Výsledková listina devatenáctého ročníku KSP po třetí sérii

		škola	ročník	sérií	1931	1932	1933	1934	1935	1936	suma	celkem
1.	Marek Nečada	G Jihlava	3	3	10	6		13	5		36,5	117,1
2.	Jakub Kaplan	GJKTyla	3	13	10		7		10	7	33,3	115,0
3.	Vojtěch Kolář	G Neratov	3	3	7	5	7	8	8	10	40,2	110,6
4.	Zbyněk Konečný	GKptJaroš	4	15	9	6	7	2	5	10	30,5	105,9
5.	Petr Kratochvíl	G SvětláNS	4	17	9.5	6	7		3	12	33,2	103,7
6.	David Brazdil	G Zlín	2	3	7	6	1	3	5	7	32,0	101,4
7.	Jan Michelfeit	G HBrod	3	3	7	7	7			10	34,1	101,3
8.	Matěj Korvas	GJSeiferPH	4	3			4	4	10		22,8	99,9
9.	Jan Žák	G HBrod	2	3	7	7	1			10	29,2	93,7
10.	Pavel Klavík	G Chrudim	4	17	10	7	7		5	12	36,0	92,1
11.	Trung Ha duc	GMasaryk	1	3		6	7		1	7	24,9	92,0
12.	Vlastimil Dort	GŠpitálsPH	1	3	5	7	7				21,5	88,1
13.	Libor Plucnar	GPBezruče	2	3	3	6	7	5	5		28,8	87,8
14.	Petr Onderka	G VKlobou	4	8	9		7		4	9	31,2	72,6
15.	Miroslav Klimoš	G Bílovec	2	17				9	10	10	25,2	71,6
16.	Jakub Balhar	GJNerudy	4	5	5	2	6	6			25,2	68,1
17.	Pavel Veselý	G Strakon	2	5		3			5		11,1	67,1
18.	Karel Pajskr		2	3		6	7			10	24,3	66,8
19.	Jiří Maršík	GJKTyla	3	7	8	7	7		4		28,1	65,3
20.	Josef Pihera	G Strakon	4	13						14	14,0	64,9
21.	Vojtěch Tůma	G Jihlava	3	2	3	3		10	5		30,1	60,2
22.	Tomáš Sýkora	G VKlobou	3	6	5	6	7		2		22,8	60,0
23.	Lukáš Kripner	G Litvínov	1	3	3	6			5		18,9	54,9
24.	Tomáš Herceg	G Třebíč	4	14		6			5	5	13,9	54,2
25.	Kristýna Krejčová	G Tišnov	4	5							0,0	52,6
26.	Zbyněk Jiráček	GArabská	3	2							0,0	50,6
27.	Viktor Lucza	G Rožňava	3	2							0,0	50,4
28.	Jan Kohout	G Roudnice	4	8		3			5		9,6	45,4
29.	Radim Cajzl	G NMnMor	0	8					5		6,0	44,4
30.	Jakub Červenka	GŠpitálsPH	1	2		6	7			10	24,4	43,8
31.	Rudolf Rosa	G Kladno	4	3							0,0	39,6
32.	Ondřej Bouda	GKptJaroš	4	8							0,0	36,9
33.	Martin Kahoun	GJNerudy	4	7							0,0	36,5
34.	Martin Majer	SPŠÚžlabin	2	5	5	6			5		19,8	36,2
35.	Lucia Simanová	GGrösslin	3	1							0,0	34,9
36.	Karel Tesař	SPŠEPlzeň	1	1							0,0	34,8
37.	Dušan Rychnovský	G Hranice	3	2							0,0	32,1
38.	Martin Němec	G Ledeč	3	2		2					3,7	31,3
39.	Mirek Jarolím	GMikuláš	1	3		3					4,5	29,2
40.	Jan Matějka	GJírovco	2	1							0,0	29,0
41.	Jakub Slavík	GJKTyla	3	1							0,0	28,8
42.	Petr Holášek	G Příbor	3	2	6	4	1	1	2	5	26,3	28,6
43.	Václav Strnad	GArabská	3	1							0,0	27,5
44.	Radim Pechal	SPŠ Rožnov	4	3							0,0	26,3
45.	Stanislav Fořt	G Tábor	0	3		2					3,5	24,5
46.	David Škorvaga	G Kralupy	4	2							0,0	24,0
47.	Marika Ivanová	SPŠ Zlín	3	1	6	3	5				19,8	19,8
48.	Jan Kučera	G Polička	4	1							0,0	19,0
49. – 50.	Milan Klášterka	SPŠKlatovy	3	3		1					2,1	18,2
	Jan Sixta	G Brandýs	4	1							0,0	18,2
51.	Roman Smrž	GOhradní	3	9	10					5	15,8	15,8
52.	Jakub Pavlík jn.	G Kladno	4	5							0,0	15,6
53.	Richard Jedlička	G Vlašim	3	7		6					6,0	14,9
54.	Karolína Burešová	G ČLípa	0	1							0,0	13,9
55.	Jan Krajdl	SPŠÚžlabin	2	4		6					6,0	12,6
56.	Tomáš Volf	G Tábor	0	1							0,0	12,4
57.	Miroslav Jančařík	G UBrod	3	3							0,0	8,5
58.	Jan Papoušek	GKptJaroš	3	1							0,0	8,0
59.	Martin Pástor	SPŠAlejová	2	1							0,0	6,0
60.	Radomír Švihel	G Zlín	3	1		0					0,0	0,0

Milí řešitelé!

Letošní zima se opravdu vydařila. Počasí přálo, Vánoce se vydařily. Až na ten sníh! Hroši měli na lyžích co dělat!

Snad jste si absenci zimních radovánek vynahradili radovánkami řešitelskými a pokud jich stále nemáte dost, můžete se s chutí zakousnout do další, letos už poslední série. Dobrou chuť.

Termín odeslání Vašich řešení páté série jest 7. května 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 páté série devatenáctého ročníku KSP

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. Ale 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 jí 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 stanice 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 třítisícetřistatřicetikoruna.

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 vyndavat 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 $\mathcal{O}(N)$ 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 $\mathcal{O}(N)$ 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 jí vlastně taky…

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

Milí účastníci a účastnice, 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).

Úloha 19-3-6 – Prolog – program

```
% 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. Zároveň se mi vrátí 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 nepoužiý zbytek z celého predikátu.
        strom2(Zbytek1, DelkaPravy, Zbytek, PravyStrom).
```

```
    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;
}
```

Úloha 19-3-5 – Pevné vztahy – program

```
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.
```


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 člověkovi, jaká zvířata chová:

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

Predikát `findall(Term, Cíl, Seznam)` totiž vytvoří seznam `Seznam` z takových termů `Term`, že splňují daný `Cíl`. 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 petrem 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:


```
  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.
```

Úloha 19-3-4 – Nejbližší rostoucí posloupnost – program

```
#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]);
```

```
      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.
```

Úloha 19-3-2 – Inventura ve spíži – program

```
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);
```


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.

Recepty z programátorské kuchařky

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

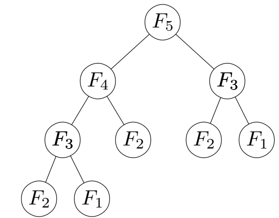
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

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

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

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

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:

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

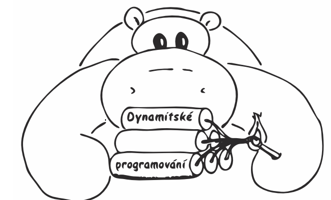
Tedy na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že:

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

z čehož plyne:

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

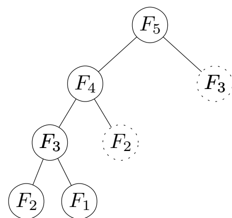
Funkce `Fibonacci` má tedy exponenciální časovou složitost, což není nic vítaného. Ovšem jak jsme už řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka z klobouku s minimem námahy.



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 nyní 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 $\mathcal{O}(MN)$.

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

Časová složitost algoritmu je $\mathcal{O}(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $\mathcal{O}(M)$. Paměťová složitost činí $\mathcal{O}(N + M)$, 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;
```


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 vytvářeného 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` $N - K1 - 1$.

Jana Kravalová

Úloha 19-3-1 – Jezírka – program

```
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;
```

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.

Tereza Klimošová

19-3-6 Prolog

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 zahradce 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 velezná 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


```
  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 $1, \ldots, k$. V průběhu k-té fáze tedy stačí vyzkoušet, zda je mezi městy i a j kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo nová cesta přes město k. Pokud nejkratší cesta prochází přes město k, 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 $\mathcal{O}(N^2)$, a tím pádem časová složitost celého algoritmu bude $\mathcal{O}(N^3)$. Vystačíme si s pamětí na uložení pole D, tedy $\mathcal{O}(N^2)$. 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.

Dnešní menu vám servírovali
Martin Mareš a Petr Škoda

Vzorová řešení třetí série devatenáctého ročníku KSP

19-3-1 Jezírka

Sice se proslýchá, že lesní duchové umí tuto úlohu řešit ještě rychleji (a to v $\mathcal{O}(\log N)$), 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).

Michal „vorner“ Vaner

19-3-2 Inventura ve spíži

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 $\mathcal{O}(N^2)$ 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 $\mathcal{O}(N \log N)$.

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 nakonec najdeme jednoduchým průchodem trií do hloubky. Časová složitost celého algoritmu je $\mathcal{O}(N)$, paměťová je rovněž $\mathcal{O}(N)$.

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.

Zbyněk Falt

19-3-3 Nevěrné ženy

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!

Martin „Bobřík“ Kruliš

19-3-4 Nejbližší rostoucí posloupnost

„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 složena 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 složena 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 rovna 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 rovnou 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 dostat 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 $O(n)$.

Petr Škoda

19-3-5 Pevné vztahy

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.