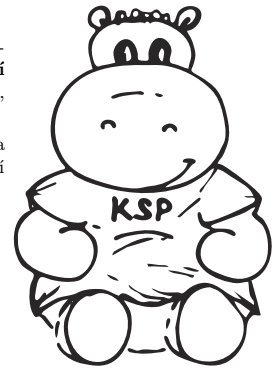
Milí řešitelé a řešitelky!



Úložky, úložky přicházejí, řešte je, přátelé! Přichází k vám třetí série 22. ročníku Korespondenčního semináře z programování. Podobně jako u minulé série k nám řešení **musí dorazit** do 8:00 SEČ dne 1. února 2010 (pondělí). Pokud tedy řešení odesíláte poštou, vhoďte jej do schránky do středy 27. ledna.

Elektronická řešení přijímáme na obvyklé stránce *https://ksp.mff.cuni.cz/submit/*, až na to, že jsme změnili protokol na HTTPS. Náš certifikát sice není podepsán žádnou komerční autoritou, nicméně pro jeho ověření zde uvádíme jeho SHA1 hash:

`10:3B:C1:E2:4F:9A:01:98:DA:DB:5D:A0:D5:5C:80:57:DE:AD:28:C3`

Papírová řešení nám pak můžete zasílat klasickou poštou na adresu

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

Třetí série dvacátého druhého ročníku KSP

Běží liška k Táboru, nese pytel zázvoru, ježek za ní utíká, že jí pytel rozpíchá ... Třetí příběh se nese v duchu nejen husitských válek a sepsal ho Jan „Moskyt“ Matějka. Poslední díl seriálu o Erlangu vám přináší Michal Vaner.

Mezi stromy prosvítalo slunce. Tomáš opatrně vykoukl z lesa. Nad řekou létali ptáci a na protějším břehu nikdo nebyl. Vrátil se ke koni, přebrodil řeku a pomalu začal stoupat k Táboru.

Po ušlapané cestě se jelo pohodlně. Daleko lépe, než když se před chvílí prodíral hustým lesem. Kdyby si ho ale všimli Zikmundovi zvědové, zabili by ho, nebo alespoň ... Ne, nemyslet na to. Tady už mohl být viděn, tady je vítán.

Vjel do otevřené brány a sesedl z koně. Na velkém prostranství pobíhaly děti, občas nějaká žena za domkem věšela prádlo. Klid a mír, jako by nevěděli, jakou zprávu přiváží. Strážný u brány na něj přátelsky pomrkával.

Hned se okolo něj seběhly děti a zvědavě se vyptávaly, kdo je, co veze, co jim dá ... Tomáš je však nevnímal a došel až k velké kádi plné stříbrných mincí uprostřed náměstí.

Vytáhl z hlubin svého pláště hrst mincí a podával je mladíkovi u kádě. „Zde je dar od bratří z Horního Blešna. Jen se mi mezi ně zatoulala jedna falešná.“

Falešná mince byla úspěšně oddělena, ostatní vhozeny do kádě a Tomáš vyrazil vstříc chlapíkovi, který vyšel z jednoho z domů okolo náměstí. „Buď zdráv, Prokope, přináším zprávy z Prahy.“ „Pokoj tobě, Tomáši. Pojď za mnou, ať zde nejsme rušeni.“

Oba vešli do domu, odkud Prokop před chvílí vyšel. Ženy dál věšely prádlo, nad Lužnicí létali ptáci. Obloha jako z Ladových obrázků, skoro bez mráčku, ale zdáli jako by zahřmělo. Nikdo tomu nevěnoval pozornost. Za lesem stoupal hustý černý dým, asi někdo pálil trávu. Děti na náměstí pokračovaly ve hře, kterou hrály před Tomášovým příjezdem.

22-3-1 Falešná mince — 10 bodů

Falešná mince se vyznačuje tím, že má odlišnou hmotnost od všech ostatních v hrsti. Jinak vypadá úplně stejně a je právě jedna v celé hrsti mincí. K dispozici máme rovnoramenné váhy. Určete, kolik vážení bude potřeba, abyste nalezli mezi N mincemi falešnou. Při jednom vážení může být na miskách libovolný počet mincí.

To by ovšem bylo příliš jednoduché. Váhy jsou totiž v celém Táboře jediné – v lékárně. Lékárník je navíc nedůvěřivý a požaduje, abyste mu předali sepsaná všechna vážení na papírku předem, on je provede, řekne vám, jak to dopadlo, a vy podle toho už musíte najít falešnou minci.

Příklad pro $N = 4$: Lékárníkovi zadáte zvážit mince takto:

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

Pokud jsou v obou případech v rovnováze, tak je falešná 4; pokud jsou v obou případech stejně nakloněné, tak je falešná 1; konečně pokud jsou jednou v rovnováze a jednou nakloněné, tak je falešná 2, resp. 3 podle toho, které jsou nakloněné.

22-3-2 Dětská hra — 8 bodů

Každé dítě si vybere jedno jiné dítě. Pak se křikne „Teď!“, děti se rozprchnou po náměstí a začíná hra. Každý chytá toho, koho si vybral (plácnutím po zádech). Chycený sdělí lovci, koho si vybral on, a lovec od této chvíle loví tuto novou oběť.

Zvědavá tetka Bětka před jednou hrou zjistila od všech dětí, koho chytají, a dlouze se zamýšlela nad tím, co se stane, když dítě zjistí, že má chytat samo sebe. To by nás zajímalo také a k tomu se hodí samozřejmě vědět, kolik dětí může do takové situace dospět.

Například pro skupinu 4 dětí, kde si první vybere druhého, druhý třetího, třetí čtvrtého a čtvrtý prvního, mohou do tohoto stavu dospět všechny čtyři děti.

Naopak pro jinou skupinu 4 dětí, kde první chytá druhého, druhý třetího, třetí prvního a čtvrtý také prvního, snadno zjistíme, že čtvrtý nikdy sebe sama chytat nebude.

Vymyslete algoritmus, který tento počet na základě informací tetky Bětky určí.

Jedno z dětí se právě začalo zuřivě bít do zad a křičet „Já, já, já!“, když Tomáš s Prokopem v družném hovoru vyšli ven z domu.

Přešli přes náměstí až ke bráně. Prokop řekl cosi strážnému, ten hned vstal a spěšně kamsi odešel. Vedle seděl zarostlý muž, který až do této chvíle hrál karty s vrátným. „Petře, čeká tě dlouhá cesta. Pozdravíš bratry v Rokycanech a sdělíš jim ... “

Kurýr Petr vstal a odešel do stáje. Strážný se vrátil k bráně s malým chlapcem, ten vyběhl ven. Pomalu začali přicházet různí lidé, vraceli se domů, do bezpečí za táborskou palisádu.

Petr se po chvíli vrátil i s osedlaným koněm, nabral do měchu vodu, přehodil přes sebe plášť, nasedl na koně a odjel, strážný za ním zavřel bránu. Tomáš a Prokop ho chvíli sledovali, jak přebrodil Lužnici a vydal se směrem na Orlík, pak se vrátili do Prokopova domku.

22-3-3 Kurýrní služba — 13 bodů

Mezi husitskými městy předávají zprávy kurýři. Každý kurýr přepravuje zprávy pouze ze svého domovského města do jednoho jiného. Opačným směrem je považován za nedůvěryhodného. Nevadí však, když je zpráva poslána postupně přes několik kurýrů (např. zprávu z Tábora do Chlumce odveze táborský kurýr do Prahy, kde ji předá jinému, který ji odveze do Chlumce).

Vymyslete algoritmus, jehož vstupem bude seznam všech kurýrů ve všech městech a který zjistí, jestli mezi každými dvěma městy lze přepravit zprávu alespoň jedním směrem. Stačí tedy, když existuje jednosměrná cesta.

Příklad: Pro města Tábor (`T`), Praha (`P`) a Rokycany (`R`) a kurýry `T`→`P`, `P`→`R` lze přepravit mezi každou dvojicí měst zprávu alespoň jedním směrem. Pro stejnou trojici měst, ale kurýry `T`→`P` a `R`→`P` nelze přepravit zprávu z Tábora do Rokycan ani naopak.

„Poplach! Hradečtí za řekou, poplach!“ ozvalo se najednou a dosud klidné náměstí jako by ožilo. Děti utekly domů, po náměstí pobíhali poloozbrojení muži. Mířili do zbrojnice. Atmosféra houstla.

Za dřevěnou palisádou se pomalu objevovali další strážní. Ozbrojeni okovanými cepy a krátkými meči byli téměř neporazitelní. Z dálky se pomalu blížilo dunění.

Prokop a Tomáš vyšli z domku a přešli náměstí. Prokop v plné zbroji, dlouhý meč a štít. Tomáš měl krátký meč, aby mu při jízdě na koni nepřekážel, ale někde ztratil rukavice. Stavili se tedy ve zbrojnici. Prokop zašel do malé temné místnosti a po chvíli se vrátil s náručí plnou rukavic. Hrály všemi barvami, až se zdálo, že Tomáš nenajde levou a pravou stejné barvy.

22-3-4 Rukavice — 10 bodů

V malé temné místnosti jsou dvě truhly a tma. V jedné z truhel jsou jen levé rukavice, ve druhé jen pravé. Bezpečně víme, kolik rukavic které barvy je ve které truhle. Prokop jednou náhodně vytáhne L levých rukavic a P pravých rukavic. Nalezněte algoritmus, který najde L a P taková, aby součet $L + P$ (celkový počet přinesených rukavic) byl co nejmenší, ale aby si Tomáš mohl vybrat pravou a levou rukavici stejné barvy.

Prokop jich ale přinesl dostatek, takže po chvíli Tomáš odcházel spokojen se dvěma hnědými rukavicemi.

Hradečtí vybíhali z lesa. Z palisády trčela kopí jak bodliny ježka, která jim znesnadňovala přístup až k ní. Táborští ale zjevně toužili po pořádné bitevní vřavě. Přelézali palisádu a bili se s hradeckými hlava nehlava.

Tomáš natáhl rukavice, vzal do ruky meč a nelítostně šermoval hned se dvěma hradeckými. Jednoho z nich odrazil, až se skutálel ze svahu dolů, druhý měl dosti tuhý kořínek, ale i ten nakonec padl. A hned se na něj vrhl další. Ještě že se skrčil. Taková příležitost k seku do nohou se jen tak nenaskytne!

Zvládl už asi deset hradeckých, když se k němu rozběhli najednou tři. Nevěděl, kam dřív skočit. Tu se otevřela brána a vyjely z ní dva vozy naložené shnilými mokrými kůly, které táborští vyměnili při poslední opravě palisády. Hradečtí nestíhali uhýbat a Tomáš měl konečně zase chvíli klid ...

⋆⋆⋆

Slunce se klonilo k západu, když se ozvalo trojí táhlé zatroubení. Hradečtí už byli dávno zpátky za řekou a utíkali rozprášeni přes pole pryč. Tomáš s Prokopem přes sebe přehodili pláště, dřevěné meče schovali pod ně, z Prokopova domu vytáhli batohy a vydali se spolu s většinou ostatních táborů na nejbližší železniční zastávku.

Přijíždějící vlak měl na sobě reklamu, kterou ještě nikdy neviděli. Na modrém pozadí byly nakresleny žluté puntíky propojené žlutými čarami.

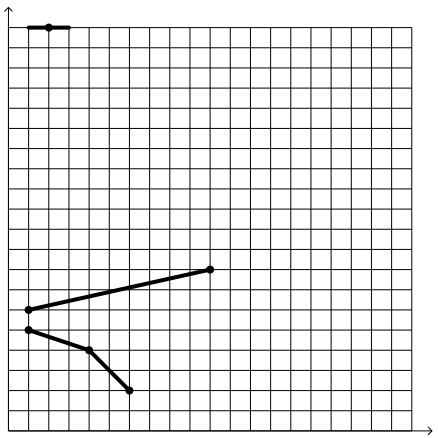
22-3-5 Reklama — 15 bodů

Ve čtvercové síti je nakresleno N bodů. Potřebujeme je pokrýt co nejméně plochými lomenými čarami.

Plochá lomená čára vypadá tak, že žádný z jejích úseků nesvírá s osou x větší úhel než 45° a lze ji nakreslit jedním tahem zleva doprava (tužka se pohybuje jen vpravo).

Na vstupu dostanete N bodů zadaných jejich celočíselnými souřadnicemi (x_i, y_i). Jako výstup vypište minimální počet potřebných plochých lomených čar.

Příklad:



Pro 6 bodů se souřadnicemi (1,6), (10,8), (1,5), (2,20), (4,4), (6,2) potřebujeme 3 ploché lomené čáry na jejich pokrytí.

Bodování:

- max. 15 bodů: řešení rychlé při $1 \leq N \leq 100000$
- max. 12 bodů: řešení rychlé při $1 \leq N \leq 1000$
- max. 10 bodů: řešení rychlé při $1 \leq N \leq 100$

Nastoupili do vlaku a vzájemně si vyměňovali zážitky. Kdo koho potkal, vypátral nebo nevypátral, kdo spadl do potoka nebo uskakoval před jedoucím vozem plným shnilých klád.

Ostatní odjeli auty. Dřevěný Tábor tady zůstane, za měsíc v něm bude dětský tábor. Všechno ostatní uklidili, přece po nich nezůstane nepořádek.

Slunce již bylo dávno pod horizontem. Praha svítila do dáli pouličním osvětlením, když Tomáš s Prokopem vystupovali na hlavním nádraží. Vešli do metra, v Holešovicích přestoupili a cestou od zastávky autobusu 112 na kolej přemýšleli, kolik času zase stráví ve výtahu, než dojedou až tam, kde bydlí.

Výsledková listina dvacátého druhého ročníku KSP po druhé sérii

		škola	*ročník*	*série*	*2221*	*2222*	*2223*	*2224*	*2225*	*2226*	*2227*	*série*	*celkem*
1.	Jiří Eichler	SlovanG OL	2	2	8	5	0	10	10	10	12	42,0	84,6
2.	Vojtěch Kolář	G Neratov	4	13	8	13	3	10	9	10	9,5	41,5	81,9
3.	Vlastimil Dort	GŠpitálsPH	4	17	7	13		10		10	12	43,4	80,1
4.	Pavol Rohár	GMRŠKošice	4	4	7	7	4	6	10	3	3	36,0	75,7
5.	Filip Hlásek	GMikul23PL	3	12	7			10	10	10		36,9	71,9
6.	Vojtěch Hlávka	GŠlapanice	1	2	2	2	5	8	5	10		34,8	63,7
7.	Karel Tesař	SPŠE Plzeň	4	9	7			6		10		23,9	52,7
8.	Štěpán Šimsa	GJungmanLT	1	6	4	5	3	6	7			28,5	49,5
9.	Petr Hudeček	GCoubTábor	2	2	4		2		5			18,1	48,5
10.	Ondřej Hübsch	ZŠJílov PH	0	2		6		6		10		28,6	43,0
11.	Miroslav Olšák	GBuďánkaPH	4	1	7	5	10	10	10	10	9,5	41,4	41,4
12.	Jiří Setnička	G25březnPH	3	8				6	7		6	22,0	40,9
13.	Petr Čermák	GEBenešeKL	4	6								0,0	34,8
14.	Martin Zikmund	G Turnov	2	5								0,0	29,1
15.	Petr Pecha	SPŠsVsetín	3	8		5		5	6			19,4	25,7
16.	Daniel Šafka	GKepleraPH	3	1	6		9	5	0			25,1	25,1
17.	Tomáš Novella	GAlejKošic	4	1								0,0	23,9
18.	Ondřej Mička	G Jírov ČB	1	2	4			6				14,6	22,7
19.	Filip Štědronský	GMikul23PL	3	10								0,0	21,3
20.	Karel Král	G Most	4	7			5	6	3			17,7	19,4
21.	Jakub Diatel	G Slavičín	2	2	3				5			13,0	17,4
22.	Martin Holec	G Slavičín	3	5							4,5	6,8	16,3
23.	Kateřina Lorenzová	G Česká ČB	3	7				6				7,2	15,7
24.	Pavel Taufer	ArcibisGPH	4	9				3				3,6	14,9
25.	Petr Zvoníček	G Slavičín	4	6					5			6,6	14,0
26.	Jonatan Matějka	G Jírov ČB	0	1	3				5			13,4	13,4
27.	Radim Cajzl	GNoMěsNMor	3	21								0,0	13,0
28. – 29.	Dana Marečková	GPatočkyPH	4	1	2			6				12,8	12,8
	Filip Matzner	GJirsíkaČB	3	1								0,0	12,8
30.	Jakub Červenka	GŠpitálsPH	4	5								0,0	12,0
31.	Tomáš Masák	GJirsíkaČB	3	1								0,0	10,7
32.	Tomáš Maleček	GEBenešeKL	4	1				7				9,1	9,1
33.	Martin Mach	G Jírov ČB	2	1				5				7,9	7,9
34.	Michal Bilanský	GLepařovJČ	4	6				5				6,6	6,6
35.	Karel Hulec	GJirsíkaČB	3	1								0,0	6,0

22-2-7 – Sto oslů umořilo nic – Centrum práce – program

```
-module(prace).
-export([start/0, prace/1, pracant/1, pracantInterni/1]).

pridej(Stare, Novy) -> prace(lists:append(Stare, [Novy])).

% Když nějakou práci máme
prace([Ukol|Zbyle]) -> receive
  % Tak dáme práci komukoliv, kdo si řekne
  {dejPraci, Pid} ->
    Pid ! {prace, Ukol},
    % A pořád dokola
    prace(Zbyle);
  {ukol, Novy} ->
    % Přídáme a zkusíme zpracovat
    pridej([Ukol|Zbyle], Novy)
end;
% Když žádná práce není, tak jen přijímáme
prace([]) -> receive {ukol, Novy} -> pridej([], Novy) end.

pracantInterni(Centrum) ->
  Centrum ! {dejPraci, self()},
  receive {prace, Ukol} ->
    Ukol(),
    pracantInterni(Centrum)
  end.

pracant(Centrum) -> spawn(prace, pracantInterni, [Centrum]).

start() ->
  Pid = spawn(prace, prace, [[]]),
  {fun(Ukol) -> Pid ! {ukol, Ukol} end, Pid}.
```


22-3-6 Kolejní výtahy — 10 bodů

V přízemí před výtahem stojí N kolejníků, kolej má K pater. Každý kolejník má své cílové patro – kde bydlí – a ochotu (inverzní veličinu k lenosti), která určuje, kolik pater je ochoten dojít poté, co vystoupí z výtahu. Výtah přijede, všichni kolejníci se do něj nastřádají.

Napište program, který dostane na vstupu seznam kolejníků s jejich cílovými patry a ochotami a který vypíše na výstup minimální počet pater, ve kterých je potřeba zastavit, aby byli všichni kolejníci uspokojeni. Jak počet pater, tak cílová patra a ochoty jsou nezáporná celá čísla.

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 stručný úvod: *http://ksp.mff.cuni.cz/about/codex.html*.

A tak skončila další dřevárna, oblíbená to víkendová zábava některých členů Bratrstva, jako Tomáše a Prokopa.

22-3-7 Pavouci internetu — 12 bodů

Pavouci dělají sítě. A také přežijí téměř cokoliv, protože mají každý orgán alespoň dvakrát a když o jeden přijdou, s jedním si chvíli vystačí a druhý po čase zase doroste.

V dnešním, posledním díle Erlangového seriálu se takovými pavouky necháme trochu inspirovat.

Pojmenované procesy

Když chceme, aby dva procesy komunikovaly, musí alespoň jeden z nich znát PID toho druhého. To je ale nepohodlné, protože by ty procesy nemohly vznikat nezávisle na sobě.

Erlang nás od tohoto problému zachrání tím, že nám dovoluje procesy pojmenovávat. Slouží k tomu funkce `register`, která přebírá jméno (které je reprezentováno atomem – tedy slovem začínajícím malým písmenem) a PID:

```
register(udrzbar, spawn(sprava, udrzuj,
                        [budova1, budova2]))
```

Poté můžeme používat tento atom v místě, kde bychom potřebovali PID procesu. Napíšeme prostě něco takového:

```
udrzbar ! {oprava,
    "Potřebuji opravit záchod, nesplachuje."}
```

Více počítačů

A nyní se dostáváme k tomu zajímavému. Máme více počítačů a chceme, aby různé kusy kódu běžely na různých počítačích. Než se však pustíme do vlastního programování, je třeba provést trochu nastavení.

Oprávnění

Při použití unixového systému je veškeré nastavení jednoduché. Vytvoříme soubor `.erlang.cookie` s oprávněním `0400` (čtení jen majitelem, nikdo jiný nesmí nic) a uložíme do něj jeden řádek obsahující nějaké heslo. Tento soubor poté umístíme na všechny počítače, kde náš program poběží (tím, že budou mít stejné heslo, budou počítače vědět, že si mají navzájem věřit, je to princip společného tajemství).

```
$ cat >.erlang.cookie
heslo
$ chmod 0400 .erlang.cookie
```

Při použití Windows je to malinko složitější. Domovský adresář je ten, který je nastavený v proměnné prostředí `HOME`, takže je potřeba zjistit, který to je, a případně tuto proměnnou na nějakou hodnotu nastavit. Pro ty, kteří nevědí, kde takovou věc najít, nachází se v Tento počítač → Vlastnosti → záložka Upřesnit → tlačítko Proměnné prostředí. Pokud nevíte, kde hledat Tento počítač, otevřete si Ovládací panely → Systém.

Komunikace

Chceme dosáhnout toho, že na různých počítačích běží různý kód, a uděláme to tak, že si na každém z nich pustíme interpretr Erlangu. Pokud se nedostává počítačů, tak je možné na jednom počítači pustit více interpretrů (pokud chceme testovat komunikaci 5 počítačů a máme jen jeden, tak na něm prostě pustíme 5 interpretrů).

Aby mezi sebou mohly interpretry komunikovat, musíme mít způsob, jak je adresovat. Adresa je podobná e-mailové a má tvar `jmeno@pocitac`. Část `pocitac` je jednoduchá – to je jméno počítače na lokální síti (ne, že by nešlo zařídit, aby spolu komunikovaly Erlangy na různých sítích, ale my si to ulehčíme). A část `jmeno` mu poskytneme při zapnutí. Předpokládejme, že tedy máme počítač zvaný `hroch` a chceme na něm pustit interpretr, který nazveme `testovaci` (tedy, adresa tohoto interpretru bude `testovaci@hroch`):

```
erl -sname testovaci
```

Posílání zpráv

PID obsahuje i identifikaci interpretru, ve kterém proces běží. To znamená, že pokud dostaneme PID jako parametr, máme ho v proměnné, je výsledkem funkce nebo podobně, nemusíme se vůbec starat o to, jestli proces běží u nás, nebo někde jinde. Zprávu mu odešleme úplně obvyklým způsobem a Erlang se o doručení postará.

Jediné, co je třeba prozkoumat, je posílání zprávy procesu registrovanému v jiném interpretru. Potom jako cíl zprávy uvedeme dvojici `{proces, interpretr}`. Tedy, kdyby náš údržbář byl registrovaný na vrátnici, hlásili bychom mu závady takto:

```
{udrzbar, vratnice@budova} !
          {oprava, "Potřebuji opravit záchod."}
```

To, kde je proces registrovaný, neříká vůbec nic o tom, kde vlastně běží. Registrování procesu je jen uložení PID do „globálního úložiště“.

Spouštění procesů

Pokud použijeme `spawn` tak, jak jsme jej až dosud používali, proces se spustí v aktuálním interpretru. Pokud bychom chtěli spustit proces v jiném interpretru, tak bychom přidali na *začátek* parametrů funkce `spawn` adresu interpretru, kde se má spustit. Samozřejmě, tento interpretr už musí běžet a musí mít k dispozici kód, který se má spouštět.

Vezměme si tedy příklad z minulého dílu, kde jsme měli producenta a konzumenta. Malinko si ho upravíme:

```
-module(spojeni).
-export([vypocet/0, registrovany_vyrobce/0,
                        vyrob/0, spotrebuj/1]).
-import(zpracovani).

vyrob() ->
  Data = zpracovani:vyrob(),
  receive
    {chciData, Pid} ->
      Pid ! data, Data,
```

```
      vyrob();
    konec -> ok
  end.

registrovany_vyrobce() ->
  register(vyrobce, self()),
  vyrob().

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

vypocet() ->
  spawn(vyrobce@hroch, spojeni,
                registrovany_vyrobce, []),
  spawn(spojeni, spotrebitel,
                [{vyrobce, vyrobce@hroch}]).
```

Čím se liší od minulé verze? Jednak, výrobce se registruje, abychom nemuseli chytat jeho PID (i když, zrovna v tomto případě z toho žádná výhoda neplyne, jen jsme si ukázali syntaxi). Ale co je hlavní, výrobce je *vzdáleně* puštěn v interpretru `vyrobce@hroch`, zatímco spotřebitel nám běží lokálně. Pokud by lokální interpretr neběžel na počítači `hroch`, ale někde jinde, tak by pracovaly oba počítače – jeden vyráběl, druhý spotřebovával. A o přenos dat mezi nimi by se staral Erlang bez naší pomoci.

Erlang je, i když se to nezdá, kompilovaný jazyk. Kompiluje se příkazem `c(jmenomodulu).` Toto vytvoří soubor obsahující bytecode pro interpretr (tou jsou ty podivné `.beam` soubory, které se při pokusech začnou povalovat po okolí). Tedy, pro použití modulu není `c` potřeba, pokud už `.beam` existuje, stačí se na něj jen odkázat. Proto, pokud chceme pouštět nějaký kus kódu na vzdáleném počítači, tak na něj stačí nakopírovat vzniklé `.beam` soubory.

Robustnost, spolehlivost

Co se stane v případě, že výrobce v našem případě bude dělit nulou a umře? Nebo v případě, že máme, podobně jako Cimrman, jako domácího mazlíčka slepici a ona dostane skvělý nápad nám klovnout do síťového kabelu?

Potřebujeme takové situace nějak ošetřit. Ne, že by Erlang dokázal zabránit přirozenému chování slepic, ale naučíme se na vzniklé situace reagovat.

Výsledek procesu

Každý proces jednou skončí, ale může skončit různými způsoby. Tyto způsoby se dělí do dvou skupin – normální konec a abnormální. Normální znamená, že je všechno v pořádku, abnormální obvykle znamená, že se něco pokazilo či nedopadlo, jak mělo.

Prvním způsobem je, když proces jednoduše doběhne na konec – všechny funkce skončí. To způsobí normální konec.

Druhý způsob je, když dojde k nějaké běhové chybě – dělíme nulou, počítač, kde proces běžel, odnese voda, zavoláme funkci, která neexistuje, a podobně. To samozřejmě způsobí abnormální konec.

Nebo může proces zavolat příkaz `exit`. Ten způsobí, že proces skončí (okamžitě, ne třeba až doběhne funkce). Přebírá jeden parametr – výsledek procesu. Pokud je jím atom `normal`, pak se jedná o normální konec, v opačném případě je to abnormální konec.

Známí

Procesy v Erlangu mohou navazovat jakési známosti. K tomu slouží příkaz `link(PID)`, který spojí aktuální proces s předaným v obousměrnou známost (aktuální zná PID a PID zná aktuální). Podobná funkce je `spawn_link`, jež funguje stejně jako `spawn`, ale navíc ještě seznámí starý proces s tím nově vzniklým.

Pokud některý náš známý proces skončí, pošle se nám o tom zpráva. Ve výchozím nastavení jsou normální konce ignorovány a abnormální způsobí, že skončíme také (abnormálně).

Již toto by stačilo na zařízení, aby, když umře některá část provázaného systému, bez které se nedá obejít, tak zbývající části „nehnily“, ale skončily také.

Řízení reakcí

Pokud by nám způsob „mor“ nevyhovoval, můžeme si změnit, co se stane, když nějaký známý skončí. To se udělá příkazem:

```
process_flag(trap_exit, true)
```

V takovém případě nám při skončení známého přijde obvyklá zpráva ve tvaru:

```
{'EXIT', PID_mrtvoly, Duvod}
```

`Duvod` bude to, co dostal `exit`, případně nějaké zdůvodnění, proč proces spadl. V případě, že vše bylo v pořádku, tak to bude `normal`.

Tuto zprávu si můžeme vybrat z fronty a známého třeba restartovat, nahlásit správci, prohlásit úlohu za neřešitelnou, či cokoliv jiného.

Netrpělivost

Normálně, pokud použijeme `receive`, tak čeká, dokud nějaká zpráva nepřijde. Můžeme však říct, že chceme čekat nejvýše nějakou dobu, a to tak, že jako poslední možnost dáme `after jakdlouhocekat`. Čas je uveden v milisekundách. Toto je tedy kód, který by čekal na autobus, ale nejvýše 5 minut:

```
cekej() -> receive
  {autobus, Pid} -> autobus ! {nastup, self()};
  after 300000 -> jdi_pesky()
end.
```

Ukázky

Náš spotřebitel potřebuje výrobce, bez něho nemůže fungovat. Takže bychom ho napsali asi takto:

```
bezpecny_spotrebitel(Vyrobce) ->
  link(Vyrobce),
  spotrebuj(Vyrobce).
```

Pokud umře výrobce, umře i spotřebitel. Dále, výrobce, pokud po něm nikdo dlouho nebude nic chtít, tak skončí (zřejmě něco nefunguje, protože si ho pustil a neposílá požadavky):


```
int main(void) {

   int perm[8], cil, permn[3][8], cur, kod, i, j;
   int hotovo[50000] = {0}, prev[50000] = {0};
   char path[50000], what[50000];
   FILE *vstup = fopen("obdelnik.in", "r"), *vystup;

   for (i = 0; i < 8; i++)
      fscanf(vstup, "%d", &(perm[i]));
   fclose(vstup);

   cil=zakoduj(perm);
   queue<int> q; // fronta

   hotovo[0] = 1;
   q.push(0); // 0 je v našem zápisu startovní obdélník

   while (!q.empty()) {
      cur=q.front();
      q.pop();

      if (cur == cil) {
         i = 0;
         while (cur) {
            path[++i] = what[cur];
            cur = prev[cur];
         }

         vystup = fopen("postup.out", "w");
         fprintf(vystup, "%d\n", i);
         while (i) {
            fprintf(vystup, "%c", path[i]);
            i--;
         }
         fclose(vystup);
         break;
      }

      dekoduj(cur, perm);

      // vygeneruj tři možné transformace aktuální permutace
      for (i = 0; i < 4; i++) {
         permn[0][i] = perm[7-i];
         permn[0][i+4] = perm[3-i];
         permn[1][i] = perm[(i+3)%4];
         permn[1][i+4] = perm[4+(i+1)%4];
      }
      permn[2][0] = perm[0];
      permn[2][1] = perm[6];
      permn[2][2] = perm[1];
      permn[2][3] = perm[3];
      permn[2][4] = perm[4];
      permn[2][5] = perm[2];
      permn[2][6] = perm[5];
      permn[2][7] = perm[7];

      for (i = 0; i < 3; i++) {
         int kod = zakoduj(permn[i]);
         if (!hotovo[kod]) {
            hotovo[kod] = 1;
            prev[kod] = cur;
            switch(i)
            {
            case 0:
                what[kod] = 'V';
                break;
            case 1:
                what[kod] = 'S';
                break;
            case 2:
                what[kod] = 'R';
                break;
            }
            q.push(kod);
         }
      }
   }

   return 0;
}
```

22-2-7 – Sto oslů umořilo nic – Producent a konzument se skladištěm – program

```
-module(produ).
-export([buffer/1, bufferInterni/3, producent/2, producentInterni/2, konzument/2, konzumentInterni/2]).
-import(lists).

bufferInterni(Volno, Mame, Sklad) ->
  receive
    % Vydá data, ale jen když nějaká jsou
    {vydej, Komu} when Mame > 0 ->
      [Prvni | Zbytek] = Sklad,
      Komu ! {data, Prvni},
      bufferInterni(Volno + 1, Mame - 1, Zbytek);
    % Přijme data, ale jen když se vejdou
    {pridej, Od, Data} when Volno > 0 ->
      Od ! prijato,
      bufferInterni(Volno - 1, Mame + 1, lists:append(Sklad, [Data]))
  end.

buffer(Velikost) -> spawn(produ, bufferInterni, [Velikost, 0, []]).

producentInterni(Buffer, Produkuj) ->
  % Uděláme data a odešleme
  Buffer ! {pridej, self(), Produkuj()},
  % A až nám je přijmou, uděláme další
  receive prijato -> producentInterni(Buffer, Produkuj) end.

producent(Buffer, Produkuj) -> spawn(produ, producentInterni, [Buffer, Produkuj]).

konzumentInterni(Buffer, Konzumuj) ->
  Buffer ! {vydej, self()},
  receive {data, Data} ->
    Konzumuj(Data),
    konzumentInterni(Buffer, Konzumuj)
  end.

konzument(Buffer, Konzumuj) -> spawn(produ, konzumentInterni, [Buffer, Konzumuj]).
```

```
kruznice opis(float A[], float B[], float C[])
{
    kruznice opsana;

    // Kde následující popis vzít? Dá se spočítat i najít v tabulkách.
    float p = 2*(A[0]*(B[1]-C[1])+B[0]*(C[1]-A[1])+C[0]*(A[1]-B[1]));
    opsana.Sx = ((A[1]*A[1]+A[0]*A[0])*(B[1]-C[1])
                +(B[1]*B[1]+B[0]*B[0])*(C[1]-A[1])
                +(C[1]*C[1]+C[0]*C[0])*(A[1]-B[1]))/p;
    opsana.Sy = ((A[1]*A[1]+A[0]*A[0])*(C[0]-B[0])
                +(B[1]*B[1]+B[0]*B[0])*(A[0]-C[0])
                +(C[1]*C[1]+C[0]*C[0])*(B[0]-A[0]))/p;

    opsana.r = sqrt((A[0]-opsana.Sx)*(A[0]-opsana.Sx)
                   +(A[1]-opsana.Sy)*(A[1]-opsana.Sy));

    return opsana;
}

int comp(const void *a, const void *b)
{
    const kdokam *C1 = (const kdokam *)a;
    const kdokam *C2 = (const kdokam *)b;

    return (C1->kam - C2->kam) > 0 ? 1 : -1;
}
```

22-2-4 – Billboard – program

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

#define MAXVLAK  100000

char vlakA[MAXVLAK], vlakB[MAXVLAK];

int A, B, d;

int nizkychA, nizkychB, N = 0;

int gcd(int x, int y)
{       // Eukleidův algoritmus
        int z;
        if (x == y) return x;
        while (y > 0)
        {
                z = x % y;
                x = y;
                y = z;
        }
        return x;
}

int main()
{
        // vstup
        A = fgets(vlakA, MAXVLAK, stdin);
        B = fgets(vlakB, MAXVLAK, stdin);
        vlakA[A] = 0;
        vlakB[B] = 0;
        // přerovnám si vlak B
        for (int i=1;2*i<B;i++)
        {
                char tmp = vlakB[i];
                vlakB[i] = vlakB[B-i];
                vlakB[B-i] = tmp;
        }
        d = gcd(A,B);
// d=1 pro nesoudělná A a B, takze nemusím dělat
// separátní větev pro tento případ.
        for (int i=0;i<d;i++)
        {
                nizkychA = 0;
                nizkychB = 0;
                for (int j=i;j<A;j+=d)
                        if (vlakA[j] == 'N')
                                nizkychA++;
                for (int j=i;j<B;j+=d)
                        if (vlakB[j] == 'N')
                                nizkychB++;
                N += nizkychA * nizkychB;
        }
// používám vzorec nsn(A,B) * nsd(A,B) = A * B
// zlomek nekrátím, zadání to nevyžadovalo
        printf("%d/%d", N, A*B/d);
        return 0;
}
```

22-2-6 – Otázka – program

```
#include <cstdio>
#include <queue>

using namespace std;

int fact[]={1,1,2,6,24,120,720,5040};

int zakoduj(int perm[8])
{
   int pouz[9] = {0};

   int kod = 0, i, j;
   for (i = 0; i < 8; i++) {
      for (j = 1; j < perm[i]; j++)
         if (!pouz[j]) {
            kod += fact[7-i];
         }
      pouz[perm[i]] = 1;
   }
   return kod;
}

void dekoduj(int kod, int perm[8])
{
   int pouz[9] = {0}, i, j, k;

   for (i = 0; i < 8; i++) {
      j = kod / fact[7-i];
      kod %= fact[7-i];
      for (k = 1; k <= 8; k++) {
         if (!pouz[k]) {
            if (!j)
               break;
            else
               j--;
         }
      }
      pouz[k] = 1;
      perm[i] = k;
   }
}
```



```
bezpecny_vyrobce() ->
  Data = zpracovani:vyrob(),
  receive
    {chciData, Pid} ->
      Pid ! data, Data,
      bezpecny_vyrobce();
    konec -> ok;
    after 10000 -> exit(timeout)
  end.
```

Nakonec si pořídíme ještě jeden proces, který na tyto dva bude dohlížet. Pokud se cokoliv nepovede (některý proces skončí a nebude to normální konec), tak celou operaci zkusí znovu.

```
hlidej(0) -> ok;
hlidej(Zbyva) -> receive
  {'EXIT', _, normal} -> hlidej(Zbyva - 1);
  {'EXIT', _, _} -> spawn(spojeni,
                          bezpecny_start, [])
end.

bezpecny_start() ->
  process_flag(trap_exit, true),
  Vyrobce = spawn_link(vyrobce@hroch, spojeni,
                       bezpecny_vyrobce, []),
  spawn_link(spojeni, bezpecny_spotrebitel,
                       [Vyrobce]),
  hlidej(2).
```

Napřed si nastavíme, že chceme dostávat zprávy o koncích známých, a poté pustíme dva procesy. Nakonec je necháme hlídat, když některý skončí normálně, odečteme si, kolik jich zbývá. Až žádný zbývat nebude, vše skončilo dobře. Pokud některý z nich skončí s chybou, celé to pustíme znovu, ale v novém procesu – druhý v té době může ještě existovat a za chvíli skončí s chybou, takže bychom dostali hlášku i o jeho smrti – ale to bychom už hlídali nový pokus a mysleli bychom si, že skončil špatně ten.

Úložka

Chceme něco, co bude rozdělovat práci mezi stroje. Budeme mít 3 druhy strojů – pracující (na těch se bude provádět práce), klienti (ti budou požadovat provedení nějaké práce) a servery, které budou práci rozdělovat. V zásadě něco podobného, jako úložka *Centrum práce* v minulém díle.

Bude několik modulů. Jeden modul (`client`) bude obsahovat funkci na zadání práce. Až práce skončí, bude nám výsledek funkce doručen jako zpráva. Druhý, `prace` se bude pouštět na pracujících strojích, třetí `server` na serverech. A v modulu `nastaveni` budou adresy serverů, ke kterým se mohou klienti a pracující připojovat. Pracující smí dělat maximálně jednu činnost v daný okamžik a nesmí se stát, že by někde práce čekala, pokud je některý pracující volný.

Samozřejmě nám jde o to, aby nám systém přežíval i v případě výpadků. Takže, co je potřeba zařídit:

- Když vypadne klient, tak přežít. *[3 body]*
- Když spadne vyhodnocování práce, pracující musí přežít a zaslat zprávu o chybě. Stejně tak, pokud práce bude trvat delší dobu, než nějakou stanovenou (pravděpodobně je zacyklená), tak je třeba ji ukončit a poslat zprávu o chybě. *[3 body]*
- Když vypadne pracující, tak přežít a práci přidělit nějakému jinému pracujícímu na udělání. Tedy, když vypadne pracující, tak by to klient vůbec neměl poznat. *[3 body]*
- Když vypadne některý ze serverů, tak se zařídit tak, aby zbytek systému přežil, daly se dál zadávat nové úkoly a stávající práce byla dokončená a výsledky doručeny. *[3 body]*

V každém z těchto případů můžete předpokládat, že z každé skupiny strojů ještě nějaké zbyly, tedy nestane se, že by vypadly všechny servery.

Vzorová řešení druhé série

22-2-1 Jednoznačný svět

Nejprve trochu o **výpočetním modelu** – zadání nebylo v pár věcech úplně důsledné, tak to nyní napravíme. V úlohách s potenciálně velkými čísly na vstupu, kde nás zajímá hlavně počet operací s těmito čísly, a ne až tak složitost jednotlivých operací, se často používá model s buňkami (integery) schopnými pojmout čísla maximálně konstanta-krát větší než max{číslo na vstupu, délka vstupu} (v našem případě považujeme délku vstupu za délku periody + délku aperiodického počátku), o čemž byla v zadání zmínka. Pro každý vstup má tedy model jinou kapacitu (ač se to může zdát divné, našemu měření to vyhovuje víc), speciálně tedy nelze v programu kalkulovat s kapacitou proměnných a maximální uložitelnou hodnotou jako s čísly, či dokonce s přetečením jako s rozpoznatelnou událostí – tyto termíny prostě nejsou v takovém slova smyslu definovány. Nekonečnou vstupní posloupnost si lze představit jako funkci, jejíž zavolání posune cestovatele na další křižovatku a vrátí její číslo. Teď už ale k řešení.

Hledání smyčky rozdělíme na dvě fáze: nejprve se do smyčky musíme dostat a všimnout si toho (smyčka rozhodně nemusí procházet první křižovatkou), poté zjistíme její délku. Jak smyčku najít? Představme si, že cestovatel si občas zapamatuje číslo nějaké křižovatky. Pak bude nějakou dobu chodit, a dorazí-li znovu na křižovatku se zapamatovaným číslem, má jistotu, že je v periodě. Pokud se mu to nějakou dobu nebude dařit, tak ono číslo zapomene, zapamatuje si místo něho současnou křižovatku a jde dál. Řekněme, že první křižovatku si bude pamatovat jen 1 krok, další 2, pak $4, 8, 16, \ldots, 2^k, \ldots$

A jak určit délku smyčky? To je již snadné, je to přesně počet kroků od posledního zapamatování křižovatky.

Pseudokódem by postup vypadal takto (`dál!` buď funkce vracející další křižovatku):

```
kde_jsem_teď := dál!
i := 1
j := 0
while ( pořád ):
    kde_jsem_byl := kde_jsem_teď
    while( j < i ):
        kde_jsem_teď := dál!
        j := j+1
        if( kde_jsem_teď == kde_jsem_byl ):
            return j
    j := 0
    i := i*2
```

Tímto postupem si cestovatel zapamatovává jiné křižovatky vždy po 2^k krocích, přičemž k počíná na 0 a po každém neúspěchu se zvětší o jedna. Ukažme nyní, že tímto postupem periodu jistě najdeme. Buď z délka počátečního úseku posloupnosti, ve kterém ještě perioda není, a p délka periody. Před l-tým zapamatováním nějaké křižovatky urazí cestovatel $\sum_{m=0}^{m<l} 2^m = 2^l - 1$ kroků. Uvažme nejmenší l

takové, že $2^l - 1 > z + p$, tedy cestovatel si zapamatuje křižovatku, která už je v periodě. Zároveň je ale i $2^l > p$, tedy dříve, než zapamatovanou křižovatku zapomene, znovu na ni narazí a pozná že je v periodě.

To, že cestovatel správně určí délku periody, je zjevné, perioda je definována jako vzdálenost dvou nejbližších výskytů libovolného prvku, což je přesně to, co cestovatel odkrokuje.

Dle modelu popsaného v úvodu potřebujeme 4 paměťové buňky (nikdy neukládáme číslo, jež by mohlo přetéci), což je asymptoticky optimální.

Časová složitost odpovídá počtu kroků. Ukázali jsme, že stačí $2^l - 1 + 2^l$ kroků, kde l je nejmenší, aby $2^l - 1 > z + p$. To znamená, že $2^{l-1} \leq z + p$, a tedy $2^l - 1 + 2^l < 2^{l+1} = 4 \cdot 2^{l-1} \leq 4 \cdot (z + p)$. Počet oběhnutých křižovatek je tedy $\Theta(z+p)$ – opět jsme na asymptotickém optimu, neboť vstup jsme nuceni číst sekvenčně.

Vojta Tůma

22-2-2 Zkouška

Pre začiatok si dom predstavíme ako graf, kde jednotlivé miestnosti sú vrcholy orientovaného grafu a hrana z vrcholu v do vrcholu u bude viesť práve vtedy, ak sa z odpovedajúcej miestnosti môžeme dvermi dostať priamo do druhej miestnosti. Ďalej si počet mincí v miestnosti odpovedajúcej vrcholu v nazvime hodnota vrchola v a označme $h(v)$.

Ak sa v dome nenachádzajú miestnosti, kde sa dá podvádzať, potom náš graf má tú vlastnosť, že je acyklický (hrany vedú len doprava a dole), a teda každú miestnosť môžeme navštíviť len raz. Všimnime si, že každá cesta v našom grafe zodpovedá validnej ceste po dome a naopak. Úlohou je teda nájsť cestu v našom grafe z vrchola odpovedajúceho miestnosti $(1, 1)$ do vrchola (M, N), na ktorej je súčet hodnôt vrcholov maximálny.

Na vyriešenie tohto problému použijeme dynamické programovanie.

Ako to tak v dynamickom programovaní chodí, na vyriešenie celého problému sa použijú riešenia podproblémov. V našom prípade budú podproblémy zodpovedať odpovediam na otázku: „Aká je najdrahšia cesta z počiatočného vrchola do vrchola v?“ Označme danú odpoveď $f(v)$. Pre $v =$ počiatočný vrchol je zrejme $f(v) = 1$.

Druhý krok v dynamickom programovaní je nájdenie obecného rekurentného vzťahu medzi podproblémami. Teda hľadáme vzťah na vyjadrenie $f(v)$, ak už vieme f pre nejakú množinu vrcholov.

Hľadaný vzťah je $f(v) = \max\{f(u)|u \text{ vedie hrana do } v\} + h(v)$. Tento vzťah je korektný, pretože každá cesta, a teda aj tá najdrahšia, končiaca vo vrchole v, musí pozostávať z cesty vedúcej do nejakého vrchola, z ktorého vedie do v hrana, plus posledný vrchol v.

Ak teda chceme vypočítať hodnotu $f(v)$, musíme vedieť hodnotu f pre všetky vrcholy, z ktorých vedie do v hrana. Dôležité je uvedomiť si, že práve acyklickosť grafu nám zaisťuje to, že nevznikne cyklická závislosť hodnôt f.

Posledným krokom v dynamickom programovaní je implementácia odvodeného vzťahu. Tá sa dá priamočiaro vykonať implementáciou zisteného rekurentného vzťahu pomocou rekurzie a memoizácie už raz vypočítaných hodnôt.

Pseudokód

```
f(v):
  if v == počiatočný vrchol
  then
    return 1
  if hodnota f(v) už raz spočítaná
  then // rovno vrátime hodnotu ktorú sme už
       // raz spočítali
    return f(v)
  else // inak danú hodnotu spočítame
    f(v) = max { f(u) | z u vedie hrana do v }
      + h (v)
    return f(v)
```

Pričom odpoveďou na náš vstup je $f(u)$, kde u je vrchol odpovedajúci miestnosti (M, N). Práve memoizácia už raz spočítaných hodnôt nám zaisťuje, že každú hodnotu $f(v)$ spočítame práve raz a celkovo pri tom vykonáme lineárne veľa práce od veľkosti grafu, pretože na každú hranu sa pozrieme práve raz.

Takto vieme vyriešiť úlohu, ak je graf acyklický. Ak sa však v dome nachádzajú podvádzacie miestnosti, potom sa v našom grafe nachádzajú cykly. Všimnime si, že ak sa v našom grafe nachádza cyklus, potom ak sa dostaneme v dome do miestnosti, ktorej odpovedajúci vrchol patrí do tohto cyklu, môžeme sa dostať do všetkých miestností na danom cykle a zase späť do danej miestnosti. Obecne môžeme povedať, že vrcholy tvoriace cyklus sú istým spôsobom ekvivalentné v tom zmysle, že ak sa dostaneme do ktoréhokoľvek z nich, potom môžeme navštíviť všetky vrcholy s ním na cykle a zase pokračovať z ľubovolného z nich. Takéto množiny vrcholov, z ktorých sa dá prechádzať z ľubovolného vrchola do ľubovolného a zase späť, sa volajú silno-súvislé komponenty orientovaného grafu.

Viacej si o nich môžete prečítať na Wikipedii na adrese `http://en.wikipedia.org/wiki/Strongly_connected_components`, rovnako aj o algoritme ako ich pre daný graf nájsť v čase lineárnom od veľkosti grafu.

Platí, že po navštívení ľubovolného vrcholu patriaceho do určitej silno-súvislej komponenty môžeme vyzbierať celú príslušnú komponentu a zase pokračovať z ľubovolného vrcholu v nej obsiahnutom. Preto si môžeme dovoliť každú silno-súvislú komponentu nahradiť jediným vrcholom s hodnotou rovnou sume hodnôt všetkých vrcholov patriacich do tejto komponenty.

Touto úpravou sme dostali acyklický graf, na ktorom použijeme algoritmus pre acyklické grafy, s tým rozdielom, že hľadáme najdrahšiu cestu z vrcholu, ktorý reprezentuje skontrahovaná komponenta obsahujúca počiatočný vrchol $(1, 1)$, do vrcholu, ktorý reprezentuje komponenta obsahujúca cieľový vrchol (M, N).

Rozbor časovej zložitosti

Treba si najskôr uvedomiť, že náš graf má MN vrcholov a každý vrchol má stupeň maximálne štyri, teda hrán bude tiež $\mathcal{O}(MN)$. Po skonštruovaní grafu aplikujeme algoritmus na nájdenie silno-súvislých komponent v čase lineárnom od veľkosti grafu, teda $\mathcal{O}(MN)$. Následne nahradíme každú komponentu jedným vrcholom, čím sa nám veľkosť grafu určite nezväčší a nakoniec aplikujeme algoritmus na nájdenie najdrahšej cesty v acyklickom grafe, ktorý tiež beží lineárne. Celková časová zložitosť je teda $\mathcal{O}(MN)$. V pamäti


```
int main()
{
    int N;
    scanf("%d", &N);
    float By[N][2];
    for (int i = 0; i < N; i++)
        scanf("%f %f", &By[i][0], &By[i][1]);

    // V těchto proměnných si budeme pamatovat
    // doposavaď nejúspěšnější kružnici.
    int maxA = 0, maxB = 1, maxC = 2, maxN = 0;

    // Teď: Pro každou dvojici bodů...
    for (int A = 0; A < N; A++)
        for (int B = 0; B < N; B++)
        {
            if (A == B) continue;

            kdokam Cy[N-2];
            int i = 0; int pozorujicich = N-2;

            // pro každý pozorující bod...
            for (int C = 0; C < N; C++)
            {
                if (C == A || C == B) continue;

                // zjisti, pod jakým úhlem úsečku AB pozoruje,
                // tedy jaký je úhel, který svírají vektory CA a CB.
                float CA[2], CB[2], BA[2];

                CA[0] = By[A][0]-By[C][0]; CA[1] = By[A][1]-By[C][1];
                CB[0] = By[B][0]-By[C][0]; CB[1] = By[B][1]-By[C][1];
                BA[0] = By[A][0]-By[B][0]; BA[1] = By[A][1]-By[B][1];

                float cosalfa = (CA[0]*CB[0]+CA[1]*CB[1])
                              / sqrt(CA[0]*CA[0]+CA[1]*CA[1])
                              / sqrt(CB[0]*CB[0]+CB[1]*CB[1]);

                if (cosalfa > 1 - eps || cosalfa < -1 + eps)
                { // C je na jedné přímce s A a B.
                    pozorujicich--;
                    continue;
                }

                if (BA[0]*CA[1]-BA[1]*CA[0] > eps)
                // Pokud se B nachází v jedné polorovině určené přímkou AB,
                    cosalfa += 2.0; // zahešuj/zatřiď úplně jinam.
                // (Předpis hned vypadne z vektorového součinu.)

                Cy[i].kdo = C;
                Cy[i++].kam = cosalfa;
            }

            qsort(Cy, pozorujicich, sizeof(kdokam), &comp);

            int nasobnost;
            for (int i = 0; i < pozorujicich; i++)
            {
                if (i > 0 && Cy[i].kam - Cy[i-1].kam < eps)
                    nasobnost++;
                else
                    nasobnost = 1;

                if (nasobnost > maxN)
                {
                    maxN = nasobnost;
                    maxA = A; maxB = B; maxC = Cy[i].kdo;
                }
            }
        }

    if (maxN == 0)
    {
        printf("V zadáni není kružnice.\n");
        return;
    }

    kruznice reseni = opis(By[maxA], By[maxB], By[maxC]);
    printf("S=(%f , %f), r=%f [%d bodu]\n",
                              reseni.Sx, reseni.Sy, reseni.r, maxN + 2);
}
```

si budeme držať vstup a reprezentáciu grafu, pamäťová zložitosť je preto tiež $\mathcal{O}(MN)$.

Peter Ondrúška

22-2-3 Kružnice

Jednoduchý přístup, o který se pokoušela valná většina z desítky zaslaných řešení, spočívá v tom, že se podíváme pro každou trojici bodů, jakou kružnici opsanou trojúhelník nad těmito body určuje, a pro každý z dalších bodů si ověříme, nachází-li se na této kružnici. Něco takového bude trvat $\mathcal{O}(N^4)$ a zabere $\mathcal{O}(N)$ paměti.

Samozřejmě je potřeba si rozmyslet, jak najít střed kružnice opsané: řešením je soustava dvou rovnic o dvou neznámých, které snadno vyjádříme z analytických předpisů pro rovnice os dvou stran daného trojúhelníka. Zvláštním případem tu bude stav, kdy tři vyšetřované body leží na jedné přímce, ten můžeme zapomenout.

Hezčí řešení tkví ve využití vlastností obvodového úhlu: zafixujeme-li si úsečku AB (pro každé dva body A, B), můžeme se pro každý další bod C ptát, pod jakým úhlem tuto úsečku AB vidí. Pokud nějaké dva C_1, C_2 koukají pod stejným úhlem (a ze stejné strany!), musejí se nacházet na společné kružnici s AB.

Když si zároveň vzpomeneme na důležitou vlastnost standardního skalárního součinu v euklidovské rovině, totiž že $\cos\alpha = u \cdot v/(|u| \cdot |v|)$, získáme snadno dobrý nástoj, jak „koukání pod stejným úhlem“ testovat: stačí porovnávat kosíny vypočítané pro vektory CA a CB.

Jak spočítat, kolik se pod jedním úhlem dívá bodů? Můžeme to řešit třeba tak, že si naměřené úhly (tedy kosíny) seřadíme podle velikosti (v čase $\mathcal{O}(N \log N)$), načež je projdeme a napočítáme shodné veličiny – už je máme vedle sebe. Tím získáme algoritmus časové složitosti $\mathcal{O}(N^3 \log N)$ při prostoru $\mathcal{O}(N)$.

Druhou možností, která se nabízí, je využít hešování. To nabízí ubití logaritmu za cenu toho, že se tak stane jen v průměrném případě. V závislosti na použitém algoritmu se nám totiž při hešování reálných čísel snadno může stát, že řešení kolizí u nevhodného vstupu zabere lineární čas.

Lukáš Lánský

22-2-4 Billboard

Označme si délky vzorů A a B. Pak si očíslujeme vagóny ve vzorech tak, že vždy v k-tém kroku budou na přejezdu vagóny s číslem k. To očíslování je pro první vlak $(0, 1, 2, \ldots, A-1)$ a pro druhý vlak $(0, B-1, B-2, \ldots, 2, 1)$, ale protože se vzor periodicky opakuje, bude na každém vagónu prvního vlaku kromě nějakého i také $i + A$ a analogicky na každém vagónu druhého vlaku kromě nějakého j také $j + B$.

Praktičtější je, když můžeme pole indexovat číslem vagónu, takže si druhý vzor přeskládáme v $\Theta(B)$ na (0, 1, 2, …, $B - 1$).

Nyní tedy víme, že v k-tém kroku bude na přejezdu vagón prvního vlaku s číslem $k \bmod A$ a vagón s číslem $k \bmod B$. Protože máme jen konečně možností setkání, ale nekonečně setkání samotných, musí se v jednu chvíli setkat znovu tytéž vagóny a od té chvíle se bude situace periodicky opakovat, protože máme periodické zadání. Kdy to bude poprvé?

V takovou chvíli musí určitě platit, že $k - p \equiv k \pmod A$ a $k - p \equiv k \pmod B$*. Pak tedy $p \equiv 0 \pmod A$ a $p \equiv 0 \pmod B$, neboli p je dělitelné jak A, tak B. Nejmenší číslo, které toto splňuje, je nejmenší společný násobek A a B. Snadno pak ověříme, že poprvé se situace musí opakovat ve chvíli, kdy se znovu setkají vagóny s čísly 0.

Pokud jsou A a B nesoudělná, není skoro co řešit. Platí totiž Čínská zbytková věta, která říká, že když mám k po dvou nesoudělných čísel $x_1, x_2, \ldots, x_k$, pak když si vyberu jakékoli celočíselné zbytky $m_1, m_2, \ldots, m_k$: $0 \le m_i < x_i$ pro všechna i, pak existuje právě jedno číslo C takové, že $C \equiv m_i \pmod{x_i}$ pro všechna i a $0 \le C \le x_1 x_2 \ldots x_k - 1$.

Podle této věty (pro $k = 2$) se tedy mezi nulou a $AB - 1$ potká každá dvojice vagónů právě jednou. Takže stačí spočítat poměr počtu všech dvojic nízkých vagónů ku počtu všech vagónů a máme vyhráno.

Když jsou soudělná, neplatí Čínská zbytková věta. To ale můžeme jednoduše obejít. Označme si $A = ad$ a $B = bd$, kde d je jejich největší společný dělitel. Pak už jsou a a b nesoudělná. V k-tém kroku máme tedy vagón s číslem $k \bmod da$ a $k \bmod db$. Když ale spočítám zbytek po dělení těchto čísel dělitelem d, zjistím, že je stejný.

Rozdělíme si tedy každý vlak na d podvlaků, Podvlak A_i bude obsahovat všechny vagóny z A, jejichž čísla dávají po dělení d zbytek i, podobně podvlak B_i. Každý vagón podvlaku A_i určitě potká každý vagón podvlaku B_i a žádný z jiného podvlaku.

Teď stačí úlohu vyřešit nezávisle pro každý z d podvlaků (nesoudělných délek), sečíst dohromady počet setkání v každém podvlaku a vydělit počtem setkání celkem. Pro každý podvlak proběhne ab setkání, podvlaků je celkem d, což dává dohromady abd – nejmenší společný násobek A a B. Tedy každé setkání proběhne opravdu právě jednou.

Časová složitost algoritmu je $\mathcal{O}(A + B)$, protože na každý vagón se podívám právě jednou. Rychleji to nejde – musím alespoň přečíst celý vstup. Kdybych dostal vstup jako seznam pozic nízkých vagónů, mohl bych se dostat na $\mathcal{O}(A_N + B_N)$, kde A_N a B_N jsou počty nízkých vagónů ve vlacích, nicméně pro vlak, který by sestával z mnoha nízkých vagónů a jednoho vysokého, dojdeme zase asymptoticky k $\mathcal{O}(A + B)$. Paměti mi stačí $\mathcal{O}(A + B)$ na uložení vstupu a na ostatní jen konstatně mnoho.

Jan „Moskyt“ Matějka

22-2-5 Pružinky

Hodně z vás hraje Pružinky natolik rádo, že jste hru nechali hrát i svůj program. Nejjednodušší bylo si v poli o každém hráči pamatovat, který z nich je ještě ve hře a který již ne. Při výpočtu složitosti nesmíte zapomenout, že ke konci můžete projít skoro všechny hráče, než narazíte na nějakého hrajícího. Pokud si označíme H počet hráčů a S délka slova, vyjde tedy časová složitost $\mathcal{O}(H^2 S)$ – provedeme H kol hry, v každém řekneme S písmenek a pro každé z nich přeskočíme až H hráčů, než najdeme dalšího hrajícího. To jde zrychlit na $\mathcal{O}(H^2)$, když hráče, který vypadne, rovnou odstraníme z pole – hrajeme H kol, v každém v konstantním čase najdeme, kdo vypadne, ale pak potřebujeme čas $\mathcal{O}(H)$ na „setřepání“ pole. Pokud místo pole použijeme seznam, zvládneme hráče odstranit v konstantním čase, ale zato musíme skákat po jednom hráči, takže jedno kolo trvá $\mathcal{O}(S)$ a celá hra $\mathcal{O}(HS)$.

* $x \equiv y \pmod z$ znamená „x a y dává stejný zbytek po dělení z“

22-2-2 – Zkouška – program

```
#include <cstdio>
#include <stack>
#include <algorithm>
using std::max;
using std::stack;

#define MAX 1047

// hodnota policka
int F[MAX][MAX];
// podvadzacie policko
bool G[MAX][MAX];
// cislo silno-suvislej komponenty v ktorej je policko
int C[MAX][MAX];

// suma v danej komponente
int H[MAX*MAX];
// najdrahsia cesta konciaca v danej komponente
int R[MAX*MAX];

// navstivene vrcholy v DFS1 a DFS2
bool V1[MAX][MAX];
bool V2[MAX][MAX];

int X, Y, K;

typedef struct { int x, y; } Point;
stack<Point> S;

// je miestnost na mape ?
bool Valid(int x, int y) {
    return x >= 1 && x <= X && y >= 1 && y <= Y;
}

void DFS1(int x, int y) {
    if (!Valid(x,y) || V1[y][x])
        return;
    V1[y][x] = true;

    DFS1(x+1,y);
    DFS1(x,y+1);
    if (G[y][x]) {
        DFS1(x-1,y);
        DFS1(x,y-1);
    }

    S.push((Point){x,y});
}

void DFS2(int x, int y, int c) {
    if (!Valid(x,y))
        return;

    // ak sme prekrocili hranicu komponenty
    // skontrolovat existenciu drahsej cesty
    // do povodnej komponenty
    if (c != C[y][x])
        R[c] = max(R[c], R[C[y][x]]);

    if (V2[y][x])
        return;
    V2[y][x] = true;

    C[y][x] = c;

    // hodnotu policka pripocitat k sume komponenty
    H[c] += F[y][x];

    DFS2(x-1,y,c);
    DFS2(x,y-1,c);
    if (G[y][x+1]) DFS2(x+1,y,c);
    if (G[y+1][x]) DFS2(x,y+1,c);
}

int main() {
    // naitat mapu
    scanf("%d %d %d", &Y, &X, &K);
    for (int y = 1; y <= Y; y++)
        for (int x = 1; x <= X; x++)
            scanf("%d", &F[y][x]);

    // nacitat podvadzacie policka
    for (int x, y, i = 0; i < K; i++) {
        scanf("%d %d", &x, &y);
        G[y][x] = true;
    }

    // prehladat transponovany graf a zoradit vrcholy
    // podla klesajuceho casu opustenia
    for (int y = 1; y <= Y; y++)
        for (int x = 1; x <= X; x++)
            DFS1(x,y);

    // prejst vrcholy v poradi klesajuceho casu opustenia,
    // oznacit silno-suvisle komponenty a rovno spocitat
    // najdrahsiu cestu konciacu v danej komponente
    for (int i = 1; i <= X*Y; i++) {
        DFS2(S.top().x,S.top().y,i);
        S.pop();
        R[i] += H[i];
    }

    printf("%d\n", R[C[Y][X]]);
    return 0;
}
```

22-2-3 – Kružnice – program

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

typedef struct
{
    int kdo;
    float kam;
} kdokam;

typedef struct
{
    float Sx, Sy;
    float r;
} kruznice;

kruznice opis(float A[], float B[], float C[]);
int comp(const void *a, const void *b);

const float eps = 0.0001f;
```

Rychlejší algoritmus získáme, když se zamyslíme nad rekurentním vzorcem pro nalezení vítěze. Nechť $F(H,S)$ říká, který z H hráčů vyhraje (hráče budeme číslovat od 0 do $H-1$, abychom mohli počítat modulo H). $F(1,S)$ je zjevně 0. Pokud $H > 1$, vypadne v prvním kole hráč S mod H, a pak bude hra vypadat podobně jako hra $F(H-1,S)$, jen s tím rozdílem, že nezačínáme hráčem 0, nýbrž S. A jelikož jsme posunuli o S (modulo H) začátek hry, musí se úplně stejně posunout i konec. Dostaneme tedy vzorec

$$F(H,S) = (F(H-1,S)+S) \bmod H.$$

Podle tohoto vzorce lze už v lineárním čase vítěze dopočítat:

```
int main(void) {
        int i, H, S;
        int vitez=0;
        scanf("%d%d", &H, &S);
        for (i=2;i<=H;i++)
                vitez=(vitez+S)%i;
        printf("%d\n", vitez+1);
        return 0;
}
```

Pojďme zkusit algoritmus ještě trochu zrychlit. Pokud je S výrazně menší než H, hra se ze začátku chová docela pravidelně: prvních $k = \lfloor H/S \rfloor$ kol se hráči nebudou opakovat, takže vypadnou ti s čísly $0, S, 2S, \ldots, (k-1)S$. Tím jsme hru převedli na nějakou s $H-k$ hráči, jen tentokrát není pouze posunutá o S kroků, ale navíc „prostrkaná“ jednotlivými vypadlými hráči. Abychom spočítali, jak, označme $f = F(H-k,S)$ a $z = H - kS = H \bmod S$ (výsledek menší hry a počet hráčů, kteří si v prvních k kolech nezahráli). Pokud $f \le z$, bude výsledek menší hry ležet v posledních z hráčích té větší a pouze jej posuneme: $F(H,S) = (f+kS) \bmod H$. V opačném případě bude ležet na $(f-z)$-té pozici od počátku, ovšem musíme přeskakovat hráče, kteří už vypadli – podíváme se, do které $(S-1)$-tice mezi $0, S, 2S, \ldots$ pozice $f-z$ padla a přičteme patřičný počet vypadlých hráčů: $F(H,S) = (f + kS + \lfloor (f-z)/(S-1) \rfloor) \bmod H$.

Jak moc tato úprava pomůže? Dokud $H > S$, vypadne v každé iteraci algoritmu přibližně jedna S-tina hráčů, takže se H zmenší na $(1-1/S)H$. Velikost hry tedy klesá exponenciálně, a proto iterací bude $\mathcal{O}(\log_{S/(S-1)} H)$. Jakmile ovšem H klesne pod S, nebude nový algoritmus o nic lepší než starý a dohrajeme v čase $\mathcal{O}(S)$. Celková složitost je tedy $\mathcal{O}(S + \log_{S/(S-1)} H)$.

Pokud si navíc vzpomeneme, že $\log_a b = \log a / \log b$ a že pro malé $\varepsilon > 0$ platí $\log(1+\varepsilon) \approx \varepsilon$, můžeme odhad složitosti upravit na:

$$\begin{aligned} &\mathcal{O}(S + \log H / \log(S/(S-1))) = \\ &= \mathcal{O}(S + \log H / \log(1 + 1/(S-1))) = \\ &= \mathcal{O}(S + S \log H) = \mathcal{O}(S \log H) \end{aligned}$$

(‘$\approx$’ není ‘$=$’, ale rozdíl se schová do $\mathcal{O}$-čka). Zrychlený algoritmus je tedy pro velká H výhodnější.

(Mimochodem, pro $S = 2$ byla naše úloha známá už v antických dobách, konkrétně od historika Josepha Flavia. Tato konkrétní varianta má moc pěkné řešení pomocí dvojkových zápisů čísel. Zkuste se Wikipedie zeptat na „Josephus Problem.“)

Martin Mareš & Jitka Novotná

22-2-6 Otázka

Ačkoli se ostřílení programátoři na prohledávání do šířky dívají trochu s despektem, pravdou je, že i dobré BFS (jak se mu zkráceně poanglicku – Breadth-First Search – říká) si občas zaslouží procvičit. Úloha s magickým obdélníkem stačila řešit právě tímto algoritmem. Stavů je sice skutečně mnoho (exponenciálně k velikosti obdélníku) a náš algoritmus by v nejhorším případě musel projít všechny, ale pro 8 to zas tolik nevadilo, neboť 8! je relativně malé číslo.

Algoritmus pracuje tak, že postupně prochází od startovního obdélníku všechny obdélníky, které z něj můžeme vytvořit pomocí povolených transformací, a ukládá si nové pozice. Jakmile zjistí, že v dané „hladině“ (množina všech obdélníků, které jde vytvořit pomocí k operací) se ještě hledaný obdélník nenachází, prohledá postupně celou další hladinu.

K prohledávání celé jedné hladiny do šířky se používá datová struktura, tzv. fronta (seznam prvků, kde kdo dřív přijde, ten dřív mele a odchází) a dovolili jsme si využít již zavedenou implementaci v C++. Pokud byste se rádi dozvěděli více o prohledávání do šířky a práci front, nahlédněte do našeho textu „Recepty z programátorské kuchařky“, sekce „Grafy“, které najdete mimo jiné na našich webových stránkách.

Jako u dalších příkladů s BFS, i zde bylo třeba si pamatovat, které permutace jsme už probrali. Jinak bychom příliš často hledali tam, kde už jsme byli, což by naši závodní želvu zpomalilo na rychlost obyčejné želvy.

Malý zádrhel tkvěl v tom, že 8! možností si musíme zapamatovat alespoň trochu chytře, jinak se nám všechny probrané permutace do paměti nevejdou. Nejlépe tak, aby se vešly do jednoho integeru. $8! = 40320$, to by se rozhodně mělo vejít. Jakou zvolit metodu, abychom uměli číslo rychle přeložit na standardní reprezentaci obdélníku = permutaci?

Metoda Céčkového vousáče

Znalce Céčka, C++, C# a sběratelé Céček hned napadne uložit si celou permutaci do jednoho integeru pomocí bitového kódování. Pro zakódování čísel od 0 do 7 včetně nám postačí $\log_2 8 = 3$ bity, v jednom integeru máme bohatých 32 bitů, což je o 8 více, než potřebujeme. K jednotlivým bitům se můžeme dostat pomocí operace „x modulo 8“, která vrátí zbytek po dělení osmi, čili poslední tři bity daného čísla x. V Céčku bychom napsali `x % 8;`.

Jakmile přečteme první tři bity z x, x můžeme „oříznout“ o poslední tři bity pomocí operace „right shift“.To znamená, že první tři číslice (zprava) v binárním zápisu smažeme, a aby to opět bylo platné číslo, tak zleva doplníme nulami. Číslo `101010` se posunem o 3 doprava změní na `000101`. V Céčku tuto operaci zapisujeme jako `x >> 3;`.

Metoda sčítajícího Pascalisty

Co dělat, když se nám takto nízkoúrovňově s čísly pracovat nechce, nebo to náš jazyk (například Pascal[1]) příliš nepodporuje? Nezbývá, než si vymyslet jinou metodu.

Jedna z metod, kterou používáme ve vzorovém řešení, je založena na klasickém principu: přičteme $i!$ tolikrát, které číslo chceme zapsat, a pro získání původního čísla na i-té pozici jen podělíme zakódované číslo správným faktoriálem $i!$. Abychom měli jednodušší život, budeme postupovat pozpátku, tedy první člen zakódujeme násobkem $7! = 5040$ a posledním $0! = 1$. Rovněž si budeme uchovávat čísla o jedno menší, než jsou, ušetří to paměť – nulu si pamatovat nepotřebujeme.

Avšak pozor! Kdyby například na páté pozici bylo číslo 8, mohli bychom nešikovně zákódovat osmičku jako $7 \cdot 3! = 42 > 24 = 4!$, a to by pokazilo rozluštění čísla zpět na permutaci. Proto si pomůžeme snadným pozorováním: Pro první číslo, násobek 7!, tento problém nenastane, a pro každé další (p-té) číslo x přičítám $(8-p)!$ jen tolikrát, kolikrát v seznamu čísel 1..8 sáhnu na doposud v zadané permutaci nepoužitá čísla, než dojdu k číslu x.

Ukážeme si to na příkladu: pokud nyní máme v permutaci na 5. pozici 8 a víme, že obdélník začíná $[4,5,3,7]$, spočítáme si, že seznam dosud nepoužitých čísel je $[1,2,6,8]$ a přičteme $3 \cdot 3! < 4!$ (neboť osmička je v tomto seznamu třetí, indexujeme-li úsporně od nuly). A máme nasčítáno!

—

Za vzorové řešení ze svých archivů děkujeme Zbyňku Faltovi.

Martin Böhm & Martin „Bobřík“ Kruliš & CodEx

22-2-7 Sto oslů umořilo nic

Producent a konzument se skladištěm

Toto se, kupodivu, skládá ze tří částí – producent, konzument a skladiště.

Producent je docela jednoduchý. Vyprodukuje jeden kousek dat a odešle ho do skladiště. Když dostane doručenku, vyprodukuje další a zase odešle.

Konzument funguje v jistém smyslu opačně. Pošle do skladu požadavek, že chce data a počká až přijdou. Poté je zpracuje a pošle nový požadavek.

Nejsložitější část je vlastní skladiště. Kdyby bylo nekonečné a mělo už nekonečně mnoho dat uložených, tak budeme jen vyřizovat požadavky o nová data (odešleme je a znovu čekáme na požadavek) a požadavky o zařazení dat (přidáme a odešleme doručenku).

Potřebujeme ale ošetřit případy, kdy jsme buď úplně plní, nebo úplně prázdní. To uděláme tak, že v době, kdy jsme plní, nebudeme zprávy s novými daty přijímat (pomocí `when`) a data počkají ve frontě zpráv. Proto bude v té době čekat i producent a nic produkovat nebude – nedostal doručenku. Obdobně, když nebudeme mít žádná data, nebudeme přijímat požadavky o data a ty počkají do doby, než nějaká data přijdou.

Všimněme si, že skladiště (v ukázkovém vzoráku se jmenuje `buffer`) si nikde nepamatuje, koho obsluhuje. Vždy má jeho PID ve zprávě, to rovnou obslouží (a nebo i s PID počká ve frontě zpráv) a zase ho zapomene. Takže nám funguje i v případě, že konzumentů či producentů je jiný počet než 1.

Balené funkce

Tato úloha byla více na pozorné čtení než na programování. Použijeme lambda funkci a uvnitř si tu opravdovou funkci zavoláme i s parametrem:

```
producent(Skladiste, fun() -> generuj(42) end).
```

Samozřejmě, její parametr nemusí být konstanta, může to být cokoliv, co je k dispozici v tomto místě kódu.

Mnoho z řešitelů chtělo přidávat nový parametr do původního rozhraní. To ovšem není řešení dané úlohy (kromě toho, že je to silně nepohodlné, přepsat celé vnitřnosti kvůli změně počtu parametrů), protože požaduje dostat funkci s parametrem do stávajícího rozhraní.

Centrum práce

Napřed si popíšeme, jaké vlastně má modul rozhraní. K použití jsou tu dvě funkce (ostatní jsou exportované jen proto, aby se daly předat do `spawn`): `start` a `pracant`.

Když spustíme `start`, tak nám vrátí dvojici. První prvek je funkce, které, když se jí předá nějaká funkce, tak ji předá do centra práce jako úkol. Druhý prvek je PID centra práce.

Druhá funkce, `pracant` spouští pracanta. Ten potřebuje znát PID centra a začne vykonávat činnosti, které mu centrum pošle.

Nyní, jak funguje funkce na zadání práce? Jen vezme parametr a centru pošle zprávu obsahující předanou funkci (je vytvořena nová funkce pro každé PID centra, nese si ho s sebou).

Pracant hned po startu pošle centru zprávu, že by rád dostal nějakou práci a k ní přiloží své PID. Když mu přijde odpověď, spustí funkci, která v ní byla. A potom vše opakuje – opět pošle žádost o novou práci a až přijde, tak ji vykoná.

Nakonec zde máme vlastní centrum práce. Mohli bychom si pamatovat pracanty a jednotlivé úkoly. Ale to by bylo pracné a dělat to nebudeme. Raději budeme fungovat tak, že vždy přijmeme jeden úkol, přijmeme jednoho pracanta a tomu úkol přijmeme.

Jak je možné, že tohle funguje? Jednoduše, fronta zpráv si za nás pamatuje vše potřebné. Když máme k dispozici jak pracanty, tak úkoly, tak je prostě vybíráme z fronty a práci přidělujeme. Pokud se nám jedněch z nich nedostává, pak nemá cenu ty druhé vybírat z fronty a někde si je shromažďovat, stejně musíme s přidělením čekat.

A nakonec, jaká chyba se často vyskytovala? Že rozhraní bylo navržené zcela nesmyslně. Tedy, byl nějaký „generátor práce“, který centrum plnil zcela zbytečnou prací (nebo, v lepším případě, k zadání nové práce bylo potřeba udělat generátor, ten vygeneroval jednu práci a skončil). Toto ale nejen že přidávalo novou (nesmyslnou) práci, ale také si ji to vymýšlelo, což je kromě toho, že je to nepohodlné, více méně v nesouladu se zadáním.

[1] Většina implementací Pascalu bitové operace podporuje, ale syntaxí se mohou lišit.