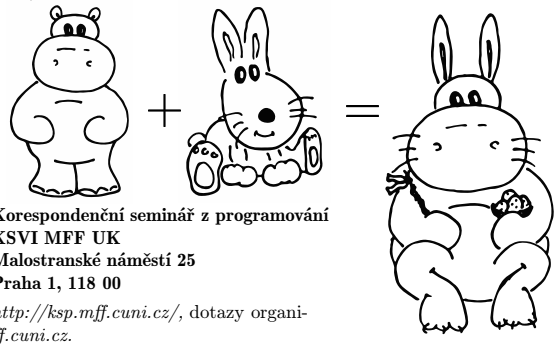
Výsledková listina sedmnáctého ročníku KSP po třetí sérii

		škola	ročník	1731	1732	1733	1734	1735	suma	celkem
1.	Miroslav Klimoš	G Lanškr	0	8	10	9	6	8	41	134
2.	Peter Černo	GLŠtúra	4	7	10	9	3		29	130
3.	Miroslav Cicko	GJGTajov	4	10	10	9			29	123
4.	Peter Perešíni	GJGTajov	3	10	10	10	10		40	121
5.	Josef Pihera	G Strakon	2	10	10	6	10	9	45	115
6.	Jan Pelc	G UBrod	3	10	10	9	5	9	43	113
7.	Ondřej Bílka	G Zlín	3	6	6	2	3	9	26	112
8.	Zbyněk Konečný	GKptJaroš	2	4	10	6	3	8	31	93
9.	Pavel Klavík	G Chrudim	2	8	9	9	7	3	36	89
10.	Adam Zivner	G UBrod	3	10	1	3	5	8	27	83
11.	Jan Bulánek	G Klatovy	4	10	9	9	3		31	71
12.	Martin Koníček	G UBrod	4						0	64
13.	Jan Hrnčíř	GFXŠaldy	3	8	4	8	6		26	60
14.	Jakub Kaplan	GJKTyla	1	3	2	4	7		16	53
15.	Martin Čech	G UBrod	4						0	52
16.	Petr Kratochvíl	G SvětláNS	2	4		2	5	3	14	48
17.	Lukáš Lánský	GJKTyla	1	1	2	2	3		8	45
18.	Stanislav Basovník	G Kroměříž	4						0	43
19.	Cyril Hrubiš	G Bílovec	3	9	3	2	6		20	40
20.	Roman Smrž	GOhradní	1	5			7		12	38
21.	Eva Schlosáriková	G Piešťany	4	2	8	1			11	36
22. – 23.	Zbyněk Falt	GNeumannov	4	7			7		14	32
	Tomáš Herceg	G Třebíč	2	3	1	1			5	32
24.	Martin Kupec	GMendel	3						0	28
25.	Josef Špak	GJírovco	2						0	25
26.	Lukáš Špalek	G Čadca	4						0	24
27.	Michal Pavelčík	G UBrod	2						0	20
28. – 29.	Ondřej Bouda	GKptJaroš	2						0	18
	Adam Ráž	GBudějo	2						0	18
30.	Marian Kaluža	GHavlíčkov	2			5	2		7	16
31. – 33.	Jiří Cabal	SPŠ DvKrál	2						0	15
	Ondřej Garncarz	G Příbor	4						0	15
	Martin Kahoun	GJNerudy	2						0	15
34.	Jan Palenčar	G Martin	2						0	14
35.	Martin Podloucký	G Strážnic	4						0	12
36. – 39.	Jakub Jenis	GsvCyrMet	1						0	11
	Hana Klempová	GUBalvanJN	4						0	11
	Jakub Porod	G Týn nV	2						0	11
	Ján Zahornadský	GZborov	4						0	11
40. – 42.	Lukáš Beleš	G Čadca	4						0	10
	Jakub Benda	GJNerudy	2						0	10
	Michal Vaner	G Turnov	3						0	10
43. – 44.	Jiří Machálek	G Holešov	3						0	8
	Petr Soběslavský	GJHeyrovs	4						0	8
45. – 46.	Daniel Sedláček	SPŠE Hav	1						0	7
	Filip Šauer	G Klatovy	4						0	7
47.	Jiří Nohavec	G Domažl	4						0	6
48. – 51.	Dalibor Adamčík	SPŠE Preš	2						0	5
	Petr Musil	G MBuděj	3	3	0				3	5
	Jan Staněk	GKptJaroš	3						0	5
	Zdeněk Vilušínský	G Turnov	4						0	5
52. – 53.	Tomáš Ehrlich	G Holešov	2						0	2
	Martin Vařák	G Bílovec	2						0	2
54. – 56.	Florián Danko		2		0				0	1
	Tamara Kuštárová		0						0	1
	Petr Zimčík	G UBrod	1						0	1
57. – 58.	Miroslav Hovorka	GJateční	4						0	0
	Adrián Lachata	G Svidník	3						0	0

Milí řešitelé!



Blížící se Velikonoce zaujaly i našeho kamaráda:

Hody hody doprovody, já jsem malý hrošíček,
utíkal jsem podle vody, dohonil mě zajíček.
V košíku však místo vajec úložky měl z Ká eS Pé,
kdo je rychle nevyřeší, ten bude mít u něj zle.

Termín odeslání poslední páté série je tentokrát 2. května 2005. Řešení můžete odevzdávat jak elektronicky na *http://ksp.mff.cuni.cz/submit/*, tak klasickou poštou na známou adresu:

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

Aktuální informace o KSP naleznete na stránkách *http://ksp.mff.cuni.cz/*, dotazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz*.

Zadání páté série sedmnáctého ročníku KSP

17-5-1 Velkovezír — 10 bodů

Když šel malý hrošík koledovat, srazil se na cestě se zajíčkem. Zajíček, který velmi pečlivě dodržoval velikonoční zvyky, také se mu říkalo **Vel**mi **ko**merční **ve**likonoční **zaj**eč**í** **r**aubíř, mu povídá: „Dávej přece pozor, když jde koledovat ten nejlepší koledník z okolí!“

„Vůbec nejsi nejlepší, Velkovezíre. Já jsem minulý rok vykoledoval čtyřicet dva vajíček!“ „No, to je sice pravda, ale za poslední tři roky jsem jich já vykoledoval sto dvanáct!“ „Ale před pěti lety jsem jich já vykoledoval šedesát čtyři!“

Když ani po notné chvíli nepřestali, rozhodli jste se jim pomoci a spravedlivě určit, který z nich je lepší koledník. Nakonec jste se dohodli, že lepší koledník je ten, jehož průměr vykoledovaných vajíček za souvislé období alespoň K roků je největší.

Napište zvířátkům program, který dostane na vstupu přirozená čísla N a K taková, že $1 \leq K \leq N$. Dále dostane posloupnost N celých čísel a Vaším cílem je najít takovou souvislou podposloupnost této posloupnosti délky alespoň K, že má největší *aritmetický průměr*, což je součet jejích prvků vydělený jejich počtem. Pokud je takových podposloupností víc, vypište libovolnou z nich. Ale protože se mezitím hádka přiostřila, měl by Váš program pracovat co nejrychleji.

Příklad: Pro $N = 5$, $K = 2$ a posloupnost $(4, 8, -2, 15, -5)$ by měl Váš program najít $(8, -2, 15)$ s průměrem 7.

17-5-2 Ranní hroše — 9 bodů

Poté, co jste rozhodli při hrošíka s Velkovezírem (bohužel v hrošíkův neprospěch), se malý hrošík vrátil domů a stěžoval si tatínkovi, že Velkovezír je lepší koledník než on. „Tatínku, proč je lepší než já?“ „A kdypak jsi dneska vstával?“ „Časně ráno, sluníčko ještě nezapadlo.“ „A vida ho, našeho lenocha. Nevíš, že ranní hroše dál doduše? Zajíček určitě vstává brzo a každý mu dá spoustu vajíček.“

To malého hrošíka nadchlo. Pokud je to pravda, určitě by dokázal vstát jednou v roce už před polednem. Ale aby zjistil, jestli je to opravdu tak, jak tatínek říká, běžel se zeptat zajíčka, odkdy dokdy chodil o Velikonocích koledovat.

Ale když se vrátil, musel jít špinit nádobí (hroši mají čisté nádobí neradi) a proto Vám dal následující úkol.

Dostanete dvě množiny H a V, každá obsahuje nějaké intervaly tvaru (od, do). Jednotlivé intervaly znamenají odkdy dokdy chodila zvířátka koledovat, v množině H jsou časy hrošíka a v množině V časy Velkovezíra. Máte zjistit, zda hrošík někdy začal a skončil s koledováním dřív než zajíček. Matematicky řečeno zjišťujete, zda existuje nějaký interval h z množiny H a v z množiny V, že h začíná dříve než začíná v a h končí dříve než končí v, čili že $h_{od} < v_{od}$ a $h_{do} < v_{do}$.

Příklad: Pro zadané množiny $H = \{(10, 100), (5, 200)\}$ a $V = \{(8, 110), (20, 105)\}$ je hledané $h = (10, 100)$ a $v = (20, 105)$, protože $10 < 20$ a $100 < 105$. Pokud by bylo $V = \{(8, 110), (9, 105)\}$, hledané intervaly by neexistovaly.

17-5-3 Nouze V-dáli-hrocha — 8 bodů

Hrošík teď už věděl, proč je zajíček lepší koledník než on, a rozhodl se, že ho příští rok překoná. Ale aby toho dosáhl, musí si své koledování podrobně naplánovat. Rozhodl se, že bude věren přísloví Nouze naučila V-dáli-hrocha houstnouti, bude jíst jen šestkrát denně, a celý rok plánovat své koledování.

Rád by si vybral nejrychlejší trasu, podél které bude koledovat. A aby byla opravdu nejrychlejší, rozhodl se projít všechny možnosti, které má, a vybrat tu nejlepší.

Hrošík bude koledovat u N svých sousedů. Jeho trasa je vlastně pořadí, v jakém bude své sousedy navštěvovat, takže je to posloupnost čísel $1, 2, \ldots, N$, ve které se každé vyskytuje právě jednou. Hrošík by po Vás chtěl, abyste mu vypsali všechny možné trasy, které má, každou právě jednou. Navíc by si ale přál, aby se dvě trasy vypsané hned po sobě lišily jenom prohozením jedné dvojice sousedů (čili aby se posloupnosti reprezentující trasy shodovaly ve všech prvcích kromě dvou, které jsou prohozené). První a poslední vypsané trasy se mohou libovolně lišit.

Bonus: Pokud se bude i první a poslední trasa lišit právě prohozením jedné dvojice prvků, dostanete bonus *3 body*.

Příklad: Pro $N = 3$ je jedním ze správných výstupů (i pro bonusovou úlohu):

1 2 3
2 1 3
3 1 2
3 2 1
2 3 1
1 3 2

17-5-4 Kudy tudy cestička — 11 bodů

Zatímco si hrošík vybíral nejkratší trasu, uběhl skoro celý rok. A tak den před Velikonocemi hrošík zjistil, že už půjde zítra koledovat, a přitom neví, jakou trasou vlastně půjde.

Protože je hrošík Váš kamarád (nebo snad proto, že nemáte rádi zajíčky?), určitě mu rádi poradíte, tentokrát trochu konkrétněji než v minulé úloze.

Hrošík chce opět navštívit N svých sousedů. Tyto sousedy si můžete představit jako body v rovině. Navíc pokud si všechny tyto body představíte jako vrcholy N-úhelníku, platí, že tento N-úhelník je konvexní (to znamená, že všechny jeho vnitřní úhly mají velikost menší než $180°$).

Mezi některými dvojicemi sousedů vedou pěšinky, každá je nějak dlouhá. Všechny pěšinky jsou úsečky, každá spojuje dané dva body odpovídající sousedům, mezi kterými vede. Různé úsečky se samozřejmě mohou křížit.

Hrošík by chtěl takovou trasu, která začíná u libovolného souseda, bude se skládat jenom z daných pěšinek a navštíví každého souseda právě jednou. (To znamená, že na své trase použije právě $N-1$ pěšinek.) Navíc se žádné dvě pěšinky, které jsou v trase použity, nesmí křížit. A aby mohl být hrošík lepší koledník než Velkovezír, Vámi nalezená trasa by měla být nejkratší možná.

Váš program tedy dostane na vstupu N, dále souřadnice N bodů v rovině, které tvoří konvexní mnohoúhelník, a dále seznam M pěšinek, každá vede mezi jinou dvojicí sousedů a má nějakou délku. Všechny pěšinky jsou obousměrné a jsou to úsečky spojující body odpovídající sousedům, mezi kterými pěšina vede. Vaším úkolem je najít takovou nejkratší trasu, která navštíví každého souseda právě jednou a žádné dvě z pěšinek této trasy se nekříží. Pokud je jich víc, vypište libovolnou z nich.

Příklad: Pro 4 body $(0,1),(1,0),(0,-1),(-1,0)$ a pěšinky

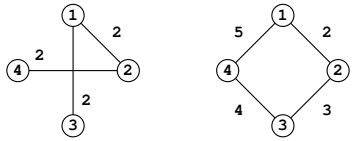
odkud	*kam*	*délka*
1	2	2
1	3	2
2	4	2

žádná hledaná trasa neexistuje. Pro pěšinky

odkud	*kam*	*délka*
1	2	2
2	3	3
3	4	4
4	1	5

hledaná trasa existuje, hrošík navštíví sousedy v pořadí $4,3,2,1$.

Pro představu následuje obrázek obou popsaných případů:



17-5-5 Jazykozpytec se loučí — 10 bodů

V posledním díle našeho automatově-gramatického seriálu jsme si slíbili povědět něco o dalších typech gramatik. (Asi se bude hodit připomenout si, co je to gramatika. Přesnou definici, komentáře a příklady čtenář najde v prvním díle seriálu.)

Bezkontextová gramatika je gramatika skládající se pouze z pravidel tvaru

$$X \to \alpha,$$

kde

- $X \in V_N$ je neterminál,
- $\alpha \in (V_T \cup V_N)^*$ je nějaká konečná posloupnost terminálních či neterminálních symbolů.

Jak vidíme, oproti gramatikám popisujícím regulární jazyky (připomeňme, že ty obsahují pouze pravidla $X \to wY$ nebo $X \to w$) je pravá strana pravidel poněkud volnější.

Příklad: Jazyk všech palindromů (množinu všech slov, co se čtou pozpátku stejně jako zepředu) nad abecedou $\{a,b\}$ popisuje následující jednoduchá bezkontextová gramatika (V_N, V_T, S, P). Jediný neterminální symbol $V_N = \{S\}$ je zároveň počáteční, terminální symboly jsou $V_T = \{a,b\}$ a přepisovací pravidla P jsou

$$S \to \lambda \mid a \mid b \mid aSa \mid bSb.$$

Například slovo *babab* dostaneme posloupností přepisů

$$S \to bSb \to baSab \to babab.$$

Pokud vás mate název „bezkontextové gramatiky“, pak vězte, že existují ještě tzv. *kontextové gramatiky,* které obsahují pouze pravidla tvaru

$$\alpha X \beta \to \alpha\gamma\beta,$$

kde

- $X \in V_N$ je neterminál,
- $\alpha, \beta \in (V_T \cup V_N)^*$ jsou libovolné konečné posloupnosti, klidně prázdné, terminálních či neterminálních symbolů,
- $\gamma \in (V_T \cup V_N)^+$ je libovolná posloupnost terminálních či neterminálních symbolů s výjimkou prázdného slova λ.

Navíc ještě povolíme pravidlo $S \to \lambda$, pokud se S nevyskytuje na pravé straně žádného pravidla. Ačkoliv na první pohled vypadá docela chaoticky, že jsme zakázali všechna zkracující pravidla a právě toto jedno povolili, vězte, že je to opravdu potřeba, protože jinak by vznikla daleko obecnější třída jazyků, než chceme, nebo by naopak kontextové jazyky nebyly rozšířením bezkontextových nebo regulárních. O podrobnostech pro tentokrát pomlčíme.

Lidsky řečeno, kontextové gramatiky mohou pro rozexpandování symbolu X využít ještě informaci o tom, jakými symboly je právě obalen, tedy v jakém se nachází „kontextu“, a na základě tohoto kontextu provádět odlišné expanze.

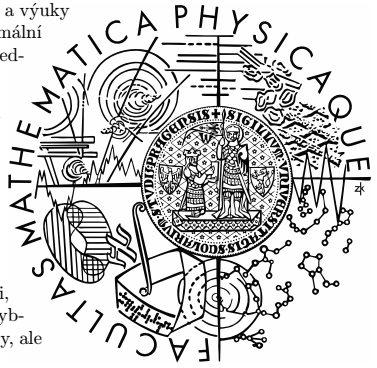
Příklad: Ukážeme si gramatiku, která popisuje jazyk $L = \{a^n b^n c^n;\ \forall n \in N, n > 0\}$. Gramatika $G = (V_N, V_T, S, P)$ bude používat neterminální symboly $V_N = \{S, B, C, X\}$, terminální symboly $V_T = \{a,b,c\}$ a množina přepisovacích pravidel P bude následující:

$S \to aSBC \mid abC$	... namnož pomocné symboly
$CB \to BC$	... setřiď je (Pozor, podvod!)
$bB \to bb$	... a zruš všechny pomocné symboly
$bC \to bc$	
$cC \to cc$	

Všimněte si řádku, kde upozorňujeme na podvod. Toto pravidlo samozřejmě není kontextové. Namísto něj ve skutečnosti použijeme tři pravidla

$$CB \to XB$$
$$XB \to XC$$
$$XC \to BC,$$

Studium informatiky na MFF UK Praha



Milí řešitelé, v minulých dílech jsme vám postupně představili Kabinet software a výuky informatiky (KSVI), Středisko informatické sítě a laboratoří (SISAL), Ústav formální a aplikované lingvistiky (ÚFAL), Centrum komputační lingvistiky (CKL), Katedru aplikované matematiky (KAM) a Institut teoretické informatiky (ITI).

Katedra softwarového inženýrství (KSI) se orientuje na výzkum a výuku těch „praktičtějších“ oborů informatiky, jako jsou např. datové inženýrství (organizace a zpracování dat, databázové systémy, dokumentografické systémy), softwarové inženýrství (návrh a analýza informačních systémů, jejich technologické a manažerské aspekty), počítačové systémy (operační systémy, překladače) xnebo tzv. distribuované systémy (počítačové sítě, komunikace). Katedra je garantem výuky těchto oboru a o studenty nemá v žádném případě nouzi, protože absolventi snadno nacházejí uplatnění u našich i zahraničních softwarových firem. Nejsou to totiž jen šikovní programátoři, ale především tvůrčí pracovníci, kteří jsou schopni systémy sami navrhovat, analyzovat a starat se o jejich bezchybný a bezpečný provoz. Na výuce se podílejí jednak kmenoví zaměstnanci katedry, ale také externisté z jiných pracovišť, např. z ČVUT, AV ČR nebo i z praxe.

Kromě výuky katedra také rozvíjí vlastní vědeckou a výzkumnou práci. Působí zde několik tématicky zaměřených výzkumných skupin, např. pro oblast dokumentografických systémů, distribuovaných systémů, operačních systémů nebo pro studium kybernetických hrozeb v počítačových a telekomunikačních sítích. Tyto skupiny jsou obvykle složeny nejen z pracovníku katedry, ale jsou v nich i posluchači doktorského studia. Často spolupracují na projektech společně s odborníky z jiných vysokých škol u nás i v zahraničí, prezentují výsledky své práce v našich i zahraničních odborných časopisech nebo na konferencích, hostují na zahraničních pracovištích.

Chcete-li se o lidech z této katedry a o jejich práci dozvědět více, můžete navštívit její internetové stránky, které se nacházejí na adrese *http://kocour.ms.mff.cuni.cz/cs/index.html*.

```
  for I:= 0 to N - 1 do
  for J:= 0 to N - 1 do
  if I = J then
  begin
    Equiv[I, I]:= True;
    Done[I, I]:= True;
  end else
  if Final[I] <> Final[J] then
  begin
    Equiv[I, J]:= False;
    Done[I, J]:= True;
  end else
    Done[I, J]:= False;

  for I:= 0 to N - 1 do
  for J:= I + 1 to N - 1 do
    if Reach[I] and Reach[J] and not Done[I, J] then Equivalent(I, J);

  for I:= 0 to N - 1 do
  for J:= 0 to N - 1 do
  if Equiv[I, J] then
  begin
    First[I]:= J;
    Break;
  end;

  NN:= 0;
  NF:= 0;
  for I:= 0 to N - 1 do
  if Reach[I] and (First[I] = I) then
  begin
    Inc(NN);
    Renamed[I]:= NN;
    if Final[I] then Inc(NF);
  end;

  Writeln(NN, ' ', A, ' ', First[P] + 1, ' ', NF);
  for I:= 0 to N - 1 do
  if Reach[I] and (First[I] = I) and Final[I] then Write(Renamed[I], ' ');
  Writeln;
  for I:= 0 to N - 1 do
  if Reach[I] and (First[I] = I) then
  begin
    for J:= 0 to A - 1 do
      Write(Renamed[First[Edges[I, J]]], ' ');
    Writeln;
  end;
end.
```


o kterých už každý snadno vidí, že jsou kontextová a dělají to samé, co původní pravidlo. Slovo $aabbcc$ dostaneme například touto posloupností přepisů:

$$S \to aSBC \to aabCBC \to aabXBC \to aabXCC \to \\ \to aabBCC \to aabbCC \to aabbcC \to aabbcc$$

S naší sadou pravidel zjevně bude všech symbolů a, b, c stejný počet a navíc jediná možnost, kdy se expandování gramatiky může zastavit, je, když jsou symboly utříděné. Gramatika G je tedy schopná vytvořit libovolné slovo z jazyka L a už nic dalšího navíc.

Soutěžní úloha 1: Sestrojte kontextovou gramatiku, která popisuje jazyk $L = \{a^i b^j c^k;\ 1 \le i \le j \le k\}$. Například slova $aabbcc$ či $abbccc$ do L patří, slova $aaabbc$ či cba do L nepatří. *[5 bodů]*

Soutěžní úloha 2: Sestrojte kontextovou gramatiku, která popisuje jazyk $L = \{a^{2^n};\ \forall n \in N\}$, čili jazyk všech slov ze symbolů a, jejichž počet je mocninou dvojky. Tedy například slova a, aa a $aaaa$ do L patří, avšak slovo aaa do L nepatří. *[5 bodů]*

Dejte si zejména pozor na to, aby vámi sestrojená gramatika byla skutečně kontextová a nepoužívala nepovolená pravidla. Vaše řešení by měla obsahovat zdůvodnění, že gramatika dělá to, co má, případně důkaz, že hledáme marně a příslušná gramatika neexistuje.

O nedeterministických zásobníkových automatech, kterým byl věnován předchozí díl seriálu, se dá dokázat, že rozpoznávají právě jazyky popsatelné bezkontextovou gramatikou. Důkaz je však poněkud obtížnější a my si ho předvádět nebudeme. Dají se zavést i podstatně mocnější výpočetní prostředky, než jsou zásobníkové automaty, například různé varianty tzv. *Turingových strojů*. U mnoha z nich se potom ukazuje souvislost s nejrůznějšími typy gramatik. Konkrétně kontextové gramatiky popisují právě jazyky, které jsou rozpoznatelné nedeterministickými Turingovými stroji v paměťovém prostoru omezeném lineárně vzhledem k délce vstupu.

V našem seriálu však již nezbývá dosti časoprostoru na to, abychom si tyto další stroje a gramatiky popsali, natož o nich ukázali něco zajímavého. Ještě bychom však rádi dodali, že teorie formálních jazyků je podkladovou teorií nejdůležitější informatické vědy – teorie složitosti. Rozhodovací algoritmické problémy se totiž dají převést na problém rozpoznávání určitého jazyka. Že se teorie gramatik hodí například při psaní překladačů programovacích jazyků, si čtenář jistě domyslí sám.

Tímto se tedy s vámi loučíme a děkujeme za pozornost, kterou jste formálním jazykům věnovali.

Recepty z programátorské kuchařky

V dnešní kuchařce se budeme zabývat převážně rekurzí a dynamickým programováním. Čemu tedy říkáme rekurze? Rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe. Pojďme se na jednoduchém příkladě podívat, jak může taková funkce vypadat.

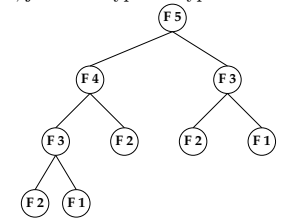
Budeme počítat n-té číslo Fibonacciho posloupnosti. To je posloupnost, jejíž první dva členy jsou jedničky a každý další člen je součtem dvou předchozích. Začíná takto:

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;
```

Podívejme se, jak bude vypadat výpočet čísla F_5:



Vidíme, že volání funkce 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:

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

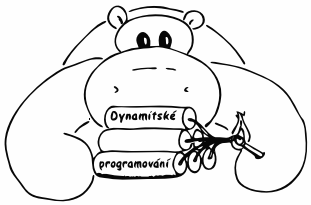
Tedy na spočítání n-tého Fibonacciho čísla spotřebujeme čas alespoň takový, kolik je ono číslo samo. Ale jak velké takové F_n vlastně je? Můžeme využít toho, že:

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

z čehož plyne:

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

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

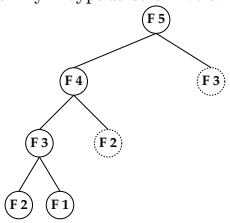


Bude nám k tomu stačit jednoduché pole P o n prvcích, na počátku inicializované nulami. Kdykoliv budeme chtít spočítat některý člen, nejdříve se podíváme do pole, zda jsme ho již jednou nespočetli. A naopak jakmile hodnotu spočítáme, hned si ji do pole poznamenáme:

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

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

Podívejme se, jak nyní vypadá strom volání:



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 změnili 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, ovšem při volání funkce si musíme zapamatovat některé údaje, jako je třeba návratová adresa, parametry funkce a její lokální proměnné, na což spotřebujeme 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 – obvykle se mu říká *dynamické programování* – funguje i pro řadu složitějších úloh. Třeba na tuto:

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

$$\begin{array}{rcccccccccccc} A & = & 2 & 3 & 3 & 1 & 2 & 3 & 2 & 2 & 3 & 1 & 1 & 2 \\ B & = & 3 & 2 & 2 & 1 & 3 & 1 & 2 & 2 & 3 & 3 & 1 & 2 & 2 & 3 \end{array}$$

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


```
  Equiv, Done: array[0..MaxN - 1, 0..MaxN - 1] of Boolean;
  First, Renamed: array[0..MaxN - 1] of Integer;

  procedure MarkReachable(X: Integer);
  var
    I: Integer;
  begin
    if Reach[X] then Exit;

    Reach[X]:= True;
    for I:= 0 to A - 1 do
    MarkReachable(Edges[X, I]);
  end;

  function Equivalent(X, Y: Integer): Boolean;
    function AllEquiv: Boolean;
    var
      I: Integer;
    begin
      AllEquiv:= False;
      for I:= 0 to A - 1 do
      if not Equivalent(Edges[X, I], Edges[Y, I]) then Exit;
      AllEquiv:= True;
    end;
  begin
    if not Done[X, Y] then
    begin
      Done[X, Y]:= True;
      Done[Y, X]:= True;
      Equiv[X, Y]:= True;
      Equiv[Y, X]:= True;
      Equiv[X, Y]:= AllEquiv;
    end;
    Equivalent:= Equiv[X, Y];
    Equiv[Y, X]:= Equiv[X, Y];
  end;

var
  I, J, X: Integer;
begin
  Readln(N, A, P, F);
  Dec(P);
  for I:= 0 to F - 1 do
    Final[I]:= False;
  for I:= 1 to F do
  begin
    Read(X);
    Final[X - 1]:= True;
  end;
  Readln;
  for I:= 0 to N - 1 do
  begin
    for J:= 0 to A - 1 do
    begin
      Read(X);
      Edges[I, J]:= X - 1;
    end;
    Readln;
  end;

  for I:= 0 to N - 1 do
    Reach[I]:= False;

  MarkReachable(P);
```

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

#define MaxN 1024

int main () {
        char Cislo1[MaxN+1], Cislo2[MaxN+1], Cislo3[MaxN+6];
        int Fib1[MaxN+7], Fib2[MaxN+7], Fib3[MaxN+7];       /* první 2 číslice jsou rezervovány pro cifry 0 a −1 */
        int index, i, Delka1, Delka2, Delka;
        printf ("Zadej číslo 1:"); scanf ("%s", Cislo1);
        printf ("Zadej číslo 2:"); scanf ("%s", Cislo2);
        Delka1 = strlen (Cislo1); Delka2 = strlen (Cislo2);
        Delka = ( (Delka1<Delka2) ?Delka2:Delka1) + 7;
        for (index = 0; index < Delka; index++)
                Fib1[index] = Fib2[index] = 0;
        for (index = 0; index < Delka1; index++)
                Fib1[Delka1 - index + 1] = (Cislo1[index] == '1');
        for (index = 0; index < Delka2; index++)
                Fib2[Delka2 - index + 1] = (Cislo2[index] == '1');      /* a je převedeno na pole */
        for (index = 0; index < Delka; index ++)
                Fib3[index] = Fib1[index] + Fib2[index];        /* sečteme po bitech */
        index = Delka - 6;                                      /* poslední zajímavá cifra */
        while (index > 1)                                       /* index je pozice kurzoru */
                if ( (Fib3[index] >= 1) && (Fib3[index + 1] == 1)) {
                        Fib3[index] -= 1;
                        Fib3[++index] = 0;
                        Fib3[++index] = 1;
                } else
                        if (Fib3[index] >= 2) {
                                Fib3[index] -= 2;
                                Fib3[index - 2] += 1;
                                Fib3[++index] = 1;
                        } else
                                index--;                        /* a je sečteno, teď už se jen zbavit cifer 0 a -1 */
        if (Fib3[1] && !Fib3[2]) {
                Fib3[1] = 0;
                Fib3[2] = 1;                                    /* prohodím cifry 0 a 1 */
                index = 2;
        } else
                index = 1;
        while (Fib3[index] && Fib3[index + 1]) {
                Fib3[index] = 0;
                Fib3[++index] = 0;
                Fib3[++index] = 1;                              /* vyhazujeme dvojice jedniček, dokud to jde */
        }
        index = Delka-1;                                        /* poslední definovaná cifra */
        while ( (index > 2) && (Fib3[index] == 0))
                index--;                                        /* a odbouráme nuly */
        i = 0;
        for (; index > 1; index--)
                Cislo3[i++] = (Fib3[index]) ?'1':'0';
        Cislo3[i] = '\0';                                       /* konec řetězce */
        printf ("Součet je: %s\n", Cislo3);
        return 0;
}
```

Úloha 17-3-5 – Jazykozpytcova naděje – program

```
program ksp17-3-5;
const
  MaxN = 100;
  MaxA = 5;
var
  N, A, P, F, NN, NF: Integer;
  Edges: array[0..MaxN - 1, 0..MaxA - 1] of Integer;
  Final, Reach: array[0..MaxN - 1] of Boolean;
```


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:

poslupnost:	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

```
program Podposloupnost;
var
  A, B, C: array[0..MaxN - 1] of Integer;
  LA, LB, LC: Integer;
  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 $O(L(A) \cdot L(B))$. 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 $O(N^2 + M)$, kde N je délka kratší posloupnosti.

Dnešní menu Vám servírovali
Petr Škoda a Martin Mareš

17-3-1 Spisovatel Vilík

Nejprve si uvědomíme, že dvě slova (úseky textu) jsou shodná, pokud obsahují stejné počty jednotlivých písmen. Tj. například „ABCAB“ a „AABBC“ jsou stejná, protože obsahují dvakrát A, dvakrát B a jednou C. Nejdříve si tedy pro každé slovo délky k v textu spočítáme, kolik kterých písmen se v něm vyskytuje – tj. každé pozici p v textu přiřadíme 30-tici čísel $T(p)$, udávající počet příslušných písmen v následujících k znacích textu. Přímočaré řešení by všechna $T(p)$ určilo v čase $O(kN)$, kde N je délka textu. Tuto složitost však můžeme snadno zlepšit na $O(N)$, pokud si povšimneme, že $T(p+1)$ se od $T(p)$ liší pouze ve dvou číslech – přibude jeden výskyt písmena na pozici $p+k$ a ubude výskyt písmena na pozici p. Čili můžeme $T(p)$ spočítat postupně od začátku do konce, a na vytvoření každé 30-tice spotřebujeme pouze konstantní množství času.

Nyní zbývá pouze najít první opakování nějaké 30-tice. Jednou z možností je použít hešování (viz kuchařka druhé série). Nevýhodou je to, že lineární časovou složitost nemáme zaručenu, ale dosáhneme jí pouze v průměrném případě, nebo pokud jsme mocní mágové (tj. umíme zvolit správnou hešovací funkci), randomizovaně.

Řešení, které tuto nevýhodu nemá, je si 30-tice setřídit lexikograficky. Pak jsme schopni jedním průchodem najít opakující se 30-tice (v setříděné posloupnosti budou následovat za sebou), a pokud si navíc pamatujeme, kde se v zadaném textu vyskytovaly, je snadné určit první z nich.

K třídění použijeme RadixSort. Podrobně je popsán v kuchařce druhé série minulého ročníku, zde jen zopakujeme základní myšlenku. RadixSort funguje tak, že nejprve setřídíme posloupnost podle poslední složky 30-tice, pak podle předposlední, ..., a nakonec podle první, přičemž si dáváme pozor, abychom nezměnili pořadí prvků, které se v dané složce shodují. Není těžké si rozmyslet, že výsledná posloupnost pak bude opravdu setříděná – protože poslední třídění proběhlo podle nejdůležitější složky, a mezi slovy, která se v ní shodují, pak rozhoduje pořadí podle druhé nejdůležitější, atd. Třídění podle i-té složky v lineárním čase zvládneme snadno – prvky rozložíme do $k+1$ přihrádek podle hodnoty i-té složky, a pak je vybereme od nejmenší k největší. Budeme tedy vybírat k přihrádek, a samotné kopírování hodnot nám zabere čas N, dohromady bude časová složitost na jeden průchod $O(N+k)=O(N)$, a průchodů je konstantně mnoho – 30.

Takto dosáhneme časové složitosti $O(N)$ i v nejhorším případě. Paměťová složitost bude také $O(N)$.

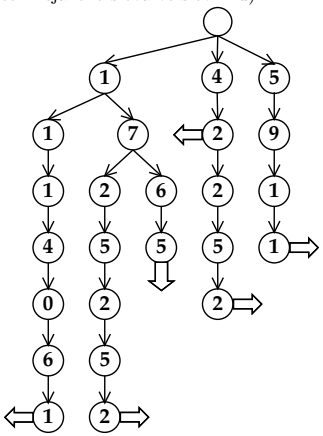
Zdeněk Dvořák

17-3-2 Popleta Truhlík

Problém pana Truhlíka popletl i řadu zkušených řešitelů. Odevzdaná řešení tak byla plná roztodivného zvířectva. Za zmínku určitě stojí housenky složené z příkazů **if** (**then**) **else**, které dosahovaly délky až šestadvaceti řádků. Rovněž šestadvacetihlavá saň **switch/case** se často snažila řešit problém převodu vstupního slovníku na slovník číselný. Řešení obsahující tyto implementační neduhy však (kromě upozornění v podobě velkého FUJ) nebyla nikterak potrestána, přesto bych takovýmto řešitelům doporučil shlédnout kód vzorového řešení. Co se již však neobešlo bez bodových ztrát, byla řešení s exponenciální složitostí, která po nalezení všech možných vět hledala větu nejkratší. Přitom leckde chybělo málo k tomu, aby časová složitost byla optimální!

Nutno podotknouti, že cest ke zdárnému vyřešení úlohy bylo poměrně mnoho. Stalo se tak hlavně proto, že úlohu bylo možné chápat z několika různých pohledů, z nichž ani jeden se po zralé úvaze nedá označit jako nesmyslný či špatný. Někteří řešitelé se tak zaměřili na provokativně zvolené N (počet telefonních čísel, ke kterým si chce Truhlík zapamatovat větu), jiní se snažili rychlost algoritmu poměřovat především s velikostí vstupního slovníku P.

Ukážeme si řešení, které pracuje v čase $O(P+\sum_{i=1}^{n} C_i^2)$. Nejprve si převedeme slova vstupního slovníku do číselné podoby. Takový převod je zřejmě jednoznačný, avšak opačný převod již ne. Číselné řetězce si uložíme do struktury zvané *trie*. Naše trie je strom, ve kterém každý vrchol představuje jednu konkrétní cifru a každý vrchol má právě deset ukazatelů na své potomky. Pokud v trii půjdeme od kořene k listům, pak vrcholy na této cestě tvoří číselný řetězec, který jsme v trii uchovali. Některé vrcholy a všechny listy jsou označené tak, že v nich číselný řetězec končí, u těchto si pamatujeme i ukazatel na slovo do původního slovníku. Příklad takové trie pro vstupní slovník „brok, kuba, je, brekeke, babizna, jedle“, kterému odpovídá slovník číselných řetězců „1765, 5911, 42, 1725252, 1114061, 42252“ je na obrázku (pro přehlednost zobrazujeme jen ty vrcholy, u nichž existuje nějaký ukazatel do slovníku, čili ty, které tvoří začátek nějakého slova ve slovníku):



Když už máme takovou trii postavenou, můžeme se zabývat skládáním věty pro telefonní číslo. K tomu budeme ještě potřebovat dvě pole wc a wi, obě délky telefonního čísla. V průběhu algoritmu bude prvek $wc[k]$, pro nějaké $k \in \{1,\ldots,C_i\}$, označovat minimální počet slov, které jsme doposud potřebovali ke složení cifer $i_1,\ldots,i_k$ telefonního čísla i. Prvek $wi[k]$ pak bude ukazatel na slovo v původním slovníku délky l takové, že číselný řetězec $i_{k-l},i_{k-l+1},\ldots,i_k$ odpovídá tomuto slovu. Na počátku inicializujeme obě pole nulami, postavíme si před sebe telefonní číslo i a začneme v kořeni trie. Načteme první cifru telefonního čísla. Pokud má kořen trie potomka, který odpovídá cifře i_1, přejdeme v trii do tohoto potomka. V případě, že u tohoto potomka je nastaven příznak konce slova, nasta-


```
      printf ("%s -> %s\n", numstr, sentence);          /* vypíšeme větu */
    } else printf ("%s nelze složit\n", numstr);
  }
  return 0;
}
```

Úloha 17-3-3 – Starosta Hafák – program

```
Program Mosty;
const
  MaxN=100;                                                  {maximální počet vrcholů}
  MaxM=10000;
var
  Sousedi:array[1..MaxM] of 1..MaxN;                         {následnici vrcholů}
  V:array[1..MaxN+1] of 1..MaxM+1;  {indexy určující, kde v Sousedi začínají následnici daného vrcholu}
  N,jedn,i,j:integer;                                        {počet vrcholů, počet jednosměrek, čítač}
  Hladina,Spojeno:array[1..MaxN] of integer;                 {jako v kuchařce}

{maximálni zjednosměrnění neorientovaného grafu}
procedure Projdi(otec,x,NovaHladina:integer);
var i:integer;
begin
  Hladina[x]:=NovaHladina;                                   {hladina nově nalezeného vrcholu}
  Spojeno[x]:=Hladina[x];                                    {zatím víme, že z něj vede spojení do}
                                                             {něj samého, tj. do té samé hladiny}
  for i:=V[x] to V[x+1]-1 do                                 {projdi všechny sousedy vrcholu V}
    if Hladina[Sousedi[i]] = -1 then begin                   {pokud sousední vrchol jestě neobjeven}
      Projdi(x,Sousedi[i], NovaHladina+1);                   {zkus z něj další pátrání}
      if Spojeno[Sousedi[i]] < Spojeno[x] then               {našlo se spojení do nižší hladiny}
        Spojeno[x] := Spojeno[Sousedi[i]];
      if Spojeno[Sousedi[i]] <= Hladina[x] then begin
        writeln('(',x,',',Sousedi[i],')');                   {proto je toto DOPŘEDNÁ hrana}
        inc(jedn);
      end;
    end
    else                                {vrchol Sousedi[i] již byl při průchodu navštíven a není otec}
      if (Hladina[Sousedi[i]] < Spojeno[x]) and (Sousedi[i] <> otec) then begin
        Spojeno[x]:=Hladina[Sousedi[i]];                     {ZPĚTNÁ hrana}
        writeln('(',x,',',Sousedi[i],')');
        inc(jedn);
      end;
end;

begin
  readln(N);
  jedn:=0; V[1]:=1; j:=1;
  for i:=1 to N do begin
    while not eoln do begin                                  {následnici vrcholu i}
      read(Sousedi[j]); inc(j);
    end;
    readln();
    V[i+1]:=j;
  end;
  for i:=1 to N do Hladina[i]:=-1;
  for i:=1 to N do if Hladina[i] = -1 then Projdi(0,i,0);    {pro vsechny komponenty grafu}
  writeln('Počet ulic k zjednosměrnění: ',jedn);
end.
```

Úloha 17-3-4 – Myslitel Cibulka – program

```
}
int make_sentence (char *numstr)                    /* utvoří větu pro telefonní číslo do word_count, word_indxs
                                                       vrací nulu, pokud větu pro dané číslo nelze stvořit */
{
  int len=strlen (numstr);                          /* počet cifer telefonního čísla */
  for (int i=0; i<=len; i++) word_count[i]=0;       /* inicializujeme na 0 = nedosazeno */
  for (int i=-1; i<len; i++)                        /* začneme "před" slovem */
  {
    PTRIE t=&trie[0];                               /* začneme v kořeni */
    int wc=0;                                       /* word_count pro i od 0 jinak nula */
    if (i+1) wc=word_count[i];
    if (! (i+1) || wc)
      for (int j=i+1; j<len; j++)
      {
        char num=numstr[j]-'0';                     /* konverze znaku na číslo */
        int next=t->succ[num];                      /* index potomka v trii */
        if (!next) break;                           /* pokud není potomek, končíme průchod */
        t=&trie[next];                              /* zanořme se do potomka */
        if (t->word && (!word_count[j]))            /* končí zde slovo a nebyla tato pozice ještě dosazená? */
        {
          word_count[j]=wc+1;                       /* zapiš nový počet slov */
          word_idxs[j]=t->word;                     /* index slova ve slovníku  */
        }
      }
  }
  return (word_count[len-1]);
}

int main (int argc, char **argv)
{
  scanf ("%d", &w);                                 /* načteme slova do slovníku  */
  for (int i=1; i<=w; i++)
    scanf ("%s", dict[i]);
  for (int i=1; i<=w; i++)                          /* zkonvertujeme slova do číselné podoby */
  {
    char wordnum[MAX_WORD_LEN];                     /* číselný řetězec */
    int len=strlen (dict[i]);                       /* délka slova */
    for (int j=0; j<len; j++)
      wordnum[j]=conv[dict[i][j]-'a'];              /* zkonveruj znak */
    wordnum[len]=0;                                 /* ukončení slova */
    trie_add (wordnum, i);                          /* přidej slovo do trie */
  }
  scanf ("%d", &n);                                 /* zpracujeme jednotlivá čísla */
  for (int i=0; i<n; i++)
  {
    char numstr[MAX_WORD_LEN];                      /* telefonní číslo */
    scanf ("%s", numstr);                           /* načti číslo */
    if (make_sentence (numstr))                     /* zkusime utvořit větu */
    {
      char sentence[2*MAX_WORD_LEN];                /* pole pro větu (i s mezerami) */
      int numlen=strlen (numstr);                   /* délka telefonního čísla */
      int j=word_count[numlen-1]-1+numlen;          /* kam zapisujeme do věty */
      sentence[j]=0;
      while (numlen>0)                              /* pro celé telefonní číslo */
      {
        int idx=word_idxs[numlen-1];                /* index slova ve slovníku */
        int len=strlen (dict[idx]);                 /* délka slova ve slovníku */
        j-=len;                                     /* posuneme se o délku slova vlevo */
        strncpy (&sentence[j], dict[idx], len);     /* zkopírujeme slovo */
        if (--j) sentence[j]='␣';                   /* uděláme mezeru mezi slovy */
        numlen-=len;                                /* posuneme se na další slovo */
      }
```


víme $wc[1] = 1$ a $wi[1]$ položíme rovno ukazateli na slovo do původního slovníku, který v tomto vrcholu máme uložen. Poté načítáme další cifry a procházíme trii tak dlouho, dokud to jde, nebo až do chvíle, kdy vyčerpáme celé telefonní číslo. Zároveň pro každý navštívený vrchol trie, který označuje konec slova, nastavujeme hodnoty polí wc a wi. Udělali jsme tedy jeden průchod a všimněme si, že pole wc nyní obsahuje jedničky na těch místech, které odpovídají délkám slov původního slovníku takovým, že začátek telefonního čísla se shoduje s jejich číselnou reprezentací. Nyní přejděme k první nenulové hodnotě $wc[j]$, postavme se opět do vrcholu trie a spusťme celý průchod znovu s tím, že již pracujeme pouze s ciframi $j, \ldots, C_i$ telefonního čísla. V tomto a dalších průchodech již měníme pole wc a wi pouze tehdy, nebylo-li dané pozice v telefonním čísle ještě dosaženo žádnou posloupností slov. Po C_i takovýchto průchodech je zřejmě $wc[C_i] = 0$ právě tehdy, když telefonní číslo nelze složit pomocí slov ze slovníku. Pro jiné hodnoty číslo složit lze a pro složení dobře poslouží právě pole wi. Stačí na konec věty vypsat slovo, na které ukazuje $wi[C_i]$ a posunou se v poli wi na pozici o délku tohoto slova doleva, tam se nalézá předposlední slovo hledané věty atd.

Jak již bylo řečeno, je časová složitost takového algoritmu rovna $O(P + \sum_{i=1}^{n} C_i^2)$. $O(P)$ je čas potřebný k sestavení trie a pak pro každé telefonní číslo délky C_i děláme až C_i průchodů. Paměťová složitost je ovlivněná hlavně nutností zapamatování vstupního slovníku, když budeme uvažovat, že libovolné telefonní číslo má zanedbatelnou délku oproti velikosti slovníku, a je tedy $O(P)$. Na závěr poznamenejme, že existuje řešení ještě rychlejší. Pokud bude telefonních čísel opravdu mnoho, pak jejich umístění do druhé trie ušetří až logaritmicky mnoho průchodů vůči počtu čísel. Zvýšily by se nám však nároky na paměť, protože bychom si museli pamatovat všechna telefonní čísla.

David Matoušek

17-3-3 Starosta Hafák

Snadno odhalíme, že hranu nesmíme zjednosměrnit právě tehdy, je-li mostem. Most je totiž hrana, jejímž odebráním se graf rozpadne na dvě komponenty souvislosti. Je tedy jediným spojením mezi těmito dvěma komponentami, a když ho učiním propustným pouze pro jeden směr, tak se dostanu z první komponenty do druhé, ale ne opačně.

Tady nám poslouží algoritmus na hledání mostů z kuchařky, který dá lehce upravit pro naše účely.

Pro každý vrchol v si stejně jako v kuchařce budeme pamatovat, v jaké hloubce vůči kořeni se nachází (kořen v hloubce 0, synové 1, synové synů 2, ...) a do jaké nejnižší hladiny se umím dostat z podstromu s kořenem v. Zde se ovšem v kuchařce vyskytla chyba, kterou naštěstí mnozí hravě odhalili. Při hledání spojení do nižší hladiny nesmíme vůbec uvažovat hranu mezi otcem vrcholu v a vrcholem v. Ta totiž způsobí, že cestu do nižší hladiny najdeme vždy (každý vrchol má otce), ale není to kýžená kružnice, nýbrž dvakrát započítaná hrana, po které jsme do vrcholu v přišli. Zrada! Takhle totiž nikdy nenajdeme žádný most.

Pokud se z podstromu s kořenem v (s otcem u) neumíme dostat do hladiny nižší, než je hladina vrcholu v, pak do tohoto podstromu vede jedna jediná hrana, a to hrana (u,v). Ta je tedy mostem, v zájmu propustnosti v obou směrech ji ponecháme obousměrnou.

Pokud se z podstromu s kořenem v (s otcem u) umíme dostat do hladiny nižší, než je hladina vrcholu v, bez použití hrany (u,v), tak jsme právě našli kružnici „$u \to v \to$ nějaké vrcholy v podstromu $v \to$ (vrcholy v nižší hladině než vrchol v, ne nutně) $\to u$“. A kružnici můžeme směle zjednosměrnit, snadno vidíme, že zjednosměrnění kružnice neublíží dosažitelnosti vrcholům na této kružnici, prostě budeme „kroužit“ dokola.

Také si můžeme všimnout, že na této kružnici jsou všechny hrany dopředné, kromě té jediné, která se z hlubší hladiny vrací do nižší, a ta je zpětná.

Nakonec ještě musíme vymyslet, jak sjednotit zjednosměrňování více kružnic. Ale k tomu se stačí dohodnout, že dopředné hrany budeme zjednosměrňovat „dopředu“, čili otec $\to$ syn, a zpětné ve směru od vrcholu na hlubší hladině k vrcholu na nižší hladině („dozadu“).

Algoritmus našel všechny mosty a ponechal je obousměrné, našel ale i všechny kružnice a zorientoval je ve shodném směru. Takže jsme nezjednosměrnili nic závadného a naopak jsme zjednosměrnili maximum. Při reprezentaci grafu seznamem následníků získáváme časovou i paměťovou složitost $O(N + M)$.

Škoda, že většinu řešitelů kuchařka sváděla k řešení „pomocí algoritmu z kuchařky odstraním mosty, pak v dalším průchodu zjednosměrním graf a pak ještě vypíšu všechny zjednosměrněné hrany“. Ve skutečnosti všechno můžu udělat přímo v jednom jediném průchodu grafem, stačí si uvědomit, že algoritmus na hledání mostů nejen že hledá mosty, ale i detekuje dopředné a zpětné hrany, a tak můžeme výsledky ihned vypisovat.

Za funkční řešení v čase O(N+M) bylo možno získat 9 bodů a kdo to všechno zvládl v jednom průchodu grafem, dostal 10 bodů.

Jana Kravalová

17-3-4 Myslitel Cibulka

Myslitel Cibulka by z Vás asi radost neměl. Došlé řešení se totiž dala rozdělit do tří skupin. V první, největší, byla řešení kvadratická. V druhé ta, u nichž autoři ani nenaznačili, proč by měly skončit (a nebylo to, jako u většiny programů zřejmé z toho, že tento cyklus proběhne 3×, jiný $N\times$, ...). Ta, ačkoliv byla, jak dále ukážeme, lineární, jsem hodnotil o něco hůř, jelikož byla opravdu triviální, a dokázat o nich, že jsou opravdu rychlá, je mnohem obtížnější, než vymyslet kvadratické řešení. Navíc vymyšlení kvadratického řešení bylo také obtížnější, než vytvoření tohoto triviálního.

No a konečně ve třetí skupině (dá-li se to tak nazvat, skoro by se dalo hovořit o výjimkách potvrzujících pravidlo) byla řešení lineární.

Tak ono „triviální“ řešení. Vezmeme FiBoNaCiHo čísla a sečteme je „po bitech“, tj. po jednotlivých cifrách. Na některých místech se vyskytnou dvojky, kterých se potřebujeme zbavit, a také číslo normalizovat. Jak na to?

Zavedeme kurzor. Budeme předpokládat, že číslo je vpravo od kurzoru normalizované, tj. kdybychom zapomněli (nebo je nastavili na nulu) na všechny cifry nižší, než je pozice kurzoru, tak dostaneme normalizované číslo. A program bude opakovat několik operací, které zřejmě nemění hodnotu čísla a zachovávají výše zmíněný invariant, dokud kurzor nenarazí na konec čísla. Zřejmě když dojde kurzor na „nultou“ cifru (cifry čísla indexujeme od jedničky) tak je celé číslo normalizované a je hotovo.

Pro pohodlnější práci zavedeme ještě nultou a mínus první cifru a zadefinujeme $F_0 = 1$ a $F_{-1} = 0$. Na začátku nastavíme cifry na nula a na konci se jich opět zbavíme. To nám umožní psát pravidlo $2 \cdot F_n = F_{n+1} + F_{n-2}$ pro každé $n \geq 1$ (nemám rád okrajové podmínky), a zjednoduší dále některé úvahy.

Teď tělo cyklu. Označme C_i i-tou cifru zpracovávaného čísla a k kurzor (resp. index cifry, na kterou ukazuje).

1) Pokud $C_k \geq 1$ a $C_{k+1} = 1$, tak hodnotu cifer k a $k+1$ snížíme o 1, hodnotu C_{k+2} nastavíme na 1 (musela být 0, jelikož číslo vpravo od kurzoru je, jak předpokládáme, normalizované) a kurzor o 2 zvětšíme (mohly se nám vedle sebe dostat dvě jedničky).

2) Jinak pokud je $C_k \geq 2$ (a $C_{k+1} = 0$, jelikož jinak se provedla předchozí větev), pak C_{k+1} nastavíme na 1, C_{k-2} zvětšíme o 1, C_k o 2 snížíme a kurzor posuneme o jedna doprava (opět možnost dvojice jedniček).

3) A pokud jsme se ještě na nějaké podmínce nezachytili, pak je na pozici kurzoru nula, nebo jednička které nula předchází, a proto se můžeme „beztrestně“ kurzorem posunout o jednu pozici doleva, aniž bychom porušili náš základní invariant.

Až tento cyklus doběhne, musíme se zbavit C_0 a C_{-1}. S C_{-1} není problém, jelikož F_{-1} je 0 a cokoliv krát nula je stále jen nula, takže tuhle cifru můžeme jednoduše vynulovat. Nyní co s C_0. Jak ukážeme později, může nabývat jen hodnot nula a jedna, stačí vyšetřit případ, kdy $C_0 = 1$. Pokud je C_1 nula, tak prohodíme nultou a mínus první cifru (protože $F_0 = F_1$) a jsme hotovi. Pokud C_0 i C_1 jsou jedna, tak použijeme pravidlo o dvou jedničkách za sebou a převedeme je na jedničku na druhé cifře. Samozřejmě musíme si dát pozor, jelikož tyto operace mohou narušit normalizaci tím, že se vyskytnou dvě jedničky vedle sebe. Nicméně tato „porucha“ se vyskytuje u čerstvě zapsané jedničky a protože každá redukce dvou jedniček na jednu sníží ciferný součet, zvládneme to vyřešit v lineárním čase.

Je v celku zřejmé, že tělo cyklu nemění hodnotu čísla a že neporuší výše definovaný invariant. Horší je to ale s tím, za jak dlouho tento program doběhne, doběhne-li vůbec. Ještě než se do toho pustíme, dokažme si jedno pomocné tvrzení.

Lemma: Každou cifru, před tím, než se na ní dostaneme kurzorem, můžeme zvýšit nejvýše o jedna. (Pokud považujeme všechny nulové cifry vpravo od čísla za již prošlé)

Důkaz: Nejdříve takové drobné pozorování. Označme L nejnižší cifru, na kterou se kurzor při běhu programu zatím dostal. Potom všechny cifry vpravo od L jsou menší než 2. To se dokáže indukcí. Nejdříve si všimněme, že jediná operace, která v těle cyklu je schopná zvětšit jedničku na dvojku, je (2), a ta to dělá vlevo od kurzoru. Pak se podíváme na to, že změnu L je schopná provést jen (3), a ta to udělá jen tehdy, je-li C_L menší než 2. A jelikož vždy platí $k \geq L$, jsme hotovi (alespoň s tímto pozorováním).

Nyní k samotnému lemmatu. Dokážeme ho indukcí dle čísla cifry (označme ho N). Na počátku je $k = L$ rovno délce čísla a pro všechna $N \geq$ (délka čísla) to zřejmě platí z předpokladů. Nyní předpokládejme, že to platí pro $N+1$ a větší. Všimněme si opět, že jediná operace, která mění hodnotu cifry vlevo od kurzoru, je (2), a ta to dělá právě o 2 pozice vlevo. Uvažujme L stejné jako v pozorování, na začátku důkazu. Mohou nastat 4 případy:

- $L > N + 2$. Protože tělo cyklu pracuje s ciframi nejvýše o 2 pozice vlevo, tak je tato cifra ještě „nedotčená“ a má svou původní hodnotu.
- $L \leq N$. Potom jsme již přes tuto cifru přešli a není co řešit.
- $L = N+1$. Potom z úvodního pozorování plyne, že C_{N+2} je menší než 2 a proto se (2) na pozici $N + 2$ již během programu neprovede, a proto se hodnota C_N, dokud kurzor nedojde na N, nezmění.
- $L = N + 2$. Z indukčního předpokladu víme, že hodnota C_{N+2} se zvýšila nejvýše o 1 a proto je menší, nebo rovna 3 (po sečtení bitů je maximálně 2). Pokud provedeme (2), pak hodnota C_{N+2} klesne na 1 nebo 0. Protože pro všechny cifry vpravo od L platí, že jsou nejvýše 1, a jediná operace, která je schopna zvýšit hodnotu C_{N+2} nad jedna je (2) a to jen tehdy, je-li kurzor na pozici $N + 4$ (což je vpravo od L), provede se (2) na pozici $N + 2$ nejvýše jednou.

Tím je lemma dokázáno.

Důsledek 1: Hodnota jakékoliv cifry je během výpočtu nejvýše 3, protože na začátku je nejvýše 2, dokud na ní nedojde kurzor. Zvýšit se může maximálně o 1, ve chvíli, kdy L stojí na této cifře, se tato cifra nemůže zvětšovat (důkaz analogicky jako 4. případ v důkazu lemmatu) a pokud L je vlevo od cifry, pak je tato cifra 1 nebo 0.

Důsledek 2: Hodnota C_0 a C_{-1} je maximálně 1, protože začínaly na nule a kurzor na nich při provádění těla cyklu nemůže stát, tedy zvýší se maximálně na 1.

No a nyní konečně k (ne)konečnosti algoritmu a jeho časové složitosti. Použijeme k tomu techniku, která se nazývá metoda potenciálu. Původně je určená k dokazování konečnosti algoritmu, ale v některých případech ji lze použít i ke stanovení složitosti algoritmu. A co to tedy je?

Odvodíme, uhodneme, nebo jinak stanovíme funkci φ (nějaké parametry, kterými charakterizujeme stav výpočtu (ne nutně jednoznačně)), která je:

a) *klesající,* čili po provedení nějaké části výpočtu, u které je zřejmé, že doběhne za $O(\text{cosi})$ (u konečnosti stačí jen to, že doběhne) se její hodnota ostře sníží.

b) *klesá „dost“ rychle,* to znamená, že existuje konstanta $\varepsilon > 0$ taková, že při poklesu v `a)` poklesne φ alespoň o ε. (Pozn. je-li funkce φ celočíselná, což v drtivé většině případů je, pak je `b)` splněno automaticky.)

Pak pokud ke každým vstupním datům dovedeme shora i zdola omezit (tj. pro každá vstupní data existují konstanty K a L takové, že po celou dobu výpočtu platí $K \leq \varphi \leq L$), výpočet je konečný. (ε se „vejde“ do intervalu $[K; L]$ jen konečněkrát). Proto jsme také požadovali „podivnou“ podmínku `b)`. Vyhnuli jsme se asymptotickému blížení φ ke K a podobným patologickým případům).

Ale vraťme se k určení složitosti našeho programu. Označme k pozici kurzoru, σ součet cifer FiBoNaCiHo čísla a X pozici nejpravějšího výskytu cifry, která je větší než 1 (pokud jsou všechny cifry menší než 2, pak zadefinujeme $X = 0$). Vezměme tento potenciál: $\varphi(k, \sigma, X) = k + 2X + 3\sigma$. Pro parametry zřejmě platí ,že $k \geq 0$, $X \geq 0$ a $\sigma \geq 1$ a tedy po celou dobu běhu programu je $\varphi \geq 3$.

Na druhou stranu, označme N délku delšího čísla. Na začátku výpočtu zřejmě platí, že $X \leq N$ a $\sigma \leq 2N$ a $k = N$. Protože, jak ukážeme v dalším odstavci, je φ klesající, platí během výpočtu $\varphi \leq 9N$.


```
    T[p].pocty[kod (vstup[p - 1])]--;
  }
for (i = 0; i < n - k + 1; i++)                    /* Setřídíme 30-tice. */
  T_sorted[i] = &T[i];
radixsort (T_sorted, n - k + 1, k);

min_opak = n;
for (i = 1; i < n - k + 1; i++)                    /* A najdeme první opakující se. */
  if (stejne_pocty (T_sorted[i], T_sorted[i - 1]))
    {
      p = T_sorted[i] - T;
      if (p < min_opak)
        min_opak = p;
    }
if (min_opak == n)
  printf ("Žádné opakování.\n");
else
  printf ("Žádné opakování do pozice %d.\n", min_opak + k - 1);
return 0;
}
```

Úloha 17-3-2 – Popleta Truhlík – program

```
#include <stdio.h>
#include <string.h>
#define MAX_WORDS 100                              /* maximální počet slov ve slovníku */
#define MAX_WORD_LEN 100                           /* maximální délka slova */
#define MAX_DICT_SIZE 1000                         /* maximální velikost slovníku */
#define MAX_NUM_LEN 100                            /* maximální délka telefonního čísla */

const char conv[26]={'1', '1', '1', '2', '2', '3', '3', '3', '4', '4',   /* konvertovací tabulka */
                     '5', '5', '5', '6', '6', '7', '7', '7', '8', '8',
                     '9', '9', '9', '0', '0', '0'};

char dict[MAX_WORDS][MAX_WORD_LEN];                /* slovník */

typedef struct _TRIE                               /* struktura trie */
{
  int succ[10];                                    /* potomci */
  int word;                                        /* ukazatel do slovníku nebo 0 */
} TRIE, *PTRIE;

int n, w;                                          /* počet čísel, počet slov */
TRIE trie[MAX_DICT_SIZE];                          /* trie jako pole vrcholů */
int trie_count;                                    /* počet vrcholů trie */
int word_count[MAX_NUM_LEN];                       /* počet slov na dosažení cisla */
int word_idxs[MAX_NUM_LEN];                        /* indexy slov ve větě */

void trie_add (char *s, int idx)                   /* přidá číselný řetězec idx-tého slova do trie */
{
  PTRIE t=&trie[0];                                /* začneme v kořeni trie */
  int len=strlen (s);                              /* délka řetězce */
  for (int i=0; i<len; i++)
  {
    char num=s[i]-'0';                             /* konverze znaku na číslo */
    int next=t->succ[num];                         /* index potomka v trii */
    if (!next)                                     /* existuje potomek? */
    {
      trie_count++;                                /* pokud není potomek, vytvoříme ho */
      t->succ[num]=trie_count;                     /* označ potomka */
      next=trie_count;                             /* půjdeme do nového vrcholu */
    }
    t=&trie[next];                                 /* zanoření o level níž */
    if (i==len-1) t->word=idx;                     /* jsme již na konci slova */
  }
  return;
```

```
}
void casesort (tice *to_sort[], unsigned l, unsigned k, unsigned slozka)      /* Setřídí L 30-tic v to_sort podle složky. */
{
  tice *tmp[MAXN];
  unsigned case_size[MAXK + 1];
  unsigned case_begin[MAXK + 1];
  unsigned i;

  for (i = 0; i <= k; i++)
    case_size[i] = 0;

  for (i = 0; i < l; i++)
    case_size[to_sort[i]->pocty[slozka]]++;

  case_begin[0] = 0;
  for (i = 1; i <= k; i++)
    case_begin[i] = case_begin[i - 1] + case_size[i - 1];

  for (i = 0; i < l; i++)
    tmp[case_begin[to_sort[i]->pocty[slozka]]++] = to_sort[i];

  for (i = 0; i < l; i++)
    to_sort[i] = tmp[i];
}
void radixsort (tice *to_sort[], unsigned l, unsigned k)          /* Setřídí lexikograficky l 30-tic v poli to_sort. */
{
  int i;

  for (i = 29; i >= 0; i--)
    casesort (to_sort, l, k, i);
}
unsigned kod (char ch)                                            /* Vrátí kód znaku ch. */
{
  if ('a'<= ch && ch <= 'z')
    return ch - 'a';
  if ('A'<= ch && ch <= 'Z')
    return ch - 'A';
  if (ch == '_')
    return 'z'- 'a'+ 1;
  if (ch == '.')
    return 'z'- 'a'+ 2;
  if (ch == '?')
    return 'z'- 'a'+ 3;
  if (ch == '!')
    return 'z'- 'a'+ 4;

  abort ();
}
int main (void)
{
  char vstup[MAXN + 1];
  tice T[MAXN];
  tice *T_sorted[MAXN];
  unsigned n, k;
  unsigned i, p;
  unsigned min_opak;

  scanf ("%d%d%s", &n, &k, vstup);

  memset (&T[0], 0, sizeof (tice));

  for (i = 0; i < k; i++)                                         /* Spočítáme četnosti písmen v k-ticích. */
    T[0].pocty[kod (vstup[i])]++;

  for (; i < n; i++)
    {
      p = i - k + 1;

      T[p] = T[p-1];
      T[p].pocty[kod (vstup[i])]++;
```


Nyní, co provede s k, X a σ tělo cyklu. Rozebereme jednotlivé větve zvlášť. Čárkované proměnné budou proměnné po provedení těla cyklu, nečárkované před tím.

1) Zřejmě nevytváří žádné číslo větší než 2, ale možná nějaké snižuje. Proto $X' \leq X$. Ciferný součet se sníží o 1, proto $\sigma' = \sigma + 1$. A kurzor posuneme o 2 doprava, tedy $k' = k + 2$. Z toho plyne, že $\varphi' \leq \varphi - 1$, tedy $\varphi' < \varphi$.
2) S ciferným součtem se nic neděje (jen ty dvě jedničky přesuneme na jiné pozice). Protože momentálně pracujeme na nejpravější cifře, která je větší než 1 (viz. pozorování v důkazu lemmatu) a protože každá cifra je během výpočtu nejvýše 3 (viz. důsledek 1), klesá cifra provedením (2) na nejvýše 1 a proto platí $X' \leq X - 1$. A kurzor se posune o jedno místo doprava. Proto $\varphi' \leq \varphi - 1$.
3) Poslední případ jen hne s kurzorem a s číslem nic nedělá. Proto φ pro změnu poklesne o 1.

Protože φ je zřejmě celočíselná, splnili jsme všechny požadavky na potenciál a výpočet tedy skončí. Nyní se na potenciál podívejme podrobněji. Během celého provádění cyklu může klesnout nejvýše o $9N$ a klesá vždy alespoň o 1. Tedy celý cyklus se provede $O(N)$ krát. Protože provedení těla cyklu stihneme v konstantním čase, můžeme tvrdit, že celý cyklus doběhne v lineárním čase. Vzhledem k tomu, že ostatní části (výpis, načtení, a zbavení se jedničky na pozici 0) zvládneme v lineárním čase, běhá celý program v lineárním čase. Paměťová složitost je zřejmě lineární.

A je to. Uffff...

Pavel Čížek

Poznámka na okraj: ačkoliv je opravdu pozoruhodné, že o tak triviálním řešení se dá dokázat, že běží v lineárním čase, zdělení ze složitosti důkazu není na míste: existují i jiná lineární řešení, která jsou trochu pracnější na naprogramování, ale obejdou se bez složitého dokazování. Například můžeme zkusit přičítat druhé číslo k prvnímu po číslicích a po každé číslici normalizovat. To sice pro některá čísla bude kvadratické, ale stačí si všimnout, že kvadratické chování nastává pouze, objeví-li se blok číslic typu $010101\ldots01$. To můžeme napravit „kompresí“ zpracovávaného čísla – v případě, že za kurzorem následuje takovýto blok, si budeme udržovat pouze jeho délku a ne pokaždé celý blok zkoumat. Hezký algoritmus založený na této myšlence najdete například na *http://www.ucw.cz/~mj/papers/fibonacci/*.

Martin Mareš

17-3-5 Jazykozpytcova naděje

Přestože je to úloha na automaty, k jejímu řešení se dalo využít znalosti grafových algoritmů. Automat si představíme jako orientovaný graf, ve kterém je pro každý stav jeden vrchol. Z každého vrcholu vede a orientovaných hran, každá pro jedno písmeno abecedy. Řešení rozdělíme do dvou částí.

Odstranění nedostupných stavů automatu znamená odstranit ty vrcholu grafu, do kterých se nedá dostat z počátečního vrcholu p – vstupního stavu. Projdeme graf do hloubky z počátečního vrcholu a označíme si, kam všude jsme se dostali. Ostatní vrcholy jsou nedostupné. Jediná věc, na kterou si musíme dávat pozor, je, abychom označili každý vrchol pouze jednou.

Složitějším problémem je nalezení ekvivalentních stavů automatu. Ekvivalentní stavy jsou ty, které stejně odmítají a přijímají každé slovo. Nechť máme p, q ekvivalentní stavy a slovo u začínající na $x \in A$. Pak $p' = \delta(p, x)$ a $q' = \delta(q, x)$ jsou také ekvivalentní. V opačném případě bychom nalezli slovo v, na které automat ve stavu p' a q' odpoví jinak, a přidali před něj x.

Vytvoříme si pole velikosti $n \cdot n$, do kterého si uložíme, zda jsou stavy i a j ekvivalentní. Pole naplníme hodnotami a pak z něj zkonstruujeme redukovaný automat. Na začátku označíme každý vrchol ekvivalentní sám se sebou a každou dvojici, kde jeden ze stavů je přijímací a druhý nikoli, označíme jako neekvivalentní. Ekvivalenci ostatních dvojic vrcholů budeme zjišťovat rekurzivní funkcí, která pro stavy p a q(parametry) provede:

- Prozatímně je označí jako ekvivalentní.
- Zeptá se pomocí rekurzivního volání, zda jsou ekvivalentní stavy $\delta(p, x)$ a $\delta(q, x)$ pro každé písmeno v abecedě A.
- Pokud ne, přeznačí stavy jako neekvivalentní.

Teď už můžeme vytvořit redukovaný automat. Místo každé skupiny ekvivalentních stavů vytvoříme jeden stav a tyto stavy pospojujeme.

V algoritmu jsme použili pole, kde jsme uchovávali přechodovou funkci a pole ekvivalencí. Paměťová složitost je tedy $O(n^2 + n \cdot a)$. Časově nejnáročnější operací v algoritmu je výpočet ekvivalence. Pro každou dvojici stavů se ekvivalence počítá právě jednou a zahrnuje a dotazů na ekvivalenci dalších stavů. Časová složitost je $O(n^2 \cdot a)$.

Petr Škoda

Úloha 17-3-1 – Spisovatel Vilík – program

```
#include <stdio.h>

#define MAXN 1000
#define MAXK 1000

typedef struct
{
  int pocty[30];
} tice;

int stejne_pocty (tice *a, tice *b)                               /* Vrátí 1 pokud a a b jsou stejné. */
{
  unsigned i;

  for (i = 0; i < 30; i++)
    if (a->pocty[i] != b->pocty[i])
      return 0;

  return 1;
```

```
}
void casesort (tice *to_sort[], unsigned l, unsigned k, unsigned slozka)      /* Setřídí L 30-tic v to_sort podle složky. */
{
  tice *tmp[MAXN];
  unsigned case_size[MAXK + 1];
  unsigned case_begin[MAXK + 1];
  unsigned i;

  for (i = 0; i <= k; i++)
    case_size[i] = 0;

  for (i = 0; i < l; i++)
    case_size[to_sort[i]->pocty[slozka]]++;

  case_begin[0] = 0;
  for (i = 1; i <= k; i++)
    case_begin[i] = case_begin[i - 1] + case_size[i - 1];

  for (i = 0; i < l; i++)
    tmp[case_begin[to_sort[i]->pocty[slozka]]++] = to_sort[i];

  for (i = 0; i < l; i++)
    to_sort[i] = tmp[i];
}
void radixsort (tice *to_sort[], unsigned l, unsigned k)          /* Setřídí lexikograficky l 30-tic v poli to_sort. */
{
  int i;

  for (i = 29; i >= 0; i--)
    casesort (to_sort, l, k, i);
}
unsigned kod (char ch)                                            /* Vrátí kód znaku ch. */
{
  if ('a'<= ch && ch <= 'z')
    return ch - 'a';
  if ('A'<= ch && ch <= 'Z')
    return ch - 'A';
  if (ch == '_')
    return 'z'- 'a'+ 1;
  if (ch == '.')
    return 'z'- 'a'+ 2;
  if (ch == '?')
    return 'z'- 'a'+ 3;
  if (ch == '!')
    return 'z'- 'a'+ 4;

  abort ();
}
int main (void)
{
  char vstup[MAXN + 1];
  tice T[MAXN];
  tice *T_sorted[MAXN];
  unsigned n, k;
  unsigned i, p;
  unsigned min_opak;

  scanf ("%d%d%s", &n, &k, vstup);

  memset (&T[0], 0, sizeof (tice));

  for (i = 0; i < k; i++)                                         /* Spočítáme četnosti písmen v k-ticích. */
    T[0].pocty[kod (vstup[i])]++;

  for (; i < n; i++)
    {
      p = i - k + 1;

      T[p] = T[p-1];
      T[p].pocty[kod (vstup[i])]++;
```


Nyní, co provede s k, X a σ tělo cyklu. Rozebereme jednotlivé větve zvlášť. Čárkované proměnné budou proměnné po provedení těla cyklu, nečárkované před tím.

1) Zřejmě nevytváří žádné číslo větší než 2, ale možná nějaké snižuje. Proto $X' \leq X$. Ciferný součet se sníží o 1, proto $\sigma' = \sigma + 1$. A kurzor posuneme o 2 doprava, tedy $k' = k + 2$. Z toho plyne, že $\varphi' \leq \varphi - 1$, tedy $\varphi' < \varphi$.
2) S ciferným součtem se nic neděje (jen ty dvě jedničky přesuneme na jiné pozice). Protože momentálně pracujeme na nejpravější cifře, která je větší než 1 (viz. pozorování v důkazu lemmatu) a protože každá cifra je během výpočtu nejvýše 3 (viz. důsledek 1), klesá cifra provedením (2) na nejvýše 1 a proto platí $X' \leq X - 1$. A kurzor se posune o jedno místo doprava. Proto $\varphi' \leq \varphi - 1$.
3) Poslední případ jen hne s kurzorem a s číslem nic nedělá. Proto φ pro změnu poklesne o 1.

Protože φ je zřejmě celočíselná, splnili jsme všechny požadavky na potenciál a výpočet tedy skončí. Nyní se na potenciál podívejme podrobněji. Během celého provádění cyklu může klesnout nejvýše o $9N$ a klesá vždy alespoň o 1. Tedy celý cyklus se provede $O(N)$ krát. Protože provedení těla cyklu stihneme v konstantním čase, můžeme tvrdit, že celý cyklus doběhne v lineárním čase. Vzhledem k tomu, že ostatní části (výpis, načtení, a zbavení se jedničky na pozici 0) zvládneme v lineárním čase, běhá celý program v lineárním čase. Paměťová složitost je zřejmě lineární.

A je to. Uffff...

Pavel Čížek

Poznámka na okraj: ačkoliv je opravdu pozoruhodné, že o tak triviálním řešení se dá dokázat, že běží v lineárním čase, zdělení ze složitosti důkazu není na míste: existují i jiná lineární řešení, která jsou trochu pracnější na naprogramování, ale obejdou se bez složitého dokazování. Například můžeme zkusit přičítat druhé číslo k prvnímu po číslicích a po každé číslici normalizovat. To sice pro některá čísla bude kvadratické, ale stačí si všimnout, že kvadratické chování nastává pouze, objeví-li se blok číslic typu $010101\ldots01$. To můžeme napravit „kompresí“ zpracovávaného čísla – v případě, že za kurzorem následuje takovýto blok, si budeme udržovat pouze jeho délku a ne pokaždé celý blok zkoumat. Hezký algoritmus založený na této myšlence najdete například na *http://www.ucw.cz/~mj/papers/fibonacci/*.

Martin Mareš

17-3-5 Jazykozpytcova naděje

Přestože je to úloha na automaty, k jejímu řešení se dalo využít znalosti grafových algoritmů. Automat si představíme jako orientovaný graf, ve kterém je pro každý stav jeden vrchol. Z každého vrcholu vede a orientovaných hran, každá pro jedno písmeno abecedy. Řešení rozdělíme do dvou částí.

Odstranění nedostupných stavů automatu znamená odstranit ty vrcholu grafu, do kterých se nedá dostat z počátečního vrcholu p – vstupního stavu. Projdeme graf do hloubky z počátečního vrcholu a označíme si, kam všude jsme se dostali. Ostatní vrcholy jsou nedostupné. Jediná věc, na kterou si musíme dávat pozor, je, abychom označili každý vrchol pouze jednou.

Složitějším problémem je nalezení ekvivalentních stavů automatu. Ekvivalentní stavy jsou ty, které stejně odmítají a přijímají každé slovo. Nechť máme p, q ekvivalentní stavy a slovo u začínající na $x \in A$. Pak $p' = \delta(p, x)$ a $q' = \delta(q, x)$ jsou také ekvivalentní. V opačném případě bychom nalezli slovo v, na které automat ve stavu p' a q' odpoví jinak, a přidali před něj x.

Vytvoříme si pole velikosti $n \cdot n$, do kterého si uložíme, zda jsou stavy i a j ekvivalentní. Pole naplníme hodnotami a pak z něj zkonstruujeme redukovaný automat. Na začátku označíme každý vrchol ekvivalentní sám se sebou a každou dvojici, kde jeden ze stavů je přijímací a druhý nikoli, označíme jako neekvivalentní. Ekvivalenci ostatních dvojic vrcholů budeme zjišťovat rekurzivní funkcí, která pro stavy p a q(parametry) provede:

- Prozatímně je označí jako ekvivalentní.
- Zeptá se pomocí rekurzivního volání, zda jsou ekvivalentní stavy $\delta(p, x)$ a $\delta(q, x)$ pro každé písmeno v abecedě A.
- Pokud ne, přeznačí stavy jako neekvivalentní.

Teď už můžeme vytvořit redukovaný automat. Místo každé skupiny ekvivalentních stavů vytvoříme jeden stav a tyto stavy pospojujeme.

V algoritmu jsme použili pole, kde jsme uchovávali přechodovou funkci a pole ekvivalencí. Paměťová složitost je tedy $O(n^2 + n \cdot a)$. Časově nejnáročnější operací v algoritmu je výpočet ekvivalence. Pro každou dvojici stavů se ekvivalence počítá právě jednou a zahrnuje a dotazů na ekvivalenci dalších stavů. Časová složitost je $O(n^2 \cdot a)$.

Petr Škoda

Úloha 17-3-1 – Spisovatel Vilík – program

```
#include <stdio.h>

#define MAXN 1000
#define MAXK 1000

typedef struct
{
  int pocty[30];
} tice;

int stejne_pocty (tice *a, tice *b)                               /* Vrátí 1 pokud a a b jsou stejné. */
{
  unsigned i;

  for (i = 0; i < 30; i++)
    if (a->pocty[i] != b->pocty[i])
      return 0;

  return 1;
```