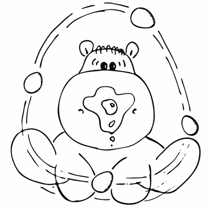
Výsledková listina devatenáctého ročníku KSP po čtvrté sérii

		škola	*ročník*	*sérií*	*1941*	*1942*	*1943*	*1944*	*1945*	*1946*	*suma*	*celkem*
1.	Marek Nečada	G Jihlava	3	4		9	7	8	9		36,3	153,4
2.	Jakub Kaplan	GJKTyla	3	14	8	9	11	10			37,6	152,6
3.	Vojtěch Kolář	G Neratov	3	4	5	8,5	7	6	1	13	38,6	149,1
4.	Zbyněk Konečný	GKptJaroš	4	16	8	9	7	10	10	11	37,6	143,5
5.	Jan Michelfeit	G HBrod	3	4		9	6	7	9	10	39,8	141,2
6.	David Brazdil	G Zlín	2	4	4	6	6	8	8,5	11	38,6	139,9
7.	Petr Kratochvíl	G SvětláNS	4	18	8	10	5	9	9,5	10	36,0	139,6
8.	Trung Ha duc	GMasaryk	1	4	7	7,5			9,5	11	39,1	131,1
9.	Jan Žák	G HBrod	2	4	4	8,5	6	5	4	12	37,3	131,0
10.	Pavel Klavík	G Chrudim	4	18	8	9	11	9	9,5	6	37,0	129,1
11.	Vlastimil Dort	GŠpitálsPH	1	4	7	9	5	10	5		34,7	122,8
12.	Libor Plucnar	GPBezruče	2	4	3	6			9		22,4	110,2
13.	Petr Onderka	G VKlobou	4	9		8		6	3	10	28,8	101,4
14.	Matěj Korvas	GJSeiferPH	4	3							0,0	99,9
15.	Vojtěch Tůma	G Jihlava	3	3	8	9		5	6		31,6	91,8
16.	Miroslav Klimoš	G Bílovec	2	18						14	14,0	85,6
17.	Pavel Veselý	G Strakon	2	6	3			2	5		13,8	80,9
18.	Tomáš Sýkora	G VKlobou	3	7	4	8		4			18,6	78,6
19.	Karel Pajskr	GJKeplera	2	4						9	11,6	78,4
20.	Jakub Balhar	GJNerudy	4	5							0,0	67,4
21.	Tomáš Herceg	G Třebíč	4	15	3	6		6			12,8	67,0
22.	Jiří Maršík	GJKTyla	3	7							0,0	65,3
23.	Jan Kohout	G Roudnice	4	9	2	3	6		6		19,5	65,0
24.	Josef Pihera	G Strakon	4	13							0,0	64,9
25.	Radim Cajzl	G NMnMor	0	9	4	8,5					13,4	57,8
26.	Martin Majer	SPŠÚžlabin	2	6	4	6	5				19,6	55,8
27.	Lukáš Kripner	G Litvínov	1	3							0,0	54,9
28.	Kristýna Krejčová	G Tišnov	4	5							0,0	52,6
29.	Zbyněk Jiráček	GArabská	3	2							0,0	50,6
30.	Viktor Lucza	G Rožňava	3	2							0,0	50,4
31.	Roman Smrž	GOhradní	3	10		10	6			13	29,5	45,3
32.	Dušan Rychnovský	G Hranice	3	3		3			5		12,9	45,0
33.	Jakub Červenka	GŠpitálsPH	1	2							0,0	43,8
34.	Petr Holášek	G Příbor	3	3	4	6					14,2	42,8
35.	Rudolf Rosa	G Kladno	4	3							0,0	39,6
36.	Roman Říha	G Prachat	3	2		9	11	8	9		38,4	38,4
37.	Marika Ivanová	SPŠ Zlín	3	2	6	3		4			18,4	38,2
38.	Ondřej Bouda	GKptJaroš	4	8							0,0	36,9
39.	Martin Kahoun	GJNerudy	4	7							0,0	36,5
40.	Lucia Simanová	GGrösslin	3	1							0,0	34,9
41.	Karel Tesař	SPŠEPlzeň	1	1							0,0	34,8
42.	Stanislav Fořt	G Tábor	0	4	3	0	3				10,0	34,5
43.	Martin Němec	G Ledeč	3	2							0,0	31,3
44.	Richard Jedlička	G Vlašim	3	8		5,5			8,5		15,5	30,4
45.	Mirek Jarolím	GMikuláš	1	3							0,0	29,2
46.	Jan Matějka	GJírovco	2	1							0,0	29,0
47.	Jakub Slavík	GJKTyla	3	1							0,0	28,8
48.	Václav Strnad	GArabská	3	1							0,0	27,5
49.	Tomáš Jakl	G MTřebová	3	1	3	6	5	1	2		26,9	26,9
50.	Radim Pechal	SPŠ Rožnov	4	3							0,0	26,3
51.	David Škorvaga	G Kralupy	4	2							0,0	24,0
52.	David Marek	SPŠ Zlín	3	1	4	7	3		0,5		23,1	23,1
53.	Amadeo Mareš	SOŠ Blatná	2	1	3	1	5	2	1,5		21,2	21,2
54.	Jan Krajdl	SPŠÚžlabin	2	5		7					8,5	21,1
55.	Jan Kučera	G Polička	4	1							0,0	19,0
56. – 57.	Milan Klášterka	SPŠKlatovy	3	3							0,0	18,2
	Jan Sixta	G Brandýs	4	1							0,0	18,2
58.	Jakub Pavlík jn.	G Kladno	4	5							0,0	15,6
59.	Karolína Burešová	G ČLípa	0	1							0,0	13,9
60.	Tomáš Volf	G Tábor	0	1							0,0	12,4
61.	Miroslav Jančařík	G UBrod	3	4					1		2,0	10,5
62.	Jan Papoušek	GKptJaroš	3	1							0,0	8,0
63.	Josef Sedlačík	G UBrod	3	1			3				6,2	6,2
64.	Martin Pástor	SPŠAlejová	2	1							0,0	6,0
65.	Matyáš Bach	G VKlobou	0	1			1				2,6	2,6
66.	Radomír Švihel	G Zlín	3	1							0,0	0,0

Milí řešitelé a řešitelky!



Přinášíme vám vaše opravená a naše vzorová řešení čtvrté série. Letošní ročník se tak chýlí ke konci, zbývá už jenom vyřešit a opravit poslední pátou sérii. Těšíme se na vaše výtvory!

Termín odeslání šesté série jest pochopitelně neurčen, alebrž žádné šesté série není. Řešení byste bývali mohli odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak na:

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*.

Vzorová řešení čtvrté série devatenáctého ročníku KSP

19-4-1 Finanční toky

De facto všichni řešitelé dokázali najít stok, pokud v daném grafu byl, nicméně je třeba si přiznat, že při použití kvadratického řešení by Přesprst sotva unikl. Otázkou jen zůstává, jak to napsat rychleji.

Otestovat, zda-li je vrchol stok dokážeme zřejmě v $\mathcal{O}(N)$. Problém je, že při použití algoritmu, který se u každého vrcholu zeptá, zda-li je stok, bude celý program potřebovat $\mathcal{O}(N^2)$ času. Celý problém v nalezení rychlejšího řešení bude tedy v nějakém šikovném výběru kandidáta na stok.

Podívejme se tedy na vlastnosti stoku. Zřejmě v grafu může existovat jen jeden (sporem: Pokud by byly alespoň 2 stoky i a j, tak dle definice musí vést do stoku i hrana z každého jiného vrcholu, tedy i z j. Nicméně pokud j je stok, tak z něj žádná hrana nemůže vést…).

A jak kandidáta na stok nalézt? V i-tém kroku budeme pamatovat jediného kandidáta na stok mezi vrcholy $1..i$ (začneme s 1). Označme ho k. Kandidáta pro $(i+1)$-ní krok zjistíme jednoduše. Když vede z k do $i+1$ hrana, tak zřejmě k nemůže být stok a jelikož mezi vrcholy $1..i$ byl jediným kandidátem, tak naději na to, stát se stokem, má jen vrchol $i+1$, což bude náš nový kandidát. V opačném případě hrana z k do $i+1$ nevede, takže vrchol $i+1$ nemůže být stok, jelikož do něj nevede hrana z každého jiného vrcholu a tedy kandidát zůstává.

Nakonec jen otestujeme, zda-li kandidát, kterého získáme n-tým krokem je stok, nebo ne.

Nalezení kandidáta i jeho otestování se zřejmě stihne v čase $\mathcal{O}(N)$ a paměťové nároky, pokud nepočítáme vstupní matici, jsou konstantní.

Pavel Čížek

19-4-2 Byrokratický aparát

Měl jsem připraveny velmi vtipné hlášky, například že bude třeba zestátnit zemědělské podniky, aby nám pomalu se courající úřední šimlík nepošel na své dlouhotrvající cestě hlady, bohužel, nebo spíš bohudík si je musím odpustit, protože většina z vás vytvořila optimálně rychlé algoritmy. A tak vám mohu vytknout jen jedinou věc. Ale k tomu až později.

Řešení úlohy se dalo rozdělit na dvě části. V první části si stačilo povšimnout, že graf (vrcholy jsou úředníci, orientované hrany pak značí směr dokumentu) se skládá z oddělených cyklů. Tyto cykly se pak dají najít v lineárním čase. Jistě, mohli bychom na to použít DFU (viz kuchařka 16-3), ale my jsme řekli ne. Je sice fakt, že se DFU chová docela dobře, ale vaše implementace fungovaly většinou v čase $\mathcal{O}(N \log N)$. Více už příslušná kuchařka. Správné řešení spočívalo v použití procházení do hloubky. Vždy si vezmu první nepoužitý vrchol, označím ho jako použitý, posunu se na jeho následníka a to opakuji tak dlouho, než dojdu do již použitého vrcholu. A protože na každý vrchol sáhnu nejvýše dvakrát, poprvé když zkoumám jestli už byl použit a podruhé ve chvíli, kdy je něčím následníkem a přesunu se na něj, tak mě lineární časová složitost nemine. Samozřejmě nebude problém si přitom délky jednotlivých cyklů počítat.

Druhá část je ta, u které se lámal chleba. Téměř každý z vás si uvědomil, že je třeba spočítat nejmenší společný násobek délek všech cyklů. To je celkem jasně vidět z toho, že hledáme takový počet kroků, pro který bude mít každý úředník svůj dokument, tzn. každý cyklus musel být ukončen. No ale tím pádem musí být počet kroků dělitelný každou délkou cyklu a zároveň musí být nejmenší. A to je právě nejmenší společný násobek (NSN). No a jelikož $NSN(a,b) = a \cdot b / NSD(a,b)$ (důkaz si jakožto jednoduché cvičení proveďte sami) a $NSN(l_1, l_2, \ldots, l_N) = NSN(NSN(l_1, l_2, \ldots, l_{N-1}), l_N)$, tak je postup nabíledni. Postupně budeme počítat NSN prvních k čísel, z něj pak $k+1$ čísel atd. až N. Zbývá jen pořešit získávání NSD. Na to slouží Euklidův algoritmus, který si najděte např. na *http://cs.wikipedia.org/wiki/Euklidův_algoritmus*.

O složitosti této složitosti vypovídala vaše řešení. Jen velmi málo z vás ji určilo opravdu správně, takže si dovolím být poněkud obšírnější. Složitost Euklidova algoritmu je $\mathcal{O}(\log a)$, kde a je větší z čísel. Jenže po prvním kroku hledáme $NSD(a \bmod b, b)$, takže se dá říct, že ta složitost je $\mathcal{O}(\log b) + 1 = \mathcal{O}(\log b)$. Při výpočtu potom počítáme vždy NSN nějakého čísla a jedné z délek cyklů. Považujme délku za menší číslo (rozmyslete si, proč si tím mohu pouze uškodit) a tím pádem jeden takový krok trvá $\mathcal{O}(\log(\textit{délka cyklu}))$. Nyní přijde jedno malé kouzlíčko: $\log(\textit{délka cyklu}) \leq \textit{délka cyklu}$ a součet všech délek je N, a tak složitost bude nejvýše $\mathcal{O}(N)$. To nám ale bohatě jako odhad stačí, protože ani první část netrvala kratší dobu. Paměť je samozřejmě lineární.

Několikrát se vyskytl Euklidův algoritmus, který místo modula používal mínus. To sice funguje, ale složitost se nám vyšplhá až do exponenciálních výšin. Nejprve si prohlédneme chování algoritmu pro K a 1 a s hrůzou zjistíme, že udělá K kroků. Pak stvoříme vhodný vstup, třeba prvních $\sqrt{N}$ délek bude mít velikost řádově $\sqrt{N}$ (jejich součet je tedy N), takže jejich NSN bude asi $\sqrt{N}^{\sqrt{N}}$, a vtipně je doplníme jedním cyklem délky 1 a složitost $\mathcal{O}(\sqrt{N}^{\sqrt{N}})$ je na světě. Nicméně to neberte jako důkaz, protože to ve skutečnosti nic nedokazuje, ale spíš jako náhled na to, kam až může záměna jedné operace vést.

Samozřejmě se vyskytly i jiné přístupy, například jste si zjistili prvočinitele každé délky a jejich pronásobením získali kýžený NSN. Zkuste si dorozmyslet detaily a zároveň si spočtěte, že je to skutečně lineární.

Jan Bulánek

19-4-3 Naskakování na vlak

Hned na začátek si neodpustím jednu poznámku: ve všech algoritmech budeme zkoumat pouze složitost, se kterou algoritmus řešení nalezne. Časovou složitost na jeho vypsání v odhadech počítat nebudeme. Vyniknou tak lépe rozdíly mezi jednotlivými algoritmy. Pokud by to někomu připadalo nefér, tak si může ke všem složitostem přičíst $\mathcal{O}(v \cdot k)$, kde v je počet navzájem různých podřetězců délky k.

Nyní již k samotné úloze. Mnoho řešitelů využilo nápovědu v zadání úlohy, a tak drtivá většina řešení využívala hashování. Ale už jenom drobná hrstka objevila, že úplně přímočaré použití kuchařky k rychlému řešení nepovede.

Základní algoritmus, který se na první pohled nabízel, byl ten, že jsme postupně brali jednotlivé podřetězce délky k, ty jsme zahashovali, a pak jsme si v nějaké tabulce (po ošetření kolizí) ukládali počet výskytů jednotlivých podřetězců. Takové řešení má v průměrném případě časovou složitost $\mathcal{O}(n \cdot k)$

Předchozí metoda měla tu nevýhodu, že jsme pro každý podřetězec museli spočítat znovu celou hashovací funkci a to zabere čas $\mathcal{O}(k)$. Co kdybychom ale našli takovou funkci, která by dokázala využít toho, že její hodnotu známe již pro předchozí podřetězec?

$$\sum_{j=0}^{k-1} A_i[j] \cdot P^{k-j-1}$$

Zápis $A_i[j]$ je totéž co $A[i+j]$, tedy j-tý znak od i-tého znaku v řetězci a P je nějaké číslo, které je řádově tak velké, jako velikost abecedy.

Pokud chceme přejít na následující podřetězec provedeme tyto operace: celou sumu vynásobíme P, škrtneme první písmeno z předchozího slova a přičteme poslední písmeno z následujícího. Matematicky zapsáno:

$$P \cdot \sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j-1}) - A_i[0] \cdot P^k + A_i[k] =$$
$$\sum_{j=0}^{k-1} (A_i[j] \cdot P^{k-j}) - A_i[0] \cdot P^k + A_i[k] =$$
$$\sum_{j=1}^{k} A_i[j] \cdot P^{k-j} = \sum_{j=0}^{k-1} A_{i+1}[j] \cdot P^{k-j-1}$$

Takže jsme použili konstantně mnoha kroků (nezávisle na k) a získali jsme hodnotu hashovací funkce pro řetězec, který začíná na pozici $i+1$, a to je přesně to, co jsme chtěli.

Zbývá dořešit několik technických detailů. V běžných programovacích jazycích máme proměnné omezeného rozsahu, takže pro velké k nemůžeme spočítat celou sumu. Ale můžeme si pomoci. Stačí všechny operace provádět modulo nějaké prvočíslo. A jako ono prvočíslo můžeme použít třeba rovnou velikost hashovací tabulky.

Za poznámku stojí, že ono prvočíslo musí být opravdu prvočíslo, jinak bychom se dostali do problému. Odpověď na otázku „Proč?" by asi nebyla nejstručnější, zájemci si ale mohou přečíst nějaké povídání o konečných tělesech.

Jak ale takové prvočíslo najít? Dle teorie čísel je pravděpodobnost toho, že libovolné přirozené číslo n je prvočíslem, je zhruba $1/\ln n$, a ověření toho, že n je prvočíslo lze základním algoritmem provést v čase $\mathcal{O}(\sqrt{n})$. Takže prvočíslo větší než nějaké n lze najít v čase $\mathcal{O}(\sqrt{n} \cdot \ln n)$, což je méně než $\mathcal{O}(n)$. Takže problém s hledáním prvočísla mít nebudeme.

A jak to bude s paměťovou složitostí? Mnoho řešitelů si pro každou položku v hashovací tabulce pamatovalo celý podřetězec. To je ale zbytečné a paměťová složitost se tím zhorší. Stačí si přeci pamatovat pouze index, kde daný podřetězec ve vstupním řetězci začíná, což zlepší časovou složitost na $\mathcal{O}(n)$.

Takže jsme nalezli algoritmus, který v průměrném případě poběží v čase $\mathcal{O}(n + v \cdot k)$, kde v je opět počet různých podřetězců. V nejhorším případě pak v čase $\mathcal{O}(n^2 \cdot k)$.

Poznámka na úplný závěr: Pokud bychom chtěli dosáhnout času $\mathcal{O}(n + v \cdot k)$ i v nejhorším případě, mohli bychom použít sufixové stromy. Povídání o této datové struktuře a i návod jak pomocí ní vyřešit tuto úlohu lze nalézt na adrese *http://mj.ucw.cz/vyuka/ga/*.

Zbyněk Falt

19-4-4 Váhy

Pošťák by asi koukal, kdyby jste se mu snažili vysvětlit, jak zvážit libovolnou celočíselnou váhu s optimální sadou závaží. Jak vypadá optimální sada závaží? Jsou to závaží $1, 3, 9, 27, \ldots$, tedy mocniny trojky. Kolik jich potřebujeme a jak jsme na to přišli? To si hned ukážeme.

Podívejme se na problém z jiného úhlu. Mějme n dané. Jaké je největší m, že s nějakou sadou o n závažích dokážeme zvážit všechny celočíselné hmotnosti $1, \ldots, m$? Představme si rovnoramenné váhy. Na jednu stranu položíme předmět neznámé hmotnosti. Nyní máme pro každé závaží právě jednu ze tří možností. Buď závaží položíme na váhu k předmětu, nebo ho položíme na druhou stranu, a nebo ho na váhu nedáme vůbec. Pro každý z n závaží jsme vybrali jednu ze tří možností, celkem lze tedy vytvořit 3^n různých rozmístění závaží. Je jasné, že můžeme zvážit maximálně tolik různých hmotností, kolik je různých rozmístění závaží. Dostáváme jednoduchý odhad $m \leq 3^n$. Ale ještě ho vylepšíme následujícími dvěma pozorováními. Pokud žádné závaží na váhu nedáme, je zřejmé, že žádnou hmotnost neodvážíme. Podobně, pokud s jedním rozmístění závaží zvážíme hmotnost $k > 0$, pak prohozením závaží z jedné strany vah na druhou dostaneme korektní rozmístění, které ale nezváží žádnou kladnou hmotnost. Proto alespoň polovina rozmístění závaží neodváží žádnou hmotnost $1, \ldots, m$. Dostáváme horní odhad $m \leq \frac{3^n-1}{2}$ a ukážeme, že s optimální sadou tohoto horního odhadu dosáhneme.

Vraťme se zpátky k sadě závaží složené z mocnin trojek. Dokážeme nyní indukcí podle počtu závaží, že s n závažími $1, 3, \ldots, 3^{n-1}$ zvážíme všechny celočíselné hmotnosti od 1 do $\frac{3^n-1}{2}$.

- Pro $n = 1$ to určitě platí, protože $\frac{3^n-1}{2} = 1$.
- Nechť tvrzení platí pro všechna $k < n$. Rozdělme vážené hmotnosti do čtyřech intervalů.
  - Hmotnosti $1, \ldots, \frac{3^{n-1}-1}{2}$ zvážíme dle indukčního předpokladu s prvními $n-1$ závažími.

Úloha 19-4-5 – Hazardní hra – program

```
program hra;
var N: integer;

procedure povely(c: integer);
begin
   if c = 0 then exit;
   if c mod 10 = 0 then begin
      povely(c div 10);
      write('*');
   end else if c mod 10 <= 5 then begin
      povely(c - 1);
      write('+');
   end else begin
      povely(c + 1);
      write('-');
   end
end;

begin
   read(N);
   povely(N);
   writeln;
end.
```

Úloha 19-4-6 – Prolog – program

```
% KSP 1946 1.Lednice

pocet(_, [], 0).
pocet(Jidlo, [Jidlo|Zbytek], Pocet) :- !, pocet(Jidlo, Zbytek, Pocet1), Pocet is Pocet1 + 1.
pocet(Jidlo, [_|Zbytek], Pocet) :- pocet(Jidlo, Zbytek, Pocet).


% KSP 1946 2.Myší spartakiáda
barva('c').
barva('b').

generuj(0, Sezn) :- vypis(Sezn), !, fail.
generuj(X, Sezn) :- Y is X - 1, barva(B), generuj(Y, [B|Sezn]).

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

mysi(N) :- generuj(N,[]).


% KSP 1946 3.Myší bludiště

% Reprezentace hrany faktem h(odkud,kam)
% Obousměrné hrany zapíšeme dvěma fakty
% Např. h(1,2). h(2,1).

h(1,2). h(2,1).
h(2,3). h(3,2).
h(3,1). h(1,3).
h(3,4). h(4,3).

cesta([Cil|_], Cil, [], _).

cesta([Start|Fronta], Cil, [[Start,Sousedi]|Zbytek], Navst) :-                  % vyber aktuální vrchol ze zásobníku
        findall(Soused, (h(Start,Soused),not(member(Soused,Navst))), Sousedi),   % najdi jeho nenavštívené sousedy
        append(Fronta, Sousedi, NovaFronta),                                     % vlož sousedy na zásobník
        append(Navst, Sousedi, NovyNavst),                                       % označ sousedy jako navštívené
        cesta(NovaFronta, Cil, Zbytek, NovyNavst).                               % spusť se s novým zásobníkem

cesta(Start, Cil,       Cesta) :- cesta([Start], Cil, Predchudci, [Start]),
        najdi_cestu(Start, Cil, Predchudci, CestaObr),
        reverse(CestaObr,Cesta).

najdi_cestu(Start, Start, _, [Start]).

najdi_cestu(Start, Cil, Predchudci, [Cil|Cesta]) :-
        najdi_predchudce(Cil, Predchudci, Predchudce),
        najdi_cestu(Start, Predchudce, Predchudci, Cesta).

najdi_predchudce(Vrchol, [[Predchudce,Naslednici]|_], Predchudce) :- member(Vrchol,Naslednici).
najdi_predchudce(Vrchol, [_|Zbytek], Predchudce) :- najdi_predchudce(Vrchol, Zbytek, Predchudce).
```

Úloha 19-4-3 – Naskakování na vlak – program

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

typedef struct {
        const char *podret;
        int pocet;
} ZAZNAM;

int main()
{
        ZAZNAM *tabulka;
        char *retezec=NULL;
        int velikost,n;
        int c,i,k,j,pom,hash;

        velikost=n=0;
        while ((c=getchar())!='\n') {                   /* Ukázka, jak načítat vstup, když předem neznáme jeho délku*/
                if (n>=velikost) {
                        velikost=(velikost)?(2*velikost):128;
                        retezec=(char*)realloc((void*)retezec,velikost);
                }
                retezec[n++]=c;
        }
        retezec[n]=0;

        for (velikost=2*n+1;;velikost+=2) {             /* Najdeme nejbližší prvočíslo vyšší než 2N+1 */
                for (i=3;i*i<=velikost && (velikost%i);i+=2);
                        if (velikost%i)
                                break;
        }

        tabulka=(ZAZNAM*)calloc(velikost,sizeof(ZAZNAM));

        scanf("%d",&k);

        pom=1;                                          /* Spočítáme hash pro prvních k znaků, za P zvolíme třeba 113 */
        for (hash=i=0;i<k;i++) {
                hash=(113*hash+retezec[i])%velikost;
                pom=(113*pom)%velikost;
        }

        tabulka[hash].pocet=1;
        tabulka[hash].podret=retezec;

        for (i=1;i<=n-k;i++) {
                hash=113*hash+retezec[i+k-1];           /* Přepočítáme funkci pro další k-tici */
                hash-=(pom*retezec[i-1])%velikost;
                hash=(hash+velikost)%velikost;

                for (j=hash;tabulka[j].pocet && memcmp(tabulka[j].podret,retezec+i,k);j=(j+1)%velikost);

                tabulka[j].podret=retezec+i;            /* A vložíme do hashovací tabulky */
                tabulka[j].pocet++;
        }

        for (i=0;i<velikost;i++) {                      /* A nakonec vypíšeme výstup */
                if (tabulka[i].pocet) {
                        for (j=0;j<k;j++)
                                putchar(tabulka[i].podret[j]);
                        printf(" %dx\n",tabulka[i].pocet);
                }
        }

        free((void*)retezec);
        free((void*)tabulka);

        return 0;
}
```


- Hmotnosti $\frac{3^{n-1}-1}{2}+1 = 3^{n-1} - \frac{3^{n-1}-1}{2}, \dots, 3^{n-1}-1$ zvážíme tak, že n-té závaží dáme naproti předmětu a pomocí prvních $n-1$ závaží, které přikládáme „obráceně“, odvážíme libovolnou hmotnost z tohoto intervalu.
- Hmotnost 3^{n-1} zvážíme pomocí n-tého závaží.
- Hmotnosti $3^{n-1}+1, \dots, \frac{3^n-1}{2} = 3^{n-1}+\frac{3^{n-1}-1}{2}$ zvážíme tak, že n-té závaží dáme naproti předmětu a pomocí prvních $n-1$ závaží odvážíme zbytek hmotnosti.

Indukcí jsme ukázali, že s n závažími odvážíme všechny hmotnosti až do $m = \frac{3^n-1}{2}$. Jak jsme ukázali na začátku, lépe už to nejde. Nyní zbývá odpovědět na původní otázku: Kolik potřebujeme závaží, abychom odvážili všechny hmotnosti od 1 do m? Potřebujeme tolik závaží, aby $\frac{3^n-1}{2}$ bylo alespoň tak velké jako m. Matematicky to zapíšeme $n = \lceil \log_3(2m+1) \rceil$.

Petr Škoda

19-4-5 Hazardní hra

Hazardní hry nejsou pro slabé povahy, kdo na to nemá nervy a není si jistý, že vyhraje, ať raději zkusí své štěstí v kuličkách. Většina z vás správně vymyslela algoritmus, který Přesprstovi pomůže vyhrát, protože používá nejmenší možný počet operací k vyskládání požadované částky. Pravda, byli řešitelé, v jejichž uvažování se vyskytla chyba či dodali pouze útržek zdrojáku beze slova vysvětlení, ale takových byla menšina. Jakkoli tedy většina vymyslela optimální postup, skutečně bezchybných důkazů, že je optimální, se sešlo jako šafránu. Pojďme se tedy společně zamyslet nad správným řešením tak, aby to pochopil i detektiv Přesprst.

Označme si částku k vyskládání jako M. Nejprve si uvědomíme, že pro asymptoticky optimální řešení, tj. používající asymptoticky nejmenší počet operací, bychom si vystačili pouze s operacemi $+1$ a $\times 10$. Představme si číslo M zapsané v desítkové soustavě. Takový zápis bude mít $\mathcal{O}(\log M)$ cifer. Podíváme na hodnotu první cifry c_1 zleva a c_1-krát přičteme jedničku. Pak se posuneme na druhou cifru c_2 zleva, aktuální hodnotu vynásobíme deseti a c_2-krát přičteme jedničku, a tak dále. Zjevně po $\mathcal{O}(\log M)$ krocích dostaneme hledané číslo, a protože nad každou cifrou čísla musíme udělat alespoň jednu operaci (násobení 10), je náš postup asymptoticky optimální.

My bychom však chtěli přesně optimální řešení. K čemu nám v tom pomohou operace $/10$ a -1? Dělení desítkou nám v ničem nepomůže; dokážeme sporem. Mějme nějakou hodnotu h a uvažme nejkratší posloupnost operací, která vyskládá h a pro spor používá dělení. V okamžiku dělení má aktuální hodnota poslední cifru, na jejíž vyrobení jsme využili jistou posloupnost ostatních operací. Vydělíme-li nyní h deseti, přijdeme o hodnotu v poslední cifře, jinými slovy, operace vedoucí k jejímu vyrobení vůbec nebylo potřeba používat. Dostali bychom tedy kratší posloupnost operací vedoucí k vyskládání h, což je spor s tím, že jsme uvažovali nejkratší takovou posloupnost.

Ukážeme nyní lepší řešení úlohy a potom o něm dokážeme, že je optimální. Problém vyřešíme rekurzivně. Pro jednociferné M si rozmyslíme, že pro $0 \leq M \leq 5$ je výhodnější M-krát použít $+1$ a pro $6 \leq M \leq 9$ je lepší použít $+1, \times 10$ a potom $(10-M)$-krát -1.

Pro víceciferné M úlohu převedeme na úlohu s číslem o jednu cifru menším. Číslo M můžeme napsat ve tvaru $M = 10a+b$, kde b je poslední cifra napravo. Nyní použijeme stejnou myšlenku jako u jednociferného čísla. Je-li $0 \leq b \leq 5$, vyřešíme úlohu pro číslo a, pak použijeme $\times 10$ a následně b-krát $+1$. Je-li $6 \leq b \leq 9$, vyřešíme úlohu pro číslo $a+1$, potom použijeme $\times 10$ a následně $(10-b)$-krát -1.

Je však potřeba si rozmyslet, že uvedený algoritmus skutečně používá minimální možný počet operací. Tuto skutečnost si dokážeme napůl neformálně matematickou indukcí podle počtu cifer čísla M.

Pro jednociferné M je to zjevně optimální postup, to lze nahlédnout rozborem možností. Uvažme nyní víceciferné $M = 10a+b$, variantu $0 \leq b \leq 5$. Z indukčního předpokladu víme, že algoritmus použije pro vytvoření a minimální počet operací. Chceme-li za a přidat další cifru, nejúspornější je a vynásobit deseti a b-krát přičíst jedničku, zjevně se nevyplatí žádné odčítání ani vícenásobné násobení deseti.

Druhá varianta, $6 \leq b \leq 9$, je o něco záludnější. V nejhorším případě, pro $b = 6$, se vykoná 6 operací (přičtení, násobení a odečítání). Potíž je však v tom, že se rekurzivně voláme na vytvoření čísla $a+1$, a není jasné, co to udělá s optimálním počtem operací na vytvoření takového čísla. Tuto skutečnost mnoho řešitelů opomnělo.

Uvědomíme si, že minimální počet operací potřebných k vyrobení $a+1$ je nejhůře o jedna větší než minimální počet operací na vyrobení a. Končilo-li a na cifru menší než 5, náš algoritmus zjevně vypíše jen o jednu operaci navíc. Byla-li poslední cifra a mezi 6 a 8, zvýšíme ji tedy o jedna, čímž díky odečítající metodě ve skutečnosti počet operací pro $a+1$ dokonce snížíme. Poslední možnost je, že poslední cifra a byla 9. Tehdy se zvýší o jedničku následující cifra, čímž stejným argumentem (pouze o cifru dále) dostaneme, že přibude maximálně jedna operace navíc.

Tím pádem v nejhorším případě $b = 6$ se provede 7 operací, což je stejný počet, jako kdyby se místo odečítací metody použila přičítací metoda, pro $b > 6$ už si však pomůžeme.

Algoritmus stráví nad každou cifrou konstantní počet kroků a rekurze se zanoří do takové hloubky, kolik je cifer, časová i paměťová složitost tedy vyjdou $\mathcal{O}(\log M)$. Samotný program je napsán tak, že se vícenásobné přičítání a odčítání realizuje jako jedna úroveň rekurzivního volání hlavní procedury.

Tomáš Valla

19-4-6 Prolog

1. Lednice

Výprava do hlubin lednice skončila úspěšně. V prográmku na pár řádek skoro nešlo udělat chybu. Bylo si jen třeba uvědomit, kam umístit řez, aby při opakovaním volání nevznikaly nesmyslné počty potravin.

2. Myší spartakiáda

Myší spartakiáda byla také velmi populární. Někteří si ztížili zadání tím, že nejprve generovali celý seznam kombinací a poté jej vypisovali, ale protože úkolem je pouze vypsat všechny kombinace na výstup, můžeme to udělat takhle jednoduše: vygenerujeme celý obrazec, vypíšeme jej na obrazovku, zařízneme a přikážeme predikátu selhat pomocí `fail`. Prolog tedy snaživě zkusí vygenerovat nový obrazec, ten opět vypíšeme, selžeme, a takto nutíme Prolog hledat další obrazce, dokud neodpoví no. Mezitím jsou už ale všechny vypsány na obrazovku.

3. Myš v bludišti

Běda! O myším bludišti se ještě dlouho budou zdát noční můry nejen nešťastným programátorům, ale také zoufalým opravujícím, kteří se museli proplést spletitým bludištěm kódu. A to ještě nemluvíme o tom, že všechny myši by zešedivěly a sešly věkem, než by většina programů vydala výsledek. Přitom na první pohled vypadá úloha jednoduše, až nevinně.

Následující text obsahuje slova jako „zásobník“, „fronta“, „dfs“, „bfs“, „graf“ a podobné. Pokud nevíte, o čem je řeč, znovu vás odkazujeme na příslušnou kuchařku *http://ksp.mff.cuni.cz/tasks/19/cook3.html*.

Nejprve rozebereme chybu, kterou udělali skoro všichni. Nápadná poznámka „hledejte libovolně dlouhou cestu“ zcela oprávněně naváděla na to, k čemu je Prolog jako dělaný, k rekurzivnímu prohledávání. Správně jste si uvědomili, že si při prohledávání grafu musíte pamatovat již navštívené vrcholy. Ale málokoho napadlo, že při návratu z neúspěšné větve (tedy z takové, kde se nenašla cesta k cíli) se pilně nastřádaný seznam navštívených vrcholů opět odunifikovává a mizí. Naštěstí (pro vás) tato chyba neublíží funkčnosti, zato se ale stále dokola prochází vrcholy, ve kterých už jsme dávno byli a časová složitost roste až k exponenciální.

Jak z toho ven: Nesmíme implementovat průchod grafem až tak přímočaře, jak nás k tom svádí Prolog. Shodli jsme se na tom, že pamatovat si navštívené vrcholy je přímo životní nutnost, ale nesmíme o ně přicházet při návratu z neúspěšných větví. Tedy musíme to udělat tak, aby žádné neúspěšné větve nebyly, abychom nikdy „ne`fail`ovali“.

V predikátu cesta si budeme udržovat zásobník vrcholů a seznam navštívených vrcholů. V každém kroku vybereme vrchol z vrcholu zásobníku a najdeme všechny jeho nenavštívené sousedy. Tyto sousedy šoupneme na vrchol zásobníku a všechny označíme jako navštívené. Poté spustíme predikát cesta znovu s novým zásobníkem a novým seznamem navštívených vrcholů.

Tímto jsme se vlastně rozhodli nepoužít „vestavěnou“ prologovskou rekurzi, ale simulujeme si ji pomocí zásobníku. Proces skončí buď úspěšně, pokud se najednou na vrcholu zásobníku objeví cíl, nebo neúspěšně, pokud se zásobník vyprázdní, což by znamenalo, že jsme prozkoumali celý graf do hloubky a cestu k cíli jsme nenašli.

Navíc si stačí uvědomit, že pokud vyměníme použitý zásobník za frontu, změní se prohledávání z průchodu do hloubky na průchod do šířky a jako bonus dostaneme cestu nejkratší (za tu jsme ovšem žádné bonusové body neudělili, protože nejkratší cestu myši nepožadovaly).

Jenomže teď bychom potřebovali ještě určit, kudy cesta vede. Nemůžeme si ji budovat odzadu návratem z úspěšné větve, protože teď jsou všechny větve úspěšné a my nerozpoznáme, kdy se vracíme z úspěšné a kdy z neúspěšné větve a kdy si tedy máme cestu zapisovat. Můžeme si ale během výpočtu zapisovat pro každý vrchol jeho předchůdce a nakonec cestu odzadu zrekonstruovat.

4. Oprava

Chybu si uvědomíme, když dosadíme

```
?-minimum(1,2,2)
      Yes.
```

Co se v predikátu děje, vidíme jasně: Nejprve se pokusíme zunifikovat první řádek, což se nepodaří, skočíme tedy na druhý a ten projde. Řešení je například takovéto:

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

Nedovolili jsme Prologu ihned zunifikovat poslední argument, ale nejprve jsme mu zakázali použít další větev. Teprve potom smíme zunifikovat `X = Z`.

Jana Kravalová

Úloha 19-4-1 – Finanční toky – program

```
const MaxN = 1000;

var N : integer;
    Matice : array[1..MaxN, 1..MaxN] of boolean;                {Matice[i,j] znamená že vede hrana z i do j}
    Stok : integer;

procedure NactiMatici();
var i,j : integer;
    c : integer;
begin
   readln(N);
   for i := 1 to N do
      for j := 1 to N do begin
         read(c);
         Matice[i, j] := ( c = 1 );
      end;
end;

function OverKandidata(Kandidat : integer) : boolean;           {ověří, zda-je opravdu kandidát stokem}

var i : integer;
    Stok : boolean;
begin
   Stok := true;

   for i := 1 to N do                                           {vede hrana z kandidáta do ...}
      if Matice[Kandidat, i] then Stok := false;

   for i := 1 to N do                                           {vede do kandidáta hrana z ...}
      if (i <> Kandidat) and not(Matice[i, Kandidat]) then Stok := false;

   OverKandidata := Stok;
end;
```



```
function NajdiKandidata():integer;                                          {najde kandidáta na stok}

var i,Kandidat:integer;
begin
   Kandidat := 1;

   for i := 2 to N do
      if Matice[Kandidat, i] then Kandidat := i;

   NajdiKandidata := Kandidat;
end;

begin
  NactiMatici();
  Stok := NajdiKandidata();
  if OverKandidata(Stok) then writeln('Stok je ve vrcholu ',Stok,'.')
  else writeln('Stok neexistuje.');
end.
```

Úloha 19-4-2 – Byrokratický aparát – program

```
#include <stdio.h>

#define MAXN 1000

int next[MAXN], was[MAXN]={0};
int cycles[MAXN]={0};
int cyc_count=0;
int N;

int nsd(int a, int b)                                   //Zjištění největšího společného dělitele
{                                                       //pomocí Euklidova algoritmu
        while (b)
        {
                a = a%b;                                //a sice může být menší než b, ale to nevadí
                if (a) return b;
                b = b%a;
        }
        return a;

}

int main()
{
        scanf("%d", &N);                                //načtení vstupu
        for (int i=0; i<N; ++i)
                scanf("%d", &next[i]);

        int cur, cyc_len;                               //zjištění délek jednotlivých cyklů
        for (int i=0; i<N; ++i)
        {
                if (!was[i]){                           //Pokud vrchol ještě nenavštívil
                        cur = i;                        //zahájí v něm procházení celého cyklu
                        while(!was[cur])
                        {
                                ++cycles[cyc_count];
                                was[cur] = 1;
                                cur      = next[cur]-1;
                        }
                        ++cyc_count;
                }
        }

        int nsn = cycles[0];
        for (int i=1; i<cyc_count; ++i)                 //z NSN K prvků spočte
        {                                               //NSN K+1 prvků
                nsn = nsn/nsd(nsn, cycles[i]) * cycles[i];
        }
        printf("A výsledek je :%d", nsn);
        return 0;
}
```