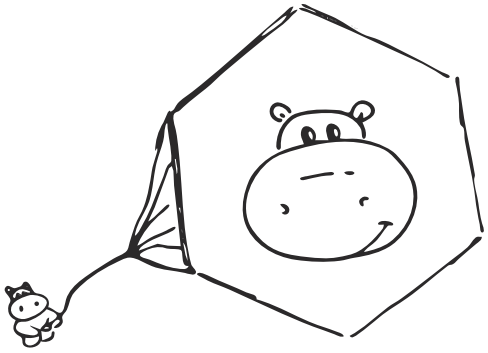
Milí řešitelé!

Poněkud s předstihem dostáváte do rukou zadání třetí série našeho semináře. S ním dostáváte taktéž opravená řešení série první, takže špinavé a podlé fígly, které se z nich naučíte, můžete použít ještě při řešení série druhé :–)



Termín odeslání Vašich řešení třetí série jest 29. ledna 2007. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

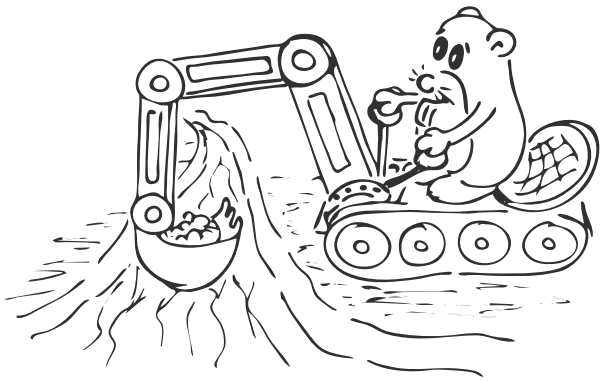
Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz*.

Zadání třetí série devatenáctého ročníku KSP

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 *Spíž*$[i]$. Potraviny jsou označeny čísly, avšak nemusejí být číslovány souvisle a $\max($*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. Po slalomu mezi jezírky se před námi 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 přijímáním 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 mého kamaráda...

To be continued...

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 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

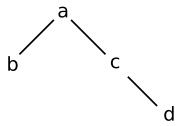
stránce *http://cs.wikipedia.org/wiki/Binární_strom*. Povídání o binárních vyhledávacích stromech najdete na stránce *http://ksp.mff.cuni.cz/tasks/18/cook4.html*. Vyzbrojeni znalostmi se teď můžeme pustit do programování stromů v Prologu.

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.`

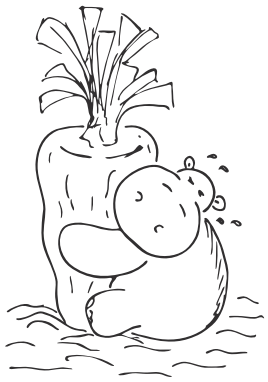
1. `Yes.`
2. `No.`
3. Nastane běhová chyba.

⋆ 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.

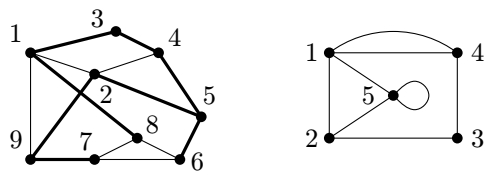
Pozor, vaším úkolem je napsat celý program. Kdykoli použijete jakýkoli predikát, musíte k němu připsat kód, v žádném případě nestačí napsat „použijeme predikát `predek` z učebního textu“ nebo něco podobného.

Recepty z programátorské kuchařky

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.

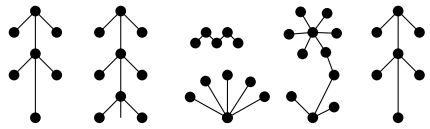
Kružnicí nazýváme cestu délky alespoň 3, ve které oproti definici platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

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 $\mathcal{O}(M \log M)$ (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 $\mathcal{O}(M \log M)$. Celková časová složitost našeho algoritmu je tedy $\mathcal{O}(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $\mathcal{O}(M)$.

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 jeda 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 $\mathcal{O}(N)$.

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 $\mathcal{O}(\log N)$, a tedy operace *find* a *union* stihneme v čase $\mathcal{O}(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $\mathcal{O}(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $\mathcal{O}(kt)$. 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 přičení jedničky k číslu, které je na počátku nula, zabere čas $\mathcal{O}(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $\mathcal{O}(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $\mathcal{O}(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek a pokud jich na N operací použijeme jen $\mathcal{O}(N)$, 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 $\mathcal{O}(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $\mathcal{O}(\log N)$, 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 $\mathcal{O}(\log N)$, 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 $\mathcal{O}(\alpha(N))$ 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 $\mathcal{O}((N+L)\log^* N)$, 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 $1 + \log^* \log N = \mathcal{O}(\log^* N)$ 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 $\mathcal{O}(N \log^* N)$ 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 $\mathcal{O}(\log^* n)$ (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 $\mathcal{O}(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce *root*(v), nejvýše $\mathcal{O}(N \log^* N)$. Protože funkce *root*(v) je volána $2L$-krát, plyne časový odhad $\mathcal{O}((N+L)\log^* N)$ 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 $\alpha(N) \leq 4$.

Dnešní menu vám servírovali
Dan Kráľ, Martin Mareš a Milan Straka

Vzorová řešení první série devatenáctého ročníku KSP

19-1-1 Zlaté časy

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 $\mathcal{O}(1)$. Při přechodu od y k $y+1$ budeme muset do haldy přidat záznam S_y, to jde v $\mathcal{O}(\log N)$. Celkově tedy získáme řešení pracující v čase $\mathcal{O}(N \cdot \log N)$.

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 $\mathcal{O}(1)$ času, tedy celková složitost algoritmu je $\mathcal{O}(N)$.

Pavel Čížek

19-1-2 Čokoláda

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.

Petr Škoda

19-1-3 Tiskárna

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 $\mathcal{O}(NlogN)$. 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 „lišekrát“. 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 „lišekrát“, 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 se`XOR`ovat všechny znaky a vypsat to, co nám zbylo. Časová složitost je $\mathcal{O}(N)$ a paměťová $\mathcal{O}(1)$.

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.

Zbyněk Falt

19-1-4 Mafiánské rodiny

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 $\mathcal{O}(N^2)$ 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 $\mathcal{O}(M+N)$, čili $\mathcal{O}(N)$.

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 vydělíme 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 $\mathcal{O}(1)$, aniž bychom museli záznamy číst.

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

Martin „Bobřík“ Kruliš

19-1-5 Zámek

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 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 $\mathcal{O}(KL^2)$ a $\mathcal{O}(L^2)$, kde L je počet cifer (byť ze zadání plynulo, že je to konstanta. Moje chyba.) a K je délka kombinace.

Jan Bulánek

19-1-6 Prolog

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 řešení 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).
```

Jak víme, `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.

Jana Kravalová

Úloha 19-1-1 – Zlaté časy – program

```
#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;
}
```

Úloha 19-1-3 – Tiskárna – program

```
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 a XOR 0 = 0, }
      idx:=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.
```

Úloha 19-1-5 – Zámek – program

```
#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)
        pocty_new[naslednici[akt_cif][j]]+=pocty[akt_cif];   //vypočtení dalšího sloupečku

    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;
}
```

Úloha 19-1-6 – Prolog – programy

```
% 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).
```

Výsledková listina devatenáctého ročníku KSP po první sérii

		škola	*ročník*	*sérií*	*1911*	*1912*	*1913*	*1914*	*1915*	*1916*	*suma*	*celkem*
1.	Zbyněk Konečný	GKptJaroš	4	13	8	6	8	10	13	12	43,0	43,0
2.	Marek Nečada	G Jihlava	3	1	8		4	6	13	9	40,8	40,8
3.	Josef Pihera	G Strakon	4	11			8	8	12	12	40,0	40,0
4.	Rudolf Rosa	G Kladno	4	3	8	6		4	13	12	39,6	39,6
5. – 6.	Jakub Kaplan	GJKTyla	3	11	8		8		13	10	39,0	39,0
	Petr Kratochvíl	G SvětláNS	4	15	1	6	10	10	13	7	39,0	39,0
7. – 8.	Vlastimil Dort	GŠpitálsPH	1	1	6	6	6	8	4	9	37,2	37,2
	Jiří Maršík	GJKTyla	3	6	8	6		8	10	7	37,2	37,2
9.	Jan Michelfeit	G HBrod	3	1	3	6			13	11	36,3	36,3
10.	Miroslav Klimoš	G Bílovec	2	15			10	8	11	10	36,1	36,1
11.	David Brazdil	G Zlín	2	1	4	2	3	6	5	12	36,0	36,0
12.	Jan Žák	G HBrod	2	1	5	4	7	2	5	7	35,3	35,3
13.	Lucia Simanová	GGrösslin	3	1	4	6	7	0	4	10	34,9	34,9
14.	Karel Tesař	SPŠKoterov	1	1	4	4	6	4	4	9	34,8	34,8
15.	Vojtěch Kolář	G Neratov	3	1	7	3	5	1	3	12	34,2	34,2
16.	Kristýna Krejčová	G Tišnov	4	4	8	4	5	7	3	8	34,0	34,0
17.	Pavel Veselý	G Strakon	2	3	1	3	6	4	4	10	33,3	33,3
18.	Pavel Klavík	G Chrudim	4	15	8		8	10		9	32,8	32,8
19.	Zbyněk Jiráček	GArabská	3	1	5	6	7	6	4	1	32,6	32,6
20.	Trung Ha duc	GMasaryk	1	1	5		4		3	11	32,4	32,4
21.	Dušan Rychnovský	G Hranice	3	2	1	4	4	8		8	32,1	32,1
22.	Libor Plucnar	GPBezruče	2	1	6	4	9	1	4		30,5	30,5
23.	Viktor Lucza	G Rožňava	3	1	8	6	4	4	4		30,1	30,1
24.	Radim Cajzl	G NMnMor	0	6	4	4	9		4	7	29,7	29,7
25.	Jan Matějka	GJírovco	2	1		6	7	3	4		29,0	29,0
26.	Jakub Slavík	GJKTyla	3	1		6	8	6		2	28,8	28,8
27.	Václav Strnad	GArabská	3	1	5	3	6	4			27,5	27,5
28.	Lukáš Kripner	G Litvínov	1	1			6	7		5	26,4	26,4
29.	Radim Pechal	SPŠ Rožnov	4	3	8			10	5		26,3	26,3
30.	Tomáš Herceg	G Třebíč	4	12	5		10	5		5	24,2	24,2
31.	David Škorvaga	G Kralupy	4	2	8	6				7	24,0	24,0
32.	Jan Kohout	G Roudnice	4	6	4	6	3			5	22,8	22,8
33.	Martin Kahoun	GJNerudy	4	6	5	4	6			2	21,8	21,8
34.	Mirek Jarolím	GMikuláš	1	1	4	6	3	1		1	21,0	21,0
35.	Jakub Balhar	GJNerudy	4	3	5	4	4			1	20,7	20,7
36. – 37.	Ondřej Bouda	GKptJaroš	4	7		6	10			2	19,0	19,0
	Jan Kučera	G Polička	4	1	1		6		4	0	19,0	19,0
38.	Jan Sixta	G Brandýs	4	1	3	3	5			0	18,2	18,2
39.	Petr Onderka	G VKlobou	4	6			8	8			17,8	17,8
40.	Tomáš Sýkora	G VKlobou	3	4	4	3	4			0	16,4	16,4
41.	Tomáš Volf	G Tábor	0	1	0	3	1	1		1	12,4	12,4
42.	Jakub Pavlík jn.	G Kladno	4	4						12	12,0	12,0
43.	Karel Pajskr		2	1						9	11,2	11,2
44.	Stanislav Fořt	G Tábor	0	1	0		5	1	0	0	10,4	10,4
45.	Richard Jedlička	G Vlašim	3	6			8				8,9	8,9
46.	Miroslav Jančařík	G UBrod	3	3		3	2				8,5	8,5
47.	Jan Papoušek	GKptJaroš	3	1	8						8,0	8,0
48.	Milan Klášterka	SPŠKlatovy	3	1		3				1	7,4	7,4
49. – 50.	Jan Krajdl	SPŠÚžlabin	2	3			4				6,6	6,6
	Martin Majer	SPŠÚžlabin	2	3	1	3					6,6	6,6
51.	Martin Pástor	SPŠAlejová	2	1		6					6,0	6,0