag = 0;
        qfirst = qfirst->next;
      }
    return queue;
}
void improve_along_path (struct vertex *v)
{                                         /* Zlepší podél nalezené cesty */
    while (v->back)
      {
        v->back->red ^= 1;
        v->back->twin->red ^= 1;
        v = v->back->twin->dest;
      }
}
void find_tracks (void)
{                                         /* Nalezne podle vyvážené množiny trasy vlaků */
    for (int i=N2V[1]; i<N2V[2]; i++)
      {
        struct vertex *v = &V[i];
        for (; ; )
          {
            struct edge *e;
            while ( (e = v->right) && !e->red)
              v->right = e->next;
            if (!e)
              break;
            if (!v->h)
              printf ("(%d,%d) ", v->x, v->y);
            v->right = e->next;
            v = e->dest;
          }
        if (v->h)
          putchar ('\n');
      }
}
int main (void)
{
    read_input ();
```

```
    struct vertex *v;
    while (v = find_improving_path ())
      improve_along_path (v);
    find_tracks ();
    return 0;
}
```

15-1-3 Kulový blesk — Miroslav Rudišin

Tento príklad pekně demonstruje nevýhodnost pažravého (hladového) prístupu k niektorým problémom. Tí, ktorí v prvej fáze chceli umiestniť čo najviac rodín do správnych bytov, nedokázali dokončiť sťahovanie na menej ako $\lceil log_2 N \rceil$ fáz. Trik spočíva v tom, že v prvej fázi si rodiny rozmiestnime po bytoch tak, aby po prvom sťahovaní sa rodina v byte i mala sťahovať do bytu j a podobne rodina v byte j do bytu i. Toto dosiahneme napríklad otočením prvých $n-1$ rodín. Po prvej fáze síce nikto nie je v správnom byte, ale po opätovnom otočení (tentoraz všetkých n rodín) sú v všetky tam, kam sme ich chceli dať.

$$(1, 2, \ldots, n-2, n-1, n) \to (n-1, n-2, \ldots, 2, 1, n) \to (n, 1, 2, \ldots, n-1, n-2)$$

Počet fáz je teda 2 (okrem prípadu N=2, v ktorom neprebehne prvá fáza) a časová zložitosť je $O(N)$ (v oboch fázach robíme spolu $\lfloor (n-1)/2 \rfloor + \lfloor n/2 \rfloor$ výmen).

```
#include <stdio.h>

int main (void) {
        int n, i, j;
        scanf ("%d", &n);
        puts ("1. faze");
        for (i=1, j=n-1; i<j; i++, j--)
                printf ("␣%d<->%d", i, j);
        puts ("␣-\n2. faze");
        for (i=1, j=n; i<j; i++, j--)
                printf ("␣%d<->%d", i, j);
        puts ("␣-");
        return 0;
}
```

15-1-4 Soustavy — Pavel Šanda

Stínovou stranou moderních hraček je, že lidé zapomněli sčítat pomocí obstarožní tužky a papíru, což se ku příkladu projevuje drobnými krádežemi ze strany obsluhujícího presonálu v restauračních zařízeních nebo též překvapivě velkým množstvím řešitelů, jež nepostřehli jednoduchou zákonitost při sčítání. Nezbývalo pak jináče, nežli postupně zkoušet všechny možné soustavy na korektnost a nebýt horního omezení počtu možných číslic, sčítáme patrně po dnes. Toho

si všiml houf řešitelů a počal vymýšlet vylepšování stran výběru možných základů – nutno dodat, že kreativitou v skutku Nešetřil; řešení byla doplňována sadou pozoru hodných výsledků v teorii dělitelnosti, žel bohu ve stylu *důkaz či protipříklad si nalezni sám*, což nejednoho opravovatele (ú)pějícího v brzkých ranních hodinách nad rozžatou svící nepotěší.

Podívejme na součet číslic $A+B=C$ uvnitř součtu čísel $a+b=c$ na našem pergamenu. Platí-li $A \leq C \wedge B \leq C$ $(*)$ pro všechny řády, může být základ soustavy libovolná číslice $x \geq max_{A\in a, B\in b, C\in c}(A, B, C)$. Stran případu, kdy výše uvedená podmínka $*$ neplatí: abychom se zbavili diskutování o přenosech jedničky z nižšího řádu, budiž naše trojice první takovou (sčítáme zprava), kde podmínka $*$ neplatí. Zde předáme jedničku o řád výše a klíčový postřeh pro hodnotu pod čarou: $C = A+B-základ$. Ježto A,B,C jsou známé číslice, máme tím pádem i hledaný základ.

Lze snadno nahlédnout lineární časovou a paměťovou složitost vzhledem k počtu cifer na vstupu. V případě, že by se nám podařilo přesvědčit zkostnatělé zadavatele vstupu o vhodnosti paralelního zadávání čísel od konce, šla by paměťová vylepšit na konstatní.

Poměrně velké množství z vás mělo odhad paměťové složitosti za konstantní, čehož bych se rád dotkl několika slovy – jestliže je vstupem jednotlivé číslo typu *počet měst v grafu*, které nacpeme do integeru, považujeme složitost za konstantní, zatímco pokud máme za vstup řetězec, se kterým budeme v tomto případě operovat, složitost je úměrná počtu znaků, tedy lineární.

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

#define Max 10
#define Num (x)  (isdigit (x) ?x-'0':x-'a'+10)   /* hodnota znaku */
#define N (x)  (m=Num (x), max=max>m ?max:m, m)   /* nejvetsi znak */

char a[Max], b[Max], c[Max];

int main (void)
{
  int z, m, max=0;
  scanf ("%s%s%s", &a, &b, &c);
  strrev (a); strrev (b); strrev (c);                /* obrat retezce */

  for (int i=0; i<strlen (c); i++)
    if (z = N (a[i]) + N (b[i]) - N (c[i])) {printf ("%d\n", z); return; }

  printf ("%d\n", max+1);
  return 0;
}
```

15-1-5 Haskell — Zdeněk Dvořák

Dvouprůchodové řešení této úlohy je tak jednoduché, že ho snad ani nemá smysl komentovat:

```
data Tree a = Tree a [Tree a] deriving (Show)

replaceByMin2 :: Tree Int -> Tree Int
replaceByMin2 t = r
  where
    m = minElem t                 -- najdi nejmenší prvek
    r = replaceBy m t             -- a nahraď jím ostatní

minElem :: Tree Int -> Int
minElem (Tree r sons) =
  minimum (r : map minElem sons)  -- minimum z kořene a minim synů

replaceBy :: a -> Tree a -> Tree a
replaceBy w (Tree r sons) =
  Tree w (map (replaceBy w) sons) -- nahraď hodnotu v kořeni a rekurzivně
                                  -- se zavolej na syny
```

Nyní jednoprůchodové řešení. Na první pohled se zdá, že to snad nemůže jít – jak mám hodnoty nečím nahradit, když to neznám, dokud neprojdu celý strom? Na druhý pohled si ovšem uvědomíme, že hodnota toho, čím prvky nahrazujeme, vůbec není důležitá. To, co nám umožní tuto myšlenku využít, je fakt, že Haskell je líně vyhodnocovaný – nepokusí se tedy tuto hodnotu zjistit, dokud ji nebude potřebovat. Program bude vyhodnocován tak, že si naalokuje místo pro hodnotu minima, vybuduje strom, jehož prvky budou odkazovat na toto místo, a na závěr do tohoto místa uloží spočítanou hodnotu (ve skutečnosti jsou věci ještě o něco složitější, ale pro hrubou představu tohle stačí).

```
replaceByMin1 :: Tree Int -> Tree Int
replaceByMin1 t =
  let (m, r) = doReplace m t  -- nahraď prvky v t odkazem na m a spočítej
   in r                       -- hodnotu m

doReplace :: Int -> Tree Int -> (Int, Tree Int)
doReplace w (Tree r sons) = (minimum (r : mins), Tree w rsons)
  where
    minsAndRsons = map (doReplace w) sons
                               -- seznam dvojic (minimum ze syna, syn s hodnotami
                               -- nahrazenými w)
    (mins, rsons) = unzip minsAndRsons
                               -- rozdělíme dvojice do dvou seznamů

unzip :: [(a,b)] -> ([a], [b]) -- rozděl seznam dvojic na dva seznamy
unzip [] = ([],[])              -- rozdělení prázdného seznamu
unzip ((a,b) : zb) = r
  where
    (ra,rb) = unzip zb          -- rozděl zbytek seznamu
    r = (a : ra, b : rb)        -- a přidej první prvky
```

15-2-1 Zatížení — Tomáš Valla

Zadání úlohy bylo formulované tak, aby napovědělo, že silniční síť mezi městy bude poměrně hustá a že úředníků je opravdu obrovské množství. Někteří řešitelé zaslali navíc i dobrá řešení jiných případů, za což mají samozřejmě pochvalu.

Jelikož úředníků je tak mnoho, potřebujeme co nejvíce urychlit operaci vyhledání délky nejkratší cesty mezi dvěma městy – vrcholy grafu. Nejlépe v konstatním čase tak, že si na začátku předpočítame do matice nejkratší vzdálenosti z každého města do každého. Budeme pak pro každého úředníka pouze sahat do tabulky a počítat z hodnot průměr. K výpočtu matice nejkratších vzdáleností si představíme neobyčejně elegantní Floyd-Warshallův algoritmus, který by neměl chybět v repertoáru žádného programátora.

Budeme si po řadě pro $k, i, j = 1, \ldots, n$ počítat čísla $T^k_{i,j}$, kde $T^k_{i,j}$ nadefinujeme jako „délka nejkratší cesty z vrcholu i do j, která používá pouze vrcholy s čísly $1, \ldots, k$ a i, j“. Na začátku tedy do $T^0_{i,j}$ načteme vzdálenosti vrcholů i a j, případně 0 pro $i = j$ a ostatní hodnoty nastavíme na nekonečno. Nechť nadále máme hodnoty T^l pro $l < k$ už spočítané. Když určujeme hodnotu $T^k_{i,j}$, bude jí menší z čísel $T^{k-1}_{i,j}$ a $T^{k-1}_{i,k} + T^{k-1}_{k,j}$ – tedy délky staré minimální cesty, kde ovšem není povolený vrchol k, a délky cesty procházející z vrcholu i do vrcholu j přes vrchol k. Poslední tabulka T^n tak bude obsahovat nejkratší cesty z každého vrcholu do každého.

Protože se při výpočtu díváme pouze na hodnoty z T^{k-1}, stačí používat dvě tabulky, T^{k-1} a aktuální T^k. Ona ale dokonce stačí tabulka jediná. Podíváme se na předpis

$$T^k_{i,j} = \min\{T^{k-1}_{i,j},\, T^{k-1}_{i,k} + T^{k-1}_{k,j}\}$$

a uvědomíme si, že nevadí, když je část tabulky už přepsaná novými hodnotami. V prvním případě určitě bude na přepisovaném místě ještě stará hodnota $T^{k-1}_{i,j}$. V případě druhém pak podle toho, jak jsme si nadefinovali hodnoty T, nutně platí $T^{k-1}_{i,k} = T^k_{i,k}$ a $T^{k-1}_{k,j} = T^k_{k,j}$.

Paměťová složitost tak bude kvadratická. Povšimněte si, že na realizaci algoritmu postačí pouhé tři vnořené for-cykly s podmínkou. Algoritmus tedy pracuje v kubickém čase vzhledem k počtu vrcholů. Do konečné časové složitosti musíme ještě započítat dotazy do tabulky (na což mimochodem spousta řešitelů zapomněla) – počet úředníků není konstanta, takže dostaneme čas $O(N^3 + K)$.

Poznámka pro koumáky: K výpočtu matice vzdáleností se dá také použít obměna Dijkstrova algoritmu, která při každém spuštění najde z jednoho vrcholu délky cest do všech ostatních. Když potom použijeme implementaci Dijkstry s Fibonacciho haldami – ďábelskou datovou strukturou, která algoritmus urychlí na $O(N \log N + M)$, dostame celkový čas $O(N^2 \log N + NM + K)$.

```
#include <stdio.h>

#define MAX 100
#define INFTY 999999

int T[MAX][MAX];
int N, M, K;

int main (void)
{
        int i, j, k, l;

        scanf ("%d %d", &N, &M);

        for (i=0; i<N; i++) for (j=0; j<N; j++) T[i][j] = INFTY;
        for (i=0; i<N; i++) T[i][i] = 0;
        for (i=0; i<M; i++) {
                scanf ("%d %d %d", &j, &k, &l);
                T[j][k] = T[k][j] = l;
        }
        for (k=0; k<N; k++)                     /* Floyd-Warshall */
                for (i=0; i<N; i++)
                        for (j=0; j<N; j++)
                                if (T[i][j] > T[i][k] + T[k][j])
                                        T[i][j] = T[i][k] + T[k][j];

        scanf ("%d", &K);
        l=0;
        for (i=0; i<K; i++) {
                scanf ("%d %d", &j, &k);
                l += T[j][k];                   /* nedosazitelne vrcholy nijak neresime */
        }
        printf ("%lf\n", (double)l/K);
        return 0;
}
```

15-2-2 Závorky — Josef Cibulka

Úloha patřila k těm jednodušším, pouze někteří si závorky zcela zbytečně načetli do paměti a pak je sekvenčně četli.

A takto mohlo správné řešení vypadat: Postupně budeme číst závorky. Je-li to levá, přidáme si ji na vrchol zásobníku. Načteme-li ale pravou, spárujeme ji s poslední ještě nespárovanou levou (ta je na vrcholu zásobníku) a tu ze zásobníku odstraníme. Pak všechny (mezi touto dvojicí budou mít příslušející) také mezi nimi (jinak jsme ji měli spárovat s právě spárovanou pravou). Platí-li toto pro všechny dvojice, pak pokud se druhý znak dvojice A vyskytuje za prvním znakem dvojice B, druhý znak B se nutně musí vyskytovat před druhým znakem A. Pokud navíc na konci nezbyde žádná nespárovaná závorka, jedná se o správné uzávorkování.

Pokud nastane případ, že je na vstupu) a nemáme žádnou nespárovanou levou závorku, pak od tohoto znaku nalevo je více pravých závorek než levých, a proto určitě nemůžeme všechny pravé spárovat a uzávorkování není korektní. Také pokud na konci zbyde nespárovaná (, uzávorkování není korektní. V ostatních případech algoritmus najde správné spárování.

Nám ale stačí rozhodnout, zda správné spárování existuje, takže místo zásobníku použijeme počitadlo nespárovaných (.

Časová složitost je lineární k délce posloupnosti, paměťová konstantní (respektive, jak jeden řešitel poznamenal, logaritmická – potřebujeme proměnnou dost velkou na to, aby se do ní vešel počet závorek).

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

FILE *vstup;
int c, zn;

int main (void)
{
    vstup=fopen ("zavorky.in", "r");
    c=0;
    while ( (zn=fgetc (vstup))!=EOF) {
      if (zn=='(') c++; else c--;
      if (c<0) break;
    }
    if (c==0) printf ("Spravne uzavorkovano.");
        else printf ("Spatne uzavorkovano.");
    fclose (vstup);
}
```

15-2-3 Defragmentace — Tomáš Vyskočil

Nejprve si řekneme základní myšlenku: Jakmile najdeme blok, budeme se ho snažit přesunout hned na správné místo. Pokud je toto místo již obsazeno, tak nejdříve přesuneme ten a tak dále (rekurzivně). Problémy nastanou až pokud se vytvoří cyklus, tedy pokud se na sebe začnou bloky navzájem odkazovat. Tento problém se řeší odsunutím jednoho bloku na volné místo a po přesunutí zbytku cyklu přesunem bloku z volného místa na cílové umístění. A to je celé. A nyní jak to napsat efektivně?

Označme si počet všech použitých bloků (bloků všech souborů dohromady) M. A budeme zde používat pole minimálně velikosti M, které bude na i-té pozici ukazovat, kde má prvek z i-té pozice být. Nejdříve se „zbavíme“ bloků, které jsou uloženy na pozici na disku větší než M. Protože žádný blok nemá skončit na pozici vetší než M, nemůže při přemísťování těchto bloků vzniknout cyklus. Snadno tedy tyto bloky přesuneme na jejich pozice.

V druhém kroku se pokusíme odstranit i zbylé „špatně“ umístěné bloky. Jak tyto bloky poznáme? Opět jednoduše nahlédneme, že pokud je blok špatně

umístěn, musí ležet na „cyklu z bloků“ – po přesunech z první části je totiž každý blok oblasti $1 \dots M$ obsazen (máme M bloků v oblasti velké M). Stačí tedy vždy pro každý špatně umístěný blok projít cyklus, posunout po něm bloky (k tomu potřebujeme jeden volný blok – použijeme blok $M + 1$, jehož existenci nám zaručuje zadání), a tím je všechny správně umístit.

Tedy dohromady první krok zabere $O(M)$ a druhý krok zabere opět $O(M)$, a tedy celková časová složitost je linearní a paměťová také.

```
#include <stdio.h>

#define MAX_USED 1024
#define MAX_BLOCK 1024

int used[MAX_USED], block[MAX_BLOCK];

void write_move (int from, int to)
{
        printf ("%d -> %d\n", from, to);
        used[from] = -1;
        used[to] = to;
}

void insert (int from, int to)
{
        if (used[to] != -1)
                insert (to, used[to]);
        write_move (from, to);
}

int main (void)
{
        int i, j, n, nd, k, sum;

        sum = 0;

        for (i=0; i<MAX_USED; i++){
                used[i] = -1;
        }

        scanf ("%d %d", &nd, &n);
        for (i=0; i<n; i++){
                scanf ("%d", &k);
                for (j=0; j<k; j++){
                        scanf ("%d", &block[sum]);
                        sum++;
                }
        }
        for (i=0; i<sum; i++){
                if (block[i] < sum)
                        used[block[i]] = i;
        }
        for (i=0; i<sum; i++){
```

```
                if (block[i] >= sum)
                        insert (block[i], i);
        }
        for (i=0; i<sum; i++){
                if (used[i] != i){
                        k = used[i];
                        write_move (i, sum);
                        insert (k, used[used[i]]);
                        write_move (sum, k);
                }
        }

        return 0;
}
```

15-2-4 Žabky — Dan Kráľ

Nejprve si rozmysleme, jaký musí být minimální možný počet skoků. Uvažme nějakou pozici žabek na značkách. Dvojici žabek nazveme *špatnou*, pokud pořadí těchto dvou žabek je opačné než má být ve výsledné pozici. Žabky mohou dělat skoky dvou typů – skoky „ob žabku“ a skoky na sousední políčko. Skok ob žabku mění počet špatných dvojic o jedna (buď zvyšuje nebo snižuje), zatímco skok na sousední políčko na tento počet nemá vliv. Protože v počáteční pozici je $n \cdot (n-1)/2$ špatných dvojic žabek a v koncové pozici naopak žádné špatné dvojice nejsou, musí žabky provést alespoň $n \cdot (n-1)/2$ skoků ob žabku.

Nyní odhadneme minimální nutný počet skoků na sousední políčko. Je-li n liché, pak žádné optimální řešení nemůže obsahovat za sebou více než $(n-1)/2$ skoků ob žabku (nakreslete si obrázek!). Tedy minimální počet skoků na sousední políčko v libovolném optimálním řešení musí být alespoň $n \cdot (n-1)/2 \cdot (2/(n-1)) - 1 = n - 1$. Rovnost v posledním případě nastane právě tehdy, když po každých $(n-1)/2$ skocích z celkových $n \cdot (n-1)/2$ skoků ob žabku následuje jeden skok na sousední políčko, ale v takovém případě nebude na závěr prázdné místo na jedné z krajních značek, což je v rozporu s požadavky zadání úlohy. Tedy můžeme uzavřít, že celkový počet skoků musí být alespoň $n \cdot (n-1)/2 + n = n \cdot (n+1)/2$ (je-li n liché).

Zbývá nám případ, kdy je n sudé. Podívejme se, jak skoky na sousední políčko rozdělují optimální řešení na bloky skoků ob žabku. Bloky skoků ob žabku, které jsou liché v pořadí, mohou mít délku nejvýše $n/2$, a ty, jejichž pořadí je sudé, délku $n/2 - 1$. Pokud $n \geq 4$, pak, podobně jako v předchozím odstavci, zjistíme, že počet skoků na sousední políčko musí být alespoň $n(n-1)/2 \cdot (2/(n-1)) - 1 = n - 1$. Tento odhad se opět nabývá pouze tehdy, pokud všechny „liché“ bloky mají délku $n/2$ a „sudé“ bloky délku $n/2-1$. V takovém případě ale na závěr nebude volná pozice na některé z krajních značek,

a tedy celkový počet skoků musí být i v tomto případě alespoň $n(n+1)/2$. Poznamenejme, že ve vyloučeném případu $n = 2$ je optimální počet skoků roven jedné.

Nyní si ukážeme, jak žabky mohou $n(n+1)/2$ skoky obrátit své pořadí (pokud $n \geq 3$). Podrobně rozebereme případ, kdy n je sudé, a případ, kdy n je liché jen stručně naznačíme. Nechť $n = 2k$ ($k \geq 2$). Počáteční pozice žabek je následující ($*$ označuje volnou pozici):

$$1\ 2\ 3\ 4\ 5\ 6 \cdots 2k-3\ 2k-2\ 2k-1\ 2k\ *$$

Nyní proveďme k skoků ob žabku a na závěr jeden skok na sousední políčko. Nové pořadí žabek tedy bude:

$$2\ *\ 1\ 4\ 3\ 6 \cdots 2k-5\ 2k-2\ 2k-3\ 2k\ 2k-1$$

Nyní proveďme $k-1$ skoků ob žabku opačným směrem a na závěr opět jeden skok na sousední políčko:

$$2\ 4\ 1\ 6\ 3\ 8 \cdots 2k-5\ 2k\ 2k-3\ 2k-1\ *$$

Žabka s číslem jedna je nyní na pozici, kde původně byla žabka číslo 3. Zopakujeme-li výše uvedený postup ještě jednou, dostane se žabka číslo 1 na pozici, kde byla žabka číslo 5 (po jednom zopakování je na této pozici žabka číslo 3). Po $k-1$ opakováních výše uvedeného postupu bude žabka s číslem 1 na pozici, kde byla původně žabka číslo $2k$. Podobně nahlédneme, že žabka s číslem 3 skončí na značce, kde původně byla žabka číslo $2k-2$, žabka s číslem 5 na značce žabky číslo $2k-4$, atd. Obecně žabka s číslem i stojí nyní na značce, kde byla žabka číslo $n-i+1$. Žabky tedy stojí v opačném pořadí, než jaké bylo na začátku. Celkový počet skoků, které provedly, je pak:

$$k \cdot (k+1+(k-1)+1) = k(2k+1) = \frac{n(n+1)}{2}.$$

V případě, že $n = 2k+1$ pro $k \geq 1$, vykonají žabky $(k+1)$-krát následující schéma skoků:

$$1\ 2\ 3\ 4\ 5\ 6 \cdots 2k-3\ 2k-2\ 2k-1\ 2k\ 2k+1\ *$$

$$*\ 1\ 3\ 2\ 5\ 4 \cdots 2k-3\ 2k-4\ 2k-1\ 2k-2\ 2k+1\ 2k$$

proložené k-krát následujícím schématem:

$$*\ 1\ 3\ 2\ 5\ 4 \cdots 2k-3\ 2k-4\ 2k-1\ 2k-2\ 2k+1\ 2k$$

$$3\ 1\ 5\ 2\ 7\ 4 \cdots 2k-1\ 2k-4\ 2k+1\ 2k-2\ 2k\ *$$

Podobně jako v minulém případě, kdy n bylo sudé, lze nahlédnout, že žabky budou na konci stát na značkách v opačném pořadí. Celkový počet skoků bude:

$$(k+1)\cdot(k+1)+k\cdot(k+1)=(2k+1)(k+1)=\frac{n(n+1)}{2}.$$

Program s časovou složitostí $O(n^2)$, kde n je počet žabek, pokud vypsání jednoho kroku zabere konstantní čas, je nyní již snadné vytvořit.

```
program zabky;
const MAX=100;
var zabky: array[1..MAX+1] of integer;     { pozice žabek, 0 = prázdné místo }
    volno: integer;
    n: integer;  { počet žabek }

procedure vypis;
var i:integer;
begin
  for i:=1 to n+1 do
    begin
      if zabky[i]=0 then
        write('*':2)
      else
        write(zabky[i]:2);
      if i<=n then
        write(' ')
      else
        writeln;
    end
end;

procedure init;
var i:integer;
begin
  for i:=1 to n do zabky[i]:=i;
  zabky[n+1]:=0;
  volno:=n+1;
  vypis;
end;

procedure skok( odkud: integer);
begin
  zabky[volno]:=zabky[odkud];
  zabky[odkud]:=0;
  volno:=odkud;
  vypis;
end;

var dozadu: boolean;
    faze: integer;
    pozice: integer;

begin
  readln(n);
  init;
  if n=1 then exit;
```

```
  if n=2 then
    begin
      skok(1);
      exit;
    end;
  dozadu:=true;
  for faze:=1 to n do
    if dozadu then
      begin
        pozice:=volno-2;
       repeat
         skok(pozice);
         pozice:=pozice-2;
       until pozice<1;
       if pozice=0 then skok(1) else skok(2);
       dozadu:=false
      end
    else
      begin
        pozice:=volno+2;
       repeat
         skok(pozice);
         pozice:=pozice+2;
       until pozice>n+1;
       skok(n+1);
       dozadu:=true
      end
end.
```

15-2-5 Haskell — Zdeněk Dvořák

Nejprve si nadefinujme třídu vypisovatelných typů:

```
class Printfovatelny x where
  format::x->Char       -- znak ve formatovacim retezci; parametr neni pouzit,
                        -- uvadime ho jen kvuli typove kontrole (parametr x
                      -- musi byt v typu kazde tridove funkce, jinak by
                      -- se nedalo poznat, kterou instanci pouzit)
  tiskni::x->String     -- prevede x na retezec

instance Printfovatelny Int where
  format = const 'd'
  tiskni = show

instance Printfovatelny Char where
  format = const 'c'
  tiskni c = [c]
```

Dále bychom chtěli, aby i `String` byl `Printfovatelny`. To má drobný háček v tom, že `String` není typ, ale pouze zkratka pro `[Char]`, a z technických důvodů pro něj není možné přímo definovat instanci. Musíme proto použít drobný trik:

```
class Znak x where
  mkChar::x->Char
  unChar::Char->x
```

```
instance Znak Char where
  mkChar = id
  unChar = id

instance (Znak a) => Printfovatelny [a] where
  format = const 's'
  tiskni = map mkChar
```

Podívejme se na typ požadované funkce `printf`:

```
printf "test"               --  printf :: String->String
printf "cislo %d" (7::Int)  --  printf :: String->(Int->String)
printf "%s %d" "a" (7::Int) --  printf :: String->(String->(Int->String))
```

Tedy chceme, aby typ `printf`u vypadal takto:

```
printf :: (Printf x) => String -> x
```

kde budeme chtít zaručit to, že

- `String` má vlastnost `Printf` a
- pro libovolný typ `a`, který chceme vypisovat, a `b` s vlastností `Printf`, i `a->b` má tuto vlastnost.

Samotná třída `Printf` vypadá takto:

```
class Printf x where
  printf':: String->
            -- uz zformatovana cast retezce (od konce k zacatku; String je
            -- vlastne spojovy seznam, pridavani na zacatek je mnohem rychlejsi)
           String->
            -- zbytek formatovaciho vzoru
           x
            -- a parametry

instance (Znak a) => Printf [a] where
            -- zadny parametr, pouze vystupni typ = String
  printf' konec fmt = map unChar (reverse konec ++ fmt)
            -- obrat jiz sformatovanou cast a pripoj za ni zbytek
           -- formatovaciho vzoru (pro uplnou korektnost bychom
           -- jeste meli testovat, zda neobsahuje dalsi %c, d nebo s)

instance (Printfovatelny a,Printf b) => Printf (a->b) where
  printf' konec "" x = error "Prilis mnoho parametru"
                -- dosel nam formatovaci vzor
  printf' konec ('%':c:rest) x =
                -- dorazili jsme k prislusnemu vzoru
    if format x == c
      then printf' (reverse (tiskni x) ++ konec) rest
                -- OK
      else error "Spatny typ"
                -- chyba
  printf' konec (h:t) x = printf' (h:konec) t x
                -- jenom pismeno
```

a vlastní funkci `printf` již napíšeme snadno:

```
printf :: (Printf x) => String -> x
printf fmt = printf' [] fmt
```

15-3-1 Potrubí — Jakub Bystroň

Řešení se k této úloze sešlo poměrně hodně. Většina z Vás ale použila algoritmus, který pro každou hranu (x, y) otestoval, zda se dá z vrcholu x do vrcholu y dostat bez použití této hrany, a pokud ano, tak tuto hranu z grafu vyhodil. Tento algoritmus ale nemusí vždy nalézt správné řešení. Příkladem může být např. úplný orientovaný graf na třech vrcholech.

A jak tedy správně? Podgrafu, který má stejné dosažitelnosti jako původní graf a navíc má minimální počet hran, říkejme MEG (minimální ekvivalentní podgraf). Úlohu si dále rozdělme na několik částí. Nalézt MEG na grafu, který je silně souvislý, tj. z každého vrcholu do každého vede orientovaná cesta, je NP-těžké. Zřejmě bychom tím vyřešili problém Hamiltonovské kružnice. Jednoduše to znamená, že zatím není známo řešení pracující s lepší časovou složitostí než exponenciální. Naopak nalézt MEG na acyklickém grafu je poměrně lehké. Idea řešení je tedy nasnadě. Vstupní graf si rozdělíme na silně souvislé komponenty. Na to existuje klasický Tarjanův algoritmus pracující v čase $O(N+M)$. Naleznete ho ve většině učebnic grafových algoritmů. Faktorgraf podle silně souvislých komponent je zřejmě acyklický (žádné dva vrcholy z různých komponent silné souvislosti neleží na kružnici). Máme-li tedy vrcholy grafu rozděleny do silně souvislých komponent, nalezneme MEG tohoto grafu tak, že použijeme metodu hrubé síly na jednotlivé komponenty, a hrany vedoucí mezi komponentami redukujeme pomocí polynomiálního algoritmu.

Ten pracuje tak, že z každého vrcholu nalezne vrcholy dostupné po orientované cestě, dále nalezne vrcholy dostupné proti orientaci a zruší všechny hrany vedoucí z vrcholů dosažených proti orientaci do vrcholů dosažených po orientaci. To lze jednoduše v čase $O(N \cdot (N + M))$. Navíc se dá poměrně jednoduše dokázat, že existuje právě jedna minimální redukce acyklického grafu. Viz např. kniha Jiří Demel: Grafy a jejich aplikace, a jiné. Paměťová složitost celého algoritmu je při vhodné implementaci $O(N + M)$.

15-3-2 Permutace — Martin Mareš

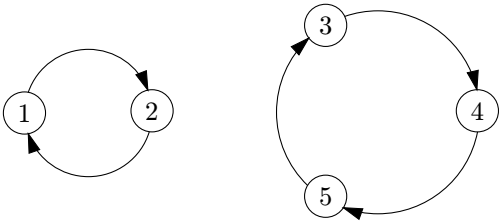
Podívejme se nejdříve, jak bude naše složená permutace P^k vypadat v nějakém konkrétním bodě i pro různá k, tedy vlastně na posloupnost

$$\begin{aligned} p_0 &= P^0[i] = P[i], \\ p_1 &= P^1[i] = P[P[i]] = P[p_0], \\ p_2 &= P^2[i] = P[P[P[i]]] = P[p_1], \\ &\vdots \\ p_k &= P^k[i] = P[p_{k-1}]. \end{aligned}$$

Možných hodnot p_k je pouze n, takže nejpozději po n krocích se nějaké číslo objeví podruhé, tedy $p_l = p_m$ pro nějaké $0 \leq l < m \leq n$. A jelikož každé číslo závisí pouze na čísle předchozím, začne se od tohoto bodu opakovat celá posloupnost. Snadno také nahlédneme, že číslo, které se nám zopakovalo jako první, musí být první člen posloupnosti – kdyby bylo $l > 0$, stačí se podívat na hodnoty p_{l-1} a p_{m-1}: buďto jsou stejné, a pak jsme objevili ještě časnější opakování, nebo jsou různé, ale pak nám P zobrazuje dvě různá čísla na jedno, tudíž to není permutace.

Posloupnost složení tedy musí tvořit cyklus délky m, na počest jeho počátečního prvku mu budeme říkat cyklus i-čkový. Navíc tento cyklus můžeme velice jednoduše najít: stačí postupně počítat $p_0 = P[i]$, $p_1 = P[p_0]$, ... a značkovat si, které hodnoty už nastaly. Takto v lineárním čase celý cyklus projdeme a pak už můžeme v konstantním čase spočítat $P^k[i] = p_k = p_{k \bmod m}$. A když to provedeme pro každé i, získáme řešení naší úložky v čase $O(n^2)$.

A to je všechno? Kdepak, jde to lineárně. Stačí k tomu jen málo – uvědomit si, že pro libovolné j ležící na i-čkovém cyklu (tedy $j = p_z$ pro nějaké z) je j-čkový cyklus stejný jako i-čkový, jen „o z otočený“, takže $P^k[j] = P^{k+z}[i] = p_{k+z}$. Náš program si tedy nejdříve permutaci rozloží na cykly (najde první cyklus, pak vezme nejbližší neoznačkovaný prvek a začne od něj hledat další cyklus a tak dále); pro permutaci $(2, 1, 4, 5, 3)$ ze zadání to například dopadne takto:



Pak pro každý cyklus projde jeho prvky a spočítá pro ně prvky „o k dál“ na cyklu a ty prohlásí za výsledek (a bude to pravda). To všechno stihne v čase lineárním s délkou cyklu, takže celkem lineárně se součtem délek všech cyklů, což je ovšem přesně n. Paměti stačí také jen $O(n)$.

Aby program (který se podle našeho algoritmu řídí takříkaje do slova a do písmene) nevypadal tak fádně, napsali jsme ho podle nové normy Céčka zvané `C99`, posuďte sami, oč se v něm programuje příjemněji.

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

int main (void)
{
        int n;                          /* Délka permutace */
        scanf ("%d", &n);
```

```
        int perm[n+1];                          /* Permutace */
        for (int i=1; i<=n; i++)
                scanf ("%d", &perm[i]);
        int k;                                  /* Kolikrát složit */
        scanf ("%d", &k);
        char znam[n+1];                         /* Na kterých pozicích jsme už byli */
        memset (znam, 0, sizeof (znam));
        for (int i=1; i<=n; i++)
                if (!znam[i]) {
                        int j = i;
                        int cyklus[n]; /* Zde si pamatujeme aktuální cyklus... */
                        int l = 0;      /* ... a jeho délku */
                        do {            /* Najdeme celý cyklus */
                                cyklus[l++] = j;
                                znam[j] = 1;
                                j = perm[j];
                        } while (j != i);
                        for (j=0; j<l; j++)   /* A znovu ho zapíšeme posunutý
                                                 o k pozic */
                                perm[cyklus[j]] = cyklus[ (j+k)%l];
                }
        for (int i=1; i<=n; i++)                /* Vypíšeme výsledek */
                printf ("%d␣", perm[i]);
        putchar ('\n');
        return 0;
}
```

15-3-3 Tajemný obraz — Peter Bella

Drvivá väčšina riešení načítala vstup do dvojrozmerného poľa, potom troma vnorenými cyklami prešla cez všetky trojice bodiek, zarátala každú trojicu tvoriacu jednofarebný trojuholník. Toto viedlo k algoritmu s kubickou zložitosťou, t.j. N^3. Našlo sa jedno exotické riešenie v zložitosti $N^{log_2 7}$.

Hŕstka riešiteľov pochopila, že ten príklad asi taký jednoduchý nebude, a stvorila riešenie s kvadratickou časovou zložitosťou – $O(N^2)$ a lineárnou pamäťovou zložitosťou – $O(N)$, podľa ktorého je napísané aj vzorové riešenie.

A nyní vzorové riešenie: Nebudeme počítať, koľko je na obrázku jednofarebných trojuholníkov, ale naopak, koľko je na obrázku pestrých trojuholníkov, t.j. takých trojuholníkov, ktorých strany nie sú rovnakej farby. Ak si označíme počet pestrých trojuholníkov K, tak potom jednofarebných trojuholníkov bude:

$$\binom{N}{3} - K = \frac{N(N-1)(N-2)}{6} - K.$$

Ako ale vypočítať K? Uvážme jednu bodku na obrázku s číslom i. Z nej vychádzajú nejaké modré úsečky a nejaké červené úsečky.

Vezmime nejakú modrú úsečku a nejakú červenú úsečku, obe vychádzajúce z našej bodky. Evidentne naša bodka spolu s koncovými bodmi modrej a červenej úsečky tvoria pestrý trojuholník.

Označme c_i počet červených úsečiek vedúcich z našej bodky, potom modrých úsečiek vychádzajúcich z našej bodky bude $N-1-c_i$. Keďže máme na výber z c_i červených a nezávisle na tom z $N-1-c_i$ modrých úsečiek, takýchto pestrých trojuholníkov tam potom bude $c_i(N-1-c_i)$.

Spočítajme takéto súčiny pre všetky bodky: $P = c_1(N-1-c_1) + c_2(N-1-c_2) + \ldots + c_N(N-1-c_N)$. Koľkokrát je v súčte P zarátaný nejaký pestrý trojuholník?

Práve dvakrát! Stačí si uvedomiť, ako vyzerá pestrý trojuholník: Má dve strany jednej farby a jednu stranu druhej farby. A teda má jeden vrchol, kde sa stretávajú strany rovnakej farby, a dva vrcholy, kde sa stretáva červená s modrou stranou (odporúčam nakresliť si to). V práve týchto dvoch vrcholoch bude zarátaný pestrý trojuholník do súčtu P, a preto $K = \frac{P}{2}$.

Napísať program je už jednoduché. Pre každú bodku si budeme pamätať v jednorozmernom poli hodnoty c_i. Na začiatku sú hodnoty c_i nulové. Pri načítaní červenej úsečky medzi medzi bodkami x,y zvýšime hodnoty c_x a c_y o jedna. Nakoniec vyrátame súčet P. Jednofarebných trojuholníkov bude

$$\frac{N(N-1)(N-2)}{6} - \frac{P}{2}.$$

Časová zložitosť je $O(M)$, teda lineárna od počtu červených hrán. Pamäťová zložitosť je lineárna – $O(N)$.

15-3-4 Výsadek — Tomáš Vyskočil

Jak to vlastně dopadlo s vrchním velením krtků? Podařilo se jim vykrtincovat zahradu strýčka Pompa? Ne, všichni by to stihli, rozhodně ne dříve než by si byl strýček koupil nějaký přípravek na hubení krtků, ale našlo se i něco rychlejších řešitelů. Většina z vás řešila problém lineárně, což sice není úplně nejpomalejší, ale protože v zadání byl nabídnut polynomiální čas na předpočítaní, není toto řešení zcela optimální. Jak tedy řešit? Tento problém šlo řešit v čase $O(\log n)$ na dotaz a s časem $O(n^2)$ na předpočítání, kde n je velikost vstupu (počet vrcholů n-úhelníka).

Jak pracuje algoritmus na předzpracování? Nejprve si vytvoříme seznam hran na obvodu n-úhelníka. Do seznamu ale nezařazujeme hrany, jejichž koncové body mají stejnou y-ovou souřadnici. Dále setřídíme body podle y-ové souřadnice a ještě se zbavíme vrcholů se stejnou y-ovou souřadnicí (v poli vrcholů máme pouze vrcholy s různými y-vými hodnotami). A nyní si setřídíme hrany tak, aby pokud mají dvě hrany společnou y-ovou souřadnici, tak byly

setříděné podle pořadí x-ových souřadnic na společné y-ové souřadnici (protože se hrany nekříží, je tento krok jednoznačný). Pokud hrany společnou y-ovou souřadnici nemají, tak nám na jejich pořadí nezáleží. Rychlé setřídění hran dle těchto pravidel se dá poměrně snadno provést pomocí metody zvané *topologické třídění*. Výklad této metody si můžete prohlédnout například v úloze *11-1-1*. Po setřídění hran si vytvoříme „pásečky“. Ty tvoříme tak, že si v utříděné posloupnosti vrcholů bereme postupně dva následující vrcholy. Ty nám jednoznačně určují páseček v y-ovém směru (díky vyřazení vrcholů se stejnou y-ovou souřadnicí nemáme žádný páseček s nulovou šířkou). Pro každý páseček si vyhradíme pole (případně spojový seznam) a do něho postupně vkládáme hrany, které do něj zasahují (postupně, znamená, že zachovávám jejich uspořádání). A to už je celé předzpracování.

Jak že to vypadá dotaz? Na vstupu dostaneme bod. Binárním vyhledáním zjistíme, v jakém je pásečku. Stejným postupem zjistíme, mezi kterými hranami v pásečku se vyskytuje. A nyní stačí zjistit paritu hrany v seznamu, neboli zda je pořadí hrany před vrcholem sudé či liché číslo. Je zřejmé, že vždy při přechodu hrany se musí střídat stav uvnitř a venku zahrady.

Jaká je složitost předzpracování? Na setřídění potřebujeme $O(n \cdot \log n)$, ale to zde nehraje roli. Složitost vybrání hran pro pásečky je pro každý páseček $O(n)$ a pásečků je až $O(n)$, a tedy celková složitost předzpracování je $O(n^2)$. Složitost dotazu závisí pouze na dvojím binárním vyhledání, které má složitost $O(\log n)$.

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

#define MAX_N 1024
#define INF 1000000

typedef struct{
        float x, y;
} t_point;

typedef struct{
        t_point max, min;
        float x, y;
} t_edge;

int n, ne, nb[MAX_N];
t_point p[MAX_N];
t_edge e[MAX_N];
int b[MAX_N][MAX_N];
float minDiff;

int return_value (float a)
{
        if (a > 0)
                return 1;
```

```
        if (a < 0)
                return -1;
        return 0;
}
int cmp (const void *a, const void *b)
{
        t_point k, l;
        k = *(t_point *)a;
        l = *(t_point *)b;
        return return_value (k.y - l.y);
}
int cmp_edge (t_edge *p, t_edge *q)
{
        float t1, t2;
        if (p->min.y > q->max.y || p->max.y < q->min.y){
                return 0;
        } else{
                if (p->max.y > q->max.y){
                        t1 = ( (q->max.y - minDiff/2) - p->min.y) / p->y;
                        t2 = ( (q->max.y - minDiff/2) - q->min.y) / q->y;
                        return return_value (p->x*t1 + p->min.x - (q->x*t2
                            + q->min.x));
                } else{
                        t1 = ( (p->max.y - minDiff/2) - q->min.y) / q->y;
                        t2 = ( (p->max.y - minDiff/2) - p->min.y) / p->y;
                        return return_value (p->x*t2 + p->min.x - (q->x*t1
                            + q->min.x));
                }
        }
}
void sort_edges (void)
{
        int less_cnt[MAX_N];
        int greater_cnt[MAX_N];
        int greater[MAX_N][MAX_N];
        int smallest[MAX_N];
        int i, j, smallest_cnt = 0;
        t_edge sortede[MAX_N];
        for (i = 0; i < ne; i++) {
                less_cnt[i] = 0;
                for (j = 0; j < ne; j++)
                        if (j != i && cmp_edge (&e[j], &e[i]) < 0) {
                                less_cnt[i]++;
                                greater[j][greater_cnt[j]++] = i;
                        }
                if (!less_cnt[i])
                        smallest[smallest_cnt++] = i;
        }
```

```
        for (i = 0; i < smallest_cnt; i++) {
                sortede[i] = e[smallest[i]];
                for (j = 0; j < greater_cnt[smallest[i]]; j++) {
                        less_cnt[greater[smallest[i]][j]]--;
                        if (less_cnt[greater[smallest[i]][j]] == 0)
                                smallest[smallest_cnt++] = grea-
                                    ter[smallest[i]][j];
                }
        }

        for (i = 0; i < ne; i++)
                e[i] = sortede[i];
}
int bs (float y, int z, int k)
{
        int i = (z + k) / 2;

        if (k < 0 || z >= n)
                return -1;
        if (p[i].y > y)
                return bs (y, z, i-1);
        if (i+1 < n && p[i+1].y <= y)
                return bs (y, i+1, k);

        return i;
}
int bsearch_edge (float x, float y, int pas, int z, int k)
{
        t_edge e1, e2;
        float t1, t2;
        int i = (z + k) / 2;

        if (i+1 >= nb[pas])
                return 0;
        e1 = e[b[pas][i]];
        e2 = e[b[pas][i+1]];

        t1 = (y - e1.min.y) / e1.y;
        t2 = (y - e2.min.y) / e2.y;

        if (k < 0 || z >= n)
                return 0;
        if (e1.x*t1 + e1.min.x > x)
                return bsearch_edge (x, y, pas, z, i-1);
        if (e2.x*t2 + e2.min.x < x)
                return bsearch_edge (x, y, pas, i+1, k);

        return ! (i % 2);
}
int in (float x, float y)
{
        int pas;
```

```
        pas = bs (y, 0, n - 1);
        if (pas >= 0 && bsearch_edge (x, y, pas, 0, nb[pas] - 1)){
                return 1;
        } else
                return 0;
}
int main (void)
{
        int i, i1, i2, j, m, max, min;
        float x, y, medy;

        scanf ("%d", &n);
        for (i=0; i<n; i++){
                scanf ("%f %f", &p[i].x, &p[i].y);
        }
        for (i1=0, ne=0; i1<n; i1++){
                i2 = (i1+1) % n;
                if (p[i1].y != p[i2].y){
                        if (p[i1].y > p[i2].y){
                                max = i1;
                                min = i2;
                        } else{
                                max = i2;
                                min = i1;
                        }
                        e[ne].max = p[max];
                        e[ne].min = p[min];
                        e[ne].x = p[max].x - p[min].x;
                        e[ne++].y = p[max].y - p[min].y;
                }
        }
        qsort (p, n, sizeof (t_point), cmp);

        /* ruseni spatnych vrcholu */
        for (i=0, j=0; i<n; i++){
                if (i+1 >= n || p[i].y != p[i+1].y){
                        p[j++] = p[i];
                }
        }
        n = j;
        minDiff = INF;
        for (i=0; i<n-1; i++)
                if (p[i+1].y - p[i].y < minDiff)
                        minDiff = p[i+1].y - p[i].y;

        sort_edges ();

        /* prirazeni pole hran pasum */
        for (i=0; i < n - 1; i++){
                medy = (p[i].y + p[i+1].y) / 2;    /* referencni ypsilon */
```

```
                nb[i] = 0;
                for (j=0; j<ne; j++){
                        if (medy > e[j].min.y && medy < e[j].max.y){
                                b[i][nb[i]++] = j;
                        }
                }
        }

        scanf ("%d", &m);
        for (i=0; i<m; i++){
                scanf ("%f %f", &x, &y);
                if (in (x, y))
                        puts ("in");
                else
                        puts ("out");
        }

        return 0;
}
```

15-3-5 Haskell — Zdeněk Dvořák

Řešení této úložky je poměrně přímočaré, bez nějakých složitějších myšlenek. Nejprve je potřeba načíst vstup; to zajišťuje funkce `runCgi`. Poté musíme přeskočit daný počet volání `cgiPage` a nechat následující volání této funkce vrátit následující dotaz. Toto zajišťuje monáda `CGI` ve spolupráci s `cgiPage`. `CGI` jen přenáší aktuální stav výpočtu (funguje na podobném principu jako v zadání popsaná monáda `IO`) mezi jednotlivými voláními `cgiPage`. `cgiPage` zvýší počítadlo aktuálního kroku a v případě, že dosáhlo hledané hodnoty, do stavu zapíše zadaný dotaz.

Navíc `cgiPage` vždy musí vrátit hodnotu, kterou jsme zadali jako odpověď. Tuto hodnotu najde v tabulce, kterou také přenášíme ve stavu výpočtu (pokud tato hodnota ještě není známa, vracíme `undefined` – to nevadí, takovou hodnotu se nikdo nepokusí vyhodnotit).

```
module CGI where

import List
  -- knihovna funkci pro manipulaci se seznamy
import Maybe
  -- knihovna uzitecnych funkci nad typem Maybe a

data State = State Int Int String [(Int,String)]
  -- State p s reply vars
  --   p      ... pocet jiz provedenych kroku
  --   s      ... pocet kroku, ktere chceme provest
  --   reply  ... odpoved
  --   vars   ... seznam dvojic (cislo dotazu, odpoved)

data CGI a = CGI (State -> (State, a))
```

```
instance Monad CGI where
  CGI f >>= g' = CGI h
    where
      h state
        -- jen kvuli efektivite (aby se zbytecne nezkousely provadet
        -- nasledujici akce):
        | p > s = (state', undefined)
        -- jinak sami o sobe nedelame nic, jen predavame stav
        | otherwise = g state'
        where
          (state', x) = f state
          State p s reply vars = state'
          CGI g = g' x
  return x = CGI (\state -> (state, x))

cgiPage :: String -> CGI String
cgiPage dotaz = CGI f
  where
    f (State p s reply vars)
      -- pokud to neni aktualni krok, jenom vratit prislusnou hodnotu
      | p < s = (State (p + 1) s reply vars, fromJust $ lookup p vars)
      -- je-li to aktualni krok, zapiseme dotaz
      | p == s = (State (p + 1) s dotaz vars, undefined)
      -- sem bychom se nemeli dostat, viz definice >>=
      | otherwise = error "Chyba"

runCgi :: CGI () -> String -> String -> String
runCgi (CGI cgi) stav odpoved =
  if reply == ""
    then ""
    else "Stav: " ++ formatState next vars ++ "; " ++ reply ++ "?"
  where
    -- zavolej funkci z cgi
    (State next _ reply _, _) = cgi (State 0 krok "" vars)

    -- s hodnotami naparsovanymi ze stavu
    (krok, vars) =
      if stav == ""
        then (0, [])
        else (qn + 1, map splitVarDef vardefs ++ [(qn, odpoved)])
    (q : vardefs) = splitOn ',' stav
    qn = read (splitOn '=' q !! 1)
    splitVarDef vardef =
      let ['o' : o, val] = splitOn '=' vardef
       in (read o - 1, val)
    splitOn x lst
      | zbytek == "" = [zacatek]
      | otherwise = zacatek : splitOn x (tail zbytek)
      where
        (zacatek, zbytek) = span (/= x) lst

    -- formatovani vystupu
    formatState next vars =
      "q=" ++ show (next - 1) ++ concatMap formatVar vars
    formatVar (n, val) = ",o" ++ show (n + 1) ++ "=" ++ val

cgi :: CGI ()
```

```
cgi =
 do
   r1 <- cgiPage "Dotaz 1"
   r2 <- cgiPage "Dotaz 2"
   r3 <- cgiPage ("Dotaz 3 (vase predchozi odpoved byla " ++ r2)
   return ()
```

15-4-1 Archiv — Martin Mareš

Nejdříve vyvraťme několik mýtů, jimiž byla některá z vašich řešení obestřena:

- Časová složitost bublinkového třídění (tj. algoritmů typu „najdu dvojici po sobě jdoucích prvků, která je ve špatném pořadí, prohodím je; to dělám tak dlouho, než je posloupnost setříděná“) není logaritmická (to by znamenalo $O(\log n)$), ani není $O(n \log n)$, je totiž kvadratická – $O(n^2)$.
- Ani QuickSort (alespoň ve formě, ve které se běžně programuje) nemá složitost $O(n \log n)$ – v nejhorším případě (někdy to dokonce bývají setříděné nebo opačně setříděné posloupnosti) dosahuje $O(n^2)$; hezkou časovou složitost $O(n \log n)$ má pouze v průměrném případě.
- Špatně zatříděné prvky nejsou nutně ty, které jsou menší než předchozí prvek – například pro posloupnost typu $1, N, 2, 3, \ldots, N-2, N-1$ bychom totiž za špatně zatříděné prohlásili prvky $2 \ldots N-1$, i když ve skutečnosti je špatně zatříděn pouze prvek N.

Tak to by bylo, s čistou myslí se teď pustíme do našeho vlastního řešení. Nejdříve si všimněme, že kdybychom věděli, kterých k spisů jsou ty špatně zatříděné, stačilo by je z posloupnosti odstranit (takže by zbyla setříděná posloupnost), zvlášť si je setřídit (to zvládneme v čase $O(k \cdot \log k)$ například tříděním slévání čili MergeSortem) a poté obě setříděné posloupnosti slít dohromady (to dokážeme v lineárním čase, vlastně to je dokonce podprogram MergeSortu).

Jak jen ale ty potvorky toulavé špatně zatříděné najít? Inu, příklad v úvodu naznačuje, že to nebude jednoduché. Proto to vyřešíme drobným úskokem: jakmile narazíme na dvojici sousedních prvků ve špatném pořadí (budeme jí říkat špatná dvojice), tak ji odstraníme a budeme to opakovat tak dlouho, dokud takové dvojice existují. Tím určitě v lineárním čase odstraníme všechny špatné prvky a možná i některé dobré, ale těch nebude mnoho: z každé špatné dvojice je alespoň jeden prvek špatný (protože každé dva dobré prvky jsou určitě ve správném pořadí), celkově tedy odstraníme maximálně $2k$ prvků a původní odhad časové složitosti bude nadále platit.

Časová složitost tedy bude $O(n+k \cdot \log k)$ a prostorová $O(n)$. A asymptoticky lépe to určitě vyřešit nelze, protože oněch k zatříděných prvků potřebujeme najít (to nelze rychleji než lineárně k n) a rovněž setřídit, což pro změnu nelze rychleji než pomocí $k \log k$ operací.

Jako zákusek servírujeme ještě program v Céčku umíchaný přesně podle našeho algoritmického receptu.

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

#define MAX 1000

long ok[MAX], bad[MAX], res[MAX];

void merge (long *x, long nx, long *y, long ny, long *z)
{                                          /* Slévání dvou setříděných posloupností */
        while (nx || ny) {
                if (!ny || (nx && *x < *y))
                        *z++ = *x++, nx--;
                else
                        *z++ = *y++, ny--;
        }
}

void mergesort (long *x, long n)
{                                          /* Třidění MergeSortem */
        long h = n/2;
        if (!h) return;
        mergesort (x, h);
        mergesort (x+h, n-h);
        merge (x, h, x+h, n-h, res);       /* res použijeme jako pomocné pole */
        memcpy (x, res, n*sizeof (x[0]));
}

int main (void)
{
        long N, i, x;
        long nok=0, nbad=0;

        scanf ("%ld", &N);                 /* Popelka přebírá hrách... */
        for (i=0; i<N; i++) {
                scanf ("%ld", &x);
                if (!nok || x > ok[nok-1])  /* Nový prvek vypadá správně */
                        ok[nok++] = x;
                else {                     /* Pokles ⇒ pryč s oběma */
                        bad[nbad++] = ok[--nok];
                        bad[nbad++] = x;
                }
        }
        mergesort (bad, nbad);             /* Setřídíme špatné */
        merge (bad, nbad, ok, nok, res);   /* Přilijeme dobré */
        for (i=0; i<N; i++)                /* Hotovo */
                printf ("%ld%c", res[i], (i==N-1) ? '\n': ' ');
        return 0;
}
```

15-4-2 Permutace podruhé — Dan Kráľ

Nejdříve si definujme k-tou mocninu permutace P, a to rekurzivně následujícím předpisem: $P^1[i] = P[i]$ a $P^{k+1}[i] = P^k[i]$. Cyklem permutace P nazveme takovou posloupnost i, $P^1[i]$, $P^2[i]$, …, $P^{k-1}[i]$, že $P^k[i] = i$ a k je nejmenší číslo s touto vlastností. Číslo k je pak délka cyklu i, $P^1[i]$, $P^2[i]$, …, $P^{k-1}[i]$. Zřejmě každý index i leží v právě jednom cyklu permutace P.

Podívejme se nyní, jak vypadá druhá mocnina permutace P. Cyklus liché délky $i_1, i_2, \ldots, i_k$ permutace P bude v permutaci P^2 odpovídat cyklu $i_1, i_3, i_5, \ldots, i_k, i_2, \ldots, i_{k-1}$. Na druhou stranu, z cyklu $i_1, i_2, \ldots, i_k$ permutace P sudé délky vzniknou umocněním dva cykly poloviční délky: $i_1, i_3, \ldots, i_{k-1}$ a $i_2, i_4, \ldots, i_k$. Obecně tedy, pokud Q je druhou mocninou nějaké permutace P, pak počet jejích cyklů sudé délky je sudý (sudé cykly mohou vznikat jen štěpením cyklů dvojnásobné délky).

Předpokládejme tedy, že máme dánu permutaci Q a zajímá nás, kolik existuje permutací P takových, že $P^2 = Q$: Zvolme jednu pevnou délku cyklu k. Je-li k sudé a Q má K cyklů délky k (K musí být nutně sudé), pak tyto cykly se mohou zkombinovat $K!\,/\,2^{K/2}(K/2)!$ způsoby do dvojic. Navíc máme-li dva cykly $i_1, \ldots, i_k$ a $j_1, \ldots, j_k$, pak existuje k různých cyklů délky $2k$, ze kterých tyto dva cykly mohly vzniknout:

$$\begin{array}{cccccccc} i_1 & j_1 & i_2 & j_2 & \ldots & i_k & j_k \\ i_1 & j_2 & i_2 & j_3 & \ldots & i_k & j_1 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ i_1 & j_k & i_2 & j_1 & \ldots & i_k & j_{k-1}. \end{array}$$

Celkem tedy existuje $K!k^{K/2}\,/\,(K/2)!2^{K/2}$ konfigurací cyklů délky $2k$ takových, že jejich umocněním vznikne právě uvažovaná množina cyklů délky k.

Je-li k liché, je situace o něco složitější, neboť kromě toho, že cyklus liché délky může vzniknout rozpadem cyklu dvojnásobné délky, může také vzniknout z jednoho cyklu téže délky. Označme si nyní $\mu_k(K)$ počet konfigurací, ze kterých může vzniknout K zadaných cyklů délky k. Zřejmě $\mu_k(1) = 1$ a $\mu_k(2) = k + 1$ (buď oba cykly vznikly rozpadem většího cyklu, to nám dává k způsobů, anebo ze dvou cyklů téže délky, tu máme jediný způsob). Obecně pak máme následující rekurzivní vztah pro $K \geq 2$:

$$\mu_k(K) = \mu_k(K-1) + k(K-1)\mu_k(K-2).$$

Buď uvažovaný cyklus nevznikl rozpadem z většího cyklu (první sčítanec) anebo vznikl, a v tom případě je $K - 1$ možnosti, který jiný cyklus vznikl z téhož velkého cyklu, a k možností, jak byly tyto dva cykly do sebe „zakousnuty“. Mimochodem, obdobně můžeme pro k sudé psát: $\mu_k(1) = 0$, $\mu_k(2) = k$ a $\mu_k(K) = k(K-1)\mu_k(K-2)$ pro $K \geq 2$.

Počet permutací P takových, že P^2 je permutace Q, se určí vynásobením počtu konfigurací pro všechny možné délky cyklů.

Navrhnout algoritmus pracující v čase $O(N)$, kde N je velikost permutace, je nyní již snadné. Nejprve spočítáme počet cyklů jednotlivých délek: Nejdříve všechny indexy 1 až N označíme. Pak procházíme postupně indexy od 1 do N. Pokud narazíme na označený index, určíme délku jeho cyklu a všechny prvky tohoto cyklu odznačíme. Časová složitost této fáze algoritmu je úměrná součtu délek všech cyklů permutace a délce permutace, tedy je lineární. V druhé fázi pak spočítáme čísla $\mu_k(K)$ pro všechny délky k cyklů zadané permutace a jejich součin. Výpočet čísla $\mu_k(K)$ lze provést rekurzivně v čase $O(K)$, a tedy časovou složitost druhé fáze algoritmu lze odhadnout počtem cyklů permutace, a tedy je opět $O(N)$. Paměťová složitost algoritmu je též $O(N)$, neboť potřebujeme pole na uložení permutace.

```
program odmocnina;
const MAX=100;
var permutace:array[1..MAX] of integer;
    cykly:array[1..MAX] of integer;
    N:integer;
procedure nacti;
  { Načte vstupní permutaci }
  var i:integer;
begin
  readln(N);
  for i:=1 to N do read(permutace[i]);
end;
procedure urci_cykly;
  { Určí počty cyklů jednotlivých délek}
  var oznac:array[1..MAX] of boolean;
  var i,j,k:integer;
begin
  for i:=1 to N do
    begin
      oznac[i]:=true;
      cykly[i]:=0;
    end;
  for i:=1 to N do
    if oznac[i] then
      begin
       j:=i; k:=0;
       while oznac[j] do
         begin
            oznac[j]:=false;
            j:=permutace[j];
           inc(k)
         end;
       inc(cykly[k])
      end;
end;
function mu(delka:integer; pocet:integer):longint;
  { Spočítá z kolika konfiguraci lze vytvorit _pocet_ cyklů délky _delka_ }
begin
  if (delka mod 2=0) then
```

```
    case pocet of
    0: mu:=1;
    1: mu:=0;
    2: mu:=delka;
    else
      mu:=delka*(pocet-1)*mu(delka,pocet-2);
    end
  else
    case pocet of
    0: mu:=1;
    1: mu:=1;
    2: mu:=delka+1;
    else
      mu:=mu(delka,pocet-1)+delka*(pocet-1)*mu(delka,pocet-2);
    end
end;
function pocet:longint;
  { Určí počet odmocnin zadané permutace }
var i:integer;
    K:longint;
begin
  K:=1;
  for i:=1 to N do K:=K*mu(i,cykly[i]);
  pocet:=K
end;

begin
  nacti;
  urci_cykly;
  writeln(pocet);
end.
```

15-4-3 Koberec — Pavel Šanda

Naisrep seděl na balkóně své vily a znuděně pozoroval cvrkot městečka pod sebou. Od doby poslední molové kalamity v dílně ho šití koberců značně stravovalo – občas jsou prostě roky, kdy se vám nechce nic dělat. Nápad na nový vzor se ukázal býti triviálně řešitelným pomocí rekurze a krom maličkého detailu představovala četba návrhů na řešení vzoru koberců cvičení v gramatice rodného jazyka. Z letargie snícího odpoledne ho probudila až želva maně užírající nohu židle, v níž trůnil. Ve vědomí se mu objevil obraz Achilea nemilosrdně stíhajícího požíračku, jenž briskním způsobem vyřešil i ten poslední detail – šlo totiž o to, zda-li postupně rostoucí rohy vnořených čtverců nemohou přesáhnout kvadrant, v němž sídlili jeho předchůdci. Klíčem mu byl vzorec pro součet geometrické řady – podobně jako želva, jíž se nikdy nepovede půlením vzdáleností dostat až na konec dráhy, čtverce nikdy nepřesáhnou svůj kvadrant (*) (v tomto případě se v nekonečnu přesně dotknou jedné z os).
A jak tedy vypadá rekurze v prvním plánu ?

0) Situace: Počátek souřadného systému je uprostřed největšího čtverce. Začínáme s délkou (strany) čtverce n a rekurzí hloubky 0.

1) V i-tém rekurzivním zanoření ověříme, jestli bod leží uvnitř čtverce délky $n/2^i$. Pakliže nikoliv, jdeme rovnou do další rekurze, bez inkrementace čverců obsahujících bod.
2) Zjistíme, ve kterém kvadrantu se nachází zkoumaný bod a rekurzivně se pustíme do tohoto kvadrantu s tím, že posuneme počátek souřadného systému doprostřed centrálního čtverce daného kvadrantu. Tím jsme v situaci analogické k 0.

Pozorování: Díky (*) vím, že žádný z vnořených čtverců sousedního kvadrantu nezasáhne do zkoumaného, a tedy jednoznačnou volbou v 2) nepřijdu o další řešení. Dále si všimněme, že volbou pouze jediného z kvadrantů, o který se kdy budu zajímat, je možné místo rekurze s paměťovou složitostí $O(\log n)$ použít while cyklus, mající paměť $O(1)$.

Druhé pozorování: struktura rozmístění čtverců je symetrická podle obou os. Místo složitého rozhodování, ve kterém kvadrantu se zrovna šťourat, překlopíme hledaný bod vždy do prvního kvadrantu. Tím jsme se vyhnuli i diskusi, kam s bodem o souřadnicích [0,0] – šlo by případně v tomto okamžiku výpočet zarazit (žádný další čtverec se do nuly nedostane).

Časová složitost odpovídá hloubce zanoření rekurze neboli počtu vydělení, kterým se dostanu ke dvojkovému čtverci, neboli logaritmu n. Některým z vás činilo obtíž určení paměťové složitosti v případě rekurze – neuvědomili si totiž, že každé nové zanoření do rekurzivní fce vezme dalších $O(1)$ na proměnné uvnitř zásobníku.

Kvůli další skupině řešitelů si neodpustím osvětovou poznámku, že platí $\log(c_1 \cdot n) = \log c_1 + \log n = c_2 + \log n$ a $\log_a n = \log n / \log a = c \cdot \log n$ – tj. ve všech těchto případech lze psát $O(\log n)$ bez diskusí, co s konstantami.

```
#include <stdio.h>

int main (void) {
  int n, k, x, y, i=0;                    /* koberec, čtverec, souřadnice, počet čtverců */
  scanf ("%d%d%d%d", &n, &k, &x, &y);
  if ( x<0 || x>n || y<0 || y>n ) return 0;    /* uvnitř koberce */

  /* počátek souř. systému uprostřed koberce */
  x = abs (x-n/2);
  y = abs (y-n/2);
  k /= 2;

  while (k) {
    if ( x<=k && y<=k ) i++;    /* uvnitř čtverce */
    /* přesun SS do středu čtverce 1. kvadrantu */
    x = abs (x-k);
    y = abs (y-k);
    k /= 2;
  }
```

```
    printf ("Ctvercu: %d", i);
    return 0;
}
```

Psychologický postřeh dí, že fantazie není v rukou lidí, ale lidé jsou v rukou fantazie; pročež mi bylo dáno neudržet se a pro otrlejší nadšence napsat trojřádkovou verzi zdrojáku, za jejíž údernost zaplatíme právě nešťastným logaritmem v paměti:

```
int r (int x, int y, int k)
{return (k &&  (k= (x<=k && y<=k) + r (abs (x-k), abs (y-k), k/2)), k); }
printf ("Ctvercu:%d", r (abs (x-n/2), abs (y-n/2), k/2));
```

15-4-4 Písaři — Tomáš Valla

Naše úloha je trochu podobná zadání úlohy o knihovně z MO-P. Je tu však jeden rozdíl – zde není počet stran scénáře omezený, může jich být klidně 2^{2^n}. Řešení „vezmi všechny kapacity jednoho písaře od 0 do součtu všech stran a zkus, jestli to tak půjde rozdělit“ tudíž bude pomalé i při metodě půlení intervalu. (Argument, že na zápis takového čísla bychom potřebovali tak jako tak 2^n bitů, ponechme stranou, pracujeme v ideálním modelu.)

My na to půjdeme dynamickým programováním. Budeme počítat dvojrozměrné pole P velikosti $K \times N$, kde hodnota $P_{i,j}$ znamená nejmenší možný počet stran, zpracovávaný nejvytíženějším z písařů $1, \ldots, i$, kteří mají za úkol přepsat scénáře $1, \ldots, j$. Pole budeme plnit po řádcích, tj. pro počet písařů i spočítáme hodnoty $P_{i,j}$ postupně pro $j = i, \ldots, N$. Na konci výpočtu bude hledaná hodnota v $P_{K,N}$ a když už ji známe, jedním průchodem seznamem stran a hladovým přidělováním scénářů jednotlivým písařům vypíšeme rozdělení.

Zbývá vyřešit, jak najít rychle číslo $P_{i,j}$. Zřejmě $P_{0,*} = \infty$ (0 písařů potřebuje nekonečně mnoho času) a $P_{*,0} = 0$ (0 scénářů lze opsat za nulový čas). Když máme k dispozici i písařů na j stran, vyzkoušíme nechat i-tému písaři postupně scénáře l až j pro všechny $l = i, \ldots, j$ (pro $i-1$ písařů a scénáře $1, \ldots, l$ už to máme spočítané), a vybereme tu nejlepší variantu. Formálně zapsáno

$$P_{i,j} = \min_{l=i,\ldots,j} \left\{ \max\{ \sum_{k=l+1}^{j} \text{scénář } k;\ P_{i-1,l} \} \right\}.$$

A ejhle – my pracujeme jen s hodnotami pro $i-1$ a i písařů. Z celé tabulky si stačí pamatovat pouhé dva řádky a rázem máme paměťovou složitost $O(N)$. Čas bude $O(N^2 \cdot K)$, vyplňujeme tabulku $K \cdot N$ a na jednom čísle strávíme $O(N)$ času. Zkusíme ale zlepšit i tohle. Všimneme si následujících dvou skutečností:

- S přidáváním dalších scénářů současná optimální hodnota nikdy neklesne.

- Když v optimálním řešení pro $j-1$ scénářů poslední písař začíná scénářem číslo l, v optimálním řešení pro j scénářů stačí zkoumat takové varianty, kdy poslední písař začíná nějakým scénářem číslo $l' \geq l$.

Při výpočtu $P_{i,j}$ tedy začneme zkoumat jen od místa, kde skončilo číslo l pro $j-1$ scénářů a zkoumat přestaneme, když optimální hodnota pro konfiguraci s posledním písařem $l+1, \ldots, j$ přestane klesat. Jinými slovy – jakmile optimální hodnota pro písaře $1, \ldots, i-1$ začne přerůstat velikost i-tého písaře, nemá význam dále pokračovat.

Při vyplňování celého řádku tedy postupně index l proběhne hodnoty $1, \ldots, N$, strávíme tak nad řádkem zhruba $N+N$ operací a celková časová složitost bude $O(N \cdot K)$. V programu jsou pak pole indexována od jedničky, to aby se co nejvíce podobal našemu popisu.

```
#include <stdio.h>

#define MAX 100
#define INFTY 999999

int N, K;
int pages[MAX+1];
int P[2][MAX+1];
int S[MAX+2];                                   /* S[i] je soucet stran i..N */

static inline int max (int p, int q)
{
        return (p>q) ? p : q;
}

int main (void)
{
        int i, j, l, s, c;

        scanf ("%d %d", &N, &K);
        for (i=1; i<=N; i++) {
                scanf ("%d", pages+i);
                P[0][i] = INFTY;
        }
        for (i=N; i>0; i--) S[i] = S[i+1]+pages[i];

        for (i=1; i<=K; i++) {
                l=i-1;
                for (j=i; j<=N; j++) {
                        P[i&1][j] = max (S[l+1]-S[j+1], P[ (i-1)&1][l]);

                        while (l<j && P[i&1][j]>max (S[l+2]-S[j+1], P[
                            (i-1)&1][l+1]) ) {
                                P[i&1][j] = max (S[l+2]-S[j+1], P[
                                    (i-1)&1][l+1]);
                                l++;
                        }
                }
```

```
        }
        i=j=1;
        while (i<=N) {
                s=c=0;
                while (i<=N && s+pages[i]<=P[K&1][N]) {
                        s+=pages[i++];
                        c++;
                }
                printf ("%d pisar: %d hercu\n", j++, c);
        }
        return 0;
}
```

15-4-5 Haskell — Zdeněk Dvořák

Do zadání této úlohy se bohužel vloudilo pár nepřesností a naopak se do ní několik podstatných detailů nedostalo, takže úloha byla o dost obtížnější, než jsem původně zamýšlel, a navíc v řešení bylo potřeba použít několik netriviálních triků.

Základní myšlenka je jednoduchá – pro každou funkci vytvoříme sekvenci instrukcí, která ji vyhodnotí, tj. provádí operace dané funkcí tak dlouho, dokud není výsledek buď číslo, nebo thunk s příliš málo parametry na to, abychom ho mohli vyhodnocovat dál. Pro každou konstrukci, kterou potřebujeme vyhodnocovat, máme k dispozici nějaké instrukce – `IExecute` a `IReturn` pro volání funkcí, `ICExecute` pro `case`, ...

Problémy nastanou, když chceme, aby vše fungovalo jako v Haskellu, tj. abychom měli zajištěnu línost vyhodnocování a sdílení. U volání funkce nemůžeme nechat vyhodnotit její parametry a pak s nimi zavolat funkci; například `f x y = y`, `f (error "chyba") 1` má mít jako výsledek 1, ne skončit s chybou. Obdobný problém máme pro výrazy ve větvích `case` výrazu. Proto ve všech těchto případech musíme vytvořit thunky, které budou vyhodnoceny, až budou-li potřeba. Bylo by poměrně obtížné to dělat přímo při kompilaci, proto nejprve program zjednodušíme tak, že všechny argumenty funkcí a výrazy ve větvích `if`ů jsou buď číselné konstanty, nebo proměnné (to můžeme vždy udělat vytvořením pomocných funkcí a definic nových proměnných v `let`-výrazech (této operaci se říká flattening – „zplošťování“)).

Obdobným trikem se zbavíme i lokálních funkcí – uděláme z nich globální; tady je potřeba dát pozor na to, že lokální funkce mohou přistupovat k lokálním definicím v dané funkci, tedy je potřeba je nově vytvořeným globálním funkcím poslat jako konstanty. Lokální definice tedy budou pouze tvaru `d = f x y z`, kde `f` je globální funkce, případně `d = x` nebo `d = 5`. Například funkce

```
f x y = let a z = fa (x + y) z b
            b t = fb (x - y) t a
        in g (a x) (b y)
```

po těchto úpravách vypadá takto:

```
f x y = let a = fa' xpy b
            b = fb' xmy a
            xpy = plus x y
            xmy = minus x y
            p1 = a' a x
            p2 = b' b y
        in g p1 p2
plus x y = x + y
minus x y = x - y
a' a x = a x
b' b y = b y
fa' xpy b z = fa xpy z b
fb' xmy a t = fb xmy t a
```

Nyní se již můžeme odvážně pustit do překladu jednotlivých příkazů. Kromě drobné technické komplikace u `case` způsobené tím, že `ICExecute` je poměrně nešikovně navrženo, narazíme na problémy až u našeho oblíbeného příkazu `let`. Podívejme se na volání `fa'` a `fb'` ve výše uvedeném příkladu. Abychom vytvořili thunk pro `fa'`, potřebujeme mu jako argument dát odkaz na thunk pro `fb'` a naopak – vypadá to, že jsme se zacyklili. Naštěstí máme instrukci `IRewrite`; my si tedy nejprve vytvoříme místa v paměti, kde budeme mít uloženy tyto thunky, pak si je vytvoříme tak, aby se jejich parametry odkazovaly na tato místa, a nakonec je pomocí `IRewrite` na tato místa nakopírujeme.

Druhý problém je zajištění sdílení. Pro lokální proměnné je to jednoduché – stačí za každé volání `IExecute` přidat volání `IRewrite`, které původní thunk nahradí novou hodnotou; vzhledem k tomu, že jak v registrech, tak v thuncích máme pouze ukazatele na data, nahradíme tímto tuto hodnotu všude, kam byla zkopírována. My bychom toto ovšem chtěli zajistit i pro globální proměnné; to uděláme snadno tak, že celý program obalíme do jednoho velkého `let`u.

No a s těmito poznatky je již sepsání kompilátoru záležitost tří odpolední a 600 řádek kódu.

```
module Main where

import Monad
import List
import Maybe
import Language.Haskell.Pretty
import Language.Haskell.Syntax

data Equation = Equation String [String] Expression
data Expression = Let [Equation] Expression |
                  Apply Expression Expression |
                  Number Int |
                  Variable String |
                  Plus Expression Expression |
                  Mul Expression Expression |
                  Minus Expression Expression |
                  Div Expression Expression |
                  Case Expression [(Int, Expression)] Expression
```

```
cLet::[Equation]->Expression->Expression
cLet [] expr = expr
cLet eqs expr = Let eqs expr

nowhere::SrcLoc
nowhere = SrcLoc "" 0 0

tre::Expression->HsExp
tre (Let eqs expr) = HsLet (map trs eqs) (tre expr)
tre (Apply f arg) = HsApp (tre f) (tre arg)
tre (Number n) = HsLit (HsInt $ toInteger n)
tre (Variable v) = HsVar $ UnQual $ HsIdent v
tre (Plus e1 e2) = HsInfixApp (tre e1) (HsQVarOp $ UnQual $ HsSymbol "+") (tre e2)
tre (Mul e1 e2) = HsInfixApp (tre e1) (HsQVarOp $ UnQual $ HsSymbol "*") (tre e2)
tre (Minus e1 e2) = HsInfixApp (tre e1) (HsQVarOp $ UnQual $ HsSymbol "-") (tre e2)
tre (Div e1 e2) = HsInfixApp (tre e1) (HsQVarOp $ UnQual $ HsIdent "div") (tre e2)
tre (Case expr alts dflt) = HsCase (tre expr) (als ++ [dalt])
  where
    dalt = HsAlt nowhere HsPWildCard (HsUnGuardedAlt $ tre dflt) []
    als = map mkAlt alts
    mkAlt (n, exp) = HsAlt nowhere (HsPLit $ HsInt $ toInteger n)
                                   (HsUnGuardedAlt $ tre exp) []

trs::Equation->HsDecl
trs (Equation name pars expr) =
   HsFunBind [HsMatch nowhere (HsIdent name) pat (HsUnGuardedRhs $ tre expr) []]
  where
   pat = map mkPat pars
   mkPat par = HsPVar (HsIdent par)

instance Show Equation where
  show eq = prettyPrint $ trs eq
  showList eq x = x ++ (prettyPrint $ HsModule nowhere
      (Module "Main") Nothing [] (map trs eq))

instance Show Expression where
  show expr = prettyPrint $ tre expr

eqName::Equation->String
eqName (Equation name _ _) = name

sampleProgram::[Equation]
sampleProgram =
 [
  Equation "mod" ["x", "y"]
     (Let [
            Equation "p" [] (Div (Variable "x") (Variable "y")),
            Equation "s" [] (Mul (Variable "p") (Variable "y"))
          ]
        (Minus (Variable "x") (Variable "s"))),
    -- mod x y = let p = x `div` y
    --               s = p * y
    --           in x - s
  Equation "nsd" ["x", "y"]
     (Case (Variable "x")
        [(0, Variable "y")]
        (Let [
               Equation "p" [] (Apply
```

```
                                (Apply (Variable "mod") (Variable "y"))
                                (Variable "x"))
            ]
          (Apply (Apply (Variable "nsd") (Variable "p")) (Variable "x"))))
    -- nsd x y = case x of
    --             0 -> y
    --             _ -> let p = mod y x
    --                  in nsd p x
 ]

test::State
test = compile sampleProgram (Apply (Apply (Variable "nsd") (Number 24)) (Number 36))

data Value = VNumber Int |
             VThunk Int [Int]
             deriving (Eq, Show)
type Regs = [(Int,Int)]
data State = State {stateCode::[(Int,String,Int,[Instruction])],
                    stateData::[(Int,Value)],
                    stateFree::Int,
                    stateRegs::Regs,
                    stateInsns::[Instruction],
                    stateStack::[(Regs, [Int])]}

showCode::(Int,String,Int,[Instruction])->String
showCode (n,name,args,insns) =
   name ++ "(" ++ show n ++ "): " ++ show args ++ " args\n" ++
   concatMap showInsn insns ++ "\n"
  where
   showInsn insn = "  " ++ show insn ++ "\n"

instance Show State where
  show state = concatMap showCode (stateCode state)

emptyState::State
emptyState=State {stateCode=[], stateData=[], stateFree=1, stateRegs=[],
                  stateInsns=[], stateStack=[]}

data Instruction =
       IMove Int Int |
       IPlus Int Int Int |
       IMinus Int Int Int |
       IMul Int Int Int |
       IDiv Int Int Int |
       IExecute Int |
       IReturn |
       ICExecute Int Int Int |
       ICall Int Int |
       ILoad Int Int |
       IApply Int Int Int |
       IRewrite Int Int
       deriving (Show)

-- interpret
interpret::State->Int
interpret state
  | null insns = ret
  | otherwise = interpret $ executeInsn insn $ state {stateInsns = rest}
```

```
  where
   insns = stateInsns state
   VNumber ret = readValue state 0
   (insn:rest) = insns

getReg::State->Int->Int
getReg state reg = fromJust $ lookup reg (stateRegs state)

setReg::Int->Int->State->State
setReg reg addr state = state {stateRegs = regs'}
 where
   regs = stateRegs state
   regs' = repl regs
   repl [] = [(reg, addr)]
   repl ((rg, add) : rest)
    | rg == reg = (rg, addr) : rest
    | otherwise = (rg, add) : repl rest

readValue::State->Int->Value
readValue state reg = fromJust $ lookup addr (stateData state)
  where
    addr = getReg state reg

writeValue::Int->Value->State->State
writeValue reg value state = state {stateData = dta}
  where
    addr = getReg state reg
    dta = (addr, value) : stateData state

allocData::Int->State->State
allocData reg state = setReg reg addr $ state {stateFree = addr + 1}
  where
   addr = stateFree state

addInsns::[Instruction]->State->State
addInsns insns state = state {stateInsns = insns ++ stateInsns state}

getFunction::State->Int->(Int,String,[Instruction])
getFunction state fn = look fns
  where
    fns = stateCode state
    look ((f, name, n, i) : rest)
      | f == fn = (n, name, i)
      | otherwise = look rest

loadArgs::[Int]->State->State
loadArgs args state =
  foldl (\state (n, addr) -> setReg n addr state) state $ zip [1..] args

pushState::[Int]->State->State
pushState restArgs state = state {stateRegs = [], stateStack = stack'}
  where
   stack = stateStack state
   stack' = (filter (\(n,_) -> n /= 0) $ stateRegs state, restArgs) : stack

popState::State->(State, [Int])
popState state = (state {stateRegs = (0, addr) : regs, stateStack = stack}, rest)
  where
```

```
    ((regs, rest) : stack) = stateStack state
    addr = getReg state 0

executeInsn::Instruction->State->State
executeInsn (IMove from to) state = setReg to (getReg state from) state
executeInsn (IPlus a b to) state = executeArith (+) a b to state
executeInsn (IMinus a b to) state = executeArith (-) a b to state
executeInsn (IMul a b to) state = executeArith (*) a b to state
executeInsn (IDiv a b to) state = executeArith div a b to state
executeInsn (IExecute reg) state =
  case value of
    VNumber _ -> setReg 0 (getReg state reg) state
    VThunk fn args -> executeExec fn args
 where
   value = readValue state reg
   executeExec fn args
       | nArgs < rArgs = setReg 0 (getReg state reg) state
       | otherwise = addInsns insns $ loadArgs args $ pushState rest state
     where
       nArgs = length args
       (rArgs, _, insns) = getFunction state fn
       (pass, rest) = splitAt rArgs args
executeInsn IReturn state
    | null remArgs = addInsns [IExecute 0] state'
    | otherwise = addInsns [IExecute 0] state''
  where
    (state', remArgs) = popState state
    VThunk fn args = readValue state' 0
    state'' = writeValue 0 (VThunk fn (args ++ remArgs)) state'
executeInsn (ICExecute a b w) state
  | readValue state a == readValue state b = addInsns [IExecute w] state
  | otherwise = state
executeInsn (ICall fn reg) state = writeValue reg (VThunk fn []) $ allocData reg state
executeInsn (ILoad n reg) state = writeValue reg (VNumber n) $ allocData reg state
executeInsn (IApply fn par reg) state = writeValue reg nThunk $ allocData reg state
 where
  VThunk fnum args = readValue state fn
  nThunk = VThunk fnum (args ++ [getReg state par])
executeInsn (IRewrite what with) state = writeValue what (readValue state with) state

executeArith::(Int->Int->Int)->Int->Int->Int->State->State
executeArith op a b to state = writeValue to (VNumber rslt) $ allocData to state
  where
    VNumber va = readValue state a
    VNumber vb = readValue state b
    rslt = op va vb

-- stav pri kompilaci
data ECompileState =
   ECompileState {ecFreeReg::Int,               -- prvni volny registr
                  ecLocal::[(String, Int)],     -- lokalni promenne
                  ecFunctions::[(String, Int)], -- funkce
                  ecState::State,               -- stav
                  ecCurFn::String}              -- aktualne kompilovana fce
   deriving (Show)

data TMonad a = TMonad (ECompileState -> (ECompileState, a))
instance Monad TMonad where
```

```
  TMonad f >>= g = TMonad h
    where
      h state = let (state1, x) = f state
                    TMonad g' = g x
                in g' state1
  return x = TMonad (\state -> (state, x))

getState::TMonad ECompileState
getState = TMonad (\state -> (state, state))

setState::ECompileState->TMonad ()
setState state = TMonad (\_ -> (state, ()))

curFN::TMonad String
curFN = getState >>= return . ecCurFn

setCurFN::String->TMonad ()
setCurFN name = getState >>= \s -> setState s{ecCurFn=name}

runCompile::TMonad a -> State
runCompile (TMonad f) = ecState ecSt
  where
    (ecSt, _) = f (ECompileState 1 [] [] emptyState "")

-- zkompiluje program
compile::[Equation]->Expression->State
compile program expression = runCompile (doCompile program expression)

doCompile::[Equation]->Expression->TMonad ()
doCompile program expression =
  do
   let program' = simplify $ Equation "main" [] (Let program expression)
       fnames = map eqName program'
   state <- getState
   setState state{ecFunctions = ("id", 1) : zip fnames [2..]}
   compileEquation $ Equation "id" ["x"] (Variable "x")
   mapM compileEquation program'
   newECState []
   code <- compileExpression (Variable "main") 0
   state <- getState
   let ecSt = ecState state
   setState state{ecState = ecSt {stateInsns=code}}

-- pripravi stav pro kompilaci vyrazu
newECState::[String]->TMonad ()
newECState params =
  do
   state <- getState
   setState state{ecFreeReg = 1, ecLocal = []}
   bind params

-- zkompiluje jednu rovnost programu a prislusny kod prida k pocatecnimu stavu
compileEquation::Equation->TMonad ()
compileEquation (Equation fc params expr) =
  do
   Just place <- findGlobal fc
   setCurFN fc
   newECState params
```

```
   exprCode <- compileExpression expr 0
   state <- getState
   let ecSt = ecState state
       mCode' = (place, fc, length params, exprCode ++ [IReturn]) : stateCode ecSt
   setState state{ecState = ecSt {stateCode = mCode'}}

-- zkompiluje zadany vyraz tak, aby vracel hodnotu v registru target, a vrati
-- prislusny kod
compileExpression::Expression->Int->TMonad [Instruction]
compileExpression expression target =
  case expression of
    Let locals expr -> compileLet locals expr target
    Apply fn param -> compileApply fn param target
    Number val -> return [ILoad val target]
    Variable var -> compileVar expression target
    Plus e1 e2 -> compileArith e1 e2 IPlus target
    Mul e1 e2 -> compileArith e1 e2 IMul target
    Minus e1 e2 -> compileArith e1 e2 IMinus target
    Div e1 e2 -> compileArith e1 e2 IDiv target
    Case expr cases dflt -> compileCase expr cases dflt target

-- kompiluje let.  Vytvorime thunky pro definovane lokalni promenne
-- a zkompilujeme expr s moznosti pristupu k temto lokalnim promennym
-- drobny trik -- musime zajistit, aby fungovala rekurzivni volani
compileLet::[Equation]->Expression->Int->TMonad [Instruction]
compileLet locals expr target =
  do
   bind defs
   claimRegs <- mapM claimReg defs
   mkThunkCodes <- mapM makeLocalThunk locals
   eval <- compileExpression expr target
   unbind defs
   return (concat claimRegs ++ concat mkThunkCodes ++ eval)
  where
   defs = map eqName locals
   claimReg def =
     do
      Just reg <- findLocal def
      return [ILoad 0 reg]
   makeLocalThunk (Equation fn [] expr) =
     case expr of
       Number _ -> prepareScalar fn expr
       Variable x ->
         do
          loc <- findLocal x
          case loc of
            Just _ -> makeApplyThunk fn $ Apply (Variable "id") expr
            Nothing -> prepareScalar fn expr
       _ -> makeApplyThunk fn expr
   prepareScalar fn expr =
     do
      Just reg <- findLocal fn
      code <- loadScalar expr 0
      return $ code ++ [IRewrite reg 0]
   makeApplyThunk fn expr =
     do
      let (Variable f, args) = splitApply expr
      Just reg <- findLocal fn
```

```
      Just lthunk <- findGlobal f
      addArgs <- mapM (addArg 0) args
      return $ ICall lthunk 0 : concat addArgs ++ [IRewrite reg 0]
   addArg f arg =
     do
      reg <- allocateReg
      code <- loadScalar arg reg
      return $ code ++ [IApply f reg f]
   splitApply app = splitApply' app []
   splitApply' (Apply x arg) rest = splitApply' x (arg:rest)
   splitApply' f rest = (f, rest)

-- kompiluje aplikaci funkce.
compileApply::Expression->Expression->Int->TMonad [Instruction]
compileApply fn x target =
  do
   temp <- allocateReg
   getFnCode <- compileExpression fn temp
   arg <- allocateReg
   pCode <- loadScalar x arg
   rthunk <- allocateReg
   return $ getFnCode ++ pCode ++
            [IApply temp arg rthunk, IExecute rthunk, IMove 0 target]

-- kompiluje vyhodnoceni promenne
compileVar::Expression->Int->TMonad [Instruction]
compileVar var target =
  do
   reg <- allocateReg
   load <- loadScalar var reg
   return $ load ++ [IExecute reg, IRewrite reg 0, IMove 0 target]

-- kompiluje aritmeticky vyraz
compileArith::Expression->Expression->(Int->Int->Int->Instruction)->
  Int->TMonad [Instruction]
compileArith a b op target =
  do
   fname <- curFN
   ra <- allocateReg
   codeA <- compileExpression a ra
   rb <- allocateReg
   codeB <- compileExpression b rb
   return $ codeA ++ codeB ++ [op ra rb target]

-- zkompiluje case
compileCase::Expression->[(Int, Expression)]->Expression->Int->TMonad [Instruction]
compileCase expr cases dflt target =
  do
   reg <- allocateReg
   code <- compileExpression expr reg
   branchCodes <- mapM (compileBranch reg) cases
   dfltLoad <- loadScalar dflt 0
   return $ code ++ dfltLoad ++ concat branchCodes ++ [IExecute 0, IMove 0 target]
  where
   compileBranch reg (val, e) =
    do
     tmp <- allocateReg
     rslt <- allocateReg
```

```
    code <- loadScalar e rslt
    return $ code ++ [ILoad val tmp, ICExecute tmp reg rslt]

-- nahraje Number nebo Variable do registru
loadScalar::Expression->Int->TMonad [Instruction]
loadScalar (Number n) target = return [ILoad n target]
loadScalar (Variable x) target =
  do
   l <- findLocal x
   case l of
     Just reg -> return [IMove reg target]
     Nothing ->
       do
        Just fn <- findGlobal x
        return [ICall fn target]

isScalar::Expression->Bool
isScalar (Number _) = True
isScalar (Variable _) = True
isScalar _ = False

-- najde registr, v nemz je lokalni definice promenne var
findLocal::String->TMonad (Maybe Int)
findLocal var =
  do
   regs <- getState >>= return . ecLocal
   return $ lookup var regs

-- najde misto v pameti pro kod, kde je ulozena definice funkce fn
findGlobal::String->TMonad (Maybe Int)
findGlobal fn =
  do
   functions <- getState >>= return . ecFunctions
   return $ lookup fn functions

-- naalokuje novy registr
allocateReg::TMonad Int
allocateReg =
  do
   state <- getState
   let reg = ecFreeReg state
   setState state{ecFreeReg = reg + 1}
   return reg

-- odstrani lokalni promenne l
unbind::[String]->TMonad ()
unbind l =
  do
   state <- getState
   let regs = ecLocal state
       l' = map (\x->(x,undefined)) l
       newRegs = deleteFirstsBy (\(n1,_) (n2,_) -> n1 == n2) regs l'
   setState state{ecLocal = newRegs}

-- naalokuje lokalni promenne l
bind::[String]->TMonad ()
bind l =
  do
```

```
   regs <- getState >>= return . ecLocal
   newRegs <- foldM bindVar regs l
   state <- getState
   setState state{ecLocal = newRegs}
   return ()
  where
   bindVar regs var =
     do
      fn <- curFN
      reg <- allocateReg
      return ((var,reg) : regs)

-- prevede program do kanonickeho tvaru (vsechny argumenty funkci jsou bud
-- konstanty nebo promenne, stejne tak vysledky vetvi case, let definice
-- jsou tvaru var = f arguments, kde pokud argumenty nejsou prazdne, je
-- f globalni funkce)
simplify::Equation->[Equation]
simplify equation = snd $ simplify' (genFreeNames "free", []) equation

type SState = (FreeNames, [Equation])
type SEState = (FreeNames, Expression, [Equation])
type FreeNames = [String]

-- vygeneruje nekonecny seznam nepouzitych jmen, koncicich zadanym suffixem
genFreeNames::String->[String]
genFreeNames suffix = map (\n->show n ++ suffix) [1..]

simplify'::SState->Equation->SState
simplify' (fNames, eqs) (Equation f params expr) =
  foldl simplify' (newFNames, Equation f params newExpr : eqs) newEquations
 where
  (newFNames, newExpr, newEquations) = simplifyExpr (fNames, expr, [])

simplifyExpr::SEState->SEState
simplifyExpr (freeNames, Let eqs expr, rest) = (free2, Let eqs' expr', r2)
  where
    (free1, expr', r1) = simplifyExpr (freeNames, expr, rest)
    (free2, eqs', r2) = foldl simplifyLetEq (free1, [], r1) eqs
    simplifyLetEq (fr, es, rst) (Equation x param exp) =
      let ((fr', var, rst'), letEq) = simplifyToVar (fr, exp, rst) param
       in (fr', maybeToList letEq ++ Equation x [] var : es, rst')
simplifyExpr (freeNames, Apply f arg, rest) =
   (free2, cLet (maybeToList letEq) (Apply f' var), r2)
  where
    ((free1, var, r1), letEq) = simplifyToVar (freeNames, arg, rest) []
    (free2, f', r2) = simplifyExpr (free1, f, r1)
simplifyExpr (freeNames, Case expr alts dflt, rest) =
    (free3, cLet letEqs (Case expr' alts' dflt'), r3)
  where
    letEqs = catMaybes (dLetEq:aLetEqs)
    (free1, expr', r1) = simplifyExpr (freeNames, expr, rest)
    ((free2, dflt', r2), dLetEq) = simplifyToVar (free1, dflt, r1) []
    (free3, alts', aLetEqs, r3) = foldl simplifyAlt (free2, [], [], r2) alts
    simplifyAlt (fr, as, les, rst) (val, exp) =
      let ((nFr, var, nRst), letEq) = simplifyToVar (fr, exp, rst) []
       in (nFr, (val, var):as, letEq:les, nRst)
simplifyExpr x@(_, Plus a b, _) = simplifyArith x Plus a b
simplifyExpr x@(_, Mul a b, _) = simplifyArith x Mul a b
```

```
simplifyExpr x@(_, Minus a b, _) = simplifyArith x Minus a b
simplifyExpr x@(_, Div a b, _) = simplifyArith x Div a b
simplifyExpr x = x

simplifyArith::SEState->(Expression->Expression->Expression)->
  Expression->Expression->SEState
simplifyArith (freeNames, _, rest) op a b = (f2, op a' b', r2)
  where
    (f1, a', r1) = simplifyExpr (freeNames, a, rest)
    (f2, b', r2) = simplifyExpr (f1, b, r1)

-- vytvori novou funkci pocitajici vyraz, vytvori promennou a vrati rovnici,
-- kterou se do teto promenne priradi funkce s parametry odpovidajicimi
-- volnym promennym vyrazu
simplifyToVar::SEState->[String]->(SEState, Maybe Equation)
simplifyToVar (freeNames, Variable var, rest) params =
  ((freeNames, Variable var, rest), Nothing)
simplifyToVar (freeNames, Number n, rest) params =
  ((freeNames, Number n, rest), Nothing)
simplifyToVar (freeNames, expr, rest) params =
  ((f1, Variable varName, Equation fName allPars expr : rest),
  Just $ Equation varName [] fCall)
  where
   (fName:varName:f1) = freeNames
   pars = freeVars expr \\ params
   allPars = pars ++ params
   fCall = foldl Apply (Variable fName) $ map Variable pars

-- nalezne volne promenne vyrazu
freeVars::Expression->[String]
freeVars expr = nub $ freeVars' expr
  where
   freeVars' (Let eqs expr) = nub (fE ++ fEq) \\ map eqName eqs
     where
      fE = freeVars' expr
      fEq = concatMap eqFVars eqs
      eqFVars (Equation _ param expr) = freeVars expr \\ param
   freeVars' (Apply e1 e2) = freeVars' e1 ++ freeVars' e2
   freeVars' (Number _) = []
   freeVars' (Variable x) = [x]
   freeVars' (Plus e1 e2) = freeVars' e1 ++ freeVars' e2
   freeVars' (Minus e1 e2) = freeVars' e1 ++ freeVars' e2
   freeVars' (Mul e1 e2) = freeVars' e1 ++ freeVars' e2
   freeVars' (Div e1 e2) = freeVars' e1 ++ freeVars' e2
   freeVars' (Case e alts d) = freeVars' e ++ freeVars' d ++
                               concatMap (freeVars' . snd) alts

main::IO ()
main =
  do
   print test
   print $ interpret test
```

15-5-1 Komprimovaný obrázek — Miroslav Rudišin

Nejdříve si zakomprimované úseky rozdělíme do jednotlivých řádků. Stejnobarevné řádky zpracováváme najednou, čímž je jeden úsek rozdělen maximálně na 3 části.

Dále je postupně po řádcích rekonstruován původní obrázek. Po zpracování celého řádku budeme znát velikosti stejnobarevných oblastí končících na tomto řádku.

Při zpracovávání jednobarevného úseku na řádku k němu připojujeme všechny sousedící stejnobarevné úseky z předchozího řádku. Je důležité si uvědomit, že v řádku se může nacházet několik různých stejnobarevných oblastí, které sousedí s jednou oblastí na předchozím řádku (např. pro obrázek s dvěma čtverci se stejným středem) a kterou bychom tak mohli nedopatřením přičíst víckrát. Řešením může být chápání úseku (jednobarevných oblastí) jako množin a operaci spojení jako jejich sjednocení.

Množinu si pamatujeme v stromu, jehož kořen „ví“, kolik má množina prvků, a ostatní vrcholy ukazují na nějaký jiný (nadřazený) prvek množiny. Sjednocení spočívá v „přivěšení“ menší množiny k větší a identifikace množiny v nalezení jejího kořene. Abychom zlepšili průměrnou časovou složitost vyhledávání kořene z $O(\log S)$ na $O(\log^* S)$, po každém nalezení kořene upravíme všem prvkům na cestě ukazatel přímo na kořen. Tento algoritmus je znám pod jménem Union-Find.

Ke zjištění maximální stejnobarevné plochy si stačí pamatovat jenom posledně zpracovaný řádek. Uvědomíme-li si, že v jednom řádku může být jenom S různých úseků (S je šířka řádku), vystačíme si s pamětí $O(S)$.

Každý úsek je zpracován maximálně třikrát a v jednom řádku je $2 \cdot S$-krát vykonána operace find na S prvcích. Časová složitost algoritmu je proto $O(N + N/S \cdot S \cdot \log^* S) = O(N \log^* S)$.

Pozn. M.M.: To je ale hezký trik! Ale nešlo by to lineárně? Na celý obrázek se můžeme dívat jako na neorientovaný graf: vrcholy jsou jednobarevné úseky, hrany odpovídají tomu, který s kterým sousedí. Hran je lineárně mnoho (je to totiž rovinný graf) a také je dovedeme v lineárním čase najít, stačí zopakovat algoritmus z předchozího řešení a místo sjednocování Union-Find stromečků přidávat hrany do grafu. A pak už jen graf prohledáním do šířky rozložíme na komponenty souvislosti a spočítáme, jak je která velká.

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

struct U { int z;                       /* zacatek */
           int k;                       /* konec */
           int b;                       /* barva */
           int g;   /* group */     } radky[2][MAXS];

int zasobnik[2*MAXS], ff,               /* zasobnik volnych skupin, ukazatel na vrchol */
```

```
            uf_data[2*MAXS],        /* datova struktura pro Union-Find */
            size[2*MAXS],           /* velikost aktualni plochy pro skupinu */
            used[2*MAXS];           /* radek posledniho vyskytu skupiny */
int s, parita,                      /* sirka obrazku */
            sr, ur,                 /* sloupec a usek na zpracovavanem radku */
            url, urlC,              /* aktivni usek v minulem radku, pocet useku
                                       v minulem radku */
            ar=1, max;              /* aktualni radek, maximalni plocha */
int uf_find (int e)                 /* nalezne reprezentanta skupiny a zapakuje cestu */
{
            int t, r = e;
            while (uf_data[r]>=0) { r = uf_data[r]; }
            while (uf_data[e]>=0) { t = uf_data[e]; uf_data[e] = r; e = t; }
            return r;
}
int uf_union (int a, int b)         /* sjednoti skupiny a vrati reprezentanta */
{
            if (a != b) {
                        if (uf_data[a] > uf_data[b]) { a ^= b ^= a ^= b; }   /* swap */
                        uf_data[a] += uf_data[b];    /* mensi skupinu "zavesime" pod vetsi */
                        uf_data[b] = a;
                        size[a] += size[b];
            }
            return a;
}
void konec_radku ()
{
            int i, g;
            for (i = 0; i < urlC; i++) {
                        g = radky[1-parita][i].g;
                        if (used[g] > 0 && used[g] != ar) {   /* uvolni nepouzite useky */
                                    if (uf_data[g] < 0 && size[g] > max) max = size[g];
                                    zasobnik[--ff] = g;
                                    used[g] = 0;
                        }
            }
            urlC = ur; ur = url = 0; ar++; parita = 1-parita;
}
void zpracuj (int b, int d, int mult)
{
            int g0, g, k = sr + d - 1;
            if (d == 0 || mult == 0) return;
            g0 = g = zasobnik[ff++];
            size[g] = d*mult;
            uf_data[g] = -1;
            while (url < urlC && sr > radky[1-parita][url].k) url++;
            while (url < urlC && radky[1-parita][url].z <= k) {
                        if (radky[1-parita][url].b == b) {
```

```
                        g = uf_union (g, uf_find (radky[1-parita][url].g));
                    }                   /* sjednocuje sousedici useky stejne barvy */
                    url++;
            };
            if (url) url--;

            radky[parita][ur].z = sr;
            radky[parita][ur].k = k;
            radky[parita][ur].b = b;
            radky[parita][ur++].g = g;
            used[g] = ar;

            if (uf_data[g0] >= 0) ff--;    /* "zavesena" skupina muze byt uvolnena */
            if (sr + d == s) konec_radku ();
}
int main ()
{
            int d, b, i;
            for (i = 2*MAXS-1; i>0; i--) zasobnik[i] = i;

            scanf ("%d", &s   /* irka */  );
            while (scanf ("%d %d", &d   /* elka */   , &b   /* arva */   ) == 2) {
                    if (s - sr < d) {
                            zpracuj (b, s - sr, 1);    /* doplni radek do konce */
                            d -= s - sr; sr = 0;
                            zpracuj (b, s, d/s);    /* zpracuje jednobarevne radky */
                            d %= s;
                    }
                    zpracuj (b, d, 1);
                    sr += d;
            }
            konec_radku ();                 /* dokoncime zpracovani posledniho radku */
            printf ("Nejvetsi souvisla plocha jedne barvy: %d\n", max);
            return 0;
}
```

15-5-2 Odečtolam — Pavel Machek

(S díky Petru Škodovi a Milanu Strakovi)

Každý tah na odečtolamu zachovává součet všech čísel $s = a_1 + a_2 + \ldots + a_n$ ($a + b + c = a + b + -b + c + b$).

Ukážeme, že pro $s = 0$ nemá hra řešení (kromě triviálního případu, kdy jsou ve všech vrcholech nuly už na začátku). Protože se součet zachovává, musí hra s $s = 0$ skončit v konfiguraci 0,0,0,...,0. Jenže každý tah otočí znaménko záporného čísla, takže není možné z nenulového čísla vytvořit nulu.

Nechť t je absolutní hodnota součtu všech záporných čísel na odečtolamu.

Výpočet bude probíhat ve fázích, z nichž každá sníží t. Každá fáze bude trvat nejvýše n^2 tahů. Fáze se bude skládat z nejvýše n podfází.

Na začátku podfáze vybereme nějaké záporné číslo (když žádné není, vyhráli jsme), které je vpravo od případných nul a kladného čísla (určitě existuje). Přečíslujeme vrcholy tak, aby vybrané číslo bylo a_2. Táhneme a_2 a dostaneme:

$$a_1 + a_2, -a_2, a_2 + a_3, a_4, \ldots, a_n.$$

Dokud se nedostaneme na konec posloupnosti a dokud je to možné, táhneme vrchol vpravo od předchozího vrcholu:

$$a_1 + a_2, a_3, -a_2 - a_3, a_2 + a_3 + a_4, \ldots, a_n.$$

Když dojdeme na konec posloupnosti (to nastane pokud $-a_2 > a_3 + a_4 + \ldots + a_n$), bude situace takováto:

$$a_1 + 2 \cdot a_2 + a_3 + \ldots + a_n, a_3, a_4, \ldots, a_n, -a_2 - \ldots - a_n,$$

což můžeme přepsat jako:

$$s + a_2, a_3, \ldots, a_n, a_1 - s.$$

Protože $-a_2 > a_3 + \ldots + a_n = s - a_1 - a_2$, tak $a_1 - s > 0$. t se snížilo o s, protože na prvním místě je $s + a_2$ a $a_1 - s > 0$ a ostatní členy se jen přesunuly. Snížili jsme t, takže skončila fáze.

Na konec posloupnosti nedojdeme, pokud existuje i takové, že $-a_2 < a_3 + \ldots + a_i$. Situace bude následující:

$$a_1 + a_2, a_3, \ldots, a_{i-1}, -a_2 - \ldots - a_{i-1}, a_2 + \ldots + a_i, \ldots, a_n.$$

Nechť $u = a_1 + ... + a_{i-1}$. Výraz pak bude:

$$a_1 + a_2, a_3, \ldots, a_{i-1}, a_1 - u, u + a_i - a_1, a_{i+1}, \ldots, a_n.$$

Zde a_1 je nezáporné (tak jsme ho vybrali) a a_i je určitě také nezáporné, protože jinak by se podfáze musela zastavit už na a_{i-1}. V posloupnosti se změnily jenom tři členy, ostatní se nejvýš přesunuly: $a_1 \to a_1 + a_2$; $a_2 \to a_1 - u$; $a_i \to u + a_i - a_1$.

Čísla $a_1 - u$ i $u + a_i - a_1$ jsou kladná, takže t kleslo o a_1 (aby se podfáze zastavila, musí být $a_2 + ... + a_{i-1} = u - a_1 < 0$ a $a_2 + ... + a_i = u + a_i - a_1 > 0$).

Problém nastane, pokud a_1 je 0, v tom případě ale můžeme vložit další podfázi. S každou podfází se záporné číslo posune doleva a ubyde jedna nula mezi vybraným číslem a číslem kladným. Nejvýše za n podfází fáze skončí.

Časová složitost je $O(n^2 \cdot t)$, paměťová složitost je $O(n)$.

15-5-3 Manhattan — Pepa Cibulka

Řešení je založeno na dynamickém programování – postupně po řádcích (západo-východních ulicích) zleva doprava počítáme počet cest vedoucích do dané křižovatky. Ten je roven součtu počtů cest vedoucích do sousedních křižovatek na západ a na sever (odjinud přijet nemůžeme), které už máme spočítané; případně 0, je-li křižovatka opravována.

Když načteme souřadnice opravované křižovatky, přiřadíme jí hodnotu např. -1 (to můžeme dělat přímo v poli, kam budeme později ukládat počty cest), takže pak zjištění, zda je opravovaná, provedeme v konst. čase. Časová i paměťová složitost tedy jsou $O(m \cdot n)$.

Někteří řešitelé si všimli, že v paměti stačí mít jen právě zpracovávaný a předcházející řádek (dokonce jen části z nich), čímž zlepšili paměťovou složitost na $O(k + n)$. Pak ale nastal problém s určováním, zdali je křižovatka opravovaná, kvůli čemuž si tito řešitelé zhoršili časovou složitost. Ve vzorovém řešení to řešíme tak, že si každou opravovanou křižovatku zařadíme do seznamu pro řádek, ve kterém leží – to se dá v čase i paměti $O(k + m)$ provést třeba použitím spojového seznamu. To, zda je křižovatka opravována, si pak pamatujeme jen pro křižovatky z právě zpracovávaného řádku. Paměťová složitost je $O(k + n + m)$, časová zůstává $O(m \cdot n)$.

Některá řešení počítala pouze počet cest do opravovaných a cílové křižovatky: Nejprve spočítala počet cest se zanedbáním předchozích opravovaných křižovatek jako kombinační číslo $\binom{i+j}{i}$, kde i, j je číslo řádky, sloupce. Správnost plyne z toho, že Bill vybírá i z $i + j$ křižovatek, kudy pojede na jih. Pro každou předcházející rozkopanou křižovatku pak od tohoto čísla odečteme součin počtu dobrých cest, které do ní vedou (ty jsme počítali v předcházejících krocích) a počtu všech cest z ní do zkoumané křižovatky (zase kombinační číslo). Tím je zaručeno, že každá špatná cesta se započítá jen pro první opravovanou křižovatku, přes kterou vede. Pak dosáhli paměťové složitosti $O(k)$ a časové $O(k^2 \cdot (m + n))$, případně, pokud si předpočítali faktoriály, paměťové $O(k + n + m)$ a časové $O(k^2 + m + n)$, což se mi zdálo méně výhodné než předchozí řešení.

Ještě bych se zastavil u velikostí k. Několik z Vás psalo, že křižovatek se v normálním městě neopravuje mnoho, čímž např. ospravedlňovali, že píší $O(m \cdot n)$ místo $O(m \cdot n + k \cdot \log k)$. Ona to sice je pravda, ale mezi počtem všech a opravovaných křižovatek platí v normálním městě přibližně přímá úměra, a potom by $O(k \log k)$ bylo $O(m \cdot n \cdot \log(m \cdot n))$.

```
#include <stdlib.h>
#include <stdio.h>
#define MAX 100000                    /* maximum z m,n a k */

typedef struct sour {int sl, r; } sour;

FILE *vstup;
```

```
int m, n, k;
int i, j, pom, pom2;
struct sour roz[MAX+5], rozusp[MAX+5];
int zac[MAX+5];
int rozvr[MAX+5];                    /* ktere kr. jsou rozbite ve zkoumanem radku */
int poccest[MAX+5];

int main (void)
{
  /* nacitani */
  vstup=fopen ("manhattan.in", "r");
  fscanf (vstup, "%d %d %d", &m, &n, &k);
  for (i=0; i<k; i++)
  {
    fscanf (vstup, "%d %d", &roz[i].sl, &roz[i].r);
    roz[i].r--; roz[i].sl--;
  }
  fclose (vstup);
  /* vytvareni seznamu */
  for (i=0; i<=n+1; i++) zac[i]=0;
  for (i=0; i<k; i++) zac[roz[i].r+2]++;    /* zac[j]...pocet rozb.kr.v radku j-2 */
  for (i=1; i<=n+1; i++) zac[i]+=zac[i-1];  /* zac[j]...pocet rozb.kr.do radku j-2 */
  for (i=0; i<k; i++)
  {
    rozusp[zac[roz[i].r+1]]=roz[i];
    zac[roz[i].r+1]++;
  }                                    /* zac[j]...pocet rozb.kr.do radku j-1...zacatek
                                          seznamu pro radek j */
  /* vypocet */
  poccest[0]=1; for (j=1; j<m; j++) poccest[j]=0;
  for (i=0; i<n; i++)
  {
    for (j=0; j<m; j++) rozvr[j]=0;
    for (j=zac[i]; j<zac[i+1]; j++) rozvr[rozusp[j].sl]=1;
    for (j=0; j<m; j++) if (rozvr[j]==1)poccest[j]=0;
                        else if (j>0) poccest[j]+=poccest[j-1];
  }
  printf ("Bill muze jet %d cestami.", poccest[m-1]);
  return 0;
}
```

15-5-4 Továrna — Dan Kráľ

Nejprve bych se chtěl za organizátory všem řešitelům omluvit, neboť v zadání úlohy se vyskytla chyba. Pro vzorový příklad je správná odpověď 23, ne 21, jak bylo v zadání uvedeno. Někteří z vás si pak vyložili zadání jinak, než bylo myšleno, a to tak, že lisování a balení tvarůžků může probíhat v libovolném pořadí či dokonce paralelně. Vzhledem k této nejednoznačnosti jsme se rozhodli těm z vás, kteří úspěšně vyřešili správné zádání, udělovat 12 bodů a těm, kteří úspěšně vyřešili špatně pochopené zadání, 11 bodů.

Povězme si nejprve pár slov k řešení úlohy, kterou jsme měli původně na mysli. Označme T počet tvarůžků, N_l počet lisovacích strojů a N_b počet balících strojů. Dále nechť λ_i, $1 \leq i \leq N_l$ a β_i, $1 \leq i \leq N_b$ jsou rychlosti jednotlivých lisovacích a balících strojů. Jako l_k budeme označovat nejmenší čas, za který je možné vylisovat k tvarůžků. Ukážeme, že hodnoty $l_1, \ldots, l_T$ lze spočítat v čase $O(T \log N_l)$. K tomu použijeme datovou strukturu zvanou halda, která je vysvětlena v následujícím odstavci.

Halda je datová struktura, která nám umožňuje hledat v zadané množině čísel nejmenší, přičemž můžeme do naší množiny čísla přidávat i odebírat. Časová složitost každé takové operace je $O(\log n)$, kde n je počet prvků v haldě. Jak je halda implementována? Halda je reprezentována vyváženým binárním stromem, ve kterém platí, že každý otec je menší než libovolný z jeho dvou synů. Tedy nejmenší prvek haldy je obsažen v kořeni binárního stromu. Nový prvek lze do haldy přidat například tak, že nejdříve vytvoříme nový list binárního stromu, a to tak, abychom neporušili jeho vyváženost. Pokud přidaný prvek je větší než jeho otec, je halda korektní. V opačném případě jej s otcem prohodíme. Pokud je menší než jeho nový otec, tak jej opět prohodíme, atd. Takto postupujeme po cestě ke kořeni, dokud v haldě nezačnou platit požadované nerovnosti. Naopak, když prvek odebereme, tak jej nahradíme prvkem z nějakého listu (opět takového, abychom neporušili vyváženost) a prohazováním na cestě z kořene nebo od kořene napravíme nerovnosti. Zřejmě každá taková operace vyžaduje čas úměrný hloubce stromu, tj. $O(\log n)$.

Nyní si popíšeme, jak lze hodnoty $l_1, \ldots, l_T$ spočítat. Vytvoříme si haldu s N_l prvky s časy, kdy nejdříve daný stroj může vylisovat další (na začátku první) tvarůžek. Čas l_1 se zřejmě rovná času nejrychlejšího stroje, tedy času uvedeného v kořeni. Tento čas nyní z haldy odebereme a nahradíme ho v haldě časem, kdy by daný stroj mohl vylisovat druhý tvarůžek. Nyní čas v kořeni haldy je roven l_2. Čas z kořene haldy odebereme a nahradíme časem, kdy daný stroj vylisuje další tvarůžek atd. To vše snadno provedeme v čase $O(T \log N_l)$. Podobně můžeme v čase $O(T \log N_b)$ určit minimální časy $b_1, \ldots, b_T$ nutné k zabalení tvarůžků.

Označme si nyní t_0 minimální čas, za který lze T tvarůžků jak vylisovat tak i zabalit. Zřejmě platí $l_i + b_{T-i+1} \leq t_0$ pro všechna $i = 1, \ldots, T$, neboť v době, kdy se vylisuje i-tý tvarůžek, musí být ještě dost času na zabalení $T - i + 1$ tvarůžků ($T - i$ ještě nevylisovaných a toho, co se právě dolisoval). Na druhou stranu, pokud budou stroje lisovat a balit tvarůžky podle harmonogramu, na základě něhož bylá čísla l_i a b_i spočtena, všech T tvarůžků bude vylisováno a zabaleno v čase $\min_{i=1,\ldots,T} l_i + b_{T-i+1}$. Tedy poté, co spočítáme čísla l_i a b_i, jediné co nám zbývá, je určit nejmenší z nich, a to snadno zvládneme v čase $O(T)$.

Celková časová složitost našeho algoritmu je $O(N_l+N_b+T(\log N_l+\log N_b))$ a paměťová $O(T+N_l+N_b)$. Poznamenejme, že opatrnější implementací výpočtu minima ze součtů $l_i + b_{T-i+1}$ lze paměťovou složitost snížit na $O(N_l + N_b)$. Implementace výše uvedených myšlenek je vcelku přímočará. Za povšimnutí snad jen stojí reprezentace binárního stromu v poli, podobně jako např. v úloze 14-1-4. Binární strom je uložen v poli indexovaném od nuly. Na pozici s indexem 0 je kořen stromu a synové prvku na pozici s indexem k jsou na pozicích s indexy $2k + 1$ a $2k + 2$. Vzhledem k tomu, že vždy odstraňujeme z haldy jen její nejmenší prvek, program neobsahuje proceduru pro odstraňování libovolného prvku haldy.

Na závěr bych ještě rád napsal pár poznámek o řešení úlohy, kdy by nebylo potřeba balit jen vylisované tvarůžky (a tedy by bylo možné tvarůžek zabalit ještě před vylisováním). V takovém případě se úloha redukuje na výpočet čísel l_T a b_T, tj. nepotřebujeme určit všechna čísla $l_1, \ldots, l_T$ a $b_1, \ldots, b_T$. Označme si nyní jako τ následující podíl:

$$\tau = \frac{T}{1/\lambda_1 + 1/\lambda_2 + \ldots + 1/\lambda_l}.$$

Zřejmě $l_T \geq \tau$. Výše uvedenou rovnost si snadno můžeme přepsat do následujícího tvaru:

$$T = \frac{\tau}{\lambda_1} + \frac{\tau}{\lambda_2} + \ldots + \frac{\tau}{\lambda_l}. \tag{1}$$

Označme si nyní jako T_0 počet tvarůžků, které lze vylisovat za čas τ:

$$T_0 = \lfloor\frac{\tau}{\lambda_1}\rfloor + \lfloor\frac{\tau}{\lambda_2}\rfloor + \ldots + \lfloor\frac{\tau}{\lambda_l}\rfloor.$$

Z (1) ihned plyne, že $T_0 \geq T-l$. Tedy τ nám umožňuje určit hodnotu l_{T_0} a nyní v čase $O((T - T_0)\log N_l) = O(N_l \log N_l)$ můžeme pomocí výše uvedeného postupu s haldou spočítat hodnoty $l_{T_0}, \ldots, l_T$. Analogicky lze v čase $O(N_b \log N_b)$ spočítat hodnotu b_T.

```
program tvaruzky;
const max_tvaruzku=1000;
const max_stroju=100;
type t_stroj=record
               stroj:integer;
              cas:longint;
             end;
var tvaruzku,stroju_lis,stroju_bal:integer;
    rychlost_lis: array[1..max_stroju] of longint;
    rychlost_bal: array[1..max_stroju] of longint;
    cas_lis: array[1..max_tvaruzku] of longint;
    cas_bal: array[1..max_tvaruzku] of longint;
    cas:longint;
    {Proměnné pro implementaci haldy}
    halda:array[0..max_stroju-1] of t_stroj;
```

```
    vhalde:integer;
procedure init_halda;
{Initializuje datovou strukturu}
  begin
    vhalde:=0
  end;
function min_halda:t_stroj;
{Vrátí a vyjme z haldy její nejmenší prvek}
  var i,j:integer;
      ts:t_stroj;
  begin
    if vhalde=0 then halt(1);
    min_halda:=halda[0];
    dec(vhalde);
    halda[0]:=halda[vhalde];
    j:=0;
    repeat
      i:=j;
      if (2*i<vhalde) and (halda[j].cas>halda[2*i].cas) then j:=2*i;
      if (2*i+1<vhalde) and (halda[j].cas>halda[2*i+1].cas) then j:=2*i+1;
      ts:=halda[i]; halda[i]:=halda[j]; halda[j]:=ts
    until i=j
  end;
procedure vloz_halda(co:t_stroj);
{Vloží do haldy další prvek}
  var i:integer;
      ts:t_stroj;
  begin
    halda[vhalde]:=co;
    i:=vhalde;
    inc(vhalde);
    while (i>0) and (halda[(i-1) div 2].cas>halda[i].cas) do
      begin
        ts:=halda[(i-1) div 2]; halda[(i-1) div 2]:=halda[i]; halda[i]:=ts;
       i:=(i-1) div 2;
      end;
  end;
var i:integer;
    ts:t_stroj;
begin
  {Načtení vstupních dat}
  readln(tvaruzku,stroju_lis,stroju_bal);
  for i:=1 to stroju_lis do read(rychlost_lis[i]);
  for i:=1 to stroju_bal do read(rychlost_bal[i]);
  {Spočítání časů l_i a b_i}
  init_halda;
  for i:=1 to stroju_lis do
    begin
      ts.stroj:=i;
      ts.cas:=rychlost_lis[i];
      vloz_halda(ts);
    end;
  for i:=1 to tvaruzku do
    begin
      ts:=min_halda;
      cas_lis[i]:=ts.cas;
      ts.cas:=ts.cas+rychlost_lis[ts.stroj];
      vloz_halda(ts);
```

```
    end;
  init_halda;
  for i:=1 to stroju_bal do
    begin
      ts.stroj:=i;
      ts.cas:=rychlost_bal[i];
      vloz_halda(ts);
    end;
  for i:=1 to tvaruzku do
    begin
      ts:=min_halda;
      cas_bal[i]:=ts.cas;
      ts.cas:=ts.cas+rychlost_bal[ts.stroj];
      vloz_halda(ts);
    end;
  {Výpočet celkového času}
  cas:=0;
  for i:=1 to tvaruzku do
    if cas<cas_lis[i]+cas_bal[tvaruzku-i+1] then
      cas:=cas_lis[i]+cas_bal[tvaruzku-i+1];
  {Vypsání výsledku}
  writeln(cas)
end.
```

15-5-5 Haskell — Zdeněk Dvořák

Chceme, aby funkce `Sum` pracovala na principu

```
Sum x y = if y == 0 then x
                    else Sum (x + 1) (y - 1)
```

Nicméně vzhledem k tomu, jak jsme si zadefinovali čísla, toto přesně dělá následující výraz:

$$Sum = \lambda x\ .\ \lambda y\ .\ y\ x\ (Sum\ (Succ\ x)).$$

Správnost dokážeme indukcí podle velikosti y:

- Pokud velikost y je nula, tj. $y = Zero$, dostáváme

$$Sum\ x\ Zero = Zero\ x\ (Sum\ (Succ\ x)) = x.$$

- Pokud $y = Succ\ y'$ a pro y' `Sum` funguje, dostáváme

$$\begin{aligned} Sum\ x\ y &= Succ\ y'\ x\ (Sum\ (Succ\ x)) = \\ &= Sum\ (Succ\ x)\ y', \end{aligned}$$

  což je podle indukčního předpokladu rovno

$$(x + 1) + y' = x + y.$$

Nyní k druhé úloze: Stejně jako operátor pevného bodu simuluje rekurzi, operátory dvojitého pevného bodu odpovídají dvěma navzájem rekurzivním funkcím. Zadání si můžeme přepsat do tvaru $a = g\ a\ b$, $b = h\ a\ b$, kde $a = Y_1\ g\ h$,

$b = Y_2\ g\ h$. Nyní z první rovnice můžeme pomocí Y vyjádřit a nerekurzivně, $a = Y\ (\lambda s\ .\ g\ s\ b)$ a dosadit do druhé rovnice, $b = h\ (Y\ (\lambda s\ .\ g\ s\ b))\ b$. Teď tedy máme rovnici pro b a opět si z ní b můžeme vyjádřit:

$$b = Y\ (\lambda t\ .\ h\ (Y\ (\lambda s\ .\ g\ s\ t))\ t).$$

Použitím definice b pak dostáváme:

$$Y_2 = \lambda g\ .\ \lambda h\ .\ Y\ (\lambda t\ .\ h\ (Y\ (\lambda s\ .\ g\ s\ t))\ t).$$

Analogicky dostaneme:

$$Y_1 = \lambda g\ .\ \lambda h\ .\ Y\ (\lambda t\ .\ g\ t\ (Y\ (\lambda s\ .\ h\ t\ s))).$$

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	25	306
1.	Milan Straka	G Strakonice	4	24	256
2.	Petr Škoda	G Ústavní, Praha	3	20	152
3.	Lukáš Turek	G Zborovská, Praha	4	17	139
4.	Jiří Danihelka	SPŠ Písek	4	15	103
5.	Jan Bulánek	G Klatovy	2	13	99
6.	David Matoušek	G Zborovská, Praha	3	15	89
7.	Jindřich Flídr	G Lanškroun	3	15	78
8.–10.	Zbyněk Falt	G Žďár nad Sázavou	2	10	75
	Stanislav Haviar	G Klatovy	2	12	75
	Peter Šufliarsky	G Nové Zámky	3	12	75
11.	Bejnamin Vejnar	G Nymburk	3	13	70
12.	Jaroslav Havlín	GOA Sedlčany	3	10	64
13.	Dalibor Zelený	G Česká Třebová	4	15	56
14.	Jiří Štěpánek	G Tř. kpt. Jaroše, Brno	3	10	55
15.	Jan Křetínský	G M. Lercha, Brno	3	7	54
16.–18.	Martin Dobroucký	G Moravská Třebová	2	8	51
	Filip Šauer	G Klatovy	2	10	51
	Petr Turbek	G Beroun	3	13	51
19.	Stanislav Basovník	G Kroměříž	2	7	49
20.	Ondřej Garncarz	G Příbor	2	13	48
21.	Jan Kaštil	G Přerov	4	9	41
22.	Jana Fabriková	G Tř. kpt. Jaroše, Brno	3	6	38
23.	Jan Kadlec	G Zborovská, Praha	4	5	33
24.	Michal Bečka	G Moravská Třebová	3	9	32
25.–26.	Hana Kozelková	G J. G. Mendela, Brno	4	5	29
	Jan Matoušek	G Prostějov	3	4	29
27.–28.	Tomáš Kučera	G Voděradská, Praha	4	5	23
	Vojtěch Šádek	G Hranice na Moravě	3	6	23
29.	Tomáš Gavenčiak	G Bílovec	3	4	22
30.	Kristýna Knapová	G Jičín	3	6	20
31.–33.	Václav Cviček	G Frýdek-Místek	4	3	19
	Daniel Marek	????	0	3	19
	Jakub Nezveda	SPŠE V Úžlabině, Praha	2	5	19
34.–35.	Josef Sedlačík	G Uherský Brod	4	2	17
	Jan Sedo	SPŠ Jihlava	1	3	17

36.–37.	Hynek Hanke	G Budějovická, Praha	3	5	16
	Marek Sterzik	SPŠ Ostrov	4	2	16
38.	Pavel Troubil	G Tř. kpt. Jaroše, Brno	3	2	15
39.–40.	Petr Baudiš	G Jihlava	3	4	14
	Martin Dobiaš	G Trenčín	4	3	14
41.–42.	Martin Šuška	G Martin	4	2	13
	Lukáš Vlk	SOŠT Glaverbel, Teplice	4	3	13
43.–44.	Marcel Dopita	G Tábor	2	3	12
	Martina Tomišová	G Zborovská, Praha	3	3	12
45.	Aleš Kresta	G Frýdek-Místek	4	2	11
46.	Oto Petřík	GOA Vrchlabí	2	2	10
47.–49.	Pavel Kocourek	SPŠST Panská, Praha	2	3	9
	Jozef Matějička	G Žilina	4	2	9
	Ján Palenčár	G Martin	7	2	9
50.–52.	Jan Doubek	G Vimperk	3	1	8
	Vojtěch Kovář	G Břeclav	3	2	8
	Petra Malá	G Moravský Krumlov	1	1	8
53.–55.	Jana Babováková	G Most	3	3	7
	Petr Kortánek	GOA Sedlčany	1	3	7
	Pavel Srb	G Karlovy Vary	3	1	7
56.	Jaromír Vojíř	G Ledeč nad Sázavou	2	2	6
57.–58.	Martin Lopatář	G Tř. kpt. Jaroše, Brno	3	1	5
	Pavel Vlašánek	SPŠ Bruntál	2	1	5
59.	Jindřich Pelíšek	G Jeronýmova, Liberec	2	1	3

Obsah

Martin Mareš a kolektiv

Korespondenční seminář z programování XV. ročník

Autoři a opravující úlohy:
Peter Bella, Jakub Bystroň, Josef Cibulka,
Zdeněk Dvořák, Jan Kára, Daniel Kráľ,
Pavel Machek, Martin Mareš, Miroslav Rudišin,
Marek Sulovský, Pavel Šanda, Tomáš Valla,
Tomáš Vyskočil

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 2003

Písmem Computer Modern v programu TEX vysázel Martin Mareš.

Ilustrace na titulní straně vytvořili:
Hayley Flump, Axel Krcek, Martin Mareš,
Philip Powell, Jens Reissenweber a Joan G. Stark.

Korektury provedla Karolína Šimková.

Vytiskla Tiskárna Graphis – P. Flodr.

108 stran
Vydání první
Náklad 300 výtisků