Milí řešitelé a řešitelky!

První série 22. ročníku Korespondenčního semináře z programování je za námi, už byl nejvyšší čas vydat sérii druhou. S novou sérii přichází i malá změna v termínu odevzdávání – vaše řešení k nám **musí dorazit** do 8. hodiny ranní dne 7. prosince 2009. Pokud tedy řešení odesíláte poštou, vhoďte jej do schránky do středy 2. 12.

I přes naše nejvyspělejší technologie se nám sem tam přihodí, že se z některých řešení vytratí číslo úlohy či jméno autora. Dokonce jeden záludný řešitel, zkoušeje nás, na tři různá řešení různých úloh napsal „úloha 22-1-2“. Prosíme, i do řešení, která nám posíláte přes submitovátko, pište kompletní hlavičky, jako byste je psali rukou na papír.

Elektronická řešení přijímáme na obvyklé stránce *http://ksp.mff.cuni.cz/submit/*. Podporujeme i papírový protokol na následující adrese:

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

Druhá série dvacátého druhého ročníku KSP

Letošní druhý příběh nesoucí honosný název „Hint“, velmi volně navazující na první dílko, sepsal Martin Böhm. Můžeme vám slíbit, že tentokrát si se stroji času hrát nebudeme – na soustředení jsme se vytrestali dostatečně. Další díl seriálu o Erlangu vám přináší Michal Vaner.

Mimochodem, také jste si jistě všimli, že v této sérii se objeví příklad s magickým číslem 22-2-2, což je vzácnost, která se objevuje jednou za 11 let, 1 sérii a jeden příklad k tomu. Není to důvod k malým oslavám?

Jsem si jist, že každý máme nějakou superschopnost. Někdo hravě vyřeší ty nejtěžší programátorské úlohy, jiný umí dělat radost jiným lidem. A někteří z nás opravdu rychle běhají a driblují s míčem. Já mám Hint. I když možná mají všichni Hint, jen se o tom nebavíme.

Nespadl na mne meteorit a nekousl mne ozářený hroch. Prostě jsem se jednoho dne probudil a věděl jsem, že mám v hlavě Hint. Jak Hint funguje? Občas jdu po ulici a přemýšlím, jestli si koupit zmrzlinu. V tu chvíli se mi v hlavě aktivuje Hint a řekne, jaká z těchto dvou možností je pro mne lepší. Na maličký okamžik jako kdybych mohl spustit prohledávání obou životů a porovnat návratové hodnoty.

Přiznávám se, moc tomu nerozumím. Vím, že to pracuje tak intuitivně, jako můj zrak nebo sluch. Častokrát v noci sedím a přemýšlím, jak by naše životy vypadaly, kdybychom jako lidstvo vůbec neměli Hint.

22-2-1 Jednoznačný svět — 8 bodů

Představme si lidské životy v alternativním vesmíru, kde lidé neumí činit rozhodnutí, a tak tam nenajdeme žádné křižovatky. V tomto vesmíru se právě vydal turista na cestu ze svého domu. Před domem má ukazatel jen s jednou cestou, a tak se vydává právě touto cestou. Dojde k druhému rozcestníku, ale zde je opět jen jeden ukazatel, kudy jít dál. Cestovatel vždy poslouchá rozcestníky a nikdy se nevrací zpět.

Mohlo by se zdát, že turista bude do nekonečna potkávat nové rozcestníky, ale i jeho planeta je konečná, a tak se po konečném počtu kroků stane, že dorazil na křižovatku, na které už byl. Turista tedy odteď bude pokračovat po smyčce (neboť zpět jít nesmí).

Vašim úkolem je napsat algoritmus, který bude postupně procházet rozcestníky (od prvního dále), ale dá si pozor na opakování, někdy skončí (pozor na zacyklení) a vypíše délku periody smyčky (jsme-li už na smyčce, tak kolik rozcestníků musíme navštívit, abychom se dostali na stejné místo, na kterém teď jsme).

Vstup si můžeme představit jako nekonečnou posloupnost přirozených čísel, která představují identifikační čísla jednotlivých křižovatek v pořadí, jak je cestovatel prochází, tedy například

1 2 5 8 12 35 123 42 8 12 35 123 ...

Aby to nebylo příliš jednoduché, tak dbejte na to, že si cestovatel (a tedy i váš algoritmus) v hlavě udrží pouze malé množství pomocné informace – snažte se tedy najít algoritmus, který spotřebuje co možná nejmenší množství paměti.

Do jednoho integeru se třeba vejde výsledná délka periody, délka začátku před periodou nebo jedno identifikační číslo jedné z křižovatek. Ne však už celá posloupnost (vždyť je nekonečná) ani její část – na uložení K hodnot vždy potřebujeme K integerů – ne více, ne méně.

Když jsem v 15 letech objevil Hint, musel jsem se s ním naučit stejně tak, jako se učíme chodit nebo mluvit. Trvalo to dlouho a raději jsem to dělal potají, aby se mi spolužáci nesmáli. Také byste se smáli, kdyby se někdo až v 15 letech učil chodit! Když jsem byl doma sám, stavěl jsem si různé labyrinty a bludiště a učil se, jak takové bludiště projít na první pokus jen s použitím Hintu.

22-2-2 Zkouška — 15 bodů

Náš hrdina si vyskládal do každého pokoje svého domu mince. Dům má tvar matice $M \times N$ a mezi každými dvěma sousedními pokoji jsou dveře. Nyní se nachází na políčku $(1, 1)$ – levý horní roh – a dovolil si chodit jen dveřmi dolů (první souřadnice) a doprava (druhá souřadnice). Tedy, ne úplně všude. V domě je ještě K speciálních místností, ve kterých může podvádět a jít i nahoru nebo doleva.

Za pomocí Hintu se mu snadno podaří dojít do místnosti (M, N) a nasbírat co nejvíce peněz. Vymyslete algoritmus, který dojde do stejné místnosti a také se mu to podaří. Na výstupu postačí maximální suma peněz, která lze sebrat.

Bodování:

- max. 15 bodů: řešení rychlé při $1 \leq M, N \leq 1000$, $1 \leq K \leq M \times N$.
- max. 11 bodů: řešení rychlé při $1 \leq M, N \leq 50$, $1 \leq K \leq M \times N$.
- max 6 bodů: řešení rychlé při $1 \leq M, N \leq 1000$, $K = 0$.

Úloha má přesně definované bodování, není však praktická. Výše zmíněné konstanty berte jako nápovědu, jak budou řešení bodována v závislosti na časové složitosti.

Příklad:

Počty mincí v místnostech:

```
1 2 1 1
1 1 1 1
2 1 1 3
```

Místnosti, kde lze podvádět: $(2, 2)$ $(3, 3)$

Výstup:

14

Jak jsem se s Hintem sžíval, začal mi pomáhat i v každodenním životě. Nemusel jsem se učit moc na písemky, Hint mi radil, co v nich bude, a já se naučil jen to minimum, které bylo třeba na jedničku. To není tak překvapující – ve třídě bylo hodně lidí, kteří také dostávali snadno jedničky. Hint mi začal radit víc a víc – věděl jsem, co si mám dát k snídani, kudy mám jít do školy, co mám dělat po obědě. Někdy nebylo hned jasné, proč je tohle či tamto rozhodnutí lepší než jeho opak, ale věřil jsem Hintu stejně pevně, jako věřím svým očím nebo uším.

Vůbec nerozumím lidským starostem. Proč se těmi věcmi trápí? Bojí se, že jejich Hint jim neporadí to správné řešení konfliktů? Nebo jsou ještě více opožděni než já, a svůj Hint ještě neobjevili? Světské starosti mi připadaly jako malinkaté tečky na papíře, kterým se lze jen smát. Některé životy jsou tak podobné, že jsem je spojoval kružnicemi. Samozřejmě jen ty, které mi poradil Hint.

22-2-3 Kružnice — 10 bodů

Máte před sebou papír, na kterém jsou rozmístěny body. Váš algoritmus by vám měl nahradit Hint a určit, která z možných kružnic se má nakreslit. Nesmíte ale zvolit libovolnou – algoritmus musí najít tu kružnici, na jejíž hraně leží co nejvíce vyznačených bodů. Pokud je takových kružnic více, stačí vrátit libovolnou splňující.

Jestli jsem o Hintu pochyboval? Ano, měl jsem jednu chvíli, kdy jsem byl na vážkách, jestli mi Hint neradí špatně. Ale zkušenost mi radila, že to se mnou Hint myslí upřímně a že se není čeho bát.

Jeli jsme s mamkou v úterý nakoupit do supermarketu a blížili jsme se ke kolejím, když mi najednou Hint poradil (když jsem se ho ptal, co je pro mne aktuálně nejlepší), ať za žádnou cenu nepřekračuji ty koleje. Široko daleko žádný vlak, ale Hint je Hint. Zakřičel jsem na matku, ať zastaví auto, a jak jsme zastavili, ihned jsem vystoupil ven.

Sedl jsem si na obrubník a na skoro zoufalý křik matky, proč jsem to probůh udělal, jsem řekl, že mi to Hint doporučil. A pak už jsem radši byl zticha, protože situace začala být opravdu divoká. Lidé se kolem mne střídali, auta houkala a troubila, vlaky se lopotily přes přejezd. Jen billboard za kolejemi se na mne klidně usmíval.

22-2-4 Billboard — 10 bodů

Byl jsem v dosti stresové situaci, a tak jsem počítal, kolikrát přes přejezd uvidím pána na billboardu. Na přejezdu byly dvoje koleje a po obou zrovna projížděl vlak (vlaky jely proti sobě) s vysokými a nízkými vagóny. Když zrovna byly dva nízké vagóny vedle sebe, bylo skrz přejezd vidět na billboard, ale když na přejezdu byl vysoký vagón, nic jsem neviděl.

Vlaky byly opravdu dlouhé (dá se říci, že nekonečné), ale oba byly řazeny tak, že se tam opakoval stále dokola vzorek vysokých a nízkých vagónů.

Pokaždé, když byly dva vagóny proti sobě, podíval jsem se, jestli vidím na billboard, a počítal jsem poměr mezi situacemi, kdy jsem jej viděl, a kdy ne. Měřit jsem začal ve chvíli, kdy se proti sobě setkaly první dva vagóny.

Vaším úkolem je napsat program, který ověří mé výpočty.

Příklad:

Jsou-li vlaky řazeny takto: první `VVNVV` a druhý `NNV`, pak nejprve nevidím billboard, protože na obou kolejích je vysoký vagón:

```
    _
<-- VVNVV
    VNN -->
    ^
```

Ve druhém kroku nevidím billboard, na druhé koleji je vysoký vagón:

```
    _
<-- VNVVV
    NVN -->
    ^
```

Ve třetím kroku ho konečně vidím (dva nízké vagóny výhledu nevadí):

```
    _
<-- NVVVV
    NNV -->
    ^
```

A tak dále. Celkový poměr je v tomto případě 2/15.

A tak jsem se přestěhoval do Bohnic. Chybí mi tu trochu kamarádi a rodina, ale není to zas tak zlé. Místní jsou uzavření a tiší, starostí je málo. Občas se mne sice snaží přesvědčit, že Hint nemám a není skutečný, ale to se jim nemůže podařit – kdybych nemohl věřit svým vlastním smyslům, tak komu?

V očích sester a lékařů vidím zklamání, že se jim nedaří mne vyléčit. Jinak jsou ale se mnou spokojeni, jsem hodný a tichý. Občas mi dovolí vyjít ven s ostatními pacienty, kde si pak společně hrajeme a dovádíme.

22-2-5 Pružinky — 10 bodů

Znáte hru na pružinky? Oblíbená hra nejen v Bohnicích, ale i mezi matfyzáky (kdo by se divil). Hráči/blázni se postaví do kruhu, první začne a řekne „p“. Následuje hráč po jeho levici a říká „r“. Takto se hraje podle hodinových ručiček, až některý hráč musí říci poslední písmenko „y“. Tento hráč vypadává, odstupuje (odskakuje) z kruhu, kruh se zmenší a hráč po jeho levici opět říká „p“. Pokračuje se tak dlouho, dokud někdo ve hře zůstává. Poslední hráč ve hře vítězí.

Na vstupu dostanete slovo, které se bude vyslovovat místo slova „pružinky“. Hráče číslujeme od 1 (začínající) podle hodinových ručiček až ke K-tému. Vymyslete program, který určí číslo hráče, který zůstane ve hře jako poslední.

Jak jsem už říkal, je to tu úžasné. Líbí se mi, že mám spoustu času na zkoušení možností, které mi Hint nabízí. Hint se dá trénovat podobně jako ruce nebo nohy. Myslím si, že bych ho mohl začít využívat pro dobro ostatních.

Ale začnu pěkně pomalu – odpovědí na dopis. Před týdnem mi přišlo psaní, kde se jakési Bratrstvo ptá, jestli nemohu použít Hint a odpovědět na jejich dvě otázky. První

mi připadala naprosto nesrozumitelná. Asi jsou příliš zapálení do jejich okultních akcí, až zapomněli psát česky.

22-2-6 Otázka — 10 bodů

Mocný věštče, poraď nám s naším problémem! Nevyřešíme-li jej, bude to mít nedozírné následky.

Po velkém putování a obětování několika Bratrů jsme se dopátrali k magickému obdélníku. V obvyklém stavu má následující podobu:

```
1 2 3 4
8 7 6 5
```

Ve starých svitcích stojí, že správné čtení je podle hodinových ručiček od levého rohu, čili této konfiguraci zapíšeme do našich análů jako 1 2 3 4 5 6 7 8.

Aby magický obdélník začal působit tak, jak my žádáme, musíme ho přemístit do jiné konfigurace. K dispozici máme tři kouzelné operace:

- *Vyměnium*, zkráceně `V` – vymění první a druhý řádek.
- *Sloupcium*, zkráceně `S` – posune pravý sloupec úplně nalevo.
- *Rotátum*, zkráceně `R` – prostřední čtyři čísla se posunou ve směru hodinových ručiček.

Předvedeme Vám, co se stane s obdélníkem v obvyklém stavu, když aplikujeme jen jednu z operací:

```
V:          S:          R:
8 7 6 5     4 1 2 3     1 7 2 4
1 2 3 4     5 8 7 6     8 6 3 5
```

Prosím, velký mudrci, pomoz nám vymyslet program, který vypíše nejkratší posloupnost magických operací takovou, že změní obvyklou konfiguraci na tu, kterou mu zadáme!

Například, zadáme-li žádanou konfiguraci

```
2 6 8 4 5 7 3 1
```

tak by měl odpovědět číslem a posloupností, tedy:

```
7
SRVSRRS
```

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx (*http://codex2.ms.mff.cuni.cz/ksp*). Pokud jste zatím žádnou praktickou úlohu neřešili, přečtěte si nápovědu u nějaké starší úlohy, např. 21-1-2 Optimalizace kotlů (*http://ksp.mff.cuni.cz/tasks/21/tasks1.html#task2*).

Druhá otázka byla podstatně kratší. Ptají se, jestli se písmeno P rovná dvěma písmenům NP. Sice bych řekl, že se nemůže rovnat, vždyť jsou to jiná písmena, ale stejně mi to přišlo o dost jednodušší, než ta první otázka. Pro jistotu jsem použil Hint, to abych pochopil, na co se vlastně ptají. Musím říci, určitě je moje odpověď potěší!

22-2-7 Sto oslů umořilo nic — 12 bodů

Opět se posuneme o kousek blíže našemu cíli. Budeme chtít, aby náš program běžel na více počítačích zároveň (tedy, byl distribuovaný). Používáme k tomu jazyk zvaný Erlang, jehož silnou stránkou je právě paralelizmus a distribuovanost. V dnešním díle se podíváme na procesy, jak jich pustit sto jako nic a jak vyřešit komunikaci mezi nimi.

Mocné funkce

V minulém díle jsme si řekli, že Erlang je funkcionální jazyk. To se jednak projevuje tím, že nepotřebuje cykly, stačí mu rekurze, jednak stylem jeho zápisu. Ale k funkcionálním jazykům patří i další věci. Jednou z nich je myšlenka, že funkce jsou také data, tedy jdou předávat, ukládat a dokonce i vytvářet za běhu.

Představme si, že nějak získáme takto uloženou funkci. Jak ji použít? Inu, úplně stejně, jako libovolnou jinou, prostě ji zavoláme – jen bude začínat velkým písmenem, protože je uložená v některé proměnné. Ukázkou může být například funkce `map` (která se dá najít v knihovně `lists`). Ta vezme seznam a funkci, na každý prvek zavolá tuto funkci a vrátí seznam výsledků. Vypadá takto:

```
map(_, []) -> [];
map(Fun, [Hlava | Ocas]) ->
      [Fun(Hlava) | map(Fun, Ocas)].
```

Nyní, jak se taková funkce předá? Možnosti jsou dvě. První, kterou jsme si v tomto příkladě předvedli také, je, že už funkci máme uloženou v nějaké proměnné, resp. je výsledkem nějakého výrazu. Jak již bylo zmíněno, chová se jako data, takže s ní lze také tak zacházet.

Druhá možnost je někam uložit (předat) pojmenovanou funkci. To se dělá tak, že se vytvoří dvojice skládající se ze jména modulu, kde funkce bydlí, a jejího jména. Pozor, taková funkce musí být z tohoto modulu exportovaná. Předpokládejme, že máme funkci `zpracuj` v modulu `data`. Potom by šlo psát toto:

```
map({data, zpracuj}, vstup).
```

Ale úplně nejzajímavější vlastnost je, že můžeme funkce vyrábět za běhu, šité na míru aktuální potřebě. Jak se to dělá? Místo jména funkce se použije klíčové slovo `fun`. Malá ukázka:

```
Funkce = fun(X) -> X * 2 end.
```

Toto vytvoří funkci, která násobí svůj parametr dvěma, a uloží ji do proměnné `Funkce`. A jak je to s tím šitím na míru? V kódu funkce můžeme použít i proměnné, které nepřebírá přímo tato funkce, ale které máme v aktuálním kontextu k dispozici. Tedy třeba takto:

```
nasobitko(Cim) ->
  fun(X) -> X * Cim end.
```

Tato funkce vrací funkci, která vždy dostane jeden parametr a přenásobí ho číslem `Cim`. Pro tuto funkci je `Cim` konstanta, ale můžeme vytvořit libovolně mnoho různých funkcí pro různá `Cim`.

Procesy

Pokud chceme vytvořit distribuovaný systém (ať už kvůli výkonu nebo proto, aby při záplavách v jedné serverovně nespadl celý systém), prvním krokem je rozdělit program na nějaké části, které běží skoro nezávisle na sobě – ve výsledku bude každý server provádět nějakou činnost (nebo více činností) a s ostatními se jen domlouvat.

Pokud dvě věci mohou běžet nezávisle na sobě, pustíme je v různých procesech. Ke spuštění nového procesu se používá funkce `spawn` a přebírá tři parametry. První dva udávají modul a funkci, která se v tomto procesu má vykonat. Třetí je seznam parametrů, se kterými bude tato funkce spuštěna (musí mít tolik položek, kolik jich daná funkce přebírá). Obdobně jako u předávání funkce, je potřeba, aby tato funkce byla exportovaná. Malá ukázka:

```
-module(blekota).
-export([start/0, vypis/2]).
```

```
vypis(_, 0) -> done;
vypis(Co, Kolikrat) ->
  io:format("~w~n", [Co]),
  vypis(Co, Kolikrat - 1).

start() ->
  spawn(blekota, vypis, ["Ahoj", 3]),
  spawn(blekota, vypis, ["Ble", 8]).
```

Funkce `vypis` prostě nějaký řetězec vypíše tolikrát, kolikrát se jí řekne. Ale když ji pustíme dvakrát, v různých procesech (jak děláme ve funkci start), tak budou „blekotat“ přes sebe.

Trocha komunikace

Procesy, které sice běží, ale navzájem se nemohou nijak ovlivňovat, jsou celkem nezajímavé. Některé procedurální jazyky mají možnost vláken, která se podobá procesům v Erlangu. Tam komunikují pomocí sdílené paměti (nějakých proměnných, kam zapisují společně). Nic takového v Erlangu není – proměnná, kam by se dalo jen tak zapisovat, neexistuje (lze si klást filozofickou otázku, jestli to, co má Erlang, lze považovat za plnohodnotnou proměnnou). Místo toho máme k dispozici mechanismus zpráv.

Když chceme některému procesu poslat zprávu, musíme vědět, kterému. K tomu slouží jeho `PID` (z Process ID). Vrací nám ho `spawn`, který tento proces pustil. Druhou možností získání PID je funkce `self()`, která vrátí PID aktuálního procesu. Pro zaslání zprávy slouží operátor `!`, kde nalevo je PID, napravo zpráva. Zpráva je libovolný výraz. Například takto:

```
pustAPosli() ->
  Pid = spawn(modul, funkce, []),
  Pid ! {posel, "zpráva"}.
```

Posílat zprávy nestačí, je třeba je také přijímat. K tomu slouží výraz `receive`, který má podobnou syntaxi, jako `case`. Obsahuje vzory, do kterých se příchozí zpráva pokouší `napasovat`, pokud se povede, použije se odpovídající podvýraz. Pro ukázku syntaxe:

```
prijimac() ->
  receive
    % Přišlo vyhlášení války
    {valka, PidUtocnika} ->
      bojuj(PidUtocnika);
    % Přišla jen obyčejná zpráva
    {posel, Zprava} ->
      io:format("~w~n", [Zprava])
  end.
```

Nyní, jak vlastně zprávy cestují? Pokud nějaká přijde, zařadí se do fronty. Když se spustí `receive` (nebo již běží), podívá se na první zprávu ve frontě a pokusí se ji postupně použít v jednotlivých vzorech. První, který bude odpovídat se použije. Pokud nebude vyhovovat žádný, zpráva se nechá ve frontě a zkusí se druhá zpráva. Pokud se nepoužije ani ta, pokračuje se na třetí. A tak dále, dokud tam nebude nějaká, která použít půjde.

Toto má tu výhodu, že proces nemusí znát všechny typy zpráv, které dostává, v jednom místě. Například můžeme mít funkci, která se optá jiného procesu na názor a počká si na odpověď. Mezitím mohou chodit zprávy, které se týkají něčeho úplně jiného, ale ty počkají do chvíle, než se dojde ke správnému místu v programu. Nevýhodou je, že pokud nějaký typ zprávy neošetřujeme, ale dostáváme, bude nám postupně plnit paměť.

Posílání zpráv je důležitá součást jazyka, proto si uvedeme i malou ukázku. Mějme modul `zpracovani`, který obsahuje dvě funkce. První, `vyrob`, vytvoří nějaká data. Těchto dat máme dostatek (kdykoliv nějaká potřebujeme, vytvoří nám nová). Druhá, `spotrebuj`, vezme blok dat a zpracuje ho. Mohli bychom udělat cyklus (za pomoci rekurze), ve kterém by se jednoduše zavolaly obě. Ale my si ukážeme řešení, při kterém budou moct tyto dvě funkce běžet paralelně.

```
-module(spojeni).
-export([vypocet/0,vyrobce/0,spotrebitel/1]).
-import(zpracovani).

vyrobce() ->
  Data = zpracovani:vyrob(),
  receive
    {chciData, Pid} ->
      Pid ! {data, Data},
      vyrobce();
    konec -> ok
  end.

spotrebitel(Vyrobce) ->
  Vyrobce ! {chciData, self()},
  receive {data, Data} ->
    case zpracovani:spotrebuj(Data) of
      dalsi -> spotrebitel(Vyrobce);
      konec -> Vyrobce ! konec
    end;
  end.

vypocet() ->
  Vyrobce = spawn(spojeni, vyrobce, []),
  spawn(spojeni, spotrebitel, [Vyrobce]).
```

Co se zde děje? Funkce `vypocet` jen spustí dva procesy, jeden, který bude data vyrábět, a druhý, který je bude spotřebovávat.

Výrobce napřed vyrobí jednu část dat a poté počká, až si o ni někdo řekne. Požadavek obsahuje i „zpáteční adresu objednávky“, tedy ví, kam data poslat. Poté jde vyrobit nová data a opakuje. V případě, že místo objednávky dostane oznámení, že už toho bylo dost, skončí.

Spotřebitel získá adresu výrobce jako svůj parametr. Na začátku mu pošle „objednávku“ (a připojí k ní i své vlastní PID). Poté si počká na odpověď a příchozí data zpracuje. Podle výsledku zpracování se rozhodne, jestli bude chtít zpracovávat další data a nebo oznámí, že již ne.

Tímto způsobem se výrobce pustí do tvorby nových dat hned po odeslání, tedy v době, kdy je spotřebitel zpracovává. Práci jsme si však malinko zjednodušili – poslední data vyrobíme zbytečně, protože to, že je nikdo nechce se dozvíme až poté, co je máme hotová.

Kdyby nám přišlo, že jsou vzory, do kterých se pokoušíme zprávu dostat, příliš slabé, máme k dispozici ještě konstrukci `when`, která funguje obdobně, jako u parametrů funkce. Jednoduše napíšeme něco takového:

```
prijmi(IChyby) ->
  receive
    zprava -> zpracujZpravu();
    chyba when IChyby -> zpracujChybu()
  end.
```


Výsledková listina dvacátého druhého ročníku KSP po první sérii

		škola	*ročník*	*sérií*	*2211*	*2212*	*2213*	*2214*	*2215*	*2216*	*2217*	*série*	*celkem*
1.	Jiří Eichler	SlovanG OL	2	1	4	8	10	7	4	7	12	42,6	42,6
2.	Vojtěch Kolář	G Neratov	4	12	1	6	10	11	2	8	12	40,4	40,4
3.	Pavol Rohár	#3401	4	3	6	3		5	1	7	14	39,7	39,7
4.	Vlastimil Dort	GŠpitálsPH	4	16	10	5	0	7		10	12	36,7	36,7
5.	Filip Hlásek	GMikul23PL	3	11	6	7	10				12	35,0	35,0
6.	Petr Čermák	GEBeneseKL	4	6	6	6		7	8	8		34,8	34,8
7.	Petr Hudeček	GCoubTábor	2	1	5	4				4	5	30,4	30,4
8.	Martin Zikmund	G Turnov	2	5	2	8			1	6	7	29,1	29,1
9.	Vojtěch Hlávka	GŠlapanice	1	1		4		6	2	4		28,9	28,9
10.	Karel Tesař	SPŠE Plzeň	4	8	7	6	0	5		7		28,8	28,8
11.	Tomáš Novella	GAlejKošic	4	1	5	4		3	1	1	1	23,9	23,9
12.	Filip Štědronský	GMikul23PL	3	10						7	14	21,3	21,3
13.	Štěpán Šimsa	GJungmanLT	1	5	4	4					7	21,0	21,0
14.	Jiří Setnička	G25březnPH	3	7	3	5					7	18,9	18,9
15.	Ondřej Hübsch	#26386	0	1		3	0				5	14,4	14,4
16.	Radim Cajzl	GNoMěsNMor	3	21		3					12	13,0	13,0
17.	Filip Matzner	GJirsíkaČB	3	1	4	3						12,8	12,8
18.	Jakub Červenka	GŠpitálsPH	4	5							12	12,0	12,0
19.	Pavel Taufer	ArcibisGPH	4	8	1						9	11,3	11,3
20.	Tomáš Masák	GJirsíkaČB	3	1	1	3			1			10,7	10,7
21.	Martin Holec	G Slavičín	3	4							7	9,5	9,5
22.	Kateřina Lorenzová	G Česká ČB	3	6		8						8,5	8,5
23.	Ondřej Mička	G Jírov ČB	1	1		6						8,1	8,1
24.	Petr Zvoníček	G Slavičín	4	5		6						7,4	7,4
25.	Petr Pecha	SPŠSVsetín	3	7	5							6,3	6,3
26.	Karel Hulec	GJirsíkaČB	3	1	3							6,0	6,0
27.	Jakub Diatel	G Slavičín	2	1		2						4,4	4,4
28.	Karel Král	G Most	4	6	1							1,7	1,7

22-1-7 – Když telefony pekly jazyk – program

```
-module(vzorak).
-export([kazdydruhy/1, fibslow/1, fiblin/1, prvo/1]).

% === Každý druhý ===

kazdydruhy([]) -> [];   % Už nic
kazdydruhy([_]) -> [];  % Jen jeden, každý druhý z něj taky nic
kazdydruhy([_, Druhy | Ocas]) -> [Druhy | kazdydruhy(Ocas)].

% === Fibonacciho čísla ===

% Pomalá verze jen přepsaná ze zadání
fibslow(1) -> 0;
fibslow(2) -> 1;
fibslow(N) -> fibslow(N - 1) + fibslow(N - 2).

% Rychlejší verze, počítaná od nejmenších

% fiblin(Posledni, Pozice, Toto, Minule):
%       Posledni - kolikáté číslo chceme
%       Pozice - kolikáté máme spočítané nyní
%       Toto - číslo spočítané na aktuální pozici
%       Minule - na předcházející pozici
% Již máme spočítané správné číslo, jen ho vrátit
fiblin(Posledni, Pozice, Toto, _) when Posledni == Pozice -> Toto;
% Dostali jsme dvě minulá čísla, spočítá aktuální a posune se o pozici dál.
% Rekurzí pokračuje ve výpočtu na další pozici
fiblin(Posledni, Pozice, Toto, Minule) ->
        fiblin(Posledni, Pozice + 1, Toto + Minule, Toto).

% Obal, který spustí vlastní rekurzi s počátečními hodnotami
fiblin(1) -> 0;
fiblin(N) -> fiblin(N, 2, 1, 0).

% === Prvočísla ===

% Přidá na konec seznamu
pridej(Co, []) -> [Co];
pridej(Co, [Hlava | Ocas]) -> [Hlava | pridej(Co, Ocas)].

% Ověří, jestli je číslo dělitelné
% delitelne(Testovane, Prvocisla)
%       Testovane - které číslo zkoušíme
%       Prvocisla - seznam prvočísel menších než Testovane, seřazené dle velikosti
% Došly prvočísla
delitelne(_, []) -> false;
% Už jen větší než odmocnina - bývalo by tam muselo být i nějaké menší
delitelne(Testovane, [Nedelitel | _]) when Nedelitel * Nedelitel > Testovane -> false;
% Tohle dělí
delitelne(Testovane, [Delitel | _]) when Testovane rem Delitel == 0 -> true;
% Nedělí, zkusíme další
delitelne(Testovane, [_ | Ocas]) -> delitelne(Testovane, Ocas).

% Testuje nová prvočísla
% prvo(Zbyva, Testovane, Nalezena)
%       Zbývá - kolik jich ještě chybí
%       Testovane - ktere testujeme nyní
%       Nalezena - ta, která již máme
% Již jich máme dostatek
prvo(0, _, Prvo) -> Prvo;
% Ještě nějaká chybí. Je to aktuální testované dělitelné?
prvo(Zbyva, Testovane, Prvo) -> case delitelne(Testovane, Prvo) of
        % Je - zkusíme nějaké další
        true -> prvo(Zbyva, Testovane + 1, Prvo);
        % Není, připíšeme na konec prvočísel
        _ -> prvo(Zbyva - 1, Testovane + 1, pridej(Testovane, Prvo))
        end.

% Obal rekurze, pustí s prázdným seznamem prvočísel a prvním testovaným dvojkou
prvo(N) -> prvo(N, 2, []).
```


Toto bude chyby přijímat jen v případě, že `IChyby` bude `true`, jinak je bude nechávat ve frontě.

Objektové programování

V dnešní moderní době musí každý programovací jazyk podporovat objektové programování. Ale Erlang klíčové slovo `class` nemá (a ani jinou přímou podporu pro objekty v jazyce). Přesto nebudeme zoufat a objekty si postavíme vlastní.

Jak na to půjdeme? Máme procesy a umíme posílat zprávy. Tak si tedy z každého objektu uděláme proces. A když po něm budeme něco chtít, pošleme mu zprávu, ve které mu vysvětlíme svůj požadavek. Pokud potřebujeme výsledek, tak si na něj počkáme.

Ukázka

Jako bonbónek na konec tu máme ukázku, která používá v podstatě vše, co bylo v tomto díle probráno. Bude jí hra Nim (každý jistě zná, dva hráči střídavě odebírají jednu až tři sirky, kdo nemůže táhnout, prohrál). Budeme mít čtyři objekty – dva hráče, hromádku a rozhodčího (pravda, kdybychom hru implementovali přímočaře přes funkce v jednom procesu, vyšla by kratší, ale tady jde o to, ukázat komunikaci).

```
-module(nim).
-export([rozhodci/3, hromadka/1,
      prvni/1, druhy/1, hraj/0, hrac/1]).

hromadka(Kolik) ->
  receive
    {dotaz, Pid} -> Pid !
          {odpoved, Kolik}, hromadka(Kolik);
    {odeber, KolikOdebrat} ->
          hromadka(Kolik - KolikOdebrat)
  end.

rozhodci(Hromadka, Prvni, Druhy) ->
  Hromadka ! {dotaz, self()},
  receive
    {odpoved, 0} ->
      Prvni ! {konec, prohral},
      Druhy ! {konec, vyhral},
      {vitez, Druhy};
    {odpoved, Zbytek} ->
      Prvni ! {hraj, self(), Hromadka},
      receive {tah, Kolik} ->
        if (Kolik >= 1) and (Kolik =< 3)
          and (Kolik =< Zbytek) ->
            Hromadka ! {odeber, Kolik},
            rozhodci(Hromadka, Druhy, Prvni);
          true -> rozhodci(Hromadka,
                           Prvni, Druhy)
        end
      end
  end.

hrac(AI) ->
  receive
    {hraj, Rozhodci, Hromadka} ->
      Hromadka ! {dotaz, self()},
      receive {odpoved, Kolik} -> Rozhodci !
                    {tah, AI(Kolik)} end,
      hrac(AI);
    {konec, Vysledek} -> Vysledek
  end.

prvni(_) -> 1.

druhy(1) -> 1;
druhy(_) -> 2.

hraj() ->
  Hromadka = spawn(nim, hromadka, [5]),
  Prvni = spawn(nim, hrac, [{nim, prvni}]),
  Druhy = spawn(nim, hrac, [{nim, druhy}]),
  rozhodci(Hromadka, Prvni, Druhy).
```

Trochu vysvětlení k tomuto kódu. Hromádka si jen pamatuje, kolik sirek na ní zbývá. Pokud je požádána, tento svůj stav sdělí. Druhá věc, kterou umí, je nějaké množství odebrat. Poté se vždy funkce pustí znovu a čeká na další požadavek.

Poté zde máme hráče. Hráč dostane funkci na umělou inteligenci. Poté si počká, až mu někdo řekne, že je na tahu a dá mu k tomu Pid hromádky a rozhodčího. Hromádky se zeptá, kolik na ní zbývá, nechá umělou inteligenci rozhodnout, kolik odebrat a sdělí to rozhodčímu.

Nejsložitější je zde rozhodčí. Každé kolo napřed zkontroluje (dotazem na hromádku), zda ještě jsou nějaké sirky. Pokud ne, oznámí hráčům, jestli vyhráli nebo prohráli a skončí. Pokud ano, řekne prvnímu hráči, že je na tahu, a počká si na jeho rozhodnutí. Zkontroluje, že je v pořádku, a pokud ano, tah provede, hráče prohodí a začne nový tah. Pokud táhne proti pravidlům, tak tah ignoruje a zeptá se znovu.

Nakonec tu máme už jen funkci, která to celé spustí. Hromádku a hráče pustí jako nové procesy a rozhodčího nechá běžet ve svém.

Úložky

Nakonec je potřeba nabyté znalosti procvičit. A jak lépe, než že si každý napíšeme nějaké malé cvičení? (Jdu se stydět za to, jak zním jako učitel.)

Producent-konzument se skladištěm: Vzpomeňte si na ukázku, kde jeden proces produkoval nějaká data a druhý je spotřebovával. Budeme chtít vylepšit tuto ukázku o skladiště. Skladiště, když se pustí, dozví se svou velikost. Dokud nebude skladiště plné, tak bude moci producent produkovat nová data; dokud nebude prázdné, tak konzument bude spotřebovávat. Pokud chybí místo nebo data, tak producent, resp. konzument čeká, než to ten druhý uvede do lepšího stavu.

Napište tedy modul s třemi veřejnými funkcemi. První bude spouštět skladiště a bude přebírat velikost. Druhá bude spouštět producenta, dostane PID skladiště a funkci na vytváření dat. Třetí bude pro konzumenta a parametry bude mít obdobné.

Můžete předpokládat, že na jednom skladišti bude pracovat maximálně jeden producent a jeden konzument. Pokud vaše řešení bude fungovat i pro libovolné množství producentů a konzumentů, dostanete další dva bonusové body. [5 bodů]

Balené funkce: Rozhraní minulé úlohy umožňovalo předat pouze funkci bez parametrů. Představme si, že máme funkci `generuj`, která generuje potřebná data, ale potřebuje k tomu dostat parametr, řekněme třeba číslo 42. Jak ji dostaneme do tohoto rozhraní? [2 body]

Centrum práce: Představme si, že máme nějaké úložiště práce. Chceme funkci, která bude do tohoto úložiště přidávat novou práci. Dále bude několik „pracovních“ procesů. Ty budou provádět tyto úkoly. Když proces práci provede, řekne si o další (a případně počká, až nějaká práce přibude).

Rozhraní nechť si každý navrhne sám – jeho kvalita bude součástí hodnocení. [5 bodů]

Vzorová řešení první série

22-1-1 Alčina interpretace

Úlohou bylo najít cestu $P = (s = v_0, v_1, \ldots, v_n = c)$, na které se nejméně mění značky na hranách.

Pro řešení je třeba modifikace algoritmu pro hledání nejkratší cesty. Chtěli bychom, aby se algoritmus ve fázi i rozlil do všech vrcholů, které jsou od počátečního vrcholu vzdáleny přesně i změn. To nám samotný algoritmus procházení do šířky nezaručí. Pokud ale v každé fázi provedeme procházení do hloubky po hranách se stejnou značkou, projdeme graf přesně tak, jak chceme.

Uděláme menší trik a rozdělíme si každý vrchol na dva, podle toho, kterou hranou jsme do něj přišli. U každého vrcholu si budeme pamatovat značku hrany, která do něj vedla, a jeho předka. Jako datová struktura pro naše prohledávání nám bude sloužit obousměrný seznam. Pokud budeme přidávat na hlavu seznamu, tak bude sloužit jako zásobník, pokud přidáme vrchol na konec seznamu, tak budeme mít frontu. Díky tomu nejprve projdeme všechny hrany se stejnou značkou a až pak teprve ty s jinou. Na začátku přidáme do fronty oba počáteční vrcholy $+s$ i $-s$. Nyní odebíráme vrcholy z hlavy seznamu, dokud není prázdný. Pro každý vrchol v se podíváme na všechny jeho sousedy, pokud jsme v nich ještě nebyli, tak jim nastavíme v jako předka, označíme je jako prošlé a zařadíme do seznamu podle toho, jestli jsme se do nich dostali po hraně stejné nebo různé značky jako do v. Ve chvíli, kdy ze seznamu vytáhneme cílový vrchol, známe nejkratší cestu k němu.

Nakonec už zbývá jen zrekonstruovat cestu. Tady nám hodně pomůže, že jsme si vrcholy rozdělili, protože tak jsou jejich předci jednoznačně určeni. Stačí jen postupovat od cílového vrcholu rekurzí po předcích, dokud nedorazíme do počátečního.

Vrcholů máme kvůli rozdělení dvakrát více, ale to nám složitost nepokazí. Každý z nich přidáme do seznamu jen jednou. Časová složitost našeho prohledávání bude tedy $\mathcal{O}(n+m)$. Paměťová složitost bude lineární. Kromě zadaného grafu potřebujeme v paměti jen frontu na ukládání vrcholů.

Bylo by možné použít i jiné algoritmy, např. Dijkstrův algoritmus. Ten jsme ovšem v podstatě použili, jen nepotřebujeme prioritní frontu, protože si dovedeme vrcholy uspořádat sami.

David Marek

22-1-2 Sad

Fareyovy posloupnosti

”Však se podívej na druhou úlohu – Catalanova čísla a nic jiného!”

My, organizátoři, máme pro Alčino mladistvé nadšení slabost a většinou se snažíme jednat tak, abychom její ideály nepoškodili svým stařeckým pragmatismem. Ale vymýšlet kvůli tomu úplně nové a nikým jiným netušené úlohy? Tůdle. Druhá úloha byla na Fareyovy posloupnosti a nic jiného.

Naštěstí to nikdo z vás nerozpoznal: nejčastějším vaším obratem bylo nagenerovat si všechny rozumné zlomky, očistit je a uspořádat. To může a nemusí být dobrý nápad! Počet generovaných zlomků je evidentně v $\mathcal{O}(N^2)$, takže užijete-li takové metody, nebude váš algoritmus běžet v čase lepším – co když ale existuje sofistikovaný lineární algoritmus? Nebudete se mu moci rovnat.

Číselná teorie

Následuje složitý matematický důkaz, proč nelze Fareyovy posloupnosti zkonstruovat rychleji než v $\Omega(N^2)$. Pokud se na to necítíte, raději ho nečtěte, a pokračujte nadpisem Implementace.

Potřebujeme zjistit, jak dlouhá taková Fareyova posloupnost pro dané N vlastně je. Pokud by se stalo, že je její délka také v $\Omega(N^2)$, byla by výše uvedená obava lichá a vaše postupy stále mohly být optimální.

Slyšeli jste už někdy o Riemannově hypotéze, a jak skvělá matematikům přijde? Teď se k ní přiblížíme tak blízko, jak se středoškolskému informatikovi málokdy naskytne – užijeme Riemannovy funkce zeta!

Členy Fareyovy posloupnosti řádu N jsou takové členy množiny všech zlomků s přirozeným číslem menším nebo rovným N ve jmenovateli, které jsou v intervalu $\langle 0, 1\rangle$ a které nemají soudělný čitatel a jmenovatel. Takže abychom odhadli jejich počet, uděláme jedinou logickou věc: vezmeme do ruky dvě kostky s N stěnami a hodíme je na stůl. Větší číslo budeme interpretovat jako jmenovatel, menší jako čitatel; padnou-li kostky stejně, pokus nepočítáme a zkusíme to znovu. Tak nikdy nedostaneme 0, ani 1, leč význam této chyby se bude s rostoucím N umenšovat k nerozpoznatelnu. To je přesně to, co budeme dělat: zvětšovat N nade všechny meze. Přitom budeme počítat pravděpodobnost, s jakou nám padnou nesoudělná čísla.

Pokud se tato bude blížit k nějakému pevnému nenulovému číslu, bude počet členů Fareyovy posloupnosti v $\Omega(N^2)$. Proč? Neformálně: pravděpodobnost nějakého jevu je přeci počet příznivých situací ku počtu všech situací. Pokud spočítáme tento poměr v závislosti na N a pokud ani pro neomezeně rostoucí N nedojde k nule, ale naopak k nějaké konstantě větší než nula, pak je určitě počet příznivých situací nezanedbatelný v porovnání s počtem všech a můžeme poměr schovat do Ω – je to přece konstanta jako každá jiná.

Hm, dejme tomu – jak tu pravděpodobnost spočítáme? No, jaká je šance, že nám na obou kostkách nepadlo číslo dělitelné dvěma? 3/4, samozřejmě, protože pravděpodobnost, že padlo, je 1/4. Číslo dělitelné třemi? 8/9. Pěti? 24/25. A tak dál. Jaká je tedy pravděpodobnost, že nejenom že nám nepadla čísla dělitelná dvěma (3/4), ale ani třemi ($3/4 \cdot 8/9$), nadto ani pěti ($3/4 \cdot 8/9 \cdot 24/25$), ...? (Bereme jenom prvočísla, protože čísla složená už jsme odfiltrovali s výskytem nejmenšího jejich prvočinitele.) Kolik je číslo vyjádřené následujícím produktem, nekonečným součinem?

$$\prod_{p \in P} 1 - \frac{1}{p^2} = \left(\prod_{p \in P} \frac{1}{1 - p^{-2}}\right)^{-1}$$

Pozn.: P je množina všech prvočísel ($P = \{2, 3, 5, 7, \ldots\}$)

Tohle vypadá obtížně, viďte? Musíme na to oklikou.


```
   struct vrchol * v = koren;
   for (i = 0; i < s->delka; i++) {
      char znak = s->hodnota[i];
      //pokud následující vrchol
      //v trii zatím není, vytvoříme ho
      if (!v->dopredu[znak]) {
         v->dopredu[znak] =
                calloc(1, sizeof(struct vrchol));
         v->dopredu[znak]->znak = znak;
         v->dopredu[znak]->rodic = v;
         //pokud tady končilo nějaké slovo,
         //je podřetězcem přidávaného
         v->koncici = NULL;
         vrcholu++;
      }
      v = v->dopredu[znak];
   }

   v->koncici = s;
}

//všechny zatím neexistující dopředné hrany
//z kořene namíříme zpátky do kořene
for (i = 0; i < ZNAKU; i++)
   if (!koren->dopredu[i])
      koren->dopredu[i] = koren;
}

//zpracuje jeden znak, vrátí nový aktuální vrchol z trie
struct vrchol * krok(struct vrchol * v, char znak) {
   while (!v->dopredu[znak])
      v = v->zpet;
   return v->dopredu[znak];
}

//určí zpětné hrany v trii
void spocitej_zpetne(void) {
   struct vrchol ** fronta =
          malloc((vrcholu - 1) * sizeof(struct vrchol *));
   int zacatek = 0, konec = 0;
   int i;

   //přidá do fronty syny kořene
   for (i = 0; i < ZNAKU; i++)
      if (koren->dopredu[i] != koren)
         fronta[konec++] = koren->dopredu[i];

   while (zacatek < konec) {
      struct vrchol * aktualni = fronta[zacatek];
      if (aktualni->rodic == koren)
         aktualni->zpet = koren; // vrchol je syn kořene,
                                 // zpětná hrana musí vést do kořene
      else
         aktualni->zpet =
            krok(aktualni->rodic->zpet, aktualni->znak);
      //pokud tam končilo slovo, je podřetězcem aktuálního
      aktualni->zpet->koncici = NULL;
      //přidáme syny aktuálního vrcholu do fronty
      for (i = 0; i < ZNAKU; i++)
         if (aktualni->dopredu[i])
            fronta[konec++] = aktualni->dopredu[i];
      zacatek++;
   }
}

//zvýší hodnotu posledního výskytu slova
void zvys(struct slovo * slovo, int posledni) {
   if (slovo->nasledujici)
   {
      if (slovo->predchozi)
         slovo->predchozi->nasledujici = slovo->nasledujici;
      else
         slova_zacatek = slovo->nasledujici;
      slovo->nasledujici->predchozi = slovo->predchozi;

      slovo->predchozi = slova_konec;
      slovo->nasledujici = NULL;
      slova_konec->nasledujici = slovo;
      slova_konec = slovo;
   }

   slovo->posledni_vyskyt = posledni - slovo->delka + 1;

   if (slova_zacatek->posledni_vyskyt >= 0) {
          //všechna slova se už v textu vyskytla
      int delka = posledni -
             slova_zacatek->posledni_vyskyt + 1;
      if (delka < nej_delka) {
         nej_delka = delka;
         nej_zacatek = slova_zacatek->posledni_vyskyt;
      }
   }
}

int main(void) {
   int i;
   struct vrchol * v;
   struct slovo * prvni;
   scanf("%d", &slov);
   getchar();

   //načtení prvního slova
   prvni = malloc(sizeof(struct slovo));
   prvni->predchozi = NULL;
   prvni->nasledujici = NULL;
   slova_zacatek = prvni;
   slova_konec = prvni;
   gets(prvni->hodnota);
   prvni->delka = strlen(prvni->hodnota);
   prvni->posledni_vyskyt = -1;

   for (i = 1; i < slov; i++) {
      struct slovo * aktualni =
             malloc(sizeof(struct slovo));
      aktualni->predchozi = slova_konec;
      aktualni->nasledujici = NULL;
      slova_konec->nasledujici = aktualni;
      slova_konec = aktualni;
      gets(aktualni->hodnota);
      aktualni->delka = strlen(aktualni->hodnota);
      aktualni->posledni_vyskyt = -1;
   }

   postav_trii();
   spocitej_zpetne();

   gets(text);
   v = koren;
   for (i = 0; text[i]; i++) {
      v = krok(v, text[i]);
      if (v->koncici)
         zvys(v->koncici, i);
   }

   if (nej_zacatek == -1)
      puts("Některé slovo nebylo nalezeno.");
   else {
      char * reseni = malloc((nej_delka+1) * sizeof(char));
      strncpy(reseni, &text[nej_zacatek], nej_delka);
      reseni[nej_delka] = 0;
      puts(reseni);
   }

   return 0;
}
```

```
// na zaklade najnizsej urovne na ktoru sa vie vrchol dostat
// nastavi hranam cisla dvoj-suvislosti
void SetLabels(int x, int y, int label) {
        Color[y][x] = 1;
        for (int i = 0; i < 4; i++) {
                int px = x+dx[i], py = y+dy[i];
                if (Map[py][px] && Color[py][px] < 2) {
                        Comp[y][x][i] = label;
                        if (MinLevel[py][px] >= Level[y][x])
                                Comp[y][x][i] = ++Counter;
                        Comp[y+dy[i]][x+dx[i]][(i+2)&3]
                                = Comp[y][x][i];
                        if (Color[py][px] == 0)
                                SetLabels(px,py,
                                        Comp[y][x][i]);
                }
        }
        Color[y][x] = 2;
}

// prehladavanie do hlbky (zistime dosiahnutelnost vrcholov)
void DFS(int x, int y) {
        if (!Map[y][x] || Vis[y][x]) return;
        Vis[y][x] = true;
        for (int i = 0; i < 4; i++)
                DFS(x+dx[i],y+dy[i]);
}

// vlozi do fronty novy vrchol pri prehladavani do sirky
void Insert(int x, int y, int d, int dist) {
        if (Map[y][x] && Dist[y][x][d] == -1) {
                Dist[y][x][d] = dist;
                Vertex v = { x,y,d };
                Q.push(v);
        }
}

int main() {
        int X, Y, sx, sy, tx, ty, kx, ky;
        scanf("%d %d\n", &Y, &X);

        // inicializovat premenne
        Counter = 0;

        for (int y = 0; y <= Y+1; y++) {
                for (int x = 0; x <= X+1; x++) {
                        Map[y][x] = Vis[y][x] = false;
                        Level[y][x] = MinLevel[y][x] =
                                Color[y][x] = 0;
                        for (int i = 0; i < 4; i++) {
                                Comp[y][x][i] = 0;
                                Dist[y][x][i] = -1;
                        }
                }
        }

        // nacitat vstup
        for (int y = 1; y <= Y; y++) {
                scanf("%s", buffer);
                for (int x = 1; x <= X; x++) {
                        Map[y][x] = true;
                        switch(buffer[x-1]) {
                                case '#': Map[y][x] = false;
                                          break;
                                case 'H': kx = x, ky = y;
                                          break;
                                case 'O': sx = x, sy = y;
                                          break;
                                case 'P': tx = x, ty = y;
                                          break;
                        }
                }
                getchar();
        }

        // oznackovat hrany cislom prislusnej
        // dvoj-suvislej komponenty
        GetLevel(kx,ky,1);
        SetLabels(kx,ky,1);

        // zistit kam sa moze hrac dostat na zaciatku
        Map[sy][sx] = false;
        DFS(kx,ky);
        Map[sy][sx] = true;

        // oznacit stavy okolo objektu za pociatocne
        for (int i = 0; i < 4; i++)
                if (Vis[sy+dy[i]][sx+dx[i]])
                        Insert(sx,sy,i,0);

        bool Found = false;

        while (!Q.empty()) {
                // vybrat vrchol z fronty
                Vertex v = Q.front(); Q.pop();
                int dist = Dist[v.y][v.x][v.d];
                if (v.x == tx && v.y == ty) {
                        printf("%d\n", dist);
                        Found = true;
                        break;
                }

                // zistit na ktore pozicie okolo bedne sa da
                // dostat a skusit posunut bednu tym smerom
                for (int i = 0; i < 4; i++)
                        if (Comp[v.y][v.x][v.d] ==
                                        Comp[v.y][v.x][i])
                                Insert(v.x-dx[i],v.y-dy[i],
                                                i,dist+1);
        }

        if (!Found)
                printf("-1\n");

        return 0;
}
```

22-1-6 – Náhradní – program

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

#define ZNAKU 256
#define MAX_SLOVO 80
#define MAX_TEXT 500

struct slovo {
   char hodnota[MAX_SLOVO];
   int delka;
   int posledni_vyskyt;
   struct slovo * predchozi;
   struct slovo * nasledujici;
};

struct vrchol {
   char znak;
   struct vrchol * zpet;
   struct vrchol * dopredu[ZNAKU];
   struct vrchol * rodic;
   struct slovo * koncici;
};

int slov, vrcholu;
//spojovy seznam slov
struct slovo * slova_zacatek, * slova_konec;
struct vrchol * koren; //ukazatel na kořen trie
//informace o zatím nejlepším nalezeném výskytu
int nej_zacatek = -1, nej_delka = INT_MAX;
char text[MAX_TEXT]; //prohledávaný text

//ze slov v seznamu slova postaví trii s kořenem koren
void postav_trii(void) {
   int i;
   struct slovo * s;
   koren = calloc(1, sizeof(struct vrchol));
   koren->rodic = koren;
   koren->zpet = koren;
   vrcholu = 1;
   for (s = slova_zacatek; s; s = s->nasledujici) {
```


V roce 1644 se jistý italský matematik ptal po nekonečném součtu převrácených hodnot kvadrátů přirozených čísel, tedy po $\sum_{n\in\mathbb{N}} 1/n^2$. O sto let později se našel jiný matematik jménem Euler, který otázku zodpověděl: je to $\pi^2/6$. *Pí* je, jak vidno, číslo užitečné nejenom při poměřování kružnic, důkaz tohoto tvrzení jde však nad rámec našeho textu.

Riemannova funkce zeta je pro dané s definována takto: $\zeta(s) = \sum_{n\in\mathbb{N}} 1/n^s$, takže Eulerova odpověď spočítala hodnotu této funkce pro $s = 2$. Euler ale přišel na ještě zajímavější věc: $\sum_{n\in\mathbb{N}} 1/n^s = \prod_{p\in P} 1/1-p^{-s}$. Platí tedy konkrétně pro $s = 2$ toto:

$$\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\ldots = \prod_{p\in P}\frac{1}{1-p^{-2}} =$$

$$= \left(\left(1-\frac{1}{2^2}\right)\cdot\left(1-\frac{1}{3^2}\right)\cdot\left(1-\frac{1}{5^2}\right)\cdot\ldots\right)^{-1}$$

To na pravé straně je to, co jsme toužili spočítat, to na levé straně je suma, kterou nám spočetl hodný pan Euler. Kýžená pravděpodobost je proto $1/\zeta(2) = 6/\pi^2 = 0,6079\ldots$ – nenulová! Délka posloupnosti je $\Omega(N^2)$!

Vaše algoritmy byly vesměs dosti správně, jen jste to nejspíš neuměli dokázat. Však jsme za to žádné body nestrhávali. V jednodušších případech je však zhodnocení kvality vymyšlených postupů důležité: už jenom proto, abyste zbytečně neztráceli na sebevědomí. Jeden z vás označil své řešení tohoto problému v čase $\mathcal{O}(N \log N)$ za trapné :-) – kdyby si uměl odhadnout velikost výstupu, zjistil by, že nejen špatně spočítal složitost, ale že i dokázat něco spočítat v $\Omega(N^2)$ může být výhra.

Implementace

Takže: rozhodli jsme se nagenerovat všechny zlomky tvaru i/j, kde $j \in \{1, 2, \ldots, N\}$ a $i \in \{1, 2, j-1\}$. Je jich $N(N-1)/2$, takže $\mathcal{O}(N^2)$. Nyní potřebujeme ověřit, které jsou v základním tvaru, a všechny setřídit.

Máme-li rozhodnout, zda můžeme krátit či nekrátit, naskočí nám většinou vzpomínka na pojem největšího společného dělitele. Ti informatičtěji vzdělaní z nás navíc vědí, že se k jeho hledání hodí Euklidův algoritmus: pokud však podlehnou pokušení užít ho při řešení této úlohy, padli do další léčky. My si totiž zlomky nejdříve uspořádáme, čímž se nám ty o stejné hodnotě (tj. nějaké takové, které zkrátit jdou, a onen jeden, který ne) dostanou k sobě a my je pak budeme moci eliminovat lokálně, prostým porovnáním při posledním průchodu seřazeným polem. Narozdíl od Euklida k tomu nebudeme potřebovat žádný další čas.

Tím se nám úloha redukuje na jediný problém: seřadit co nejrychleji vygenerované zlomky. Použijeme k tomu přihrádkové třídění: vytvoříme si N^2 přihrádek a každý zlomek tvaru a/b vložíme do přihrádky $\lfloor \mathbb{N}^2 \cdot a/b \rfloor$. Bylo by pěkné, kdyby nám tak do jedné přihrádky nemohly spadnout dva zlomky o rozdílné hodnotě: a vskutku, nemůže se tak stát, protože rozdíl dvou zlomků se jmenovatelem menším nebo rovným N nemůže mít ve jmenovateli číslo větší než N^2, takže jakmile nám při zaplňování přihrádek jeden spadne do nějaké přihrádky i, dalšímu se to už nemůže stát – musí odskočit alespoň do $(i+1)$.

Většina z vás použila obecný třídicí algoritmus se složitostí $O(N \log N)$. Nenechávejte se tolik ukolébat tím, že v obecném případě nic lepšího nejde! Máte-li pod rukou hromadu dosti speciálních zlomků, pravděpodobně to zvládnete i lineárně.

Sláva! Máme algoritmus mající časovou i prostorovou složitost $\mathcal{O}(N^2)$, který generuje $\Omega(N^2)$ hodnot, takže je nejspíš docela optimální ...

I když! Potřebujeme tolik paměti? Řešení několika z vás si vystačila s paměťovou složitostí $\mathcal{O}(N)$, i když pak zpravidla vedla na čas v $O(N^2 \log N)$. V zásadě postupovali tak, že si nagenerovali nejmenšího kandidáta na dalšího člena posloupnosti od každé třídy zlomků se stejným jmenovatelem (tj. toho s jedničkou v čitateli), vhodili ho do řadící datové struktury (třeba haldy) a odebírali minimum. No a kdykoliv vyhodili prvek c/j, přihodili do struktury (haldy) další ve tvaru $(c+1)/j$.

Jaké si z toho vzít poučení do našeho přístupu, aniž bychom museli platit zhoršenou časovou složitostí? Rozdělíme si činnost našeho programu do N kroků, v i-tém z nich budeme zpracovávat zlomky větší než i/N a menší než $(i+1)/N$. Jak je najdeme? Budeme si v paměti držet pole, které nám pro každého jmenovatele řekne, s jakým čitatelem jsme ho naposled použili. Takže ho v i-tém kroku projdeme, zjistíme, jestli s o jednotku větším čitatelem spadá daný zlomek do našeho intervalu, pokud ano, vhodíme ho do přihrádkového třídění (které teď pracuje s podstatně menším univerzem, takže mu stačí malá paměť) a patřičně upravíme pole. No a pak už jenom z přihrádek vytiskneme setřízené zlomky.

Pěkné, ne? Otázkou je, jestli má cenu klást při návrhu algoritmů důraz na paměťovou složitost menší, než je velikost výstupu. Ale to samozřejmě má! Z teoretického hlediska je dobrá jakákoliv další optimalizace, která nás donutí přemýšlet, z praktického hlediska se nám už jen kvůli rozdílným rychlostem přístupu do různě velkých počítačových keší a pamětí hodí mít malou pracovní množinu.

Závěrem

Pokud vás úloha zaujala, vřele vám doporučuji otevřít si na Wikipedii heslo „Farey sequence“ a začíst se. Objevíte spoustu dalších hezkých vlastností Fareyových posloupností. Máte-li raději knihy než hesla, určitě neprohloupíte, přečtete-li si Conwayovu „The Book of Numbers“, kde se podává popularizační úvod do mnoha koutů všemožných číselných oborů.

Lukáš Lánský

Jednodušší algoritmus

Existuje i jiný algoritmus, který je daleko jednodušší a jeho časová složitost je na první pohled optimální. Ale něco za něco – zase budeme muset trochu přemýšlet nad tím, proč opravdu vypíše to, co má.

Představme si, že máme dva zlomky $a/b < c/d$. Za jejich *mediant* prohlásíme zlomek $(a+c)/(b+d)$ [to je takové „divné sčítání zlomků“]. Všimněte si, že mediant leží mezi a/b a c/d. To snadno ověříme: $a/b < (a+c)/(b+d)$ je totéž jako $a(b+d) < b(a+c)$, po roznásobení $ab+ad < ab+bc$, čili $ad < bc$. To ale není nic jiného než $a/b < c/d$. Podobně ukážeme $(a+c)/(b+d) < c/d$.

Nyní začneme vytvářet posloupnost zlomků takto: začneme s $0/1$ a $1/1$, pak mezi tyto dva zlomky vložíme jejich mediant $1/2$, pak zase mezi každé dva zlomky vložíme jejich mediant a tak dále. Kdekoliv by jmenovatel překročil dané N, přestaneme na příslušném místě vkládat. Z toho získáme snadný rekurzivní algoritmus, zapíšeme si ho třeba v Pythonu následovně:

```
def sb(a,b,c,d,N):
   (x,y) = (a+c,b+d)
   if y <= N:
      sb(a,b,x,y,N)
      print x,"/",y
      sb(x,y,c,d,N)

def farey(N):
   print "0 / 1"
   sb(0,1,1,1,N)
   print "1 / 1"
```

Zavedli jsme rekurzivní funkci `sb`, která vypisuje zlomky z intervalu $(a/b, c/d)$. Funguje jednoduše tak, že spočte mediant $M = (a+c)/(b+d)$, rekurzivně se zavolá na interval $(a/b, M)$, pak vypíše M a nakonec se zavolá na $(M, c/d)$.

Tento algoritmus určitě vypisuje nějaké zlomky s čitateli a jmenovateli menšími nebo rovnými N, činí tak v ostře rostoucím pořadí (takže žádný nevypíše dvakrát) a vypsáním každého stráví jednotkový čas, a proto je jeho časová složitost lineární v počtu vypsaných zlomků. Jinak ale vzbuzuje spíš otázky:

Jsou vypsané zlomky v základním tvaru? Dokážeme, že kdykoliv se při vkládaní mediantů v naší posloupnosti vyskytnou za sebou čísla a/b a c/d, platí $bc - ad = 1$. Z toho ihned vyplyne, že jak a/b, tak c/d jsou v základním tvaru – kdyby totiž existovalo nějaké $k > 1$ dělící jak a, tak b, muselo by tímto k být dělitelné i $bc - ad$, a tedy i jednička, což nejde. Podobně pro c/d. A jak naši rovnost dokázat? Indukcí: na počátku platí $(1 \cdot 1 - 0 \cdot 1 = 1)$, jakmile vložíme mediant $x/y = (a+c)/(b+d)$, vzniknou nám dvě nové dvojice sousedních zlomků. Naše rovnost jistě platí pro dvojici $(a/b, x/y)$: $bx - ay = b(a+c) - a(b+d) = ab + bc - ab - ad = bc - ad = 1$. Podobně pro dvojici $(x/y, c/d)$: $yc - xd = (b+d)c - (a+c)d = bc + cd - ad - cd = bc - ad = 1$.

Nezapomeneme na nějaký zlomek? Co by se muselo stát, abychom nějaký zlomek x/y s $x, y \leq N$ zapomněli vypsat? Nejdříve si všimněme, že to nemůže být způsobené tím, že jsme rekurzi zastavili příliš brzy – jakmile jednou jmenovatel nějakého zlomku překročí N, mají jmenovatele většího než N i všechny medianty s tímto zlomkem utvořené, takže zastavením rekurze přijdeme pouze o zlomky s příliš velkými jmenovateli. Mohli bychom tedy podmínku $y \leq N$ nahradit omezením hloubky rekurze libovolným obrovským číslem a algoritmus by stále dělal totéž, jen by vypisoval navíc nějaké zlomky, které nás nezajímají.

Udělejme to a sledujme, do kterých intervalů, se kterými algoritmus pracuje, padne pohřešovaný zlomek x/y. V počátečním intervalu $(0, 1)$ evidentně je. Kdykoliv interval podrozdělíme na podintervaly $(a/b, M)$ a $(M, c/d)$, musí určitě padnout do právě jednoho z nich, jinak by totiž byl mediantem, a tudíž vypsán. Jakkoliv hluboko se tedy zanoříme do rekurze, vždy najdeme interval, který obsahuje x/y. Ukážeme, že to není možné.

Pokud $x/y \in (a/b, c/d)$, musí platit nerovnosti $bx - ay > 0$ a $cy - dx > 0$, ale protože čísla a, b, c, d, x, y jsou celá, tak také platí nerovnosti $bx - ay \geq 1$ a $cy - dx \geq 1$. Proto výraz $\Phi = (c+d)(bx - ay) + (a+b)(dy - cx)$ je větší nebo roven $a + b + c + d$. Pokud Φ roznásobíme a vytkneme $(x + y)$, dostaneme $\Phi = (bc - ad)(x + y)$, jenže jak už víme z úvahy o základním tvaru, pro kterýkoliv interval, který potkáme, platí $bc - ad = 1$. Proto $\Phi = x + y$. Složením obou vztahů pro Φ získáme nerovnost $a+b+c+d \leq x+y$. Na každé úrovni rekurze se ovšem alespoň jedno z čísel a, b, c, d zvýší alespoň o jedničku, takže nejpozději po $x + y$ úrovních přestane x/y ležet v intervalu, což je spor.

Hotovo. Teď už tedy víme, že náš algoritmus vypíše každý zlomek právě jednou a učiní tak v lineárním čase s velikostí výstupu, což je, jak víme z výpočtů u předchozího řešení, $\Theta(N^2)$. Paměti jsme spotřebovali nejvýše lineární množství na zásobník od rekurze.

Poznámka: Funkce `sb` vám může připomínat in-orderový průchod nějakým binárním vyhledávacím stromem. To není náhoda – zlomky opravdu můžeme popsat tzv. Sternovým-Brocotovým stromem, což je nekonečný strom zlomků, jehož každý vrchol je mediantem dvou vrcholů z předchozí hladiny. V takovém stromu se pak každé racionální číslo z intervalu $(0, 1)$ vyskytuje právě jednou a iracionální čísla odpovídají nekonečným cestám z kořene dolů.

Ještě jednodušší řešení

... tentokrát dokonce s konstatní paměťovou složitostí, ale důkaz správnosti si už v zájmu zachování lesů odpustíme (také je založený na podobných úvahách o mediantech, zkuste si ho vymyslet). Vypadá takto:

```
def farey(N):
    a,b,c,d = 0,1,1,N
    print a, "/", b
    while c < N:
       k = int((b+N)/d)
       a, b, c, d = c, d, k*c-a, k*d-b
       print a, "/", b
```

Martin Mareš

22-1-3 Sazba

Škoda, že většina řešitelů se ještě z prázdninové hibernace neprobudila, neboť úloha nebyla příliš těžká. Nebo to bylo způsobeno složitým a místy kostrbatým zadáním? Někteří lidé se také chytili na malou zákeřnost v zadání – krása byla dobře definována i v momentě, když se na řádek vešlo několik slov bez mezer mezinimi. Příště už zlí nebudeme, slibujeme!

A nyní přistupme k řešení. Rozložení slov do bloku je velice pravidelné, díky tomu umíme v konstatním čase spočítat krásu jednoho řádku, máme-li načtené délky slov.

Pak už si stačilo jen rozmyslet, jak počítat minimální krásu (logicky správně spíše minimální ošklivost) pro $K + 1$ slov, pokud už známe všechna minima pro K slov a méně. Postupně budeme zkoušet, kolik se nám s aktuálním slovem vejde předcházejících slov na ten samý řádek. Pro každý takový počet slov P spočítáme krásu řádku a tu sečteme s minimální krásou pro $K - P + 1$ slov, kterou již známe. Najdeme-li minimum ze všech těchto součtů, získáme minimální krásu pro $K + 1$ slov.

Typičtější úlohu na dynamické programování aby člověk pohledal! Časová složitost pro N slov bude $\mathcal{O}(N^2)$ (pro K slov počítáme K minim, a $\sum_{K=1}^{N} K = (N \times (N+1))/2$) a paměťová $\mathcal{O}(N)$.

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

22-1-4 Bludiště

Pre začiatok si predstavíme bludisko ako graf G, kde voľné políčka reprezentujú vrcholy grafu a hrana medzi dvoma políčkami existuje práve vtedy ak sú tieto políčka susedné.


```
                        prihr[k][1] = j;
                }

        for (int j = 0; j < N; j++)
                if (prihr[j][0] != 0)
                        printf("%d/%d, ",
                                prihr[j][0],
                                prihr[j][1]);
    }

    printf("1/1");
    scanf("%d", &N);
}
```

22-1-3 – Sazba – program

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

int main(void)
{

    int delka, slov, suma, mezer, minimum;
    int i=0, j=0, k=0, l=0, m=0, n=0, s=0;
    int *krasy, *optima, *slova;
    char *slovo;
    FILE *inp, *outp;

    inp = fopen("text.in", "r");

    fscanf(inp, "%d %d\n", &slov, &delka);

    slovo = malloc((delka+1) * sizeof(char));
    slova = malloc(slov * sizeof(int));
    optima = malloc(slov * sizeof(int));
    krasy  = malloc(slov * sizeof(int));

    for (i = 0; i < slov; i++)
    {
        fscanf(inp, "%s", slovo);
        slova[i] = strlen(slovo);
    }

    fclose(inp);
    free(slovo);

    for (i = 0; i < slov; i++)
    {
        s += slova[i];
        suma = s;
        /* předpočítání součtů krásy */
        for (j = 0; j <= i; j++)
        {
            mezer = i-j;
            if (j > 0) suma -= slova[j-1];
            l = delka - suma;

            /* speciální případy */
            if ( l < 0) krasy[j] = -1; /* příliš dlouhé */
            else if (l == 0) krasy[j] = mezer;
                        /* na řádku nejsou mezery */
            else if (i == j) krasy[j] = (l-1) * (l-1);
                                /* jen jedno slovo */
            else {
                m = l / mezer;
                k = l % mezer;
                n = mezer - k;

                krasy[j] = (mezer*m*m) + (n*(-2*m + 1));

            }
        }
        /* nalezení minima */
        if (krasy[0] >= 0)
        {
            optima[i] = krasy[0];
        } else {
            minimum = -1;
            for (j = 0; j < i; j++)
            {
                if (krasy[j+1] >= 0)
                {
                    k = optima[j] + krasy[j+1];
                    if ((k < minimum) || minimum == -1)
                            minimum = k;
                }
            }
            optima[i] = minimum;
        }
    }

    outp = fopen("ples.out", "w");
    fprintf(outp, "%d\n", optima[slov-1]);
    fclose(outp);

    return 0;
}
```

22-1-4 – Bludiště – program

```
#include <cstdio>
#include <queue>
#include <algorithm>
using std::queue;
using std::min;

#define MAX 1234

const int dx[4] = { -1, 0, +1, 0 };
const int dy[4] = { 0, -1, 0, +1 };

typedef struct {
        int x, y, d;
} Vertex;

char buffer[MAX];

// Mapa bludiska
bool Map[MAX][MAX];
// Navstivene vrcholy v prehladavani
bool Vis[MAX][MAX];

// najkratsia vzdialenost do vrchola
int Dist[MAX][MAX][4];

// uroven pri hladani dvoj-suvislych komponentov
int Level[MAX][MAX];
// najnizsia uroven na ktoru sa da z vrcholu dostat
// pri hladani dvoj-suvislych komponentov
int MinLevel[MAX][MAX];

// cislo komponenty dvoj-suvislosti do ktorej
// patri hrana z daneho vrchola iduca v smere d
int Comp[MAX][MAX][4];

// farba vrcholu pri prehladavani do hlbky
// 0 - nenavstiveny vrchol
// 1 - vrchol na zasobniku
// 2 - opusteny vrchol
int Color[MAX][MAX];

// pocitadlo komponentov dcoj-suvislosti
int Counter;

// fronta pri prehladavani do sirky
queue<Vertex> Q;

// spocita najnizsiu uroven na ktoru sa da z vrchola
// dostat pri hladani dvoj-suvislych komponentov
int GetLevel(int x, int y, int level) {
        if (!Map[y][x]) return level;
        if (Level[y][x]) return Level[y][x];
        MinLevel[y][x] = Level[y][x] = level;
        for (int i = 0; i < 4; i++) {
                int child = GetLevel(x+dx[i],y+dy[i],
                                Level[y][x]+1);
                MinLevel[y][x] = min(MinLevel[y][x],child);
        }
        return MinLevel[y][x];
}
```

Funkce `prvo` (verze s 3 parametry) plní funkci vnějšího cyklu – dokud nemá dostatek prvočísel, tak postupně zkouší jednotlivá čísla a když nejsou ničím dělitelná, tak je přidává na konec seznamu (aby zůstal setříděný vzestupně – což je výhodné, neboť mnoho čísel je dělitelných malými čísly a můžeme si dovolit optimalizaci s odmocninou).

Každé číslo testuje funkcí `delitelne` – ta zkouší, jestli zbytek po dělení jednotlivými prvočísly je nula. Skončí ve chvíli, kdy již nejsou žádná prvočísla, aktuální prvočíslo je větší než odmocnina a nebo jsme našli nějakého dělitele.

Všimněte si, že ve funkci `prvo` je použitá podmínka za pomocí `case` a ne `when` jako v ostatních případech. To proto, že `when` a `if` nedovolují volat funkce (kvůli optimalizacím). Ti, kteří si kód vyzkoušeli, na takovou věc jistě přišli.

> Při programování nějakých reálných úloh se samozřejmě používají různé knihovny. Například v této úloze bychom si mohli ušetřit práci a nepsat funkci `pridej`, neboť taková se vyskytuje v modulu `lists` a jmenuje se `append`.

22-1-1 – Alčina interpretace – program

```
#include <cstdio>
#include <cstdlib>
#include <list>

using namespace std;

enum Stav { PLUS = 0, MINUS = 1, UZAVREN };

#define INF 1<<20
#define MAXN 5000
#define MAXM 100000

int vrcholy[2*MAXN+1];
int pocet_s[2*MAXN+1];
int sousedi[MAXM];
bool prosly[2*MAXN+1];
int predek[2*MAXN+1];

void nacist_graf()
{
    int n;
    scanf("%d", &n);
    int k = 0;
    for (int i = 1; i <= n; i++) {
        int ps;
        scanf("%d", &ps);
        pocet_s[MAXN+i] = pocet_s[MAXN-i] = ps;
        vrcholy[MAXN+i] = vrcholy[MAXN-i] = k;
        for (int j = 0; j < ps; j++) {
            int v;
            scanf("%d", &v);
            sousedi[k++] = v;
        }
        prosly[MAXN+i] = prosly[MAXN-i] = false;
        predek[MAXN+i] = predek[MAXN-i] = 0;
    }
}

int sgn(int x) {
    return x >= 0 ? 1 : -1;
}

int prochazeni(int s, int c)
{
    list<int> l;
    /* Pridame do seznamu startovni vrchol */
    l.push_front(s); l.push_front(-s);
    prosly[MAXN+s] = prosly[MAXN-s] = true;
    while (!l.empty()) {
        int v = l.front(); l.pop_front();
        /* Pokud jsme z fronty vytahli cilovy vrchol,
         * tak koncime. */
        if (abs(v) == c) {
            return v;
        }
        /* Projdeme vsechny sousedy vrcholu v. */
        for (int i = 0; i < pocet_s[MAXN+v]; i++) {
            int soused = sousedi[vrcholy[MAXN+v]+i];
            /* Pokud jsme jej uz zpracovali tak pokracujeme
             * na dalsi. */
            if (prosly[MAXN+soused]) {
                continue;
            } else {
                /* Pokud jsme do souseda prisli po hrane se
                 * stejnou znackou, zaradime jej na hlavu
                 * seznamu. */
                if (sgn(soused) == sgn(v)) {
                    l.push_front(soused);
                /* Jinak jej zaradime na konec seznamu. */
                } else {
                    l.push_back(soused);
                }
                /* Vrchol jsme uz zpracovali. */
                prosly[MAXN+soused] = true;
                /* Nastavime mu vrchol v jako predka. */
                predek[MAXN+soused] = v;
            }
        }
    }
    printf("Cesta nenalezena.");
    return s;
}

void rekonstrukce(int v, int s) {
    /* Dokud nejsme v pocatku, pokracujeme rekurzivne. */
    if (abs(v) != s) { rekonstrukce(predek[MAXN+v], s); }
    else { printf("Prosle vrcholy: "); }
    /* Nakonec vrcholy vypiseme ve spravnem poradi. */
    printf("%d ", abs(v));
}

int main()
{
    nacist_graf();
    int s, c;
    scanf("%d %d", &s, &c);
    int v = prochazeni(s,c);
    rekonstrukce(v,s);
    printf("\n");
}
```

22-1-2 – Sad – program

```
#include <stdio.h>

#define MAX_N 1000

void main(void)
{
        int N;
        scanf("%d", &N);

        // jakeho citatele bude mit dalsi zlomek
        // se jmenovatelem i?
        int jm[MAX_N];
        for (int i = 2; i <= N; i++)
                jm[i] = 1;

        printf("0/1, ");

        int prihr[MAX_N][2];
        for (int i = 0; i < N; i++)
        { // nyni hledame zlomky nalezici (i/n, i+1/n]
                for (int j = 0; j < N; j++)
                        prihr[j][0] = 0;

                for (int j = N; j >= 2; j--)
                        if (jm[j]*N < (i+1)*j)
                        {
                                int k = N*N*jm[j]/j-N*i;
                                prihr[k][0] = jm[j]++;
```


Na vyriešenie úlohy si vytvoríme druhý graf G_2. V ktorom vrcholy budú dvojice: (Pozícia hráča, pozícia objektu). Takýto graf má $\mathcal{O}(M^2N^2)$ vrcholov, pričom hrany z vrcholu vedú do vrcholov, do ktorých sa dá dostať po jednom presune hráča, týmto hranám dáme dĺžku 0. Alebo ak hráč stojí vedľa objektu, tak do pozície, keď hráč zatlačí na objekt (ak je to možné), týmto hranám dáme dĺžku 1. Vidíme, že z každého vrcholu vedie maximálne 5 hrán, takže počet hrán je úmerný počtu vrcholov, teda tiež $\mathcal{O}(M^2N^2)$.

Odpoveď sa potom rovná dĺžke najkratšej cesty z počiatočnej pozície hráča do ľubovolnej pozície, kde sa pozícia objektu rovná pozícii koncového miesta. Ak takáto cesta neexistuje, potom sa objekt na danú pozíciu nedá presunúť.

Na nájdenie najkratšej cesty môžeme použiť Dijkstrov algoritmus, čím dostaneme časovú zložitosť $\mathcal{O}(M^2N^2\log(MN))$ alebo len dokonca $\mathcal{O}(M^2N^2)$, prečítajte si vzorové riešenie úlohy 22-1-1 a skúste sa zamyslieť ako na to.

Za takéto riešenie ste mohli získať 7 bodov.

Veľa riešiteľov si všimla, že nám stačí uvažovať len vrcholy grafu G_2, keď je pozícia hráča susedná s pozíciou objektu. Takýchto dvojíc je $4MN$. A označme podgraf grafu G_2 tvorený týmito vrcholmi G_3. Teda pozícia objektu a hráč môže stáť z jednej zo štyroch strán objektu. Na začiatku však hráč nemusí byť na pozícii susednej s objektom. To vyriešime tým, že na začiatku spustíme prehľadávanie (do šírky alebo hĺbky) na grafe G a zistíme, kam sa môže dostať hráč z počiatočnej pozície, pričom nás bude zaujímať, na ktoré susedné pozície s objektom sa dá dostať. Tieto dvojice budú naše počiatočné pozície v grafe G_3.

K rýchlemu riešeniu však musíme vyriešiť nasledujúci problém: Potrebujeme zistiť pre vrcholy G_3, ktoré sa líšia len na pozíciou hráča (teda strana objektu na ktorej hráč stojí) či medzi pozíciami hráča existuje nejaká cesta v grafe G, ktorá neobsahuje políčko na ktorom práve stojí objekt.

Môžeme zase potrebnú cestu nájsť prehľadávaním, avšak tým by sme zase dostali riešenie fungujúce v čase $\mathcal{O}(M^2N^2)$.

Správnym riešením sú vrcholovo dvoj-súvislé komponenty grafu G.

O dvojsúvislých komponentách a algoritme, ako ich nájsť si môžete prečítať v kuchárke KSP. Krátke zhrnutie: Vrcholovo-dvojsúvislým komponentom grafu G nazveme takú množinu hrán, že pre každú dvojicu vrcholov, ktoré sú incidentné s týmito hranami, existujú dve vrcholovo-disjunktné cesty. O vrchole v povieme, že patrí do nejakého dvoj-súvislého komponentu, ak aspoň jedna hrana má jeden koniec vo v. Všimnime si, že jeden vrchol môže patriť do viacerých vrcholovo-dvojsúvislých komponentov.

Pre nás je podstatné, že vrcholovo-dvojsúvislé komponenty vieme nájsť v lineárnom čase od veľkosti grafu.

Teda na začiatku nájdeme vrcholovo dvojsúvislé komponenty grafu G a pre každú hranu grafu G si uložíme číslo dvoj-súvislej komponenty do ktorej patrí. Následne pre vrcholy grafu G_3, ktoré sa líšia len pozíciou hráča, zistíme existenciu cesty medzi pozíciami hráča v G, ktorá neobsahuje pozíciu objektu na základe toho, či hrana spájajúce pozíciu objektu a pozíciu hráča v jednom vrchole je v rovnakej dvoj-súvislej komponente ako hrana spájajúca pozíciu objektu a pozíciu hráča v druhom vrchole. Ak takáto cesta existuje, potom medzi tieto vrcholy pridáme hranu dĺžky 0.

A opäť platí, že odpoveďou na náš problém je dĺžka najkratšej cesty z nejakej počiatočnej pozície do ľubovolnej pozície, kde sa pozícia objektu rovná pozícii koncového miesta.

Celková časová zložitosť sa skladá z počiatočného prehľadania grafu G, aby sme zistili, na ktoré susedné pozície objetku sa dá dostať, to stihneme v čase $\mathcal{O}(MN)$. Následne nájdeme dvoj-súvislé komponenty grafu G v čase $\mathcal{O}(MN)$ a nakoniec nájdeme najkratšiu cestu v grafe G_3, v ktorom je počet vrcholov $4MN$, hrán tiež $\mathcal{O}(MN)$, pričom hrany majú dĺžky len 1 alebo 0 v čase $\mathcal{O}(MN)$. Celková časová a pamäťová zložitosť je teda $\mathcal{O}(MN)$.

Peter Ondrúška

22-1-5 Šachovnice na pneumatice

Se ženami to není nikdy lehké, obzvlášť je-li jich n na pneumatice a navzájem se ohrožují. Ukážeme si však, že pro informatika s příslušným aparátem žádný problém nepředstavují.

Políčka šachovnice značme (x, y) pro $x, y \in \{0..n-1\}$, pozice jednotlivých dam budou (x_i, y_i). Chceme-li dámy rozestavět bezpečně, musí mít každá z nich vlastní jednu horizontální, jednu vertikální a dvě diagonální přímky. K popisu diagonál (a lecčehos jiného) se nám bude hodit termín *kongruence* – řekneme, že $a \equiv b \pmod n$ (tedy a je kongruentní s b modulo n), pokud čísla a a b dávají po dělení n stejný zbytek – tím elegantně popíšeme, že šachovnice má slepené konce. Hlavní diagonála h_i má pak na pneumatice rovnici $(x+y) \equiv i \pmod n$ (obsahuje tedy body $(i, 0), (i-1, 1), ...$), vedlejší diagonála v_i má rovnici $(x-y) \equiv i \pmod n$ (tedy $(i, 0), (i+1, 1), ...$).

Zkusme na to jít jednoduše, budeme dávat dámy postupně na řádky (y-souřadnice) s odskokem ve sloupečku o dva vůči předchozí (tak jako to bylo v ukázkovém příkladě pro $n = 5$); což lze popsat dvojicí rovnic $y_i \equiv i \pmod n$, $x_i \equiv 2y_i \pmod n$. Pro jaká n toto bude fungovat? Každá dáma má určitě vlastní y-souřadnici, a pokud je n liché, tak funkce $2y_i$ nejdříve skáče po sudých číslech (počínaje nulou), pak nabyde hodnoty $n+1 \equiv 1 \pmod n$ a skáče po lichých hodnotách. Dáma i tedy dostane jinou x-souřadnici než všechny předchozí. Hlavní diagonála příslušející dámě i má rovnici $(x_i + y_i) \equiv (3y_i) \pmod n$ – obdobně si všimneme, že není-li n dělitelné třemi, tak se nejdříve skáče po diagonálách čísla $3k$, pak se n přeskočí o 1 nebo o 2, a tedy se skáče po $3k+1$ nebo $3k+2$, a nakonec se n přeskočí o 2 nebo o 1 (podle toho co bylo v minulém kroku) a skáče se po $3k+2$ nebo $3k+1$ – a tedy opět má každá dáma unikátní hlavní diagonálu. Vedlejší diagonála bude mít situaci nejsnazší, její rovnice je totiž $-y_i \pmod n$, a tedy se skáče o jedničku po všech číslech. Tahle konstrukce tím pádem povolí všechna n, co nejsou dělitelná 2 a 3.

Zkusme na to jít obecněji a stavět následujícím předpisem: $y_i = i,\ x_i \equiv ky_i \pmod n$, kde k bude námi zvolený parametr (tedy dámy necháme odskakovat o k). Teď nám poslouží argument z teorie čísel, konkrétně Eukleidův rozšířený algoritmus. Ten umožňuje pro čísla n, k a libovolné $c \in \mathbb{N}$ najít takové koeficienty a a $b \in \mathbb{Z}$, že $a \cdot n + b \cdot k = c \cdot \mathrm{nsd}(n, k)$ (nsd je největší společný dělitel). Pokud celou rovnici vymodulíme n, dostaneme $b \cdot k \equiv c \cdot \mathrm{nsd}(n, k) \pmod n$. Je-li $\mathrm{nsd}(n, k) = 1$, pak umíme pro každé číslo c najít takové b, aby rovnice platila – jinak řečeno, zvolíme-li hodnoty $y_i \cdot k \equiv x_i \pmod n$, umíme ke každé přímce x najít její dámu (a má-li každá přímka svou dámu, tak má každá dáma svou přímku – dam a přímek je totiž stejně). Pokud

$\mathrm{nsd}(n,k) \neq 1$, pak k některým přímkám dámu nenajdeme (platí totiž, že $\mathrm{nsd}(n,k)|k \Rightarrow \mathrm{nsd}(n,k)|y_i k$, a tedy nic jiného než $x_i \equiv c \cdot \mathrm{nsd}(n,k)$ nedostaneme).

Stejný argument lze použít i pro hlavní a vedlejší diagonálu – je-li dáma na pozicích $(y_i k \pmod n, y_i)$, pak sedí na diagonálách s rovnicemi $(x_i + y_i) \equiv (y_i k + y_i) \equiv y_i(k+1) \pmod n$, popř. $(x_i - y_i) \equiv y_i(k-1) \pmod n$; a tedy musí být $\mathrm{nsd}(k-1,n) = \mathrm{nsd}(k+1,n) = 1$.

Nastanou-li tyto tři podmínky, pak každá dáma si ohrožuje jen své vlastní přímky, a rozestavění na šachovnici je tedy korektní. Zbývá se ptát, pro jaká n tohoto lze dosáhnout. Všimneme si, že mezi čísly $k-1, k, k+1$ je vždy alespoň jedno dělitelné třemi a alespoň jedno dělitelné dvěma – tedy pro žádné k nezvládneme řešit více šachovnic, než pro první popsaný postup s $k = 2$. Na druhou stranu to ale ukazuje poměrně zajímavý fakt, že šachovnice prvočíselných rozměrů dávají pro tento předpis „nejvíce volnosti“, k lze volit nejvíce způsoby (takovýchto srandovních vlastností mají prvočísla mnoho). Když už jsme u toho, ani rozestavění předpisem $x_i \equiv k y_i$ určitě není jediné možné, například pro $N = 19$ existuje celkem 820 496 rozestavění neohrožených dam (za tuto cennou informaci děkujeme Vojtovi Hlávkovi).

Předpokládali jsme ale konkrétní předpis pro pozice dam ($x_i \equiv k y_i \pmod n$), nešly by třeba pro zlá n rozestavit dámy jinak? Marná snaha, pro sudá n dokážeme neexistenci řešení následujícím způsobem: $\sum(y_i + x_i) \equiv \sum i$ (každá dáma má svou hlavní diagonálu h_i) $\equiv n \cdot (n-1)/2 \pmod n$ (součet čísel $0..n-1$); na druhou stranu $\sum y_i + \sum x_i \equiv$ $\equiv n \cdot (n-1)/2 + n \cdot (n-1)/2 \pmod n$ (každá dáma má svou vlastní horizontální i vertikální přímku) – tedy $n \cdot (n-1)/2 \equiv$ $\equiv 0 \pmod n$.

Zamysleme se: číslo x dá po dělení n zbytek nula tehdy a jen tehdy, když $x = c \cdot n$ pro nějaké celé c. Jinak řečeno, napíšeme-li $x = \prod p_i^{e_i}$ a $n = \prod p_i^{f_i}$ (tedy rozložíme x i n na součin mocnin prvočísel), tak musí platit $f_i \leq e_i$ pro každé i. Ale protože $\mathrm{nsd}(n, n-1) = 1$, tak $n-1$ neobsahuje žádné prvočíslo z rozkladu n a nehraje v otázce $n(n-1)/2 \equiv$ $\equiv 0 \pmod n$ roli. Pokud je n sudé, tak je zjevně u rozkladu výrazu $n(n-1)/2$ dvojka na exponent o jedna menší než u rozkladu n, a tedy zbytek po dělení nemůže být nula. Na druhou stranu, pokud je n liché, tak je $n-1$ sudé a snížení exponentu dvojky v rozkladu $n-1$ vůbec nevadí, výraz bude díky tomu že obsahuje n obsahovat všechna prvočísla z rozkladu n s dostatečným exponentem. Dostáváme tedy, že n musí být sudé.

Pro $3|n$ je situace podobná, uvážíme ale rovnice

$$H = \sum (x_i + y_i)^2,\ V = \sum (x_i - y_i)^2,$$

opět díky neohroženosti dam navštíví obě sumy kvadráty všech čísel $0..n-1$, a tedy

$$H \equiv V \equiv \sum i^2 \pmod n;$$

zároveň ale roznásobením druhých mocnin získáme

$$H \equiv \sum (x_i^2 + 2x_i y_2 + y_i^2) \equiv \sum x_i^2 + 2\sum x_i y_i^2 + \sum y_i^2 \equiv$$
$$\equiv \sum i^2 + 2\sum x_i y_i + \sum i^2 \pmod n,$$

podobně

$$V \equiv 2\sum i^2 - 2\sum x_i y_i \pmod n,$$

a tedy

$$2\sum i^2 \equiv H + V \equiv 4\sum i^2 \pmod n \Rightarrow 2\sum i^2 \equiv 0 \pmod n,$$

z čehož již dostaneme pomocí vzorce pro sumu třetích mocnin $n(2n+1)(n+1)/3 \equiv 0 \pmod n$ a tím nutnost $3 \not| n$ (obdobná argumentace, je-li $n = 3k$, tak nám $2n+1$ ani $n+1$ výsledek neovlivní, protože neobsahují v rozkladu trojku – která tím pádem bude vlevo chybět; na druhou stranu pro $n = 3k + \{1,2\}$ se trojka ztratí v některém z $\{2n+1, n+1\}$ a vítězíme). Pokud tedy šachovnici nevyřešíme pomocí $y_i = i, x_i \equiv 2y_i \pmod n$, tak už nijak.

vyčarovali pro Vás Martin Mareš & Vojta Tůma
na motivy dávného papyru George Pólyi

22-1-6 Náhradní

Hledáme nejkratší úsek textu, který obsahuje všechna slova ze zadaného slovníku. Slovo chápeme jako libovolnou posloupnost znaků, mezery nehrají žádnou zvláštní roli. Navíc, protože to zadání nezakazuje, budeme předpokládat, že se jednotlivé výskyty slov mohou překrývat. Algoritmus, který vyhledá slova v textu tak, jak potřebujeme, vymyslíme později, zatím budeme předpokládat, že nějaký takový máme.

Uvědomíme si, jak hledaná posloupnost znaků vypadá. Je jasné, že na jejím prvním znaku začíná nějaké z hledaných slov a na posledním nějaké končí. Kdyby to tak nebylo, mohli bychom ji zkrátit tak, že by pořád obsahovala všechna hledaná slova. To by ale znamenalo, že nalezená posloupnost nebyla nejkratší.

Když tedy pro každou pozici v prohledávaném textu, na které končí nějaké hledané slovo, najdeme nejkratší posloupnost, která tam končí a která obsahuje všechna hledaná slova, bude mezi nimi určitě i ta nejkratší, kterou chceme najít. Nejkratší vyhovující posloupnost s daným koncem najdeme tak, že se v textu podíváme na předchozí výskyty všech hledaných slov. Pokud jsme nějaké slovo v textu ještě nenašli, nemůže na dané pozici končit hledaná posloupnost. V opačném případě tady nějaká vyhovující posloupnost končí. Ze všech takových chceme vybrat tu nejkratší, což uděláme tak, že si pro každé hledané slovo určíme jeho poslední (ten, který je nejvíc vpravo) výskyt a z nich vybereme ten, který je nejvíc vlevo. Takto zjistíme délku nejkratší vyhovující posloupnosti s daným koncem. Průběžně si budeme pamatovat nejkratší zatím nalezenou posloupnost, takže na konci algoritmu budeme znát tu úplně nejkratší, kterou hledáme.

Stačí nám tedy pamatovat si poslední výskyt každého hledaného slova. Protože potřebujeme často zjišťovat pozici nejlevějšího z těchto výskytů, hodilo by se nám udržovat je setříděné podle jejich začátků. My ale procházíme výskyty podle jejich konců, a museli bychom tak každý výskyt znova zatřiďovat. Opakované zatřiďování je nutné proto, že začátky a konce výskytů hledaných slov mohou být v jiném pořadí. Uvědomíme si ale, že taková situace nastane jedině tehdy, pokud je jedno hledané slovo podřetězcem druhého, například máme ve slovníku zároveň slova `klíč` a `paklíček`. Jenže když nějaká posloupnost obsahuje delší slovo z takové dvojice, obsahuje určitě také kratší z nich, a tedy toto kratší slovo můžeme ze slovníku odstranit a stejně získáme stejný výsledek. (Jak přesně najít slova, která jsou podřetězcem jiného, si ukážeme za chvíli.) To ale znamená, že výskyty mají stejné pořadí, ať už je třídíme podle jejich začátků nebo konců, a můžeme tedy použít spojový seznam. Ten bude setříděný podle pozice posledního výskytu pro každé slovo a vždy, když narazíme na výskyt nějakého slova, přesuneme jemu odpovídající záznam na konec seznamu, což zvládneme v konstantním čase.

Konečně se dostáváme k samotnému vyhledávání slov. Tímto tématem se zabývá kuchařka 5. série 18. ročníku, která je dostupná na webu mezi spoustou jiných studijních textů na adrese *http://ksp.mff.cuni.cz/tasks/18/cook5.html*. My z ní použijeme algoritmus Aho-Corasick, který upravíme tak, jak je uvedeno výše, to znamená, že nebude vyhledávat slova, která jsou podřetězcem nějakého jiného slova ve slovníku.

Základní myšlenkou tohoto algoritmu je to, že vždy, když máme načtený nějaký kus textu, tak si budeme pamatovat jeho nejdelší konec, který je také začátkem nějakého z hledaných slov. Protože takovýchto začátků není moc, předem si pro každý z nich spočítáme, kam půjdeme dál, když na vstupu dostaneme nějaký znak. Vytvoříme si orientovaný graf (říká se mu trie), ve kterém vrcholy odpovídají všem počátečním podřetězcům hledaných slov, včetně prázdného. Navíc si budeme pamatovat, které podřetězce odpovídají hledaným slovům. Pokud mají dvě slova stejný nějaký začáteční podřetězec, bude těmto podřetězcům odpovídat stejný vrchol. Z vrcholu, kterému odpovídá nějaký řetězec, pak povedou hrany do všech vrcholů, kterým odpovídají řetězce o jedna delší a pro které je tento řetězec jejich začátkem.

Trii postavíme tak, že pro každé slovo začneme ve vrcholu, který odpovídá prázdnému řetězci. Dál postupně čteme znaky slova. Pokud aktuální vrchol už má hranu pro tento znak, tak po ní projdeme, jinak ji nasměrujeme na nově vytvořený vrchol a stejnětak po ní přejdeme. V obou případech pokračujeme dalším znakem. Nakonec si ještě u posledního vrcholu poznačíme, že tady končí toto slovo.

Po vytvořených hranách můžeme postupovat dopředu (budeme jim proto říkat dopředné), pokud to jde, ale my se musíme nějak vypořádat i se situací, že to nejde. To uděláme tak, že si u každého vrcholu budeme navíc pamatovat ještě tak zvanou zpětnou hranu. Ta povede do vrcholu, jehož řetězec je nejdelším možným koncem řetězce v tomto vrcholu. Při vyhledávání pak vždy zkusíme jít dopředu po odpovídající dopředné hraně a pokud to nejde, tak použijeme zpětnou hranu a celý postup opakujeme.

Jak ale zpětné hrany určíme? Budeme postupovat od nejkratších řetězců. Prázdný řetězec je speciální případ, u něho budou existovat dopředné hrany pro každý znak. Pokud neexistuje vrchol, kam by nějaká taková hrana mohla vést, tak povede zpět do tohoto vrcholu. Z vrcholů pro řetězce délky jedna musí vést zpětné hrany do vrcholu pro prázdný řetězec. Dále budeme u každého řetězce počítat s tím, že zpětné hrany už známe u všech kratších řetězců. Pak cíl zpětné hrany pro tento vrchol určíme stejně, jako bychom hledali následující stav pro poslední znak řetězce tohoto vrcholu, kdybychom postupovali z jeho rodiče (tj. jediného vrcholu, ze kterého sem vede dopředná hrana) a pokud bychom nemohli jít do tohoto vrcholu. Použijeme při tom určitě zpětnou hranu rodiče a možná i nějakých dalších vrcholů, ale o těch vím, že už existují.

Teď se ještě potřebujeme zbavit slov, která jsou podřetězcem nějakého z hledaných slov. To provedeme ve dvou fázích. Jednak už při stavění trie se může stát, že vytváříme syna vrcholu, ve kterém už nějaké slovo končí. Takové slovo pak ale musí být podřetězcem právě přidávaného, takže si u tohoto vrcholu poznačíme, že tam žádné slovo nekončí a tím se nám podařilo na něj zapomenout, jak jsme chtěli. Druhou fázi budeme provádět při budování zpětných hran. Pokud vytvoříme zpětnou hranu do vrcholu, ve kterém končí nějaké slovo, opět platí, že musí být podřetězcem nějakého slova, které prochází tímto vrcholem, a tak jej musíme zapomenout. Tím jsme docílili odstranění všech slov, u kterých jsme to požadovali.

Upravený algoritmus Aho-Corasick má časovou složitost $\mathcal{O}(S+T)$, kde S je součet délek hledaných slov a T je délka prohledávaného textu. Musíme sice zpracovat každý výskyt hledaného slova (kromě odstraněných), ale těch je nejvýše tolik, kolik je znaků v prohledávaném textu (pokud by na nějakém znaku končily dvě slova, tak musí jedno být podřetězcem druhého a taková slova jsme z vyhledávání odstraňovali). Paměťová složitost je $\mathcal{O}(S)$.

Celková časová složitost bude $\mathcal{O}(S+T)$, protože každý výskyt hledaného slova zvládneme zpracovat v konstantním čase. Paměťová složitost je $\mathcal{O}(N+S)$, kde N je počet hledaných slov.

Petr Onderka

22-1-7 Když telefony pekly jazyk

Každý druhý

Princip je jednoduchý. Budeme ze vstupního seznamu ukusovat od začátku po dvou prvcích. Z každého takového bloku (no, bločku) dáme do výstupu jen ten druhý a rekurzivně necháme zpracovat zbytek.

Ve chvíli, kdy již nebude k dispozici celý dvoubloček, tedy zbývá maximálně jeden prvek, není již co dávat na výstup, tedy skončíme.

Fibonacciho čísla

První verze (`fibslow`) je jen přepsáním definice ze zadání. V případě malých čísel přímo vrací výsledek, u větších spočítá dvě menší čísla a vrátí součet. Bohužel, toto je pomalé – má to složitost $\mathcal{O}(2^n)$ – viz například kuchařku 21. ročníku 5. série.

Inspirujeme se tedy kuchařkou a vyřešíme to tak, že budeme počítat postupně fibonacciho čísla od nejmenších postupně až k požadovanému. Nové se jednoduše spočítá sečtením dvou posledních, která si průběžně pamatujeme. Tímto snížíme časovou složitost na $\mathcal{O}(n)$.

Myšlenka zřejmá, jak ale takovou věc napíšeme, když není k dispozici žádný cyklus? Inu, všemocná rekurze nás zachrání. V každém kroku spočítáme jedno další číslo a rekurzí necháme spočítat ten zbytek. Až budeme na správném čísle, rekurzi zastavíme a vrátíme výsledek.

Nakonec jen zbývá rekurzivní funkci s velkým počtem parametrů obalit do něčeho, co má jen potřebný jeden, a je hotovo. Tato lineární verze se ve vzorovém řešení nazývá `fiblin`. Rychlostní rozdíl je možné vyzkoušet – již např. u 40 je rozdíl vidět pouhým okem velmi zřetelně.

> Existuje ještě vzoreček na spočítání n-tého Fibonacciho čísla, mohli bychom ho použít a zvládnout to v logaritmickém čase (mocní se v něm). Obdobný trik používá mocnění matic. Za takové řešení byl malý bodový bonus.

Prvočísla

Na hledání prvočísel existují dva jednoduché algoritmy – Eratosthenovo síto a zkoušet dělit všemi prvočísly do odmocniny. My použijeme druhý, protože nevíme, jak velké bychom potřebovali síto.