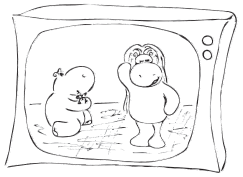
Milí řešitelé a řešitelky!



Přinášíme vám již poslední sérii tohoto ročníku. Nastává finále, které všechno rozhodne. Přejeme příjemnou zábavu a hodně štěstí!

Svá řešení posílejte do pondělí 31. 5. 2009 elektronicky na *http://ksp.mff.cuni.cz/submit/*, nebo klasickou poštou na známou adresu:

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

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*.

Pátá série dvacátého prvního ročníku KSP

Píše se rok 2050. Konečně se podařilo rozpustit poslední zbytky ledovců a hladiny světových oceánů se netriviálně zvedly. Hlavní město hlavního telenovelového státu se začalo potýkat s problémy. A právě v těchto časech začíná příběh našeho hlavního hrdiny Chulia.

21-5-1 Polomáčené mrakodrapy — 10 bodů

Na hlavní město našeho státu – Chuenos Aires – se valí pohroma. Chulianovo rodné město, dříve tak prosperující metropole nudlovitého tvaru, bude brzy zatopeno stoupající vodou z oceánů. Úředníci vlády teď nutně potřebují vědět, kolik úseků města si bude i během záplav moci naladit pravidelnou večerní telenovelu.

Město Chuenos Aires si můžeme představit jako jednu dlouhou ulici, na které je namačkán mrakodrap vedle mrakodrapu. Mrakodrapy jsou tak natěsno, že nemají mezi sebou žádné mezery. Každý mrakodrap (s plochou střechou) má kladnou celočíselnou výšku měřenou v telemetrech nad mořem. Chodník má výšku právě 0 telemetrů nad mořem.

Předpověď počasí hlasí, že zatím jsou všechny mrakodrapy v pořádku, ale počínaje dnem 1 se výška moře zdvihne o jeden telemetr denně. Postupem času se některé věžáky zatopí až po špičku a z hlavní ulice v Chuenos Aires zbudou pouze souvislé úseky mrakodrapů, které ční nad hladinu. Každému takovému úseku stačí jedna anténa na přijímání televize. Vaším úkolem je spočítat, kolik antén bude v jednotlivých dnech potřeba (tedy kolik zbude souvislých bloků mrakodrapů v onen den).

Tato úloha je praktická, takže bližší informace o formátu vstupu programu i ukázkový příklad najdete v CodExu. Povídání o tom, jak se praktická úloha odevzdává, najdete například v zadání úlohy 21-1-2 „Optimalizace kotlů“.

Když i nejvyšší mrakodrap v Chuenos Aires, ve kterém Chulio bydlel, začal být těžko obyvatelný, protože jeho obyvatelé museli každé ráno vyhánět z postele žraloky, rozhodl se Chulio odjet na venkov, kde by začal žít nový život.

A jak se rozhodl, tak udělal. Odešel z města a začal pracovat na kávové plantáži. Tam se brzy seznámil s krásnou Ochechulínou. Začali snít o tom, že se jednou vezmou a sami budou vlastnit podobnou plantáž, každé ráno budou vstávat se šálkem kávy a úsměvem nad dalším dnem. Bohužel zloduch Choachým jim každé snění zkazil, neboť je stále budil před ránem a nutil pracovat.

Z Choachýma se stal úhlavní nepřítel a Chulio s Ochechulínou začali snovat plán pomsty.

Jejich plán byl geniální. Propracovaný do nejmenšího detailu. Skoro. Neměli na něj peníze. A protože prací ještě nikdo nezbohatl, začali vymýšlet, jak na to. Naštěstí bankovní servery v té době ne zcela plánovaně chlazené mořskou vodou vykazovaly o něco vyšší chybovost, a tak stačilo jen trocha šikovnosti a investice ve výši 10 pesos k odlehčení kont nenasytných bankéřů.

21-5-2 Banky — 10 bodů

Aby se podobné odlehčování nekonalo příliš často, nebo pokud možno už nikdy, požádali vás nenasytní bankéři, abyste jim pomohli. Jak vlastně Chulio zbohatnul? Banka obchoduje valutami a má vypsané kurzy pro některé dvojice měn. Banka je hodná a za směnu si neúčtuje žádný poplatek. Pokud ale není dostatečně opatrná, může se stát, že vhodnou posloupností směn získáte více, než jste měli na začátku. Váš program tedy dostane na vstupu směnný kurz banky a měl by rozhodnout, zda je na jeho základě možné zbohatnout výše popsaným způsobem.

Například pro 4 měny a kurzy (zápis $X \to YZ$ znamená, že za 1 jednotku měny X získáte Z jednotek měny Y):

```
1->2 0.8        2->3 24
2->1 0.8        2->4 129
1->4 0.08       3->1 0.5
```

lze např. posloupností výměn `1->2`, `2->3` a `3->1` (za jednu jednotku měny 1 získáte po této sérii výměn 9,6 jednotek) na úkor banky vydělat. Takže v tuto chvíli by měl program odpovědět, že směnný kurz je prodělečný. Pokud žádná taková posloupnost neexistuje, váš program by měl odpovědět, že banka bude opět o něco bohatší.

Chulio s Ochechulínou skoupili celou plantáž a mohli dokonat svůj dokonale zosnovaný plán – Choachýma přeřadili na tu nejhorší práci: ochutnávač kávy. Je jasné, že Choachýmovi tato práce nedovolila pořádně se vyspat, takže nedostatkem spánku psychicky narušený Choachým začal vymýšlet svůj plán pomsty.

Chulio a Ochechulína vedli šťastný život. Zcela se jim splnil sen. To ale netrvalo dlouho. Bratr Chulia Chosé se dozvěděl o Chuliově úspěchu a přicestoval za ním. Hodný Chulio zaměstnal Chosého a pověřil ho úkolem vybudovat vedle plantáží farmu pro dobytek.

21-5-3 Krávy — 9 bodů

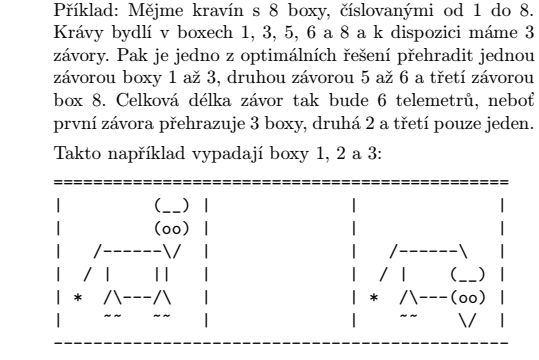
Navrhovat stáje pro krávy není vůbec jednoduchý úkol. Navíc máte-li omezené množství prostředků. Chuliův kravín se skládá z řady boxů stejných rozměrů těsně vedle sebe. V každém boxu může bydlet nejvýše jedna kráva. Tyto boxy jsou z jedné (stejné) strany vždy otevřené. Vaším úkolem je rozmístit před tyto boxy závory boxy tak, aby každý, ve kterém bydlí kráva, byl přehrazený.

Závor ale máte k dispozici pouze určitý počet. Naštěstí každá závora může být libovolně dlouhá a není zakázáno přehradit i boxy, ve kterých žádná kráva nebydlí.

Navrhněte algoritmus, který pro zadaný kravín (počet boxů a rozmístění krav v jednotlivých boxech) nalezne takové rozmístění závor, aby součet jejich délek (jeden box má šířku jeden telemetr) byl minimální a všechny boxy s kravami byly přehrazeny. Pokud má úloha více řešení, stačí nalézt libovolné z nich.

Příklad: Mějme kravín s 8 boxy, číslovanými od 1 do 8. Krávy bydlí v boxech 1, 3, 5, 6 a 8 a k dispozici máme 3 závory. Pak je jedno z optimálních řešení přehradit jednou závorou boxy 1 až 3, druhou závorou 5 až 6 a třetí závorou box 8. Celková délka závor tak bude 6 telemetrů, neboť první závora přehrazuje 3 boxy, druhá 2 a třetí pouze jeden.

Takto například vypadají boxy 1, 2 a 3:



Chosé byl ze začátku spokojený, a tak navrhoval stáje, obdělával pastviny, kupoval dobytek a farma vzkvétala. Jenže brzy začal pociťovat závist, protože dřel na svého bratra, který jenom popíjel kávu a trávil čas s krásnou Ochechulínou, po které Chosé stále více toužil.

Ochechulínu to uvádělo do rozpaků, ale příliš se návrhům Chosého nebránila, neboť narozdíl od Chulia neměl hrb, měl obě ruce stejně dlouhé a také měl delší . . . vlasy.

Ani Choachým během probdělých dní a nocí nemarnil čas a stále promýšlel svůj plán pomsty. Protože zbohatnout stejně jako Chulio již nebylo možné, chtěl k pomstě využít to, čeho měl nejvíc. Šarm. Rozhodl se svést Chulia, následně jej obžalovat z obtěžování a potom převzít vládu nad farmou i Ochechulínou.

Plán se mu podařil a Chulio skončil se zlomeným srdcem u soudu. Zatímco se Chulio pokoušel prokličkovat ve spleti zákonů, Choachým seč mohl pospíchal na farmu.

21-5-4 Zákony — 12 bodů

Obhájit se u soudu bývá podobné jako vymotat se z hustého a temného lesa. Všude kolem jsou stromy, totiž jednotlivé zákony, a čím těžší přestupek jste spáchali, tím obtížnější je se mezi nimi prosmyknout a proniknout z právního lesa zpět na svobodu.

Napište program, který na vstupu dostane popis spleti zákonů a velikost vašeho přestupku. Program pak odpoví, zda je možné se u soudu obhájit nebo ne.

Program dostane dvě čísla: přirozené N a kladné reálné R. Číslo N udává počet zákonů a R velikost vašeho hříchu. Dále dostane vaše výchozí souřadnice v lese X a Y a popis jednotlivých stromů (zákonů). Každý strom je zadán svými souřadnicemi. Všechny souřadnice jsou reálná čísla.

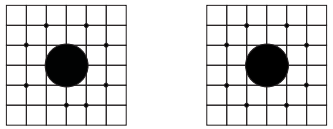
Hodnota R určuje, jak nejvíc se na cestě ven z lesa můžete přiblížit k libovolnému stromu. Předpokládejte, že stromy mají nulový průměr a jsou nekonečně vysoké a že výchozí pozice je vždy korektní, tzn. že na začátku nejste v kolizi s žádným stromem.

Úlohu lze formulovat i tak, že zjišťujete zda se koule o poloměru R může vykulálet ven ze zadaného lesa.

A co znamená dostat se ven z lesa? Třeba mít možnost dostat se do bodu se souřadnicemi (∞, ∞).

Například pro $N = 8$, $R = 1,1$, $X = 0$, $Y = 0$ a stromy $(-2, 1)$, $(-1, 2)$, $(1, 2)$, $(2, 1)$, $(2, -1)$, $(1, -2)$, $(0, -2)$, $(-2, -1)$ se z lesa dostat lze.

Pro $N = 8$, $R = 1.1$, $X = 0$, $Y = 0$ a stromy $(-2, 1)$, $(-1, 2)$, $(1, 2)$, $(2, 1)$, $(2, -1)$, $(1, -2)$, $(-1, -2)$, $(-2, -1)$ se z lesa dostat nejde.



Choachým dorazil, právě když vládu nad Ochechulínou přebíral Chosé, využívaje k tomu svých dlouhých vlasů. Takové setkání prostě muselo skončit soubojem.

Chosé proklál Choachýma svojí cestovní šavlí.

21-5-5 Cestovní šavle — 10 bodů

Určitě znáte klasický zednický metr. Ten se skládá z několika segmentů. Vždy dva sousední segmenty jsou spojeny pantem, takže je metr možné rozložit, pokud měříme nějakou vzdálenost, nebo složit, pokud ho přenášíme. Přesně tímto způsobem funguje Chosého cestovní šavle. Jenom s tím rozdílem, že délky segmentů nejsou stejné.

Napište program, který pro zadané délky jednotlivých segmentů šavle zjistí, zda je možné šavli poskládat do pouzdra zadané délky. Program dostane dvě přirozená čísla N a L. Číslo N je rovno počtu segmentů šavle a L je délka pouzdra. Následuje N přirozených čísel, které postupně udávají délky segmentů šavle zleva doprava. Můžete předpokládat, že N i $L \leq 10000$. Program by měl odpověď „Ano“ nebo „Ne“ podle toho, zda je možné šavli složit do pouzdra.

Například pro $N = 3$, $L = 6$ a délky $6, 3, 3$ je možné šavli poskládat. Pro stejné N i L, ale pro délky $6, 3, 4$ šavli poskládat nelze.

Mezitím se od soudu vrátil Chulio. Když se dozvěděl, co se tu stalo, využil toho, že se Chosému stále nedařilo složit šavli zpět do pouzdra a proklál s ní i svého proradného bratra. Od té doby již žil Chulio s Ochechulínou šťastně až do konce . . . až do konce 14223. dílu, kdy se ukázalo, že předchozích 14221 dílů bylo jenom snem a Chulio s Ochechulínou se probouzí do dalšího pracovního dne na farmě.

21-5-6 Kržnc — 12 bodů

Ke konci spěje nejen tento ročník KSP, ale i náš seriál o programování na počítači Rapl (připomínáme, že seriálovou úlohu je nutné řešit ve speciálním programovacím jazyku, jehož popis najdete v zadání úlohy 21-1-6). Už jsme si vyzkoušeli psát co nejkratší programy, ba i programy, které za běhu použijí co nejméně instrukcí. Na závěr zkusíme, jak šetřit paměť, kterou program potřebuje k výpočtu.

Mějme nějaký neorientovaný graf (viz kuchařka u 3. série), jehož každý vrchol sousedí s právě čtyřmi hranami. Takový graf můžeme Raplu popsat následovně: očíslujeme si vrcholy od 0 do `n-1` a do pole `S` naskládáme čísla sousedů jednotlivých vrcholů tak, že sousedé vrcholu `i` budou uloženi v `S[4*i]`, `S[4*i+1]`, `S[4*i+2]` a `S[4*i+3]`.

```
var
  n: integer;

begin
  read(n);
  rozloz(n, 0, n);
end.
```

Výsledková listina dvacátého prvního ročníku KSP po třetí sérii

		škola	*ročník*	*sérií*	*2131*	*2132*	*2133*	*2134*	*2135*	*2136*	*série*	*celkem*
1.	Vítězslav Plachý	GJiříPoděb	3	3	4	2	6	4	10	12	37,0	118,3
2.	Vojtěch Kolář	G Neratov	3	9	4		8	9	10	12	39,8	114,9
3.	Filip Hlásek	GMikul23PL	2	8		4	8		10	10	34,2	113,2
4.	Petr Čermák	GEBenešeKL	3	3			2	3	10	12	31,5	109,0
5.	Michal Bilanský	GLepařovJČ	3	3		1	6		10	10	31,7	108,4
6.	Jiří Cidlina	GVoděraPH	4	3	0	10	4	0	10	10	37,9	106,7
7.	Jitka Novotná	G Bílovec	4	5	4,5		6	8	10	12	39,3	100,5
8.	Pavel Veselý	G Strakon	4	15	7	2	8		10	12	35,2	95,2
9.	Lukáš Ptáček	GJAKŽeliez	3	3		1			10	8	22,6	92,5
10.	Karel Tesař	SPŠE Plzeň	3	5		3	10		10	9	35,3	86,4
11.	Vlastimil Dort	GŠpitálsPH	3	13			8		10	12	29,6	72,7
12.	David Věčorek	GTNovákBO	3	3	0		5			8	17,9	72,6
13.	Filip Štědronský	GMikul23PL	2	8	11				10	12	33,0	71,9
14.	Alexander Mansurov	GNVPlániPH	0	2			6		10	12	30,4	70,1
15.	Martin Zikmund	G Turnov	1	3	0				10		10,0	65,1
16.	Jan Vaňhara	G Holešov	4	3							0,0	56,2
17.	Pavel Taufer	ArcibisGPH	3	5					10	8	19,9	54,3
18.	Barbora Janů	GKepleraPH	2	3					10		10,0	44,3
19.	Jiří Setnička	G25březnPH	2	4							0,0	42,2
20.	Alžběta Pechová	SPŠSVsetín	4	6		1		5	4	4	20,1	39,4
21.	Petr Pecha	SPŠSVsetín	2	4	4,5				0		6,5	38,6
22.	Filip Sládek	GNámestovo	3	1							0,0	38,1
23.	Karel Kolář	GŠpitálsPH	4	3	1	3				12	19,6	36,3
24.	Libor Plucnar	GBezručeFM	4	10							0,0	36,2
25. – 26.	Tomáš Pikálek	GBoskovice	2	1							0,0	35,6
	Jan Veselý	G Strakon	2	1							0,0	35,6
27.	Lukáš Chmela	GJŠkodyPŘ	0	1							0,0	35,2
28.	Ondřej Pelech	GJNerudyPH	4	1							0,0	33,4
29.	Karel Král	G Most	3	3	8						8,7	29,3
30.	Štěpán Šimsa	GJungmanLT	0	2							0,0	29,2
31.	David Formánek	GJarošeBO	2	2							0,0	27,9
32.	Karolína Burešová	G ČesLípa	2	1							0,0	27,7
33.	Milan Rybář	GJungmanLT	4	3							0,0	27,4
34.	Petr Zvoníček	G Slavičín	3	2							0,0	26,5
35.	Honza Žerdík	G Příbor	4	1							0,0	25,7
36.	Radim Cajzl	GNoMěsNMor	2	18						8	5,4	25,1
37.	Jan Škoda	GMikul23PL	2	4							0,0	23,3
38.	Alena Bušáková	G Trutnov	2	1							0,0	22,6
39.	Kateřina Lorenzová	G Česká ČB	2	3			3				5,4	19,7
40.	Hynek Jemelík	GJarošeBO	2	2							0,0	19,0
41.	Jakub Sochor	G Bílovec	4	1							0,0	17,1
42.	Martin Holec	G Slavičín	2	3						2	4,1	15,5
43.	Mirek Jarolím	GMikul23PL	3	4							0,0	15,3
44.	David Vondrák	GDašickáPA	3	2							0,0	14,8
45. – 46.	Petr Babička	VOŠGSvetla	4	7					2		3,0	10,2
	Pavel Kratochvíl	VOŠGSvetla	1	6					2		3,2	10,2
47. – 48.	Jiří Daněk	GKřenováBO	3	1							0,0	10,0
	Jan Matějka	G Jírov ČB	4	5					10		10,0	10,0
49.	Dominik Smrž	GOhradníPH	0	3							0,0	8,7
50.	Jiří Keresteš	SPŠE Plzeň	3	5							0,0	7,4
51.	Stanislav Fořt	GCoubTábor	1	8							0,0	5,9
52.	Igor Koníček	G UherBrod	3	1							0,0	5,7
53.	Ladislav Maxa	GKepleraPH	3	1							0,0	5,5
54.	Jan Kostecký	VOŠŠumperk	2	1							0,0	4,5

Bude nás zajímat, zda v tomto grafu existuje kružnice, která obsahuje každý vrchol právě jednou. Napište program v RAPLu, který pro libovolný zadaný graf odpoví v registru `k` jedničkou, pokud taková kružnice existuje, a jinak nulou.

Váš program by měl použít co nejméně paměti. Množství použité paměti přitom definujeme jako počet paměťových buněk (registrů či prvků polí), do kterých program během výpočtu zapsal. Pokud tedy vstup pouze čtete a nepřepisujete, jeho velikost se nepočítá. Záleží nám i na multiplikativních konstantách – mezi $2n$ a $10n$ buněk je podstatný rozdíl. Naproti tomu nemusíte vůbec brát ohled na časovou složitost, může být klidně megasuperexponenciální :-)

Při programování si tentokrát můžete odpustit nezajímavé detaily, stačí nám, když dostatečně výstižně popíšete, co přesně do paměti jak uložíte a jak s tím budete zacházet.

[+2 body] Ukažte, jak úlohu vyřešit s méně než $n/19$ paměťovými buňkami, pokud víte, že každý vrchol sousedí jen s třemi hranami.

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

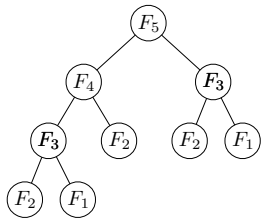
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

1 1 2 3 5 8 13 21 34 55 89 ...

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

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

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



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

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

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

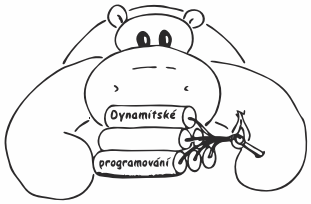
Tedy na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme třeba využít toho, že:

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

z čehož plyne:

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

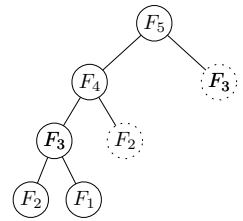
Funkce `Fibonacci` má tedy exponenciální časovou složitost, což není nic vítaného. Ovšem jak jsme už řekli, některé výpočty opakujeme stále dokola. Nenabízí se proto nic snazšího, než si tyto mezivýsledky uložit a pak je vytáhnout jako pověstného králíka z klobouku s minimem námahy.



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

```
var P: array[1..MaxN] of Integer;
function Fibonacci(n: Integer): Integer;
begin
  if P[n] = 0 then
  begin
    if n <= 2 then
      P[n] := 1
    else
      P[n] := Fibonacci(n-1) + Fibonacci(n-2)
  end;
  Fibonacci := P[n]
end;
```

Podívejme se, jak vypadá strom volání nyní:



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

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

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

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

Zopakujme si, co jsme postupně udělali: nejprve jsme vymysleli pomalou rekurzivní funkci, tu jsme zrychlili zapamatováváním si mezivýsledků a nakonec jsme celou rekurzi „obrátili naruby“ a mezivýsledky počítali od nejmenšího k největšímu, aniž bychom se starali o to, jak se na ně původní rekurze ptala.

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

Problém batohu

Je dáno N předmětů o hmotnostech $m_1, \ldots, m_N$ (celočíselných) a také číslo M (nosnost batohu). Úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, a přitom nepřekročil M. Předvedeme si 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 bude prvek $A[i]$ nenulový právě tehdy, jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i. 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, jak kroky algoritmu odpovídají podúlohám, které řešíme: v prvním kroku vyřešíme podúlohu tvořenou jen prvním předmětem, ve druhém kroku prvními dvěma předměty, pak prvními třemi předměty, atd.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč zrovna na k). Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k 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 hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k-1$ předmětů, jejíž hmotnost je $i - m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i. Naopak, pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k-1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X. Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ 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: protože v k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, tak v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu, atd. Zdrojový kód tohoto 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;
  writeln;
end.
```

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


```
  // Teď to zajímavé. Najdeme si nejdelší cestu.
  // Zavolat rekurzivní fci. zakořeněnou v 0. vrcholu a s neexistujícím rodičem
  path_search(0, tree_size, 0);
  // Vyplnit cestu
  unsigned path_len = vertices[0].longest[0].subtree;
  unsigned *path = malloc((path_len + 1) * sizeof *path);
  path_fill(0, 0, path);
  // Co jsou konce cesty? Budeme z nich znovu hledat.
  unsigned a = path[0], b = path[path_len];
  // Spočítat velikosti cest ve zbylých stromech
  path_search(a, tree_size, 0);
  path_search(b, tree_size, 1);
  // Která hrana je ta nejlepší?
  unsigned best;
  unsigned best_val = tree_size + 1; // Něco dostatečně velkého
  for(unsigned i = 0; i < path_len; i ++)
  {
    unsigned left = vertices[path[i]].longest[1].subtree;
    unsigned right = vertices[path[i + 1]].longest[0].subtree;
    unsigned trough = (left + 1) / 2 + (right + 1) / 2 + 1;
    unsigned total = (left > right) ? left : right;
    total = (total > trough) ? total : trough;
    // Tahle hrana je lepší
    if(total < best_val)
    {
      best_val = total;
      best = i;
    }
  }

  // Načíst nejlepší hranu
  unsigned e[] = { path[best], path[best + 1] };
  free(path); // Zbytek cesty není potřeba
  printf("Odebrat: %zu-%zu\n", e[0] + 1, e[1] + 1);
  // Použít už spočítané cesty k nalezení středů, spojit
  for(unsigned i = 0; i < 2; i ++) {
    unsigned path_len_short = vertices[e[i]].longest[1 - i].subtree;
    unsigned *path_short = malloc((path_len_short + 1) * sizeof *path_short);
    // Zkopírovat cestu (nebyla by potřeba celá, ale takto je to méně práce a může se hodit při ladění)
    path_fill(e[i], 1 - i, path_short);
    // Načíst novou, optimálnější hranu
    e[i] = path_short[path_len_short / 2];
    free(path_short);
  }
  printf("Přidat: %zu-%zu\n", e[0] + 1, e[1] + 1);
  // Zrušit graf
  free(edges);
  free(vertices);
  return EXIT_SUCCESS;
}
```

Úloha 21-3-5 – Rozklad na součty – program

```
program Rozklad;
const MAX_N = 40;
var
  scitance: array [1..MAX_N] of integer; { Globální pole, do kterého si rekurze připravuje sčítance. }

{ Rozloží číslo n na sčítance, mezi kterými jsou jen čísla od 1 do max. }
{ Číslo i představuje počet již hotových čísel v poli scitance. }
procedure rozloz(n, i, max: integer);
var
  j: integer;
begin
  if n = 0 then begin
    { Máme připraveny všechny sčítance, vypíšeme je (konec rekurze). }
    for j := i downto 1 do begin
      if j < i then write('+');
      write(scitance[j]);
    end;
    writeln;
  end else begin
    { Provedeme další krok rekurze. }
    if n < max then max := n;
    for j := 1 to max do begin
      scitance[i+1] := j;
      rozloz(n-j, i+1, j);
    end;
  end;
end;
```

```
    // Je to zajímavé? Je tam něco dostatečně dlouhého?
    if(vertices[vertex].longest[result].subtree > longest_subtree) {
      subtree_vertex = vertices[vertex].longest[result].vertex;
      longest_subtree = vertices[vertex].longest[result].subtree;
    }
    for(int j = 1; j >= 0; j --)
      if(vertices[vertex].longest[result].local + 1 >= local_lengths[j]) {
        longest_neigh[j + 1] = longest_neigh[j];
        local_lengths[j + 1] = local_lengths[j];
        longest_neigh[j] = vertex;
        local_lengths[j] = vertices[vertex].longest[result].local + 1;
      }
  }
  // Vyplnit sebe z toho, co se spočítalo
  memcpy(vertices[index].longest[result].neighbors, longest_neigh, 2 * sizeof *longest_neigh);
  vertices[index].longest[result].local = local_lengths[0];
  // Nejdelší prochází skrz mě
  if(local_lengths[0] + local_lengths[1] > longest_subtree) {
    vertices[index].longest[result].subtree = local_lengths[0] + local_lengths[1];
    vertices[index].longest[result].vertex = index;
  } else { // Nejdelší v některém podstromu
    vertices[index].longest[result].subtree = longest_subtree;
    vertices[index].longest[result].vertex = subtree_vertex;
  }
}

// Vytáhne nejdelší cestu v subtree z grafu do output
static void path_fill(unsigned subtree, unsigned result, unsigned *output) {
  // Najdi nejvyšší vrchol na cestě
  unsigned vertex = vertices[subtree].longest[result].vertex;
  unsigned length = vertices[subtree].longest[result].subtree;
  unsigned index = vertices[vertex].longest[result].local;
  // Umísti ho na správné místo
  output[index] = vertex;
  // Projdi oba konce cesty odsud
  for(unsigned i = 0, pos = vertices[vertex].longest[result].neighbors[0]; i < index;
                i ++, pos = vertices[pos].longest[result].neighbors[0])
    output[index - 1 - i] = pos;
  for(unsigned i = index + 1, pos = vertices[vertex].longest[result].neighbors[1]; i <= length;
                i ++, pos = vertices[vertex].longest[result].neighbors[0])
    output[i] = pos;
}

int main(int argc, char *argv[]) {
  // Jen otravné načítání souboru
  FILE *f = fopen("optstro.in", "rt");
  fscanf(f, "%zu", &tree_size);
  if(tree_size < 2)
  {
    fprintf(stderr, "Nemá hrany, nelze optimalizovat\n");
    return EXIT_FAILURE;
  }
  // Jeden navíc - zarážka
  vertices = malloc((tree_size + 1) * sizeof *vertices);
  for(unsigned i = 0; i < tree_size; i ++)
    vertices[i].edges = 0;
  // O jednu hranu méně než vrcholů
  edges_raw = malloc((tree_size - 1) * sizeof *edges_raw);
  // Načíst hrany, spočítat, kolik patří kterému vrcholu
  for(unsigned i = 0; i < tree_size - 1; i ++)
    // Každá hrana má 2 konce, uložme si je
    for(unsigned j = 0; j < 2; j ++) {
      fscanf(f, "%zu", &edges_raw[i][j].vertex);
      edges_raw[i][j].vertex --; // Soubor je od 1, v programu indexujeme od 1 od 0
      edges_raw[i][j].index = vertices[edges_raw[i][j].vertex].edges ++;
    }
  fclose(f);
  // Zařadit hrany, kam patří, připravit vrcholy
  unsigned current_pos = 0;
  for(unsigned i = 0; i <= tree_size; i ++) {
    unsigned size = vertices[i].edges;
    vertices[i].edges = current_pos;
    current_pos += size;
  }
  // Každá hrana je tu 2* - jednou tam, jednou zpět
  edges = malloc((tree_size - 1) * 2 * sizeof *edges);
  for(unsigned i = 0; i < tree_size - 1; i ++)
    for(unsigned j = 0; j < 2; j ++)
      edges[vertices[edges_raw[i][j].vertex].edges + edges_raw[i][j].index] = edges_raw[i][1 - j].vertex;
  // Neroztříděné hrany už nejsou potřeba
  free(edges_raw);
```


Nejkratší cesty a Floydův-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 (nezáporné) 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 nejkratších cest mezi všemi dvojicemi měst. *Cestou* rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojené silnicí, a délka cesty je součet délek silnic, které tato města spojují. [V grafové terminologii tedy máme daný ohodnocený neorientovaný graf a chceme zjistit délky nejkratších cest mezi všemi dvojicemi jeho vrcholů.]

Půjdeme na to následovně: Vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu). V průběhu výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

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 algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j. Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $\mathcal{O}(N^2)$. Celková časová složitost našeho algoritmu tedy je $\mathcal{O}(N^3)$. Co se paměti týče, vystačíme si s polem D a to má velikost $\mathcal{O}(N^2)$. Program bude vypadat následovně:

```
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 nejkratší cesty mezi nimi. 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.

Nejdelší společná podposloupnost

Poslední příklad dynamického programování, který si předvedeme, se bude týkat posloupností. Mějme dvě posloupnosti čísel A a B. Chceme najít jejich nejdelší společnou podposloupnost, tedy takovou posloupnost, kterou můžeme získat z A i B odstraněním některých prvků. Například pro posloupnosti

$$A = 2\ 3\ 3\ 1\ 2\ 3\ 2\ 2\ 3\ 1\ 1\ 2$$
$$B = 3\ 2\ 2\ 1\ 3\ 1\ 2\ 2\ 3\ 3\ 1\ 2\ 2\ 3$$

je jednou z nejdelších společných podposloupností tato posloupnost:

$$C = 2\ 3\ 1\ 2\ 2\ 3\ 1\ 2.$$

Jakým způsobem můžeme takovou podposloupnost najít? Nejdříve nás asi napadne vygenerovat všechny podposloupnosti a ty pak porovnat. Jakmile si ale spočítáme, že všech podposloupností posloupnosti o délce n je 2^n (každý prvek nezávisle na ostatních buď použijeme, nebo ne), najdeme raději nějaké rychlejší řešení.

Zkusme využít následující myšlenku: vyřešíme tento problém pouze pro první prvek posloupnosti A. Pak najdeme řešení pro první dva prvky A, přičemž využijeme předchozích výsledků. Takto pokračujeme pro první tři, čtyři, ... až n prvků.

Nejprve si rozmyslíme, co všechno si musíme v každém kroku pamatovat, abychom z toho dokázali spočíst krok následující. Určitě nám nebude stačit pamatovat si pouze nejdelší podposloupnost, jenže množina všech společných podposloupností je už zase moc velká. Podívejme se tedy detailněji, jak se změní tato množina při přidání dalšího prvku k A: Všechny podposloupnosti, které v množině byly, tam zůstanou a navíc přibude několik nových, končících právě přidaným prvkem. Ovšem my si podposloupnosti pamatujeme proto, abychom je časem rozšířili na nejdelší společnou podposloupnost, takže pokud známe nějaké dvě stejně dlouhé podposloupnosti P a Q končící nově přidaným prvkem v A a víme, že P končí v B dříve než Q, stačí si z nich

pamatovat pouze P, jelikož v libovolném rozšíření Q-čka můžeme Q vyměnit za P a získat tím stejně dlouhou společnou podposloupnost.

Proto si stačí pro již zpracovaných a prvků posloupnosti A pamatovat pro každou délku l tu ze společných podposloupností $A[1 \ldots a]$ a B délky l, která v B končí na nejlevějším možném místě, a dokonce nám bude stačit si místo celé podposloupnosti uložit jen pozici jejího konce v B. K tomu použijeme dvojrozměrné pole $D[a, l]$.

Ještě si dovolíme jedno malé pozorování: Koncové pozice uložené v poli D se zvětšují s rostoucí délkou podposloupnosti, čili $D[a, l] < D[a, l + 1]$, protože posloupnosti délky $l+1$ nejsou ničím jiným než rozšířeními posloupností délky l o 1 prvek.

Teď již výpočet samotný: Pokud už známe celý a-tý řádek pole D, můžeme z něj získat $(a + 1)$-ní řádek. Projdeme postupně posloupnost B. Když najdeme v B prvek $A[a+1]$ (ten právě přidávaný do A), můžeme rozšířit všechny podposloupnosti končící před aktuální pozicí v B. Nás bude zajímat pouze ta nejdelší z nich, protože rozšířením všech kratších získáme posloupnost, jejíž koncová pozice je větší než koncová pozice některé posloupnosti, kterou již známe. Rozšíříme tedy tu nejdelší podposloupnost a uložíme ji místo původní podposloupnosti. Toto provedeme pro každý výskyt nového prvku v posloupnosti B. Všimněte si, že nemusíme procházet pole s podposloupnostmi stále od začátku, ale můžeme se v něm posouvat od nejmenší délky k největší.

Ukážeme si, jak vypadá zaplněné pole hodnotami při řešení problému s posloupnostmi z našeho příkladu. Řádky jsou pozice v A, sloupce délky podposloupností.

D	1	2	3	4	5	6	7	8	9	10	11	12
1	2	–	–	–	–	–	–	–	–	–	–	–
2	1	5	–	–	–	–	–	–	–	–	–	–
3	1	5	9	–	–	–	–	–	–	–	–	–
4	1	4	6	11	–	–	–	–	–	–	–	–
5	1	2	5	7	12	–	–	–	–	–	–	–
6	1	2	3	7	9	14	–	–	–	–	–	–
7	1	2	3	7	8	12	–	–	–	–	–	–
8	1	2	3	7	8	12	13	–	–	–	–	–
9	1	2	3	5	8	9	13	14	–	–	–	–
10	1	2	3	4	6	9	11	14	–	–	–	–
11	1	2	3	4	6	9	11	14	–	–	–	–
12	1	2	3	4	6	7	11	12	–	–	–	–

Zbývá popsat, jak z těchto dat zvládneme rekonstruovat hledanou nejdelší společnou podposloupnost (NSP). Ukážeme si to na našem příkladu: jelikož poslední nenulové číslo na posledním řádku je ve 12. sloupci, má hledaná NSP délku 12. $D[12, 8] = 12$ říká, že poslední písmeno NSP je na pozici 12 v posloupnosti B. Jeho pozici v posloupnosti A určuje nejvyšší řádek, ve kterém se tato hodnota také vyskytuje, v našem případě je to řádek 12. Druhé písmeno tedy budeme určovat z $D[11, 7]$, třetí z $D[9, 6]$, atd. Jednou z hledaných podposloupností je:

posloupnost:	2	3	1	2	2	3	1	2
indexy v A:	1	2	4	5	7	9	10	12
indexy v B:	2	5	6	7	8	9	11	12

Ještě trochu konkrétněji:

```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;  { Délky posloupností }
  D: array[0..MaxN, 1..MaxN] of Integer;
  I, J, L, T: Integer;
begin
  ...
  if LA > LB then  { A bude kratší z obou }
  begin
    C := A;
    A := B;
    B := C;
    T := LA;
    LA := LB;
    LB := T;
  end;

  for I := 1 to LA do
    D[0, I] := LB;

  L := 0;
  for I := 1 to LA do
  begin
    for J := 1 to LA do
      D[I, J] := D[I-1, J];

    L := 1;
    for J := 0 to LB-1 do
      if B[J] = A[I-1] then
      begin
        while D[I-1, L] < J do Inc(L);
        if D[I, L] >= J then
          D[I, L] := J;
      end;
  end;

  LC := L;
  J := LA;
  for I := LC downto 1 do
  begin
    while D[J-1, I] = D[J, I] do Dec(J);
    C[I-1] := A[J-1];
    Dec(J);
  end;
  ...
end.
```

Již zbývá jen odhadnout složitost algoritmu. Časově nejnáročnější byl vlastní výpočet hodnot v poli, který se skládá ze dvou hlavních cyklů o délce $L(A)$ a $L(B)$, což jsou délky posloupností A a B. Vnořený cyklus while proběhne celkem maximálně $L(A)$-krát a časovou složitost nám nezhorší. Můžeme tedy říct, že časová složitost je $\mathcal{O}(L(A) \cdot L(B))$. Posloupnosti jsme si prohodili tak, aby první byla ta kratší, protože pak je maximální délka společné podposloupnosti i počet kroků algoritmu roven délce kratší posloupnosti a tedy i velikost pole s daty je kvadrát této délky. Paměťovou složitost odhadneme $\mathcal{O}(N^2 + M)$, kde N je délka kratší posloupnosti a M té delší.

Na rozmyšlenou: Proč jsme si z více posloupností zapamatovali zrovna tu, která v B končí nejlevějším možným prvkem?

Dnešní menu vám servírovali
Martin Mareš a Petr Škoda

Úloha 21-3-3 – Topologie snů – program

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

int N, prefix_pos = 0, infix_pos = 0; // počet prvků; pozice v prefixu a infixu
int prefix[MAX], infix[MAX]; // prefixový a infixový zápis

void vypisuj_postfix(int rodic) // rekurzivní výpis
{
        if (infix[infix_pos] == rodic || prefix_pos >= N)
                return;
        int hodnota = prefix[prefix_pos++]; // kořen podstromu

        vypisuj_postfix(hodnota); // vypíšeme levý podstrom
        infix_pos++; // posuneme se v infixu
        vypisuj_postfix(rodic); // vypíšeme pravý podstrom
        printf("%d ", hodnota); // nakonec vypíšeme samotný kořen
}

int main()
{
        scanf("%d", &N); // načteme zápisy
        for (int i = 0; i < N; i++) scanf("%d", &prefix[i]);
        for (int i = 0; i < N; i++) scanf("%d", &infix[i]);

        vypisuj_postfix(-1); // vypíšeme postfixový zápis
        return 0;
}
```

Úloha 21-3-4 – Optimalizace stromu – program

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

// Přednačtená, nezpracovaná polovina hrany
struct edge_raw {
  unsigned vertex, index;
};

// Jeden vrchol
struct vertex {
  // První hrana tohoto vrcholu. Další následují za ní, až do první hrany následujícího vrcholu
  unsigned edges;
  // Spočítané nejdelší cesty
  struct longest {
    // Nejdelší vedoucí tudy, nejdelší celková
    unsigned local, subtree;
    // Vrchol, kterým prochází nejdelší cesta v tomto podgrafu
    unsigned vertex;
    // Sousedi v cestě, která prochází tudy
    unsigned neighbors[2];
  } longest[2];
};

// Počet vrcholů
static unsigned tree_size;
// Přednačtené hrany
static struct edge_raw (*edges_raw)[2];
// Graf
static struct vertex *vertices;
static unsigned *edges;

// Najde nejdelší cestu v podstromu s kořenem index a nevrací se do parent
// Výsledky ukládá do longest[result]
static void path_search(unsigned index, unsigned parent, unsigned result) {
  // Případ, kdy nic hezkého nebylo nalezeno
  unsigned longest_subtree = 0, subtree_vertex = index;
  unsigned longest_neigh[3];
  unsigned local_lengths[3] = {};
  // Hrany vedoucí ze mě
  for(unsigned i = vertices[index].edges; i < vertices[index + 1].edges; i ++) {
    unsigned vertex = edges[i];
    if(vertex == parent) // Odsuď jsem přišel
      continue;
    if(vertex == tree_size) // Tahle hrana tu není (je "vypnutá")
      continue;
    // Prohledat tento podstrom
    path_search(vertex, index, result);
```

```
/* Vstupní texty */
char word[2][MAXW+2][MAXLEN+2];
int n,n0,n1;
Words w[2*MAXW+2];
int numseq[2][MAXW+2], diffwcnt;
/* 0 == slovo společné oběma sekvencím,
   1 == použijeme originální vzpomínku,
   2 == použijeme novou vzpomínku */
int best[MAXW+2][MAXW+2];
int bestlen[MAXW+2][MAXW+2];
int bestbeg[MAXW+2][MAXW+2];

int main(void)
{
	int i,j,tmp;

	/* Načteme originální vzpomínky */
	for(i=0;; i++) {
		scanf(" %s ", word[0][i]);
		if(word[0][i][0]=='.') break;
		w[i].w = word[0][i];
		w[i].sentence = 0;
		w[i].pos = i;
	}
	n0 = i;
	if(n0==0) return 0;

	/* Načteme vkládané vzpomínky */
	for(i=0;; i++)          {
		scanf(" %s ", word[1][i]);
		if(word[1][i][0]=='.') break;
		w[n0+i].w = word[1][i];
		w[n0+i].sentence = 1;
		w[n0+i].pos = i;
	}
	n1 = i;
	n=n0+n1;

	diffwcnt=0;
	for(i=0;i<n;i++) {
		if(i>0 && strcmp(w[i].w, w[i-1].w)) diffwcnt++;
		numseq[w[i].sentence][w[i].pos] = diffwcnt;
	}

	for(i=0;i<n0;i++) { bestlen[i][n1] = n0-i; bestbeg[i][n1] = numseq[0][i]; best[i][n1] = 1; }
	for(i=0;i<n1;i++) { bestlen[n0][i] = n1-i; bestbeg[n0][i] = numseq[1][i]; best[n0][i] = 2; }
	bestlen[n0][n1] = 0; bestbeg[n0][n1] = 0; best[n0][n1] = -1;

	for(i=n0-1;i>=0;i--)
		for(j=n1-1;j>=0;j--) {
			if(numseq[0][i]==numseq[1][j]) {
				bestlen[i][j] = 1+bestlen[i+1][j+1];
				bestbeg[i][j] = numseq[0][i];
				best[i][j] = 0;
				continue;
			}
			if(bestlen[i+1][j] < bestlen[i][j+1] ||
			   (bestlen[i+1][j] == bestlen[i][j+1] && numseq[0][i] < numseq[1][j])) {
				bestlen[i][j] = bestlen[i+1][j]+1;
				bestbeg[i][j] = numseq[0][i];
				best[i][j] = 1;
			} else {
				bestlen[i][j] = bestlen[i][j+1]+1;
				bestbeg[i][j] = numseq[1][j];
				best[i][j] = 2;
			}
		}

	i=0; j=0;
	while(1) {
		tmp = best[i][j];
		if(tmp==0 || tmp==1) printf("%s ", word[0][i]);
		if(tmp==2) printf("%s ", word[1][j]);
		if(tmp==-1) break;
		if(tmp==0 || tmp==2) j++;
		if(tmp==0 || tmp==1) i++;
	}
	printf(".\n");
	return 0;
}
```


21-3-1 Kódování memgramů

Nejprve vyřešíme jednoduchý případ, kdy má memgram 4 políčka, a pak se pustíme do případu pro obecné N.

V tabulce se 4 políčky lze přenést 2 bity následujícím způsobem: První bit informace bude `xor` hodnot v prvním řádku (tedy počet jedniček v prvním řádku), druhým bitem bude `xor` hodnot v prvním sloupci. Když dostaneme memgram v nějakém konkrétním stavu, podíváme se, které bity jsou nastaveny jinak, než jak je chceme odeslat. První bit opravíme změnou na políčku b, druhý opravíme převrácením hodnoty políčka c a pokud jsou špatně oba, stačí změnit políčko a. A pokud jsou oba bity nastaveny správně, změníme políčko d, které nemá na zprávu žádný vliv.

a	b
c	d

1. bit $= a \operatorname{xor} b$
2. bit $= a \operatorname{xor} c$

Více než 2 bity ale v této malé tabulce přenést nelze: jsou totiž 4 možné zprávy (4 možné kombinace hodnot posílaných 2 bitů), a my můžeme provést jen 4 různé akce (změnit jedno ze 4 políček). Při přenášení více bitů by bylo možných zpráv více, ale my dokážeme rozlišit jen 4.

Tento důkaz lze snadno rozšířit pro obecný případ, kdy má tabulka N políček. Umíme provést pouze N různých akcí, tedy umíme poslat nejvýše N různých zpráv, což odpovídá $\log_2 N$ přeneseným bitům informace.

A jak bude vypadat přenos informace většími tabulkami? Necháme se inspirovat případem $N = 2$: Najdeme v tabulce $\log_2 N$ množin políček, každá bude nést jeden bit informace v podobě `xor`u hodnot na všech svých políčkách. A navíc potřebujeme, aby pro každou (i prázdnou) množinu bitů existovalo v tabulce políčko, jehož změna způsobí změnu právě těchto vybraných bitů (a žádných jiných). Pak totiž budeme moci opravit libovolnou kombinaci bitů, které budou v náhodně nastaveném memgramu špatně.

Stačí tedy, když o každém políčku řekneme, do kterých množin patří (které bity ovlivní jeho změna). Takovou informaci o jednom políčku si můžeme představit jako číslo ve dvojkové soustavě – i-tá číslice bude 1, pokud dané políčko leží v i-té množině (ovlivňuje i-tý bit zprávy), jinak bude 0. Aby byly množiny správně rozděleny (a bylo možné opravit libovolnou kombinaci bitů), musí v tabulce existovat všechna čísla od 0 do $2^{\log_2 N} - 1 = N - 1$. To lze ale zařídit jednoduše, prostě políčka popořadě očíslujeme čísly 0 až $N - 1$. Po převodu těchto čísel do dvojkové soustavy pro každé políčka zjistíme, které bity ovlivňuje, a můžeme jít vesele kódovat :-)

V konkrétním případě, kdy tabulka má velikost 8×8, je možné přenést $\log_2 64 = 6$ bitů informace, a lépe to už nejde.

Petr Kratochvíl

21-3-2 Nadposloupnost

Nadposloupnost zřejmě hodně lidí odradila svojí komplikovaností; ten zbytek aspoň zmátla. Část problému byla v tom, že sny byly zadány jako stringy a řešení se snažila pracovat s nimi opravdu efektivně.

Nechť L je délka obou posloupností ve znacích a N je délka obou posloupností ve slovech.

Řešení používající jednoduché strcmp má složitost $\mathcal{O}(L^2)$. S použitím trie se dá vyrobit řešení v čase $\mathcal{O}(N^2+L)$, které je ale zbytečně komplikované, a dá se předpokládat, že slova stejně budou krátká.

Jak takové řešení bude fungovat? Bylo by možné použít kuchařku na hledání společné podposloupnosti, a potom mezi její prvky naskládat přebytecná slova.

Ale přímé řešení je možná názornější: Budeme si pamatovat nejlepší řešení pro prvních N slov z originálních vzpomínek a prvních M slov z přidávaných vzpomínek (pole `best/bestlen/bestbeg`). Nejlepší řešení pro N/M zjistíme na základě již spočítaných řešení pro $1 \ldots N-1/1 \ldots M-1$.

Pokud slovo je společné v obou sekvencích, tak nejlepší řešení bude přidat do výstupu tuto vzpomínku, jinak máme na vyběr mezi řešením pro $n-1/m$ a přidat originální vzpomínku a řešením pro $n, m-1$ a přidat vkládanou vzpomínku.

Časová složitost bude $\mathcal{O}(L^2)$, paměťová by byla $\mathcal{O}(L+N^2)$, kdybychom ovšem používali dynamickou alokaci.

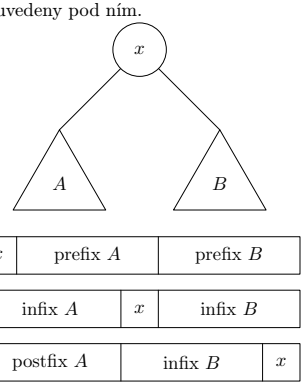
S díky CTU Open Contestu 2008 a Josefu Cibulkovi.

Pavel Machek

21-3-3 Topologie snů

Skoro všechna došlá řešení se úspěšně vypořádala s problémem a snoví inženýři mohli být nadmíru spokojeni. Jak už to ale bývá, některá byla pomalejší a jiná o maličko rychlejší. Udělejme si malou vycházku do lesa a podívejme se na stromy pořádně.

Jakpak vypadá prefixový, infixový a postfixový zápis? Na obrázku máme strom s kořenem x a podstromy A a B. Tři zápisy jsou uvedeny pod ním.



Kořenem stromu je x, které se nachází na první pozici v prefixovém zápisu. Naproti tomu v infixovém zápisu odděluje prvky levého podstromu a prvky pravého podstromu. Navíc k těmto dvěma podstromům máme i jejich prefixové zápisy. Tím jsme problém rozdělili na dva a můžeme použít pro stromy typický postup *rozděl a panuj*. Pro stromy se hodí obzvlášť, neboť pro ně je rozdělení na menší podproblémy naprosto přirozené, máme podstromy a elementární podproblémy mohou být listy.

Nalezneme kořen x v infixovém zápisu, třeba tak, že projdeme všechny prvky, a problém rozdělíme na dva podstromy, které budeme řešit rekurzivně. Rekurze se zastaví v listech,

pro které jsou všechny tři zápisy stejné. Strom si nemusíme vytvářet v paměti, ale můžeme postfixový zápis rovnou vypisovat. Jaká je časová složitost? V každém kroku si potřebujeme vyhledat kořen v infixovém zápisu. Na to potřebujeme lineární čas v délce úseku. Pokud bude strom vyvážený, dostaneme celkový čas $\mathcal{O}(N \log N)$, v nejhorším případě však $\mathcal{O}(N^2)$. Analýza je velice podobná jako u QuickSortu.

Pokud bychom chtěli zrychlit výše popsaný algoritmus, potřebovali bychom zrychlit vyhledávání kořenů v infixovém zápisu. Mohli bychom si pro každý prvek stromu předpočítat, kde přesně se v infixovém poli nachází. Pro to není potřeba konstruovat žádné komplikované struktury, stačí nám si ke každé hodnotě v infixovém zápisu ještě pamatovat její pozici a poté infixový zápis utřídit. Vyhledávat pak můžeme pomocí půlení intervalu. S tímto zrychlením dosáhneme časové složitosti $\mathcal{O}(N \log N)$ v nejhorším případě. Podaří se nám to vyřešit ještě rychleji? Co kdybychom se na věc podívali z jiného úhlu?

Co v předchozím algoritmu stálo tolik času? Lineární čas $\mathcal{O}(N)$ stojí procházení a vypisovaní prvků, tady zjevně algoritmus nezrychlíme. Velice drahé je však vyhledávání v infixovém zápisu, zkusíme se obejít bez něj. Vždy jsme nejprve problém rozdělili na dva menší a pak je řešili. Co kdybychom s rozdělením počkali, tedy nejprve pustili řešení na levý podstromu a až v průběhu zjistili, kde se nachází kořen v infixu. Jak toho ale dosáhnout? V infixovém seznamu se postupně budeme posouvat. Na začátku ukazujeme na jeho první prvek. Prefixový seznam procházíme. Ve chvíli, kdy narazíme v prefixovém seznamu na prvek z infixového, víme, že jeho levý podstrom máme vyřešený. V infixu se posuneme na další prvek a zbývá vyřešit pravý podstrom. Když narazíme na rodiče o jedna výše, víme, že i pravý podstrom je vyřešený, protože jsme vyřešili levý podstrom umístěný výš. Takto rekurentně dokážeme vypsat strom v lineárním čase $\mathcal{O}(N)$.

Pavel Klavík

21-3-4 Optimalizace stromu

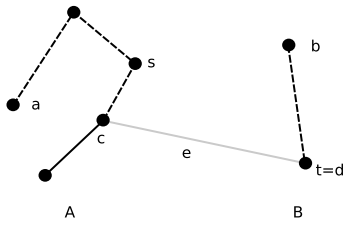
Vyberme si dva nejvzdálenější vrcholy a a b v neoptimalizovaném stromě T. Mezi nimi vede nějaká cesta P, jejíž délka je rovna průměru stromu $d(T)$.

Pokud je optimalizace úspěšná a zoptimalizovaný strom T' má menší průměr, potom musí existovat i kratší cesta mezi a a b. A protože T' je strom, P již existovat nemůže, tedy jsme při optimalizaci museli odebrat některou hranu e z P.

Po odebrání hrany e (ale před přidáním nové) vznikly dva stromy, říkejme jim třeba A a B. Každý z nich obsahuje jeden z vrcholů a a b, nechť jsme si je tedy pojmenovali tak, aby $a \in A$ a $b \in B$.

Dále si označme c a d konce hrany e ve stromech A a B.

Na pomoc těm, kteří se ztrácejí ve značení, zde je obrázek.



Nyní si v každém stromu najdeme střed (pro A to bude s, pro B t). Střed bude takový vrchol, že cesta do nejvzdálenějšího vrcholu téhož stromu bude nejmenší možná. Je zřejmé, že pokud jsme již správně zvolili hranu e, tak spojením s a t získáme optimální strom T' – kdybychom zvolili nějaký vrchol, který má k některému vrcholu ve svém stromě dále, vznikne tím delší cesta. Lze si všimnout, že s, resp. t, lze umístit doprostřed nejdelší cesty v A, resp. B (pokud jsou prostředky dva při sudém počtu vrcholů na cestě, pak lze vybrat libovolný z nich). Kdyby totiž vzdálenost do některého vrcholu byla větší než polovina této cesty, lze cestu prodloužit a není tudíž nejdelší. Naopak, alespoň polovinu určitě bude potřebovat libovolný vrchol na této cestě (k tomu vzdálenějšímu konci cesty).

Rozmysleme si tedy, jak správně vybrat hranu e. Už víme, že se musí nacházet na cestě P. Jaký bude průměr grafu po optimalizaci? Nová nejdelší cesta buď povede přes nově přidanou hranu, pak její délka bude:

$$\left\lceil \frac{d(A)}{2} \right\rceil + 1 + \left\lceil \frac{d(B)}{2} \right\rceil$$

Tato vznikne spojením cesty z s do nejvzdálenějšího vrcholu v A, nové hrany a cesty z t do nejvzdálenějšího vrcholu v B.

Další možnost je, že nejdelší cesta je uvnitř jednoho malého stromu, tedy $d(A)$ nebo $d(B)$ je větší, než nejdelší spojená cesta. Na tento případ někteří řešitelé zapomněli.

Jiní si úlohu zjednodušili tak, že odebírali prostřední hranu cesty. To, bohužel, také nefunguje (protipříkladem budiž cesta o 12 vrcholech).

Jak tedy najdeme správnou hranu? Jednoduše, vyzkoušíme všechny na cestě P. U každé hrany si spočítáme průměry stromů, které by vznikly, když bychom ji odebrali. Z nich spočítáme nejdelší cestu, která v celém stromě poté povede. Nakonec vybereme nejlepší hranu a odebereme.

Potřebujeme tedy najít napřed nejdelší cestu stromu. Strom si zakořeníme a projdeme ho do hloubky. V každém vrcholu vždy spočítáme hodnoty pro celý jeho podstrom, za použití již spočítaných hodnot ze všech synů.

Budeme počítat délku nejdelší cesty v celém podstromu a délku nejdelší cesty, která končí v kořeni aktuálního podstromu. V listech je situace jednoduchá, obě cesty mají délku 0. Pokud má vrchol syny, pro spočítání nejdelší končící v něm vezme „nejlepší nabídku“ od synů – nejdelší, končící v některém synovi. Tu poté prodlouží o 1 hranu. Nejdelší v podstromu bude buď nejdelší v podstromu některého syna a nebo spojení dvou nejdelších končících v synech. Je třeba ještě ošetřit, když je syn jen jeden (pak je druhá nabídka nulová).

Nakonec si výsledek vyzvedneme v kořeni.

Protože potřebujeme nejen délku cesty ale i cestu samotnou, budeme si pamatovat vždy některý vrchol nejdelší cesty v podstromu (nejjednodušší bude si pamatovat ten nejblíže ke kořeni) a v každém vrcholu cesty si pamatovat, kam je třeba pokračovat. Pak lze cestu jednoduše zrekonstruovat.

Proč to bude fungovat? Dokážeme indukcí. Že jsou informace pravdivé v listech je zřejmé. To, že nejdelší cesta končící do tohoto vrcholu je prodloužením jedné z nejdelších cest v listech je také vidět. A nejdelší cesta v podstromu buď prochází kořenem vrcholu, pak se skládá ze dvou končících v něm, a nebo neprochází. V takovém případě ale musí být uvnitř některého synovského podstromu.

Nyní máme tedy cestu, jak z ní vybrat správnou hranu? Potřebujeme znát průměr každého „zajímavého“ stromu (vzniklého odebráním některé hrany cesty P). Všimneme si, že délka nejdelší cesty v podstromu je průměr tohoto podstromu. Kdybychom tedy vybírali kořeny podstromů při výpočtu nejdelší cesty tak, že leží na cestě P, dostali bychom vždy jeden ze stromů odebráním hrany otec-syn.

Napřed tedy najdeme cestu P. Poté strom zakořeníme, tentokrát ve vrcholu a a prohledáme znovu. Cesta P vede z a (kořene) do některého z listů. Tedy dostaneme všechny zajímavé stromy obsahující b. Poté uděláme ještě jednou totéž, ale opačně – zakořeníme v b.

Poté projdeme hodnoty a odebereme správnou hranu. Nyní potřebujeme najít středy zbylých stromů. Ty ale, jak jsme si již všimli, leží uprostřed nejdelších cest těchto stromů. Nejdelší cesty těchto stromů máme již spočítané, tudíž středy dostaneme zadarmo.

Časová složitost je lineární. V každém vrcholu provedeme konstantně mnoho operací. Tedy, pokud porovnávání nabízených hodnot budeme považovat ještě za zpracování každého syna (každý bude s nejlepší a 2. nejlepší nabídkou porovnáván jen jednou, ve svém otci). Paměťová složitost je také lineární, více než lineárně mnoho paměti v lineárním čase nestihneme spotřebovat.

Je ale potřeba si dát pozor při načítání a pokud je v něm potřeba třídit (např. hrany podle zdrojového vrcholu), tak použít přihrádkové třídění, aby nám to složitost nezhoršilo.

Michal „vorner“ Vaner

21-3-5 Rozklad na součty

Nejeden řešitel se nyní právem raduje z velkého součtu svých bodů, neboť tato úložka byla pouze jednoduchým cvičením na rekurzi. Klíčové bylo vhodně rozmyslet, v jakém pořadí vytvářet jednotlivé součty, aby se nám žádný neopakoval víckrát.

Jeden z možných způsobů je generovat součty uspořádané vždy do monotónní posloupnosti. V našem příkladu vytváříme nerostoucí posloupnost, kterou pak vypíšeme v opačném pořadí (díky tomu je vypíšeme dokonce ve stejném pořadí jako v zadání, i když to není nezbytně nutné). Zbývá prozradit technický detail, jak jednoduše vytvářet uspořádané posloupnosti sčítanců. Zavedeme si při generování maximální číslo m a všechny zkoušené sčítance pak budou pouze z rozsahu 1 až m. Když pak sčítanec na pozici i má hodnotou k, necháme zbylé sčítance (na pozicích $i+1$ a vyšších) generovat s parametrem $m = k$, takže následující sčítance budou nejvýš tak velké, jako ten na pozici i. Nyní stačí jen přesypat naše úvahy do nějakého programovacího jazyka, trochu zamíchat a voilà ...

Na závěr bychom rádi poznamenali, že někteří vykutálení řešitelé se snažili optimalizovat svůj program vložením předpočítaných výsledků přímo do zdrojového kódu. Po načtení vstupu pak pouze vypsali příslušnou množinu výsledků. To je sice naprosto legitimní a v některých situacích i velmi rozumná optimalizační technika, avšak v tomto případě není nikterak užitečná. Nejpomalejším místem této úlohy je beztak vypsání výstupu a časové limity byly dostatečně velké, abyste mohli použít rekurzi. Nám nezbývá než pochválit tuto invenci a zároveň upozornit, že jsme nastavili omezení na maximální velikost odevzdávaného řešení (na $128kB$), takže příště již tato technika nepůjde použít.

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

21-3-6 Pan Cowmess

Nejprve rozluštěme, co program pana Cowmesse dělal. Číslo zadané v registru `x` nejprve zapsal ve dvojkové soustavě pomocí právě 32 cifer. Pak opakovaně hledal páry tvořené nulou a jedničkou a přepisoval je na páry dvojek. Nakonec zkontroloval, jestli byly všechny číslice přepsané, a podle toho odpověděl. Jinými slovy testoval, zda je v dvojkovém zápisu čísla právě 16 jedniček. Použít na tak triviální úlohu celých 21 instrukcí je skutečně naprostá nehoráznost.

Můžeme napsat daleko jednodušší program, který bude při převodu do dvojkové soustavy rovnou počítat jedničky:

```
jupiii: a=x%2
        b=b+a
        x=x/2
        if x<>0 => jump jupiii
        if b<>16 => jump ooops
        y=1
ooops:
```

Krásných 6 instrukcí by si zasloužilo potlesk, ale vavřínový věnec ještě ne. Stačí jich totiž pouhá polovina. Dopomůže nám k tomu starý dobrý operátor bitové selekce, který nám už nejednou nečekaně zamíchal karty.

Pokud spočítáme `x@x`, získáme číslo, které obsahuje tytéž jedničky co `x`, jenže „sklepané doprava.“ Pak už jen toto číslo porovnáme s číslem 65535 (16 jedniček umístěných úplně vpravo) a máme vyhráno:

```
        a=x@x
        if a<>65535 => jump oooops
        y=1
oooops:
```

Po tomto skandálním odhalení již zbývá jen popřát panu Cowmessovi, aby mu jeho zákazníci snížili odměnu na polovic za každou přebytečnou instrukci :)

Martin Mareš & Milan Straka

Úloha 21-3-2 – Nadposloupnost – program

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

#define MAXW 2100
#define MAXLEN 1000

typedef struct Words {
        char *w;
        int sentence;
        int pos;
} Words;
```