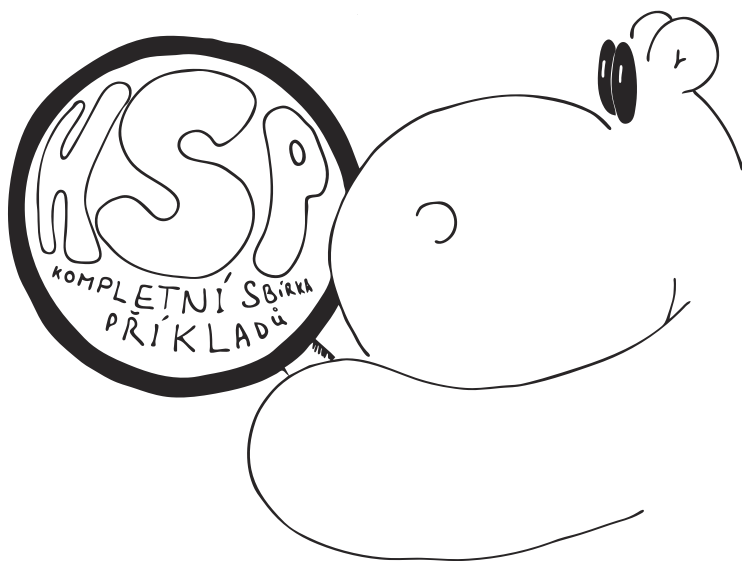
MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XIX. ročník – 2006/2007



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XIX. ročník – 2006/2007

matfyzpress

Praha 2007

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

Copyright © (Ed.) Petr Kratochvíl, 2007
© MATFYZPRESS, vydavatelství Matematicko-fyzikální fakulty
Univerzity Karlovy v Praze, 2007

ISBN 978-80-7378-021-0

Úvod

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

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání obvykle šesti úloh, v klidu domácího krbu je (ne nutně všechny, počítají se nejlepší čtyři) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na naši adresu (ať už fyzickou či elektronickou). My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

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

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

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

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

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

Zadání úloh

Je trochu malý zázrak, že teď čtete tyto řádky. Řádky, na nichž vás očekává příběh o mně, detektivu Přesprstovi. A tak Ti ještě jednou děkuji, mé milé KSP. Nejen že jsi opravilo chyby v mém rukopisu, ale dokonce jsi bylo tak laskavo a jako jediné jsi mé dílko vydalo.

19-1-0 Vzorová úloha

Poznámka KSP: To si zaslouží vysvětlit. Rukopisy pro náš speciální tiskařský lis je třeba psát v následující formě:

```
name jméno_kapitoly
        text kapitoly
        name jméno_podkapitoly
                text kapitoly
        end
        text kapitoly
        name jméno_další_podkapitoly
                text kapitoly
                name jméno_vnořené_podkapitoly
                        text kapitoly
                end
        end
        text kapitoly
end
```

Každá kapitola tedy začíná řádkem, jehož prvním slovem je `name`, a končí řádkem obsahujícím jediné slovo `end`. V každé kapitole pak může být libovolný počet podkapitol s totožnou syntaxí. V čem je problém? Přesprst totiž občas nedodržoval tuto syntaxi a slova `name` a `end` psal víceméně nahodile. Buď začal novou kapitolu, aniž by ukončil předchozí, nebo ukončil kapitolu, kterou ani nezačal. A chudák mistr tiskař, který nevěděl, co si s rukopisem počít, ze samého zoufalství počal jíst barvu. Je jasné, že mistrů tiskařů zase nemáme tolik, a tak bychom rádi poznali, že rukopis je chybný, abychom ho mohli Přesprstovi vrátit.

19-1-0 Vzorové řešení vzorové úlohy – Jak na to?

Nejprve se zamyslíme, co se vlastně může pokazit. Můžeme neukončit existující kapitolu a ukončit neexistující kapitolu. Dále by se mohlo stát, že se nějaké kapitoly „překříží“, čili že ukončíme nějakou kapitolu dříve než její podkapitolu. Ale počkat – jednotlivé `end` nejsou pojmenované a každý patří k „nejbližšímu“ `name` nad ním, takže nemůžeme ukončit kapitolu předtím, než ukončíme její podkapitolu. Jména kapitol nejsou tím pádem vůbec důležitá, roli hraje pouze jejich vnořený počet.

Když jsme si to uvědomili, úlohu již vyřešíme jednoduše. Stačí nám pamatovat si, kolik kapitol zatím začalo a nebylo ještě ukončeno. Každý `name` na vstupu zvýší tento počet o jedna, každý `end` ho o jedna sníží (pokud může). Správný text pak poznáme tak, že každý `end` snižuje nenulový počet začatých kapitol a navíc je na konci tento počet nulový. Řešení v pseudopascalu může vypadat například takto:

```
počet_kapitol := 0;
while not konec_textu do begin
  if řádek_začíná_na_name then inc(počet_kapitol);
  if řádek_začíná_na_end then
    if počet_kapitol=0 then begin
      write('Ukončena neexistující kapitola!'); halt;
   end else dec(pocet_name);
end

if počet_kapitol<>0 then write('Neukončená kapitola!')
else write('Rukopis je v pořádku.');
```

Nyní určíme časovou a paměťovou náročnost našeho řešení. Řešení načítá řádky textu, každý právě jednou, a s každým řádkem provede jednoduchou operaci (zjistí, zda začíná slovem `name` či `end` a případně příslušně upraví proměnnou `počet_kapitol`) v čase úměrném délce tohoto řádku. Celkový čas našeho řešení je tedy lineární vzhledem ke vstupnímu textu, což je jistě asymptoticky nejlepší možné, protože v menším než lineárním čase bychom ani nedokázali načíst celý vstup. Paměti potřebujeme jenom konstantní množství (proměnná `počet_kapitol` a čtyři první nemezerové znaky načítaného řádku pro porovnání s `name` a `end`), což také můžeme těžko zlepšit. Naše řešení je tedy v určitém smyslu „nejlepší možné“. Hurá!

Můj příběh se odehrává v dobách, kdy dané slovo mělo větší váhu než tisíc smluv a přijít o čest bylo horší než přijít o život. A v mnoha ohledech i těžší. Být soukromým očkem v divočině mafiánských rodin bylo pořádně nebezpečnou hrou a člověk si musel dávat sakra, ale sakra pozor, aby nešlápl na něčí kuří oko, jestli mi rozumíte. Jenže občas stačí jen chvilka nepozornosti a…

Ten den jsem seděl v kanceláři. Stejně jako den předtím. A i den předtím. Vlastně už si ani nevzpomínám, kdy jsem měl trochu víc případů a hlavně teda peněz.

19-1-1 Zlaté časy — 8 bodů

Pojďme Přesprstovi pomoci zjistit, kdy byly jeho zlaté časy, aby si mohl zavzpomínat, a nahlédněme do jeho knihy příjmů. V té si pečlivě vedl své příjmy (kladná čísla) a výdaje (záporná čísla). No a zlaté časy je přirozeně takové souvislé období, kdy dosáhl nejvyššího součtu příjmů a výdajů.

Vstupem bude posloupnost příjmů a výdajů a na výstupu by měl váš program vypsat číslo záznamu, kdy zlaté časy začaly, a číslo záznamu, kdy zlaté časy skončily.

Příklad: Pro posloupnost $1, 6, -13, 12, 3, -2, 1, 5, 6, -4$ se Přesprstovi nejlépe vedlo mezi 4 a 9 (tehdy byl součet 25).

Ještě že mi zbylo alespoň na mou oblíbenou whisky, doutníky a čokoládu. Čokoládu… Kde jen ji mám. Začal jsem se přehrabovat v šuplíku. Konečně jsem ji našel a třesoucíma se rukama ji začal lámat.

19-1-2 Čokoláda — 6 bodů

Třesoucíma se rukama se ale láme špatně, a tak by Přesprsta zajímalo, kolikrát bude muset lámat. Představte si tabulku čokolády a dvě ruce. Ruce uchopí čokoládu a rozlomí ji (zlom je úsečka vedená zúženým místem mezi dílky) na dvě ne nutně stejně velké části. Poté ruce uchopí jednu z částí a opět ji rozlomí na dva kusy. A Přesprsta by zajímalo, kolikrát je třeba lámat v nejlepším a v nejhorším případě, aby získal jednotlivé dílečky tabulky čokolády.

Pomůžete mu? Na vstupu dostanete dvě čísla N a M, což jsou rozměry tabulky čokolády v dílcích. Na výstupu by měla být rovněž dvě čísla: minimální a maximální počet zlomů nutných k dosažení cíle. A protože Přesprst už nikomu nedůvěřuje, měli byste mu i dokázat, že na menší a větší počet zlomů to nejde.

Příklad: Má-li Přesprst čokoládu o rozměrech 1×3, musí v každém případě lámat dvakrát, protože jedním zlomením jednotlivé dílky čokolády nedostane, a třikrát lámat nelze.

A je to. Vložil jsem hladově jeden dílek do úst. Pak jsem si nalil whisky, sedl si a čokoládu zapil. Jak mě whisky tak hladila po jazyce, hledal jsem východisko ze své situace a asi bych se prohledal až ke dnu lahve, kdyby někdo nezaklepal.

„Dále,“ promluvil jsem překvapeně ke dveřím.

„Dobrý den, mohu se posadit?“

Mlčky jsem kývl a zavřel pusu.

Nebudu vás otravovat popisem té dámy, která zabloudila do mé kanceláře, byla prostě kus.

„Co máte na srdci?“ prohodil jsem nedbale a nenápadně jsem si ji prohlížel v místech, kde jsem tušil srdce.

To, co mi ta chudinka říkala, mě ani nepřekvapilo. Jenom jsem netušil, proč mi to všechno vykládá. Byla ženou jednoho z místních mafiánských kmotrů, Carla Assassina. Vyprávěla o tom, jak její manžel obchoduje s alkoholem a zbraněmi, pere špinavé a tiskne falešné peníze a pořádá večírky pro podsvětní smetánku.

19-1-3 Tiskárna 10 bodů

Během vyprávění se Carlova žena rozpovídala o tom, jak se ony falešné peníze tiskly. Tiskárna měla vstup a výstup. Na vstup se daly bankovky a na výstupu se objevily bankovky původní a ještě jedna kopie každé z nich. Tedy přesně dvojnásobný počet. Když se všechny bankovky okopírovaly, dal se celý balík z výstupu zpět na vstup a na vršek se přidala jedna bankovka s novým sériovým číslem. Carlově ženě se podařilo získat všechny bankovky ze vstupu ještě před tím, než tiskárna začala kopírovat, a Přesprstovi je donesla. Přesprsta by teď zajímalo, která bankovka se přidala naposledy na vršek, tedy ta bankovka, která má v celém balíku unikátní sériové číslo. Bohužel se během cesty promíchalo pořadí bankovek.

Na vstupu dostane program seznam sériových čísel bankovek přinesený Carlovou ženou a na výstupu by měl program vypsat sériové číslo bankovky, která byla přidána jako poslední, tedy takové, která je v seznamu právě jednou, zatímco ostatní lze spárovat do dvojic. Sériové číslo je řetězec kratší než 100 znaků obsahující pouze číslice a velká písmena anglické abecedy.

Příklad: Pro vstup `9G873W`, `Z8D43`, `9G873W`, `9G873W`, `A456C`, `Z8D43` a `9G873W` by měl program napsat `A456C`.

Všechno trpělivě snášela a snažila se do jeho věcí nemíchat, ale to, že si našel jinou ženu, pro ni byla asi poslední kapka k tomu, aby se do jeho věcí míchat začala. Život je boj, pomyslel jsem si, a vstal jsem, abych ji utěšil, neboť vypadala, že se každou chvilkou zhroutí…

Nevím, co přesně se v tu chvíli stalo, vím jen, že se rozrazily dveře a já se probudil svázaný v temné místnosti. Nejprve jsem si jazykem přepočítal zuby a pak jsem si překontroloval všechny kosti. Zdálo se, že až na bolesti hlavy se mi nic nestalo. Začal jsem přemýšlet, komu jsem stál za únos. Ve městě byla spousta mafiánských rodin. Popravdě řečeno, nikdo nevěděl přesně kolik. A přitom bylo tak snadné je spočítat.

19-1-4 Mafiánské rodiny 10 bodů

Policie si vedla o každém mafiánovi záznam o jeho přímých nadřízených a podřízených. Každý mafián má nejvýše jednoho přímého nadřízeného, pokud nadřízeného nemá, je to kmotr. Každá rodina má právě jednoho kmotra. No, zas tak snadné to nebylo, protože na popud rodiny Farmot byly spisy poničeny vandaly a nebylo možné vyčíst, jestli je záznam ve tvaru šéf – podřízený nebo naopak.

Na vstupu dostane váš program seznam policejních záznamů a na výstupu by měl program vypsat počet rodin ve městě.

Příklad: pro záznamy: $1-2, 2-1, 3-6, 4-6, 5-6, 6-3, 6-4, 6-5, 5-7, 7-5$ lze zjistit, že ve městě jsou dvě rodiny.

Nestihl jsem si ani v duchu přeříkat jména rodin, jimž bych se nechtěl znelíbit, když tu mě z uvažování vytrhly hlasy.

„Já bych navrhoval nejdžíve thání nechtů, pak vlasů, pak usekat pfsty a… “

„Neblbni má to přežít. Aspoň napoprvé.“

„Aha, na to šem šapomněl, tak co tšeba vyřžazit žubi a žlámat nohi?“

„Ne, to už jsme dělali minule.“

„Ale žubi byšme mohli, né? Mě to proště baví a š těma novejma kleštiškama, co nám pořžídil šéfík, to jde jako po mášle.“

„Ach jo. Si jak malej. Tak jo, ale jen tak, aby mohl mluvit.“

„Hulá! Mušeníško, to je moje potěšeníško.“

Snažil jsem se je moc nevnímat, což stejně moc nešlo, protože jsem se všemožně snažil uniknout ze svých pout.

19-1-5 Zámek — 13 bodů

Přesprst měl štěstí, únosci použili pouta s číselným zámkem, který on velmi dobře znal. Věděl, že tento typ zámku lze vždy odemknout jednou z kombinací, které splňují podmínku, že po nějaké číslici následuje pouze číslice z určité množiny. Dokonce si ty seznamy pro jednotlivé číslice dobře pamatoval a teď by ho zajímalo, kolik různých kombinací musí prozkoumat.

Na vstupu dostane program K, počet cifer čísla na zámku, a seznam číslic, které mohou následovat za jednotlivými číslicemi. Na výstup by měl program vypsat, kolik všech možných čísel je na zámku možno nastavit.

Příklad: Pro tříciferné číslo ($K = 3$) a seznam

cifra	*možné následující cifry*
1	$1, 3$
2	3
3	1

je počet různých kombinací 6 $(111, 113, 131, 231, 311, 313)$.

Už jsem měl odzkoušeno několik desítek kombinací, když najednou hlasy utichly a otevřely se dveře…

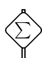
19-1-6 Prolog — 12 bodů

V letošním seriálu se budeme zabývat poněkud zvláštním, ale velmi zajímavým programovacím jazykem *Prolog*. Většina programovacích jazyků, které znáte (Pascal, C) patří do skupiny tzv. *procedurálních jazyků*. Programátor píšící kód v procedurálním jazyce přesně popíše, jakým způsobem se má daná úloha vyřešit. V Prologu budeme programovat jinak, logicky. Nejprve nějakým způsobem popíšeme nám známý svět a poté se Prologu zeptáme na řešení daného problému. Nepřikazujeme tedy, jak se má Prolog dobrat výsledku, pouze

říkáme, co chceme vyřešit, ale ne přesně, jakým způsobem. Zapomeňme tedy na chvíli na klasické proměnné coby „škatulky“, na přiřazovací příkaz a na alokování paměti. Programujeme logicky – *PROgramming in LOGic*.

Jak na Prolog

Aby se vám s Prologem seznamovalo co nejlépe a nejradostněji, připravili jsme pro vás kromě klasického „papírového“ seriálu také internetovou prologovskou poradnu. Na adrese *http://ksp.mff.cuni.cz/prolog/* najdete fórum, kam můžete posílat své otázky a my vám s vaším problémem poradíme. Nestyďte se ptát, rádi vám pomůžeme. Kromě poradny najdete na uvedené adrese také zadání úloh, učební texty, užitečné odkazy a také on-line interpreter Prologu, kde si můžete zkoušet své prográmky.



Poznámka: Tento symbol označuje obtížnou část, která ale není nutná pro pochopení dalšího textu.

Instalace Prologu

Rady k instalaci a odkazy na Prolog jak pro Windows, tak pro Linux najdete také na *http://ksp.mff.cuni.cz/prolog/*.

Program v Prologu, predikát, fakt, klauzule, proměnná

Program v Prologu je možné napsat buď přímo v prostředí Prologu, pokud to vaše prostředí umožňuje, nebo v libovolném textovém editoru (vim, emacs, ...). Ukážeme si příklad jednoduchého prográmku.

```
muz(antoch).
muz(bonifac).
zena(cecilka).
rodic(cecilka,antoch).
rodic(bonifac,antoch).
manzele(bonifac,cecilka).
```

Prolog popisuje situaci pomocí *predikátů*. Predikát `muz(X)` říká „`X` je muž“. V našem prográmku máme čtyři predikáty: *unární* predikáty `muz(X)` a `zena(X)` a *binární* predikáty `manzele(X,Y)` a `rodic(X,Y)`.

Proměnné se v Prologu značí velkým písmenem na začátku, např. `X`, `Y`, zatímco konkrétní hodnoty – *atomy* (můžete si je představovat trošku jako stringy) začínají malým písmenem, např. `antoch`, `bonifac`.

Takže jakmile jsme v programu použili predikát `muz` a „dosadili“ do něj `antoch`, dali jsme světu najevo, že `antoch` je muž.

Základní jednotkou prologovského programu je *klauzule*. Klauzule vždy končí *tečkou*. V našem programu jsme zatím použili nejjednodušší typ klauzulí, *fakta*.

Jakmile jsme napsali náš první program v Prologu, můžeme si jej pustit v prologovském prostředí. Prolog je interaktivní jazyk, takže po spuštění se objeví výzva:

```
?-
```

Nejprve musíme Prologu sdělit, že si přejeme pracovat s naším programem. To uděláme pomocí

```
?-['nasprogram.pl'].
```

Teď začneme konečně náš program využívat. Zeptejme se, jestli je `antoch` muž.

```
?-muz(antoch).
```

Co nyní Prolog udělá? Projde námi dodaný program a podívá se, jestli v něm existuje predikát `muz(antoch)`. A odpoví nám:

```
?-muz(antoch).
    yes.
?-
```

Poznámka: `muz(antoch)` jsme napsali my. Nesmíme zapomenout na tečku na konci. Pak stiskneme Enter a výsledné `yes` napsal zase Prolog, pochopitelně. Nakonec se znovu vypíše otazník a čeká se na další dotaz. Kdyby Prolog nenašel v programu `muz(antoch)`, odpověděl by `no`.

Můžeme se ale zeptat jinak:

```
?-muz(X).                         % kdo je muž?
    X=antoch.                     % antoch je muž
```

Prolog opět projde celý program a pokusí se za `X` „dosadit“, správně říkáme *unifikovat*, někoho, kdo je muž, a nabídne nám `antoch`a. Kdyby žádného muže nenašel, odpoví nám `no`.

Pokud jsme s `antoch`em spokojení, dáme Enter a program odpoví `yes` a opět otazníkem čeká na další dotaz. Může se ale stát, že nechceme `antoch`a, nýbrž `boniface`. Pak můžeme tuto odpověď odmítnout a vyzvat Prolog, aby našel jiného muže tím, že zmáčkneme středník:

```
?-muz(X).                         % kdo je muž?
    X=antoch.   ;                 % chceme dalšího muže
    X=bonifac.  ;                 % ještě dalšího muže
    no.                           % už žádný není
```

Jistě víte, co dělá dotaz

```
?-rodic(bonifac,X).
```

Vidíme, že dotazy činíme pomocí proměnných. Na začátku je proměnná `X` *volná*, tedy nevíme, kdo je dítě `boniface`. Proměnná `X` zatím *není svázaná*.

Prolog zjistí, že dítětem `boniface` je `antoch` a *sváže* neboli unifikuje proměnnou `X` s `antoch`em.

```
X=antoch.
```

Pravidla

Zatím by se mohlo zdát, že Prolog je jen šikovná databáze. Bylo by opravdu slabé, kdybychom k dispozici měli jen klauzule typu fakta a mohli se na ně jen ptát. Existují tedy ještě klauzule typu *pravidla*. Přidejme do programu řádku:

```
je_otec(X) :- rodic(X,Y), muz(X).
```

Tohle pravidlo říká „`X` je otec, pokud platí predikáty, že `X` je něčí rodič (vyskytuje se v predikátu rodič) a ještě k tomu `X` je muž“. *Čárka* mezi predikáty má význam *a zároveň*. Tedy aby se splnilo, že je někdo otcem, musím splnit oba predikáty na pravé straně.

Poznámka: Je jasné, že za predikáty na pravé straně se mohou skrývat další pravidla.

Vyhodnocení dotazu, unifikace volných proměnných

Jak tedy Prolog vyhodnotí dotaz:

```
?-je_otec(cecilka).
```

Nejprve zjistí, že `cecilka` je skutečně rodičem (rodičem `antoch`a, ale to nás moc nezajímá), a pak se snaží splnit také predikát, že `cecilka` je muž, což se mu nepovede, a proto skončí s `no`. Měli jsme splnit oba dva predikáty a to se nám nepovedlo.

A co dotaz:

```
?-je_otec(X).
```

To už je trošku zapeklitější. Prolog ví, že musí splnit nejprve predikát `rodic`, takže se podívá, kdo je rodičem. Nejprve mu „padne do oka“ predikát `rodic(cecilka,antoch)`, vybere si tedy `cecilk`u. Jinými slovy unifikuje `X=cecilka`. Pak hledá, jestli je `cecilka` mužem, jenže zjistí, že `cecilka` není mužem, proto `cecilka` nebyla ta správná volba. A teď přichází kouzlo Prologu: Prolog *odunifikuje* `X=cecilka`, tedy uvolní proměnnou `X` a zkusí novou volbu, tedy znovu unifikuje `X=bonifac`, zkusí to s `bonifac`em a tentokrát uspěje, neboť `bonifac` je rodič i muž. V tomto okamžiku samozřejmě vypíše `X=bonifac`.

Když se tedy Prolog snaží splnit nějaký predikát a má nějakou volnou, nesvázanou proměnnou `X`, zkusí za ni „dosadit“, unifikovat nějakou hodnotu. Nejprve vybere tu, která je v programu na nejbližším řádku od začátku programu. Když má proměnnou `X` svázanou s nějakou hodnotou, zkusí splnit všechny predikáty, které mu přikazuje pravá strana pravidla, a používá přitom tuto zvolenou hodnotu `X`. Když se to podaří, skončí s úspěchem a vypíše tuto zvolenou hodnotu proměnné `X` jako správnou. Když Prolog někde narazí na nesplnitel-

ný predikát (prostě `cecilka` holt není muž), musí se vrátit a proměnnou `X` odunifikovat. `X` je zase volná. Pokud máme na výběr ještě nějaké jiné hodnoty, zkusíme proměnnou `X` znovu zunifikovat s jinou hodnotou (další v pořadí v programu) a vyhodnotit predikáty znovu. Pokud takto vyčerpáme všechny možnosti, musíme bohužel skončit neúspěšně a vypsat `no`.

Poznámka: Vidíme, že neexistuje žádná jiná možnost, jak změnit hodnotu proměnné, než že se v průběhu vyhodnocování odunifikuje při návratu z neúspěšné větve a zunifikuje při vstupu do nové větve. Do jednou zunifikované proměnné neumíme už „přiřadit“ novou hodnotu.

Ve skutečnosti je tento popis dost nepřesný, Prolog neunifikuje zvlášť proměnné, ale celý predikát s hlavou klauzule. (Že to zní ďábelsky :–)

Vyhodnocení dotazu, predikáty

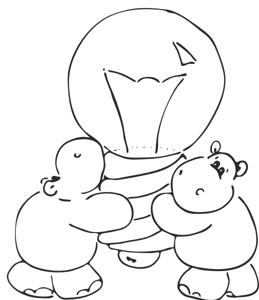
Stejným způsobem, jako Prolog postupně zkouší unifikovat volné proměnné, pracuje i s výběrem predikátů. Ukážeme si příklad. Máme program:

```
kocka(micka).
pes(alik).
zelva(matylda).
je_savec(X) :- kocka(X).
je_savec(X) :- pes(X).
```

Zeptejme se:

```
?-je_savec(alik).
```

Prolog nejprve zkusí klauzuli `je_savec(X) :- kocka(X)` s alíkem a samozřejmě zjistí, že alík není kočka. Ale nevzdá se tak snadno, protože má na výběr ještě jednu variantu predikátu `je_savec(X) :- pes(X)` a tam s alíkem uspěje.



```
?-Kolik programátorů v Prologu
  je potřeba na výměnu žárovky?
   no.
```

Pro Prolog predikáty se stejným názvem a stejným počtem argumentů jaksi „patří k sobě“ a postupně je vyzkouší všechny.

Pozor, predikát `je_savec(X,Y)` je jiný predikát, Prolog tedy rozlišuje predikáty stejného názvu a rozdílného počtu argumentů.

Poznámka: Místo rozepsání na dva řádky můžeme napsat také:

```
je_savec(X) :- kocka(X) ; pes(X).
```

Středník má tedy při splňování pravé strany pravidla význam *nebo*.

Anonymní proměnná

Vzpomeňme si na predikát `je_otec`:

```
je_otec(X) :- rodic(X,Y), muz(X).
```

V predikátu je_otec nám šlo o to, zda je někdo otcem, ale nezajímalo nás, čí otec to je. Je nám popravdě úplně jedno, jak se ono dítko jmenuje, hlavně, že nějaké je. Proměnná `Y` je tady vlastně docela zbytečná a může v ní být cokoliv. To můžeme vyjádřit tzv. *anonymní proměnnou*:

```
je_otec(X) :- rodic(X,_), muz(X).
```

Anonymní proměnná se značí `_` (podtržítkem). Pokud máme v klauzuli více anonymních proměnných, tak spolu nemají vůbec žádný vztah, i když jsou všechny značeny podtržítkem. Prostě je to hromada proměnných, z nichž každá může mít jakoukoli hodnotu a ta nás nezajímá.

Porovnávání

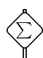
Zjistit, jestli se dvě proměnné rovnají, není v Prologu tak jednoduché. Záleží na tom, jestli už jsou proměnné zunifikované a co v nich vlastně je. Pro naše účely zatím stačí vědět, jak porovnáváme obsahy dvou již zunifikovaných proměnných, které obsahují atomy (například jména).

```
jsou_stejne(X,Y) :- X=Y.                % Prolog greenhorn
```

Predikát `jsou_stejne` uspěje, pokud je `X` rovno `Y` v tom smyslu, jak bychom čekali, tedy pokud atomy v nich jsou stejné. Namísto pravidla můžeme takovéto porovnání vlastně provést ještě jednodušeji:

```
jsou_stejne(X,X).                       % Prolog guru
```

Tento predikát uspěje, pokud dostane dvě proměnné svázané se stejnými hodnotami atomů.



A co by se stalo, kdybychom do predikátu `jsou_stejne` pustili proměnnou `X` zunifikovanou třeba `X=pavel` a proměnnou `Y` zatím nezunifikovanou? Odpověď se nabízí – `Y` by se zunifikovala na `Y=pavel` a predikát by uspěl.

Rozsah platnosti proměnných

Platnost proměnné se omezuje na jednu klauzuli. Tedy proměnná `X` je platná (čili jedna a tatáž) v klauzuli

```
je_otec(X) :- rodic(X,_), muz(X).
```

Ale máme-li klauzule s predikáty

```
je_otec(X) :- ...
je_matka(X) :- ...
```

`X` v jedné a druhé klauzuli jsou různá.

Představme si, že bychom chtěli naprogramovat predikát `prarodic(X,Y)`, který by dokázal zjišťovat, kdo je čím prarodičem, resp. kdo je čím vnukem nebo vnučkou, a případně zdali udaná dvojice má vztah prarodič–vnuk. Jak na to? Uvědomíme si, co vlastně znamená být prarodičem. Prarodičem jste, pokud máte dítě a toto dítě má zase dítě. Zapsáno v Prologu:

```
prarodic(Pra,Vnuk) :- rodic(Pra,Rod),
    rodic(Rod,Vnuk).
```

Napsat babičku a dědečka by pro vás jistě bylo jednoduché.

Rekurze

Chtěli bychom napsat predikát `predek(Pred,Pot)`, který bude zjišťovat, zdali je `Pred` předkem potomka `Pot`, čili jeho rodič, prarodič, praprarodič atd. V příkladu s prarodičem jsme dopředu věděli, že hledáme vztah přes jednu generaci a také jsme tak daný predikát napsali. Jenže teď nemáme ani tušení, kolik generací může mezi Pred a Pot být. Pomůže nám rekurze:

```
predek(Pred,Pot) :- rodic(Pred,Pot).
predek(Pred,Pot) :- rodic(Pred,X),predek(X,Pot).
```

A zeptáme se:

```
?-predek(anna,kvetos).
```

Jak už víme, Prolog se nejdřív podívá na první řádek a zjistí, jestli náhodou `anna` není přímo rodičem `kvetose`. Rodič je přeci také předek. Pokud `anna` opravdu je rodičem `kvetose`, problém je vyřešen a končíme s `yes`.

Dobře, ale co když zjistíme, že `anna` není přímým rodičem `kvetose`? Pak se musíme zamyslet nad tím, co znamená být předkem: `anna` je předkem `kvetose`, pokud má `anna` nějaké dítě `X`, které je předkem `kvetose`. Prolog najde nějaké dítě `anny`, třeba `pavla`. Pak vezme `pavla` a `kvetose` a zkoumá predikát `predek(pavel,kvetos)`. Opět najde třeba nějakého `cyrila`, který je dítětem `pavla` a měl by být předkem `kvetose`. Takto pokračuje dál a dál, až najde celý řetězec dětí a pradětí `anny`, začínající `pavel`, `cyril`, ... a poslední dítě je přímým rodičem `kvetose`.

Samozřejmě v každé generaci může mít Prolog na výběr spoustu dětí, ale on je všechny vyzkouší (to už víme, postupně unifikuje proměnné), a tak prohledá celý generační strom a najde cestu od `anny` ke `kvetosovi`. (Samozřejmě pouze pokud nějaká existuje.)

Kvíz

Vyzkoušejte si, co si vám utkvělo v paměti. Správné výsledky s vysvětlením jsou na *http://ksp.mff.cuni.cz/prolog/*.

⋆ Jakým písmenem může začínat proměnná

1. velkým písmenem
2. malým písmenem
3. podtržítkem

⋆ Označte řádek, na kterém je právě jedna klauzule

1. `pes(alik). pes(brok).`

2. `pes(hafistek).`
3. `savec(X) :- pes(X).`

⋆ Jaký je vztah pravidla a faktu

1. Fakt je pravidlo, které nemá žádnou pravou stranu.
2. Fakt je pravidlo, které vždy uspěje.
3. Mezi faktem a pravidlem není žádný vztah.

⋆ Jakého rodinného příslušníka hledá `prislusnik(X)`?

```
manzele(X,Y).
rodic(X,Y).
prislusnik(X) :- manzele(A,B), rodic(A,Y),
    rodic(B,Z), Y=Z, X=Y.
```

1. každého rodiče, který je v manželském svazku
2. všechny manželské děti
3. všechny manžele, kteří mají vnuka

⋆ Uspěje dotaz `pred(_)`, pokud máme program:

```
pred(a).
```

1. Uspěje.
2. Neuspěje.

Soutěžní úlohy

1. **Tchyně (2 body)** Napište predikát `tchyne(Tch,X)`.
   Vysvětlení pro ty, kdo neví, co je tchyně: Pokud je někdo ženatý/vdaná, tak tchyně je matka jeho/jejího partnera/partnerky.
2. **Oprava (3 body)** Popište, proč tento program nefunguje, a zkuste jej opravit, aby fungoval tak, jak nejspíš zamýšlel autor:

   ```
   predek(Pred,Pot) :- predek(MlPred,Pot),
       rodic(Pred,MlPred).
   predek(Rod,Pot) :- rodic(Rod,Pot).
   ```

3. **Evoluce (7 bodů)** Biologové vás požádali o řešení následujícího problému.
   Existuje databáze rostlin, ve které jsou uloženy informace o tom, která rostlina se vyvinula z které. Máte tedy predikát `mutace(X,Y)`, který popisuje, že rostlina `Y` se vyvinula z rostliny `X` mutací. Biologové vědí, že některé rostliny jsou nejpůvodnější, takže nemají žádného evolučního předka. Lze rozpoznat, které rostliny to jsou, takže máte k dispozici predikát `je_puvodni_druh(X)`, který uspěje, pokud

byla rostlina `X` na začátku evoluce. Dále máte k dispozici predikát `je_odvozeny_druh(X)`, který uspěje, pokud je rostlina odvozená od nějaké původní (jinými slovy není původní). Je dokázáno, že každá rostlina se vyvinula mutací z právě jedné rostliny, tudíž neexistuje křížení. Každá rostlina tedy odvozuje svůj původ od jedné z evolučně původních rostlin. Biologa by zajímalo, jestli daná dvojice rostlin odvozuje svůj původ od stejné původní rostliny.

Napište predikát `stejny_druh(X,Y)`, který uspěje, pokud rostliny `X` a `Y` odvozují svůj původ od stejného druhu od počátku evoluce.

Posílejte i nefunkční a částečná řešení, bodové odměny budou i za ně!

Rozloučení

Děkujeme vám za pozornost a doufáme, že se na nás brzy obrátíte se svými dotazy. Nakonec se s vámi rozloučíme příkazem pro ukončení Prologu:

```
?-halt.
```

Ve dveřích stála má nová klientka a dvě nadřený gorily. Něžně ji hodily na podlahu vedle mě a zazubily se. Teda, ony se moc nezubily, protože neměly čím. A hned jsem pochopil důvod – obě v ruce třímaly tabulku čokolády. V tom mě osvítil nápad hodný mého génia.

„Čo ťak blbě čumíš?“ zeptal se mě ten kolozubější.

„Koukám se, co držíš v ruce, a tak mi napadá, to jíte tu čokoládu jen tak? Vy nevíte, co legrace se s ní dá užít, než jí sníte?“

Tázavé přiblblé kukuče mi prozradily, že netuší, a tak jsem začal s výkladem pravidel:

19-2-1 Čokoláda podruhé — 6 bodů

Typická gangsterská čokoláda vypadá jako obdélník o rozměrech $M \times N$ dílků, kde alespoň jeden rozměr je sudý. Dva hráči se střídají v lámání tak, že si ten, kdo je na tahu, vybere nějakou celistvou část čokolády a rozlomí jí na dvě části. Gangster, který odlomí dílek 1×1, prohrál. Vaším úkolem je najít takovou strategii, aby začínající mafián vždy vyhrál.

Příklad: Pro čokoládu o rozměrech 2×3 musí první zloduch lámat na dvě části 1×3, načež druhý zloduch prohrává, protože ať láme, jak láme, vždy získá dílky 1×1, 1×2 a 1×3.

Pozn.: Protože čokolády o rozměrech 1×1 a 1×2 neposkytovaly gangsterům dostatečné mlsavé uspokojení, upustily nelegální čokoládovny od jejich výroby, takže je zanedbejte.

Dveře se zabouchly a my slyšeli jen křupání tabulek čokolád, skřípání zubů a po chvíli . . . „Ty podvodníku!“

„Čože? Já hlaju naplošto a školo češtně.“

A pak třeskly dva výstřely, ozvaly se dvě tupé rány a já jsem velmi odvážně vykoukl dírkou ve dveřích.

„Je po nich, jsme volní!!!“ zajásal jsem.

„A jak se dostaneme ven, Einsteine?“ zpražila mě pohledem.

Faktem je, že na tento způsob jednání jsem byl od žen, zvláště tak krásných jako ona, zvyklý. Nechtěl jsem se ale nechat zahanbit, a tak jsem začal usilovně přemýšlet. Po patnácti minutách mého přemýšlení konečně na něco přišla, zvedla se, chvíli něco šťourala v takové krabičce s dráty na dveřích a najednou se dveře otevřely. Když jsem po letech zjistil, jak je to snadné, tak jsem se divil, že jsem na to nepřišel sám.

19-2-2 Kvalitní hesla — 6 bodů

Původní obsah následující pasáže jsme museli vystřihnout na nátlak rodiny Panoraiků, která popisovaný systém stále používá. A tak vás tato rodina požádala alespoň o pomoc při rozpoznávání hesel, na která je policejní program krátký. Heslo je, jak známo, posloupnost alfanumerických znaků délky N. Díky znalosti algoritmu, který policejní počítač používá, se nám povedlo zjistit, jak „kvalitu“ hesla spočítat – heslo je tím lepší, čím větší jeho část se v něm opakuje.

Vaším úkolem je po zadání hesla najít maximální d a různé indexy i a j takové, aby se *souvislé* podúseky délky d začínající na těchto indexech shodovaly. Podúseky se mohou překrývat. Pokud těchto dvojic s maximálním d existuje několik, tak stačí najít jednu z nich.

Příklad: Pro $N = 13$ a heslo `abrakadabraka` jsou hledané indexy 1 a 8 (`abraka`) a délka je 6. Pro $N = 7$ a heslo `aaaaaaa` jsou hledané indexy 1 a 2 a délka je opět 6.

Bonus: Pokud bude váš program pracovat opravdu rychle, dostanete až 5 bonusových bodů.

Dveře se otevřely a my začali prchat. Teda, já sem začal prchat. Ona stála ve dveřích a rozhlížela se. „Na co čekáš? Mizíme!“

„Ne, nejdřív musíme do jeho kanceláře, ukradneme vše, co se týká jeho práce!“

„Já si nemysl… ženská pitomá, kam zase běžíš?“

Za chvilku jsme dorazili ke dveřím. Sice byly zamčené, ale má nová průvodkyně se ukázala být nejen šarmantní, ale i velmi zručná. Práce se sponkou jí trvala jen několik vteřin.

Když jsme vešli, okamžitě se vrhla k šuplatům a začala se přehrabovat ve stole. Poté stejně sebevědomě zplundrovala trezor (kde jen vzala heslo?). Jen jsem tupě zíral a jediné, na co jsem se zmohl, byla otázka.

„Tak málo papírů? Vždyť je skoro prázdnej. To mu určitě všechno dělaj jeho účetní.“

„Blázníš? K těmhle věcem nikoho nepouští, všechno si dělá sám.“

„Tomu nevěřím, já dřu od rána do večera, popíšu stohy papírů a div nebydlím pod mostem.“

„No jo, ale ty nemáš program, který ti naplánuje činnosti tak, že málo děláš a hodně vyděláš.“

19-2-3 Moneymaker 10 bodů

Vám je už určitě jasné, že takový program máte napsat. Na vstupu dostane váš program číslo N, což je počet úkolů ke zpracování. Zpracování každé úlohy zabere jednotkový čas. Dále pak N řádků, každý se dvěma čísly. První číslo znamená, dokdy je třeba úkol vykonat, a druhé číslo je odměna, kterou za splněný úkol dostaneme. V jednom čase mohu pracovat právě na jednom úkolu. Výstupem programu by pak mělo být takové pořadí úkolů, aby zisk byl maximální. Pokud je takových pořadí více, stačí libovolné z nich.

Příklad: Pro $N = 4$ a záznamy

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

je optimální pořadí 3, 4 a 1.

Pobrali jsme rychle vše, co se dalo, a vyběhli ven z budovy. Po cestě jsme zneškodnili několikery spící stráže, až na jednoho. Ten bohužel zburcoval všechny ostatní a ti zaujali obrannou formaci. To nám útěk značně zkomplikovalo a jedinou naší nadějí bylo to, že jsme věděli, jak taková formace vzniká. Ale jak taková obranná formace vypadá? To už jsme nevěděli. S tím nám budete muset poradit vy.

19-2-4 Optimální formace 11 bodů

Formace je tvořena N střelci. Střelec číslo i má dostřel a_i. Střelci stojí ve vrcholech konvexního N-úhelníku (konvexní N-úhelník je takový, že všechny jeho vnitřní úhly jsou v intervalu $(0°, 180°)$) v takovém pořadí, v jakém jsou zapsáni na vstupu. Cílem je vytvořit takovou formaci, aby každý střelec dostřelil k následujícímu a aby obvod N-úhelníku byl maximální. Program by měl libovolnou jednu takovou formaci nalézt a vypsat souřadnice jednotlivých střelců.

Příklad: Pro $N = 5$ a dostřely střelců $(2.5, 2.5, 3, 5, 2)$ leží jeden z možných N-úhelníků na souřadnicích $[0, 0]$, $[0, 2.5]$, $[-2, 4]$, $[-5, 4]$, $[-2, 0]$. Pro $N = 4$ a dostřely $(10, 2, 3)$ nelze N-úhelník sestrojit.

Naštěstí se nám díky znalosti přesného tvaru formace podařilo najít slabinu v jejich obraně, a tak se nám povedlo uprchnout.

Pozn. KSP: Přesprstovi zjevně nedošlo, že popis formace neurčuje jednoznačně její tvar. Záhadou zůstává to, že právě vaše odpověď byla ta správná.

Celí zadýchaní jsme doběhli do hlubokého lesa.

„Co si teď počneme? Kam půjdeme? On si nás najde všude a příště už takové štěstí mít nebudeme!“ ptala se zděšeným hlasem moje společnice.

„Znám křišťálovou studánku, kde nejhlubší je les …“

„Co to meleš za nesmysly?“

„Já jsem to řekl nahlas? Já jako myslel, že poblíž tý studánky je opuštěná chalupa, kde bychom se mohli aspoň na noc schovat.“

„A jak jí asi najdeme?“

„No jednoduše, prostě najdeme dva stromy v lese, které jsou u sebe nejblíže ze všech, a tam je les zákonitě nejhlubší.“

„No jo, to mě vlastně nenapadlo …“

19-2-5 Hluboký les — 13 bodů

A zatímco si Přesprst vychutnavá svůj malý triumf, napište program, který takové stromy najde. Váš program dostane na vstupu číslo N a dále N řádků s reálnými souřadnicemi jednotlivých stromů v lese. V případě, že je takových dvojic stromů víc, stačí vypsat libovolnou z nich.

Příklad: Pro $N = 4$ a stromy

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

by měl program vypsat: Stromy 2 a 3 jsou si k sobě nejblíže.

Po hodinách prodírání se lesem nás sama Prozřetelnost přivedla před práh chaty. Unaveně jsme padli do postele a tvrdě usnuli …

19-2-6 Prolog 12 bodů

Milí programátoři v Prologu,

jsme rádi, že se vám první díl seriálu o programovacím jazyku Prolog líbil, a přinášíme vám další zajímavosti ze světa logického programování :o)

Termy

Základní jednotkou programovacího jazyka Prolog je *term*. Termy se v Prologu dělí na jednoduché (atomy, čísla, proměnné) a na struktury. Atomy, čísla a proměnné už znáte. *Struktura* je rekurzivní, složený term, tedy součástí struktury mohou být další termy. Příklady struktur jsou:

```
datum(den(3),mesic(10),rok(2006)).
osoba(jmeno(bretislav),prijmeni(rozsejpal)).
```

Ve skutečnosti je strukturou dokonce i klauzule

```
matka(X) :- rodic(X,_), zena(X).
```

`:-` je totiž binární predikát, který má dva argumenty: hlavu a tělo klauzule, a píše se doprostřed, tedy infixově. Klidně byste mohli psát

```
:-(matka(X), (rodic(X,_), zena(X))).
```

Unifikace pořádně a naposled

Po zadání dotazu, například

```
?-je_matka(X).
```

začne Prolog procházet program shora dolů a snaží se najít, „přiřadit“, neboli *unifikovat* zadaný dotaz s hlavou nějaké klauzule v programu. Jinými slovy prostě najít jaksi odpovídající řádek programu. Jak ale přesně funguje unifikace, když jsme teď zjistili, že úplně všechno v Prologu je term a ty můžou být pěkně složité? K naší radosti to funguje přesně tak, jak byste čekali a jak byste to dělali intuitivně:

Jestliže jeden z termů je (nezunifikovaná, volná) proměnná a druhý libovolný term (různý od proměnné, například nějaká struktura, atom nebo číslo), okamžitě se do proměnné dosadí daný term. To jsme viděli v minulém díle.

Jestliže jsou oba termy proměnné, pak je výsledkem jejich ztotožnění.

Příklad:

```
?-A = B.
```

Jsou-li `A` i `B` volné, unifikace uspěje a proměnné se od tohoto okamžiku budou chovat jako jedna. Pokud `A` bude v budoucnu unifikována například s atomem `kleofac`, pak samozřejmě bude i `B = kleofac`.

Jestliže jsou oba termy atomy nebo čísla, pak unifikace uspěje pouze tehdy, pokud jsou oba stejné.

Jestliže jsou oba termy nějaké struktury, pak provedeme unifikaci rekurzivně. Podíváme se na jejich argumenty a pokud jich je stejný počet, zkusíme každý odpovídající si pár argumentů unifikovat. Pokud se to podaří pro každý argument (a samozřejmě pokud se struktury jmenují stejně), jsou struktury shodné.

V žádném jiném případě nejsou termy shodné a nelze je ani unifikovat.

Tím jsme si podrobně vysvětlili mechanismus unifikace. Teď už také chápeme, co přesně dělá binární predikát `=`, totiž že vyvolává unifikaci na své argumenty.

Co myslíte, že by se stalo, pokud byste napsali `A = f(A)`?

Seznamy

V Prologu máme k dispozici datovou strukturu *seznam*. Seznam je posloupnost termů, například čísel, struktur, nebo dalších seznamů.

Prázdný seznam se značí atomem `[]`. Neprázdný seznam se zapíše například takto: `[a,b,c]` nebo `[1,2,3,4]`.

Prolog chápe seznam jako dvě části: *hlavu* a *tělo*. Hlava je první prvek seznamu a tělo celý zbytek seznamu. Tento zbytek chápe Prolog opět jako seznam, tedy v případě seznamu `[a,b,c]` je hlavou prvek `a` a tělem opět seznam `[b,c]`.

K tomu, abychom ze seznamu snadno oddělili hlavu, slouží ještě jiný způsob zápisu seznamu: seznam `[a,b,c]` můžeme napsat jako `[a|[b,c]]`.

Už nás také napadá, jak budeme s prologovskými seznamy pracovat. Vždy si oddělíme hlavu, něco s ní uděláme a potom se pustíme rekurzivně na zbytek seznamu. Hned si to ukážeme na příkladu hledání prvku v seznamu:

```
% prvek(X,Seznam) je X prvkem seznamu Seznam
prvek(X,[X|_]).
prvek(X,[_|Telo]) :- prvek(X,Telo).
```

V prvním řádku se díváme, jestli hledaný prvek není náhodou přímo v hlavě seznamu. Pokud hledaný prvek není v hlavě seznamu, musíme vzít zbytek (tělo) seznamu a pátrat v něm.

Jiný příklad, tentokrát hledáme poslední prvek seznamu:

```
% posl(Seznam,X) vraci posledni prvek seznamu
posl([X],X).
posl([_|Telo],Posl) :- posl(Telo,Posl).
```

První řádek je jasný, poslední prvek jednoprvkového seznamu je onen jediný prvek. Druhý řádek je zajímavější, pokud máme seznam s jedním prvkem, za kterým je ještě nějaký další seznam, zavoláme si rekurzivně predikát `posl` na tělo seznamu s utrženou hlavou. Takto se postupně volají predikáty `posl` na

čím dál kratších seznamech. Proměnná `Posl` je přitom stále volná. Jakmile dojedeme na konec seznamu a máme už jen jednoprvkový seznam, dostaneme se na dno rekurze, uplatní se první řádek programu a v tom okamžiku se nám úspěšně unifikuje proměnná `Posl`.

Nakonec příklad bez komentáře:

```
% vypust(X,Sezn,NovySezn)
% vypusti jeden vyskyt X ze seznamu Sezn a vrati novy seznam NovySezn
vypust(X,[],[]).
vypust(X,[X|T],T).
vypust(X,[Y|T],[Y|L]) :- vypust(X,T,L).
```

Kvíz

⋆ Který zápis seznamu *není* správný?

1. `[ [a,b], c]`
2. `[a, b, [c] ]`
3. `[a, [b,c] ]`
4. `[ [a,b] | c]`
5. `[a,b,c,[]]`

⋆ Kolik prvků má seznam `[a,b,c,[]]`?

1. 2
2. 3
3. 4

⋆ Jaký výsledek dostaneme při `?-prvek(a,Seznam).`?

1. `No.`
2. `Yes.`
3. `[a,a,a,a,a,...]`
4. `[a,a,a,a,a,...] ; No.`
5. `[a|_] ; [_,a|_] ; [_,_,a|_] ; ...`

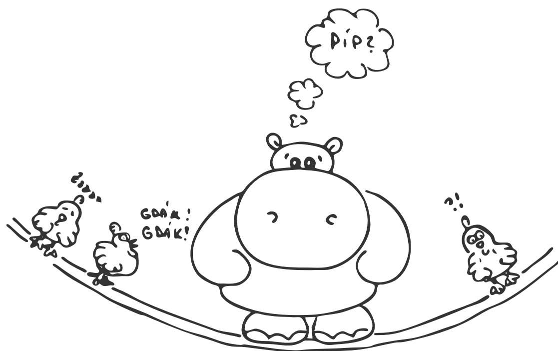
⋆ Jaký bude výsledek dotazu
`p(A,B,q(c,A,B))=p(a,b,q(c,a,d))`?

1. Yes.
2. No.

Soutěžní úložky

1. Příliš těžké slepice (4 body) Slepice sedí na hřadě. Bidýlko se pod nimi prohýbá a každou chvíli hrozí, že praskne. Nejprohnutější a nejzatíženější je bidýlko pod prostřední slepicí. Slepice ale neví, která z nich je uprostřed. Napište slepicím program v Prologu, který dostane seznam slepic tak, jak sedí na bidýlku zleva doprava, a vybere prostřední z nich. Pokud je slepic sudý počet, vybere tu, která sedí více vpravo. Nezapomeňte, že slepice neumí počítat, takže nesmíte používat žádnou aritmetiku. Program by měl být co nejrychlejší. Slepice také neoplývají přílišnou chytrostí, takže byste svůj program měli náležitě okomentovat a popsat :o))

Příklad: Pro vstup `[a,b,c]` je prostřední slepice `b`, pro vstup `[1,2,3,4]` je prostřední slepice `[3]`.



2. Permutující slepice (4 body) Poté, co jste vyřešili problém bidla praskajícího pod slepičími špeky, začalo slepice pálit dobré bydlo a obrátily se na vás s dalším problémem. Slepice se mezi sebou neustále hašteří, ve které části bidla bude která sedět. Už to tak dál nejde, a proto se budou každou noc střídat v pořadí. Napište program, který vypíše všechny možné rozmístění slepic na bidýlku, každé právě jednou. Vstupem programu je seznam slepic a výstupem seznam seznamů obsahující všechny permutace slepic, tj. všechny možnosti, jak mohou slepice sedět, každou právě jednou.

Příklad: Pro vstup `[a,b]` je výstupem tento seznam seznamů: `[ [a,b], [b,a] ]`, pro vstup `[1,2,3]` je výstupem následující seznam permutací: `[ [1,2,3],[1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2] ]`.

3. Palindromické slepice (4 body) Jelikož jste dokázali usmířit rozhádané opeřence, byli jste požádáni o vyřešení posledního problému. Slepice zjistily, že nejbezpečnější rozmístění na bidýlku je takové, že proti sobě symetricky sedí vždy slepice stejné váhy. Například rozmístění vah `[1,2,3,3,2,1]`, `[4]` či `[4,3,5,6,5,3,4]` jsou bezpečná umístění, zatímco `[4,3]`, `[4,3,1]`, `[5,6,7,7,6,6]` nejsou bezpečná umístění. Napište program, který zjistí, zda

slepice sedí na bidýlku bezpečně. Jinými slovy, zjistěte, zda číslo zapsané jako seznam číslic je palindrom, čili slovo, které se čte stejně zepředu i zezadu. Pro sladkou bodovou odměnu se snažte program co *nejvíce* urychlit (nejlépe na lineární :–).

Tímto se s vámi loučíme a těšíme se na setkání v příštím díle. Kokokodák!

Probudil jsem se a zamžoural do denního světla. Zlověstné ticho rušil jen můj dech a vzdálené šumění stromů. Rozhlédl jsem se kolem sebe. Postel vedle mě byla prázdná, ale vyležený důlek naznačoval, že společný útěk s mojí novou známou nebyl jen sen. Ale kam se poděla? Rychle jsem se oblékl a vyběhl z chaty ven. Stála na verandě s hrnkem ranní kávy a zírala do mlhy, která obklopovala chatu jako rozlité mléko.

„Ehm, brý ráno,“ vymáčkl jsem ze sebe a podrbal se v týlu.

„Dobré ráno,“ odvětila a stále hleděla do mlhy. „Máme tu malý problém…“

Upřel jsem pohled přibližně stejným směrem, kterým se dívala ona, a opravdu.

„To jsou ty tvoje křišťálové studánky,“ ušklíbla se jízlivě.

Všude, kam až mé oko v té mlze dohlédlo, se rozprostírala jezírka. A aby toho nebylo málo, stavěli mezi nimi bobři své vodní cesty.

19-3-1 Jezírka — 10 bodů

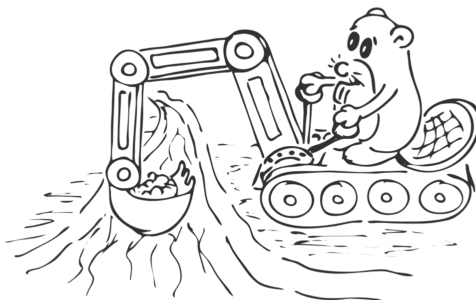
Bobři ve svém teritoriu udržují N jezírek. Mezi jezírky mohou vést obousměrné vodní cesty, které umožňují bobrům rychlé přesouvání z jezírka do jezírka. Každá taková cesta má nějakou kladnou délku (což je přirozené číslo, protože bobři s reálnými čísly pracovat neumějí).

Na počátku žádné cesty nevedou. Každou noc vyrobí kanci, jejichž zemních prací bobři hojně využívají, novou vodní cestu mezi dvěma jezírky. Každé ráno se sejde Bobří rada a ta se rozhodne, zda nově vytvořenou cestu přijmou bobři k udržování, či nikoliv. Aby se mohli bobři rozhodnout, potřebují vědět, zda po přijmutí této cesty budou propojena všechna jezírka, a jaký je součet délek udržovaných cest. Nezapomeňte, že udržování cesty něco stojí a bobři chtějí udržovat co nejméně cest (resp. co nejkratší součet délek těchto cest).

Máte tedy dán počet jezírek N, přičemž jezírka jsou číslována od 1 do N. Každé ráno za vámi bobři přijdou a řeknou vám, jaká cesta byla vytvořena (tzn. čísla jezírek odkud kam vede a její délku). Vy aktualizujete vaše data a *ihned* (tj. před načtením dalšího vstupu, tj. před načtením cesty vytvořené další den) jim oznámíte, zda už jsou všechna jezírka propojena a vypíšete minimální délku udržovaných cest. Cílem je, aby všechna jezírka byla propojena co nejdříve, a v každém kroku byl součet délek udržovaných cest co nejmenší.

Příklad: Buď $N = 4$ a na vstupu cesty z tabulky:

Denní cesty	Okamžitý výstup
$1 \leftrightarrow 2$ délky 5	Jezírka nespojena, délka udrž. cest 5
$2 \leftrightarrow 3$ délky 6	Jezírka nespojena, délka udrž. cest 11
$1 \leftrightarrow 3$ délky 4	Jezírka nespojena, délka udrž. cest 9
$2 \leftrightarrow 3$ délky 8	Jezírka nespojena, délka udrž. cest 9
$2 \leftrightarrow 4$ délky 3	Jezírka spojena, délka udrž. cest 12
$1 \leftrightarrow 4$ délky 1	Jezírka spojena, délka udrž. cest 8



„Měli bychom se vydat co nejdříve na cestu," pronesla má společnice, když dopila svou kávu. „Podívej, támhle hloubí další cestu a támhle ještě dvě!"

„To ano, ale nejdřív bych rád něco snědl. Není tu něco … jedlého?"

Přikývla. „Vzadu jsou celkem slušné zásoby konzerv, sucharů a jiných věcí, které vypadají jedle."

Sebral jsem odvahu a vydal se proslídit spíž. Ať ta chata patřila komukoli, musel to být podivín. Od každé potraviny měl přesně sedm kousků. Sedm konzerv lančmítu, sedm konzerv párků, sedm balíků sucharů … Zajímalo by mě, jestli až se sem dotyčný vrátí, pozná, že mu něco schází.

19-3-2 Inventura ve spíži — 6 bodů

Je dáno pole *Spíž* velikosti N, ve kterém jsou uloženy potraviny ve spíži. Na každé pozici i se nachází nějaká potravina $\mathit{Spíž}[i]$. Potraviny jsou označeny čísly, avšak nemusejí být číslovány souvisle a $\max(\mathit{Spíž}[i])$ může být klidně mnohem větší než N.

Dále dostanete číslo k a máte vypsat, které potraviny se v poli *Spíž* vyskytují právě k-krát.

Příklad: Pro $k = 2$ a potraviny $1234, 654321, 1234, 5$ máte vypsat, že dvakrát se vyskytuje jenom potravina s číslem 1234.

Po krátké ale výživné snídani jsme se vydali na cestu. Podařilo se nám prokličkovat mezi jezírky a před námi se objevila lesní pěšina.

„Bezva. Tahle pěšina vede přímo k hlavní silnici, tam nám stačí zastavit

nějaké auto a máme vyhráno," prohlásila sebejistě a vydala se napřed.

„Jak to můžeš vědět?"

„Znám to tu. Tahle chata totiž patří Angelu Criminallisovi. To je jeden z Carlových právníků."

Chvíli jsme pokračovali mlčky a můj mozek pracoval na plné obrátky. Jak je to možné? Že by se tak dobře znala s nějakým poskokem samotného Carla Assassina? Jedině že by ...

„Předpokládám, že tvůj manžel o téhle chatě neví," pronesl jsem jen tak polohlasem.

„Ne, neví," odpověděla stroze. Otočila se na mě a pozvedla obočí.

„Ale, potom ... jak to!? Vždyť by tě zabil!"

Vzpomněl jsem si na jeden případ, který se stal asi před rokem. Bylo to ve všech novinách. Několik mafiánů zavraždilo svoje manželky v jeden den. A pochopitelně jim to prošlo. Tehdy se říkalo, že je zabili, protože jim byly ženy nevěrné, ale copak může něco takového ospravedlnit vraždu?

19-3-3 Nevěrné ženy — 7 bodů

Mafiáni jsou mocní lidé. Každý mafián ví o všech ostatních mafiánech téměř všechno. Také ví, kterého mafiána podvádí žena a kterého ne. Bohužel to ale žádný mafián neví o své ženě, a tak ji zkrátka věří. V okamžiku, kdy mafián zjistí že ho žena podvádí, tak ji přesně v poledne následujícího dne veřejně zabije (tzn. dozví se to všichni ostatní mafiáni).

Všichni spokojeně žili až do chvíle, kdy se na jednom večírku jeden mafián strašně opil a nechtěně před všemi přítomnými prohlásil: „Alespoň jedna žena tady podvádí svého muže." Devatenáct dní nato byly všechny nevěrné ženy nalezeny krátce po poledni mrtvé. Kolik těchto žen bylo a jak na to podvedení mafiáni přišli?

Abychom předešli častým dotazům, shrneme zde zadání a upřesníme některá fakta:

- každý mafián ví o všech ostatních mafiánech, zda je podvádí ženy,
- žádný mafián neví o své ženě, zda ho podvádí, a nemůže se to nijak přímo dozvědět (nikdo mu to neprozradí – ani nedobrovolně, nikde to nevyčte atp.),
- podvádění je binární – každá žena buď podvádí, nebo ne (jiné stavy nejsou),
- pokud mafián přijde (logickou úvahou) na to, že ho žena podvádí, zabije ji následující den přesně v poledne (tzn. čas je diskrétní, kvantovaný na dny),
- mafiánů je mnoho (pro vaše úvahy můžete předpokládat, že je jich více, než libovolná konečná konstanta),

- všechny vraždy se odehrály najednou v poledne 19. dne (večírek se konal 0. dne) a předtím ani potom se žádné jiné vraždy neudály.

Chceme po vás, abyste zjistili, kolik žen bylo zavražděno. Zároveň popište deduktivní postup mafiánů, jak přišli na to, že jsou jim ženy nevěrné.

„Ale ne,“ usmála se. „S Angelem jsem se seznámila až po tomhle incidentu.“

Mlčky jsme pokračovali dál. Po pár hodinách nám zvuk projíždějících aut a řídnoucí les napověděl, že se blížíme k silnici. Mávnutí rukou a její okouzlující úsměv zastavil první kolemjedoucí auto. Řidič byl lehce nevrlý, když zjistil, že stopujeme dva, ale nakonec se nechal přesvědčit, aby nás odvezl k nejbližšímu motorestu.

Motorest nepatřil zrovna k nejnovějším, nebo snad dokonce nejluxusnějším. Sebejistým krokem vešla dovnitř a kývnutí číšníka na pozdrav dávalo tušit, že ji tu nevidí poprvé. Prošla celým lokálem a sebejistě vkročila do kuchyně. V tichosti jsem ji následoval a čekal, co se bude dít.

„Jak jdou kšefty, Marconi?“ usmála se na kuchaře a zřejmě i majitele v jedné osobě. Kuchař jí úsměv oplatil: „Znáš to, bývalo líp.“

„Mohl bych si zavolat?“ vnořil jsem se do jejich uvítacího rozhovoru.

„A koho chceš prosím tě volat?“ podívala se na mě skoro pobaveně.

„No přece policii.“

Kuchařů úsměv zamrzl a v jeho ruce se s neuvěřitelnou rychlostí objevil velký kuchyňský nůž. Snad by ho i použil, ale ona ho zadržela.

„Policii? Proboha proč? Poldové jsou buď zkorumpovaní nebo si hledí svých problémů, aby si to náhodou u někoho nerozlili.“

„Mám tam známého … jmenuje se detektiv Zamříž a několikrát už mi pomohl. I z horších malérů,“ vypravil jsem ze sebe skoro ublíženě. Na chvíli se zamyslela.

„Když nad tím tak přemýšlím, stejně nemáme co ztratit.“

Kývla na Marconiho a ten mě nevrle zavedl k telefonu. Vytáhl jsem z kapsy papírek s telefonním číslem, ale ouha. Papírek byl celý zmačkaný a některé číslice byly špatně čitelné. Naštěstí jsem si pamatoval, že posloupnost čísel tvořících telefonní číslo je ostře rostoucí.

19-3-4 Nejbližší rostoucí posloupnost — 13 bodů

Máme posloupnost čísel $a_1, a_2, \ldots, a_n$. Chceme najít takovou ostře rostoucí posloupnost $b_1, b_2, \ldots, b_n$, aby byla „co nejpodobnější“ posloupnosti $a_1, \ldots, a_n$. (Slovy ostře rostoucí posloupnost myslíme to, že každý prvek je větší než předchozí.)

Napište tedy program, který dostane na vstupu n a n-prvkovou posloupnost $a_1, \ldots, a_n$. Jeho cílem je najít co nejbližší n-prvkovou posloupnost $b_1, \ldots, b_n$.

Nejbližší myslíme v tom smyslu, aby byl součet vzdáleností odpovídajících členů posloupností co nejmenší. Matematicky zapsáno, chceme nalézt ostře rostoucí posloupnost $b_1, \ldots, b_n$ tak, aby byl součet $\sum_{i=1}^{n} |b_i - a_i|$ co nejmenší. Pokud je nejlepších posloupností $b_1, \ldots, b_n$ víc, stačí najít libovolnou z nich.

Příklad: Pro posloupnost $9, 4, 8, 20, 14, 15, 18$ je nejlepší například $6, 7, 8, 13, 14, 15, 18$. Vzdálenost této posloupnosti od původní je $|6-9|+|7-4|+|8-8|+ +|13-20|+|14-14|+|15-15|+|18-18| = 3+3+0+7+0+0+0 = 13$.

Po několika nesprávných pokusech jsem se konečně dovolal. Zamříž si vyslechl můj problém a slíbil, že se pro nás zajede svým vozem.

Netrvalo dlouho a ocitli jsme se na policejní stanici v Zamřížově kanceláři. Zamříž se pečlivě probíral papíry ukořistěnými v Assassinově trezoru. Napadlo mě, že po všem, co jsem s Assassinovou manželkou prožil, neznám ani její křestní jméno. Není nic jednoduššího, než se zeptat, ale tehdy mě to stálo chvilku přemáhání.

„Isabela,“ odpověděla a obdařila mě jedním z těch úsměvů, po kterém se chlapům podlamují kolena. I mně by se podlomila, kdybych neseděl na židli.

„Nerad vám ruším romantickou chvilku,“ přerušil nastalé ticho Zamříž, „ale tohle nebude tak jednoduché. Mafiánské vztahy jsou příliš komplikované na to, abychom teď mohli libovolného mafiána zavřít.“ Opřel se v křesle a zapálil si doutník. „Kdybychom tak našli slabý článek v jeho organizaci…“

19-3-5 Pevné vztahy — 10 bodů

Abychom zjistili, jak pevné vztahy jsou mezi jednotlivými členy mafiánské organizace, musíme sledovat, jak tyto vztahy vznikly. Když přijde nový mafián X do organizace, naváže vztahy s několika dalšími mafiány $M_1, \ldots, M_k$. Aby vztahy byly pevné, musí v tu chvíli být mezi každými dvěma mafiány M_i, M_j už nějaký vztah.

Na počátku je mafián pouze jeden a ten si začíná budovat organizaci. V každém kroku dostane váš algoritmus nového mafiána a neprázdný seznam již existujících mafiánů, se kterými navazuje vztahy při přijetí do organizace. Algoritmus prověří, zda jsou všichni stávající mafiáni, ke kterým se chce nováček připojit, vzájemně propojeni (mezi každými dvěma jsou nějaké vztahy). Pokud je vše v pořádku, algoritmus začlení nového mafiána do organizace a pokračuje příjímáním dalšího mafiána. V opačném případě ohlásí, že tento nový mafián by byl slabým článkem, a skončí. Algoritmus buď nalezne první slabý článek v mafiánské organizaci a hned skončí, nebo oznámí, že organizace má pevné vztahy.

Příklad: Na začátku je jediný mafián. K mafii se připojují následující mafiáni:

Nový mafián	Navazuje vztahy s mafiány
2	1
3	1,2
4	2,3
5	1,2,4
6	4

Program by měl vypsat, že mafián 5 byl slabým článkem. (Mafiáni 1 a 4 nemají mezi sebou žádný vztah.)

Seděl jsem v Zamřížově kanceláři, poslouchal jeho výklad o vztazích mafiánů a má nálada rychle klesala. Byla policie opravdu tak zbabělá, nebo měl Zamříž pravdu a svržení jednoho mafiána by vyústilo v obrovské nepokoje a vlnu násilností? Moji mysl zaplavovala beznaděj. Co teď, když nám ani policie nepomůže? Nebo pomůže? Tázavě jsem se zahleděl na svého kamaráda...

19-3-6 Prolog — 12 bodů

Milí pokročilí programátoři v Prologu,

vítáme vás u třetího kurzu programovacího jazyka Prolog. Z mnoha došlých řešení a hojných bodových zisků 1. série je vidět, že jste úvodní díl dobře pochopili. Přesto vám doporučujeme přečíst si pozorně povídání k vzorovému řešení 1.série. Pokusili jsem se do něj propašovat několik zajímavých informací, které by se vám při dalším řešení mohly hodit.

Ale teď už se pojďme podívat na další zajímavou pasáž z jazyka Prolog.

Aritmetika

Počítání s čísly je v Prologu, jako všechno ostatní, trošku... jiné. Prolog samozřejmě umí sčítat, odčítat, porovnávat, přiřazovat, atd., ale musíme u toho být opatrní. Na začátek jeden příklad. Chceme sečíst 1 + 2 a přiřadit výsledek do proměnné `X` (tedy, chceme výsledek zunifikovat s proměnnou X). Snad každý by intuitivně napsal něco jako

```
?-X = 1 + 2.
```

To ale nefunguje tak, jak bychom chtěli, totiž nedostaneme kýžený výsledek `3`. Jak jsme si říkali v minulém díle, operátor `=` vyvolává unifikaci na své argumenty, takže místo sčítání se dočkáme toho, že do proměnné `X` se zunifikuje výraz `1+2` tak, jak je. Pokud chceme opravdu vyvolat aritmetickou operaci, musíme tam, kde bychom v matematice použili `=`, použít `is`.

```
?-X is 1 + 2.
      X = 3
```

V tomto případě se skutečně vyhodnotí aritmetický výraz `1 + 2` a do `X` se zunifikuje číslo `3`.

Použití operátoru `is` má ale svá úskalí. Jak si jistě pamatujete, jednou zunifikovanou proměnnou už nesmíme znovu unifikovat za něco jiného. Nová unifikace se provádí jen a pouze, pokud zkoušíme novou větev rekurzivního výpočtu. To platí i pro operátor `is`. Pokud napíšeme

```
?-X is 1 + 2.
```

tak pokud `X` ještě nebyla zunifikovaná, zunifikuje se na `3`. Pokud už ale zunifikovaná byla, například na `4`, místo přiřazení se porovnají hodnoty `3` a `4` a výsledkem bude samozřejmě `No.` Toto má pro nás trošku nepříjemné důsledky – pokud potřebujeme do proměnné uložit novou hodnotu, musíme si vyrobit i novou, dosud volnou proměnnou a do ní si novou hodnotu uložit. Podívejte na tento příklad. Budeme počítat délku seznamu:

```
delka([], 0).                              % delka [] je 0
delka([Hlava|Telo], Delka) :- delka(Telo, Delka2),
       Delka is Delka2 + 1.
```

Délku seznamu počítáme rekurzivně. Nejprve si necháme spočítat délku zbytku seznamu (`Telo`) do proměnné `Delka2` a poté přičteme jedničku za to, že jsme seznamu utrhli hlavu. Musíme ale použít dvě proměnné, `Delka` a `Delka2`.

Druhé omezení na predikát `is` říká, že kdykoliv chceme vyhodnotit nějaký aritmetický výraz na pravé straně, musí být všechny proměnné v něm již unifikované. Příkaz

```
?-X is Y + 1.
```

neuspěje, pokud `Y` není unifikována nějakou konkrétní hodnotou. Pozor tedy na pořadí predikátů v následujícím příkladu:

```
delka([Hlava|Telo], Delka) :- Delka is Delka2 + 1,
       delka(Telo,Delka2).
```

S takovým výpočtem samozřejmě pohoříme, výpočet výrazu `Delka2 + 1` neuspěje, protože proměnná `Delka2` se unifikuje až v dalším predikátu.

Stejným způsobem jako `is` se vyhodnocují i další operátory

`=.=`	aritmetická rovnost
`=\=`	aritmetická nerovnost
`<`	je menší
`>`	je větší
`=<`	je menší nebo rovno
`>=`	je větší nebo rovno

a samozřejmě `+`, `-`, `*`, ...

Ukážeme si ještě jeden zajímavý příklad, tentokrát máme pole a chceme na `n`-tou pozici vložit hodnotu `K` a vrátit výsledek v proměnné `NSezn`.

```
vloz_nty(0,K,Sezn,[K|Sezn]).
vloz_nty(N,K,[H|Telo],[H|NSezn]) :- N2 is N - 1,
```

```
          vloz_nty(N2,K,Telo,NSezn).
```

Postupujeme tak, že si postupně odpočítáváme N. Pokud už je N rovno 0, jsme na správném místě v seznamu a vložíme prvek K do hlavy seznamu. Pokud je N ještě příliš velké, odtrhneme seznamu hlavu, od N odečteme jedničku a pustíme se rekurzivně na zbytek seznamu. Rekurze nám vrátí zbytek seznamu se správně zařazeným prvkem K. Před tento zbytek nesmíme zapomenout předřadit hlavu seznamu, kterou jsme předtím odtrhli.

Stromy

Než začnete číst tuhle kapitolu, doporučujeme přečíst si, co to je binární strom. Pěkný obrázek binárního stromu je na stránce

http://cs.wikipedia.org/wiki/Binární_strom

Povídání o binárních vyhledávacích stromech najdete na stránce

http://ksp.mff.cuni.cz/tasks/18/cook4.html

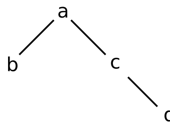
Vyzbrojeni znalostmi se teď můžeme pustit do programování stromů v Prologu.

Nejprve potřebujeme vymyslet pěknou strukturu reprezentující jeden uzel stromu. V uzlu obvykle chceme ukládat nějakou hodnotu a potom levý a pravý podstrom. Použijeme tedy následující strukturu:

t(L,H,R) znamená, že L je levý podstrom, H je hodnota uložená v daném stromě a R pravý podstrom.

Ještě potřebujeme atom pro prázdný strom, ať je to tedy nil.

Následující strom



se tedy zapíše jako

```
t( t(nil, b, nil), a, t(nil, c, t(nil, d, nil)))
```

Abychom nemuseli strom v interpreteru neustále zadávat, uložíme si jej takto:

```
muj_strom(t( t(nil, b, ...))).
```

Pak se na něj můžeme odkazovat dotazy

```
?-muj_strom(T), udelej_neco_se_stromem(T).
```

A jak tedy budeme se stromy pracovat? Jednoduše, protože sám Prolog jako rekurzivní jazyk nám nabízí prostředky pro pohodlný průchod stromu:

```
pruchod(nil,[]).
pruchod(t(L,H,R), [H|Sezn]) :-
        pruchod(L,SeznL),              % projdi levy podstrom
        pruchod(R,SeznR),              % projdi pravy podstrom
        conc(SeznL,SeznR,Sezn).        % spoj seznamy
```

Tento průchod stromem nám dá na výsledku seznam všech uzlů v daném stromu. Všimněte si pořadí průchodu stromem – jakmile přijdeme do nějakého uzlu, nejprve se pustíme do levého podstromu, potom do pravého podstromu, výsledné seznamy uzlů z levého a pravého podstromu spojíme predikátem `conc` do jednoho seznamu, před který přiřadíme hodnotu z aktuálního vrcholu. Výsledný seznam na našem stromě `muj_strom` tedy bude `[a,b,c,d]`.

Poznámka: Predikát `conc` si coby pokročilí programátoři v Prologu dokážete jistě napsat sami.

P.P.P Pěkné Programování v Prologu

Aby váš program měl všech „pět P" a vypadal jako napsaný opravdovým znalcem, přidáváme tentokrát kapitolu o dobrém prologovském stylu:

Poznámky v Prologu vypadají takto:

% Cokoliv za tímto znakem do konce řádky se ignoruje

Před každý nový predikát je třeba napsat, jak se predikát jmenuje, kolik má parametrů, jaké vstupy očekává a co dělá. Dále můžete vsouvat kratičké komentáře k jednotlivým řádkům programu.

Pokud napíšete velmi **dlouhé pravidlo**, můžete jednotlivé predikáty z těla pravidla odsadit na další řádek a mírně je posunout doprava, aby se program zpřehlednil, asi takto:

```
silene_pravidlo(X) :- prvni_pred(X), druhy_pred(X),
          treti_pred(X), mocty_pred(X),
          strasne_mocty_pred(X), a_jeste_jeden_pred(X).
```

Proměnné místo `X`, `Y`, pojmenovávejte radši `Sez`, `Posl`, `Head` a podobně, aby alespoň trošku prozrazovaly, co skrývají.

Vyhněte se **zbytečné unifikaci** pomocí `=`. Kde to jde, unifikujte v parametrech predikátů:

```
jsou_stejne(X,Y) :- X = Y.            % Fuj!
jsou_stejne2(X,X).                    % Krásné
```

Ladění prologovského programu můžete udělat tak, že necháte prologovský interpret sledovat všechna volání zvoleného predikátu, takto:

```
?-spy(muj_nefungujici_predikat).
```

Od tohoto okamžiku bude Prolog vypisovat, s jakými parametry se volá daný predikát, kdy uspěl a kdy se co zkouší znovu...

Kvíz

⋆ Co odpoví Prolog na dotaz: `?- 1 + 2 = 2 + 1.`

1. `Yes.`
2. `No.`
3. Nastane běhová chyba.

⋆ Co odpoví Prolog na dotaz: `?- 2 + 3 =.= 3 + 2.`

1. `Yes.`
2. `No.`
3. Nastane běhová chyba.

⋆ Co odpoví Prolog na dotaz: `?- X < 3.`

1. `Yes.`
2. `No.`
3. Nastane běhová chyba.

Soutěžní úložky

1. Kozel zahradníkem (5 bodů) Kozel sází mrkev a petržel na své zahradě. Má k dispozici `N` záhonků, na jednom záhonu smí být právě jedna plodina. Sadba ale není tak jednoduchá a některé plodiny vyžadují zvláštní způsob rozmístění na záhonku. Tak například dvě petržele nikdy nesmí být zasazeny vedle sebe. Tedy například `mmmpm` je správná výsadba, ale `mmppm` není správná výsadba. Napište v Prologu program, který pro daný počet záhonů `N` určí, jaký je počet všech možných rozesazení mrkve a petržele na záhoncích. Nemusíte vypisovat, jakým způsobem jsou plodiny vysázeny, stačí jen počet možností. Snažte se program urychlit na lineární časovou složitost (vzhledem k počtu záhonů).

Příklad: Pro `N = 2` je počet všech správných vysázení roven `3`. Konkrétně je to `mm`, `mp`, `pm`.

2. Vánoční stromeček (7 bodů) Vánoční nadešel čas… a vy jste se rozhodli nazdobit vánoční stromeček. Máte sadu vánočních ozdob očíslovaných přirozenými čísly ve vzestupném pořadí. Váš vánoční stromeček, protože jste informatici, nevypadá nijak jinak nežli jako binární vyhledávací strom, který má v každém uzlu jednu vánoční ozdobu. A jelikož jste dobří informatici, chcete z ozdob postavit perfektně vyvážený binární vyhledávací strom, tedy takový strom, kde pro každý uzel platí, že počty uzlů v jeho levém a pravém podstromě se liší maximálně o jedna. Prostě a jednoduše nechcete, aby se váš vánoční stromeček nakláněl a nedejbože se třeba ještě někam zřítil.

Vaším úkolem je napsat v Prologu program, který ze vstupního seznamu vyrobí perfektně vyvážený binární strom. Vstupní seznam je seznam vzestupně setříděných přirozených čísel, tedy například `[3, 6, 8, 9, 10]`. Možné řešení úlohy na vstupním seznamu `[3, 6, 8, 9, 10]` je třeba

```
t(t(t(nil,3,nil),6,nil), 8, t(nil,9,t(nil,10,nil))).
```

Špatné řešení by bylo

```
t(nil,3,t(nil,6,t(nil,8,t(nil,9,t(nil,10,nil))))),
```

takový strom samozřejmě není vyvážený.

Pozor, vaším úkolem je napsat celý program. Kdykoli použijete jakýkoli predikát, musíte k němu připsat kód, v žádném případě nestačí napsat „použijeme predikát `predek` z učebního textu“ nebo něco podobného.

Milí čtenáři, vzpomínáte si ještě na mne? Jsem detektiv Přesprst a rozhodl jsem se zvěčnit svůj životní příběh. Bohužel se nenašlo nakladatelství, které by o něj mělo zájem, a tak jsem přijal nabídku organizátorů KSP, kteří se rozhodli můj příběh vydat a ukázat na něm, že i běžná detektivní práce vyžaduje nemalé informatické znalosti. Takže, kde jsem to jen skončil…

Seděl jsem v Zamřížově kanceláři a poslouchal jeho výklad o vztazích mafiánů. Nad mým životem se stahovala mračna. Možná by to bylo ještě smutnější, kdyby nad ním slunce už dávno nezapadlo. Zamříž dokončil svou řeč. Chvíli jsme na sebe mlčky hleděli a nastalé ticho rušilo jen tikání nástěnných hodin.

„Takže tu budeme jen tak sedět a čekat, až si pro nás přijdou?“ ozvala se jízlivě Isabela.

Zamříž se zamyslel. Znal jsem tenhle výraz v jeho tváři. Tohle rozhodně nebude procházka růžovým sadem.

„Tohle rozhodně nebude procházka růžovým sadem,“ promluvil nakonec. „Pokud by se nám podařilo prokázat styky se zahraničím, mohli bychom do celé věci zatáhnout Interpol a požádat o posily, ale jinak nevím.“

„Není nic snazšího,“ prohlásila Isabela. „V těch materiálech z jeho trezoru by měl být i soupis zahraničních plateb.“

19-4-1 Finanční toky — 8 bodů

Máme k dispozici kompletní přehled všech plateb mezi jistými podezřelými organizacemi. Tento přehled tvoří orientovaný graf (viz kuchařka 19-3), ve kterém jsou jednotlivé organizace vrcholy a z i do j vede hrana, právě když organizace i převedla peníze na účet organizace j.

Tento graf máme již uložený jako matici sousednosti. Matice sousednosti M má velikost $N \times N$ (kde N je počet vrcholů grafu) a na pozici M_{ij} je 1, pokud z i do j vede hrana, a 0 v opačném případě. (Na M_{ii} je vždy nula.)

Navrhněte algoritmus, který v takto zadaném grafu nalezne stok. Stok je vrchol, do kterého vedou hrany ze všech ostatních vrcholů a z něho samotného už žádná hrana nevede.

Uvědomte si, že Přesprstovi a Isabele jde o život, a tak by váš algoritmus měl pracovat opravdu rychle. Navíc můžete předpokládat, že matici sousednosti již máte v paměti, a tak nemusíte připočítávat čas potřebný k jejímu načtení.

Příklad: Graf zadaný maticí

$$\begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{matrix}$$

má právě jeden stok – vrchol 2.

„Ano to je ono!“ zajásal Zamříž. „Podívejte, vy dva. Už teď u mě máte metál, ale aby to klaplo, musím vyřídit pár telefonů. Co kdybyste si zatím dali kafe nebo tak něco, a já se o to postarám.“ A se sluchátkem u ucha nám jemně naznačil, abychom prozatím vypadli z jeho kanceláře.

Víčka jsem měl pěkně těžká a kafe bodlo. Chutnalo příšerně, i když v tuhle chvíli mi to bylo úplně jedno. Isabela stála u okna a pozorovala dění na ulici. Z venku se ozval pištivý zvuk pneumatik rychle projíždějícího auta. Přiskočil jsem k ní a trhnutím ji odtáhl od okna.

„Co blázníš,“ stačila ze sebe vypravit, než její slova přehlušila střelba a zvuk tříštícího se skla. Chvíli jsme mlčky hleděli na roztříštěné sklo a rozdýchávali tenhle incident.

„Asi bych ti měla poděkovat,“ prolomila ticho Isabela.

„Řekl bych, že tím jsme vyrovnáni,“ usmál jsem se.

Ze své kanceláře vykoukl Zamříž: „Obvolal jsem několik lidí a dal věci do pohybu. Bohužel náš byrokratický aparát funguje někdy velmi pomalu…“

19-4-2 Byrokratický aparát — 10 bodů

Při schvalování určité záležitosti postupuje byrokratický aparát následujícím způsobem. Každý úředník má na počátku na svém stole nějaký dokument. Úředník si dokument přečte, orazítkuje a pošle ho dalšímu úředníkovi. Práce všech úředníků končí v okamžiku, kdy mají všichni úředníci na stole dokument, který schvalovali jako první (tzn. celý aparát se vrátí do výchozího stavu).

Aby to nebylo tak jednoduché, jsou zavedena speciální pravidla, komu má úředník předat dál orazítkovaný dokument. Každý úředník i má určeného právě jednoho úředníka $f(i)$, kterému své orazítkované dokumenty předává. Aby se dokumenty nehromadily u některých úředníků, zatímco jiní budou bez práce, dostává každý úředník dokumenty od právě jednoho úředníka. Tedy každý úředník jeden dokument pošle dál a jeden od někoho dostane, a tak má stále stejně práce. A protože úřad je úřad, někteří úředníci klidně mohou orazítkovat tentýž dokument vícekrát.

Navrhněte algoritmus, který pro dané přiřazení úředníků f zjistí, kolik minimálně schvalovacích kroků (více než nula) bude potřeba, aby schvalovací proces skončil, tj. aby každý úředník měl na stole dokument, který schvaloval jako první.

Úředníci jsou očíslováni od 1 do N a pravidla předávání dokumentů jsou zadána jako seznam $(1 \to 3, 2 \to 1, \ldots)$. Nezapomeňte, že mohou existovat izolovaní úředníci, kteří dokumenty posílají sami sobě (tzn. pravidla $i \to i$).

Příklad: Pro 5 úředníků a pravidla $(1 \to 5, 2 \to 4, 3 \to 1,\ 4 \to 2, 5 \to 3)$ trvá schvalovací proces 6 kroků.

„Úředního šimla ke cvalu nepřinutíš," pokýval jsem smutně hlavou.

Počkali jsme, než se setmělo a pak jsme se společně se Zamřížem a jeho kolegou vydali opatrně dolů na parkoviště. Vypadalo to, že nás nikdo nesleduje. Nasedli jsme do auta a vydali se na cestu. Isabela se schoulila na zadní sedačce a začala podřimovat. Byl to dlouhý den. Víčka mi těžkla a chtělo se mi spát. Projeli jsme kolem známého motorestu, kde nás dnes dopoledne Zamříž vyzvedl. Ale tahle silnice přece vede …

„Kam to jedeme?" zeptal jsem se.

„Uvidíš," usmál se Zamříž. „Už tam skoro jsme."

Teď to do sebe začínalo zapadat. Přednáška o „pevných vztazích", střelba na Isabelu i směr naší cesty. Ten prašivý skunk Zamříž mě podrazil. Po tolika letech přátelství!

„Proč?! Pro prachy? Nebo je v tom snad něco jiného?!"

„Neber si to osobně," odpověděl Zamříž s klidnou tváří. „Není to jen pro prachy. Je to pro spoustu prachů … "

Dál jsem nečekal. Udeřil jsem ho vší silou a strhl volant. Do rvačky se vložil Zamřížův kolega a nejspíš bych i prohrál, kdyby automobil nesjel z cesty a nenarazil do stromu.

Probral jsem se. Hlava bolela jako střep a ruka také nevypadala dobře. Poplácal jsem Isabelu po tváři, aby se probrala. Zamříž i jeho kolega byli v bezvědomí. Nebyl čas na hrdinství. Buď se probudí a pokusí se nás zabít, nebo přijede policie a přišije nám dvojnásobnou vraždu. Navíc nevím, komu věřit. Sebral jsem Zamřížovi služební zbraň a vytáhl Isabelu z auta.

„Musíme pryč, a honem! Nedaleko odsud je železnice. S trochou štěstí chytíme nákladní vlak."

Běželi jsme, co nám síly stačily a modřiny dovolily. Ani nevím, jak dlouho nám to trvalo, ale nakonec jsme doběhli k trati. A dokonce jsme měli i štěstí. Ozvalo se houkání a za zatáčkou se objevil nákladní vlak…

19-4-3 Naskakování na vlak — 11 bodů

Naskakování na vlak není věc jednoduchá. Přesprst a Isabela jsou navíc celí potlučení, a tak si musí zatraceně dobře rozmyslet, na který vagón naskočí a na který ne. Navíc musí počítat s tím, že se jim nemusí podařit na nějaký vagón naskočit, takže by rádi věděli, jestli se podobný vagón (resp. posloup-

nost vagónů) vyskytuje ve vlaku víckrát. A tady je příležitost pro vás, abyste se zkoumáním vlaku pomohli.

Vlak si představte jako řetězec délky N, kde každé písmeno představuje jeden vagón (např. `U` je uhelný vagón, `P` je poštovní vůz atp.). Dále máte dáno číslo k ($k \leq N$) a máte zjistit, kolik navzájem různých podřetězců délky k se v řetězci (tedy ve vlaku) vyskytuje. Zároveň tyto podřetězce a počty jejich výskytů vypište.

Pozor, vlak už se blíží, takže byste to měli spočítat pekelně rychle. Nebojte se k tomu využít znalostí, které načerpáte z aktuální kuchařky, avšak pokud vymyslíte ještě efektivnější a podlejší postup, bodová odměna vás nemine.

Příklad: Pro řetězec (vlak) `UPDUPDUDUP` a $k = 3$ jsou nalezené podřetězce

```
UPD  2×
PDU  2×
DUP  2×
DUD  1×
UDU  1×
```

Podařilo se nám naskočit na poloprázdný vagón se dřevem. Nebyl příliš pohodlný, ale hned sousední vagón převážel poštovní zásilky. Uvelebili jsme se mezi balíky a pytlem s dopisy a drncání vlaku nás pomalu ukolébalo.

„Hej! Ty … vstávat!“

Probudil jsem se a zamžoural do světla před sebou. Očividně bylo ráno a vlak už nedrncal. Přede mnou stála postava oblečená v pošťácké uniformě a mířila na nás revolverem.

„Tak pohyb, vy dva!“ zarámusil pošťák a naznačil revolverem, abychom se zvedli. Odvedl nás do malého skladiště poštovních zásilek, které se krčilo hned vedle kolejí.

„Tady počkáte, než vyložím zásilky. Pak uvidíme, co s vámi uděláme.“

Strčil nás dovnitř, zamkl dveře a odešel.

„To je prostě skvělé. Co teď budeme dělat, hm?“ pronesla skoro vyčítavě Isabela a posadila se na poštovní balík.

„Já osobně bych si dal snídani.“

„Cože? Ty bys sis dal … “ rozkřikla se, ale pak se zarazila. Podívala se na mě a rozesmála se na celé kolo. Je zajímavé, jak některé věci přijdou člověku veselé, když je až po uši v průšvihu.

Vykoukl jsem z okénka. Pošťák právě skládal veliký balík na váhu. Chvíli jsem pozoroval, jak si hraje se závažími, když v tom mě napadla spásná myšlenka.

„Hej, pane pošťáku, nechcete s tím pomoct?“

19-4-4 Váhy — 6+4 bodů

Pošťák zápasí s váhami, protože nemají vhodnou sadu závaží. Navrhněte optimální sadu závaží, která bude postačovat na zvážení libovolného předmětu o celočíselné hmotnosti 1 až m kilogramů s přesností na jeden kilogram. Předmět považujeme za odvážený, když se misky vah ustálí v rovnovážné poloze, a pozor – závaží můžete pokládat na obě misky vah.

Aby vám pošťák věřil (a ocenil vás šesti body), musíte také dokázat, že vámi navržená sada závaží je funkční (tedy že s ní umíte zvážit libovolný přípustný předmět). Pokud uvedete i důkaz, že daná sada je optimální (tzn. neexistuje menší sada, která by také byla funkční), přidá vám pošťák 4 body navrch.

Pokud existuje optimálních sad více, stačí najít jednu libovolnou.

Poznámka: Při vážení 1 kg předmětu potřebujeme skutečně jedno kilogramové závaží. Nestačí vzít např. 2 kg závaží a předměty, které jsou lehčí, prostě prohlásit za jednokilové.

Příklad: Pro zadané $m = 3$ je jedna z možných optimálních sad závaží $\{1, 2\}$. Věci o hmotnosti 1 a 2 kilogramy zvážíme přímo, 3 kg odvážíme tak, že dáme obě závaží na opačnou misku než vážený předmět.

Tato sada je optimální, protože menší sada by měla pouze jedno závaží a snadno nahlédneme, že s jedním závažím umíme určit hmotnost pouze u předmětů, které váží stejně, jako závaží samo.

„To je dobré," poplácal mě pošťák po zádech. „Nechtěl bys pracovat u nás?"

„No, víš...," začal jsem nesměle s podíval se na Isabelu, která seděla na poštovním balíku.

„Nic mi neříkej. Úplně tě chápu," mrknul na mě šibalsky. „A teď odsud zmizte, než přijde šéf."

Vydali jsme se z nádraží do města. K čertu, vždyť jsem ani nevěděl, co je to za město. Ale zůstat tady nemůžeme. Musíme zmizet za hranice. Zběžně jsem si prošacoval kapsy. Jen pár drobáků, navlhlý doutník a zbraň, kterou jsem sebral Zamřížovi.

„Musíme sehnat nějaké peníze a vypadnout ze země," nahodil jsem, aby řeč nestála.

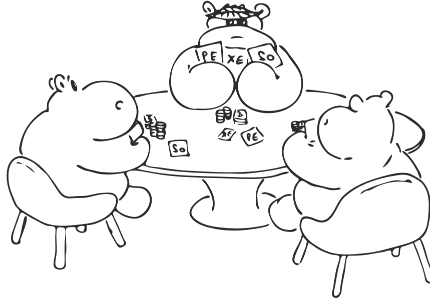
„A co chceš dělat?" podívala se na mě Isabela. „Vykrást banku?"

Rozhlédl jsem se po ulici, ale nikde žádná banka v dohledu. Zato na protější straně ulice jsem zahlédl bar. Nápis nade dveřmi „U Tří Es" hlásal, že se jedná o nefalšované hráčské doupě.

„A co takhle zkusit štěstí..."

19-4-5 Hazardní hra — 10 bodů

Hazardní hráči jsou samozřejmě všemi mastmi mazaní. Nechali Přesprsta vyhrát několik her Pokeru a teď ho chtějí oškubat na jiné, nepříliš známé hře.



Pravidla jsou následující. Bankéř vyhlásí číslo m, což je shodou okolností částka, o kterou se hraje. Hráči se poté snaží vymyslet, jak co nejrychleji tuto částku vysázet na stůl. Přitom ovšem smí používat pouze předem definované tahy:

- přidat jednu minci (všechny mince mají stejnou hodnotu, takže tento tah je vlastně zvýšení sumy o 1)
- odebrat jednu minci (snížení sumy o 1)
- dorovnat na desetinásobek aktuální sumy (tedy vynásobit deseti)
- vydělit aktuální sumu deseti (zaokrouhluj dolů)

Kdo vysází danou částku nejrychleji, vyhrává všechny peníze, které jsou momentálně na stole. Přesprst opravdu potřebuje vyhrát, a tak se neobejde bez vaší pomoci.

Příklad: Pro číslo $m = 192$ jsou nejrychlejší tahy: $+1, +1, *10, -1, *10, +1, +1$.

Klidným krokem jsem vyšel ven a přepočítával vyhrané peníze. Isabela na ně užasle zírala. „Jak jsi to dokázal?“

„Stačí mít trochu štěstí a umět počítat,“ usmál jsem se.

Schoval jsem peníze do kapsy a vykročil směrem k nádraží. Ušli jsme sotva pár kroků, když v tom hned vedle nás zastavilo u chodníku policejní auto…

19-4-6 Prolog — 14 bodů

Milí znalci a přátelé Prologu,

tentokrát vás čeká velmi zajímavá, ale také náročná kapitola. Dozvíte se něco o negaci v Prologu, ale také o myších, logice, filozofii a jiných nebezpečných věcech.

Jemný začátek – vstup a výstup

Při startu programu je aktuální vstup nastaven z klávesnice, aktuální výstup na obrazovku.

Základní jednotkou, kterou můžete načíst nebo vypsat, je term. Predikát `read(T)` načte celý aktuální term (číslo, znak, řetězec, struktura) a unifikuje ho do proměnné `T`. Predikát `write(T)` vypíše term `T`. Pokud je v termu `T` zunifikovaná proměnná, vypíše se místo ní její hodnota.

Pokud chcete číst a vypisovat znaky, použijte predikáty:

`get0(Z)`	přečte znak
`get(Z)`	přečte znak a ignoruje přitom řídící znaky
`put(Z)`	vypíše znak (nevypisuje řídící znaky)
`tab(N)`	vypíše N mezer
`nl`	nový řádek

Příklad:

```
?-write('Zadej zvire: '), read(Zvire), write(Zvire).
Zadej zvire: kachna.
kachna
Zvire = kachna
Yes
```

Příklad: Tento predikát vypíše seznam, každá položka bude na novém řádku:

```
vypis([]).
vypis([H|T]) :- write(H), nl, vypis(T).
```

Pozor: Vstup a výstup je tak trochu neprologovský – jistě víte, že když Prolog splňuje nějaký predikát `p`, postupně splňuje jednotlivé predikáty v jeho těle. Pokud nějaký z nich neuspěje, Prolog „zapomene“ dosud provedené predikáty z těla `p` a hledá na dalších řádcích jiná těla splňovaného predikátu `p`. U vstupů a výstupů ale nejde „zapomenout“ už provedenou hodnotu. Pokud tedy nějaký predikát vypíše něco na obrazovku, zůstane to vypsané i tehdy, když ten predikát nakonec neuspěje.

Filozofické zastavení

Na rozdíl od klasických programovacích jazyků, které přesně popisují algoritmus na řešení dané úlohy, Prolog se snaží vyřešit problém pomocí *matematické logiky*. Každá klauzule (řádek programu) odpovídá nějaké *výrokové formuli*. Příkladem výrokové formule je formule „a & b $\rightarrow$ c“. Prologovský program je tedy *množinou klauzulí*, každá klauzule má svůj *význam*, který vyjadřuje nějaká logická formule. Pokud vezmeme všechny logické formule, které odpovídají všem klauzulím programu, dostaneme *význam programu*. Když pochopíme, že prologovký program je vlastně souborem logických formulí, mezi kterými je „a

zároveň", všimneme si, že v Prologu nezáleží na pořadí jednotlivých klauzulí, ani na pořadí jednotlivých predikátů v jejich tělech.

Bylo by hezké, kdyby fungovala představa čistě logického jazyka, ve kterém nezáleží na pořadí vyhodnocování predikátů, ale bohužel to tak nejde. S takovým jazykem bychom toho mnoho nenaprogramovali, takže se musíme uskromnit.

Opouštíme ideály – predikát řezu

Jistě už jste přemýšleli nad tím, jak se v Prologu udělá negace a proč jsme si ji ještě neukázali. Abychom prologovskou negaci byli schopni vytvořit, vysvětlíme si nejprve *predikát řezu.*

Predikát řezu slouží k vyjádření negace a zrychlení prologovských programů. Značí se vykřičníkem „!". Názornou představu o predikátu řezu si můžeme udělat už z jeho názvu – pokud použijeme řez v nějaké větvi výpočtu, řez zakáže použít další možné větve tohoto výpočtu, tedy zakáže další backtrackování. Nejlepší bude příklad:

```
a(X) :- b(X), !, c(X).
a(X) :- d(X).
```

Prolog zde zkouší splnit první řádek. Pokud se splní predikát `b(X)`, dojdeme k predikátu řezu. Ten okamžitě uspěje, ale přitom zakáže nový vstup do predikátu, ve kterém se nachází, tedy nesmíme už znovu zkoušet splnit predikát `a(X)`. Pokud tedy neuspěje následující predikát `c(X)`, Prolog už díky řezu v prvním řádku nesmí zkoušet další možnost v druhém řádku. Odřezali jsme tedy druhou větev výpočtu. Kdyby v prvním řádku nebyl operátor řezu, Prolog by zkoušel splnit nejprve první řádek, a kdyby se mu to nepovedlo, skočil by hned na druhý tak, jak to známe.

Operátor řezu lze použít dvěma způsoby:

1. Operátor řezu jako tzv. *zelený řez*, ten nemění význam programu, pouze ho urychluje tím, že uřezává neperspektivní větve zbytečné pro výpočet, ale program by fungoval i bez něj.

2. Operátor řezu jako tzv. *červený řez*, kterým změníme průběh vyhodnocování programu.

Příklad zeleného řezu:

V tomto příkladu chceme slít dohromady dvě setříděné posloupnosti tak, aby výsledná posloupnost byla setříděná. Například z posloupností `[1,3,5]` a `[2,4,6]` dostaneme `[1,2,3,4,5,6]`.

```
slij([X|A],[Y|B],[X|C]) :- X=<Y, !, slij(A,[Y|B],C).
slij([X|A],[Y|B],[X,Y|C]) :- X=Y, !, slij(A,B,C).
slij([X|A],[Y|B],[Y|C]) :- Y=<X,slij([X|A],B,C).
slij(A,[],A).
slij([],B,B).
```

Operátor řezu v prvních dvou řádcích se dá chápat jako rada – pokud je například `X=<Y`, nemá cenu zkoušet, jestli je `X=Y` nebo `Y=<X`, protože víme, že to vždy bude nepravda. Operátor řezu zde tedy interpreteru říká, že pokud uspělo `X=<Y`, ostatní větve už by neuspěly.

Příklad červeného řezu:

Červený řez se často používá v predikátech, které při prvním spuštění dají dobrý výsledek, ale při odmítnutí středníkem nebo při znovuzavolání jiným predikátem by mohly začít dávat nesmysly. Příkladem je třeba tento predikát, který má odstranit všechny výskyty prvku `X` v predikátu `Sezn`:

```
odstran(_,[],[]).
odstran(X,[X|Y],Z) :- odstran(X,Y,Z).
odstran(X,[Q|Y],[Q|Z]) :- odstran(X,Y,Z).
```

Prohlédněte si tuto konverzaci s interpreterem Prologu:

```
?-odstran(b,[a,b,c],Vysl).
Vysl = [a, c] ;
Vysl = [a, b, c] ;
No
```

To ale není správné chování. Na odmítnutí středníkem měl Prolog reagovat okamžitým `No.`, protože `[a,b,c]` není přece původní seznam s vymazaným prvkem `b`. Problém je v tom, že po odmítnutí uživatelem zkouší Prolog jiné možnosti, jak splnit predikát `odstran`, a použije třetí variantu `odstran` i v případě, že `X=Q`. Tuto chybu odstraníme tím, že přidáme operátor řezu do druhého řádku, čímž zakážeme případné použití třetího řádku v případě, že `X=Q`:

```
odstran(_,[],[]).
odstran(X,[X|Y],Z) :- !, odstran(X,Y,Z).
odstran(X,[Q|Y],[Q|Z]) :- odstran(X,Y,Z).
```

Jiný příklad: Chceme vložit do seznamu prvek, pokud tam ještě není:

```
vloz(A,Sezn,Sezn) :- prvek(A, Sezn), !.
vloz(A,Sezn,[A|Sezn]).
```

Predikát fail

Predikát `fail` má jedinou funkci – okamžitě si vynutí selhání. Tedy napíšeme-li `predikat :- fail`, tento predikát nikdy neuspěje. Na první pohled se může zdát trošku divné, proč bychom mohli potřebovat takový predikát, ale ve skutečnosti je to jeden z nejužitečnějších predikátů v Prolog, protože ho potřebujeme pro negaci.

Negace

Jak už jsme prozradili, negaci v Prologu tvoříme pomocí řezu. Ukážeme si tedy definici vestavěného predikátu `not(A)`, který uspěje, pokud neuspěje `A`.

```
not(A) :- A, !, fail.
not(A).
```

Predikát `not(A)` se nejprve pokusí splnit cíl `A`. Pokud se `A` splní, zakážeme zkoušet další větve výpočtu a přikážeme predikátu selhat. Pokud se cíl `A` nesplní, Prolog zastaví vyhodnocování tohoto řádku, tudíž se nedostaneme ani k operátoru řezu, takže nezakážeme další větev, která se automaticky splní.

Vidíte, že bez operátoru řezu a bez `fail` bychom toto chování neuměli přikázat.

Příklad: Opět chceme vložit do seznamu prvek, pokud tam ještě není:

```
vloz(A,Sezn,[A|Sezn]) :- not(prvek(A, Sezn)).
vloz(A,Sezn,Sezn) :- prvek(A,Sezn).
```

Kvíz

⋆ Máme následující konstrukci:

```
p(X,Y) :- X > Y, !, fail.
p(X,Y) :- write(X), tab(1).
p(X,Y) :- X1 is X + 1, p(X1,Y).
```

Co se stane, zavoláme-li:

```
?-p(3,6),fail.
```

1. Vypíše se `6 5 4 3`
2. Nekonečná smyčka
3. Vypíše se `3 4 5 6`
4. `Yes.`
5. Nastane běhová chyba.

⋆ Máme konstrukci:

```
p :- !, p.
```

Co udělá dotaz:

```
?-p.
```

1. Nastane běhová chyba
2. `No.`
3. Nekonečná smyčka
4. `Yes.`

⋆ Máme predikát:

```
nacti_jmeno(Jmeno) :- write('Zadej jmeno: '),
      read(Jmeno).
```

Co se stane po následující konverzaci:

```
?-nacti_jmeno(Jmeno).
Zadej jmeno: Kleofac
```

1. Nekonečná smyčka
2. Interpret vypíše `Kleofac`
3. Nastane běhová chyba
4. Interpret vypíše nějakou volnou proměnnou
5. Interpret napíše `No.`

Soutěžní úložky

Důležité upozornění: Všechna řešení musí vracet smysluplná řešení i při opětovném volání, např. odmítání středníkem.

1. Lednice (3 body) Myši opět chystají útok na vaši lednici. Myší útok je samozřejmě potřeba dobře naplánovat, a proto myši vyslaly zvěda, který má za úkol zjistit zásoby v lednici. Lednice je zadána jako seznam potravin, které se mohou opakovat. Myši by potřebovaly program, který dostane na vstupu ledničku jako seznam potravin a jednu konkrétní potravinu, a vypíše, kolikrát se daná potravina v lednici vyskytuje.

Příklad:

Pro lednici `[syr, maslo, syr, cibule, syr, syr]` a potravinu `syr` by měl program odpovědět `4`.

2. Myší spartakiáda (3 body) Vážení a vážené, myši, myšáci a myšáčata! Vítejte na myší spartakiádě! Jako první číslo vystupují bílé a černé myši v čísle „Myší obrazec“! Jak vidíte, na cvičišti se nachází `N` soudků. Na každém soudku smí stát právě jedna myš, buď černá nebo bílá. Myši se nyní vystřídají tak, že na `N` soudcích vytvoří všechny možné barevné kombinace černé a bílé.

Dokážete napsat program, který vypíše na obrazovku všechny možné kombinace myší?

Příklad:

Pro 2 soudky jsou možné obrazce: `bb`, `bč`, `čb`, `čč`.

Pro 3 soudky jsou možné obrazce: `bbb`, `bbč`, `bčb`, `bčč`, `čbb`, `čbč`, `ččb`, `ččč`.

3. Myš v bludišti (5 bodů) Myši se rozhodly skoncovat s nejstrašnější noční myší můrou, kterou je dostat se do myšího bludiště. Poprosily vás o program, který by dokázal najít východ z bludiště. Myší bludiště vypadá tak, že máte pokoje očíslované celými čísly `1,2,3,4,...,N` a mezi některými pokoji vede chodba a mezi některými nevede. Technické detaily nechaly myši na vás. Vymyslete vlastní rozumnou reprezentaci bludiště. Myši vám poskytnou plán bludiště ve vaší reprezentaci, startovní pokoj myši a cílový pokoj myši. Váš program by měl najít libovolnou (ne nutně nejkratší) cestu z bludiště nebo odpovědět, že cesta neexistuje.

Hint: Přečtěte si kuchařku KSP o grafech, kterou můžete najít na adrese *http://ksp.mff.cuni.cz/tasks/19/cook3.html.*

4. Oprava (3 body) Najděte chybu v tomto predikátu, popište, proč nefunguje, a opravte jej. Nejdůležitější je podrobný popis a příklad vstupu, na kterém predikát nefunguje.

```
minimum(X,Y,X) :- X =< Y, !.
minimum(X,Y,Y).
```

V tu chvíli nám došlo, že nemáme co ztratit, a bezhlavě jsme se rozběhli nejbližší uličkou pryč.

„Dáš si ten koblížek s jahodovou polevou nebo s čokoládou?“ zeptal se první policista vystupující z auta.

„Když já nevim... Jé, hele koukej, to je sranda, jak někteří lidé pořád pospíchají,“ funěl při vylézání z auta jeho parťák. „A víš ty co? Koupíme oba nebo radši tři, kdyby se nějaký cestou k autu ztratil,“ zachroptěl blaženě si hladíce bachor.

Po mnoha uběhnutých metrech a úhybných manévrech jsme konečně dotěrné policisty setřásli. Nemohli jsme si ale být jisti, že v pronásledování nepokračují, a tak jsme se rozhodli použít MHD k co nejrychlejšímu úprku z tý pasti, kde jsme se právě ochomejtali, do přístavu. Jenže život není vždycky fér, takže ač jsme byli jistě nejbohatší ve městě, naše zásoba mincí do automatu na jízdenky byla poněkud skrovná. Schválně si zkuste za tisícidolarovku koupit jízdenku s omezenou přestupností na dvě pásma!

19-5-1 Útěk — 10 bodů

Vaším úkolem bude najít co nejrychlejší cestu z počáteční pozice do cílové takovou, že nebude dražší, než obnos v mincích, který máte k dispozici.

Vstupní data budou vypadat následovně: na prvním řádku budou čtyři čísla \$ (čti dolárek), N, S a C, kde \$ značí obnos, který nesmíte překročit, N počet autobusových linek ve městě a S a C jsou počáteční a cílová stanice.

Následuje N řádků, přičemž každý řádek popisuje právě jednu linku. Každá linka a je popsána číslem K, což je počet stanic, kterými linka projíždí, a K trojicemi ve tvaru *číslo zástávky*, *čas odjezdu* (který se shodou okolností rovná i času příjezdu) a *celková cena cesty z první zastávky linky*. Zastávky jsou vypsány v takovém pořadí, v jakém jimi autobus projíždí. Každá linka jede jednou denně a všechny časy odjezdu, které ji popisují, patří do jednoho dne (čili žádná linka nejede „přes půlnoc“). Cena cesty mezi dvěma zastávkami na stejné lince je rozdíl jejich položek *celková cena cesty z první zastávky linky*.

Přesprst a Isabela jsou na začátku ve stanici S brzo, *velmi brzo* ráno (pro potřeby algoritmu v 0:00 hodin). Jejich úkolem je dostat se co nejdříve do sta-

nice C. Najděte tedy takovou cestu pomocí MHD, která stojí nejvýš \$ dolárků a přitom se Přesprst a Isabela dostanou do C co nejdřív. Pokud se do půlnoci do stanice C dostat nejde nebo každá cesta do C stojí víc než \$ dolárků, vypište odpovídající zprávu.

Příklad: Pro vstup

```
50 3 1 5
2 1 8:00 0 5 8:20 60
4 1 8:05 0 2 8:10 25 3 8:18 50 5 8:22 60
4 1 8:06 0 4 8:12 10 3 8:16 20 5 8:26 35
```

je nejlepší cesta následující: linkou 3 na stanici 3, přestoupit na linku 2 a jet až do stanice 5. Cena této cesty je 30 dolárků.

Proč zrovna přístav? To semeniště hříchu? V hlavě se nám (především Isabele) začal rodit plán, jak z téhle pakárny vyváznout se zdravou kůží. Bohužel jediná loď plující na tichomořské ostrovy byla vedena gramotným kapitánem, a tak se o nás doslabikoval v novinách, že prcháme před zákonem.

„Pěkný zákon, zavrčel jsem,“ ale nebylo mi to nic platné. Trval na tom, že chce přesně polovinu našeho těžce nabytého majetku.

19-5-2 Nesnadné dělení — 10 bodů

Najít přesné rozdělení bankovek na polovinu ale není vůbec snadné, zvlášť když mezi nimi jsou i falešné, např. dvěstěčtyřicetikoruna nebo třitísícetřistatřicetřikoruna.

Na vstupu se vyskytuje počet bankovek N a pak N přirozených čísel menších než nějaké přirozené číslo M. Vaším úkolem je rozdělit všechny bankovky na dvě stejně hodnotné části, nebo vypsat, že to není možné. Pokud je takových dělení víc, postačí nám libovolné.

Příklad: Pro $N = 7$ a bankovky $(1, 3, 6, 1, 5, 2, 2)$ je vhodným rozdělením například $(2, 3, 5)$ a $(1, 1, 2, 6)$. Pro $N = 3$ a bankovky $(3333, 240, 240)$ takové rozdělení neexistuje.

Plavba na lodi Submarine *probíhala celkem klidně. Nebyli jsme zřejmě sami, kdo se s kapitánem musel rozdělit o polovinu poctivého výdělku, takže o nás bylo pečováno jako v bavlnce. Ale jinak jsme se na lodi celkem nudili, i když mořská nemoc nám plavbu zajímavě zpestřila.*

Přesto šlo na palubě uniknout šedi všedního dne – šlo o hru Hamtyhamtyhamtyaťmámvícnežtamty. Její pravidla jsou následující:

19-5-3 Hamtyhamtyhamty — 8 bodů

Pokud náhodou neznáte pravidla, což je opovrženíhodné, přinášíme vám jejich popis. Máme posloupnost $2N$ čísel a dva hráče. V každém kroku si hráč vybere jedno číslo na libovolném konci posloupnosti a to odebere. Vyhraje ten, kdo má na konci větší součet. Pokud jsou oba součty stejné, jedná se o remízu.

Vaším úkolem je najít strategii pro prvního hráče takovou, aby vyhrál vždy, když je to možné. V opačném případě musí alespoň remizovat. Asi není třeba zdůrazňovat, že alespoň malý náznak důkazu správnosti vaší strategie je nezbytnou součástí řešení.

Příklad: Pro posloupnost $(10, 100, 3, 1)$ odebere první hráč jedničku, druhý cokoliv, načež první vezme 100 a tím zjevně vítězí.

Dodnes si nedokážu vysvětlit, proč Isabela pokaždé vyhrála. Ona to sice vysvětlovala tím, že od manžela leccos pochytila, ale stejně si myslím, že to bylo pouze začátečnické štěstí. Zajisté chápete, že mě ta hloupá hra brzy omrzela.

A tak když nás i hraní na letadlo na přídi přestalo bavit, zašel jsem z kapitána vymámit alespoň část našich peněz. Neúspěšně ovšem. Zato se mi dostalo pojednání o tom, jak lze absenci kvalitního vybavení nahradit jeho množstvím a, cituji, „důftipem.“

19-5-4 Lodní mrazáky — 10 bodů

O co vlastně šlo? Typický lodní mrazák je vlastně takový zásobník. To znamená, že potraviny je možno přidávat pouze navrch a opět pouze z vrchu odebírat.

Pro kuchaře je to ale docela nepříjemné, protože chtějí vařit vždy z co nejstarších potravin (které jsou úplně naspodu mrazáku). Kuchařům by vyhovovala místo zásobníku fronta, kam by se potraviny přidávaly navrch a odebírat by šly jenom odspodu.

Máte k dispozici několik mrazáků, čili několik zásobníků, a chcete simulovat frontu, tj. musíte umět vyřizovat požadavky *vlož potravinu* (vloží „navrch“) a *vydej potravinu, která byla vložena nejdříve* (vydá „odspodu“). Smíte používat pouze několik mrazáků, čili vkládat a vyndávat z nich potraviny. Pokud nějakou potravinu vyndáte, musíte ji do nějakého mrazáku vrátit dřív, než vyndáte libovolnou jinou potravinu.

Kapitán po vás chce, aby tato simulace fronty byla co nejrychlejší. Požaduje, aby vyřízení N frontových požadavků zabralo nejvýš $O(N)$ pomocných zásobníkových operací (vyndání a vložení potraviny z mrazáku do mrazáku).

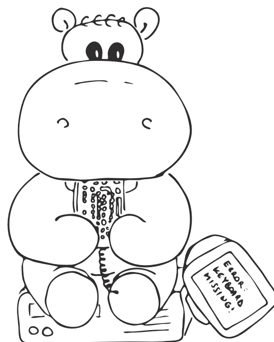
Poznámka: Mrazáků můžete použít jenom konstantně mnoho, řekněme nejvýše 5. Snažte se jich ale použít co nejméně.

Hintík: Všimněte si vychytralé kapitánovy formulace. Nechce, aby jedna frontová operace použila konstantně mnoho pomocných zásobníkových operací. Nejspíš se může stát, že několik frontových operací použije velké množství pomocných zásobníkových operací. Nicméně ostatní frontové operace pak musí být rychlé. Celkově nesmí být na N frontových požadavků použito více než $O(N)$ pomocných zásobníkových operací.

Po zbytek plavby se už nic význačného nestalo, jenom jsem po kapitánově přednášce značně omezil množství zkonzumované potravy, což se pozitivně projevilo na mé postavě.

Nakonec jsme přistáli u malebného ostrůvku Greenland. Naštěstí i po kapitánově zásahu nám zbylo dost peněz na vybudování šestnáctipokojové luxusní chatrné chýše v retro havaj stylu s výhledem na moře – kam taky jinam, že? A když teď tak po letech koukám na Isabelu a našich deset dětí, myslím, že ta její návštěva u mě v kanceláři byla to nejlepší, co mě mohlo potkat. A ji vlastně taky…

19-5-5 Praktická úložka – Počet inverzí — 10 bodů



Po dlouhém přemýšlení a debatování jsme se rozhodli, že pro vás připravíme malé překvapení. KSP byl vždy čistě teoretickým seminářem, ve kterém šlo především o algoritmicky správné řešení a na implementaci nebyl kladen velký důraz. V tomto trendu chceme samozřejmě pokračovat, avšak s malou výjimkou. Jako pátou úložku této série jsme pro vás připravili praktický test, který prověří vaši programátorskou zručnost.

V praktické úložce nemusíte vaše řešení vůbec popisovat, nebo jakkoli komentovat, ale zato musíte odladit funkční program, který danou úlohu vyřeší. Odevzdávat budete pouze zdrojový kód, a to přes speciální webovou aplikaci *CodEx* (The Code Examiner), která sídlí na adrese

https://codex2.ms.mff.cuni.cz/ksp/

Přihlašovací jméno a heslo do CodExu je totožné s přihlašovacím jménem a heslem do webového submitovátka, které již znáte řadu let. Pokud nemáte dosud zřízený účet na submitovátko, musíte se nejprve zaregistrovat.

Opravování probíhá tak, že CodEx převezme váš zdrojový kód, zkompiluje ho a následně jej pustí na sadu testovacích dat. Každý test má navíc nastaven časový a paměťový limit, který vaše řešení nesmí překročit. Za úspěšně vyhodnocené testy dostanete body a celkový součet bodů ze všech testů tvoří hodnocení vašeho řešení.

Vzhledem k tomu, že je velice obtížné napsat perfektní řešení na první pokus, budete mít pokusů více (detaily se dozvíte přímo v CodExu). Do výsledku se vám bude počítat nejlepší odevzdané řešení.

Aby byla úloha pokud možno co nejspravedlivější pro všechny, můžete odevzdávat pouze zdrojové kódy napsané v jazycích `Pascal` a `C`. Příznivcům ostatních jazyků se omlouváme, ale není v našich silách rozumně testovat i jiné jazyky (zvláště pak některé exotické, nebo interpretované).

Další podrobnosti a technické detaily můžete nalézt přímo v CodExu. Pokud byste měli jakékoli dotazy, technické potíže apod., obraťte se na známou adresu KSP, případně na diskusní fórum. Rovněž bychom velice rádi znali váš názor na zavedení praktické úložky do KSP, zda se vám (ne)líbí, návrhy na vylepšení atd. V případě, že se letos osvědčí, zařadili bychom ji v příštím ročníku „naostro“. Přejeme hodně zábavy při řešení...

Zadání:

Je dána posloupnost celých čísel $P_1, P_2, \ldots, P_N$. Čísla P_i a P_j jsou v *inverzi*, pokud $i < j$ a zároveň $P_i > P_j$. Inverze je tedy porucha ve vzestupném uspořádání posloupnosti. Vaším úkolem je zjistit, kolik inverzí posloupnost obsahuje.

Vstup je uložen v textovém souboru `cisla.in`, kde na prvním řádku je číslo N a na druhém řádku následuje N celých čísel v desítkovém zápisu oddělených mezerami. Počet inverzí vypište na standardní výstup.

Čísla mohou být i záporná a vejdou se do 32-bitového integeru (`int` v Céčku, `LongInt` v Pascalu). Čísel v posloupnosti je maximálně 100000 a počet inverzí se vejde do 32-bitového integeru. Vstupní soubor se vejde do operační paměti (i několikrát).

Příklad: Pro soubor `cisla.in`:

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

vypište na standardní výstup 8.

19-5-6 Prolog — 13 bodů

Milí ProloGuru,

vítejte u pátého, posledního dílu seriálu o Prologu.

Zjednodušíme si život – seznamy výsledků

Protože už máte za sebou své programátorské začátky, dozvíte se za odměnu o třech užitečných predikátech, které za vás udělají spoustu práce: `findall`, `bagof` a `setof`. Jejich použití si vysvětlíme na příkladu.

Máme fakta o chovatelích a jejich zvířatech:

```
chova(petr,leguan).
chova(pavel,sklipkan).
```

```
chova(petr,krajta).
chova(petr,pirana).
chova(jan,sklipkan).
```

Predikát `findall` můžeme použít k tomu, abychom zjistili o zadaném člověku, jaká zvířata chová:

```
?-findall(Zvire,chova(petr,Zvire),Zvirata)
      Zvirata = [leguan, krajta, pirana]
```

Predikát `findall(Term, Cil, Seznam)` totiž vytvoří seznam `Seznam` z takových termů `Term`, že splňují daný `Cil`. Tedy v našem případě jsme zadali jako `Term` proměnnou `Zvire` a jako cíl predikáty, ve kterých vystupuje `petr` jako chovatel a `Zvire` jako chované zvíře. Predikát `findall` našel tudíž všechny proměnné `Zvire` takové, které splňují cíl `chova(petr,Zvire)`, tedy zvířata chovaná `petr`em a vytvořil z nich seznam `Zvirata`.

Predikát `findall` tedy vytvoří seznam všech dostupných řešení. Kdybychom se tedy zeptali následovně:

```
?-findall(Zvire,chova(Chovatel,Zvire),Zvirata).
```

dostaneme tento seznam všech dostupných řešení: `[leguan, sklipkan, krajta, pirana, sklipkan]`, což nemusí být přesně to, co bychom si představovali. Můžeme proto použít predikát `bagof`, který dává výsledky postupně pro každého chovatele:

```
?-bagof(Zvire, chova(Chovatel,Zvire),Zvirata).
      Chovatel = petr
      Zvirata = [leguan, krajta, pirana] ;
      Chovatel = pavel
      Zvirata = [sklipkan] ;
      Chovatel = jan
      Zvirata = [sklipkan] ;
      No
```

Jinak pokud bychom zavolali `bagof` jako v přecházejícím případě, tj. `bagof(Zvire,chova(petr,Zvire),Zvirata)`, dostali bychom stejný výsledek jako s `findall`.

Predikát `setof` se chová jako `bagof`, ale ve výsledném seznamu se každý term smí vyskytovat pouze jednou. K tomu, aby `setof` dokázal vyloučit opakující se hodnoty, výsledný seznam se třídí, tím se opakující se hodnoty dostanou k sobě a `setof` nechá jenom jeden výskyt. Na výstupu je pak seznam také setříděný.

Databáze v Prologu

Na prologovský program se můžeme podívat také z jiného úhlu – program v Prologu pro nás může být jakási „databáze“ faktů. Dosud jsme vám ale

(z pedagogických důvodů) utajili, že do prologovské databáze lze přidávat nové klauzule za běhu, případně je zase odebírat. Můžeme si tedy ukládat i něco jako „globální proměnné“.

Novou klauzuli (resp. fakt) můžeme do databáze uložit predikátem `assert(K)`. Přitom platí, že proměnná `K` už musí být unifikovaná s nějakou klauzulí – pozor, zdůrazňujeme *unifikovaná s klauzulí*, tedy například s `zena(petronela)`. Pokud je proměnná `K` již unifikovaná, predikát `assert(K)` okamžitě uspěje a `K` se uloží na konec databáze.

Příklad:

Máme následující program v Prologu:

```
jidlo(spagety).
jidlo(vdolecky).
```

Je samozřejmé, ze kdybychom se zeptali

```
?-jidlo(zelenina).
```

odpoví nám Prolog `No`.

Představme si, že bychom se ale v průběhu programu nějak dozvěděli, že zelenina je opravdu jídlo, například by nám to někdo zadal z klávesnice. Bez databáze bychom byli v zapeklité situaci, protože jak víme, Prolog by okamžitě po opuštění daného predikátu na zeleninu zapomněl. My to můžeme vyřešit v průběhu programu jednoduše:

```
read(Novejidlo), assert(jidlo(Novejidlo)).
```

Od tohoto okamžiku máme v databázi Prologu další druh jídla, který nám zadal uživatel (ať je to třeba `zelenina`):

```
?-jidlo(zelenina).
        Yes
```

Zatím jsme si ukázali použití predikátu `assert` pro ukládání faktů. Na začátku jsme ale slíbili, že si pomocí něj můžeme uložit klauzuli, tedy i pravidlo. To skutečně jde, ale musíme při tom být opatrní. Pravidlo je při ukládání potřeba opatřit závorkami:

```
?-assert((pravidlo:-cil(X)))
```

Dále si povíme, jak odstranit klauzuli z databáze. Slouží k tomu predikát `retract(K)`. Predikát `retract(K)` odstraní z databáze první klauzuli, která je celá unifikovatelná s `K`. Když voláme predikát `retract(K)`, musí `K` obsahovat nějaký term, který má aspoň hlavu (název termu), aby Prolog věděl, co má smazat:

```
?-retract(jidlo(zelenina))
```

Od této chvíle už zelenina není jídlo.

Dalším užitečným predikátem pro práci s databází je predikát `retractall(H)`. Tento predikát smaže z databáze úplně všechny klauzule, jejichž hlava se unifikuje s termem `H`, tedy například:

```
?-retractall(jidlo(_)).
```

smaže veškeré jídlo z databáze.

Nakonec jedno velmi důležité upozornění. Aby vaše hrátky s databází mohly fungovat, musíte na začátek programu Prologu říct, které predikáty budete za běhu ukládat a odebírat. To se udělá takto:

```
:- dynamic(jidlo/1).                    % Nezapomeňte na „:-“
```

Tímto říkáte Prologu, že jednoparametrový (proto /1) predikát jídlo bude dynamický.

Poznámka: Zápis `predikát/počet_argumentů` je obecný a používá se, pokud chceme označit, popsat, identifikovat nějaký predikát. Tento popis je totiž jednoznačný, žádné dva predikáty nemůžou mít zároveň stejné jméno a stejný počet argumentů.

Repeat-until v Prologu

V minulém dílu jsme se seznámili s predikátem řezu. Ukážeme si, jak s jeho pomocí naprogramovat cyklus repeat-until. V Prologu existuje nulární predikát `repeat`, který vždy okamžitě uspěje, a to i při návratu. Cyklus repeat-until tedy můžeme napsat pomocí predikátu `repeat` a predikátu řezu `!` takto:

```
repeat, Cil1, Cil2, ..., PosledniCil, Podminka, ! .
```

Jistě vidíte, co se v tomto cyklu děje. Predikát `repeat` uspěje hned napoprvé, poté se splňují cíle, nakonec `Podminka`. Pokud podmínka neuspěje, predikát `repeat` zaručí nové zkoušení cyklu, pokud `Podminka` uspěje, následující predikát řezu zakáže další cyklení.

Rozdílové seznamy

Naším cílem bude vymyslet reprezentaci seznamů tak, abychom v ní mohli provést zřetězení seznamů rychleji než lineárně, tedy jinak, než že bychom brali prvky z prvního seznamu jeden po druhém a postupně je připojovali k druhému seznamu. Ukážeme si strukturu, která je příkladem *neúplně definovaných datových struktur*. Neúplně definovaná datová struktura je struktura, která obsahuje nějakou volnou proměnnou, která ještě nebyla s ničím unifikována.

Vezmeme si seznam `[a,b,c]`. Takový seznam má v původní reprezentaci prvky `a`, `b`, `c`. Mohli bychom ho také zapsat takto: `[a,b,c|[]]`. My teď uděláme jen a pouze to, že místo závěrečného prázdného seznamu `[]` dáme nějakou volnou, neunifikovanou proměnnou, třeba `L`. Abychom s ní ale mohli pracovat, aniž bychom celý seznam museli projít a zjistit, jakou volnou proměnnou seznam končí, „uložíme“ si ji ještě „bokem“ – formálně to zapíšeme

jako [a,b,c|L]-L (zvídaví čtenáři nechť vědí, že jsme místo mínus mohli použít i jiné operátory; operátor zde nemá svůj normální význam, slouží jenom k uložení volné proměnné „vedle" seznamu). Tento seznam je sice trošku divný, ale pořád ještě obsahuje prvky a, b, c. A k čemu je to dobré?

Vezměme si takové seznamy dva (například [a,b,c|X]-X a [d,e|Y]-Y) a budeme je chtít zřetězit. Čeho chceme dosáhnout? Chceme za X dosadit druhý seznam [d,e|Y] a získat tak seznam [a,b,c,d,e|Y]-Y. Stačí tedy napsat pravidlo:

```
zretez(A-B, B-C, A-C).
```

a při volání do něj dosadíme

```
zretez([a,b,c|X]-X, [d,e|Y]-Y, Vysl).
```

Pojďme detailně sledovat, co se kam dosadí. Pojedeme po řádku zleva doprava. Nejprve se unifikuje A za [a,b,c|X], pak se za B dosadí odečtené X, které je ještě pořád volné! Tedy B = X a pořád v nich ještě nic není. Ale teď přijde kouzlo. V dalším argumentu se B unifikuje s [d,e|Y], ale protože B = X, tak v A máme místo dříve volného X najednou [a,b,c,d,e|Y]. Poslední unifikace jenom dává výsledný seznam do pořádku, tedy říká: Vysl je A-Y, tedy Vysl je [a,b,c,d,e,f|Y]-Y.

Jelikož unifikace proměnná-term (čili dosazení do proměnné) je konstantní, získali jsme způsob, jak napojit dva seznamy za sebe v konstantním čase.

Převod z „normální" reprezentace na rozdílový seznam je ale lineární, takže pokud chceme využívat tento rychlý způsob zřetězení, musíme si seznamy převést do rozdílové reprezentace na začátku, a pak s nimi počítat jako s rozdílovými po celou dobu.

Nakonec si ukážeme, jak na sebe obě reprezentace převést:

```
%preved(NormSezn, RozdSezn)
preved([],X-X) :- var(X).                % var(X) uspěje, je-li X
                                         % volná neunif. prom.
preved([H|Puvodni], [H|Nove]-Prom) :- var(Prom),
    prevod(Puvodni, Nove-Prom).
```

Co tento prográmek dělá: Postupným odtrháváním hlavy dojede až na konec seznamu, kde vytvoří nový prázdný rozdílový seznam X-X pomocí predikátu var(X). Predikát var(X) uspěje, pokud X je volná, neunifikovaná proměnná, my ho zde naopak používáme na vytvoření nové volné, neunifikované proměnné. Potom se vracíme z rekurze zpátky a odtrhané hlavy seznamu předřazujeme před nový rozdílový seznam.

Závěr a rozloučení

Tímto se s vámi loučíme a doufáme, že se vám programovací jazyk Prolog líbil.

Kvíz

⋆ Máme tento program: `zvire(sklipkan).`
Co se stane, zavoláme-li: `?-assert(pirana).`

1. Syntaktická chyba.
2. Dojde k běhové chybě.
3. Zacyklí se.
4. Přidá zvíře piraňa.
5. Vloží se nulární predikát `pirana.`
6. `No.`

⋆ Co bude výsledkem dotazu:

```
?-repeat, assert(p(a)), fail.
```

1. Syntaktická chyba.
2. Dojde k běhové chybě.
3. Vloží jedno `p(a)` do databáze a odpoví `Yes.`
4. Vloží jedno `p(a)` do databáze a odpoví `No.`
5. Nevloží nic a odpoví `No.`

Soutěžní úložky

1. Nejkratší program (2 body): Napište co nejkratší program v Prologu (co do znaků), který pro zadaný (nesetříděný) seznam přirozených čísel a zadané číslo K najde v seznamu nejmenší číslo X takové, že $X > K$, tedy číslo, které je shora nejblíž hodnotě K. Nehodnotí se rychlost, ale délka programu.

Příklad: Pro seznam `[3,8,1,10,4]` a číslo 5 má program vrátit 8.

2. Fronta (4 body): Navrhněte datovou reprezentaci fronty a napište predikáty, které umí zjistit, jestli je fronta prázdná, odebrat prvek ze začátku fronty a přidat prvek na konec fronty. Všechny operace musí být opravdu konstantní, takže žádné triky ve stylu mrazáků :–)

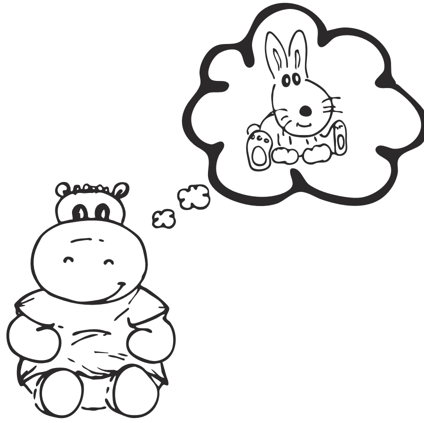
3. Expertní systém (7 bodů): Jako vrchol svého prologovského programátorského umění zkuste naprogramovat jednoduchý expertní systém. Budeme programovat hru "Mysli si zvíře". Uživatel hry si myslí zvíře a program se snaží vhodnými otázkami, na které uživatel odpovídá „ano“ a „ne“, uhádnout o jaké zvíře se jedná, případně konstatovat, že takové zvíře nezná.

Expertní systém se skládá ze dvou částí:

1. Databáze, což je soubor obsahující všechna existující zvířata, všechny možné otázky typu ano/ne („Je to savec?“, „Žere maso?“) a pro každé zvíře též správné odpovědi na všechny otázky. Formát souboru si vymyslete sami. Soubor musí jít načíst do interpretru Prologu pomocí `?-['databaze.pl'].`

2. Program v Prologu, který klade uživateli otázky typu ano/ne a rozhodne, která zvířata (nemusí to být také žádné) odpovídají zadaným odpovědím. Program předpokládá, že databáze je již načtená v interpreteru Prologu, takže se celý expertní systém používá následujícím způsobem:

```
?-['databaze.pl'].
?-['program.pl'].
```



Cílem není napsat perfektní expertní systém (nezkoušíme vás z biologie), ale zúročit všechny vaše znalosti programování v Prologu.

Programátorské kuchařky

16-2-K Kuchařka druhé série – Rozděl a panuj

Rozděl a panuj

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

QuickSort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce" v první sérii 18. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

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

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

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

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
{pivotem se stane poslední prvek úseku}
```

```
begin
 x:=a[r];                                              {hodnota pivota}
 i:=l-1;                             {a[i] bude vždy poslední <= pivotovi}

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

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

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
 if l<r then                                      {máme ještě co dělat?}
 begin
  m:=prer(l,r);                              {přerovnej, m pozice pivota}
  QuickSort(l,m-1);                               {setřiď prvky napravo}
  QuickSort(m+1,r);                                {setřiď prvky nalevo}
 end;
end;
```

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

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

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část

posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $O(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\log_2 N$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\log_2 N$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $O(N \log N)$.

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

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti v prostřední polovině (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $O(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek $\leq (1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = O(\log N)$. Místo 1/4 by dokonce fungovala libovolná jiná konstanta, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se to tak často dělá.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních algoritmech v 16. ročníku). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $O(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

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

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

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

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

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

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

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

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

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme InsertSortem (opět viz třídící kuchařka) a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku InsertSortem. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z + 1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n - z - 1$ prvky.

Řečeno s panem Pascalem:

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

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

```
begin
 pocet:=r-l+1;                        {s kolika prvky pracujeme}

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

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

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

Zbývá dokázat, že tato dvojitá rekurze opravdu má lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián

menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

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

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

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

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

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

Násobení dlouhých čísel

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

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

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

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

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

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

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

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

Program si pro dnešek odpustíme, šetřímeť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

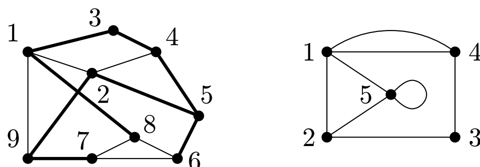
- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také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

16-3-K Kuchařka třetí série – grafy

V dnešním dílu kuchařky si zavedeme základní pojmy z teorie grafů a ukážeme si, jak řešit problém nalezení minimální kostry grafu. Také si popíšeme datovou strukturu *Disjoint-Find-Union* (její název je často zkracován na DFU), kterou šikovně použijeme právě na řešení tohoto problému.

Grafy

Neorientovaný graf je určen množinou vrcholů V a množinou hran, což jsou neuspořádané dvojice vrcholů. Hrana $e = x, y$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a jeho kostra; multigraf

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

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

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

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

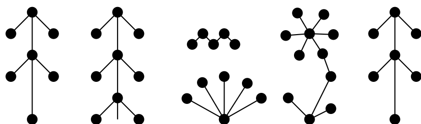
Kružnicí nazýváme cestu délky alespoň 3, ve které oproti definici platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

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

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

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

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



Les, jak ho vidí matematici

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

Ještě se nám bude hodit nahlédnout, že strom s n vrcholy má právě $n-1$ hran: Budeme postupovat matematickou indukcí podle počtu vrcholů stromu. Strom s jedním vrcholem neobsahuje žádnou hranu. Pokud máme strom s $n > 1$ vrcholy, vezměme libovolný jeho list a odeberme ho ze stromu. Tím získáme opět strom (souvislost jsme porušit nemohli a kružnici jsme také nevytvořili)

a jeho počet vrcholů je o 1 menší. Podle indukčního předpokladu má o jednu hranu méně než vrcholů. Nyní list „přilepíme“ zpět, čímž zvýšíme počet vrcholů i hran o 1 a tvrzení stále platí.

A nyní k slibovaným kostrám. Mějme nějaký souvislý graf. Jeho *kostrou* nazveme libovolný podgraf, který obsahuje všechny vrcholy a nejmenší počet hran takový, aby každé dva vrcholy byly spojeny nějakou cestou. Všimněte si, že kostra musí být sama souvislá a navíc neobsahuje žádnou kružnici (jinak bychom mohli libovolnou hranu ležící na kružnici z kostry beze škody odebrat, čímž bychom získali menší kostru, a to nám definice zakazuje.) Čili každá kostra je strom. Na prvním obrázku je kostra levého grafu znázorněna silnými hranami.

Pokud každou hranu grafu ohodnotíme nějakou *vahou,* což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* což je taková, pro kterou je součet vah jejích hran nejmenší možný. Graf může mít více minimálních koster, například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n - 1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální. Pokud si graf představíme jako města spojená silnicemi, problém nalezení minimální kostry můžeme vidět následovně: Chceme určit silnice, které se budou v zimě udržovat sjízdné tak, aby součet délek silnic, které je třeba udržovat, byl co nejmenší možný a zároveň se stále bylo možné přepravit mezi každými dvěma městy.

Algoritmus pro hledání minimální kostry

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: Pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $O(M \log M)$ (použijeme některý z rychlých třídících algoritmů popsaných v jednom z minulých dílů kuchařky) a poté se pokusíme přidat každou z M hran. V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $O(M \log M)$. Celková časová složitost našeho algoritmu je tedy $O(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $O(M)$.

Důkaz správnosti hladového algoritmu

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídícím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e, která je v T_{alg}, ale není v T_{min}. Ze všech takových hran si vyberme tu, která má nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e, vidíme, že sestrojil nějakou částečnou kostru F, která je ještě součástí jak T_{min}, tak T_{alg}.

Přidejme nyní hranu e ke kostře T_{min}. Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jedna hrana e', která není v T_{alg}.

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e: hrana e' neleží v T_{min} na žádném cyklu, takže tím spíš netvoří cyklus v F a kdyby ji algoritmus zpracoval, musel by ji přidat do F, což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e. Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e, musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min}, což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Disjoint-Find-Union

Datová struktura DFU slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva

vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele, trochu nezvykle, od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme se ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce $root(v)$ vrátí kořen stromu, který obsahuje prvek v.

```
var parent:array[1..N] of integer;

procedure init;
var i:integer;
begin
  for i:=1 to N do parent[i]:=0;
end;

function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else root:=root(parent[v]);
end;

function find(v,w:integer):boolean;
begin
  find:=(root(v)=root(w));
end;

procedure union(v,w:integer);
begin
  v:=root(v);
  w:=root(w);
  if v<>w then parent[v]:=w;
end;
```

S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“ a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $O(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression:** Ve funkci $root(v)$ přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:

```
var parent:array[1..N] of integer;
    rank:array[1..N] of integer;

procedure init;
var i:integer;
begin
  for i:=1 to N do
    begin
      parent[i]:=0;
      rank[i]:=0;
    end;
end;

{zmena kvuli path compression}
function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else begin
    parent[v]:=root(parent[v]);
    root:=parent[v];
  end;
end;

{stejna jako minule}
function find(v,w:integer):boolean;
begin
  find:=(root(v)=root(w));
end;

{zmena kvuli union by rank}
procedure union(v,w:integer);
```

```
begin
  v:=root(v);
  w:=root(w);
  if v=w then exit;
  if rank[v]=rank[w] then
    begin
      parent[v]:=w;
      rank[w]:=rank[w]+1;
    end
  else
    if rank[v]<rank[w] then
      parent[v]:=w
    else
      parent[w]:=v;
end;
```

Zaměřme se nyní blíže na metodu *union by rank*. Nejprve učiníme následující pozorování: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků. Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r - 1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r - 1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali. Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank*, je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $O(\log N)$, a tedy operace *find* a *union* stihneme v čase $O(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $O(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $O(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

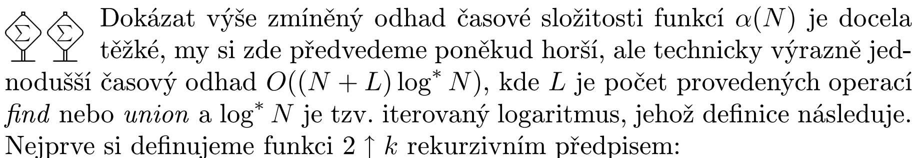
Nejdříve si předveďme tento pojem na jednoduchém příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že N přičtení jedničky

k číslu, které je na počátku nula, zabere čas $O(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $O(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $O(N)$? Použijeme k tomu „penízkovou metodu". Každá operace nás bude stát jeden penízek a pokud jich na N operací použijeme jen $O(N)$, bude tvrzení dokázáno. Každé jedničce, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Přičítání bude probíhat tak, že přičítaná jednička se „podívá" na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu, atd. Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $O(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $O(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $O(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $O(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.



Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $O((N+L)\log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší

přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (k-1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = O(\log^* N)$ skupin. Odhadněme shora počet prvků v k-té skupině:

$$\frac{N}{2^{(2\uparrow(k-1))+1}} + \cdots + \frac{N}{2^{2\uparrow k}} = \frac{N}{2^{2\uparrow(k-1)}} \cdot \left(\sum_{i=1}^{2\uparrow k - 2\uparrow(k-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2 \uparrow k}.$$

Teď můžeme provést časovou analýzu funkce $root(v)$. Čas, který spotřebuje funkce $root(v)$, je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce $root(v)$, a ty, které zahrneme do faktoru $O(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce $root(v)$ započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $O(\log^* n)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow k$ přepojeních je rodič prvku v v $(k+1)$-ní nebo vyšší skupině. Pokud v je prvek v k-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce $root(v)$ nejvýše $(2 \uparrow k)$-krát. Protože k-tá skupina obsahuje nejvýše $N/(2 \uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $O(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce $root(v)$, nejvýše $O(N \log^* N)$. Protože funkce $root(v)$ je volána $2L$-krát, plyne časový odhad $O((N + L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Jednoparametrová Ackermannova funkce $A(k)$ je pak rovna hodnotě $A_k(2)$, čili $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd... Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

16-4-K Kuchařka čtvrté série – hešování

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

- Aplikace typu slovník – máme zadán seznam slov a jejich významů a chceme k zadanému slovu rychle najít jeho význam. Vytvoříme si heš, kde klíče budou slova a hodnoty jim přiřazené budou překlady.
- Rozpoznávání klíčových slov (například v překladačích programovacích jazyků) – klíče budou klíčová slova, hodnoty jim přiřazené v tomto příkladě moc význam nemají, stačí nám vědět, zda dané slovo v heši je.
- V nějaké malé části programu si u objektů, se kterými pracujeme, potřebujeme pamatovat nějakou informaci navíc a nechceme kvůli tomu do objektu přidávat nové datové položky (třeba proto, aby nám zbytečně nezabíraly paměť v ostatních částech programu). Klíčem heše budou příslušné objekty.
- Potřebujeme najít v seznamu objekty, které jsou „stejné" podle nějakého kritéria (například v seznamu osob ty, co se stejně jmenují). Klíčem heše je jméno. Postupně procházíme seznam a pro každou položku zjišťujeme, zda už je v heši uložena nějaká osoba se stejným jménem. Pokud není, aktuální položku přidáme do heše.

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

Samozřejmě používat jako klíč libovolný typ, o kterém nic nevíme (speciálně ani to, co znamená, že dva objekty toho typu jsou stejné), dost dobře nejde. Proto potřebujeme ještě *hešovací funkci* – funkci, která objektu přiřadí nějaké malé přirozené číslo $0 \leq x < K$, kde K je velikost heše (ta by měla odpovídat počtu objektů N, které v ní chceme uchovávat; v praxi bývá rozumné udělat si heš o velikosti zhruba $K = 2N$). Dále popsaný postup funguje pro

libovolnou takovou funkci, nicméně aby také fungoval rychle, je potřeba, aby hešovací funkce byla dobře zvolena. K tomu, co to znamená, si něco řekneme níže, prozatím nám bude stačit představa, že tato funkce by měla rozdělovat klíče zhruba rovnoměrně, tedy že pravděpodobnost, že dvěma klíčům přiřadí stejnou hodnotu, by měla být zhruba $1/K$.

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

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

A operace naprogramujeme zřejmým způsobem:

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

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

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

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

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

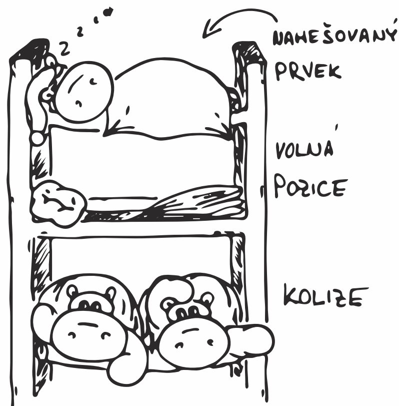
Normálně samozřejmě takové štěstí mít nebudeme a vyskytnou se klíče, jimž hešovací funkce přiřadí stejnou hodnotu (říká se, že nastala *kolize*). Co potom?

Jedno z řešení je založit si pro každou hodnotu hešovací funkce seznam, do kterého si uložíme všechny prvky s touto hodnotou. Funkce pro vkládání

pak bude v případě kolize přidávat do seznamu, vyhledávací funkce si vždy spočítá hodnotu hešovací funkce a projde celý seznam pro tuto hodnotu. Tomu se říká *hešování se separovanými řetězci.*

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

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

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

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

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

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

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



Jaká je časová složitost tohoto postupu? V nejhorším případě bude mít všech N objektů stejnou hodnotu hešovací funkce. Hledání může v nejhorším přeskakovat postupně všechny, čili složitost v nejhorším případě může být až $O(NT + H)$, kde T je čas pro porovnání dvou klíčů a H je čas na spočtení hešovací funkce. Laicky řečeno, pro nalezení jednoho prvku budeme muset projít celý heš (v lineárním čase). Nicméně tohle se nám obvykle nestane – pokud velikost pole bude dost velká (alespoň dvojnásobek prvků heše) a zvolili jsme dobrou hešovací funkci, pak v průměrném případě bude potřeba udělat pouze konstantně mnoho porovnání, tj. časová složitost hledání i přidávání bude jen $O(T+H)$. A budeme-li schopni prvky hešovat i porovnávat v konstantním čase (což například pro čísla není problém), získáme konstantní časovou složitost obou operací.

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

A jakou hešovací funkci tedy použít? To je tak trochu magie a dobré hešovací funkce mají mimo jiné hlubokou souvislost s kryptografií a s generátory pseudonáhodných čísel. Obvykle se dělá to, že se hešovaný objekt rozloží na posloupnost čísel (třeba ASCII kódů písmen v řetězci), tato čísla se nějakou operací „slejí“ dohromady a výsledek se vezme modulo K. Operace na slévání se používají různé, od jednoduchého xoru až třeba po komplikované vzorce typu

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

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

Rozumná funkce pro hešování řetězců je třeba:

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

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

  return r;
}
```

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

A co když nestačí pevná velikost heše? Použijeme „nafukovací“ heš. Na začátku si zvolíme nějakou pevnou velikost, sledujeme počet vložených prvků a když se jich zaplní víc než polovina (nebo třeba třetina; menší číslo znamená větší rychlost [méně kolizí], ale větší paměťové plýtvání), vytvoříme nový heš dvojnásobné velikosti (případně zaokrouhlené na vyšší prvočíslo, pokud to naše hešovací funkce vyžaduje) a starý heš do něj prvek po prvku vložíme.

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

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

Pár poznámek na závěr:

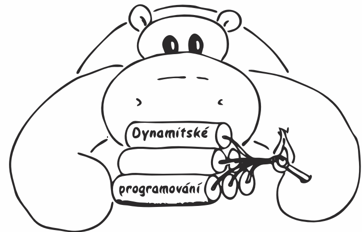
- S hešováním se separovanými řetězci se zachází podobně, nafukování také funguje a navíc je snadno vidět, že po vložení N náhodných prvků bude v každé přihrádce (přihrádky odpovídají hodnotám hešovací funkce) průměrně N/K prvků, čili pro K velké řádově jako

N konstantně mnoho. Pro srůstající řetězce to pravda být nemusí (protože jakmile jednou vznikne dlouhý řetězec, nově vložené prvky mají sklony „nalepovat se“ za něj), ale platí, že bude-li heš naplněna nejvýše na polovinu, průměrná délka kolizního řetízku bude omezená nějakou konstantou nezávislou na počtu prvků a velikosti heše. Důkaz si ovšem raději odpustíme, není úplně snadný.

- Bystrý čtenář si jistě všiml, že v případě prvočíselných velikostí heše jsme v důkazu časové složitosti nafukování trochu podváděli – z heše velikosti N přeci přehešováváme do heše velikosti větší než $2N$. Zachrání nás ale věta z teorie čísel, obvykle zvaná Bertrandův postulát, která říká, že mezi čísly t a $2t$ se vždy nachází alespoň jedno prvočíslo. Takže nový heš bude maximálně 4-krát větší, a tedy počet přehešování na jedno vložení bude nadále omezen konstantou.

16-5-K Kuchařka páté série – rekurze a dynamika

V poslední kuchařce tohoto ročníku se budeme zabývat převážně rekurzí a dynamickým programováním. O čem že je řeč? Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe. Dynamické programování pak bude technika, kterou často půjde z exponenciálně pomalého rekurzivního algoritmu vyrobit pěkný polynomiální. Ale nepředbíhejme, nejdříve se podíváme na jednoduchý příklad rekurze:



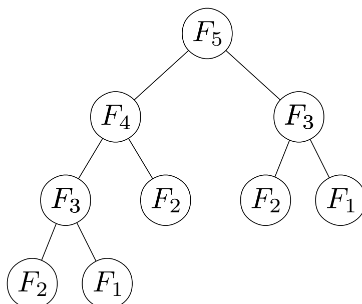
Fibonacciho čísla

Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto: 1 1 2 3 5 8 13 21 34 55 89 ...

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

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

To, jak funkce volá sama sebe, si můžeme snadno nakreslit třeba pro výpočet čísla F_5:



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

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

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

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

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

z čehož plyne:

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

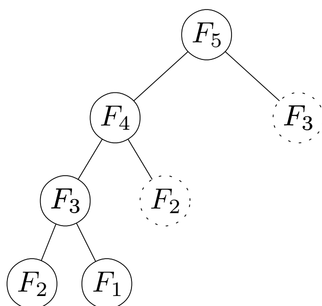
Funkce `Fibonacci` má tedy exponenciální časovou složitost, což není nic vítaného. Ovšem jak jsme už řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka z klobouku s minimem námahy.

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

```
var P: array[1..MaxN] of Integer;
function Fibonacci(n: Integer): Integer;
begin
  if P[n] = 0 then
  begin
```

```
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```

Podívejme se, jak vypadá strom volání nyní:



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

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

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

```
function Fibonacci(n: Integer): Integer;
var
  P: array[1..MaxN] of Integer;
  I: Integer;
begin
  P[1] := 1;
  P[2] := 1;
  for I := 3 to n do
    P[I] := P[I-1] + P[I-2];
  Fibonacci := P[n]
end;
```

Zopakujme si, co jsme postupně udělali: nejprve jsme vymysleli pomalou rekurzivní funkci, tu jsme zrychlili zapamatováváním si mezivýsledků a nakonec

jsme celou rekurzi „obrátili naruby" a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

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

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, a přitom nepřekročil M. My si popíšeme algoritmus, který tento problém řeší v čase $O(MN)$.

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

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

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, které máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti nejtěžší podmnožiny předmětů, která nepřekročí hmotnost M. Nalézt jednu množinu této hmotnosti také není obtížné: v $A[i_0]$ je uloženo číslo jednoho

z předmětů nějaké takové podmnožiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Samotný kód našeho algoritmu lze nalézt níže.

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

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

Na rozmyšlenou: Proč pole A procházíme pozadu a ne popředu?

Nejkratší cesty a Floyd-Warshallův algoritmus

Náš další příklad bude z oblasti grafových algoritmů (o grafech se dočtete například v kuchařce třetí série), ale zkusíme si ho nejdříve říci bez grafů:

Bylo-nebylo-je N měst. Mezi některými dvojicemi měst vedou (obousměrné) silnice, jejichž délky jsou dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratší cest mezi všemi dvojicemi měst. Cestou rozumíme posloupnost měst spojených silnicemi a délkou cesty součet délek silnic, které spojují po sobě následující města. [V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.]

Půjdeme na to následovně: Na začátku si uložíme vzdálenosti mezi městy do dvourozměrného polé D, tj. $D[i][j]$ inicializujeme na vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$,

v praxi tedy nějaké dostatečně velké číslo. V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty.

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

Z popisu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j. Každá z N fází algoritmu vyžaduje čas $O(N^2)$, a tím pádem časová složitost celého algoritmu bude $O(N^3)$. Vystačíme si s pamětí na uložení pole D, tedy $O(N^2)$. Program bude vypadat takto.

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

Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i samotné nejkratší cesty. To lze jednoduše vyřešit například tak, že si navíc budeme udržovat pomocné pole $E[i][j]$ a do něj při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k). Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

Na rozmyšlenou:

- Jak by algoritmus fungoval, kdyby silnice byly jednosměrné?
- Na první pohled nejpřirozenější hodnota, kterou bychom mohli použít pro ∞, je `maxint`. To ovšem nebude fungovat, protože $\infty+\infty$ přeteče. Stačí `maxint div 2`?
- Hodnoty v poli si sice přepisujeme pod rukama, takže by se nám mohly poplést hodnoty z předchozí fáze s těmi z fáze současné. Ale zachrání nás to, že čísla, o která jde, vyjdou v obou fázích stejně. Proč?

- Popis algoritmu vysloveně svádí k „rejpnutí“: Jak víme, že spojením dvou cest, které provádíme, vznikne zase cesta (tj. že se na ní nemohou nějaké vrcholy opakovat)? Inu, to samozřejmě nevíme, ale všimněte si, že kdykoliv by to cesta nebyla, tak si ji nevybereme, protože původní cesta bez vrcholu k bude vždy kratší nebo alespoň stejně dlouhá ... tedy alespoň pokud se v naší zemi nevyskytuje cyklus záporné délky. (Což, pokud bychom chtěli být přesní, musíme přidat do předpokladů našeho algoritmu.)
- Pozor na pořadí cyklů – program vysloveně svádí k tomu, abychom psali cyklus pro `k` jako vnitřní ... jenže pak samozřejmě nebude fungovat.

Vzorová řešení

19-1-1 Zlaté časy — Pavel Čížek

Přesprst by měl radost, jelikož poměrně hodně z vás přišlo na to, že úloha je řešitelná v lineárním čase. Jak tedy na to? Budeme se koukat, kde by začínalo zlaté období, pokud končí nějakým pevně zvoleným záznamem. Když projdeme všechny možné konce, tj. všechny záznamy, tak určitě na zlaté časy musíme jednou narazit.

Na nalezení začátku zlatých časů při pevně zvoleném konci existují tři postupy, které vedou ke třem různě efektivním programům:

1) Triviální postup je zkusit všechny možné začátky a vybrat z nich největší. To vede na kvadratické řešení.

2) Jak si někteří z vás uvědomili, tak součet podposloupnosti záznamů je možno rychle zjistit, pokud známe součet prvních x záznamů (označme ho S_x). Pak již součet posloupnosti mezi záznamy x a y je $S_y - S_{x-1}$ (pokud za S_0 považujeme nulu). Pokud ale máme pevně zvolený konec y, tak začátek zlatých časů je zřejmě takové $x < y$, pro které je S_{x-1} nejmenší.

Bohužel všechna řešení, která se postupnými součty zabývala, se v tomto bodě vrátila k výše uvedenému triviálnímu postupu a postupné součty používala jako pomůcku pro výpočet součtu intervalu záznamů. Nicméně v tomto místě lze postupovat i jinak. Nalezení minima - to se nejlépe udělá pomocí haldy. Budeme si tedy udržovat haldu, ve které budeme mít S_{x-1} pro všechny $x < y$ (y je opět konec).

Budeme-li teď chtít zjistit optimální začátek, zvládneme to v $O(1)$. Při přechodu od y k $y+1$ budeme muset do haldy přidat záznam S_y, to jde v $O(\log N)$. Celkově tedy získáme řešení pracující v čase $O(N \cdot \log N)$.

3) No a nyní slibované řešení pracující v lineárním čase. Pro jeden záznam je řešení triviální. Uvažujme tedy, že známe zlaté časy, které končí záznamem y (označme začátek Z_y)a ptáme se, jak vypadají zlaté časy končící záznamem $y + 1$.

Platí, že stačí buď pokračovat od minulého začátku Z_y, nebo začít až aktuálně zpracovávaným záznamem $y + 1$. Víme, že součet od Z_y do y je největší možný součet končící záznamem y. Pokud je tento součet kladný, nemá cenu vzít kratší součet, který by byl menší. Naopak je-li tento součet záporný, neexistuje žádný kladný součet končící v y a je tedy vždy lepší začít od právě zpracovávaného prvku.

Algoritmus tedy funguje tak, že počítá postupně nejlepší součty končící záznamem y (na začátku nastaví na 0). Poté vždy načte další prvek a je-li

aktuální součet kladný, přičte k němu načtený prvek. Byl-li aktuální součet záporný, přiřadí do něj načtený prvek. Poté vždy zkontroluje, zda není aktuální součet nejlepší možný.

Tento postup nám v každém kroku zabere $O(1)$ času, tedy celková složitost algoritmu je $O(N)$.

```
#include <stdio.h>

#define MaxN 10000

int N;
int Prijmy[MaxN];

int main(void) {
   int i;
   int MaxZacatek,MaxKonec,MaxSoucet;
   int Zacatek,Soucet;

   scanf("%d",&N);
   for (i = 0;i < N;i++) scanf("%d",Prijmy+i);
   MaxZacatek = MaxKonec = Zacatek = 0;
   Soucet = MaxSoucet = Prijmy[0];
   for (i = 1;i < N;i++) {
      if (Soucet < 0) {         // zakladame novou posloupnost
         Zacatek = i;
         Soucet = Prijmy[i];
      } else                    // zustavame u stare ...
         Soucet += Prijmy[i];

      if (Soucet > MaxSoucet) {
         MaxSoucet = Soucet;
         MaxZacatek = Zacatek;
         MaxKonec = i;
      }
   }
   printf("%d %d %d\n",MaxZacatek+1,MaxKonec+1,MaxSoucet);
   return 0;
}
```

19-1-2 Čokoláda — Petr Škoda

Podle počtu došlých řešení vás úloha zřejmě zaujala. Otázka je, zda z důvodů ryze matematických, či spíše vidinou důkladného testování svých domněnek a teorií. Pravdou je, že jste se ke správnému výsledku dobrali téměř všichni. Bohužel, ne všichni jste správný výsledek i dokázali. Jaké je tedy řešení?

Mějme čokoládu o velikosti $m \times n$, například oříškovou. Všimněme si, že na začátku je v jednom velkém lákavém kuse, zatímco po nalámání na kostičky je těchto kousků $m \cdot n$, přestože jsou neméně lákavé. Jistě souhlasíte, že ať rozlomím libovolný kousek čokolády na dva, celkový počet kousků čokolády

se zvětší právě o jedna. Žádný kousek se mi nemůže ztratit, pokud vydržím neujídat, a tak potřebuji právě $m \cdot n - 1$ lámání, abych rozlámal čokoládu na kostičky, ať budu lámat v libovolném pořadí.

Program je opravdu jednoduchý. Stačí načíst vstup a vypsat výsledek násobení. Vše stihneme v konstantním čase i paměti.

19-1-3 Tiskárna — Zbyněk Falt

Sešel se nám velký počet řešení této úlohy a velký počet byl i použitých algoritmů. Nejčastějším postupem bylo nalézt dvě stejné bankovky, odstranit je a pokračovat, dokud nezbude jedna nespárovatelná. Tato řešení ale měla kvadratickou časovou složitost. Kdo se zamyslel nad tím, jak tiskárna funguje (tedy, že na vstupu jsou jednotlivá čísla v počtech 1,2,4,8,16 atd., jinými slovy, že počet navzájem různých čísel je pouze $log_2(N+1)$), odstranil všechny stejné bankovky najednou a zlepšil tím časovou složitost na $O(NlogN)$. Kdo spočítal počet výskytů jednotlivých bankovek a vypsal tu, která se vyskytla právě jednou, dosáhl sice stejné časové složitosti, ale paměťovou touto úpravou zlepšil na logaritmickou. Ten, kdo předchozí postup upravil tak, že si bankovky nepamatoval v poli, nýbrž v trii (neznalí a zvědaví najdou stručný popis trií v řešení úlohy 17-3-2), dosáhl časové složitosti lineární.

My si ale ukážeme řešení, které pracuje rovněž v lineárním čase, ale na rozdíl od trií má paměťovou složitost konstantní. Začněme jednoduchým pozorováním: prvním znakem sériového čísla hledané bankovky je ten, který se mezi všemi prvními znaky všech bankovek jako jediný vyskytuje „liše-krát“. Stejné pozorování můžeme uplatnit i na zbylé znaky, a tak snadno najít hledané číslo. Jediný problém, který nás může trochu potrápit, je ten, že čísla bankovek nejsou stejně dlouhá. Ten ale snadno vyřešíme tak, že kratší čísla dorovnáme nějakým pevným znakem (třeba `chr(0)`) na délku nejdelšího z nich. Teď už jenom stačí vytvořit pole 100×36 (100 je maximální délka bankovek a 36 je počet možných znaků) a jedním průchodem přes všechny bankovky spočítáme počet výskytů jednotlivých znaků na daných pozicích. Nakonec stačí vypsat řešení. Požadované časové i paměťové složitosti jsme už sice dosáhli, ale problém jde řešit ještě jinak a elegantněji.

Na to, abychom našli jediný znak, který se vyskytuje „liše-krát“, totiž nemusíme počítat počet výskytů všech znaků. Kdybychom dokázali jednotlivé znaky jednoduše párovat, tak ten, na kterého nezbyl partner, je náš hledaný. A párovat znaky jednoduše umíme – jediné, co potřebujeme, je operace `XOR`. Tato operace se aplikuje na dvě čísla o stejném počtu bitů. Výsledný i-tý bit dostaneme pomocí i-tých bitů původních čísel pomocí následující tabulky:

XOR	0	1
0	0	1
1	1	0

Z této tabulky plynou následující pravidla:

1) a XOR $0 = a$
2) a XOR $a = 0$
3) (a XOR b) XOR $c = a$ XOR (b XOR c)
4) a XOR $b = b$ XOR a

Z rovnic 1) a 2) vyplývá, že dva stejné znaky se navzájem zruší (spárují) a dál nepřekáží, a z rovnic 3) a 4) plyne, že nám nezáleží na vstupním pořadí znaků.

A to je vlastně celé. Stačí postupně seXORovat všechny znaky a vypsat to, co nám zbylo. Časová složitost je $O(N)$ a paměťová $O(1)$.

Poznámka na závěr: V odhadech složitostí jsem neuvažoval délku sériových čísel bankovek, neboť podle zadání je shora omezená 100 znaky, což je konstanta. Pokud by takové omezení neexistovalo, musela by se do odhadů jejich délka samozřejmě zahrnout.

```
program Tiskarna;

var cislo : array[0..100] of char;
var idx : integer;
var znak : char;

begin
  for idx:=0 to 100 do cislo[idx]:=chr(0);

  idx:=0;
  while not eoln do begin
    read(znak);
    if (znak=' ') then  { Mezera znamená začátek další bankovky a víme, že }
      idx:=0            { a XOR 0 = 0, takže nemusíme "doXORovat" zbytek }
                        { a můžeme jít hned na další }
    else begin
      cislo[idx]:=chr(ord(cislo[idx]) xor ord(znak));
      inc(idx);
    end;
  end;

  write('Poslední vložená bankovka má kód ');
  idx:=0;
  while cislo[idx]<>chr(0) do begin
    write(cislo[idx]);
    inc(idx);
  end;
  writeln('.');
end.
```

19-1-4 Mafiánské rodiny — Martin „Bobřík" Kruliš

Kriminalisté řešící tuto úlohu se až na pár výjimek rozdělili na čtyři tábory: Barviči, Třídiči, Prohledávači a Teoretici. A jaké byly jejich detektivní postupy?

Barviči si řekli, že na to půjdou přímočaře. Na počátku prohlásili, že každý mafián je samostatná rodina a také ho příslušně obarvili. Pak pročítali záznamy, a když zjistili, že záznam spojuje dvě rodiny, jednu z nich přebarvily na barvu té druhé. Barviči nepoužívali žádné jiné triky, a tak jejich snažení zabralo $O(N^2)$ času.

Třídiči si všimli zajímavého faktu, že když se záznamy setřídí vzestupně podle prvního čísla, stačí si pamatovat u každého mafiána, zda jsme ho již viděli nebo ne. Při následném procházení záznamů inkrementujeme počet rodin pokaždé, když narazíme na dvojici, ze které jsme oba mafiány ještě neviděli. Málem zamotali hlavu i samotnému Přesprstovi, protože toto řešení sice funguje pro vzorový vstup, avšak již jednoduchá hříčka: $1-3, 2-4, 3-1, 3-4, 4-2, 4-3$ zamotá Třídičům hlavu.

Prohledávači pochopili velice rychle, že záznamy lze převést na graf a že každá rodina bude v tomto grafu představovat jednu komponentu souvislosti. A tak vyzbrojeni znalostmi z programátorské kuchařky, počali Prohledávači prohledávat. Někteří to vzali zeširoka, jiní do hloubky, ale všichni se zdárně dopracovali k řešení v čase $O(M + N)$, čili $O(N)$.

Nejvypečenější skupinka řešitelů zapojila všechny své teoretické znalosti grafů a vyplodila nejlepší řešení. To si teď ukážeme podrobněji:

Představme si mafiány jako vrcholy grafu a hrany jako vztahy mezi nimi. Nevíme, které hrany představují nadřízenost a které podřízenost, takže necháme graf neorientovaný. Protože má každý mafián právě jednoho nadřízeného a zároveň existuje kmotr, který nadřízeného nemá, musí být tento graf stromem. V případě, že je rodin více, bude graf nesouvislý, a tudíž bude lesem, kde každá rodina představuje samostatný strom.

Strom na N vrcholech má tu krásnou vlastnost, že obsahuje právě $N-1$ hran. Pokud jednu hranu ze stromu odebereme, rozpadne se na právě dva stromy (důkaz si dovolím vynechat - stačí si to trochu promyslet). Když odebereme ze stromu k hran, rozpadne se na $k+1$ stromů (důkaz indukcí z předchozího tvrzení). Takže pokud máme les na N vrcholech, který má dohromady M hran ($M <= N-1$), tak víme, že je složen právě z $N-1-M+1 = N-M$ stromů. Pro zjištění počtu komponent lesa nám tedy stačí znalost počtu vrcholů a počtu hran.

Budeme tiše předpokládat, že mafiáni jsou číslováni souvisle (tzn. pokud máme N mafiánů, tak jsou číslováni od 1 do N). Bohužel ale nevíme, kolik mafiánů je. Při čtení záznamů tedy zjistíme dvě věci: jednak kolik záznamů (tzn. hran) vlastně máme a zároveň si budeme držet největší číslo dosud nalezeného mafiána (což bude zároveň počet mafiánů). Počet hran (záznamů) následně vy-

dělíme dvěma, protože záznamy udržují informace o podřízených i nadřízených a každá hrana se tam vyskytne právě dvakrát - jako (u, v) a (v, u). Nyní máme potřebné údaje a po odečtení počtu záznamů od počtu mafiánů dostaneme, kolik rodin ve městě vlastně je.

Časová složitost algoritmu je lineární, musíme totiž všechny záznamy přečíst a nalézt v nich maximum. Paměťová složitost je konstantní, protože si nepotřebujeme ukládat všechny záznamy, ale vystačíme si pouze s jejich počtem a největším indexem mafiána. Všimněte si, že pokud bychom znali dopředu počet záznamů a počet mafiánů, tak jsme schopni určit počet rodin v konstantním čase $O(1)$, aniž bychom museli záznamy číst.

Tímto vám Přesprst děkuje za příkladnou spolupráci s policií a přeje vám hezký den.

19-1-5 Zámek — Jan Bulánek

Mnozí z vás zřejmě přehlédli, že životní styl Přesprsta mu nejspíš neumožní žít ještě stovky, tisíce, miliony... let, a tak vyprodukovali správná řešení, která však už pro kódy délky 20 poběží asi tři tisíce miliard let a je poměrně pravděpodobné, že tohoto výsledku se nedočkáme ani my. Jak jistě správně tušíte, mluvím o řešeních, která zkoušela všechny možnosti procházením do hloubky, což jistě k výsledku vede vždy. A tak jen maličkou poznámku. Každé volání procedury spotřebovává jisté množství paměti (lokální proměnné, kam se má vrátit po skončení procedury atd.). Z toho plyne, že tyto programy, krom jiného, potřebují paměť úměrnou hloubce rekurze a je třeba ji v odhadech paměťové složitosti uvažovat. Protentokrát jsem za to body nestrhával, protože mi to nepřipadá jako úplně intuitivní věc, ale příště už tak milosrdný (laskavý, hodný... seznam můžete libovolně a vhodně prodloužit :-)) nebudu.

Vzorové řešení se opírá o myšlenky dynamického programování. Pokud si nyní myslíte, že vám nadávám, přečtěte si, prosím, kuchařku v 17. ročníku série první a potom pokračujte. Základní chybou vašeho výše uvedeného řešení bylo, že opravdu generovalo všechna řešení, ale to po nás nikdo nechtěl, protože jenom vypsat všechna možná řešení trvá do skonání světa. Nám ale stačí *počet* možných řešení. To nás přivádí na velmi zajímavou myšlenku. Pokud totiž bude aktuální kombinace končit (třeba) na trojku, tak číslice, které mohu připojit, jistě nejsou ovlivněny tím, co té trojce předcházelo. Toto jednoduché pozorování, které přímo plyne ze zadání, nám umožňuje pohlédnout na věc dynamicky. Vezměme si nějakou kombinaci délky n, která končí na číslici c. Z té lze vytvořit kombinace délky $n + 1$ končící na $c1, c2, \ldots, ck$, kde $c1$ až ck jsou možní následníci c. A díky našemu pozorování víme, že toto můžeme udělat se všemi kombinacemi délky n, které končí na c, protože nám je jedno, co tomu c předcházelo.

Stačí si tedy pro každou cifru pamatovat pouze počet kombinací končících na tuto cifru. A jak provedeme samotný výpočet? Vytvořme si tabulku, kde v n-tém sloupci a c-tém řádku je uložen počet kombinací délky n končící na cifru c. V prvním sloupci jsou samé jedničky (kombinace délky jedna „končící“ na určitou cifru je vždy právě jedna). Když teď budeme chtít spočítat $(n+1)$-ní sloupec, tak jednoduše pro každou cifru c do políček $c1$ až ck přičteme počet kombinací končících na c. V $(n+1)$-ním sloupci tak bude po dokončení výpočtu pro každou cifru uložen počet kombinací, ke kterým může být cifra připojena a to je přesně to, co jsme chtěli. Na konci výpočtu jen počty kombinací končících na jednotlivé cifry sečteme a tím získáme výsledek. Pokud si navíc všimneme, že celou dobu používáme jen poslední sloupeček tabulky, paměťová složitost kvadratická ku počtu cifer (na uložení následníků) nás nemine. Odhady složitostí tedy jsou $O(KL^2)$ a $O(L^2)$, kde L je počet cifer (byť ze zadání plynulo, že je to konstanta. Moje chyba.) a K je délka kombinace.

```
#include <stdio.h>

int L,K;
int **naslednici;
int *poc_nasl;

void get_data() {                        //nezajimavé načítání dat
  printf("Zadej počet cifer a délku.\n");
  scanf("%d%d",&L,&K);
  naslednici = malloc(L * sizeof(int*));
  poc_nasl = malloc(L * sizeof(int));
  for(int i=0;i<L;++i) {
    printf("Pocet nasledníku %d\n",i+1);
    scanf("%d",poc_nasl+i);
    naslednici[i] = malloc(poc_nasl[i] * sizeof(int));
    for (int j=0;j<poc_nasl[i];++j) {
      scanf("%d",&naslednici[i][j]);
      naslednici[i][j]--;
    }
  }
}

int main(int argc, char **argv) {
  int i,j,*pocty,*pocty_new,*swap;

  get_data();
  pocty = malloc(L * sizeof(int));
  pocty_new = malloc(L * sizeof(int));

  for (i=0;i<L;++i) pocty[i]=1;
  for (i=0;i<K;++i) {
    for (j=0;j<L;++j) pocty_new[j]=0;
```

```
    for (int akt_cif=0;akt_cif<L;++akt_cif)
      for (j=0;j<poc_nasl[akt_cif];++j)
        //vypočtení dalšího sloupečku
        pocty_new[naslednici[akt_cif][j]]+=pocty[akt_cif];

    swap=pocty;pocty=pocty_new;pocty_new=swap;
    //povšimněte si, jak šikovně prohazuji
    //jen pointery na pole a ne celá pole:-)
  }

  int Vysl=0;
  for (i=0;i<L;i++) {
    free(naslednici[i]);
    Vysl+=pocty[i];                     //finální sečtení počtů klíčů
  }
  free(naslednici); free(pocty);        //dealokace proměnných
  free(pocty_new); free(poc_nasl);

  printf("Výsledek je %d\n",Vysl);
  return 0;
}
```

19-1-6 Prolog — Jana Kravalová

1. Tchyně

Co dodat… úloha byla opravdu jednoduchá. Ukázalo se, že pro většinu programátorů byl největší problém vyznat se v rodinných vztazích.

Ale přesto nebyla úložka až tak lehká, jak by se mohlo zdát. Problém nastal u predikátu `manzelstvi(X,Y)`. Můžeme se totiž dohodnout, že první bude vždy muž a predikát tedy bude vypadat jako `manzelstvi(Manzel, Manzelka)` (nebo naopak, to je jedno). Pak samozřejmě musíme v predikátu `tchyne(Tch,X)` otestovat, zda `X` je muž nebo žena, abychom ho dosadili na správné místo v predikátu manželství. Pokud se nechceme rozhodovat, v jakém pořadí budeme manžele a manželku do predikátu `manzelstvi` zadávat, nebo to nevíme, musíme vyzkoušet zavolat predikát `manzelstvi` dvakrát, pokaždé s prohozenými argumenty, aby se chytil a uspěl ten správný zápis pořadí partnerů.

2. Oprava

Úložka za 3 body, ale zdání klame, čeká nás divoký rekurzivní hon. Držte si klobouky, pojedeme z kopce!

Snad každý ihned odhalil, co je na zadaném programu špatně. Predikát `predek` nemá, jak kdosi poznamenal, „šanci dostat se z rekurze“. Prolog neustále vyhodnocuje predikát `predek` a ten donekonečna volá sám sebe. Predikát `rodic` za ním se nikdy nevyhodnotí, nikdy nedojdeme na dno rekurze.

Nu dobrá, ale jak z toho ven? Možné byly dva postupy:

První postup: Snažím se dno rekurze dostat dopředu, aby se vyhodnocovalo jako první, tedy přehodím řádky a program vypadá následovně:

```
predek(Rod,Pot) :- rodic(Rod,Pot).
predek(Pre,Pot) :- predek(MlPr,Pot),rodic(Pre,MlPr).
```

Kdo si tento program alespoň jednou spustil, hned pochopil, že takovéto prohození řádku na opravu ještě není dostatečné. Pokud zadáme existující dvojici `Pred` a `Pot`, predikát funguje. Stačí ale, aby nějaký zlomyslník zadal dvojici, která není navzájem ve vztahu předka a potomka a program se zacyklí. Správně by ale jako slušně vychovaný prologovský program měl odpovědět `No`.

Na toto se nachytalo hodně řešitelů, a proto vysvětlíme, co se děje špatně: Zadáme-li neexistující dvojici `Pred` a `Pot`, první řádek programu se nepovede. Prolog skočí na další řádek a snaží se jej naplnit. Ale tam se zacyklí v marném hledání uspokojivé dvojice pro predikát `predek` a k vyhodnocení predikátu `rodic` nikdy nedojdeme.

Musíme tedy ještě prohodit predikáty `predek` a `rodic` na druhém řádku a dostaneme:

```
predek(Rod,Pot) :- rodic(Rod,Pot).
predek(Pre,Pot) :- rodic(Pre,MlPr),predek(MlPr,Pot).
```

Druhý postup: Nechám řádky tak, jak jsou a jednoduše prohodím predikáty. Dostanu:

```
predek(Pre,Pot) :- rodic(Pre,MlPr),predek(MlPr,Pot).
predek(Rod,Pot) :- rodic(Rod,Pot).
```

Tohle překvapivě také funguje, pouze vydává výsledky při odmítání středníkem v jiném pořadí.

3. Evoluce

Ani tato úložka nebyla záludná pro toho, kdo si přečetl a správně pochopil predikát `predek` a rozmyslel si správně rekurzi.

Plán bude následující: Pro obě rostliny z predikátu `stejny_druh(X,Y)` najdeme jejich nejpůvodnější předky a porovnáme je.

Napišme si tedy predikát `prarost(PraX,X)`, který pro rostlinu `X` najde jejího nejpůvodnějšího předchůdce. Můžeme k tomu použít třeba nám dobře známý predikát `predek`, ale nejdřív si ho trošku upravíme. Tak, jak máme predikát předek napsán teď, je pro nás nevýhodný. Podívejme se na jeho rekurzivní část:

```
predek(Pr,Pot) :- rodic(Pr,MlPr), predek(MlPr,Pot).
```

Takto napsaný predikát vezme daného předka, najde k němu mladšího předka, k němu ještě mladšího předka, až dojde k hledanému potomkovi. Postupujeme tedy ve stromě shora dolů a můžeme se dostat do všech možných

potomků daného předka. Tento dotaz se hodí, pokud bychom chtěli opravdu vyhledávat všechny potomky, ale nás by toto zdržovalo, a tak napíšeme predikát opačně, půjdeme ve stromě zdola nahoru:

```
predek(Pr,Pot) :- rodic(StPr,Pot), predek(Pr,StPr).
```

Vidíte ten rozdíl? K danému potomkovi najdeme jeho rodiče a postupujeme stromem rekurzivně přímo nahoru, nezabýváme se nějakými vedlejšími větvemi.

Ještě potřebujeme dopsat dno rekurze. Kdy skončíme prohledávání? Tady byl kámen úrazu většiny řešitelů. Většina totiž napsala:

```
predek(Pr,Pot) :- mutace(Pr,Pot).
```

Pokud zadáme do takto napsaného predikátu rostlinu, která je již původní, samozřejmě neuspěje. Nesmíme tedy rekurzi zastavit příliš brzy, musíme ji nechat doběhnout až k původnímu druhu:

```
predek(Pr,Pot) :- je_puvodni_druh(Pr), Pr = Pot.
```

Pozor také na konstrukci `Pr=Pot`. Správně bychom měli psát:

```
predek(X,X) :- je_puvodni_druh(X).
```

`X` se správně zunifikuje, pokud se splní predikát `je_puvodni_druh(X).`.

Důvod, proč tyto „triviality“ tak podrobně rozebírám, je ten, že víc jak polovina řešitelů udělala jednu nebo druhou chybu.

A když teď máme predikát `predek` hotový, zbytek je hračka:

```
stejny_druh(X,Y) :- predek(Pr,X), predek(Pr,Y).
```

Existuje ještě jedno pěkné řešení, které vůbec nepoužívá predikáty `predek` ani `je_puvodni_druh`. Obě řešení najdete ve zdrojovém programu.

```
% KSP 19-1-6 Tchyne

% žena(X) znamená, že X je žena
zena(brunhilda).
zena(krasomila).
zena(kazimira).

% muž(X) znamená, že X je muž
muz(jarous).

% rodič(Rodič,Dítě) znamená, že Rodič je rodičem Dítěte
rodic(brunhilda, jason).
rodic(kazimira, krasomila).

% manželé(Manžel, Manželka) znamená, že Manžel a Manželka jsou manželé.
% Domluvme se, že na prvním místě je manžel a na druhém vždy manželka.
manzele(jason,krasomila).

% tchýně(Tchýně, X) zjišťuje, zda Tchýně je tchýní X
tchyne(Tchyne, X) :- zena(Tchyne), manzele(X,Y), rodic(Tchyne,Y).
```

```
tchyne(Tchyne, X) :- zena(Tchyne), manzele(Y,X), rodic(Tchyne,Y).

% muzeme psát také
tchyne(Tchyne, X) :- zena(Tchyne), (manzele(X,Y);(manzele(Y,X)),rodic(Tchyne,Y).


% KSP 19-1-6 Evoluce
% nejprve nějaká ta vstupní data
je_puvodni_druh(rulik1).
je_puvodni_druh(bolehlav1).

mutace(rulik1,rulik2).
mutace(rulik1,rulik3).
mutace(rulik2,rulik4).
mutace(bolehlav1, bolehlav2).

% prarost(PraX,X) uspěje, je-li PraX je evolučním předkem rostliny X
prarost(X,X) :- je_puvodni_druh(X).
prarost(PraX,X) :- mutace(StarsiX,X), prarost(PraX,StarsiX).

% Necháme si najít prarostliny, tedy evoluční předky rostlin X a Y,
% a pokud jsou stejní, jsou i rostliny X a Y
% z jedné vyvojové větve
stejny_druh(X,Y) :- prarost(Pra,X), prarost(Pra,Y).

% Mužeme psát i takhle, ale je to škaredé a neprologovské:
stejny_druh2(X,Y) :- prarost(PraX,X), prarost(PraY,Y), PraX = PraY. % FUJ!

% Úplně jiné řešení, také pěkné
stejny_druh3(X,X).
stejny_druh3(X,Y) :- mutace(StarsiX,X), stejny_druh3(StarsiX,Y).
stejny_druh3(X,Y) :- mutace(StarsiY,Y), stejny_druh3(X,StarsiY).
```

19-2-1 Čokoláda podruhé — Cyril Hrubiš

Jak si mnoho z vás všimlo, existuje docela jednoduchá vyhrávající strategie pro prvního hráče. V prvním kroku náš začínající bandita rozlomí čokoládu na dvě totožné části (všimněme si, že je to možné pouze tehdy, máme-li alespoň jeden rozměr sudý) a dále už jen opakuje podle osy prvního zlomu soupeřovy tahy do té doby, než vyhraje.

Uvědomme si, že taková strategie určitě vede k vítězství. Pokud jsme hráli podle popsané strategie a prohráli jsme, tj. odlomili jsme kostičku 1×1, udělali jsme to proto, že náš soupeř udělal to samé v minulém tahu – musel tedy prohrát on.

To bylo v případě, že hra skončí. Mohlo by se teoreticky stát, že budeme lámat donekonečna a nikdo neprohraje. V našem případě se to ale určitě nestane, protože se čokoláda skládá jen z konečného počtu čtverečků.

19-2-2 Kvalitní hesla — Jan Bulánek

Jako obvykle jsem chtěl začít své řešení vtipnou poznámkou či glosou, ale vzhledem k tomu, že mi zubařka o Vánocích vyvrtala 4 zuby a já mohl všechny ty cukrovinkami se cpoucí lidi jen sledovat, jistě chápete, že na vtipy nemám náladu. Takže k řešení.

První nápad je vyzkoušet všechny možné podřetězce hesla a vzájemně je porovnat. Bez ohledu na vámi navrhované heuristiky pracuje toto řešení v čase $O(N^3)$ vzhledem k délce hesla v nejhorším případě s pamětí $O(N)$, což jste si povětšinou správně uvědomovali a zasloužíte si pochvalu. Pokud jste mi přece jen tvrdili, že je to rychlejší, tak se ještě jednou zamyslete, zda skutečně neexistuje nějaký protipříklad. Pokud by vás přece jen nenapadl (případně já udělal chybu), ozvěte se.

Ono to ale jde o něco (a následně o dost) lépe. Představme si, že máme dobrého kamaráda a ten nám poví, jak daleko od sebe leží začátky shodných podstringů. Pak je ale snadné zjistit, kde se takové podřetězce nachází, prostě jen projdeme celé heslo a nalezneme nejdelší posloupnost shodných dvojic v této vzdálenosti. A to zvládneme jedním průchodem. No a co když nemáme dobré kamarády? Tak prostě prozkoušíme všechny možné vzdálenosti a jen si zapamatujeme, kde jsme dosáhli maxima. Že takových vzdáleností je N a časová složitost $O(N^2)$ nás tudíž nemine, jistě nemusím dodávat.

Nyní si sedněte, udělejte si pohodlí a připravte kyblíčky (na nervy). Začnu slovem *suffixový strom*. Už tím jsem vás jistě odradil, ale pro ty skutečně otrlé dodám ještě odkaz

http://www.dogma.net/markn/articles/suffixt/suffixt.htm

a zároveň velmi poděkuji Kubovi Kaplanovi za tento odkaz a potřebnou inspiraci. Pojednání je bohužel anglicky, takže přidávám ještě jedno české:

http://mj.ucw.cz/vyuka/ga/,

jež je obklopeno ještě několika dalšími zajímavými algoritmy. Po přečtení těchto článečků zjistíte, co to takový suffixový strom je, ke svému údivu objevíte, že se dá zkonstruovat lineárně a s lineární pamětí (no já taky zíral) a ... využijete služeb kyblíčku. Pro ty méně otrlé připojím malý popis. Nejprve vysvětlení: *suffix* je „koncovka“ slova, např. pro slovo `book` získáme suffixy `book`, `ook`, `ok`, `k` a prázdný suffix. Suffixový strom je pak *trie,* ve které jsou uloženy všechny suffixy zadaného řetězce (pokud nevíte co je trie, zapátrejte v ročenkách). Ta má zjevně paměťovou složitost až $O(N^2)$ a logicky s tím i čas na její stavbu je $O(N^2)$. Tím bychom si moc nepomohli, a tak tuto trii vylepšíme. Všechny vrcholy, které mají jen jednoho následníka, sloučíme s jejich otci. V naší trii tak například budou sloučeny vrcholy `b`→`o`→`o`→`k` do jednoho vrcholu `book`. Tím se paměťová složitost zmenší na lineární (uvědomte si, že přidáním suffixu do takto komprimované trie přidáme maximálně dva vrcholy a že všechny řetězce přiřazené k hranám trie jsou podslova zadaného slova, takže

si je stačí pamatovat jako polohu začátku a konce v tomto slovu). Technické detaily již ponechám článku a teď k vlastnímu řešení.

Na konec hesla připojím nějaký nikde se nevyskytující znak a nad takto vytvořeným řetězcem vytvořím suffixový strom. Jak se nám projeví společná část suffixů? To je část od kořene stromu k prvnímu místu, kde se tyto suffixy rozdělí. Je velmi důležité si uvědomit, že k tomu musí dojít, protože máme-li dva suffixy, tak se tyto musí lišit v délce, a tudíž kratší se oddělí na nějakém znaku od delšího (liší se přinejmenším koncovým znakem hesla, který byl přidán a jistě se uprostřed delšího suffixu nevyskytuje). Tyto společné části suffixů ale hledáme, protože každý podstring je zjevně začátkem (libovolně dlouhým!) alespoň jednoho suffixu. Námi hledaný řetězec se tedy jistě vyskytuje na začátku alespoň dvou suffixů (rozmyslet!).

Příklad: pro heslo `ananas` a suffixy `ananas`, `nanas`, `anas`, `nas`, `as`, `a` mají nejdelší společný začátek `ananas` a `anas` a výsledkem je `ana`. Dále si všimněte, že suffix `anas` krom toho začíná na řetězce `a`, `an`, `ana`, `anas`.

Nejdelší společnou část suffixů pak snadno najdeme jedním průchodem stromu do hloubky, při kterém budeme hledat nejdelší cestu z kořene do vrcholu s alespoň dvěma následníky. To je díky lineární velikosti stromu též lineární, a tak jsou celkové složitosti paměťová i časová $O(N)$. Ale ptám se vás, stálo to za to? :–)

Poznámka M.M.: Byl bych sice ten poslední, kdo by se ošklíbal nad suffixovými stromy, jenže mi to přece jen nedá, abych neukázal ještě jedno řešení, které má sice o trochu horší časovou složitost, ale vystačí si s daleko jednodušší mašinerií, konkrétně s hešováním z kuchařky v této sérii a přihrádkovým tříděním.

Když vymyslíme algoritmus (budeme mu říkat třeba *pohrabáč,* protože se jím prohrabujeme řetězcem), který v lineárním čase pro dané k zjistí, zda se v zadaném řetězci vyskytuje nějaký podřetězec délky k vícekrát, můžeme použít půlení intervalu na nalezení největšího takového k, přičemž pohrabáč použijeme jen $\log N$-krát.

Lineární pohrabáč vám sice nenabídnu, ale ukážu, jak to udělat alespoň v *průměrně lineárním čase.* Zvolíme si šikovnou hešovací funkci, která bude hešovat k-znakové řetězce do N^2 přihrádek a bude mít navíc tu vlastnost, že pokud jsme ji spočítali pro znaky $a_1 \ldots a_k$, dokážeme ji z toho v konstantním čase spočítat pro $a_2 \ldots a_{k+1}$ (zkuste si rozmyslet, že funkce `hash_string` z kuchařky toto splňuje). Tak dokážeme zahešovat všechny podřetězce délky k v lineárním čase a víme, že pokud se nějaký vyskytl vícekrát, určitě oba výskyty skončí v jedné přihrádce. Stačí tedy po zahešování projít všechny kolize a zjistit, jestli existovaly dva stejné podřetězce. Navíc platí, že při tomto počtu přihrádek je průměrný počet kolizí $O(1)$ (to si nedokážeme, ale s pomocí povídání o pravděpodobnosti a středních hodnotách z 16. ročníku to snadno vymyslíte),

takže probrání všech kolidujících párů zvládneme v čase $O(k) = O(N)$.

Jenže ještě tu je jeden háček: přihrádek jsme zvolili N^2, a tak si nemůžeme dovolit přihrádky reprezentovat polem, protože jenom na jeho projití potřebujeme kvadratický čas. Proto si prostě ke každému začátku podřetězce poznamenáme stranou hodnotu hešovací funkce a začátky podle těchto hodnot setřídíme – to jde přihrádkovým tříděním s N přihrádkami stihnout v lineárním čase, což přesně potřebujeme. Poté porovnáme všechny podřetězce se stejnou hešovací funkcí a zjistíme, zda jsou nějaké dva shodné.

Celkově tedy umíme pohrabovat v čase $O(N)$ průměrně a požadované k najít v $O(N \log N)$ a lineární paměti.

```
program heslo;

var buf:string;
    max_start1, max_start2, max_len : integer;
    len, i, j , buf_len: integer;
begin
    readln(buf);
    buf_len:= length(buf);

    max_len:=0;
    for i:= 1 to buf_len-1 do begin  { pro každý rozestup začátků }
        len:=0;                      {hledam nejdelší shodující se posloupnosti}
        for j:= 1 to buf_len-i do begin
            if (buf[i+j]=buf[j]) then begin
                Inc(len);
                {pokud se znaky shodují, prodloužíme posloupnost shodných znaků}
                if (len>max_len) then begin
                    max_len:=len;
                    max_start1:=j-len+1;
                    max_start2:=max_start1+i;
                end;
            end
            else len:=0;  { jinak počitadla vynuluju }
        end;
    end;

    writeln('Řetězce mají délku ', max_len, ' a začínají na ',
            max_start1, ' a ', max_start2);
    readln;

end.
```

19-2-3 Moneymaker — Pavel Čížek

Asi nejjednodušší řešení této úlohy by se dalo popsat slovy „když metoda *hrr na ně* nezabere, tak se stáhneme a zkusíme to zezadu". Až na jedno řešení využívající intervalové stromy skončili všichni řešitelé začínající od počátku kvadratickou, popř. ještě horší časovou složitostí. Nyní ale zpět k tomu, jak se úloha měla řešit.

Označme T termín nejméně spěchající zakázky. Budeme postupně, pro jednotlivé časy $t < T$, generovat pořadí plnění zakázek (označme je $A^t_t, A^t_{t+1}, \ldots, A^t_T$), kterým maximalizujeme zisk v časovém úseku $< t; T >$. Pokud zjistíme, jak toto pořadí vypadá pro $t = 1$, tak máme hotovo.

Pro $t = T$ je to jednoduché. Mezi všemi zakázkami s termínem T vybereme tu, která je nejlépe placená. Nyní předpokládejme, že známe optimální pořadí zakázek od času $t+1$ (tj. známe $A^{t+1}_{t+1}, A^{t+1}_{t+2}, \ldots, A^{t+1}_T$). Pak tvrdím, že jedna z možných sekvencí zakázek s maximálním ziskem je:

- $A^t_i = A^{t+1}_i$ pro $i \geq t+1$
- A^t_t nalezneme jako zakázku s maximální odměnou, která má termín t, nebo pozdější, a kterou jsme ještě nepoužili (tj. není mezi $\{A^{t+1}_i\}$).

Dokáže se to snadno. Pro spor předpokládejme, že známe nějaké pořadí $B^t_t, B^t_{t+1}, \ldots, B^t_T$, které nám zajistí lepší zisk.

Zároveň ale víme, že odměna za úkoly $A^t_{t+1}, A^t_{t+2}, \ldots, A^t_T$ je alespoň stejně velká jako za zakázky $B^t_{t+1}, B^t_{t+2}, \ldots, B^t_T$ (z toho, že jsme předpokládali, že $\{A^{t+1}_i\}$ maximalizuje zisk na časovém intervalu $< t+1; T >$). Z toho plyne, že odměna za B^t_t je větší než odměna za A^t_t. Jelikož ale A^t_t má maximální odměnu ze všech zakázek, která nebyly obsaženy v $\{A^{t+1}_i\}$, musí tedy existovat $j > t$ takové, že $A^{t+1}_j (= A^t_j) = B^t_t$, nebo jsme dostali spor. Prohodíme tedy v posloupnosti $\{B^t_i\}$ pozice úkolů B^t_t a B^t_j (to si můžeme dovolit, jelikož pak úkol B^t_j splníme dřív a zakázku B^t_t můžeme splnit až v čase j, protože se až tak pozdě vyskytovala v posloupnosti $\{A^{t+1}_i\}$, která termíny respektuje). Tím jsme zřejmě nezměníme celkovou odměnu za úkoly v $\{B^t_i\}$ a tedy celý tento odstavec můžeme použít na novou posloupnost $\{B^t_i\}$ úplně stejně.

To jsme ale ještě nic dokázali, jak si jistě čtenář všiml. Spor dostaneme, až když si uvědomíme, že výše uvedené nemůžeme opakovat donekonečna. Pokud budeme uvažovat počet úkolů, které jsou v $\{A^t_i\}$ a $\{B^t_i\}$ na stejném místě (tj. počet takových k, že $A^t_k = B^t_k$), tak v každém cyklu stoupne o 1 (úkol B^t_t se dostane na stejné místo jako je v posloupnosti $\{A^t_i\}$), tedy po několika opakováních výše uvedeného musíme někdy dostat spor.

No a jak toto nejlépe implementovat? Nejdřív setřídíme úkoly dle termínu. Pak budeme odzadu generovat jednotlivé úkoly, které je třeba v daný čas t udělat. K tomu použijeme haldu. Budeme si v ní udržovat úkoly, které mají termín t či pozdější a které jsme zatím ještě nezařadili mezi zakázky, které

splníme. Na začátku bude prázdná a v každém kroku do haldy přidáme všechny úkoly, které mají termín t (pozdější tam již máme z předchozích kroků) a odebereme maximum. Tím jsme skoro hotovi.

Kdybychom implementovali výše uvedené doslovně, tak čas běhu programu bude kromě velikosti vstupu záviset i na nejpozdějším termínu úkolu. Toho je ale možno se jednoduše zbavit. Pokud bude halda prázdná, tak můžeme rovnou posunout čas na nejbližší dřívější termín zakázky, čímž si ušetříme čas.

No a složitost. Setřídění pomocí rychlého třídícího algoritmu trvá $O(N \cdot \log N)$. Přidání do haldy zabere $O(\log N)$ a provádíme ho N-krát, tedy opět $O(N \cdot \log N)$. No a ještě z haldy odebíráme kořen. To uděláme také maximálně N-krát a trvá to $O(\log N)$. Dohromady tedy $O(N \cdot \log N)$.

V paměti máme vstup a haldu. Jejich velikost je přímo úměrná velikosti vstupu, tedy paměťová složitost je $O(N)$.

```
const MaxN = 10000;

type TUkol = record
   Odmena:integer;
   Termin:integer;
   CisloUkolu:integer;
end;

var Ukoly:array[1..MaxN] of TUkol;  {Za co jsou nám ochotni lidi zaplatit ...}
    N:integer; {počet úkolů}
    DalsiUkol:integer;   {První úkol, na který ještě nepřišla řada,
                         tj. zatím jsme uvažovali jen vyšší časy.}

    VelikostHaldy:integer;
    {Halda, ze které vybíráme nejvhodnější úkol pro daný čas.}
    Halda:array[0..MaxN-1] of TUkol;
    {U kolika úkolů už jistě víme, že je uděláme a kdy.}
    VykonanychUkolu:integer;
    Vykonane:array[1..MaxN] of integer; {Čísla zakázek, co doopravdy uděláme.}

    Cas:integer;     {Čas, pro který se rozhodujeme, co uděláme.}

procedure NactiVstup();
var i:integer;
begin
   readln(N);
   for i:=1 to N do with Ukoly[i] do begin
      readln(Termin,Odmena);
      CisloUkolu:=i;
   end;
end;

procedure VypisVysledek();
var i:integer;
```

```
begin
   write('Nejvýhodnější pořadí je: ');
   {máme je uloženy v opačném pořadí, než je je třeba vykonat}
   for i:=VykonanychUkolu downto 1 do begin
      write(Vykonane[i],' ');
   end;
   writeln;
end;

procedure SeradDleTerminu(Min,Max:integer);
{Seřadí dle termínu dokončení sestupně, tj. nejméně spěchající úkoly na konec.}
{Je to obyčejný QuickSort.}
var L,R:integer;
    Pivot:integer;
    Swap:TUkol;
begin
  L:=Min; R:=Max;
  Pivot:=Ukoly[(Min + Max) div 2].Termin;
  repeat
    while Ukoly[L].Termin>Pivot do inc(L);
    while Ukoly[R].Termin<Pivot do dec(R);
    if L<=R then
      begin
        Swap:=Ukoly[L];
        Ukoly[L]:=Ukoly[R];
        Ukoly[R]:=Swap;
        inc(L); dec(R);
      end;
  until L >= R;
  if R>Min then SeradDleTerminu(Min,R);
  if L<Max then SeradDleTerminu(L,Max);
end;

procedure PridejDoHaldy(Co:TUkol);
{Přidá úkol do haldy}
var Pozice:integer; {Na jakém místě je (možná) nekonzistence haldy}
    Rodic:integer;  {Úkol, který je v haldě o hladinu výš}
    Swap:TUkol;
begin
   Pozice:=VelikostHaldy;
   inc(VelikostHaldy);
   Halda[Pozice]:=Co;
   while (Pozice > 0) do begin  {Bubláme ke kořeni}
      Rodic:=(Pozice - 1) div 2;
      if (Halda[Rodic].Odmena > Halda[Pozice].Odmena)
        then break; {Dál se to už nezmění ...}
      Swap:=Halda[Pozice];
      Halda[Pozice]:=Halda[Rodic];
      {tak jsme vybublali o hladinu výš a všechno můžeme zopakovat}
      Halda[Rodic]:=Swap;
      Pozice:=Rodic;
```

```
   end;
end;

procedure VyradKorenZHaldy();
{Odstraní kořen z haldy ...}
var Pozice:integer; {Kde je (možná) nekonzistence v haldě...}
    Syn:integer; {Syn vrcholu, který právě uvažujeme}
    Swap:TUkol;
begin
   dec(VelikostHaldy);    {Halda se zmenší}
   Halda[0]:=Halda[VelikostHaldy];
   Pozice:=0;
   repeat
      Syn:=Pozice * 2 + 1;
      if Syn >= VelikostHaldy then break;   {už jsme úplně dole}
      if Syn+1 < VelikostHaldy then
         {uvažovaný vrchol má 2 syny, tak vybereme úkol s vyšší odměnou}
         if (Halda[Syn+1].Odmena > Halda[Syn].Odmena) then inc(Syn);

      if Halda[Syn].Odmena < Halda[Pozice].Odmena then break;
         {pokud všechny zakázky níž už jsou hůř placené, tak jsme hovovi ...}

      Swap:=Halda[Syn];
      Halda[Syn]:=Halda[Pozice];
      Halda[Pozice]:=Swap;
      {prohodíme, aby to na dané hladině bylo v pořádku a klesneme níž}
      Pozice:=Syn;
   until false;   {ven budeme skákat pomocí breaku}
end;


begin
   NactiVstup();
   if (N < 0) then begin
      writeln('Není co dělat.');
   end else begin
      SeradDleTerminu(1,N);

      VelikostHaldy:=0;      {Inicializace haldy}
      Cas:=Ukoly[1].Termin;  {Máme setřízeno -> tohle je maximální možný čas}
      VykonanychUkolu:=0;    {Zatim jsme nic neudělali}
      DalsiUkol:=1;          {a taky jsme se ještě na nic nepodívali}

      while (Cas > 0) do begin  {Projdeme všechny časy odzadu}
         while ((DalsiUkol <= N) and (Ukoly[DalsiUkol].Termin = Cas)) do begin
            {přidáme do haldy všechny úkoly, které končí v tento čas}
            PridejDoHaldy(Ukoly[DalsiUkol]);
            inc(DalsiUkol);
         end;

         inc(VykonanychUkolu);
```

```
        {nejvýhodnější je v kořeni haldy}
        Vykonane[VykonanychUkolu]:=Halda[0].CisloUkolu;
        VyradKorenZHaldy();                    {a vyřadíme ho, je už hotový}

        if VelikostHaldy = 0 then begin
           if DalsiUkol > N then
              break {už jsme udělali všechno, co se dalo a ještě nám zbyl čas}
           else
              Cas:=Ukoly[DalsiUkol].Termin;
 {nějakou dobu nemáme co dělat a tak skočíme rovnou na další zajímavou položku}
        end else
           dec(Cas);  {Jinak se jen posuneme zase v čase o trochu zpět}
     end;

     VypisVysledek();
  end;
end.
```

19-2-4 Optimální formace — Tomáš Gavenčiak

Tato úloha, na první pohled docela snadná – stačilo najít souřadnice vrcholů *jakéhokoli* konvexního mnohoúhelníka, byla nakonec docela těžká. První, byť jen malý zádrhel, byl v tom, uvědomit si, že každý střelec musí svůj dostřel využít naplno. Pokud bychom totiž u střelce S_i využili jen část dostřelu tak, aby všechny vnitřní úhly byly ostře menší než 180°, mohli bychom útvar malilinko „zplacatit" a dostat tak ještě o kousek větší obvod. Takhle bychom se buď u každého dostali na maximální dostřel, nebo maximum neexistuje.

Kdy jde nakreslit konvexní mnohoúhelník o zadaných stranách? Pokud splňuje zobecnění trojúhelníkové nerovnosti – mnohoúhelníkovou nerovnost, která říká, že pro každou hranu S_i musí součet délek ostatních hran být větší než délka S_i. Tuto nerovnost stačí ověřit pro S_i nejdelší stranu. Nejdelší stranu budu dále označovat S_0, ostatní pak S_1, S_2, S_{n-1} v pořadí v jakém navazovaly na S_0. Vrcholy označím $V_0, \ldots V_{n-1}$, přitom $S_i = (V_i, V_{i+1})$ a $S_{n-1} = (V_{n-1}, V_0)$.

Jak teď sestrojit konkrétní souřadnice? Mohli bychom si mnohoúhelník představit jako trojúhelník $V_0V_1V_l$, kde l je co nejblíže polovině vzdálenosti $V_1 - V_0$ mimo S_0. Takový trojúhelník splní trojúhelníkovou nerovnost a celkem snadno můžeme spočíst souřadnice jeho vrcholů. Teď už stačí body na přímkách $V_1 - V_l$ a $V_l - V_0$ lehce „vyboulit" abychom dosáhli úhlů menších než 180° a přitom si nepokazili sestrojitelnost ani konvexnost. Jak na to?

Jedna z možností je umístit strany $S_1, S_2, \ldots S_{n-1}$ jako tětivy po obvodu *dostatečně velké* kružnice k a tu pak ve vhodném bodě V_l „zlomit" a přiblížit tím body V_0 a V_1 na délku úsečky S_0. Jak velkou kružnici k zvolit? Stačí, když po tom, co na ní umístíme S_0 jako tětivu, délka oblouku V_0V_1 bude menší nebo rovna obvodu mnohoúhelníka bez nejdelší strany. Nyní se dostává ke slovu analytická geometrie černější než černá magie, a proto nebudu všechny kroky

zdůvodňovat a důkazy korektnosti jednotlivých voleb přenechám laskavému čtenáři.

Obvod mnohoúhelníka bez nejdelší strany označím S_u Poloměr kružnice k zvolím

$$r \geq \frac{S_u S_0}{\sqrt{S_u^2 S_0^2}}.$$

Střed této kružnice umístím na souřadnici $[r, 0]$, od souřadnice $[0, 0]$ začnu směrem nahoru po kružnici umisťovat body $V_1, V_2, \ldots V_{n-1}, V_0$ ve správných vzdálenostech na souřadnice $[x_i, y_i]$. Jak spočítat souřadnice těchto bodů? Další bod $V_{i+1} = [x_{i+1}, y_{i+1}]$ musí být ve správné vzdálenosti od bodu $V_i = [x_i, y_i]$ i středu kružnice k $[0, r]$, musí tedy platit

$$(x_{i+1} - r)^2 + y_{i+1}^2 = r^2$$

$$(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 = S_i^2$$

Vyřeším kvadratickou rovnici, ze které dostanu dvě možnosti pro V_{i+1}. Kružnici jsem si zvolil dvakrát tak velkou, než bych nutně potřeboval, a proto mohu spoléhat na to, že pokládám vždy směrem nahoru. Označím-li si pro jednotlivá i

$$A_{i+1} = 1 + \frac{(r - x_i)^2}{y_i^2}$$

$$B_{i+1} = \frac{(r - x_i)(x_i^2 + y_i^2 + S_i^2)}{y_i^2} - 2r$$

$$C_{i+1} = \frac{(x_i^2 + y_i^2 + S_i^2)^2}{4y_i^2},$$

dostanu $x_{i+1} = (B_{i+1} + \sqrt{B_{i+1}^2 - 4A_{i+1}C_{i+1}})/2A_{i+1}$ a

$$y_{i+1} = \frac{x_i^2 + y_i^2 + S_i^2 + 2x_{i+1}(r - x_i)}{2y_i}.$$

Bod V_l zvolím co nejblíže polovině vzniklého oblouku V_1V_0. Je třeba dopočíst finální souřadnici bodu V_0 tak, aby byl od V_1 vzdálen S_0 a od V_0 vzdálen F. Toto opět vede na soustavu kvadratických rovnic, z jejichž dvou řešení vybereme to s větší souřadnicí y. Výpočet souřadnic $[x_0, y_0]$ je jen rutina a přenechám ji odvážnému čtenáři, který se dočetl až sem.

Předposlední částí výpočtu je otočit body $V_{l+1}, V_{l+2}, \ldots, V_{n-1}$ okolo bodu V_l o stejný úhel, o jaký byl otočen bod V_0 (ten se pohyboval na kružnici se středem v V_l). To je možné třeba spočtením tohoto úhlu goniometrickými funkcemi a sestavením transformační matice nebo řešením dalších soustav kvadratických rovnic.

Poslední částí je pak jednoduché vypsání výsledků, které je po tom všem už hračkou.

Časová i paměťová složitost tohoto algoritmu je lineární. Paměťovou složitost by bylo možno snížit na konstantní, pokud bychom některé hodnoty zapomínali a v průběhu výpočtu si je znovu dopočítali. Program neuvádíme, protože se skládá pouze z výpočtů pomocí zde navržených vzorců.

19-2-5 Hluboký les — Martin Mareš

Je zajisté triviální nalézt les nejhlubší zkoumáním vzdáleností všech dvojic stromů, ale uznejte sami, že za to bychom sotva slibovali 13 bodů, protože je to cca desetiřádkový program s ošklivou kvadratickou složitostí. Zkrátka to, čemu se říkává dřevorubecké řešení. Pojďme se raději zakoukat do hladiny křišťálové studánky, jestli nám neporadí, jak na to jít lépe (třeba od lesa):

Stromy si představme jako body v rovině, x-ová souřadnice bude odpovídat směru zleva doprava, y-ová shora dolů. Vzdálenost stromů $S_1 = (x_1, y_1)$ a $S_2 = (x_2, y_2)$ bude činit:

$$d(S_1, S_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Kdo jste tento vzoreček ještě nepotkali, vzpomeňte si na pana Pythagora a jeho větu – chceme změřit přeponu pravoúhlého trojúhelníka S_1TS_2 s pravým úhlem u vrcholu $T = (x_2, y_1)$. Místo vzdáleností budeme ale raději porovnávat jejich druhé mocniny, což jsou pro celočíselné souřadnice bodů také celá čísla. Tak si ušetříme starosti se zaokrouhlovacími chybami a program bude nadále fungovat, jelikož $x < y$ platí právě tehdy, když $x^2 < y^2$, tedy aspoň pro nezáporná čísla, což výraz pod odmocninou bezpochyby je.

Ještě si všimněme jednoho zajímavého faktu: pokud chceme do čtverce velikosti $d \times d$ umisťovat body tak, aby vzdálenost každých dvou byla alespoň d, vejdou se tam maximálně čtyři (třeba do vrcholů čtverce). Dokázat to můžeme například tak, že čtverec rozřežeme na čtyři menší čtverce velikosti $d/2 \times d/2$, které budou mít společné hrany, a nahlédneme, že do každého z nich můžeme umístit nejvýše jeden bod. Nejvzdálenější body v malém čtverci jsou totiž jeho protilehlé vrcholy a ty mají vzdálenost $d\sqrt{2}/2 < d$.

Jak onehdy naznačili jistí programátorští kuchaři, hodit by se mohla metoda Rozděl a panuj. Ta by se pro hledání nejbližší dvojice bodů dala použít zhruba následovně:

- Rozděl všechny body vodorovnou přímkou do dvou stejně velkých množin X_1 a X_2.
- Rekurzivním zavoláním algoritmu najdi minimální vzdálenost d_1 dvojic bodů v X_1 a d_2 v X_2.
- Doplň dvojice sahající přes hraniční přímku: zajímají nás jen takové dvojice, které mohou změnit výsledek, čili jejichž vzdálenost je

menší než $d = \min(d_1, d_2)$. Proto stačí uvážit body vzdálené od hraniční přímky méně než d (ostatní body mají moc daleko k hraniční přímce, natož k bodům v druhé množině). Projdeme všechny dvojice takových bodů a označíme d_3 minimum z jejich vzdáleností.

- Vrať jako výsledek $\min(d_1, d_2, d_3)$.

Pokud by první a třetí krok algoritmu běžely v lineárním čase, choval by se celý algoritmus podobně jako QuickSort s rovnoměrným dělením, který jsme ukazovali v kuchařce, a tedy by jeho časová složitost byla $O(N \log N)$ a paměťová $O(N)$. Stručně: Na vstup délky N spotřebujeme čas $O(N)$ plus ho rozložíme na dva vstupy délky $N/2$. Pro ty potřebujeme dohromady také čas $O(N)$ plus je rozdělíme na čtyři vstupy délky $N/4$, a tak dále, až se po $\log_2 N$ krocích dostaneme ke vstupům délky 1 a celkem tedy spotřebujeme čas $O(N \log N)$. To je velmi lákavá představa, jen zatím poněkud efemérní, jelikož není vůbec jasné, jak první a třetí krok provést.

Rozdělování bodů: Nabízí se vybrat souřadnici rozdělovací přímky náhodně (podobně jako u QuickSortu bychom se tak dostali na průměrně rovnoměrné rozdělení) nebo si vzpomenout na lineární algoritmus pro výpočet mediánu uvedený v kuchařce. Oba přístupy ale mají společný háček: pokud většina stromů leží na jedné vodorovné přímce, vybereme nejspíš tuto přímku a body rozdělíme nerovnoměrně. Tomu by se dalo odpomoci dělením na tři části – body ležící na dělící přímce bychom zpracovali úplně zvlášť, beztak padnou do pásu, ve kterém dvojice kontrolujeme explicitně.

Mnohem jednodušší je na začátku algoritmu setřídit všechny body podle svislé souřadnice a rozdělit je prostě na prvních $\lfloor N/2 \rfloor$ a zbylých $\lceil N/2 \rceil$. Různé body na dělící přímce sice mohou padnout do různých polovin, ale to není nikterak na škodu, stejně je následně všechny probereme. Třídění nám časovou složitost nepokazí a rozdělování pak dokonce zvládneme v konstantním čase.

Porovnávání hraničních dvojic: Dvojic může být až kvadraticky mnoho (představte si všechny body ležící na dvou vodorovných přímkách), takže je musíme probírat šikovně. Kdybychom je měli setříděné zleva doprava, stačilo by pro každý bod B prozkoumat jen několik bodů od něj doprava – jakmile x-ová vzdálenost překročí d, nemá smysl dál hledat. Zajímají nás tedy body z X_1 ležící ve čtverečku $d \times d$ bezprostředně nad přímkou a body z X_2 ve stejně velkém čtverečku pod přímkou. A my už víme, že v každém z těchto dvou čtverečků mohou ležet nejvýše 4 zajímavé body (každé dva body ležící v téže množině jsou přeci vzdálené aspoň d a použijeme pozorování o umisťování do čtverečků). To je celkem 8 bodů, navíc jedním z nich je náš bod B, čili pro každý bod B zbývá prozkoumat jen 7 následníků. To snadno stihneme v lineárním čase.

Předpokládali jsme ale, že prvky máme setříděné. To skutečně máme, jenže podle druhé souřadnice, než potřebujeme. Jak z toho ven? Jistě můžeme body

na počátku setřídit podle každé souřadnice zvlášť a při rozdělování udržovat obě poloviny také setříděné oběma způsoby, ale opět bychom se dostali do potíží s mnoha body na jedné přímce. Proto se uchýlíme k drobnému úskoku: zabudujeme do naší funkce třídění sléváním: funkce na vstupu dostane body setříděné podle y a vrátí je setříděné podle x. To půjde snadno, jelikož z rekurzivních volání dostane každou polovinu správně setříděnou, a tak je jen v lineárním čase slije.

Tím jsme doplnili bílá místa v algoritmu a zbývá ukázat program. Je napsaný v C99 a drží se téměř doslovně našeho algoritmu. Vypisuje pouze nalezenou vzdálenost, ale sami jistě vymyslíte, jak do něj dodělat, aby vypisoval i souřadnice místa, kde nejhlubší je les.

Pár poznámek na závěr:

- Sedmička je trochu přemrštěný odhad: zajímají nás pouze ty dvojice, jejichž vzdálenost je *ostře* menší než d, takže čtverce, ve kterých body mohou ležet, jsou o maličko menší než $d \times d$ a do takových se už vejdou jen tři body (zkuste si dokázat). Správná konstanta je tedy 5.
- Také bychom mohli zkoumat na švu body z X_1 a hledat k nim do páru body z X_2. Pro každý bod z X_1 leží kandidáti z X_2 v obdélníku $2d \times d$ a do něj se vejde nejvýše 6 bodů, což Marek Nečada pěkně dokázal rozřezáním na 6 kousků velikosti $2d/3 \times d/2$ s úhlopříčkou délky $5d/6$.
- Algoritmus, který jsme použili pro zkoumání dvojic ležících na švu, by bylo možné použít i na celou úlohu: body setřídíme podle jedné ze souřadnic a pro každý bod zkoušíme do dvojice jen ty, které jsou v této souřadnici vzdálené maximálně tolik, kolik činí zatím nejmenší nalezená vzdálenost. To může být v nejhorším případě také kvadratické, ale v průměru se dostaneme na $O(N \cdot \sqrt{N})$. Idea důkazu (podle Zbyňka Konečného): leží-li všechny body v obdélníku $a \times b$ a minimální vzdálenost činí d, nesmí se kruhy o poloměru $d/2$ se středy v zadaných bodech protnout, takže součet jejich obsahů $N\pi d^2/4$ smí být maximálně $(a+2d)(b+2d)$ (kruhy mohou na krajích z obdélníků přečuhovat až o d). Dostaneme kvadratickou nerovnici pro d a z ní po pár úpravách $d = O(\min(a,b)/\sqrt{N})$.

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

#define MAX 1000                                    // Maximální velikost vstupu
#define INFTY 1000000000                            // Nekonečno :-)

typedef struct {                                    // Pozice jednoho stromu
        int x, y;
} tree;
```

```
int N;                                                  // Počet stromů
tree trees[MAX];                                        // Stromy
tree temp[MAX];                                         // Pole pomocné víceúčelové

// Minimum a druhá mocnina
int min(int x, int y) { return (x < y) ? x : y; }
int sqr(int x) { return x*x; }

// Místo vzdáleností počítáme vždy jejich druhé mocniny,
// předpokládáme, že jsou < INFTY.
int distq(tree a, tree b) { return sqr(a.x - b.x) + sqr(a.y - b.y); }

// Slití dvou setříděných úseků (a[0...mid-1], a[mid...n-1]) do jednoho
// (a[0...n-1]). Pokud by_x>0, třídíme podle X, jinak podle Y.
void merge(tree *a, int mid, int n, int by_x)
{
        int i=0, j=mid, k=0;
        while (k < n)
                if (j >= n || i < mid &&
                    (by_x ? (a[i].x <= a[j].x) : (a[i].y <= a[j].y)))
                        temp[k++] = a[i++];
                else
                        temp[k++] = a[j++];
        for (i=0; i<n; i++)
                a[i] = temp[i];
}

// Třídění pole stromů podle Y (nerekurzivní varianta MergeSortu,
// když už máme slévání)
void sort_by_y(void)
{
        for (int i=1; i<N; i *= 2)
                for (int j=0; j+i < N; j += 2*i)
                        merge(trees+j, i, min(2*i, N-j), 0);
}

// Jádro pudla -- rekurzivní funkce na hledání vzdálenosti.
// Na vstupu stromy setříděny podle Y, na výstupu už podle X.
int find_dist(tree *a, int n)
{
        if (n < 2)                              // Jo tyhle triviální případy...
                return INFTY;
        int mid = n/2;                          // Rozdělíme přesně v polovině
        int mid_y = a[mid].y;                   // Pozice dělící přímky
        int d1 = find_dist(a, mid);             // Rekurzivně zpracujeme poloviny
        int d2 = find_dist(a+mid, n-mid);
        int d = min(d1, d2);                    // Dosavadní minimum
        merge(a, mid, n, 1);                    // Dotřídíme podle X

        int p = 0;                              // Najdeme body ležící na švu
        for (int i=0; i<n; i++)
```

```
                if (sqr(a[i].y - mid_y) < d)
                        temp[p++] = a[i];
        for (int i=0; i<p; i++)            // Porovnáváme je se sedmi následníky
                for (int j=i+1; j<p && j<=i+7; j++)
                        d = min(d, distq(temp[i], temp[j]));
        return d;                          // Výsledná vzdálenost (kvadratická)
}

int main(void)
{
        scanf("%d", &N);
        for (int i=0; i<N; i++)
                scanf("%d%d", &trees[i].x, &trees[i].y);
        sort_by_y();
        printf("minimální vzdálenost: %f\n", sqrt(find_dist(trees, N)));
        return 0;
}
```

19-2-6 Prolog — Jana Kravalová

1. Příliš těžké slepice

Navzdory názvu nebyla úložka příliš těžká, pokud by se ovšem slepice spokojily s kvadratickým řešením. Snad každého napadlo vzít seznam, uškubnout mu hlavu, dojet na konec seznamu, uškubnout poslední prvek a takhle pokračovat, dokud bychom nezískali prostřední prvek, případně dvojici prvků, v závislosti na tom, zda byl vstupní seznam sudé nebo liché délky. Slepicím se takové řešení moc nelíbilo, možná také proto, že nerady slyší zmínku o škubání.

Zkusme tedy použít trik. Vyšleme v řadě slepic dva signály – jeden pomalý, jeden dvakrát rychlejší. Jakmile dojede rychlý signál na konec, pomalý bude právě uprostřed. V Prologu toto realizujeme tak, že si na vstup dáme tentýž seznam dvakrát a v každém kroku utrhneme v prvním seznamu jenom hlavu, v druhém seznamu první i druhý prvek zároveň.

2. Permutující slepice

Ukázalo se, že tato úloha byla poměrně obtížná. Problém byl hlavně s manipulací se seznamy, většině řešitelů se nepovedlo vytvořit požadovaný seznam permutací.

Jak na to: Ze vstupního seznamu postupně vyberu každý prvek X (tedy jakýsi cyklus for přes všechny prvky seznamu) a vytvořím seznam bez tohoto prvku. Tento seznam nechám rekurzivně zpracovat a dostanu seznam všech možných permutací, jen bez prvku X, načež prvek X předřadím jako hlavu před každou z těchto permutací. Když toto provedu se všemi prvky v zadaném seznamu, vygeneruji všechny permutace.

Jako další argument si musím předávat seznam již vytvořených, hotových permutací, abychom je nezapomněli (neb Prolog nemá globální proměnné) a kdykoliv vytvořím novou permutaci, vložit ji do tohoto seznamu.

Nápad se lehce řekne, ale hůř napíše. Prohlédněte si tedy připojený program.

Nakonec dodáme, že se opravovatelé příliš nešťourali v syntaktických detailech a okrajových případech této úlohy a body se udělovaly i za přibližné řešení.

3. Palindromické slepice

Tato úložka byla opět jednoduchá pro toho, kdo se rozhodl pro jednoduché kvadratické řešení. Snadno vidíme, že stačí vzít vždy první a poslední prvek seznamu, porovnat je a takto pokračovat, až dojedeme doprostřed seznamu.

Rychlejšího, lineárního řešení dosáhneme tak, že seznam otočíme a potom porovnáme původní vstupní seznam s otočeným seznamem. Pokud se shodují, byl vstupní seznam palindromem.

Jak ale otočíme seznam v lineárním čase? To dokážeme pomocí známého triku – použitím tzv. akumulátoru. Princip je velmi jednoduchý – vezmeme si původní seznam, z něj budeme trhat prvky a předřaďovat je jako hlavu do druhého seznamu. Až nám dojdou prvky v původním seznamu, v druhém seznamu (akumulátoru) budeme mít otočený původní seznam.

```
% KSP 19-1-6 1.Příliš těžké slepice

% prostredni(Sezn, Prost)       Prost je prostřední prvek Sezn

% Vyšleme seznamem dva signály, jeden dvakrát rychlejší nez druhý.
% Až dojede rychlejší signál na konec, pomalejší ukazuje na
% prostředek seznamu.

prostredni([A], [C|T2], C).            % seznam liché délky
                                       % rychlý signál A dojel na konec,
                                       % pomalý C je prostředek

prostredni([A,B], [C,D|T2], D).        % seznam sudé délky
                                       % rychlý signál A dojel na konec,
                                       % pomalý D je prostředek
                                       % (dle zadání je vic vpravo)

% z prvního seznamu reprezentující rychlý seznam utrhneme dva prvky
% (posuneme se o dva prvky), z druhého seznamu utrhneme jeden prvek
% (posuneme se o jeden prvek), zavoláme se rekurzivně na zbytky
% seznamů T1 a T2 a necháme si z rekurze vrátit výsledek Prost

prostredni([A,B|T1],[C|T2],Prost) :- prostredni(T1,T2,Prost).
```

```
% KSP 19-2-6 2.Permutující slepice

perm([],[[]]).
perm(S,P) :- perm([],S,[],P).

% Procyklíme přes všechny prvky v zadaném seznamu.
% Každý prvek utrhneme se seznamu,
% rekurzivně vytvoříme všechny permutace se seznamu
% bez tohoto prvku a tento prvek předřadíme
% před všechny tyto částečné permutace
perm(_,[],P,P).
perm(Sused,[X|S],P,R) :-
        spoj(Sused,S,SbezX),    % vytvoříme seznam bez daného prvku
        perm(SbezX,Y),          % z tohoto seznamu uděláme všechny permutace
        pripoj(X,Y,P,Q),        % před tyto permutace předřadíme daný prvek
        perm([X|Sused],S,Q,R).  % zavoláme se na další prvek
                                % (cyklíme seznamem dál)

% pripoj(Prvek,Sezn,MeziVysl,Vysl)

% Předřadí Prvek jako hlavu před všechny seznamy v seznamu
% seznamů Sezn (permutace bez prvku Prvek)
% Takto vytvořené permutace přidá do seznamu již
% vytvořených permutací MeziVysl a výsledek uloží do Vysl

pripoj(_,[],P,P).
pripoj(X,[Y|Ys],P,[[X|Y]|R]) :- pripoj(X,Ys,P,R).

% spoj(Sezn1,Sezn2,VyslSezn) spojí seznamy Sezn1 a Sezn2
% za sebe do VyslSezn
spoj([], Sezn2, Sezn2).
spoj([Hlava|Telo], Sezn2, [Hlava|Sezn3]) :-     % předřadíme Hlavu pred Sezn3,
        spoj(Telo,Sezn2,Sezn3).                 % který se nám vrátí z rekurze

% KSP 19-2-6 3.Palindromické slepice

% Sezn je palindromem prave tehdy,
% shoduje-li se přesně se svým obráceným seznamem
palindrom(Sezn) :- obrat(Sezn,Sezn).

% obracení uděláme známou technikou akumulátoru,
% aby bylo lineární
obrat(Sezn,Vysl) :- akumulator(Sezn,[],Vysl).

% ze zadaného seznamu trháme prvky
% a skládáme je před sebe do druhého seznamu,
% až nám dojdou prvky v prvním seznamu,
% v druhém seznamu jsou prvky v opačném pořadí
akumulator([],Sezn,Sezn).
akumulator([A|S1],S2,S3) :- akumulator(S1,[A|S2],S3).
```

19-3-1 Jezírka — Michal „vorner“ Vaner

Sice se proslýchá, že lesní duchové umí tuto úlohu řešit ještě rychleji (a to v $O(\log N)$), nám smrtelníkům bude stačit řešení lineární v čase i paměti. Jak tedy na to?

Napřed si všimněme, že ve struktuře jezírek nejsou cykly. Kdyby tam nějaký byl, dá se z něj některá hrana odebrat a tím cenu snížit. Dále si všimněme, že jednou odmítnutá cesta se již nikdy nepoužije (jednak by ji museli bobři zrekonstruovat, jednak je delší než něco co tam je místo ní).

Nyní za námi přijde předák kanců a nahlásí nám cestu mezi jezírky J a K dlouhou d. Co se může stát? Buď spojuje oblasti, mezi kterými zatím bobři běhali po souši, a v takovém případě ji s radostí přijmou, čímž celkovou délku cest zvýší o d. Druhým případem je, když se i předtím dalo z J do K dostat. Nyní tedy jsou 2 cesty mezi nimi, což je zbytečné a jednu lze nechat chátrat (samozřejmě tu nejdelší) a upravit aktuální celkovou délku (snížit o rozdíl nejdelší a nové).

Nyní, jak tedy rozhodnout? Mezi prvním a druhým případem by se dalo rozlišit pomocí DFU z kuchařky, ale protože si tím stejně časovou složitost nevylepšíme, je to zbytečná práce. Tak tedy rovnou zkusíme najít cestu z J do K. Protože nemáme cykly, existuje nejvýše jedna. Pokud taková cesta neexistuje, tak se jedná o první případ a hranu přidáme. Když cestu najdeme, tak vezmeme její nejdelší hranu (kterou můžeme zjistit při hledání) a porovnáme s novou, v případě že se nám to hodí, je vyměníme.

Nalezení cesty můžeme provést kupříkladu prohledáním do hloubky (je jednodušší na napsání).

Protože graf nemá kružnice, je to les. A les může mít maximálně $N-1$ hran a v takovém případě je souvislý (tedy je to strom). Počítání hran lze využít k určení, jestli už jsou všechna jezírka propojená.

Protože si stačí pamatovat jen aktuální cesty, tak nám stačí paměť lineární s počtem jezírek. Stejně tak, při průchodu do hloubky se každá hrana a každé jezírko navštíví nejvýše jednou a tedy i časová složitost je lineární. Pokud si vybereme správnou reprezentaci v paměti. (Ve vzorovém programu je použit spojový seznam sousedů v každém vrcholu).

```
program Jezirka;

const
  MaxJezer = 1000;                                {Víc jich v lese nebude ;-)}

type
  PSoused = ^TSoused;
  TSoused = record
    Konec, Zacatek: integer;
    Delka: integer;
```

```
    Dalsi: PSoused;
  end;

  TJezirko = PSoused;                              {Jezírko má jen sousedy}
  TJezirka = array[1..MaxJezer] of TJezirko;

var
  Jezirka: TJezirka;
  Hran, Delka: integer;
  Jezirek: integer;
  I: integer;
  J, K, D: integer;                                {Nová cesta}
  Max: PSoused;
  Zarad: boolean;

procedure Vloz(J, K, D: integer);                  {Vlož jeden směr cesty}
var
  Tmp: PSoused;
begin
  new(Tmp);
  with Tmp^ do begin
    Konec := K;
    Zacatek := J;
    Delka := D;
    Dalsi := Jezirka[J];
  end;
  Jezirka[J] := Tmp;
end;

procedure Odeber(J, K: integer);                   {Odebere cestu z J do K}
var
  Tmp, Posledni: PSoused;
begin
  Tmp := Jezirka[J];
  Posledni := nil;
  while Tmp <> nil do begin
    if (Tmp^).Konec = K then begin
      if Posledni <> nil then
        (Posledni^).Dalsi := (Tmp^).Dalsi
      else
        Jezirka[J] := (Tmp^).Dalsi;
      dispose(Tmp);
      break;
    end else
      Tmp := (Tmp^).Dalsi;
  end;
end;

{Pokusí se najít cestu a vrací nejdelší její úsek. Nevrací se zpět do Zpet}
function Cesta(Start, Cil, Zpet: integer): PSoused;
var
```

```
  Aktual, Vysledek: PSoused;
begin
  Aktual := Jezirka[Start];
  while Aktual <> nil do begin                        {Kouknout do všech}
    if (Aktual^).Konec = Cil then begin              {Jsem tam}
      Cesta := Aktual;
      exit;
    end;
    if (Aktual^).Konec <> Zpet then begin
      Vysledek := Cesta((Aktual^).Konec, Cil, Start);
      if Vysledek <> nil then begin                   {Tudy to vede}
        {Která je delší?}
        if (Vysledek^).Delka > (Aktual^).Delka then Cesta := Vysledek
        else Cesta := Aktual;
        exit;
      end;
    end;
    Aktual := (Aktual^).Dalsi;                        {Tady to nevyšlo, co jinudy?}
  end;
  Cesta := nil;                                       {Nic nevede :-(}
end;

begin
  WriteLn('Kolik jezírek?');
  ReadLn(Jezirek);
  for I := 1 to Jezirek do
    Jezirka[I] := nil;                                {Ještě nic nevede}
  Hran := 0;
  Delka := 0;
  while(true) do begin                                {Nechť orají navěky}
    WriteLn('Další cesta:');
    ReadLn(J, K, D);
    Max := Cesta(J, K, 0);                            {Je nějaká cesta?}
    if Max <> nil then begin
      if (Max^).Delka > D then begin                  {Tak, vyměnit}
        Zarad := true;
        Dec(Delka, (Max^).Delka - D);
        Odeber((Max^).Konec, (Max^).Zacatek);
        Odeber((Max^).Zacatek, (Max^).Konec);
      end else Zarad := false;                        {Ta nová se nevyplatí}
    end else begin                                    {Tudy to nejde, tu berem}
      Zarad := true;
      Inc(Hran);
      Inc(Delka, D);
    end;
    if Zarad then begin
      Vloz(J, K, D);
      Vloz(K, J, D);
    end;
    if Hran = Jezirek - 1 then Write('Jsou spojená')
    else Write('Nejsou spojená');
```

```
    WriteLn(', délka je ', Delka);
  end;
end.
```

19-3-2 Inventura ve spíži — Zbyněk Falt

Ač řešení této úlohy přišlo mnoho, bylo je možné rozdělit do pouhých tří skupin. Nejprostším řešením bylo pro každou ještě nezpracovanou potravinu projít všechny zbývající potraviny a počítat kolikrát se mezi nimi vyskytla. Časová složitost $O(N^2)$ však byla příliš vysokou daní jednoduchosti.

Druhá skupina si potraviny seřadila podle velikosti některým z rychlých třídících algoritmů a pak jedním průchodem vypsala hledané potraviny. Časová složitost se tak zlepšila na $O(N \log N)$.

A konečně poslední skupina použila hashování, s jehož pomocí získáme v průměrném případě lineární časovou složitost.

My si ukážeme řešení, které je rovněž lineární k délce vstupu, ale na rozdíl od hashování i v nejhorším případě a navíc není omezené předpokladem, že se nám čísla potravin vejdou do proměnné. Řešení využívá triviálního pozorování - desítkové číslo je posloupnost znaků 0-9. A vhodnou datovou strukturou pro vyhledávání posloupností znaků (neboli řetězců) jsou *trie* (nebo také slovníkové/prefixové stromy).

Trie je vícecestný strom, z jehož libovolného uzlu může vést až N hran. Oproti obyčejným vyhledávacím stromům, kde každá hrana odpovídá nějakému intervalu, si můžeme u trií dovolit (díky omezené abecedě) odkazovat se na hranu přímo pomocí znaků použité abecedy.

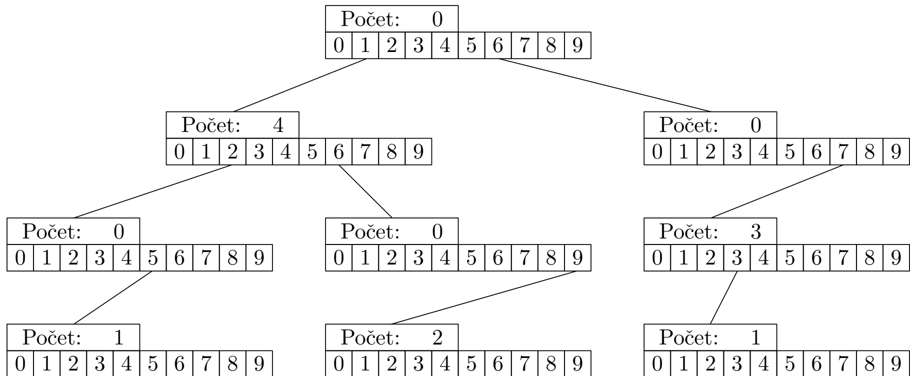
Přidání slova do trie je pak jednoduché. Postupně beru znaky slova a jdu v trii od kořene po hranách příslušných aktuálnímu znaku nebo, pokud takové hrany neexistují, tak je vytvářím.

Zjevně pak platí, že posloupnost hran z kořene do libovolného uzlu odpovídá předponě nějakého slova, které je v trii uloženo, což nám umožňuje jednoduše všechna slova z trie vypsat.

Pokud tedy v tomto případě zvolíme za N číslo 10, tj. počet desítkových číslic, a jako slova slovníku použijeme čísla potravin, můžeme v čase lineárním vzhledem k délce čísla potraviny zjistit, jestli se taková již vyskytla, resp. ji můžeme v lineárním čase do trie přidat. Abychom mohli počítat, kolikrát se jednotlivé potraviny ve spíži vyskytují, přidáme si do každého uzlu proměnnou, ve které si budeme pamatovat, kolik čísel potravin v tomto uzlu končí.

Hledané potraviny pak najdeme jednoduchým průchodem trií do hloubky. Časová složitost celého algoritmu je $O(N)$, paměťová je rovněž $O(N)$.

Pro lepší představu, jak takové trie vypadají, se můžete podívat na obrázek, kde je zachycená pro vstup 1, 125, 1, 169, 67, 1, 67, 673, 169, 67, 1.




```
program Inventura;

type
  ptUZEL = ^tUZEL;
  tUZEL = record
    pocet : integer;
    hrany : array[0..9] of ptUZEL;
  end;

  tZASOBNIK = record
    uzel : ptUZEL;
    idx : integer;
  end;

var
  i,k : integer;
  koren : tUZEL;
  uzel : ptUZEL;
  c : char;

  zas : array[0..100] of tZASOBNIK;
  vrchol : integer;

begin
  for i:=0 to 9 do
    koren.hrany[i]:=nil;
  koren.pocet:=0;

  readln(k);

  uzel:=@koren;
  while not eoln do begin                         { Přidávám čísla do trie }
    read(c);
```

```
    if c=',' then begin                         { Čárka odděluje dvě čísla }
      inc(uzel^.pocet);
      uzel:=@koren;
    end else if c in ['0'..'9'] then begin
      if uzel^.hrany[ord(c)-ord('0')]<>nil then { Buď jdu po hranách }
        uzel:=uzel^.hrany[ord(c)-ord('0')]
      else begin                                { nebo je vytvářím }
        new(uzel^.hrany[ord(c)-ord('0')]);
        uzel:=uzel^.hrany[ord(c)-ord('0')];
        for i:=0 to 9 do
          uzel^.hrany[i]:=nil;
        uzel^.pocet:=0;
      end;
    end;
  end;
  inc(uzel^.pocet);                             { Ošetřím poslední slovo }

  vrchol:=0;
  zas[0].uzel:=@koren;
  zas[0].idx:=-1;

  while vrchol>=0 do begin                      { Procházím trií do hloubky }
    inc(zas[vrchol].idx);
    while (zas[vrchol].idx<=9) and
          (zas[vrchol].uzel^.hrany[zas[vrchol].idx]=nil) do
      inc(zas[vrchol].idx);

    if zas[vrchol].idx<=9 then begin
      zas[vrchol+1].idx:=-1;
      zas[vrchol+1].uzel:=zas[vrchol].uzel^.hrany[zas[vrchol].idx];
      inc(vrchol);
    end else begin
      if zas[vrchol].uzel^.pocet=k then begin
        for i:=0 to vrchol-1 do                 { A vypisuji hledané potraviny }
          write(chr(zas[i].idx+ord('0')));
        writeln;
      end;
      dispose(zas[vrchol].uzel);
      dec(vrchol);
    end;
  end;
end.
```

19-3-3 Nevěrné ženy — Martin „Bobřík“ Kruliš

Většina z vás vyřešila tuto úlohu perfektně. Pokud nemáte plný počet bodů, bylo to zřejmě proto, že jste dostatečně neodůvodnili vaše řešení. Nebylo důležité napsat formální důkaz správnosti, ale rozumně vysvětlit, proč je vaše řešení správné. Důležitý byl zejména zobecňující krok, kterým jste ukázali, že vaše tvrzení platí pro obecně k manželek. Tzn. nestačilo pouze říci, že když to platí pro čísla 1, 2 a 3, tak to určitě platí i pro k. Nebudu vás déle napínat a ukážu vám, jak mělo řešení vypadat.

Budeme krůček po krůčku rozebírat možnosti, kolik mohlo být nevěrných žen, až se dostaneme ke kýženému číslu. Nultého dne se na večírku dozvídáme (patrně pravdivou informaci), že alespoň jedna žena je nevěrná. Tak se zamysleme, jak by vypadalo chování mafiánů, kdyby byla právě jedna žena nevěrná. Všichni mafiáni, až na jejího manžela, by věděli, která to je. Manžel této ženy by nevěděl o žádné jiné nevěrné ženě (všechny ostatní jsou věrné), a tak by mu došlo, že nevěrná musí být jeho žena. Tím pádem by ji zabil hned následujícího dne (tzn. k vraždě by došlo již první den).

Zkusme se podívat o krok dál – kdyby byly právě dvě ženy nevěrné. Manželé obou žen (označme je A, B) jsou první den v klidu. Oba totiž znají jednu ženu, která podvádí manžela (A ví nevěře manželky B a naopak). Jenže když A zjistí, že první den B svoji ženu nezabil, dojde mu, že B zná ještě jednu nevěrnici a protože A zná jen jednu, je jasné, že to musí být jeho žena. Oba tedy zabijí své manželky druhý den.

Analogicky můžeme tento postup iterovat. Obecně můžeme říci, že pokud je právě k žen nevěrných, dovtípí se to jejich manželé právě za k dní. Každý podváděný manžel ví o $k-1$ jiných nevěrnicích, a tak teprve když se $k-1$ dne dozví, že ženy jsou stále naživu, nezbývá jiná možnost, než že nevěrnic je o jednu víc (a ta jedna nemůže být žádná jiná, než jeho žena).

Nebylo to zase tak těžké, ne? Ale kdyby se i přesto našel někdo, kdo se do předchozího vysvětlení zamotal, nechť se ozve na diskusním fóru a já se pokusím ho rozmotat.

A rada na závěr: Nepodvádějte (ať jste jakéhokoli pohlaví) a když už musíte, dejte si pozor na mafiány!

19-3-4 Nejbližší rostoucí posloupnost — Petr Škoda

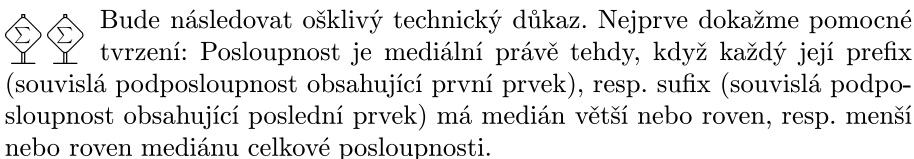
„Posloupnost čísel? Ty se přeci řeší pomocí dynamiky.“ Zdá se, že dynamické programování by měl být ten správný prostředek na vyřešení této úlohy. Když se ale na problém podíváme ze správného úhlu, zjistíme, že ho dokážeme vyřešit ještě rychleji. Popíši zde řešení, které nám přišlo od Marka Nečady.

Nejprve si zadání trochu upravíme. Od i-tého prvku vstupní posloupnosti odečteme i. Nyní hledáme nejbližší posloupnost k takto upravené posloupnosti, která je neklesající. Dva sousední prvky se tedy již mohou rovnat.

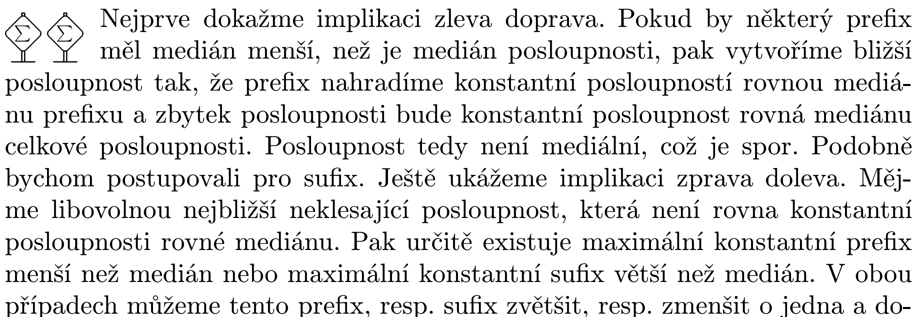
Speciální posloupností je nerostoucí posloupnost, kde je každý prvek menší nebo roven předchozímu. Všimněme si, že nejbližší neklesající posloupnost k nerostoucí posloupnosti je konstantní posloupnost. Všechny prvky konstantní posloupnosti se rovnají jednomu číslu, které bude v případě nejbližší konstantní posloupnosti medián posloupnosti (medián je takové číslo m posloupnosti, že maximálně polovina čísel posloupnosti je ostře větší než m a maximálně polovina čísel je ostře menší než m). Proč je to právě medián?

Ukážeme, že nejbližší konstantní posloupnost k libovolné posloupnosti je složena z mediánu posloupnosti. Nechť je nejbližší konstantní posloupnost složena z čísel a. Rozdělme si posloupnost na čísla větší než a, rovna a a menší než a. Jejich počty označme po řadě a_+, $a_=$ a a_-. Označme Δ_a vzdálenost této posloupnosti od upravené posloupnosti. Pokud je a_+ větší než polovina délky posloupnosti, pak konstantní posloupnost složená z čísel $a + 1$ má vzdálenost $\Delta_{a+1} = \Delta_a - a_+ + a_= + a_- < \Delta_a$. Podobně, pokud a_- je větší než polovina délky posloupnosti, pak konstantní posloupnost složená z čísel $a - 1$ má menší vzdálenost. Proto a_+ a a_- jsou menší nebo rovny polovině délky posloupnosti a tedy a je medián.

To nás vede k definici tzv. *mediální* posloupnosti. Mediální posloupnost je taková posloupnost, že k ní nejbližší neklesající posloupnost je konstantní posloupnost rovna jejímu mediánu. Již jsme ukázali, že nerostoucí posloupnost je mediální. Naším cílem je ukázat, že posloupnost skládající se ze dvou po sobě jdoucích mediálních posloupností je také mediální posloupnost, pokud medián první posloupnosti je větší nebo roven mediánu druhé posloupnosti.

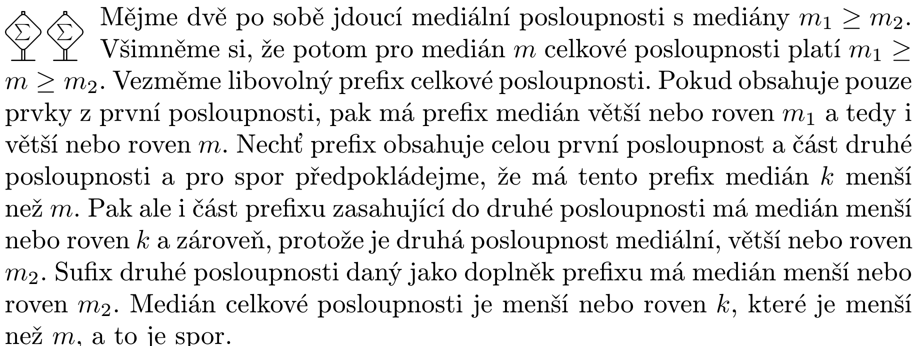


Bude následovat ošklivý technický důkaz. Nejprve dokažme pomocné tvrzení: Posloupnost je mediální právě tehdy, když každý její prefix (souvislá podposloupnost obsahující první prvek), resp. sufix (souvislá podposloupnost obsahující poslední prvek) má medián větší nebo roven, resp. menší nebo roven mediánu celkové posloupnosti.



Nejprve dokažme implikaci zleva doprava. Pokud by některý prefix měl medián menší, než je medián posloupnosti, pak vytvoříme bližší posloupnost tak, že prefix nahradíme konstantní posloupností rovnou mediánu prefixu a zbytek posloupnosti bude konstantní posloupnost rovná mediánu celkové posloupnosti. Posloupnost tedy není mediální, což je spor. Podobně bychom postupovali pro sufix. Ještě ukážeme implikaci zprava doleva. Mějme libovolnou nejbližší neklesající posloupnost, která není rovna konstantní posloupnosti rovné mediánu. Pak určitě existuje maximální konstantní prefix menší než medián nebo maximální konstantní sufix větší než medián. V obou případech můžeme tento prefix, resp. sufix zvětšit, resp. zmenšit o jedna a do-

stat tak bližší posloupnost. Tím je tvrzení dokázáno.



Mějme dvě po sobě jdoucí mediální posloupnosti s mediány $m_1 \geq m_2$. Všimněme si, že potom pro medián m celkové posloupnosti platí $m_1 \geq m \geq m_2$. Vezměme libovolný prefix celkové posloupnosti. Pokud obsahuje pouze prvky z první posloupnosti, pak má prefix medián větší nebo roven m_1 a tedy i větší nebo roven m. Nechť prefix obsahuje celou první posloupnost a část druhé posloupnosti a pro spor předpokládejme, že má tento prefix medián k menší než m. Pak ale i část prefixu zasahující do druhé posloupnosti má medián menší nebo roven k a zároveň, protože je druhá posloupnost mediální, větší nebo roven m_2. Sufix druhé posloupnosti daný jako doplněk prefixu má medián menší nebo roven m_2. Medián celkové posloupnosti je menší nebo roven k, které je menší než m, a to je spor.

Po dlouhé odbočce se vraťme zpět k algoritmu. Ten bude nyní přímočarý. Rozdělme si upravenou posloupnost na souvislé nerostoucí podposloupnosti. Nyní máme posloupnost složenou z mediálních podposloupností. Potřebujeme rychle zjistit medián mediální posloupnosti a medián spojení dvou mediálních posloupností. Protože mediální posloupnost nahradíme vždy konstantní posloupností, nepotřebujeme si pamatovat pořadí jednotlivých prvků a můžeme si je setřídit podle velikosti. Tak dokážeme rychle najít medián a spojení dvou mediálních podposloupností přechází na slití dvou setříděných posloupností, což dokážeme v čase lineárním s délkou podposloupností. Algoritmus tedy vezme dvě po sobě jdoucí mediální posloupnosti, kde první má medián větší nebo roven mediánu druhé posloupnosti, a tyto dvě posloupnosti sloučí do jedné. Protože je posloupnost konečná, algoritmus skončí a dostaneme posloupnost mediálních podposloupností s rostoucími mediány. Každou z nich nahradíme nejbližší konstantní posloupností a provedeme zpětný převod na ostře rostoucí posloupnost.

Každým spojením se nám sníží počet mediálních podposloupností o jedna. Na začátku jich je maximálně n, kde n je délka vstupní posloupnosti. Každé spojení je slití dvou setříděných posloupností, které trvá $O(n)$ a proto celková časová složitost je $O(n^2)$. Paměťová složitost je $O(n)$.

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

#define MAXN 50

int n;
int seq[MAXN];
int next[MAXN];
int med[MAXN];

int b[MAXN];
```

```
void merge(int f, int s, int e) {
  int i = f, j = s, k;

  k = f;
  while (i < s && j < e) {
    if (seq[j] > seq[i]) b[k++] = seq[j++];
    else b[k++] = seq[i++];
  }
  while (i < s) b[k++] = seq[i++];
  while (j < e) b[k++] = seq[j++];
  for (k=f; k<e; k++)
    seq[k] = b[k];
}

int main() {
  int i, a, done;

  scanf("%d", &n);
  for (i=0; i<n; i++) {
    scanf("%d", &seq[i]);
    seq[i] -= i;
  }

  a = 0;
  i = 1;
  while (a < n) {
    while (i < n && seq[i] <= seq[i-1]) i++;
    next[a] = i;
    med[a] = seq[(a+i)/2];
    a = i++;
  }

  done = 0;
  while (!done) {
    done = 1;
    i = 0;
    while (next[i] < n) {
      if (med[i] > med[next[i]]) {
        merge(i, next[i], next[next[i]]);
        next[i] = next[next[i]];
        med[i] = seq[(i+next[i])/2];
        done = 0;
      }
      i = next[i];
    }
  }

  a = 0;
  while (a < n) {
    for (i = a; i<next[a]; i++)
      printf("%d ", med[a]+i);
```

```
    a = next[a];
  }
  printf("\n");
  return 0;
}
```

19-3-5 Pevné vztahy — Tereza Klimošová

Ačkoli se v zadání slovo graf nevyskytovalo, úloha byla, jak mnozí správně poznamenali, v podstatě grafová. Mafiáni představovali vrcholy a vztahy mezi nimi byly hrany grafu. Cílem bylo ověřovat, zda zadané vrcholy (známí určitého mafiána) tvoří úplný indukovaný podgraf, neboli kliku, ve stávajícím grafu mafie.

Toto pozorování ovšem při řešení úlohy nebylo důležité, a proto se neděste, neznáte-li některé výše uvedené pojmy, a bez obav čtěte dál, už je nebudu používat.

Úloha měla svůdně jednoduché řešení, na které přišla většina z vás. Stačilo načítat vztahy například do "matice známostí" (v grafové terminologii zvané matice sousednosti). Je to tabulka (tedy dvourozměrné pole), která má v i-tém řádku a j-tém sloupci 1, pokud se i-tý a j-tý mafián znají, 0 pokud se neznají.

Podle takové tabulky umíme snadno zkontrolovat, zda se nějací mafiáni znají. Jelikož naším cílem je zjistit, zda se navzájem znají všichni mafiáni, s nimiž se seznamuje nově příchozí (označme jejich počet k), rozhodně se nespleteme, pokud ověříme vztah mezi všemi dvojicemi mafiánů, které nově příchozí poznává. Těch bude $\frac{k*(k-1)}{2}$, což asymptoticky odpovídá k^2. Jednu dvojici stihneme zkontrolovat v konstantním čase, ověření všech známých nově příchozího zvládneme v kvadratickém čase. Paměti spotřebujeme také kvadraticky, jelikož si pamatujeme matici známostí.

Toto řešení se dá vylepšit. Když si totiž uvědomíme, že ověřujeme-li pevnost vztahů pro některého mafiána, víme, že všichni předchozí, tudíž i všichni mafiáni, s nimiž navazuje vztahy, jsou pevnými články. Označme zkoumaného mafiána M_x a jeho známé M_1 až M_k v pořadí, v jakém se přidali k mafii. Mafián M_k je tedy "služebně nejmladší" ze známých M_x a víme o něm, že je pevným článkem, tedy zná-li se mafián M_k se všemi mafiány $M_1, \ldots, M_{k-1}$, určitě se znají libovolní dva z nich (jinak by M_k nebyl pevným článkem). V takovém případě se vzájemně znají všichni $M_1, \ldots, M_k$, a M_x je tedy podle naší definice pevným článkem. Naopak, pokud M_k nezná některého z $M_1, \ldots, M_{k-1}$, existují dva známí M_x, kteří se neznají, a on je tudíž slabým článkem.

Abychom zjistili, zda je M_x pevným článkem, stačí ověřit, jestli M_k zná všechny $M_1, \ldots, M_{k-1}$, stejným způsobem, jako v minulém řešení. Jelikož nám tentokrát stačí zkontrolovat pouze k vztahů, přijmout jednoho mafiána dokážeme v lineárním čase. Paměti budeme opět potřebovat kvadraticky, jelikož si

opět pamatujeme matici známostí. To bychom mohli vylepšit tak, že nahradíme tabulku polem spojových seznamů, kde bude pro každého mafiána uložen seznam jeho známých. Tím bychom docílili lineární paměťové složitosti v závislosti na počtu vztahů.

Tolik moje řešení. A teď ta vaše ... většinou jste přišli na tu jednodušší variantu, ačkoli jste se dost rozcházeli v názorech na její časovou složitost - jedni psali $O(N^3)$, druzí $O(N^2)$, což je obojí správně, ale mnozí nenapsali, co za tu dobu udělají - jestli prověří jednoho mafiána, nebo všechny. Pokud taková řešení neměla nějaké evidentní nedostatky (například absolutní nedostatek zdrojového kódu), byla oceněna pěti body. Za nedostatečné zdůvodnění správnosti jsem strhla body pouze u rychlejší varianty řešení.

```
program mafiani;

const N=42;                             {maximální velikost mafie}

var
 i,j,m:integer;
 pevnost,zna:boolean;
 mafie:array [2..N,1..N] of boolean;   {matice vztahů}
 vstup:text;

begin
  assign(vstup,'vstup.txt');
  reset(vstup);
  for i:=2 to N do
    for j:=1 to N do
      mafie[i,j]:=FALSE;
  pevnost:=TRUE;
  i:=2;
  while pevnost and not EOF(vstup) do begin
    repeat                              {pro každého mafiána načteme jeho známé}
      read(vstup,m);
      mafie[i,m]:=TRUE;
    until EOLN(vstup);
    if (m<>1)then begin
      zna:=TRUE;
      for j:=1 to m-1 do                {kontrola pevnosti vztahů nového mafiána}
        if mafie[i,j] and not mafie[m,j] then
          begin
           zna:=FALSE;
           break;
          end;
    end;
    if (m<>1) and not zna then pevnost:=FALSE;
    i:=i+1;
  end;
  if not pevnost then writeln(i-1,'-tý mafián je slabým článkem.')
  else writeln('Mafie nemá slabý článek.');
```

```
  readln;
  close(vstup);
end.
```

19-3-6 Prolog — Jana Kravalová

1. Kozel zahradníkem

Pojďme se podívat, jakým způsobem můžeme osázet našich N záhonků. Označme si počet možností osázení N záhonků jako p_N. Stojíme tedy na zahrádce a přemýšlíme, co vysadíme na první záhonek. Když vysadíme na první záhonek mrkev, můžeme si na zbylý druhý, třetí a všechny zbylé vysadit libovolné plodiny. Vysazením mrkve ztratíme jeden záhonek a na zbylém se problém s počtem kombinací opakuje, můžeme si tedy poznačit, že vysazením mrkve způsobíme, že budeme mít p_{N-1} kombinací výsadby. A co když vysadíme na první záhonek petržel? Potom na druhý záhonek musíme zákonitě vysadit mrkev, takže přijdeme o 2 záhonky a na zbylých $N-2$ záhoncích se problém opakuje. Pro N záhonků tedy dostáváme vzoreček $p_N = p_{N-1} + p_{N-2}$. Startovní podmínky si spočítáme ručně a máme krásnou rovnici přímo si říkající o rekurzivní řešení.

Mimochodem, tato posloupnost je ve skutečnosti veleznámá Fibonacciho posloupnost. Tato posloupnost je v matematice jedna z nejzajímavějších a existuje spousta jevů, které se takto chovají. Tak například si představte, že stoupáte na schodišti, které má N schodů a smíte udělat buď krok na následující schod, nebo krok ob dva schody. Počet možných výstupů na schodiště je (překvapivě) N-té Fibonacciho číslo. Úplně původní je úloha Fibonacciho králíci: Máte dva nově narozené králíky, samečka a samičku. Králičí samička má od věku 1 měsíce jeden pár malých králíčat (samečka a samičku) každý měsíc a nikdy neumírá. Mezitím samozřejmě dorůstají další páry a od věku jednoho měsíce mají další pár králíků, a tak dál. Kolik králičích párů budete mít za rok? Pokud vás zajímají další úlohy na Fibonacciho čísla, nebo jak třeba Fibonacciho čísla souvisí s tzv. zlatým řezem, podívejte se na stránky

http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html
a *http://en.wikipedia.org/wiki/Fibonacci_number*

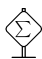
Ale vraťme se zpět k výpočtu N-tého Fibonacciho čísla. Jako první nápad jsme měli počítat je přímo podle definice, tedy

```
fib(N,F) :- N1 is N-1, fib(N1,F1),
            N2 is N-2, fib(N2,F2),
            F is F1 + F2.
```

To je samozřejmě strašlivě pomalé, protože počítáme stále dokola čísla, která už jsme dávno spočítali.

Budeme si tedy čísla budovat odspodu. Začneme se startovními hodnotami 0 a 1 a budeme vždy postupně sčítat poslední dvě čísla, přesně tak, jako bychom

vytvářeli řadu na papíře, kdybychom si ji sami počítali. Důležité je pozorování, že v každém okamžiku si stačí pamatovat poslední dvě čísla $F1$, $F2$, ta sečíst $F3$ `is` $F1+F2$ a do dalšího kroku si zapamatovat $F2$ a $F3$. Kroky opakujeme, dokud nenasčítáme N-té číslo. Vidíme, že tento postup je lineární vzhledem k N.



Spočítat N-té Fibonacciho číslo jde i v logaritmickém čase pomocí násobení matic. Programovat tento výpočet v Prologu je odvážný počin, který jsme ohodnotili bonusovými body.

2. Stromeček

Mírně přeformulujeme zadání úlohy: V zadaném setříděném vstupním seznamu máme najít prostřední prvek, ten dát do kořene vytvářeného binárního vyváženého stromu, do levého podstromu dát prvky, které byly vlevo od prostředního prvku (tedy první polovinu seznamu) a do pravého podstromu dát prvky, které byly vpravo od prostředního prvku (tedy pravou polovinu seznamu). No a protože levý a pravý podstrom jsou také binární vyvážené stromy, můžeme se na levou a pravou půlku pustit rekurzivně. Rekurze nám vrátí vyrobený správný levý a pravý podstrom, které připojíme pod kořen. Protože seznam byl už na začátku setříděný, zachováme i vlastnost binárního vyhledávacího stromu, tedy prvky vlevo od kořene budou menší a prvky vpravo od kořene budou větší. Podle tohoto návodu je snadné napsat jednoduchý program v Prologu. Měli bychom si ale uvědomit, jak dlouho tento postup bude trvat.

Na každé úrovni musíme vždy hledat prostřední prvek seznamu a seznam dělit na levou a pravou část - to trvá lineárně vzhledem k délce seznamu. A kolik těch úrovní je? Tolik, kolikrát můžu vydělit délku seznamu číslem 2, což je matematicky vyjádřeno číslo $\log_2 N$. Tedy dohromady je časová složitost takového postupu $O(N \log N)$.

Jistě už tušíte, že míříme k rychlejšímu, lineárnímu řešení. Ve skutečnosti je překvapivě jednoduché.

Základem řešení bude rekurzivní procedura, která dostane seznam a k němu zadaný počet prvků K. Tato procedura bude mít za úkol ukousnout ze začátku seznamu K prvků a z nich vyrobit binární vyvážený strom, který vrátí zpátky jako výsledek. Zároveň procedura vrátí zbytek seznamu, který pro výrobu stromu nepoužila, tedy prvky za K-tým prvkem.

Seznam tedy nebudeme v průběhu výpočtu vůbec dělit na levou a pravou část. Na začátku výpočtu si spočítáme, jak dlouhý je seznam a vypočítanou délku seznamu N vydělíme dvěma $K1$ `is` $N//2$. Pak se rekurzivně pustíme „sami na sebe“, předáme si celý seznam (žádné dělení na levou a pravou část) a předáme si číslo $K1$. Jinými slovy, požádáme rekurzivní proceduru, aby si ze vstupního seznamu ukousla potřebných $K1$ prvků, z nich vyrobila vyvážený binární podstrom, ten nám vrátila a ještě nám vrátila nepoužitý zbytek seznamu. Vrácený vyrobený podstrom použijeme samozřejmě jako levý podstrom vytvá-

řeného binárního stromu. Ze zbylého seznamu, který se nám vrátil z rekurze, utrhneme hlavu a ta bude kořenem vytvářeného binárního stromu (protože $K1$ jsme zvolili jako polovinu z délky původního vstupního seznamu). Po utrhnutí hlavy nám, jak už jistě vidíte, zůstane právě pravá půlka seznamu, kterou opět rekurzivně zpracujeme s číslem $K2$ `is` $N - K1 - 1$.

```
% KSP 19-3-6 Kozel zahradníkem

fib(N,F) :- fibiter(N,0,1,F).

fibiter(0,F1,F2,F) :- F is F1 + F2.
fibiter(N,F1,F2,F) :- N1 is N - 1,
        F3 is F1 + F2,
        fibiter(N1,F2,F3,F).

% KSP 19-3-6 Stromeček

delka([],0).
delka([_|Telo],Delka) :- delka(Telo,Delka1), Delka is Delka1 + 1.

strom([],nil).
strom(Seznam,VyslStrom) :- delka(Seznam,Delka),
        strom2(Seznam,Delka,_,VyslStrom).

% strom2(Seznam, Delka, Zbytek, VyslStrom)
% Ze seznamu Seznam udělá binární vyvážený strom o délce Delka
% Použije Delka prvků od počátku seznamu a nepoužitý zbytek vrátí
% pro další použití
strom2(Seznam,0,Seznam,nil).

% Ze vstupního seznamu si uříznu první prvek (tedy hlavu),
% udělám z ní strom a nepoužitý zbytek vrátím. Hotovo.
strom2([H|Telo], 1, Telo, t(nil, H, nil)).

strom2(Seznam, Delka, Zbytek, t(LevyStrom,Stred,PravyStrom)) :-
        DelkaLevy is Delka // 2,            % Zjistím, kolik prvků má být
                                            % v levém podstromu
        DelkaPravy is Delka - DelkaLevy - 1, % Zjistím, kolik prvků má být
                                            % v pravém podstromu
        % Chci ze vstupního seznamu ukousnout DelkaLevy prvků a z nich vyrobit
        % LevyStrom, tj. levý podstrom. Vrátí se mi nepoužitý zbytek seznamu.
        strom2(Seznam, DelkaLevy, [Stred|Zbytek1], LevyStrom),

        % Vezmu nepoužitý zbytek seznamu, jeho hlavu použiju jako střed stromu,
        % z těla si ukousnu DelkaPravy prvků a z těch udělám pravý podstrom.
        % Zase vrátím nepoužitý zbytek seznamu a ten se vrátí jako nepoužitý
        % zbytek z celého predikátu.
        strom2(Zbytek1, DelkaPravy, Zbytek, PravyStrom).
```

19-4-1 Finanční toky — Pavel Čížek

De facto všichni řešitelé dokázali najít stok, pokud v daném grafu byl, nicméně je třeba si přiznat, že při použití kvadratického řešení by Přesprst sotva unikl. Otázkou jen zůstává, jak to napsat rychleji.

Otestovat, zdali je vrchol stok, dokážeme zřejmě v $O(N)$. Problém je, že při použití algoritmu, který se u každého vrcholu zeptá, zdali je stok, bude celý program potřebovat $O(N^2)$ času. Celý problém v nalezení rychlejšího řešení bude tedy v nějakém šikovném výběru kandidáta na stok.

Podívejme se tedy na vlastnosti stoku. Zřejmě v grafu může existovat jen jeden (sporem: Pokud by byly alespoň 2 stoky i a j, tak dle definice musí vést do stoku i hrana z každého jiného vrcholu, tedy i z j. Nicméně pokud j je stok, tak z něj žádná hrana nemůže vést…).

A jak kandidáta na stok nalézt? V i-tém kroku si budeme pamatovat jediného kandidáta na stok mezi vrcholy $1..i$ (začneme s 1). Označme ho k. Kandidáta pro $(i+1)$-ní krok zjistíme jednoduše. Když vede z k do $i+1$ hrana, tak zřejmě k nemůže být stok, a jelikož mezi vrcholy $1..i$ byl jediným kandidátem, tak naději na to, stát se stokem, má jen vrchol $i+1$, což bude náš nový kandidát. V opačném případě hrana z k do $i+1$ nevede, takže vrchol $i+1$ nemůže být stok, jelikož do něj nevede hrana z každého jiného vrcholu a tedy kandidát zůstává.

Nakonec jen otestujeme, zdali kandidát, kterého získáme n-tým krokem, je stok, nebo ne.

Nalezení kandidáta i jeho otestování se zřejmě stihne v čase $O(N)$ a paměťové nároky, pokud nepočítáme vstupní matici, jsou konstantní.

```
const MaxN = 1000;

var N : integer;
    {Matice[i,j] znamená že vede hrana z i do j}
    Matice : array[1..MaxN, 1..MaxN] of boolean;
    Stok : integer;

procedure NactiMatici();
var i,j : integer;
    c : integer;
begin
   readln(N);
   for i := 1 to N do
      for j := 1 to N do begin
         read(c);
         Matice[i, j] := ( c = 1 );
      end;
end;
```

```
{oveří, zda-je opravdu kandidát stokem}
function OverKandidata(Kandidat : integer) : boolean;
var i : integer;
     Stok : boolean;
begin
   Stok := true;

   for i := 1 to N do                            {vede hrana z kandidáta do ...}
      if Matice[Kandidat, i] then Stok := false;

   for i := 1 to N do                            {vede do kandidáta hrana z ...}
      if (i <> Kandidat) and not(Matice[i, Kandidat]) then Stok := false;

   OverKandidata := Stok;
end;

{najde kandidáta na stok}
function NajdiKandidata():integer;
var i,Kandidat:integer;
begin
   Kandidat := 1;

   for i := 2 to N do
      if Matice[Kandidat, i] then Kandidat := i;

   NajdiKandidata := Kandidat;
end;

begin
  NactiMatici();
  Stok := NajdiKandidata();
  if OverKandidata(Stok) then writeln('Stok je ve vrcholu ',Stok,'.')
  else writeln('Stok neexistuje.');
end.
```

19-4-2 Byrokratický aparát — Jan Bulánek

Měl jsem připraveny velmi vtipné hlášky, například že bude třeba zestátnit zemědělské podniky, aby nám pomalu se courající úřední šimlík nepošel na své dlouhotrvající cestě hlady, bohužel, nebo spíš bohudík si je musím odpustit, protože většina z vás vytvořila optimálně rychlé algoritmy. A tak vám mohu vytknout jen jedinou věc. Ale k tomu až později.

Řešení úlohy se dalo rozdělit na dvě části. V první části si stačilo povšimnout, že graf (vrcholy jsou úředníci, orientované hrany pak značí směr dokumentu) se skládá z oddělených cyklů. Tyto cykly se pak dají najít v lineárním čase. Jistě, mohli bychom na to použít DFU (viz kuchařka 16-3), ale my jsme řekli ne. Je sice fakt, že se DFU chová docela dobře, ale vaše implementace fungovaly většinou v čase $O(N \log N)$. Více už příslušná kuchařka. Správné řešení spočívalo v použití procházení do hloubky. Vždy si vezmu první nepoužitý

vrchol, označím ho jako použitý, posunu se na jeho následníka a to opakuji tak dlouho, než dojdu do již použitého vrcholu. A protože na každý vrchol sáhnu nejvýše dvakrát, poprvé, když zkoumám, jestli už byl použit, a podruhé ve chvíli, kdy je něčím následníkem a přesunu se na něj, tak mě lineární časová složitost nemine. Samozřejmě nebude problém si přitom délky jednotlivých cyklů počítat.

Druhá část je ta, u které se lámal chleba. Téměř každý z vás si uvědomil, že je třeba spočítat nejmenší společný násobek délek všech cyklů. To je celkem jasně vidět z toho, že hledáme takový počet kroků, pro který bude mít každý úředník svůj dokument, tzn. každý cyklus musel být ukončen. No ale tím pádem musí být počet kroků dělitelný každou délkou cyklu a zároveň musí být nejmenší. A to je právě nejmenší společný násobek (NSN). No a jelikož $NSN(a, b) = a \cdot b/NSD(a, b)$ (důkaz si jakožto jednoduché cvičení proveďte sami) a $NSN(l_1, l_2, \ldots, l_N) = NSN(NSN(l_1, l_2, \ldots, l_{N-1}), l_N)$, tak je postup nabíledni. Postupně budeme počítat NSN prvních k čísel, z něj pak $k+1$ čísel atd. až N. Zbývá jen pořešit získávání NSD. Na to slouží Euklidův algoritmus, který si najděte např. na *http://cs.wikipedia.org/wiki/Euklidův_algoritmus*.

O složitosti této složitosti vypovídala vaše řešení. Jen velmi málo z vás ji určilo opravdu správně, takže si dovolím být poněkud obšírnější. Složitost Euklidova algoritmu je $O(\log a)$, kde a je větší z čísel. Jenže po prvním kroku hledáme $NSD(a \bmod b, b)$, takže se dá říct, že ta složitost je $O(\log b) + 1 = O(\log b)$. Při výpočtu potom počítáme vždy NSN nějakého čísla a jedné z délek cyklů. Považujme délku za menší číslo (rozmyslete si, proč si tím mohu pouze uškodit) a tím pádem jeden takový krok trvá $O(\log(\textit{délka cyklu}))$. Nyní přijde jedno malé kouzlíčko: $\log(\textit{délka cyklu}) \leq \textit{délka cyklu}$ a součet všech délek je N, a tak složitost bude nejvýše $O(N)$. To nám ale bohatě jako odhad stačí, protože ani první část netrvala kratší dobu. Paměť je samozřejmě lineární.

Několikrát se vyskytl Euklidův algoritmus, který místo modula používal mínus. To sice funguje, ale složitost se nám vyšplhá až do exponenciálních výšin. Nejprve si prohlédneme chování algoritmu pro K a 1 a s hrůzou zjistíme, že udělá K kroků. Pak stvoříme vhodný vstup, třeba prvních $\sqrt{N}$ délek bude mít velikost řádově $\sqrt{N}$ (jejich součet je tedy N), takže jejich NSN bude asi $\sqrt{N}^{\sqrt{N}}$, a vtipně je doplníme jedním cyklem délky 1 a složitost $O(\sqrt{N}^{\sqrt{N}})$ je na světě. Nicméně to neberte jako důkaz, protože to ve skutečnosti nic nedokazuje, ale spíš jako náhled na to, kam až může záměna jedné operace vést.

Samozřejmě se vyskytly i jiné přístupy, například jste si zjistili prvočinitele každé délky a jejich pronásobením získali kýžený NSN. Zkuste si dorozmyslet detaily a zároveň si spočtěte, že je to skutečně lineární.

```
#include <stdio.h>

#define MAXN 1000
```

```
int next[MAXN], was[MAXN]={0};
int cycles[MAXN]={0};
int cyc_count=0;
int N;

int nsd(int a, int b)               //Zjištění největšího společného dělitele
{                                   //pomocí Euklidova algoritmu
        while (b)
        {
                a = a%b;            //a sice může být menší než b, ale to nevadí
                if (a) return b;
                b = b%a;
        }
        return a;

}

int main()
{
        scanf("%d", &N);            //načtení vstupu
        for (int i=0; i<N; ++i)
                scanf("%d", &next[i]);

        int cur, cyc_len;           //zjištění délek jednotlivých cyklů
        for (int i=0; i<N; ++i)
        {
                if (!was[i]){       //Pokud vrchol ještě nenavštívil
                        cur = i;    //zahájí v něm procházení celého cyklu
                        while(!was[cur])
                        {
                                ++cycles[cyc_count];
                                was[cur] = 1;
                                cur      = next[cur]-1;
                        }
                        ++cyc_count;
                }
        }

        int nsn = cycles[0];
        for (int i=1; i<cyc_count; ++i) //z NSN K prvků spočte NSN K+1 prvků
        {
                nsn = nsn/nsd(nsn, cycles[i]) * cycles[i];
        }
        printf("A výsledek je :%d", nsn);
        return 0;
}
```

19-4-3 Naskakování na vlak — Zbyněk Falt

Hned na začátek si neodpustím jednu poznámku: ve všech algoritmech budeme zkoumat pouze složitost, se kterou algoritmus řešení nalezne. Časovou složitost na jeho vypsání v odhadech počítat nebudeme. Vyniknou tak lépe rozdíly mezi jednotlivými algoritmy. Pokud by to někomu připadalo nefér, tak si může ke všem složitostem přičíst $O(v \cdot k)$, kde v je počet navzájem různých podřetězců délky k.

Nyní již k samotné úloze. Mnoho řešitelů využilo nápovědu v zadání úlohy, a tak drtivá většina řešení využívala hashování. Ale už jenom drobná hrstka objevila, že úplně přímočaré použití kuchařky k rychlému řešení nepovede.

Základní algoritmus, který se na první pohled nabízel, byl ten, že jsme postupně brali jednotlivé podřetězce délky k, ty jsme zahashovali, a pak jsme si v nějaké tabulce (po ošetření kolizí) ukládali počet výskytů jednotlivých podřetězců. Takové řešení má v průměrném případě časovou složitost $O(n \cdot k)$

Předchozí metoda měla tu nevýhodu, že jsme pro každý podřetězec museli spočítat znovu celou hashovací funkci a to zabere čas $O(k)$. Co kdybychom ale našli takovou funkci, která by dokázala využít toho, že její hodnotu známe již pro předchozí podřetězec?

$$\sum_{j=0}^{k-1} A_i[j] \cdot P^{k-j-1}$$

Zápis $A_i[j]$ je totéž co $A[i+j]$, tedy j-tý znak od i-tého znaku v řetězci a P je nějaké číslo, které je řádově tak velké jako velikost abecedy.

Pokud chceme přejít na následující podřetězec, provedeme tyto operace: celou sumu vynásobíme P, škrtneme první písmeno z předchozího slova a přičteme poslední písmeno z následujícího. Matematicky zapsáno:

$$P \cdot \sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j-1}) - A_i[0] \cdot P^k + A_i[k] =$$

$$\sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j}) - A_i[0] \cdot P^k + A_i[k] =$$

$$\sum_{j=1}^{k} A_i[j] \cdot P^{k-j} = \sum_{j=0}^{k-1} A_{i+1}[j] \cdot P^{k-j-1}$$

Takže jsme použili konstantně mnoha kroků (nezávisle na k) a získali jsme hodnotu hashovací funkce pro řetězec, který začíná na pozici $i+1$, a to je přesně to, co jsme chtěli.

Zbývá dořešit několik technických detailů. V běžných programovacích jazycích máme proměnné omezeného rozsahu, takže pro velké k nemůžeme spočítat celou sumu. Ale můžeme si pomoci. Stačí všechny operace provádět modulo nějaké prvočíslo. A jako ono prvočíslo můžeme použít třeba rovnou velikost hashovací tabulky.

Za poznámku stojí, že ono prvočíslo musí být opravdu prvočíslo, jinak bychom se dostali do problému. Odpověď na otázku „Proč?" by asi nebyla nejstručnější, zájemci si ale mohou přečíst nějaké povídání o konečných tělesech.

Jak ale takové prvočíslo najít? Dle teorie čísel je pravděpodobnost toho, že libovolné přirozené číslo n je prvočíslem, je zhruba $1/\ln n$, a ověření toho, že n je prvočíslo, lze základním algoritmem provést v čase $O(\sqrt{n})$. Takže prvočíslo větší než nějaké n lze najít v čase $O(\sqrt{n} \cdot \ln n)$, což je méně než $O(n)$. Takže problém s hledáním prvočísla mít nebudeme.

A jak to bude s paměťovou složitostí? Mnoho řešitelů si pro každou položku v hashovací tabulce pamatovalo celý podřetězec. To je ale zbytečné a paměťová složitost se tím zhorší. Stačí si přeci pamatovat pouze index, kde daný podřetězec ve vstupním řetězci začíná, což zlepší časovou složitost na $O(n)$.

Takže jsme nalezli algoritmus, který v průměrném případě poběží v čase $O(n + v \cdot k)$, kde v je opět počet různých podřetězců. V nejhorším případě pak v čase $O(n^2 \cdot k)$.

Poznámka na úplný závěr: Pokud bychom chtěli dosáhnout času $O(n+v \cdot k)$ i v nejhorším případě, mohli bychom použít sufixové stromy. Povídání o této datové struktuře a i návod jak pomocí ní vyřešit tuto úlohu lze nalézt na adrese *http://mj.ucw.cz/vyuka/ga/*.

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

typedef struct {
        const char *podret;
        int pocet;
} ZAZNAM;

int main()
{
        ZAZNAM *tabulka;
        char *retezec=NULL;
        int velikost,n;
        int c,i,k,j,pom,hash;

        velikost=n=0;
        /* Ukázka, jak načítat vstup, když předem neznáme jeho délku*/
        while ((c=getchar())!='\n') {
                if (n>=velikost) {
```

```
                velikost=(velikost)?(2*velikost):128;
                retezec=(char*)realloc((void*)retezec,velikost);
        }
        retezec[n++]=c;
}
retezec[n]=0;

/* Najdeme nejbližší prvočíslo vyšší než 2N+1 */
for (velikost=2*n+1;;velikost+=2) {
        for (i=3;i*i<=velikost && (velikost%i);i+=2);
                if (velikost%i)
                        break;
}

tabulka=(ZAZNAM*)calloc(velikost,sizeof(ZAZNAM));

scanf("%d",&k);

/* Spočítáme hash pro prvních k znaků, za P zvolíme třeba 113 */
pom=1;
for (hash=i=0;i<k;i++) {
        hash=(113*hash+retezec[i])%velikost;
        pom=(113*pom)%velikost;
}

tabulka[hash].pocet=1;
tabulka[hash].podret=retezec;

for (i=1;i<=n-k;i++) {
        /* Přepočítáme funkci pro další k-tici */
        hash=113*hash+retezec[i+k-1];
        hash-=(pom*retezec[i-1])%velikost;
        hash=(hash+velikost)%velikost;

        for (j=hash;tabulka[j].pocet &&
                memcmp(tabulka[j].podret,retezec+i,k);j=(j+1)%velikost);

        /* A vložíme do hashovací tabulky */
        tabulka[j].podret=retezec+i;
        tabulka[j].pocet++;
}

/* A nakonec vypíšeme výstup */
for (i=0;i<velikost;i++) {
        if (tabulka[i].pocet) {
                for (j=0;j<k;j++)
                        putchar(tabulka[i].podret[j]);
                printf(" %dx\n",tabulka[i].pocet);
        }
}
```

```
        free((void*)retezec);
        free((void*)tabulka);

        return 0;
}
```

19-4-4 Váhy — Petr Škoda

Pošťák by asi koukal, kdybyste se mu snažili vysvětlit, jak zvážit libovolnou celočíselnou váhu s optimální sadou závaží. Jak vypadá optimální sada závaží? Jsou to závaží $1, 3, 9, 27, \ldots$, tedy mocniny trojky. Kolik jich potřebujeme a jak jsme na to přišli? To si hned ukážeme.

Podívejme se na problém z jiného úhlu. Mějme n dané. Jaké je největší m, že s nějakou sadou o n závažích dokážeme zvážit všechny celočíselné hmotnosti $1, \ldots, m$? Představme si rovnoramenné váhy. Na jednu stranu položíme předmět neznámé hmotnosti. Nyní máme pro každé závaží právě jednu ze tří možností. Buď závaží položíme na váhu k předmětu, nebo ho položíme na druhou stranu, a nebo ho na váhu nedáme vůbec. Pro každý z n závaží jsme vybrali jednu ze tří možností, celkem lze tedy vytvořit 3^n různých rozmístění závaží. Je jasné, že můžeme zvážit maximálně tolik různých hmotností, kolik je různých rozmístění závaží. Dostáváme jednoduchý odhad $m \leq 3^n$. Ale ještě ho vylepšíme následujícími dvěma pozorováními. Pokud žádné závaží na váhu nedáme, je zřejmé, že žádnou hmotnost neodvážíme. Podobně, pokud s jedním rozmístění závaží zvážíme hmotnost $k > 0$, pak prohozením závaží z jedné strany vah na druhou dostaneme korektní rozmístění, které ale nezváží žádnou kladnou hmotnost. Proto alespoň polovina rozmístění závaží neodváží žádnou hmotnost $1, \ldots, m$. Dostáváme horní odhad $m \leq \frac{3^n-1}{2}$ a ukážeme, že s optimální sadou tohoto horního odhadu dosáhneme.

Vraťme se zpátky k sadě závaží složené z mocnin trojek. Dokážeme nyní indukcí podle počtu závaží, že s n závažími $1, 3, \ldots, 3^{n-1}$ zvážíme všechny celočíselné hmotnosti od 1 do $\frac{3^n-1}{2}$.

- Pro $n = 1$ to určitě platí, protože $\frac{3^n-1}{2} = 1$.
- Nechť tvrzení platí pro všechna $k < n$. Rozdělme vážené hmotnosti do čtyřech intervalů.
  - Hmotnosti $1, \ldots, \frac{3^{n-1}-1}{2}$ zvážíme dle indukčního předpokladu s prvními $n-1$ závažími.
  - Hmotnosti $\frac{3^{n-1}-1}{2} + 1 = 3^{n-1} - \frac{3^{n-1}-1}{2}, \ldots, 3^{n-1} - 1$ zvážíme tak, že n-té závaží dáme naproti předmětu a pomocí prvních $n-1$ závaží, které přikládáme „obráceně“, odvážíme libovolnou hmotnost z tohoto intervalu.
  - Hmotnost 3^{n-1} zvážíme pomocí n-tého závaží.

- Hmotnosti $3^{n-1}+1,\ldots,\frac{3^n-1}{2} = 3^{n-1}+\frac{3^{n-1}-1}{2}$ zvážíme tak, že n-té závaží dáme naproti předmětu a pomocí prvních $n-1$ závaží odvážíme zbytek hmotnosti.

Indukcí jsme ukázali, že s n závažími odvážíme všechny hmotnosti až do $m = \frac{3^n-1}{2}$. Jak jsme ukázali na začátku, lépe už to nejde. Nyní zbývá odpovědět na původní otázku: Kolik potřebujeme závaží, abychom odvážili všechny hmotnosti od 1 do m? Potřebujeme tolik závaží, aby $\frac{3^n-1}{2}$ bylo alespoň tak velké jako m. Matematicky to zapíšeme $n = \lceil \log_3(2m+1) \rceil$.

19-4-5 Hazardní hra — Tomáš Valla

Hazardní hry nejsou pro slabé povahy, kdo na to nemá nervy a není si jistý, že vyhraje, ať raději zkusí své štěstí v kuličkách. Většina z vás správně vymyslela algoritmus, který Přesprstovi pomůže vyhrát, protože používá nejmenší možný počet operací k vyskládání požadované částky. Pravda, byli řešitelé, v jejichž uvažování se vyskytla chyba či dodali pouze útržek zdrojáku beze slova vysvětlení, ale takových byla menšina. Jakkoli tedy většina vymyslela optimální postup, skutečně bezchybných důkazů, že je optimální, se sešlo jako šafránu. Pojďme se tedy společně zamyslet nad správným řešením tak, aby to pochopil i detektiv Přesprst.

Označme si částku k vyskládání jako M. Nejprve si uvědomíme, že pro asymptoticky optimální řešení, tj. používající asymptoticky nejmenší počet operací, bychom si vystačili pouze s operacemi $+1$ a $\times 10$. Představme si číslo M zapsané v desítkové soustavě. Takový zápis bude mít $O(\log M)$ cifer. Podíváme na hodnotu první cifry c_1 zleva a c_1-krát přičteme jedničku. Pak se posuneme na druhou cifru c_2 zleva, aktuální hodnotu vynásobíme deseti a c_2-krát přičteme jedničku, a tak dále. Zjevně po $O(\log M)$ krocích dostaneme hledané číslo, a protože nad každou cifrou čísla musíme udělat alespoň jednu operaci (násobení 10), je náš postup asymptoticky optimální.

My bychom však chtěli přesně optimální řešení. K čemu nám v tom pomohou operace $/10$ a -1? Dělení desítkou nám v ničem nepomůže; dokážeme sporem. Mějme nějakou hodnotu h a uvažme nejkratší posloupnost operací, která vyskládá h a pro spor používá dělení. V okamžiku dělení má aktuální hodnota poslední cifru, na jejíž vyrobení jsme využili jistou posloupnost ostatních operací. Vydělíme-li nyní h deseti, přijdeme o hodnotu v poslední cifře, jinými slovy, operace vedoucí k jejímu vyrobení vůbec nebylo potřeba používat. Dostali bychom tedy kratší posloupnost operací vedoucí k vyskládání h, což je spor s tím, že jsme uvažovali nejkratší takovou posloupnost.

Ukážeme nyní lepší řešení úlohy a potom o něm dokážeme, že je optimální. Problém vyřešíme rekurzivně. Pro jednociferné M si rozmyslíme, že pokud $0 \le M \le 5$, je výhodnější M-krát použít $+1$ a pro $6 \le M \le 9$ je lepší použít $+1, \times 10$ a potom $(10-M)$-krát -1.

Pro víceciferné M úlohu převedeme na úlohu s číslem o jednu cifru menším. Číslo M můžeme napsat ve tvaru $M = 10a + b$, kde b je poslední cifra napravo. Nyní použijeme stejnou myšlenku jako u jednociferného čísla. Je-li $0 \leq b \leq 5$, vyřešíme úlohu pro číslo a, pak použijeme $\times 10$ a následně b-krát $+1$. Je-li $6 \leq b \leq 9$, vyřešíme úlohu pro číslo $a + 1$, potom použijeme $\times 10$ a následně $(10 - b)$-krát -1.

Je však potřeba si rozmyslet, že uvedený algoritmus skutečně používá minimální možný počet operací. Tuto skutečnost si dokážeme napůl neformálně matematickou indukcí podle počtu cifer čísla M.

Pro jednociferné M je to zjevně optimální postup, to lze nahlédnout rozborem možností. Uvažme nyní víceciferné $M = 10a + b$, variantu $0 \leq b \leq 5$. Z indukčního předpokladu víme, že algoritmus použije pro vytvoření a minimální počet operací. Chceme-li za a přidat další cifru, nejúspornější je a vynásobit deseti a b-krát přičíst jedničku, zjevně se nevyplatí žádné odčítání ani vícenásobné násobení deseti.

Druhá varianta, $6 \leq b \leq 9$, je o něco záludnější. V nejhorším případě, pro $b = 6$, se vykoná 6 operací (přičtení, násobení a odečítání). Potíž je však v tom, že se rekurzivně voláme na vytvoření čísla $a+1$, a není jasné, co to udělá s optimálním počtem operací na vytvoření takového čísla. Tuto skutečnost mnoho řešitelů opomnělo.

Uvědomíme si, že minimální počet operací potřebných k vyrobení $a + 1$ je nejhůře o jedna větší než minimální počet operací na vyrobení a. Končilo-li a na cifru menší než 5, náš algoritmus zjevně vypíše jen o jednu operaci navíc. Byla-li poslední cifra a mezi 6 a 8, zvýšíme ji tedy o jedna, čímž díky odečítající metodě ve skutečnosti počet operací pro $a + 1$ dokonce snížíme. Poslední možnost je, že poslední cifra a byla 9. Tehdy se zvýší o jedničku následující cifra, čímž stejným argumentem (pouze o cifru dále) dostaneme, že přibude maximálně jedna operace navíc.

Tím pádem v nejhorším případě $b = 6$ se provede 7 operací, což je stejný počet, jako kdyby se místo odečítací metody použila přičítací metoda, pro $b > 6$ už si však pomůžeme.

Algoritmus stráví nad každou cifrou konstantní počet kroků a rekurze se zanoří do takové hloubky, kolik je cifer, časová i paměťová složitost tedy vyjdou $O(\log M)$. Samotný program je napsán tak, že se vícenásobné přičítání a odčítání realizuje jako jedna úroveň rekurzivního volání hlavní procedury.

```
program hra;
var N: integer;

procedure povely(c: integer);
begin
   if c = 0 then exit;
   if c mod 10 = 0 then begin
```

```
      povely(c div 10);
      write('*');
   end else if c mod 10 <= 5 then begin
      povely(c - 1);
      write('+');
   end else begin
      povely(c + 1);
      write('-');
   end
end;

begin
   read(N);
   povely(N);
   writeln;
end.
```

19-4-6 Prolog — Jana Kravalová

1. Lednice

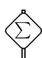
Výprava do hlubin lednice skončila úspěšně. V prográmku na pár řádek skoro nešlo udělat chybu. Bylo si jen třeba uvědomit, kam umístit řez, aby při opakováním volání nevznikaly nesmyslné počty potravin.

2. Myší spartakiáda

Myší spartakiáda byla také velmi populární. Někteří si ztížili zadání tím, že nejprve generovali celý seznam kombinací a poté jej vypisovali, ale protože úkolem je pouze vypsat všechny kombinace na výstup, můžeme to udělat takhle jednoduše: vygenerujeme celý obrazec, vypíšeme jej na obrazovku, zařízneme a přikážeme predikátu selhat pomocí `fail`. Prolog tedy snaživě zkusí vygenerovat nový obrazec, ten opět vypíšeme, selžeme, a takto nutíme Prolog hledat další obrazce, dokud neodpoví no. Mezitím jsou už ale všechny vypsány na obrazovku.

3. Myš v bludišti

Běda! O myším bludišti se ještě dlouho budou zdát noční můry nejen nešťastným programátorům, ale také zoufalým opravujícím, kteří se museli proplést spletitým bludištěm kódu. A to ještě nemluvíme o tom, že všechny myši by zešedivěly a sešly věkem, než by většina programů vydala výsledek. Přitom na první pohled vypadá úloha jednoduše, až nevinně.



Následující text obsahuje slova jako „zásobník“, „fronta“, „DFS“, „BFS“, „graf“ a podobné. Pokud nevíte, o čem je řeč, znovu vás odkazujeme na příslušnou kuchařku *http://ksp.mff.cuni.cz/tasks/19/cook3.html*.

Nejprve rozebereme chybu, kterou udělali skoro všichni. Nápadná poznámka „hledejte libovolně dlouhou cestu“ zcela oprávněně naváděla na to, k čemu je Prolog jako dělaný, k rekurzivnímu prohledávání. Správně jste si uvědomili, že si při prohledávání grafu musíte pamatovat již navštívené vrcholy. Ale málokoho napadlo, že při návratu z neúspěšné větve (tedy z takové, kde se nenašla cesta k cíli) se pilně nastřádaný seznam navštívených vrcholů opět odunifikovává a mizí. Naštěstí (pro vás) tato chyba neublíží funkčnosti, zato se ale stále dokola prochází vrcholy, ve kterých už jsme dávno byli a časová složitost roste až k exponenciální.

Jak z toho ven: Nesmíme implementovat průchod grafem až tak přímočaře, jak nás k tom svádí Prolog. Shodli jsme se na tom, že pamatovat si navštívené vrcholy je přímo životní nutnost, ale nesmíme o ně přicházet při návratu z neúspěšných větví. Tedy musíme to udělat tak, aby žádné neúspěšné větve nebyly, abychom nikdy „ne`fail`ovali“.

V predikátu cesta si budeme udržovat zásobník vrcholů a seznam navštívených vrcholů. V každém kroku vybereme vrchol z vrcholu zásobníku a najdeme všechny jeho nenavštívené sousedy. Tyto sousedy šoupneme na vrchol zásobníku a všechny označíme jako navštívené. Poté spustíme predikát cesta znovu s novým zásobníkem a novým seznamem navštívených vrcholů.

Tímto jsme se vlastně rozhodli nepoužít „vestavěnou“ prologovskou rekurzi, ale simulujeme si ji pomocí zásobníku. Proces skončí buď úspěšně, pokud se najednou na vrcholu zásobníku objeví cíl, nebo neúspěšně, pokud se zásobník vyprázdní, což by znamenalo, že jsme prozkoumali celý graf do hloubky a cestu k cíli jsme nenašli.

Navíc si stačí uvědomit, že pokud vyměníme použitý zásobník za frontu, změní se prohledávání z průchodu do hloubky na průchod do šířky a jako bonus dostaneme cestu nejkratší (za tu jsme ovšem žádné bonusové body neudělili, protože nejkratší cestu myši nepožadovaly).

Jenomže teď bychom potřebovali ještě určit, kudy cesta vede. Nemůžeme si ji budovat odzadu návratem z úspěšné větve, protože teď jsou všechny větve úspěšné a my nerozpoznáme, kdy se vracíme z úspěšné a kdy z neúspěšné větve a kdy si tedy máme cestu zapisovat. Můžeme si ale během výpočtu zapisovat pro každý vrchol jeho předchůdce a nakonec cestu odzadu zrekonstruovat.

4. Oprava

Chybu si uvědomíme, když dosadíme

```
?-minimum(1,2,2)
        Yes.
```

Co se v predikátu děje, vidíme jasně: Nejprve se pokusíme zunifikovat první řádek, což se nepodaří, skočíme tedy na druhý a ten projde. Řešení je například takovéto:

```
minimum(X,Y,Z) :- X =< Y, !, X = Z.
minimum(X,Y,Y).
```

Nedovolili jsme Prologu ihned zunifikovat poslední argument, ale nejprve jsme mu zakázali použít další větev. Teprve potom smíme zunifikovat `X = Z`.

```
% KSP 1946 1.Lednice

pocet(_, [], 0).
pocet(Jidlo, [Jidlo|Zbytek], Pocet) :- !, pocet(Jidlo, Zbytek, Pocet1),
                                        Pocet is Pocet1 + 1.
pocet(Jidlo, [_|Zbytek], Pocet) :- pocet(Jidlo, Zbytek, Pocet).


% KSP 1946 2.Myší spartakiáda
barva('c').
barva('b').

generuj(0, Sezn) :- vypis(Sezn), !, fail.
generuj(X, Sezn) :- Y is X - 1, barva(B), generuj(Y, [B|Sezn]).

vypis([]) :- nl.
vypis([H|T]) :- put(H), vypis(T).

mysi(N) :- generuj(N,[]).


% KSP 1946 3.Myší bludiště

% Reprezentace hrany faktem h(odkud,kam)
% Obousměrné hrany zapíšeme dvěma fakty
% Např. h(1,2). h(2,1).

h(1,2). h(2,1).
h(2,3). h(3,2).
h(3,1). h(1,3).
h(3,4). h(4,3).

cesta([Cil|_], Cil, [], _).

% vyber aktuální vrchol ze zásobníku
cesta([Start|Fronta], Cil, [[Start,Sousedi]|Zbytek], Navst) :-
        % najdi jeho nenavštívené sousedy
        findall(Soused, (h(Start,Soused),not(member(Soused,Navst))), Sousedi),
        % vlož sousedy na zásobník
        append(Fronta, Sousedi, NovaFronta),
        % označ sousedy jako navštívené
        append(Navst, Sousedi, NovyNavst),
        % spusť se s novým zásobníkem
        cesta(NovaFronta, Cil, Zbytek, NovyNavst).
```

```
cesta(Start, Cil,      Cesta) :- cesta([Start], Cil, Predchudci, [Start]),
        najdi_cestu(Start, Cil, Predchudci, CestaObr),
        reverse(CestaObr,Cesta).

najdi_cestu(Start, Start, _, [Start]).

najdi_cestu(Start, Cil, Predchudci, [Cil|Cesta]) :-
        najdi_predchudce(Cil, Predchudci, Predchudce),
        najdi_cestu(Start, Predchudce, Predchudci, Cesta).

najdi_predchudce(Vrchol, [[Predchudce,Naslednici]|_], Predchudce) :-
        member(Vrchol,Naslednici).
najdi_predchudce(Vrchol, [_|Zbytek], Predchudce) :-
        najdi_predchudce(Vrchol, Zbytek, Predchudce).
```

19-5-1 Útěk — Pavel Machek

Na začátek jedna metoda, jak neutíkat: Je pravda, že síť MHD připomíná graf, ale udělat co stanici, to uzel a co spoj, to hranu opravdu nejde. Pokud bychom nebyli limitováni penězi (nebo nebyli limitováni časem), Dijkstrův algoritmus by fungoval. Jenže my jsme limitováni obojím – a ani „projdeme všechny cesty“, ani „když dojdou peníze, zkusíme to jinak“ k řešení v polynomiálním čase nevedou.

Je možné stvořit jakýsi „časoprostorový“ graf, ve kterém každá zastávka bude zastoupena tolika vrcholy, kolik z ní odjíždí spojů, a hrany budou reprezentovat autobusy (s váhou ohodnocenou dolárky) a čekání (s váhou nulovou). Na časoprostorovém grafu již Dijkstrův algoritmus najde potřebné spojení, a to i v rozumném čase. (Ano, mohli bychom stvořit i „dolaroprostorový“ graf.)

Programovat Dijkstrův algoritmus se všemi optimalizacemi je ovšem dost práce, a navíc je náš časoprostorový graf poměrně speciálního tvaru. Nabízí se proto jednodušší řešení:

Události buďtež odjezdy autobusů a jejich příjezdy. Setřídíme je podle času vzestupně (umíme to v konstantním čase, protože minut ve dni je 1440). U každé linky a každé stanice si budeme pamatovat, jestli je dostupná *v tomto čase*, a případně za jakou cenu. Nedostupným linkám-stanicím můžeme prostě přiřadit nekonečno dolárků.

Výpočet bude probíhat tak, že půjdeme s časem od půlnoci dopředu, a v každém kroku zpracujeme všechny události. Pokud autobus odjíždí z dostupné stanice, zkontrolujeme, jestli cena linky náhodou není vyšší, než kdybychom zaplatili při nástupu z této stanice. Pokud je vyšší, snížíme cenu linky.

Při příjezdu zkontrolujeme, jestli cena linky plus cena, kterou musíme zaplatit za cestu, není nižší než cena stanice. Pokud ano, snížíme cenu stanice.

Ve chvíli, kdy se cena cíle stane nižší než limit v dolárcích, máme hledané spojení a můžeme ukončit výpočet. Jdeme s časem dopředu, takže spojení určitě bude nejrychlejší.

Časová složitost bude lineární s velikostí vstupu, protože událostí je tolik jako součet zastávek na všech linkách (to je velikost vstupu) a každou zvládneme zpracovat v konstantním čase. Pokud by časy byly zadány reálnými čísly, časová složitost by kvůli třídění událostí stoupne na $O(Z \log Z)$, kde Z je součet zastávek na všech linkách, tj. velikost vstupu.

```
/* S díky Romanu Smržovi*/

#include <stdlib.h>
#include <stdio.h>
#define MAX_STANIC 1000
#define MAX_ZASTAVENI 1000

struct odjezd {
        int odkud, kam;
        int prijezd;                    /* čas příjezdu do další stanice */
        int cena;                       /* cena cesty do další stanice */
        int linka;
        struct odjezd *dalsi;
};
struct prijezd {
        struct prijezd *odkud;          /* předchozí zastávka */
        int stanice;
        int cena;                       /* celková cena cesty do dané stanice */
        int linka;
        struct prijezd *dalsi;
};
int d,n,s,c;
struct odjezd *odjezdy[1440];
struct prijezd *prijezdy[1440];
/* Aktuálně nejlevnější cesta do dané stanice */
struct prijezd *cesty[MAX_STANIC];

void vypsat_cestu(struct prijezd *p) {
        if (!p) return;
        vypsat_cestu(p->odkud);
        printf("stanice %d linka %d, ", p->stanice, p->linka);
}

int main(int argc, char *argv[]) {
        int i, j, t;
        struct prijezd *p;
        struct odjezd *o;

        scanf("%d %d %d %d\n", &d, &n, &s, &c);
        for (i=0; i<n; i++) {   /* Načtení odjezdů a jejich přiřazení k časům */
                int k, mzst, mcas, mcena, szst, scas, scena, ch, cm;
                scanf("%d %d %d:%d %d", &k, &mzst, &ch, &cm, &mcena);
                mcas = 60*ch+cm;
                for (j=1; j<k; j++) {
```

```
                scanf( "%d %d:%d %d", &szst, &ch, &cm, &scena);
                scas = 60*ch+cm;

                struct odjezd *odjezd = malloc(sizeof(struct odjezd));
                odjezd->odkud = mzst; odjezd->kam = szst;
                odjezd->linka = i+1;
                odjezd->cena = scena-mcena; odjezd->prijezd = scas;
                odjezd->dalsi = odjezdy[mcas]; odjezdy[mcas] = odjezd;
                mzst = szst; mcas = scas; mcena = scena;
        }
    }

    struct prijezd start = { 0, s, 0, -1000, NULL };
    cesty[s] = &start;
    for (t = 0; t<24*60; t++) {              /* Postupný průchod časem */
        for (p = prijezdy[t]; p; p=p->dalsi)
            if (!cesty[p->stanice] ||
                cesty[p->stanice]->cena > p->cena)
                    cesty[p->stanice] = p;

        for (o = odjezdy[t]; o; o=o->dalsi)
            if (cesty[o->odkud] &&
                cesty[o->odkud]->cena + o->cena <= d) {
                    struct prijezd *prijezd =
                            malloc(sizeof(struct prijezd));
                    prijezd->odkud = cesty[o->odkud];
                    prijezd->stanice = o->kam;
                    prijezd->linka = o->linka;
                    prijezd->cena =
                            cesty[o->odkud]->cena + o->cena;
                    prijezd->dalsi = prijezdy[o->prijezd];
                    prijezdy[o->prijezd] = prijezd;
            }
        if (cesty[c]) {                  /* Nalezli jsme cestu do cíle */
            vypsat_cestu(cesty[c]);
            printf("\n");
            return 0;
        }
    }
    printf("Cesta neexistuje\n");
    return 0;
}
```

19-5-2 Nesnadné dělení — Zbyněk Falt

Většina řešitelů nenechala Isabelu s Přesprstem na holičkách, a pokud to bylo možné, tak peníze na přesné poloviny rozdělila. Ať už dynamicky s pomocí kuchařky nebo exponenciálně. Algoritmy, které nejdříve bankovky setřídily a pak se je snažily hladově rozdělit jedním průchodem na dvě části, bankovky rovněž sem tam rozdělily na dvě poloviny. Bohužel, občas jedna z nich byla větší než ta druhá :–) No, a jak mělo vypadat správné řešení?

Nejdříve si úlohu přeformulujeme trochu obecněji. Máme posloupnost čísel S (hodnot bankovek) a chceme z ní vybrat podposloupnost, která bude mít nějaký konkrétní součet s (v našem případě je s rovno polovině celkového součtu S).

Jako první každého určitě napadne jednoduchý algoritmus: Vyzkoušíme všechny možné podposloupnosti, každou zkusíme sečíst a podíváme se, jestli má „správný“ součet. Pokud má součet s, tak jsme rovnou našli jednu z vhodných podposloupností a tím i vhodné rozdělení bankovek. Pokud žádná podposloupnost nemá součet s, máme jistotu, že bankovky rozdělit nepůjdou.

Tento algoritmus má jednu vadu. Každý prvek ve vybrané podposloupnosti buď je, nebo není, takže nepotřebujeme tým matematiků, abychom viděli, že časová složitost bude exponenciální, tj. $O(2^N)$.

Teď se připravte na to, že přijde špatná zpráva: Problém výběru podposloupnosti s daným součtem je NP-úplný, takže pro obecně zadaná čísla je výše uvedený algoritmus to nejlepší, co umíme. Avšak nezoufejte – záchrana se blíží.

Náš problém naštěstí není tak úplně obecný. Využijeme faktu, že velikosti bankovek (a tedy i čísla z našich posloupností) jsou omezena nějakým relativně malým číslem M. Z toho vyplývá, že náš hledaný součet s bude nejvýše $O(M \cdot N)$, takže můžeme nasadit rafinovanou metodu, které se říká dynamické programování (viz kuchařka).

V první fázi algoritmu nejprve sečteme všechny prvky (resp. bankovky) a proměnnou s položíme rovnou polovině tohoto součtu. Pokud je součet lichý, můžeme rovnou skončit a oznámit nedočkavému uživateli, že tyto bankovky opravdu rozdělit na polovinu nejdou. Dále si připravíme pole V indexované od 1 do s, které je na počátku celé inicializované na samé nuly. Postupně si do něho budeme ukládat bankovky následujícím způsobem:

Nyní budeme brát bankovky jednu po druhé a s každou provedeme následující. Projedeme celé pole V a pokaždé, když narazíme na nenulový prvek, vezmeme jeho index i (přesně tak – index představuje vlastně celkový dosažený součet), přičteme k němu hodnotu aktuální bankovky b a na pozici s nově získaným indexem $(i + b)$ uložíme naši bankovku (třeba jako její index v posloupnosti S). Co jsme tím vlastně udělali? Nalezený nenulový index i nám říká, že existuje výběr z bankovek (z těch, které předchází b), které mají součet i. Když k nim přidáme bankovku b, tak umíme poskládat i součet $i + b$, takže si

na tento index uložíme poslední bankovku, která tento součet tvoří. Pokud je součet $i + b \geq s$, pak jej do pole neuložíme, protože už nás nezajímá (je příliš velký). Stejně tak pokud na pozici $i + b$ už je zapsaná jiná bankovka, necháme ji tam (abychom si neničili, co už máme spočítáno).

Při výše popsaném průchodu polem V musíme postupovat vždy odshora dolů, jinak by mohl nastat zajímavý efekt. Představte si, co by se stalo, kdybychom postupovali opačně a přidávali např. bankovku s hodnotou 1 do prázdného pole. Na první pozici bychom zapsali 1. Následně bychom se podívali na 1. pozici, přičetli hodnotu bankovky (tedy 1) a zapsali ji – na pozici 2. Takto bychom krásně vyplnili celé pole, tj. bankovku 1 bychom vlastně použili opakovaně.

Poslední, co zbývá popsat, je, jak ze získaných hodnot pole V zjistit rozdělení bankovek. Nejprve se podíváme na položku $V[s]$. Pokud je nulová (není tam uložena žádná bankovka), tak víme, že neumíme vybrat podposloupnost se součtem s. Dále postupujeme jednoduše. Bankovku na pozici $V[s]$ vypíšeme a podíváme se, jak složit podposloupnost se součtem $i = s - V[s]$. Ale poslední bankovka, která do takové podposloupnosti patří, se přece nachází na pozici $V[i]$. Tak ji vypíšeme a opět si položíme $i = i - V[i]$. Takto pokračujeme, dokud nevypíšeme všechny prvky (tj. i nám neklesne na nulu).

Jak již bylo naznačeno, časová složitost algoritmu bude $O(N \cdot s)$, kde N je počet bankovek a s jejich součet. Paměťová složitost je o trochu menší, a to sice $O(N + s)$. Dá se nahlédnout, že pokud bychom neměli žádné omezení na velikost součtu s, byla by časová složitost exponenciální k velikosti vstupu. Na to, abychom zakódovali číslo s, potřebujeme $B = \log_2 s$ bitů, proto by časová složitost byla $O(s) = O(2^B)$, tedy exponenciální.

Snad se vám z toho příliš nezamotala hlava, a pokud ano, tak vyražte někam ven – třeba k vodě nebo na zmrzlinu a nezapomeňte si vzít s sebou „správný“ obnos bankovek :o))

```
type TIntArray = array[1..1000000] of LongInt;
     PIntArray = ^TIntArray;

{ tahle procedůra vytvoří pole Bankovky potřebné délky a načte do něj data,
také inicializuje proměnnou N }
procedure loadData(var Bankovky: PIntArray; var N: LongInt);

{ Spočítá a vrátí celkový součet všech prvků v poli Bankovky - tj. celkovou
hodnotu všech bankovek }
function soucet(Bankovky: PIntArray; N: LongInt): LongInt;
var res: LongInt;
begin
    res := 0;
    while N > 0 do begin
        res := res + Bankovky^[N];
        dec(N);
```

```
    end;
    soucet := res;
end;

{ Hlavní algoritmus - zjistí, jak dosáhnout všech možných součtů
(naplní pole "Vybrane") }
procedure spoctiSoucty(Bankovky: PIntArray; N: LongInt;
                       Vybrane: PIntArray; Sum: LongInt);
var i, max, item: LongInt;
begin
    for i := 1 to Sum do Vybrane^[i] := 0;  { inicializujeme si pole "Vybrane" }
    max := 0;
    for item := 1 to N do begin                     { zkusíme všechny bankovky }
        { nalezli jsme bankovku,která je cennější,než půlka...máme smůlu }
        if Bankovky^[item] > Sum then Exit;
        for i := max downto 1 do     { postupně prověřuji všěchny možné součty }
            if (Vybrane^[i] > 0) and (i + Bankovky^[item] <= Sum) and
               (Vybrane^[i + Bankovky^[item]] = 0) then
                { umím součet i => umím také součet i + nová bankovka }
                Vybrane^[i + Bankovky^[item]] := item;
        if Vybrane^[ Bankovky^[item] ] = 0 then
                Vybrane^[ Bankovky^[item] ] := item;
        { upravím maximální součet, který jsem doposud viděl }
        max := max + Bankovky^[item];
        { ale stačí mi jen součty do velikosti částky, kterou hledám }
        if (max > Sum) then begin
            max := Sum;
            { pokud jsem našel způsob, jak dostat hledanou částku, končím }
            if (Vybrane^[Sum] > 0) then Exit;
        end;
    end;
end;

{ Vytahá z pole Vybrane bankovky a vypíše je na výstup }
procedure zapisVysledky(Bankovky: PIntArray; N: LongInt;
                        Vybrane: PIntArray; Sum: LongInt);
var i: LongInt;
begin
    { neexistuje způsob, jak složit součet velikosti Sum }
    if (Vybrane^[Sum] = 0) then begin
        writeln('Bankovky nelze rozdělit na polovinu.');
        Exit;
    end;
    i := Sum;                { i představuje částku, kterou je tě zbývá vypsat }
    while i > 0 do begin
        writeln(Vybrane^[i]);                      { vypíšeme vybranou bankovku }
        { zmenšíme akt. částku o právě vypsanou bankovku }
        i := i - Bankovky^[ Vybrane^[i] ];
    end;
end;
```

```
var N, Sum: LongInt;
    Bankovky, Vybrane: PIntArray;
begin
    loadData(Bankovky, N);
    Sum := soucet(Bankovky, N);
    if (Sum mod 2 = 0) then begin
   { chceme vybrat bankovky, jejich  součet je právě polovina celkového součtu }
        Sum := Sum div 2;
        GetMem(Vybrane, Sum * sizeof(LongInt));
        spoctiSoucty(Bankovky, N, Vybrane, Sum);      { hlavní algoritmus }
        { vytaháme výsledky z pole "Vybrane" a vypíšeme je }
        zapisVysledky(Bankovky, N, Vybrane, Sum);
    end
    { je-li celkový součet lichý, určitě nepůjde rozdělit na dvě stejné části }
    else writeln('Bankovky nelze rozdělit.');
end.
```

19-5-3 Hamtyhamtyhamty — Martin Mareš

Nejdříve si předvedeme jednoduchou *neprohrávající* strategii, tedy takovou, se kterou pokaždé buď vyhrajeme, nebo aspoň remízujeme. Obarvíme si $2n$ čísel, o která hrajeme, střídavě černě a bíle. Všimneme si, že pokud začínáme černým (bílým) číslem, musí soupeř vždy sebrat bílé (černé) a my dostaneme opět pozici, která má na jednom kraji černé a na druhém bílé číslo. Můžeme tedy snadno soupeře donutit k tomu, aby posbíral všechna bílá nebo všechna černá a my ta druhá. My se samozřejmě rozhodneme podle toho, která mají větší součet, a tím vyhrajeme. Pokud se nám stane, že obě skupiny čísel mají součet stejný, vede naše strategie alespoň k remíze.

Zadání úlohy po nás ovšem chce, aby naše strategie vyhrála, kdykoliv je to možné. Je to opravdu tak, že kdykoliv černobílá strategie remízuje, je remíza nevyhnutelná? Bohužel nikoliv – například pro čísla 1,2,4,2,1,2 můžeme odebráním dvojky dostat soupeře do stavu 1,2,4,2,1, ze kterého musí odebrat jedničku (teď vedeme o jedna), a pokud spustíme černobílou strategii až teď, určitě o svůj náskok nepřijdeme. Tento problém se mnozí z vás pokoušeli obejít tím, že po každém tahu testovali, jestli nezačal být součet černých a bílých různý (to v našem příkladu nepomůže, protože rozhodující je už první tah), nebo třeba při rovnosti odebírali větší číslo (tedy je o něco těžší přijít s protipříkladem, ale 4,2,1,6,8,5 zabere). Takových fíglů se dá vymyslet spousta, ale neznám žádný, který opravdu funguje.

Půjdeme na to tedy trochu jinak: Nebudeme se snažit jenom vyhrát, ale dokonce soupeře obrat o co nejvíce, zkrátka co nejvíce nahamtat – chceme tedy, aby rozdíl mezi tím, co získáme my a co získá soupeř, byl co největší. Když si označíme zadaná čísla $A_0, \ldots, A_{N-1}$, budou všechny pozice v průběhu hry vždy nějaké intervaly $A_i, \ldots, A_{i+\ell-1}$. Pro každý takový interval si spočítáme hodnotu $H_{i,\ell}$, která nám řekne, kolik je do konce hry schopen nahamtat ten,

kdo je v tomto okamžiku na tahu. (Ti zkušenější už samozřejmě správně větří dynamické programování.)

Všimneme si, že $H_{i,1} = A_i$ (pokud už je ve hře jen jediné číslo, co zbývá, než ho sebrat). Pokud je interval delší, máme dvě možnosti, jak hrát:

- Buďto sebereme A_i a soupeři předáme $A_{i+1}, \ldots, A_{i+\ell-1}$. Tehdy soupeř, pokud bude hrát nejlépe, jak může, nahamtá $H_{i+1,\ell-1}$, pročež my můžeme celkově nahamtat $A_i - H_{i+1,\ell-1}$.
- Nebo sebereme $A_{i+\ell-1}$ a předáme $A_i, \ldots, A_{i+\ell-2}$, a proto nahamtáme $A_{i+\ell-1} - H_{i,\ell-1}$.

Z těchto dvou možností si samozřejmě vybereme tu, ve které nahamtáme víc. Platí tedy:

$$H_{i,\ell} = \max(A_i - H_{i+1,\ell-1}, A_{i+\ell-1} - H_{i,\ell-1}).$$

Navíc hodnoty pro úseky délky ℓ počítáme z hodnot pro úseky délky $\ell - 1$, takže stačí postupovat indukcí od nejmenšího ℓ k největšímu. Podle $H_{0,n}$ pak okamžitě poznáme, jestli je pro nás hra vyhraná nebo prohraná, a v každém stavu hry snadno zjistíme, zda máme odebrat levé nebo pravé číslo podle toho, která z možností nám dala hodnotu $H_{i,\ell}$. Přesně na tomto principu je založen následující program, který v čase a paměti $O(n^2)$ pro všechny intervaly spočítá jak $H_{i,\ell}$, tak optimální tah $T_{i,\ell}$.

```
int N;                  // Počet čísel na vstupu
int A[MAX];             // Číslíčka, se kterými hrajeme
int H[MAX][MAX];        // Maximální zisk v každém úseku
int T[MAX][MAX];        // Vyhrávající tah pro úsek

void hamtyhamtyhamty(void)
{
  for (int i=1; i<=N; i++)      // Úseky délky 1
    H[i][1] = A[i];
  for (int l=2; l<=N; l++)      // A teď ty delší
    for (int i=1; i<=N-l+1; i++) {
        int levy = A[i] - H[i+1][l-1];
        int pravy = A[i+l-1] - H[i][l-1];
        if (levy > pravy) {
            T[i][l] = 'L';
            H[i][l] = levy;
        } else {
            T[i][l] = 'P';
            H[i][l] = pravy;
        }
     }
}
```

Ostatně, k témuž výsledku bychom se také mohli dobrat tak, že bychom si napsali rekurzivní funkci, která by počítala $H_{i,\ell}$ podle našich vzorečků. Přímočaře naprogramovaná by běžela v exponenciálním čase, ale mohli bychom si v nějakém pomocném poli pamatovat, které hodnoty jsme už spočítali, a nepočítat je znovu. Tak bychom se dostali opět na kvadratickou časovou a paměťovou složitost.

Nechci tedy nikterak podceňovat naši milou Izabelu, ale tipoval bych, že v jejím úspěchu byla přeci jen trocha začátečnického štěstí, a to jí přineslo posloupnosti, na kterých funguje černobílá strategie snadno spočítatelná zpaměti. Konec konců, není divu, náhodný vstup toto s velkou pravděpodobností splňuje. Ať tedy takové máte i vy :-)

Finta na závěr: Kdybychom se na naši úložku dívali jako na programátorskou úlohu místo původně zamýšlené jednoduché logické hádanky, mohli bychom se ještě pokusit snížit nepříjemně vysokou paměťovou náročnost. Na co tu paměť vlastně potřebujeme? Samotný výpočet hodnot hodnot $H_{i,\ell}$ si vystačí s hodnotami $H_{j,\ell-1}$ a na všechny menší délky můžeme zapomenout. Ovšem potřebujeme si zapamatovat, který tah byl ve které pozici optimální, abychom pak dokázali při libovolném vývoji hry zjistit, jak máme hrát. Jak ušetřit tady? Místo toho, abychom si pamatovali všechny stavy hry, můžeme si třeba uložit jen stavy s délkou úseku dělitelnou nějakým k, a jakmile se dostaneme do intervalu délek mezi ki a $k(i+1)$, spočítáme z uložených výsledků pro délku ki výsledky pro všechny mezilehlé délky. Tak algoritmus příliš nezpomalíme (každou pozici počítáme jen dvakrát) a paměť zredukujeme na $O(kn + (n/k)n)$, tedy při volbě $k = \sqrt{n}$ na $O(n\sqrt{n}) = O(n^{3/2})$. Mezilehlé délky ale můžeme podrozdělovat stejným způsobem a pokud do sebe zasunutých podrozdělení bude p, vyjde, že optimální je podrozdělování na $n^{1/p}$ částí, časová složitost $O(n^2 p)$ a prostorová $O(n^{1+1/p})$. Mezním případem je podrozdělení na dvě části, které nám dá hloubku $O(\log n)$, čas $O(n^2 \log n)$ a paměť $O(n \log n)$. To je poměrně univerzální trik, kterým jde u mnoha úloh ušetřit spousta paměti za cenu mírného (obvykle logaritmického) zpomalení.

19-5-4 Lodní mrazáky — Michal „vorner“ Vaner

Nejprve si všimněme, že jeden mrazák nám stačit nebude. Kdybychom použili pouze jeden, věci by se do něj nastrkaly, ty nejstarší by skončily vespod a už by se k nim nikdo nedostal.

Proto použijeme mrazáky dva, a aby se nám to nepletlo, stanovíme si jistá pravidla na uspořádání potravin.

1) V prvním mrazáku, říkejme mu třeba příchozí, budou potraviny seřazeny tak, že čím hlouběji v mrazáku budeme, tím starší budou.
2) Druhý mrazák, budiž nazýván odchozí, bude seřazen přesně naopak, tedy čím hlouběji, tím novější potraviny.

3) Libovolná potravina v odchozím mrazáku bude starší, než libovolná v příchozím.

Teď zbývá vytvořit operace ulož a sněz tak, aby žádnou z předchozích podmínek neporušily. Operace ulož je jednoduchá, prostě vložíme potravinu do příchozího mrazáku. Tím určitě neporušíme podmínku 1) - tato potravina je určitě novější, než vše, co bylo v mrazáku předtím a přišlo to nahoru, ani podmínku 2) - odchozího mrazáku se vůbec nedotkneme, ani podmínku 3), protože tato potravina je určitě novější, než cokoliv v odchozím mrazáku.

Pokud bude v odchozím mrazáku alespoň jedna potravina, je operace sněz také triviální. Potravina navrchu tohoto mrazáku je nejstarší, protože je starší než cokoliv pod ní a je starší než cokoliv v příchozím mrazáku. Tudíž ji stačí jen vzít. Toto samozřejmě neporuší žádnou z podmínek, odebráním libovolné potraviny se nic zkazit nedá.

Co ale v případě, že odchozí mrazák zeje prázdnotou? Předpokládejme, že je alespoň jedna potravina v příchozím mrazáku, protože jinak by na lodi panoval hladomor a nebylo by možné provést operaci sněz. Celkově nejstarší potravinou je tedy nejstarší potravina v příchozím mrazáku, která je dle podmínky 1) vespod. Proto přeházíme všechny potraviny z příchozího do odchozího mrazáku. Tím se otočí pořadí potravin, takže nejstarší bude navrchu a postupně budou novější a novější. Tím je splněna třetí podmínka a zařízeno, že lze provést operaci sněz tak, jak byla popsána výše. První a druhá jsou splněny také, protože příchozí mrazák je prázdný.

Nyní, kolik se provede operací, tedy jaká je časová složitost? Každou potravinu, která projde úschovou v podpalubí, čeká uložení a vyndání z každého ze dvou mrazáků. Celkem tedy 4 operace. Pro zpracování N potravin bude potřeba $4N$ operací, z nichž právě $2N$ budou pomocné. Splnili jsme tedy kapitánovu žádost, aby pomocných operací bylo $O(N)$.

Paměťovou složitost můžeme brát buď dle počtu mrazáků, které potřebujeme 2, nebo podle počtu obsazených mrazících pozic, ale protože každá potravina může být v jeden okamžik uložena nejvýše jednou, je paměťová složitost lineární.

```
unit  Podpalubi;

uses  Mrazaky;
var   Prichozi, Odchozi: Mrazak;                           { Naše dva mrazáky }


procedure uloz( Co: Potravina );
begin
        vlozDoMrazaku( Co, Prichozi );
end;
```

```
function snez: Potravina;
begin
	if prazdnyMrazak( Odchozi ) then begin
		if prazdnyMrazak( Prichozi ) then begin    { Hladomor }
			snez := HrnicekOdMedu;
			exit;
		end;
		while not prazdnyMrazak( Prichozi ) do
			vlozDoMrazaku( vyndejZMrazaku( Prichozi ), Odchozi );
	end;
	snez := vyndejZMrazaku( Odchozi );
end;
```

19-5-5 Praktická úložka – Počet inverzí   Martin „bobřík“ Kruliš

Jak už to tak v životě bývá, způsobů řešení této úlohy je více. Zde si popíšeme jeden velice jednoduchý a naznačíme některé další možné. Posaďte se, prosím, na svá místa, připoutejte se a během startu nekuřte.

Náš postup je založen na známém třídícím algoritmu merge-sort, neboli třídění pomocí slévání. Tento algoritmus pracuje na principu rozděl a panuj. Tříděnou posloupnost rozdělí na dvě poloviny (tedy podúlohy menšího rozsahu), které setřídí rekurzivním použitím stejného algoritmu. Setříděné poloviny následně slije do jedné posloupnosti.

Pro lepší pochopení našeho algoritmu si ještě raději zopakujeme průběh slévání. Řekněme, že máme dvě vzestupně setříděné posloupnosti (uložené jako pole) A a B a chceme je slít do jedné opět vzestupně setříděné posloupnosti C. Vytvoříme si indexy a, b a c, které inicializujeme tak, aby ukazovaly na první prvky jednotlivých posloupností (tj. a ukazuje na první prvek A atd.). Dokud se index a, nebo b nedostane mimo rozsah jeho posloupnosti, budeme provádět následující krok: Porovnáme prvky $A[a]$ a $B[b]$, menší z nich zkopírujeme do $C[c]$ a posuneme index v poli s menším prvkem na další prvek v posloupnosti. Rovněž posuneme c na další volné místo ve výsledné posloupnosti. Když některý z indexů (a, nebo b) dojede za konec své posloupnosti, algoritmus končí, avšak ještě je třeba dokopírovat zbývající prvky z druhé posloupnosti (z té, která ještě nebyla zpracována celá). Např. pokud a dojede za konec A, musí se ještě zpracovat zbytek posloupnosti B.

Nyní zbývá rozmyslet, jak nám tento algoritmus pomůže při počítání inverzí. Celkový počet inverzí v posloupnosti lze spočítat jako součet počtu inverzí v obou polovinách (tj. v obou menších podproblémech) plus počet inverzí, které objevíme při slévání těchto polovin. Z principu fungování algoritmu je jasné, že nám stačí počítat pouze inverze objevené sléváním (o ostatní se postará rekurze).

Máme tedy algoritmus na slévání dvou posloupností popsaný výše. Jako A si označíme první polovinu tříděné posloupnosti a jako B polovinu druhou. Pokud by bylo uspořádání správné (tj. neobsahovalo by žádné inverze), budou všechny prvky z A menší než prvky B. V každém kroku algoritmu nastává právě jedna z možností:

- $A[a] \leq B[b]$ – prvek v první posloupnosti je menší nebo roven prvku ve druhé posloupnosti, takže je vše v pořádku a žádnou inverzi jsme neobjevili.
- $A[a] > B[b]$ – prvek v první posloupnosti je větší než prvek ve druhé posloupnosti. To znamená, že $B[b]$ bude ve výsledku zařazen před všechny zbývající prvky v A, což je rozhodně porucha v uspořádání. Každý zbývající prvek v A je tím pádem v inverzi s prvkem $B[b]$, takže nám stačí přičíst k celkovému počtu inverzí počet zbývajících prvků v A.

Časová složitost tohoto algoritmu je stejná jako časová složitost merge-sortu, tzn. $O(N \cdot \log_2 N)$. Paměťová složitost je při vhodné implementaci pouze $O(N)$, neboť nám stačí jedno pole na načtené prvky a jedno pomocné pole na slévání.

Paměťové limity pro jednotlivé testy byly nastaveny tak, abyste mohli použít i staticky alokované pole (jak je to ve vzorovém programu). Na toto pohodlí si ovšem příliš nezvykejte, neboť v dalších praktických úlohách již budete muset alokovat paměť dynamicky podle toho, kolik jí skutečně budete potřebovat.

Závěrem bych ještě zmínil další možné způsoby řešení. Prvním způsobem je použít jiné třídící algoritmy místo merge-sortu. Problém je v tom, že ne každý algoritmus nám bude vyhovovat. Např. quick-sort použít nemůžeme, neboť přehazuje prvky mezi oběma polovinami tříděných dat, a tak nám může během třídění vytvářet inverze, které v původní posloupnosti nebyly. Druhou možností je použít vhodně upravené binární vyhledávací stromy, avšak detailnější popis by si vyžádal poměrně velké množství dalšího textu, a tak si jej dovolím vynechat.

return E_OK;

```
program 19-5-5;
const MAX = 100000;
type TCisla = array[1..MAX] of LongInt;


{ Hlavní funkce programu. Pracuje na principu algoritmu merge sort a přitom
  počítá inverze. Parametry: main - hlavní pole s čísly, tmp - pomocné pole,
  i,j - tříděný rozsah }
function merge(var main, tmp: TCisla; i, j: LongInt) : Integer;
var     mid, count: LongInt;
        a, b, res: LongInt;
```

```
begin
        merge := 0;
        if (i >= j) then Exit;                   { prázdný interval => končíme }

        { určíme střed tříděného intervalu a spočítáme podúlohy }
        mid := (i+j) div 2;
        count := merge(tmp, main, i, mid);
        count := merge(tmp, main, mid+1, j) + count;

        { nyní slijeme setříděné poloviny }
        a := i;  b := mid+1;                 { indexy v setříděných polovinách }
        res := i;                                    { index ve výsledném poli }
        { sléváme, dokud jeden z indexů nedojede do konce svého úseku }
        while (a <= mid) and (b <= j) do begin
                { "porucha" v uspořádání - v druhé půlce je menší prvek }
                if (tmp[a] > tmp[b]) then begin
                        { přičteme tolik inverzí, kolik prvků zbývá v 1. části }
                        count := count + (mid-a+1);
                        main[res] := tmp[b];
                        inc(b);
                end else begin
                { prvek z 1. části je menší - žádné inverze nepřičítáme }
                        main[res] := tmp[a];
                        inc(a);
                end;
                inc(res);
        end;
        { zkopírujeme zbytky slévaných částí (jsou-li nějaké) }
        while (a <= mid) do begin main[res] := tmp[a]; inc(a); inc(res); end;
        while (b <= j) do begin main[res] := tmp[b]; inc(b); inc(res);end;
        merge := count;
end;

var     F        : Text;
        N, i     : LongInt;
        cisla, cisla2 : TCisla;
        { čísla jsou alokovaná staticky pro zpřehlednění kódu }
begin
        Assign(F, 'cisla.in');                         { Načtení vstupu }
        Reset(F);
        ReadLn(F, N);                                  { počet čísel N }
        for i := 1 to N do begin
                Read(F, cisla[i]);
                cisla2[i] := cisla[i];
        end;
        Close(F);

        { spočítáme počet inverzí a rovnou jej vytiskneme na výstup }
        Write( merge(cisla, cisla2, 1, N) );
end.
```

19-5-6 Prolog — Jana Kravalová

1. Nejkratší program

Ukázalo se, že není problém vytvořit krátký program, ale vytvořit funkční program. Řešení, která cyklí při hledání neexistujícího prvku, prostě nemohou získat víc než půl bodu. Při tvoření krátkého programu jste mohli zvolit několik cest, záleželo na tom, jak moc jste chtěli používat vestavěné predikáty Prologu. Všeobecně ale platilo, že nejkratší programy bylo možné napsat na jeden predikát (na jeden řádek).

2. Fronta

Toto cvičení vás mělo navést na řešení pomocí rozdílových seznamů, což většina řešitelů pochopila. Správných řešení ale bylo málo. Prohlédněte si tedy autorské řešení.

3. Expertní systém

Protože se jedná o větší program, ve kterém je potřeba si pořádně rozmyslet datové struktury, spolupráci s uživatelem a jiné záludnosti, a protože se našlo jenom několik odvážlivců, rozhodla jsem se nešťourat do syntaktických chyb a jiných nedostatků. Pokud myšlenka vypadala alespoň trochu použitelně, pokud autor působil dojmem, že ví, co dělá, a pokud se zdálo, že by se nakonec program dal odladit, připsala jsem plný počet bodů. V této úloze nelze žádné řešení prohlásit za jediné optimální. V autorském řešení najdete jedno z nejjednodušších a nejvíce srozumitelných řešení vytvořené pomocí stromu otázek.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  KSP 19-5-6 1. Nejkratší program  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

p(S,K,X):-sort(S,[Y|T]),(Y>K,X=Y,!;p(T,K,X)).    % nejlepší řešení účastníků (Roman Smrž)

q(S,K,X):-setof(H,(member(H,S),H>K),[X|_]).      % původní autorské řešení

r(S,K,X):-sort([K|S],R),nextto(K,X,R).           % extra řešení (Milan Straka)

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  KSP 19-5-6 2. Fronta  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% zřetězení rozdílovych seznamů (potřeba pro vloz2)
zretez(A-B, B-C, A-C).

% vloz(StaraFronta, NovyPrvek, NovaFronta)
vloz(Fronta-[NovyPrvek|Y], NovyPrvek, Fronta-Y).

% kdo nechápe vloz, ať si přečte vloz2
vloz2(X-X, NovyPrvek, [NovyPrvek|X]-X) :- var(X).
vloz2(StaraFronta-X, NovyPrvek, NovaFronta) :-
        zretez(StaraFronta-X, [NovyPrvek|Y]-Y, NovaFronta).
```

```
% odeber(StaraFronta, NovaFronta, Prvek)
odeber(X-X) :- var(X), fail.
odeber([H|Zbytek]-X, Zbytek-X, H).

% je_prazdna(Fronta)
je_prazdna(X-X) :- var(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     KSP 19-5-6 3. Expertní systém     %
%  pěkné a jednoduché řešení Jana Žáka  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% databaze.pl:
q([
        q('Je to kockovita selma?', [
                q('Je to kocka?', kocka),
                q('Je to lev?', lev)
        ]),
        q('Je to psovita selma?',[
                q('Je to pes?', pes),
                q('Je to liska?', liska)
        ])
]).

% program.pl:

hadej :- write('Mysli si zvire...'), nl, q(Otazky), hadej(Otazky).
hadej([]) :- write('Takove zvire neznam').         % prošli jsme celý strom
hadej([q(Otazka,Ano)|Ne]) :-
        write(Otazka), nl, read(Odpoved),          % vypiš otázku a přečti odpověď
        (Odpoved = ano, hadej(Ano)) ;              % kladná odpověď, použij podvětve
        (Odpoved = ne, hadej(Ne)).                 % záporná odpověď, použij další větev
hadej(Zvire) :- write('Myslis si zvire '), write(Zvire).
```

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	30	224
1.	Marek Nečada	G Jihlava	3	21	190,6
2.	Vojtěch Kolář	G Neratov	3	28	189,5
3.	Jakub Kaplan	GJKTyla	3	20	187,0
4.	Jan Michelfeit	G HBrod	3	22	183,8
5.	Zbyněk Konečný	GKptJaroš	4	30	174,2
6.	Jan Žák	G HBrod	2	23	163,6
7.	Trung Ha duc	GMasaryk	1	20	161,3
8.	Pavel Klavík	G Chrudim	4	21	149,1
9.	David Brazdil	G Zlín	2	24	147,7
10.	Vlastimil Dort	GŠpitálsPH	1	21	143,5
11.	Petr Onderka	G VKlobou	4	18	143,1
12.	Petr Kratochvíl	G SvětláNS	4	21	139,6
13.	Libor Plucnar	GPBezruče	2	20	138,3
14.	Vojtěch Tůma	G Jihlava	3	16	131,6
15.	Tomáš Sýkora	G VKlobou	3	19	106,4
16.	Matěj Korvas	GJSeiferPH	4	15	99,9
17.	Pavel Veselý	G Strakon	2	17	96,7
18.	Miroslav Klimoš	G Bílovec	2	10	95,6
19.	Josef Pihera	G Strakon	4	8	82,2
20.	Lukáš Kripner	G Litvínov	1	10	79,6
21.	Karel Pajskr	GJKeplera	2	10	78,4
22.	Roman Smrž	GOhradní	3	9	75,3
23.	Roman Říha	G Prachat	3	8	73,9
24.	Radim Cajzl	G NMnMor	0	11	73,2
25.	Jiří Maršík	GJKTyla	3	10	70,3
26. – 27.	Jakub Balhar	GJNerudy	4	13	67,4
	Martin Majer	SPŠÚžlabin	2	11	67,4
28.	Tomáš Herceg	G Třebíč	4	13	67,0
29.	Jan Kohout	G Roudnice	4	14	65,0
30.	Petr Holášek	G Příbor	3	13	57,0
31.	Tomáš Jakl	G MTřebová	3	9	56,7
32.	Marika Ivanová	SPŠ Zlín	3	8	55,7
33.	Ondřej Bouda	GKptJaroš	4	8	53,2
34.	Kristýna Krejčová	G Tišnov	4	9	52,6
35.	Zbyněk Jiráček	GArabská	3	8	50,6
36.	Viktor Lucza	G Rožňava	3	8	50,4

37.	Dušan Rychnovský	G Hranice	3	7	45,0
38.	Jakub Červenka	GŠpitálsPH	1	5	43,8
39.	David Marek	SPŠ Zlín	3	6	40,4
40.	Michal Kozák	G Jihlava	3	4	39,9
41.	Rudolf Rosa	G Kladno	4	5	39,6
42.	Martin Kahoun	GJNerudy	4	6	36,5
43.	Jan Matějka	GJírovco	2	5	35,8
44.	Lucia Simanová	GGrösslin	3	6	34,9
45.	Karel Tesař	SPŠEPlzeň	1	6	34,8
46.	Stanislav Fořt	G Tábor	0	14	34,5
47.	Martin Němec	G Ledeč	3	5	31,3
48.	Richard Jedlička	G Vlašim	3	4	30,4
49.	Mirek Jarolím	GMikuláš	1	7	29,2
50.	Jakub Slavík	GJKTyla	3	4	28,8
51.	Jan Krajdl	SPŠÚžlabin	2	5	28,5
52.	Amadeo Mareš	SOŠ Blatná	2	9	27,7
53.	Václav Strnad	GArabská	3	4	27,5
54.	Hana Bendová	G ČLípa	4	3	27,0
55.	Radim Pechal	SPŠ Rožnov	4	3	26,3
56.	David Škorvaga	G Kralupy	4	3	24,0
57.	Jan Kučera	G Polička	4	5	23,0
58.	Alena Skálová	GNaVPláni	3	3	20,5
59. – 60.	Milan Klášterka	SPŠKlatovy	3	4	18,2
	Jan Sixta	G Brandýs	4	4	18,2
61.	Jakub Pavlík jn.	G Kladno	4	2	15,6
62.	Karolína Burešová	G ČLípa	0	2	13,9
63.	Tomáš Volf	G Tábor	0	5	12,4
64.	Alžběta Pechová	GPodStrání	2	3	12,1
65.	Miroslav Jančařík	G UBrod	3	3	10,5
66.	Matěj Pacovský	G Tábor	3	1	9,5
67.	Jan Papoušek	GKptJaroš	3	1	8,0
68.	Josef Sedlačík	G UBrod	3	1	6,2
69.	Martin Pástor	SPŠAlejová	2	1	6,0
70.	Matyáš Bach	G VKlobou	0	1	2,6
71. – 72.	Veronika Paštyková	G KHora	1	1	2,5
	Petr Pecha	ZŠŠkValKlo	0	1	2,5
73.	Jan Musílek	G NBydžov	3	1	2,4
74.	Radomír Švihel	G Zlín	3	1	0,0

Obsah

Úvod . . . 3
Zadání úloh . . . 5
První série . . . 5
Druhá série . . . 17
Třetí série . . . 25
Čtvrtá série . . . 35
Pátá série . . . 46
Programátorské kuchařky . . . 57
Kuchařka druhé série – Rozděl a panuj . . . 57
Kuchařka třetí série – grafy . . . 64
Kuchařka čtvrté série – hešování . . . 74
Kuchařka páté série – rekurze a dynamika . . . 79
Vzorová řešení . . . 86
První série . . . 86
Druhá série . . . 96
Třetí série . . . 112
Čtvrtá série . . . 127
Pátá série . . . 141
Pořadí řešitelů . . . 156
Obsah . . . 159

Milan Straka a kolektiv

Korespondenční seminář z programování
XIX. ročník

Autoři a opravující úloh:
Jan Bulánek, Pavel Čížek, Zbyněk Falt, Tomáš Gavenčiak,
Cyril Hrubiš, Tereza Klimošová, Jana Kravalová,
Martin Kruliš, Pavel Machek, Martin Mareš,
Milan Straka, Petr Škoda, Tomáš Valla, Michal Vaner

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 215. publikaci.

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.

S jejich pomocí ročenku vysázel Petr Kratochvíl.

Korektury provedla Jana Kravalová.

Ilustrace (včetně té na obálce) vytvořil Martin Kruliš.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 160 stran
Náklad 300 výtisků
Praha 2007

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

ISBN 978-80-7378-021-0

ISBN 978-80-7378-021-0