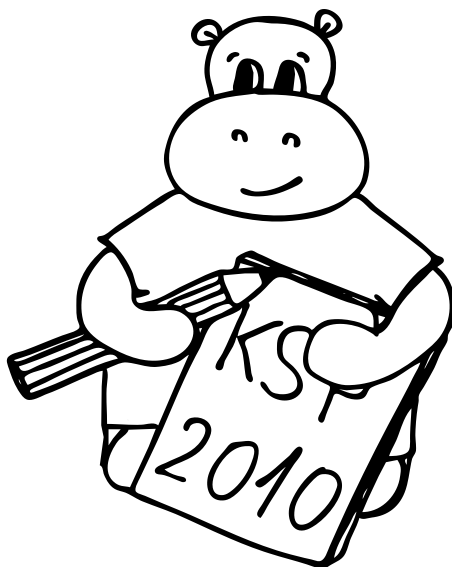
MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXII. ročník – 2009/2010



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXII. ročník – 2009/2010

matfyzpress

Praha 2010

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-124-8

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý druhý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešením úloh našeho semináře získávají středoškoláci praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé disciplíny informatiky.

Ročník *KSP* je obvykle rozdělen do pěti *sérií*, neboli kol. Během každé rozešleme řešitelům zadání sedmi úloh okořeněných příběhem. Poslední úloha je doplněna tzv. *seriálem*, což je povídání o nějakém zajímavém informatickém tématu prolínající se celým ročníkem. Ten je zde uveden samostatně.

Na sepsání řešení v klidu domácího krbu a odevzdání přes naše stránky nebo poštou bývá několik týdnů, poté vše opravíme a výsledkovou listinu se vzorovými řešeními vystavíme na internet a pošleme poštou s další sérií. Závěrečným bonbónkem je pak pravidelné týdenní *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě). Pro začínající řešitele již několik let pořádáme trochu kratší jarní soustředění, kam může jet kterýkoliv středoškolák se zájmem o programování či informatiku, i když třeba ještě nic nevyřešil.

KSP se i přes svou dlouhou tradici neustále vyvíjí. V tomto ročníku přibyla jedna těžká úloha pro pokročilé a již pár let zařazujeme do zadání také *praktickou úlohu* (odevzdávanou pouze ve formě zdrojového kódu přes internet do vyhodnocovacího systému CodEx). Vylepšili jsme odevzdávání přes internet, nyní je možné stáhnout si opravené úlohy, ještě než je donese pošta. V prosinci se navíc objevila na internetu nová soutěž nazvaná Programátorská džungle, jež umožňuje komukoliv vyzkoušet své dovednosti na tzv. *open datových úlohách* (soutěžící si stáhne u každé úlohy vstup a má za úkol vypočítat jakýmikoliv prostředky do hodiny výstup) nebo na krátkých logických hříčkách.

Chcete-li se na cokoliv zeptat, ať už ohledně semináře, studia na naší fakultě nebo nějakého informatického či programátorského problému, neváhejte a napište nám na diskusní fórum na stránce *http://ksp.mff.cuni.cz/forum/* nebo na naši poštovní adresu:

Korespondenční seminář z programování
KSVI MFF
Malostranské náměstí 25
118 00 Praha 1
e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Zadání úloh

Díky neuvěřitelnému pozitivnímu ohlasu na kratší příběhy (přesné číslo z bezpečnostních důvodů neuvádíme, pouze napovíme, že bylo nezáporné a zároveň nekladné) jsme se rozhodli v nich pokračovat. Dnešní porci úložek s hranolky vám s úsměvem na tváři servíruje Lukáš Lánský.

Byl jednou jeden vlak. 5. července 2009, kolem jedenácté hodiny dopolední, v Brně, právě vyjížděl z nádraží. Oplýval elektrickou lokomotivou a šesti vagónky.

Takto určený vlak byl nejspíš jenom jeden: jel z Brna do Pardubic a na české poměry rychle. Po celý zbytek vyprávění nás na něm bude zajímat toliko třetí kupé v druhém vagónku odpředu.

Naše motivace ke sledování konkrétně tohohle objektu bude taková, že si tam mezi sebou lidé začali povídat (což už je samo o sobě zajímavé) a že si tam začali povídat o něčem důležitém: tomto textu. Tedy, „tomto“ …

⋆⋆⋆

Pavel dostudoval první ročník čtyřletého gymnázia a nudil se. To tam bylo prvotní okouzlení novým prostředím a do jeho chování a postojů se během posledního půlroku znovu vkradla apatie: vždyť proplouvat školou je tak snadné!

Nebylo tomu tak vždy. V osmé třídě základky zavadil o matematickou olympiádu: dostal diplom a poukázku do knihkupectví, vůbec mu to přišlo jako příjemné setkání, které by si byl rád zopakoval, kdyby nebyl tak strašně líný dělat příští rok domácí kolo. Hm, dobrá, možná tomu tak vždy bylo: Pavlovi se však nedal upřít talent.

Do kupé přišel bez většího zavazadla, jel k babičce. Vlastně měl s sebou jenom jednu knížku.

„Tak ta je hustá,“ osloví ho dívka sedící u okénka. Kouká po knížce. „Sice trošku mainstreamová, ale hustá.“

Pavel se chtěl nejdřív nevrle ohradit, že Gaiman není žádný mainstream, ale po pečlivém prostudování dívčina vzezření zvolil vlídnější „Ahoj, jmenuju se Pavel!“

Vypadala totiž takto: Středně vysoká, snad osmnáctiletá, dlouhé hnědé vlasy.

Odpověděla: „Já jsem Alča.“

V kupé nebyli sami, ale o tom až někdy dál. Prozatím jen: Třetí člověk, zhruba dvacetiletý, seděl u dveří a vypadal, že je hluboce pohroužen do knížky o Haskellu. Našim dvěma hlavním aktérům to po celou dobu cesty nepřišlo zajímavé, ani jeden z nich neprogramoval v něčem, čeho název by nezačínal písmenem C.

Ta slečna že programuje? Pavel na to přišel záhy, když poznamenala:

A: Co ti z ní čouhá, je KSPéčko?

P: Těžko, vždyť ani nevím, co to „KSPéčko" je.

A: Ale je to KSPéčko! Poznám to podle obrázků.

P: Hroši? Divný. Aha, dali nám to ve škole, použil jsem to jako záložku.

A: Tohle zadání je divný, ale ne protože by na něm byli hroši. To je normální a v pořádku a vůbec opovaž se říct cokoliv proti hrochům!

Vypadala naštvaně!

P: Promiň! Hm, můžeš...

A: No?

P: Můžeš mi říct, co to KSP je?

A: Můžu ti to ukázat. Ale jak říkám, tohle zadání je divné ... Podívej: KSP znamená Korespondenční seminář z programování. Zhruba to funguje tak, že když vyřešíš, co je na tomhle papíře za programátorské příklady, dostaneš body a budeš se cítit chytře.

P: Žádný peníze?

A: Ne. Je to prostě něco jako škola, má tě to vzdělávat.

P (skepticky): Aha. Hm, takže Gaiman...

A: Gaiman je proti tomuhle nudnej. Podívej, obyčejně ta zadání vypadají tak, že tam je nějaký příběh, v rámci kterého jsou ty úlohy.

P: Aby se mi v tom hůř hledalo zadání?

A: Ne, to aby se orgové vyblbli. S řešiteli to nesouvisí ... Nekoukej tak nedůvěřivě, tak to je!

P: Tady na tom papíře příběh není. Jen pár příkladů.

A: Jo. Podívej, jak tady vypadá první úloha: „Buď G orientovaný graf, jehož hrany jsou ohodnoceny právě jedním prvkem z množiny $\{+,-\}$. *Je dán počáteční vrchol S a cílový vrchol C. Nechť P je cesta z S do C: najděte takové P, že se po ní co nejméněkrát střídají znaménka na hranách. Kupříkladu pokud existuje cesta se znaménky* `+++-------+`*, vybereme spíš ji než takovou se znaménky* `+-+-+`*, přestože je co do počtu použitých hran delší."*

P: To je strašný! Co je to graf? Co je to hrana, vrchol?

A: Kus matematiky. Když si na něj zvykneš, budou se ti takovýhle úlohy řešit snáz, dobrou knížku o tom napsal třeba Deml nebo Matoušek s Nešetřilem[1]. Podstatný je, že takhle se tam podobné věci nezadávají! Nesmí z toho tak trčet matika, na tu si musíš přijít sám, v tom je kus legrace!

P: Jak by to teda mělo vypadat?

A: Jejda, to záleží. Vlastně na tom nezáleží, jen ... Já nevím, třeba takhle:

[1] viz *http://ksp.mff.cuni.cz/study/paperbooks.html*

22-1-1 Alčina interpretace — 10 bodů

Máme velký dům se spoustou pokojů, mezi některými z nich vedou schodiště: z jednoho pokoje do druhého se buď stoupá, nebo klesá. Z nějakého důvodu hledáme cestu z jednoho pokoje do druhého tak, abychom co nejméněkrát museli přestat vycházet schody a začít scházet, nebo naopak přestat scházet a začít vycházet.

A: Rozumíš?

P: Hm, jo. Ale můžu být upřímný? Proč kolem toho tak jančíš? Tak změnili styl. Nová krev, mladí lidé s názory radikálně odlišnými od svých předchůdců … Nic zvláštního, nic, nad čím stojí za to mluvit, když slunko svítí, prázdniny sotva začaly a my oba jsme tak báječně mladí. (Mrk.)

A: (Jemně zdvižené obočí, jinak nic.) Ale ty další úlohy jsou ještě horší! Vždyť podívej na tu dvojku … Na Catalanova čísla a nic jiného. Kdo je zná, má to hned hotové!

P: Život není peříčko: vždyť tak to bude vždycky! Každou úlohu někdo zná.

A: Ale Catalanova čísla jsou mainstream.

P: Jako Gaiman, viď …

A: Jo!

P: Tak vymysli něco lepšího! Vymysli krásnou novou úlohu a pošli jim ji spolu se svým řešením, vždyť jistě budou rádi.

A: Vidíš ten sad, kolem kterého projíždíme?

P: Ty vzory jsou zajímavé. Jak jsou stromy vysázené do čtvercové mřížky, ustupují nám z výhledu souhlasně, a stejně tak do něj vstupují. Je to pěkné a dělá to zajímavé obrázky.

A: (Zamyšleně.) Představ si, že stojíš v počátku kartézské soustavy souřadnic a sad jsou všechny body se souřadnicemi (a, b), kde a, b jsou nezáporná celá čísla, jejichž součet je menší než nějaké N.

A: Které stromy v takové zjednodušené situaci vidíš?

P: Hmpf. :-(

A: Zkus si to nakreslit takhle!

```
. . . . . . . .
. x . . . . . .          P    1/1  2/2  3/3 ...
. x x . . . . .          0/1  1/2  2/3  ...
. x x x . . . .          0/2  1/3  ...
. x x x x . . .          0/3  ...
. P x x x x . .          ...
```

P: Aha, takže je to vlastně úloha na hledání zlomků v základním tvaru! Protože ty stromy, které „nejsou v základním tvaru“, jsou takovými určitě zakryté! A ty, které jsou, nejsou!

A: Příznačně řečeno.

22-1-2 Sad — 9 bodů

Váš program dostane číslo N a vy máte za úkol vypsat podle velikosti seřazené všechny zlomky v základním tvaru, které jsou větší nebo rovny nule a menší nebo rovny jedničce a ve jmenovateli mají méně nebo rovno N.

Pro $N = 5$ nechť vypíše $0/1, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 1/1$.

Nebo, chcete-li: dostanete velikost strany trojúhelníka z Alčina obrázku a máte vypsat seznam stromů, které jsou vidět z bodu $0/0$.

P: Uznávám, i když jsi evidentně dobrá (mrk), je divné, že vymýšlíš z ničeho úlohy lepší, než jaké zadávají v KSP.

A: Matematika je všude kolem. Koukej se kolem sebe trošku a uvidíš to taky.

P: Kolik ti je let?

A: Dvacet.

P: Vypadáš na osmnáct!

A: Děkuju. :-)

P: Mně je šestnáct…

A: Vypadáš na osmnáct!

P: Děkuju. :-)

A: Hlavně to znamená, že není nic ztraceno: máš spoustu času na sobě pracovat, pozorně číst a pečlivě řešit a jednou se třeba sám postaráš o to, aby se už víckrát nestal tenhle trapas … Já sama jsem začala řešit taky ve druháku.

P: Co je tam dál?

A: Codexová úloha. Abys věděl, ostatní úlohy vlastně ani nemusíš programovat nebo dokonce ladit. Stačí na kus papíru dobře popsat a odůvodnit vymyšlený algoritmus. V každé sérii je ale jedna praktická, tu napíšeš buď v Pascalu, Céčku, C++ nebo C#, odladíš ji a pak ji necháš automaticky vyhodnotit přes webový rozhraní.

P: Aha. No, to není tak zlé. Nemusím nic popisovat ani dokazovat…

A: Má to svoje jemnosti. Musíš si dávat pozor na implementační detaily, aby sis dobrý algoritmus nezpomalil nějakou špatnou prací se vstupem nebo nevhodnou reprezentací dat. Musíš si taky dobře uvědomit, jak vypadají všechny korektní vstupy, abys nezapomněl ošetřit různé speciální případy. Už jenom to, že se tvůj program kompiluje a spouští někde daleko, je trošku nepříjemný, protože musíš programovat podle toho, co tam mají. Oni se teda snaží jít podle

norem: myslím, že v případě C++ to znamená, že tam STL mají, zatímco Crt v Pascalu ne.

P: A proč se tomu říká Codexová úloha?

A: Tak se jmenuje to vyhodnocovadlo, CodEx, Code Examiner. Je to na webu KSP, hlásíš se tam stejně jako do submitovátka.

P: Submitovátka?

A: Tím se elektronicky odevzdávají řešení.

P: Jo. Jasně. Tak co tam třeba teď mají za Codexovou úlohu?

22-1-3 Sazba — 10 bodů

Na vstupu dostanete text a číslo N. Vaším úkolem je zarovnat ho do bloku tak, aby byl co nejhezčí. Protože krása je věc názoru, zadefinujeme si pro naše účely vhodné objektivní měřítko: pro každou posloupnost mezer mezi slovy délky k vezmeme číslo $(k-1)^2$ a tato čísla sečteme přes všechny posloupnosti mezer ve vysázeném textu. No a sazba je nejhezčí, pokud je tento součet nejmenší.

Slova nelze dělit mezi řádky a máte zaručeno, že se v textu neobjeví slovo delší než N. Na výstup od vás nechceme vypisovat vytvořené zarovnání, ale pouze minimální výše popsaný součet pro daný text, tedy číslo.

Například pro text „This is the example you are actually considering.“ a $N=28$ má program vypsat 12, protože optimální zarovnání je

```
This  is  the  example  you
are actually  considering.
```

a ohodnocení $1+1+1+4+1+4=12$.

P: To zní rozumně!

A: To je mé převyprávění. Ušetři se sám prosím toho trápení zkoumat, jak hloupě to tam popisují oni.

P: (zklamaně) Áách ... A já jim tolik věřil!

A: Další úloha je taková zvláštní ... Jiná, nová. Tohle tam minulý rok taky nebylo.

P: (čte) „Tato úloha je určena pro celoroční přípravu řešitelů programátorských soutěží, jako je MO-P či IOI.“ To mi přijde docela fajn.

A: Ale jo, proč ne. Jak to mají zadané?

22-1-4 Bludiště — 15 bodů

Váš program na vstupu dostane bludiště $M \times N$, každé jeho políčko se skládá buď ze stěny, nebo volného místa. Do tohoto bludiště je vložen hráč, objekt a je také určená koncová pozice, do které má hráč objekt dostat.

Jak se v bludišti pohybuje? Hráč může volně popocházet na sousední volná políčka. Pokud před ním stojí bedna, za kterou je volné místo, může objekt odsunout na toto volné místo.

Vaším úkolem je najít nejmenší počet posunutí bedny, na který ho lze dostat na koncové místo, a nebo -1, pokud to nejde. Nezáleží tedy na počtu tahů hráče!

Formát vstupu: Na prvním řádku jsou čísla M a N udávající velikost bludiště, následuje M řádků, každý s N znaky. Ty mohou být:

- `#` – stěna
- `.` – volné políčko
- `H` – počáteční pozice hráče
- `O` – počáteční pozice objektu
- `P` – koncová pozice objektu

Znaky `H`, `O`, `P` se na vstupu vyskytují právě jednou.

Pokud tedy program dostane něco takového:

```
12 10
#####...##
#####.#.##
###P....##
#####.####
#.......##
#....O#H##
#.###.####
#.###.####
#.###.####
#.###.####
#.....####
##########
```

Má vypsat: `7`

Tato úloha má i své vlastní bodování: Pokud bude vaše řešení rychlé pro $M, N \leq 1000$, dostanete plných 15 bodů. Napadne-li vás program, který úlohu bude řešit rychle jen pro $M, N \leq 50$, budete ohodnoceni 7 body. Za chyby se bodíky strhávají jako obvykle.

P: Takže Sokoban!

A: Ano. Inu, proč ne. Hádám, že tím uvedeným obodováním se chtějí přiblížit podmínkám na soutěžích.

P: Asi jo. Bylas někdy na nějaké?

A: Ne. Víš, já mám matiku a programování ráda, ale vždycky jsem měla radši zvířátka a evoluční genetiku vůbec. Od září budu studovat na Přírodovědě.

P: Myslel jsem, že jsi úplně jasná matfyzačka!

A: Hm, jak to myslíš?

P: Jaké tam jsou další úlohy?

A: Ošklivé, nezajímavé. Jaks to myslel s tou matfyzačkou?

P: Jako že jsi chytrá. Hele, povídej mi třeba o nějakých minulých zadáních …

A: Najdi si je na webu. Chytré nejsou jenom matfyzačky, ale třeba i historičky. Tys to myslel jinak, viď?

P: Dobře, tak neznáš nějaké pěkné úlohy, které nebyly v KSPéčku? Hrozně rád bych je slyšel, jsem strašně žhavý do řešení programátorských problémů!

A: Tuhle jsem viděla jednu pěknou …

P: Ufff …

22-1-5 Šachovnice na pneumatice — 10 bodů

Existuje docela známá úloha naskládat N královen na šachovnici $N \times N$ tak, aby se podle normálních šachových pravidel vzájemně neohrožovaly.

Co když takové $N \times N$ veliké šachovnici slepíme levý okraj s pravým a horní s dolním, takže bude najednou jedna dáma ohrožovat jednak políčka, která viděla původně, ale i taková, která nyní vidí přes tyto slepené okraje?

Pro jaká N jde za takových podmínek naskládat N dam do šachovnice $N \times N$ tak, aby se neohrožovaly?

Pro $N = 2$ to kupříkladu určitě nejde: položíme-li dámu na kterékoliv ze čtyř políček, bude ohrožovat všechny další. Pro $N = 5$ to můžeme udělat takto:

```
D____
__D__
____D
_D___
___D_
```

A: Pravda, není to ani tolik programátorská, jako spíš logická úložka. Řešením není algoritmus, ale vzoreček nebo několik vzorečků, prostě popis takových N. Ale i takové, logické úlohy, v KSP bývají. Nebo spíš bývávaly.

P: Zajímavé. Rozumím tomu názvu: když vezmeš papír a slepíš ho tak, jak se tam popisuje, vznikne pneumatika. Jestlipak by se ta úloha dala řešit na ještě nějakém jiném povrchu? Möbiově pásce, Kleinově lahvi …

A: Nejspíš bys zjistil, že to o moc zajímavější není. Ale líbí se mi, jak nad tím přemýšlíš.

P: Už jsme v Pardubicích. Viď, že ještě někdy uvidíme?

A: Proč ne.

Vyměnili si sadu identifikátorů a rozešli se. Spolucestující zavřel knížku na stejné straně, na jaké ji otevřel. Rozhlédl se, to aby věděl, že je vzduch skutečně čistý. Vytáhnul z kapsy mobil, nalistoval vhodnou adresu a zmizel.

Kdo říká, že stroj času nemůže být serverová aplikace?

Objevil se o měsíc dřív ve velmi podobném vlaku a rychle odešel, protože už se nemohl chlubit platným jízdním dokladem.

Byl 5. červen a spolucestující si to pelášil na smluvené, tajné místo, kde už na něj čekal mistr Sazeč.

MS: Ahoj R ... Ehm, bratře Cestovateli!

BC: Ahoj. Mám to. :-)

MS: Fajn. Co s tím provedli?

BC: Byla to hrozná patlanina, posunuli to hodně. Vždyť víš, že jsme nemohli doufat, že už po téhle iteraci to bude použitelné.

MS: Tak jo. Máš všechny úlohy?

BC: Odstřelili jich víc, než zavedli nových, jedna teď chybí.

MS: No tak dáme třeba takovou tu náhradní ... Vždyť víš ...

22-1-6 Náhradní — 10 bodů

Vymyslete algoritmus, který když na vstupu dostane velký objem textu a několik klíčových slov, vyplivne nejkratší úsek textu, který obsahuje všechna zadaná klíčová slova, a nebo zahlásí, že se v něm nějaké z nich vůbec nevyskytuje.

Dostane-li kupříkladu algoritmus na vstupu text předchozího odstavce a dotaz na slova „klíč“ a „text“, vrátí „textu a několik klíč“. Dostane-li stejná klíčová slova k prohledání v tomto odstavci, najde řetězec „klíč” a ”text“.

Dostane-li v tomto odstavci vyhledat po slovech nasekanou první větu z Alenky v říši divů, nenajde slovo, které tu z taktických důvodů neuvedeme, ale jehož znění algoritmus vypsat má.

BC: Dobře. Tady máš ten diktafon ... Hodně štěstí při sepisování!

MS: Děkuju pěkně. Hezký zbytek zkouškového!

⋆ ⋆ ⋆

Byl jednou jeden vlak. 5. července 2009, kolem jedenácté hodiny dopolední, v Brně, právě vyjížděl z nádraží. Oplýval elektrickou lokomotivou a šesti vagónky.

Takový vlak jenom jeden určitě nebyl. Ve skutečnosti jich byla docela spousta a navzájem se lišily verzí papíru v třetím kupé druhého vagónu.

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.

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).

Vaším 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:

```
0 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.

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!

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.

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.“

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.

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.

22-3-2 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 táboritů 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:

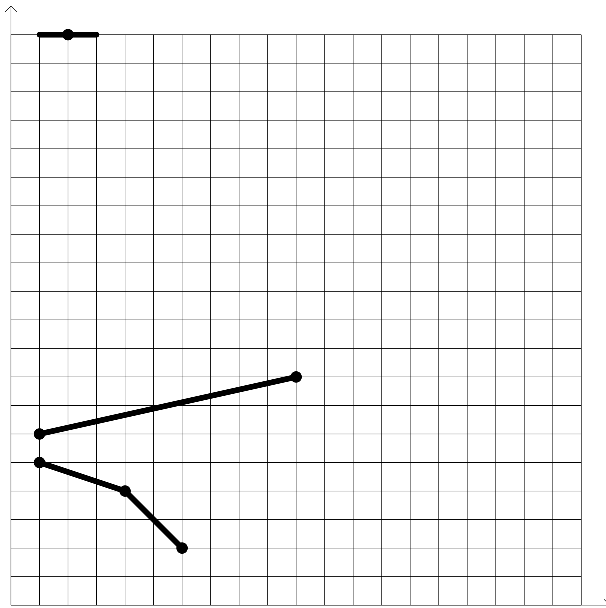
- max. 15 bodů: řešení rychlé při $1 \leq N \leq 100\,000$
- max. 12 bodů: řešení rychlé při $1 \leq N \leq 1\,000$
- max. 10 bodů: řešení rychlé při $1 \leq N \leq 100$

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.



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.

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

Tentokrát se pro změnu přesouváme do daleké budoucnosti. Lidstvo dávno vymřelo, poslední člověk zmizel roku 4291. Země je osídlena roboty. Příběh má na svědomí Nekra Zuzka Dortová.

Dobrý den,
jmenuju se `CPU7DR3X` *a jsem výzkumný robot. Mám docela štěstí, že jsem výzkumný robot. Víte, roboti se dělí do několika skupin: zásobovací roboti, výrobní roboti, výzkumní roboti, ... Zásobovací roboti se starají, abychom měli vždy dostatek energie. Výrobní roboti vyrábějí různé spotřebiče (a samozřejmě i další roboty). My výzkumní roboti vymýšlíme nové věci, zkoumáme planetu a její historii, zkrátka zajišťujeme přísun nových informací.*

Nedávno nám opět centrála rozdělovala práci. Obvykle každému přiřazuje, co má zkoumat (samozřejmě i s termínem, do kdy se očekává slušný výsledek), ale tentokrát na mě nevyšla ŽÁDNÁ práce!! Nevím, jak se to mohlo stát, možná byla chyba v systému, nicméně měl jsem volno a chystal se toho využít. Hodlal jsem udělat nějaký velký objev, tak velký, že ani centrála by takový úkol nepřidělila, neboť by ho považovala za velmi složitý.

Zajel jsem k encyklopedii (to je takový velký počítač, ve kterém jsou uloženy všechny dosud známé informace), napojil jsem se a hledal nápady na to, co bych mohl objevit.

22-4-1 Zaheslované stránky — 10 bodů

`CPU7DR3X` se chce dostat na N různých stránek s důležitými informacemi. Stránky jsou naneštěstí zaheslované, na některých stránkách jsou kromě informací ještě hesla k některým z dalších stránek. Aby se `CPU7DR3X` dostal na všechny stránky, potřebuje u některých prolomit bezpečnostní systém, a protože prolamování je nebezpečná činnost, chce ji provádět co možná nejméně. Úlohu mu ulehčila rada od kolegů trojských koní, kteří mu vyzradili, že ke každé stránce existuje heslo nejvýše na jedné jiné stránce.

Na vstupu dostanete seznam stránek očíslovaných $1 \ldots N$ a pro každou i informaci, ke kterým jiným stránkám je na ní uvedeno heslo. Napište program, který řekne, kolik nejméně stránek musí `CPU7DR3X` prolomit, aby si mohl přečíst informace na všech stránkách.

Strávil jsem nad tím několik hodin, ale pak jsem zjistil něco úžasného – kdysi prý na této planetě žily bytosti, které si říkaly lidé. Už podle obrázku vypadali divně, neměli žádná kolečka, žádné anténky, žádné senzory ani kamerky, ale údajně to byli jedni z nejvyspělejších historických živočichů, uměli vyrábět různé nástroje, derivovat, psát. Ale našel jsem o nich i záznamy o tom, jak poškozovali planetu – pozměnili ji natolik, že na ní už ani nedokázali žít. Bylo mi divné, že taková vyspělá zvířata jsou ohledně zachování vlastního života velmi

hloupá, a rozhodl jsem se, že zkusím takové zvíře zrekonstruovat. Tohle ještě nikdo nezkoušel – nejvyspělejší zvíře, které jsme dosud odchytili do laboratoře na pozorování, byl králík.

Nahrál jsem si do paměti všechny důležité informace a vyrazil jsem do terénu, abych získal podklady pro znovuoživení onoho divného zvířete. Potřeboval jsem sehnat DNA, neboť právě ona nesla veškeré informace o zvířeti.

A tak jsem se ještě toho dne vypravil hledat lidskou DNA. Napadlo mne, že nějaká by mohla být v muzeu. A tak jsem se vypravil k transportéru, který měl cestu kolem muzea, abych ušetřil pár hodin času. Během transportu jsem zaregistroval venku několik pracovních robotů hlasitě nadávajících na nezkontrolované zásilky zboží.

22-4-2 Rozvoz zásilek — 10 bodů

Transportér je jedno velké vozidlo, převážející zboží a součástky na jednosměrné, pravidelné trase. Trasa má několik zastávek – skladišť. Transportér zastavuje na všech zastávkách v pevném pořadí daném jejich čísly – naše měkké lidské mozky si mohou představit jednu autobusovou linku.

Zboží, které se průběžně ve skladištích nakládá a vykládá, může být buď v pořádku nebo poškozené, a aby se ušetřil čas (a pracovní roboti mohli dělat jiné užitečné věci), je v transportéru zboží skener, který mezi stanicemi zásilky zboží zkontroluje a označí je, zda jsou, nebo nejsou v pořádku. Organismy založené na uhlovodících si mohou představit jakéhosi hodného revizora, který nikoho z autobusu nevyhodí, jen vás označí. Na vaší výstupní stanici si už vás „přeberou“.

Naneštěstí ne vždy je skener plně nabitý, aby mohl kontrolovat zboží mezi všemi stanicemi. Určete, jak nastavit skener (rozhodněte, mezi kterými stanicemi se skener má aktivovat), aby zkontrolovaného zboží bylo v součtu co nejvíc, když víte, že skener je nabitý natolik, že je schopen kontrolovat zboží právě N-krát.

U každé stanice znáte počet kusů naloženého zboží a počet kusů zboží transportovaného do každé z následujících stanic (z tohoto nákladu). Pozor, každý kus zboží se započítává do součtu jen jednou, i když jste ho zkontrolovali během celé cesty třeba osmkrát.

Očíslujme stanice $1 \ldots S$, kde S je počet stanic. Například pro zadání (formát `počet kusů:odkud->kam`):

`4:1->4`, `2:1->2`, `6:2->3`, `3:2->4`, `5:3->4`; $N = 2$

je správným řešením dvojice úseků `2-3`, `3-4`.

Stěžování pracovních robotů mne natolik zaujalo, že jsem málem přehlédl, že už jsem u největšího biologického muzea, kam jsem se mohl během dne dostat. Největší problém bude se nenápadně dostat do muzea a opatřit si lidskou

DNA. Biologické materiály v muzeu jsou střeženy a označeny jen číselnými a písmennými kódy a pro jejich získání potřebují i výzkumní roboti povolení od centrály zadávání práce, že k pokusu biologický materiál skutečně potřebují. Povolení jsem neměl, a tak jsem se musel do muzea dostat potají, najít lidskou DNA a kus jí ukrást. To rozhodně nebylo jednoduché. Kdyby na mě přišli, mohli by mě i sešrotovat! Ale v zájmu vědy …

Utěšoval jsem se, že až dokončím rekonstrukci člověka, všichni budou jásat nad mým objevem. A tak jsem se vydal k zadnímu vchodu do muzea, kudy se do něj dováží různé spotřebiče a materiály. Jelikož dovoz je častý a skoro pořád tam jezdí pracovní roboti, není to tam moc hlídané. Schoval jsem se za nedalekou bednu a pozoroval jsem zadní vchod. Za několik hodin přivezl transportér zboží a krabice. Chvíli nato vyjeli ven pracovní roboti, zboží popadli a vezli ho dovnitř. Vyjel jsem ze svého úkrytu, sebral nejbližší krabici a zamíchán mezi pracovní roboty jsem pronikl do muzea. V muzeu jsem se chodbami dostal až do výstavních míst, kde byly biologické materiály. Problém byl, že jsem neznal kód, pod kterým zde byla uchována lidská DNA.

22-4-3 Muzeum — 10 bodů

V muzeu v biologickém oddělení se uchovává spousta biologických materiálů, které jsou tématicky roztříděny a každému tématu je vyhrazena jedna místnost. Místností je K a každá je hlídána přesně $(N-1)/K$ kamerami. (N je celkový počet kamer, přičemž $(N-1)$ je vždy dělitelné K.) Kamery jsou navzájem pospojované dráty. Navíc se v muzeu nachází centrální kamera na chodbě, z níž vede právě jeden kabel do každé místnosti, jenž je napojen na jednu z kamer. (Přes ni se pak dalšími spoji lze dostat k libovolné kameře v místnosti.) Mezi místnostmi jinak dráty vůbec nevedou.

`CPU7DR3X` se podařilo zjistit, jaké kamery jsou spojeny drátem. Tím získal souvislý graf oddělení (snad biologického), v němž kamery jsou vrcholy a dráty mezi nimi neorientované hrany. V tomto grafu potřebuje najít centrální kameru, tedy vrchol, z něhož vede právě jedna hrana do každé místnosti a jehož odebráním by se graf rozpadl na K komponent souvislosti. Bohužel neví, jaká kamera patří do které místnosti. Navíc si vůbec není jist, jestli se dostal do biologického oddělení, a tak zároveň potřebuje zjistit, zda je v každé místnosti $(N-1)/K$ kamer.

Pokud například bude 10 kamer (očíslovaných od 1 do 10), 3 místnosti a spoje mezi kamerami `1-5`, `2-7`, `2-4`, `1-10`, `7-1`, `4-6`, `3-8`, `3-7`, `2-6`, `8-9`, má centrální kamera číslo 7 a nacházíme se skutečně v biologickém oddělení (v první místnosti jsou kamery 1, 5, 10, v druhé 2, 4, 6 a v třetí 3, 8, 9). Ovšem pro spoje `1-5`, `2-7`, `2-4`, `1-10`, `7-1`, `4-6`, `3-8`, `3-7`, `2-6`, `2-9` má jedna místnost 4 kamery a jiná jen 2, takže máme graf jiného oddělení.

A tak jsem prozkoumával kamery a zjišťoval jejich počet, abych si ověřil, že jsem ve správném oddělení. Muzeum bylo obrovské, ale díky rychlému prozkoumávání kamer jsem našel správné oddělení už za 8 hodin. Zbývalo to nejdůležitější – pomalu a pracně jsem se naboural do centrálního monitoringu a vypnul jsem monitorování v celém oddělení. Pak jsem se rychle vloupal do několika vitrín, sebral kusy vzorků a prchal jsem, abych byl co nejrychleji z muzea, nejlépe ještě dřív, než přijdou na výpadek jednoho z centrálních monitoringů. Již po chvíli se strhl poplach a panika ... ani si nepamatuju, jak jsem se v tom zmatku dostal z muzea. V laboratoři jsem pak vzorky pořádně prozkoumal a zjistil, že jeden z nich je opravdu lidská DNA!

Konečně jsem ho měl. Zbývalo jen to hlavní – probudit z něj k životu ono podivné zvíře. A tak jsem zajel do laboratoře. Věděl jsem, že k oživení zárodku bytosti je třeba DNA probudit k životu, ale protože u takového vyspělého zvířete to bude jistě náročné, rozhodl jsem si DNA namnožit – tak s ní budu moct dělat pokusy a vždy mi nějaká zbyde pro ty další.

A tak jsem spoustu dalších týdnů strávil tím, že jsem se snažil probudit lidskou DNA k životu. Přidával jsem k ní všechno možné, píchal do ní injekce s rozličnými látkami, pracoval s ní v různých teplotách, vlhkostech i tlacích, ale stále bezvýsledně. Teprve po několika dalších týdnech jsem v jedné ze zkumavek zaregistroval něco, co připomínalo lidský zárodek. Izoloval jsem zárodek do skleněné nádoby, vložil ho do 3D mikroskopu s největším přiblížením (jaké bylo v budově dostupné) a pořádně ho prohlédl. Nebylo pochyb – byl to skutečně lidský zárodek. Okamžitě jsem jel vložit zárodek do urychlovače – naneštěstí se přístroj zasekl a zárodek se zničil. Prohlédl jsem svoje záznamy a okamžitě jsem začal pracovat na tvorbě dalších zárodků. Zárodky jsem střídavě mutoval a ořezával, aby mi vzniklo to správné zvíře.

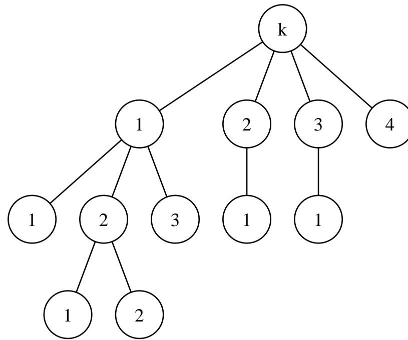
22-4-4 Ořez zárodků 10 bodů

`CPU7DR3X` dává zárodky v různém stupni mutace na ořezání špatných větví. Zárodek je různě rozvětvený, navíc je na něm poznamenáno místo, odkud větvení začíná.

Kdybychom měli špatný mikroskop, viděli bychom vlastně strom (souvislý graf bez kružnic) s pevně daným kořenem. Tyto stromy mají navíc jasné uspořádání synů, takže podívám-li se na některý vrchol, tak umím vždy jasně říci, který syn je první, který je druhý, a tak dále.

Není-li vám jasné, co takový strom je, podívejte se do naší Kuchařky o grafech.[2] Ale hlubokou algebru v tom nehledejte, obrázek popisuje situaci více než dobře.

[2] Kuchařka o grafech: *http://ksp.mff.cuni.cz/tasks/20/cook3.html*



Příklad: původní strom

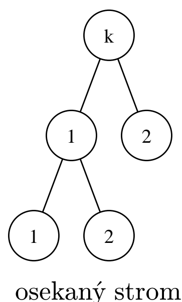
Nyní je třeba zárodek ořezat. To se udělá tak, že se nejprve vybere rozbočení (vrchol, například první syn kořene z přikladu) a v tomto místě se ještě zvolí souvislý interval synů (např. druhý až třetí syn). Původně vybrané rozbočení se prohlásí za kořen, synové kořene budou jen ty vrcholy, které byly jeho syny ve vybraném intervalu, a uspořádání se zachová (druhý syn bude prvním synem nového kořene). Spolu s tímto intervalem patří do ořezaného stromu také celé podstromy pod těmito syny. Vrcholy, které neležely v příslušném intervalu nebo byly jinde v původním stromě, v novém stromě prostě nebudou.

Naneštěstí je přístroj na ořezávání nedokonalý, takže občas oseká strom ne přesně tak, jak jsme popsali v odstavci výše. Navíc `CPU7DR3X` nemá v záznamech úplně pořádek a proto v nich má počáteční zárodek a ořezaný zárodek zaznamenány v nahodilém pořadí (tj. nelze předpokládat, že neořezaný je ten první). Od toho jste tu vy.

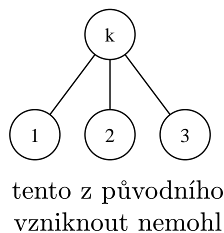
Na vstupu dostanete dva zárodky (zakořeněné stromy) a máte zjistit, jestli jeden mohl vzniknout ořezem druhého.

Stromy mohou být zadány například takto: vrcholy si očíslujeme od 1 do N, kořenem bude vrchol s číslem 1 a na vstupu dostaneme pole spojových seznamů, kde I-tý prvek pole je spojový seznam, který obsahuje číselná označení synů I-tého vrcholu, uspořádána zleva doprava. V této reprezentaci můžeme zadat „osekaný strom“ z obrázku níže například takto:

```
1: 2 3
2: 4 5
3:
4:
5:
```



osekaný strom



tento z původního vzniknout nemohl

(Psst! Zaslechl jsem organizátory si šuškat o tom, že některé úlohy tohoto ročníku se dají vyřešit pomocí přiložené kuchařky o řetězcích. Tahle to ale asi nebude, že? To by bylo ujeté. . . váš Štek Tiskařský, v.r.)

A tak jsem dával opakovaně do urychlovače zárodky, kontroloval, zda se dobře vyvíjejí, a s napětím čekal, kdy se mi konečně vyvine kompletní zvíře. Až jednou . . . Hurá, konečně se to povedlo! Po otevření urychlovače se v něm hýbal nějaký bledý tvor. Zvíře bylo asi metr dlouhé a obrázku ze záznamu se podobalo jen velmi hrubě. Položil jsem zvíře na podlahu. Vstalo a postavilo se na dvě úzké tyčky s klouby. (Tyto tyčky tento druh zvířat prý nazýval „nohy“.) Otočil jsem se na zvíře, připravil jsem si počítač pro záznam a zadal zvířeti jednoduchou otázku:

„Kolik je desátá derivace z $x^{20} + 4x^{18} - e^{x^{14}} \cdot x^7 - x^{13\,\mathrm{tg}\,x} - 6$*?“ Zvíře se ke mně otočilo a nepatrně se mu rozšířily hnědé kuličky na vrchní kulaté části (těmhle věcem říkala ta zvířata oči). Udělal jsem si do počítače záznam, že zvíře derivovat neumí. Popadl jsem zvíře (samozřejmě se vzpouzelo a vydávalo různé otravné zvuky) a zavezl jsem je do skladu součástek. Tam jsem zvíře pustil a (zatímco se rozhlíželo kolem) jsem mu zadal jiný jednoduchý úkol: „Vyrob solární panel!“ Zvíře se na mne notnou dobu dívalo, a pak se slabým hláskem zeptalo: „Co je to solární panel?“ „Ty nevíš, co je solární panel?“ rozkřikl jsem se a udělal záznam o tom, jak je zvíře neuvěřitelně hloupé.*

Takovéhle bytosti že prý byly nejvyspělejší? To to tu bylo před desítkami tisíc let velmi zaostalé! Prohledal jsem svou paměť, abych si ověřil, že zvíře není příliš staré – stará zvířata byla podle záznamů mnohem blbější než mladá. Zvířata začínala podle záznamů blbnout stářím asi kolem 70 let. Otočil jsem se na zvíře a zeptal jsem se: „Kolik je ti let?“ Zvíře odpovědělo: „Je mi pět let.“

Tady něco nehrálo. Zvíře nebylo staré, a přesto nefungovalo tak, jak by podle záznamů fungovat mělo. Po projití dat v paměti jsem si připomněl, že příliš mladá zvířata toho také moc neumí. Přes protesty a pohyby zvířete jsem je popadnul a odvezl je do urychlovače. Potřeboval jsem, aby bylo o něco starší. Dal jsem zvíře do urychlovače, zaklapl víko a přes jeho vřeštění jsem přístroj zapnul. Řev slábnul a když na přístroji zasvítily kontrolky, otevřel jsem urych-

lovač. Zvíře se ohnulo v půlce, slezlo z urychlovače a já mu opět položil otázku: „Kolik ti je let?“

Tentokrát zvíře odpovědělo: „Je mi třiadvacet …“ Zadal jsem mu opět jednoduchý příklad na derivování. Tentokrát zvíře chtělo něco na psaní. Ukázal jsem mu zastaralý model počítače, který se často sekal (přeci jen, nemohu riskovat, že by zvíře zničilo nějaký drahý přístroj), zvíře se uvelebilo před obrazovkou, zmáčklo tlačítko, ale starý počítač se nenastartoval. Co to? Po bližším zkoumání se ukázalo, že v počítači nebyl energetický článek. Otevřel jsem zásobník článků a začal uvažovat, který dát do počítače.

22-4-5 Energetické články 10 bodů

`CPU7DR3X` otevřel zásobník, ve kterém je N různých druhů energetických článků. Každý článek se vyznačuje tím, že je souměrný vzhledem k tomu, z jakých částic se skládá. (Např. `RAR` je energetický článek, `RAN` není energetický článek. My neroboti říkáme, že článek je palindromem.) Články lze spojovat, ale jen tehdy, pokud vzniklý článek je opět souměrný. `CPU7DR3X` chce vložit do počítače článek složený ze dvou článků a zajímá ho, z kolika různých příslušných uspořádaných dvojic článků si může vybrat.

Na vstupu dostanete N různých článků, zadaných slovním schématem (N slov, palindromů). Vaším úkolem je napsat program, který spočítá, kolik uspořádaných dvojic (dvojici přečteme jako jedno slovo zleva doprava) opět tvoří energetický článek (palindrom).

Například dvojice (`RAR`, `ARA`) palindrom netvoří, ale dvojice (`BAB`, `BABBAB`) palindrom tvoří.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému Codex.

Po úspěšném zapojení počítače začlo zvíře konečně počítat příklad. Přitom se mnou mluvilo a kladlo spoustu otázek – zjevně se mu okolní prostředí líbilo a zajímalo ho. Udělal jsem si záznam, že ve 23 letech zvíře derivovat umí, a právě jsem si chtěl zapsat, jak dlouho mu to trvá, když se rozletěly dveře a dovnitř vjelo a vlétlo několik masivních bezpečnostních robotů. Někteří mne popadli a někam mne odváželi (a dočasně mi vyzkratovali kamerky), poslední, co jsem zahlédl, bylo, že ostatní se vrhli na zrekonstruované pokusné zvíře.

Když jsem konečně mohl vnímat obraz, byl jsem znehybněn uprostřed malé místnosti a okolo mne bylo asi osm robotů, kteří na mne přísně shlíželi. Jeden z nich popojel ke mně a řekl: „`CPU7DR3X`*, toto je tvůj soud za ohrožení bezpečnosti všech robotů!!“ Zeptal jsem se, co jsem provedl. A ten robot popojel ještě o kousíček blíž, až jsem se bál, že do mne naboură, a zahřímal: „Cos provedl?! Ohrozil jsi celou naši existenci obnovením velmi nebezpečného druhu!“ Nechápal jsem, jak by nám jeden člověk mohl být nebezpečný, a obhajoval jsem se,*

že jsem učinil významný objev, ale nic mi nevysvětlovali a odešli do vedlejší místnosti se radit, jak mne potrestají.

Přemýšlel jsem o tom, jak se dozvěděli, že jsem vyrobil člověka, a vzpomněl jsem si, že mezi roboty již dlouho kolují nepodložené informace, že všechny činnosti robotů pozoruje jakési Tajné Bratrstvo, které má všude rozmístěny skryté kamery (a které tudíž muselo hned zaznamenat, že jsem odpojil centrální monitoring v muzeu). Nikdy jsem těmto zkazkům nevěřil – až dnes jsem se přesvědčil, že jsou pravdivé.

Když se Bratrstvo vrátilo, oznámil mi jeden z nich, že s konečnou platností přestávám být výzkumným robotem a přeřazuji se mezi roboty pracovní. Pak mne převezli do laboratoře, kde mi z paměti vymazali všechna hesla a přístupové kódy do místností sloužících k výzkumu, sběru informací, prostě do místností informačního a výzkumného charakteru. A tak jsem teď jen pracovním robotem. Pracovním robotem té nejnižší kategorie.

22-4-6 Umisťování panelů 13 bodů

`CPU7DR3X` byl přeřazen mezi nejnedůvěryhodnější pracovní roboty. Nyní místo výzkumu bude s ještě jedním pracovním robotem umisťovat na louku panely pro příjem větrné a sluneční energie. Louka je rozdělena pomocí kolíků na čtvercovou síť (kolíky jsou v rozích čtverců), v některých čtvercích mají být umístěny panely (jeden nebo i více). Na louce má být celkem umístěno N panelů. Na louce budou stavět panely dva nedůvěryhodní roboti, z nichž každý má postavit K panelů. A protože ti dva pracovní roboti jsou nedůvěryhodní, je potřeba v rámci zabezpečení nalézt pro každého z nich obdélníkový prostor, kde má být celkem postaveno K panelů, a tyto prostory ohraničit z kolíků bezpečnostními paprsky. Jelikož na paprsky je potřeba energie v závislosti na délce paprsku (kterou se pokud možno šetří), má součet obvodů vytyčených pracovních míst být co nejmenší. Oba pracovní prostory se navíc nesmějí překrývat a pokud někde mají společnou hranu, je stejně potřeba v tomto místě vést paprsek pro každou pracovní plochu zvlášť (tj. v takové hraně budou dva paprsky).

Na vstupu dostanete na prvním řádku rozměry louky (délka a šířka), na dalším čísla N a K, a na dalších N řádcích dostanete souřadnice pro umístění jednotlivých panelů (délka a šířka) – na každém řádku jeden. Napište program, který vypíše minimální počet jednotek (jednotka – mezi dvěma kolíky) paprsků, které jsou potřeba k ohraničení dvou disjunktních obdélníkových pracovišť (z nichž na každém má být dle plánu K panelů) nebo vypíše, že pro příslušná data řešení neexistuje.

Bodování:

- max. 13 bodů za řešení rychlé při $N, K \leq 300$,
- max. 10 bodů za řešení rychlé při $N, K \leq 50$.

Nevím, co udělali s člověkem, ale nejspíš ho rozebrali. To je mi moc líto, takový velký objev jsem učinil a nikdo to neocení. Když mě propouštěli ze soudní místnosti, jeden z nich mi dokonce pošeptal „Máš štěstí, moh' jsi skončit ve šrotu …“

Blíží se konec 22. ročníku, ale ještě než odjedete k moři, do hor, na vandr či na jinou skvělou letní akci, přilétá k vám na křídlech větru poslední série sepsaná na starý pergamen Pavlem „Pauliem“ Veselým.

„Je rozhodnuto, vyrazíš hned,“ zahřměl otec a skály vše zopakovaly ozvěnou.

„Ale …“

„Žádné ale!“ zaburácel, až se země zatřásla.

„…když já jsem ještě moc malý,“ pípl jsem. „Tys byl určitě větší, než jsi odletěl.“

„Řekla to veliká pramáti a té nelze odporovat!“

„Jenže ona je netrpělivá a sklerotická. Chce mít všechno hned a zapomněla, že jsem příliš mladý.“

„Neurážej pramáti! A teď upaluj pro princeznu.“

Jestli si myslíte, že draci mají život lehký, když umí chrlit oheň, přetrvají věky a jsou velcí, šeredně se pletete. Téměř nikdo nás sice neohrožuje, až na lidi, a o ty právě jde. Jsou sice drobní, jenže je jich jak mravenců. A tak se musíme skrývat v pustých horách mezi skalami, lovit zvěř a doufat, že na nás nepřijde armáda lučištníků.

Z bezpečí mezi štíty, tyčícími se vysoko do nebe, vycházíme jen z jediného důvodu: ukrást si princeznu, v případě dračic samozřejmě prince. Pramáti pak člověka očaruje tak, aby věrně sloužil svému drakovi. Princ či princezna je vlastně celkem užitečná věc, jelikož má šikovné prťavé ručičky, což nám drakům chybí. Navíc se jedná o tradici a zároveň rituál vstupu mezi velké draky.

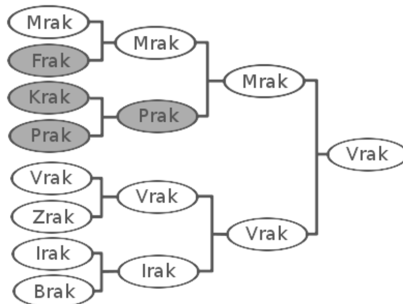
22-5-1 Turnaj — 10 bodů

Ještě před odletem jsem musel dohrát turnaj v ohnivém zápase, což je bezkontaktní souboj, v němž se postaví dva draci proti sobě a chrlí na sebe oheň, dokud jednomu nezačne hořet tráva nandaná na čenich.

Už před pár dny jsme sehráli celého pavouka vyřazovacího turnaje (dva draci se utkají, do dalšího kola postupuje jen jeden, jenž se utká s vítězem jiného souboje …), takže známe a obdivujeme vítěze, jenže se neumíme dohodnout, kdo je na druhém místě. Občas totiž nelze určit, jak by dopadl souboj dvou draků, když se spolu neutkali, nemají společného soupeře, ani jejich soupeři spolu nebojovali a nemají společného protivníka, s nímž se utkali … Platí však alespoň tranzitivita: jestli drak A porazil draka B a drak B draka C, porazil by i drak A draka C.

Potřebujeme zjistit, jací jsou kandidáti na druhé místo, kolik jich je a jak mezi nimi co nejefektivněji vybrat, tedy kolik zápasů je třeba navíc ještě sehrát. Tato úloha je logická, takže nemusíte psát program (draci stejně nedisponují

počítači), stačí popsat a zdůvodnit postup určení druhého nejlepšího draka, který by určitě porazil všechny ostatní kromě prvního.



Příklad pavouka hry. Šedě jsou zvýrazněni ti, které by určitě porazil drak Mrak.

Po zdárném obhájení druhého místa jsem vyrazil. Tížil mě jen pocit, že nevím, jestli za tohle dobrodružství nějaká princezna stojí. Ta poslední, již na svých zádech před 100 lety přitáhl bratranec, byla podle otce nějaká podivná. Možná si jenom na princeznu hrála. Uvidíme, jakou seženu teď.

22-5-2 Stráže údolí — 10 bodů

Nejprve jsem ovšem musel prosvištět kolem stráží hlídajících naše údolí. Dělají nám je havrani rozmístění poblíž ústí do jiného údolí tak, aby byli na přímce. Jenže někteří jsou moc blízko u sebe a mají tendenci se místo hlídání vybavovat, takže jsme se rozhodli počet stráží zredukovat. Nevíme však, které propustit.

Známe polohu všech N havranů na přímce, zadanou celočíselnými mezerami mezi nimi, a chceme jich vyřadit K, aby byli dva nejbližší havrani od sebe co nejdále. Potřebujeme tedy maximalizovat minimální vzdálenost mezi nimi. Dokážete pro nás rychle najít K havranů, jež pošleme do výslužby?

Například pro $N = 6$, $K = 3$ a mezery mezi havrany `4, 6, 2, 5, 7` je správným řešení propustit 2., 3. a 5. havrana (bráno zleva), takže zůstanou mezery `12, 12`. Pro $N = 14$, $K = 7$, mezery `5, 12, 6, 3, 8, 1, 4, 1, 1, 9, 15, 1, 16` vyhodíme 2., 4., 6., 7., 8., 9. a 12. (možností je tentokrát více).

Hned, jak jsem přeletěl lesy, mi něco nepřišlo úplně v pořádku. Žádný drak se totiž nikdy ve svém vyprávění nezmínil o dlouhých černých liánách vedoucích mezi roztodivnými stromy s mnohými tenkými kmeny. Raději jsem je nadletěl.

Jakmile jsem uviděl první lidské osídlení, vlétl jsem do něj jako bouře. Tedy, za pár desítek let už budu dost velký, abych tam vletěl jako bouře. Nicméně jsem

způsobil nemalé pozdvižení. Několik lidí sedících u stolu venku všechno převrhlo a dalo se na zmatený únik do domu, jedna ženská zavřeštěla „Zavolejte policií!“ (kdo je to k čertu ta policie?) a mladý pár jdoucí po ulici se pokusil vzít přede mnou nohy na ramena.

Rozhodl jsem se dohonit prchající pár, chytl do svých spárů mladíka a pak ho shodil na zem, aby mi neutekl. Jekot ostatních lidí nabíral na intenzitě.

„Kde najdu krále?“ zařval jsem lidským jazykem.

Kluk měl v obličeji barvu vápence, chvíli vypadal, že strachy zapomněl mluvit. Poté, co jsem se pochlubil malým plamínkem, dokázal ze sebe vyloudit „Co ... Cože? My nemáme krále.“

„Nelži! Kde přebývá princezna?“ Nenechám se přece oblafnout nějakým zbabělcem.

Mladík na mě ještě okamžik zíral a pak ukázal na sever. „Tam ... Támhle. Velké město.“

Ještě jsem mu předvedl, že pro mě není problém zapálit celou vesnici, a rozletěl jsem se.

Než jsem dorazil k tomu městu, setmělo se. Lidé se odjakživa báli tmy, ale netušil jsem, že kvůli tomu rozsvěcují tolik světel. Dokonce i po cestách se pohybovaly zářivé body, asi už dávají louče i na své koně. Mě však víc zaujala zvláštní síť ulic.

22-5-3 Zrcadla — 10 bodů

V téhle okrajové části města byly dost podivné cesty. Tvořily totiž rozlehlou čtvercovou síť, ale domy byly postaveny jen někde. V jednom místě se nacházel velmi silný světelný zdroj záhadného původu svítící pouze jedním směrem.

Vzpomněl jsem si, jak mi strýc vyprávěl o zrcadlech, a napadlo mě, jestli by se nedala využít spolu se zdrojem silného záření na osvětlení nějakého konkrétního domu.

Je tedy dána čtvercová síť o rozměrech $M \times N$, v níž se nachází K domů a jeden další, na který chceme z libovolné strany dosvítit. V jednom bodě máme světelný zdroj, který může svítit pouze vertikálně nebo horizontálně (ve smyslu čtvercové sítě), přičemž směr si můžeme vybrat.

Ptáme se, jestli lze do čtvercové sítě rozmístit na políčka zrcadla tak, aby byl jeden daný dům osvícen, a pokud ano, kolik nejméně jich je potřeba. Domy jsou neprůsvitné a zrcadla umísťována do políček diagonálně (světlo se tedy šíří po čtvercové síti vždy jen horizontálně nebo vertikálně).

Pokud například máme čtvercovou síť o rozměrech 3×3, světlo na souřadnicích `[2, 1]`, dům, jenž chceme osvítit, na `[2, 3]` a další domy na `[1, 2]` a `[2, 2]`, je správným řešením, že stačí umístit dvě zrcadla (na políčka `[3, 1]` a `[3, 3]`).

Stačila chvíle a vlétl jsem přímo nad moře světel. A jak byla některá silná! To musí být ale oheň, který dokáže takhle zářit.

A ta obydlí … Mnohdy byla velmi vysoká, až se mi nechtělo letět nad ně.

I když je noc, můžou mě takhle klidně zahlédnout, říkal jsem si. Obzvlášť v takovém velkém množství, v jakém jsou dole na náměstí.

22-5-4 Davy lidí 12 bodů

Jak jsem se díval na stovky lidiček pod sebou, přišlo mi, že část věnuje pozornost soše a druhá část fontáně. Navíc spousta z nich měla za objektem svého zájmu jiného člověka tak, že socha či fontána ležela ve středu úsečky tvořené těmito lidmi.

Mě by zajímalo, jestli se dá dav N lidí, u nichž známe jejich souřadnice v rovině, rozdělit na dvě části, které jsou obě středově souměrné. Střed souměrnosti jedné části tvoří fontána a střed druhé socha, ale jejich souřadnice nejsou zadané. Někdy se čirou náhodou na fontáně či na soše může vyskytovat člověk. Z pohledu draka jsou všichni lidé stejně body, takže jejich rozměry zanedbejte.

Například pro 8 lidí se souřadnicemi `[6, 2]`, `[6, 4]`, `[1, 3]`, `[-4, -2]`, `[-4, 2]`, `[1, -3]`, `[6, -2]`, `[6, -4]` rozdělení na dvě středově souměrné části existuje (1., 3., 4., 5., 6. a 8. člověk v jedné části, 2. a 7. v druhé), ale 6 lidí a souřadnice `[4, 0]`, `[0, 4]`, `[0, 2]`, `[4, 4]`, `[2, 3]`, `[2, 0]` už rozdělit nelze.

Let nad městem mě už začínal docela unavovat, když jsem konečně uviděl hrad, nebo alespoň něco jemu podobného. Každopádně jsem žádné lepší místo, kde hledat princeznu, v okolí neviděl.

S žuchnutím jsem přistál na dlažbě a protáhl si křídla, ve vzduchu jsem byl přeci jenom docela dlouho. Na nádvoří nikdo nebyl, takže jsem prošel branou na další. A tam jsem ji spatřil.

Seděla na lavičce sklopená nad něčím hnědým rozlámaným na kousky a tvářila se nešťastně. Na sobě měla dlouhý černý kabát, vysoké černé lesklé boty, což ladilo k jejím rozpuštěným vlasům barvy temné noci s jediným červeným pruhem.

Sice prý princezny běžně vypadají jinak, ale za 100 let se lidská móda může radikálně změnit. Popošel jsem blíže a oslovil ji.

„Milá princezno, mohu vám s něčím pomoci?“ zeptal jsem se, jak nejgalantněji jsem uměl.

V odpověď jsem dostal škytnutí a smutný pohled. Pak vstala, popošla pár kroků směrem ke mně, ale trochu se motala, takže se jí rozsypaly ty hnědé věci po zemi. Zjevně jí to však nevadilo.

„Potřebuju vyřešit jeden příklad,“ říkala ztěžka. „Na zítra do školy. Jinak ten matfyz neudělám.“

„A když ti pomůžu, poletíš na mých zádech ke mně domů?“

Podívala se na mě a zasmála se. „Jasně. Mimochodem, hustej převlek.“
„Tak povídej.“

22-5-5 Čokolámání — 10 bodů

Princezna dostala něco, čemu se říká čokoláda. Má to obdélníkový tvar a $M \times N$ dílků, které se dají lámat podle os mezi nimi. Ke každé této ose (horizontální i vertikální) měla napsané číslo, totiž cenu zlomu.

Její úkol spočíval v nalezení postupu, jak co nejrychleji zjistit, za jakou nejnižší cenu lze rozlámat čokoládu na jednotlivé dílky. Cena každého zlomu může být započítána vícekrát bez závislosti na délce zlomu. Například rozdělíme-li čokoládu na řádky a ty pak budeme lámat, započítá se cena každého vertikálního zlomu M krát, kdežto každého horizontálního jen jednou.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému Codex.

Chviličku mi úloha zabrala, ale už jsem viděl mnohem těžší. Princeznu moje řešení zjevně nadchlo, vyskočila radostí a pak bez váhání vylezla na můj hřbet.

„Jakého království jsi ty vlastně princezna?“ osmělil jsem se zeptat, když přestala nadšeně vykřikovat nad úžasnou podívanou na noční Prahu, jak město nazvala.

„Já jsem princezna … princezna … matfyzu! A tohle celé město patří jenom nám! Podívej, támhle v těch vysokých zelených budovách sídlíme.“

Zadíval jsem se na novodobé zelené hrady a přemýšlel, jak radikálně se změnila architektura, když tu náhle se vpravo ozval hukot. Po pár sekundách i za mnou. A vlevo taky.

„Leť, ty můj draku, leť!“ křičela z plna hrdla moje princezna. Strach v jejím hlase jsem však nepostřehl.

Ohlédl jsem se a uviděl spoustu zvláštních ptáků bez křídel. Všichni vyluzovali monotónní hluk. Že by princezna měla až takovouhle ochranu proti drakům?

22-5-6 Hlídači princezny — 13 bodů

Princezna je v království vždy velmi ceněna, a tak má spoustu bodyguardů, strážců a hlídačů. Kdyby se všichni kryli navzájem, vznikl by akorát zmatek, takže každý hlídač kryje právě jednoho jiného.

Do útoku proti nepříteli se vždy posílá co nejvíce hlídačů, ale každý z nich musí být kryt někým, kdo nejde do útoku. Jak najít takové hlídače?

Úlohu si můžete představit jako orientovaný graf, kde z každého vrcholu vede jen jedna hrana (do něj však může vést více hran, nebo žádná). Chceme najít největší množinu vrcholů takovou, že do každého vrcholu z této množiny vede alespoň jedna hrana z vrcholu mimo tuto množinu.

Pokud máme 5 hlídačů (očíslovaných od 1 do 5) a `1 ->` (hlídá) `3`, `2 -> 3`, `3 -> 4`, `4 -> 5` a `5 -> 3`, pak vybereme do útoku hlídače 3 a 5.

„Vyzýváme vás, abyste okamžitě sletěl dolů, jinak bude zahájena varovná palba,“ ozvalo se silným, avšak podivně plochým hlasem zezadu.

„Vyprdni se na ně a ukaž jim, zač je toho loket!“ dodala mi odvahu moje princezna.

„Tak se pořádně drž!“

Prudce jsem slétl dolů. Přímo mezi domy. Neztrácel jsem čas obdivováním výšky zdejších budov a rozletěl se ulicí. Za mnou se však ozval svist hlídačů. Ihned jsem zabočil a uslyšel obrovskou ránu. Jednomu se manévr nepodařil.

Máchal jsem křídly jako nikdy v životě, jenže oni byli pořád za mnou. Drželi se jak klíšťata. Ještě jsem neslyšel, že by něco tak velkého dokázalo stíhat draka.

„Prásk! Prásk! Prásk!“ ozvalo se a já pocítil bolest na levém křídle. Provedl jsem bleskový úhyb.

„Kličkuj!“ zakřičela princezna zděšeně z hřbetu.

Nebylo třeba mě pobízet. O lučištnících jsem slyšel. Jenže tihle mají sakra rychlé šípy.

Rychle jsem zabočil do jedné úzké uličky. „Schovej se támhle,“ zařvala ihned. Neváhal jsem a vletěl do velkých otevřených vrat, až jsem se bouchl jedním křídlem. Po pár sekundách kolem s hukotem prolétlo pár princezniných strážců.

„Tak tady chvíli zůstaneme a pak rychle vyletíme.“

Byl jsem tak zadýchaný, že jsem jenom zakýval hlavou. Na vymýšlení lepšího plánu nebyl čas.

Po několika minutách odpočinku jsme se odvážili vyjít z úkrytu. Ihned jsem vyletěl, jak nejvýše to šlo. Naštěstí nás nezpozorovali, takže jsme se mohli v klidu vzdálit od města a tam si odpočinout. I ona toho měla dost, hned usnula pod mým ocasem suplujícím peřinu.

Další den jsme už naštěstí bez nepříjemných příhod dolétli domů. S princeznou jsem nyní velmi spokojen, je chytrá, akční a šikovná. Taková dračice …

Seriál: Erlang

Michal „vorner“ Vaner

22-1-7 Když telefony pekly jazyk — 12 bodů

„Ale né, už zase!“ – tak takto bude reagovat nebohý agent, který tupě zírá do mobilu, protože spadl server stroje času.

„Ale né, už zase!“ – tak takto bude reagovat nebohý programátor, který je nucený mistrem Motivátorem napsat software na server tak, aby mohl běžet na více počítačích, a pokud některý z nich odnese povodeň, ostatní ho zvládly zastoupit.

V této a příští sérii se podíváme na zoubek jazyku, který takovým programátorům dokáže ulehčit život – na Erlang, který je dělaný na snadné spouštění mnoha vláken na více počítačích. Na zvýšení výkonu dokoupením dalších počítačů sice není nejlepší (je drobátko líný), ale na zvýšení spolehlivosti je (jako) dělaný.

Napřed si zkusíme jednoduchý příklad. Nainstalujte si Erlang, který je na adrese *http://erlang.org/download.html* a spusťte. Měla by se objevit jeho příkazová řádka. Pokud po napsání `2*21.` odpoví odpovědí na otázku života, vesmíru a vůbec, tak je vše v pořádku. Všimněte si té tečky na konci, ta ukončuje výraz a říká, že bychom rádi získali odpověď.

Podívejme se na to, jak se programy v Erlangu zapisují. Erlang pochází z rodiny funkcionálních jazyků, základním stavebním kamenem jsou tedy funkce, nikoliv příkazy, jako tomu je například u Pascalu. A protože bychom raději trvanlivější programy, které není potřeba pokaždé přímo zadávat, budeme je psát do souboru.

Zatímco klasická ukázka procedurálních jazyků je program „Hello world“, funkcionální klasikou je faktoriál. Uložme tento kód do souboru `factorial.erl`. (Bez tečky na konci, ta ukončuje větu.)

```
-module(factorial).
-export([fac/1]).

fac(0) -> 1; % Konec cyklu
fac(N) -> N * fac(N - 1). % Ještě jednou, prosím
```

Na začátku vidíme direktivu `module`, která dává jméno aktuálnímu modulu (takové knihovničky funkcí), a direktivu `export`, která říká, které funkce mají být vidět zvenčí. Ona `1` za lomítkem udává, kolika parametrovou verzi dané funkce máme na mysli (funkce se rozlišuje podle jména a počtu parametrů).

Část s vlastní funkcí je již zajímavější. První věc, která stojí za povšimnutí – jména proměnných začínají velkým písmenem. Tímto způsobem jazyk rozlišuje

proměnné od ostatních věcí, jako jsou funkce, atomy (k nim později) a klíčová slova.

Proměnné v Erlangu jsou trochu jiné, než v „běžných“ programovacích jazycích. Nepředstavují ani tak pojmenované místo v paměti, jsou to spíš matematické definice. To znamená, že do jedné proměnné se za dobu jejího života může přiřadit jen jednou (v době jejího vzniku).

Druhou zajímavou věcí je to, že funkce má dvě „verze“ – jednu pro případ, kdy je parametrem nula, druhou, když je to cokoliv jiného. Při zavolání funkce Erlang vezme první verzi, zkusí jestli do ní parametry „pasují“, pokud ano, zavolá tu (a tím skončí), pokud ne, zkouší dále. Každá verze je ukončená středníkem, poslední tečkou.

Třetí věcí jsou komentáře. Cokoli je za znakem `%` až do konce řádku, je ignorováno. Stejně tak jsou ignorovány všechny bílé znaky (mezery, nové řádky, tabulátory), ty jen od sebe oddělují tokeny.

A poslední věc, která stojí za zmínku, funkce volá sama sebe. To je povolené a v Erlangu je to jediný způsob, jak akce opakovat, Erlang nemá cykly jako procedurální jazyky. Pro zkušenější, Erlang dělá takzvanou ocasní optimalizaci[3], takže pokud jako poslední věc ve funkci je volána jiná funkce, aktuální se ze zásobníku odstraní. Toto umožní napsat nekonečný „cyklus“ bez strachu o přetečení zásobníku.

Nyní zkusíme tento modul použít, opět z Erlangovské příkazové řádky.

```
1> c(factorial).
{ok,factorial}
2> factorial:fac(3).
6
```

Funkce `c` zkompiluje a zpřístupní modul daného jména. Pokud nastane chyba (například chybí někde tečka), tak vypíše hlášku, která by měla pomoci s jejím nalezením.

Pokud voláme funkci ze stejného modulu, stačí napsat jméno funkce. Pokud ale voláme nějakou „cizí“ funkci, je potřeba před dvojtečku zadat i modul původu.

Další věc, která se hodí, jsou nějaké podmínky. První způsob jsme již probrali – uvnitř parametrů funkce. Pokud toto nestačí, můžeme vhodnost funkce upřesnit pomocí konstrukce **`when`**, například takto by mohla vypadat funkce počítající absolutní hodnotu:

```
abs(N) when N < 0 -> - N;
abs(N) -> N.
```

[3] Překlad „tail optimalization“ :-)

Další možností je použít `if`. Ten funguje podobně jako podmínky za `when`, ale je „uvnitř“ funkce. Absolutní hodnota by s ním vypadala takto:

```
abs(N) -> if
      N < 0 -> (-1) * N;
      true -> N
      end.
```

Před každou větví se nachází podmínka: když bude pravdivá, vybere se tato větev, když nebude, pokračuje se ve zkoušení. Každá větev (může jich být libovolně mnoho) je ukončena středníkem – kromě té poslední. Celý `if` končí klíčovým slovem `end`. Je třeba si uvědomit, že `if` má výsledek – výsledek obsahu té větve, která se vybrala.

Všimněte si „podmínky“ `true`, která platí vždy, tedy se používá místo obvyklého `else` – jako poslední možnost, kam spadne vše, co nespadlo nikam dřív.

Obdobně funguje `case`, ale v něm probíhá test na „napasování“ do proměnných, podobně jako uvnitř závorek funkce. Ukážeme si s ním upravený faktoriál:

```
fac(N) -> case N of
      0 -> 1;
      _ -> N * fac(N - 1)
      end.
```

Toto vezme ono `N` a zkusí ho uložit (podobně jako v parametrech) do té věci nalevo od `->`. Pokud se to povede, provede se vnitřek (a proměnné použité nalevo se tím případně naplní).

Další věc, která stojí za zmínku, je „proměnná“ `_`. Aby nám `case` fungoval, musíme mu dát proměnnou (nebo výraz), do které obsah uloží. Ale pokud její hodnotu nechceme použít, překladač by si stěžoval. Proměnná `_` je speciální v tom, že se používá jako odpadní a znamená, že hodnotu tady chceme pouze zahodit, nikoliv ukládat. Lze ji samozřejmě použít i na jiných místech – na levé straně přiřazení, uvnitř parametrů a podobně. Do této „proměnné“ se smí uložit vícekrát, ale zase se z ní nesmí nikdy číst.

U všech podmínek (i těch přes parametry) je potřeba, aby některou z voleb program vybral. Pokud se výpočet dostane až „za konec“ možností, program spadne s chybou, protože neví, jaký je výsledek daného výrazu.

Co se týče jazykových konstrukcí, v jedné funkci smí být více „příkazů“, které se provedou za sebou. Lze tak například rozložit složitější výpočet či vložit ladící výpis.

```
kvadratic(x, a, b, c) -> kvad = a * x * x,
      linear = b * x,
      kvad + linear + c.
```

V takovém případě je výsledkem funkce poslední výraz.

Nejen jazykovými konstrukcemi živ je program, ještě jsou potřeba data. Zatím jsme používali jen čísla. Co se jich týče, tak se s nimi smí dělat obvyklé věci, jako sčítat, násobit, dělit a podobně. `/` dělí neceločíselně, k celočíselnému dělení se používají `div`, resp. `rem` pro zbytek.

Celá čísla mohou být libovolné délky, tedy nám nehrozí, že by číslo přeteklo, jako to může nastat u jiných jazyků.

Čísla jsou užitečná, ale co kdybychom chtěli rozlišovat mezi několika druhy informace? Potom použijeme atomy. Atom je nějaké slovo, které začíná na malé písmeno, dá se použít přímo jako hodnota proměnné. Mezi atomy nefunguje žádná aritmetika a jedinou věc, kterou s nimi (kromě ukládání) smíme dělat, je porovnávat.

Například, pokud chceme rozlišovat pohlaví našich řešitelů, tak budeme mít hodnoty `girl` (slečny mají přednost, přestože jsou v KSP v menšině) a `boy`.

Dalším typem, i když trochu falešným, jsou pravdivostní hodnoty. Pravdivostní hodnotu dostaneme například porovnáním dvou hodnot. Reprezentovány jsou atomy `true` a `false`. Mezi nimi fungují obvyklé spojky `not`, `and` a `or`. U `and` a `or` nefunguje zkrácené vyhodnocování výrazů – pokud potřebujeme vyhodnocovat zkráceně, použijeme `andalso` a `orelse`.

Data je možné seskupovat do n-tic. Je to jakási obdoba recordu v Pascalu, jen jsou položky nepojmenované a jejich význam je stanoven pořadím. Každá n-tice se uzavírá do složených závorek a dá se k ní přistupovat jako k jedné proměnné, nebo ji rozložit na části (v parametrech, v přiřazení, v `case`). Mohli bychom například mít n-tici, která by obsahovala jméno člověka, věk a jeho pohlaví. Pak by jeden záznam mohl vypadat třeba takto:

```
{"Tomáš Marný", 14, boy}
```

Řetězec, použitý v tomto případě, zajisté nikoho nezmate, k němu se dostaneme později.

Funkce, která vybere věk „gentlemanským“ způsobem (ženám ho snižuje), by vypadala takto:

```
age({_, Age, boy}) -> Age;
age({_, Age, girl}) -> if
      Age > 1 -> Age - 2;
      true -> 0
      end.
```

Všimněme si, že n-tici rozebereme hned v parametrech (nemuseli bychom), jméno zahodíme a pomocí atomů rozlišujeme, o koho se jedná. Taktéž, pokud by nám někdo chtěl do funkce propašovat jedince jiného nějakého třetího pohlaví,

tak program spadne (což je správné řešení – pokud funkce něco neumí, tak je lepší spadnout, než to dělat špatně).

Problém s n-ticemi je, že obsahují pevný počet hodnot. Většinou se používají, když patří několik informací různých druhů k sobě (proto také přirovnání k recordu v Pascalu). Ale často je potřeba uchovávat víc hodnot stejného druhu, ale v době psaní programu se neví, kolik (technicky vzato, jazyk nevynucuje, aby měly stejný typ, ale bývá zvykem uchovávat pohromadě jen věci, které k sobě patří). Na toto se používají seznamy (obdoba pole v Pascalu).

Seznamy mají závorky hranaté, hodnoty se opět oddělují čárkami. Prázdný seznam jsou prostě prázdné hranaté závorky, tedy `[]`.

Není nutné pracovat vždy s celým seznamem. Pomocí svislítka lze „odseknout“ přední část a pracovat se začátkem a zbytkem. (Často se používají termíny hlava a ocas, představme si hada, který, když mu uřízneme hlavu, tak ten zbytek je zase had, kterému lze uříznout hlavu, … až nám zbude jen hlava a zbylý had bude mít nulovou délku. Ne, možná si to radši nepředstavujte, nebudeme ubližovat nevinným hadům.) Tímto způsobem lze na začátek přidávat a ze začátku odebírat hodnoty. Tedy seznam `[1, 2, 3]` je totéž jako seznam zapsaný `| [1 | [2 | [3 | []]]]`. Lépe zhlédnout na příkladu:

```
first([First | _]) -> First.

second([_, Second | _]) -> Second.

rest([_ | Rest]) -> Rest.

add(List, Item) -> [Item | List].

length([]) -> 0;
length([_ | Rest]) -> 1 + length(Rest).
```

První funkce vrací první prvek seznamu (usekne, uloží do proměnné `First`, zbytek odloží do „odpadní proměnné“ a vrátí tu první hodnotu). Druhá funkce vrací druhý prvek (obdobně). Třetí vrací seznam bez prvního prvku (tedy, o jeden ho zkrátí), čtvrtá dělá pravý opak, jeden prvek na začátek přidá. Pátá počítá délku seznamu – postupně odebírá, než nezbude nic, a přičítá jedničky. Na ní je také vidět, jak lze zařídit projití celého seznamu.

Všimněte si, že nelze pracovat s druhým koncem seznamu, máme k dispozici jen jeho začátek a k věcem dál se musíme prokousat (ukusovat hlavičky).

A ještě slíbená návštěva řetězců. Řetězce jsou seznamy jako každé jiné. Řetězec `"ahoj"` je totéž jako zápis `[97, 104, 111, 106]`, jen vypadá méně l33t (čti: je použitelný).

To bude pro tuto sérii stačit, v příští si rozebereme, jak pouštět více procesů, jak mezi nimi probíhá komunikace a jak k tomu použít více počítačů.

Aby vám čekání na další sérii nepřišlo tak dlouhé, je tu několik malých úložek:

1) Každý druhý: Napište funkci, která dostane seznam (libovolných hodnot) a vrátí seznam, ve kterém se bude vyskytovat každá druhá z tohoto seznamu. Znáte film „Nesmrtelná teta“, ne? *[3 body]*

2) Fibonacciho čísla: Funkce dostane číslo N a vrátí N-té Fibonacciho číslo v pořadí. Fibonacciho čísla jsou definována takto:

$$\mathrm{Fib}_1 = 0$$
$$\mathrm{Fib}_2 = 1$$
$$\mathrm{Fib}_n = \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2} \quad \forall n \geq 2$$

Dejte si pozor na časovou složitost. *[4 body]*

3) Prvočísla: Funkce dostane číslo N a vrátí seznam prvních N prvočísel. *[5 bodů]*

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 vynásobí 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. Skončí v případě, že místo objednávky dostane oznámení, že už toho bylo dost.

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.
```

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, 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]

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):

```
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.

Seriál: APL

Martin Mareš

22-4-7 Pozdrav z pravěku 12 bodů

Láká vás podívat se na programovací jazyk z dob, kdy muži byli ještě praví muži, ženy pravé ženy a počítače praví mamuti, aspoň co do velikosti? Jazyk, v němž opravdoví programátoři píší programy jen tehdy, když se vejdou na jeden řádek, což je ovšem překvapivě často? Jazyk, který má tolik jednoznakových příkazů, že by se na psaní všech těch znaků hodila klaviatura od varhan? Pak jste na správné adrese, protože dnes si budeme povídat o jazyku APL.

Historie APL se začala psát v roce 1964, kdy Kenneth Iverson dumal nad tím, proč chce-li něco jednoduchého spočítat, musí kvůli tomu psát složitý program, když přitom celý výpočet lze příjemně a stručně popsat matematickou notací. Chvíli experimentoval, až napsal interpreter jazyka založeného právě na matematické notaci a k tomu knížku s prostým názvem A Programming Language. A APL bylo na světě. Pojďme si ukázat pár jeho nejdůležitějších vlastností a zkusit si v něm zaprogramovat. Mezi řečí občas potkáte jednoduché úkoly, tentokrát po vás budeme chtít, abyste vymysleli co nejelegantnější (zejména co nejkratší) řešení bez ohledu na časovou složitost.

Předně bychom měli vědět, že *APL je interaktivní* – když mu napíšete výraz, interpreter ho ihned vyhodnotí a vypíše vám výsledek (v dnešní době nic moc překvapivého, ale v době dřevěných počítačů …). Třeba na `1+2+4+8` vypíše `15`, na `3*6` vypíše poněkud nečekaně `729`, protože `*` značí umocňování, kdežto násobení se značí `×`. Tedy `3×6` je `18`. I ostatní operace se píší trochu nezvykle: dělení je `÷`, zbytek po dělení `|` (pozor, má opačně argumenty, takže `3|7` = 7 mod 3 = 1). Ještě nás asi překvapí, že `3×4+1 = 15`, protože neexistují priority operací a vše se striktně vyhodnocuje zprava doleva, pokud neurčíte jinak závorkami.

Většina operací existuje ve dvou formách: *dyadické* (ty chtějí dva argumenty, dneska bychom jim asi spíš říkali *binární*) a *monadické* (žádají jeden argument, jinak též *unární*). Obvykle obě formy počítají něco podobného: například dyadické `x÷y` dělí a monadické `÷y` počítá převrácenou hodnotu. Celý zvěřinec *aritmetických operací* vypadá takto:

`x+y`	sčítání	`+x`	identita (vrací `x`)
`x-y`	odčítání	`-x`	otočení znaménka
`x×y`	násobení	`×x`	signum (viz níže)
`x÷y`	dělení	`÷x`	1/x
`x|y`	y mod `x`, `0|x=x`	`|x`	absolutní hodnota
`x⌈y`	maximum	`⌈x`	horní celá část
`x⌊y`	minimum	`⌊x`	dolní celá část

Operace `×x` vrátí jedničku pro kladné `x`, `-1` pro záporné `x` a nula od nuly pojde.

Logické operace se zapisují jako v matematice, přičemž na vstupu považují 0 za nepravdu a cokoliv nenulového za pravdu; na výstupu dávají vždy 0 nebo 1. Podobně při porovnávání čísel dostanete vždy výsledek 0 nebo 1:

`x∨y`	nebo	`x<y`	menší než
`x∧y`	a zároveň	`x>y`	větší než
`~x`	negace	`x≤y`	menší nebo rovno
`x=y`	rovnost	`x≥y`	větší nebo rovno
`x≠y`	nerovnost		

Přiřazení funguje, jak čekáte, a značí se šipkou: `x←5` přiřadí do proměnné `x` hodnotu `5`. Příjemné je, že se chová jako funkce a vrací hodnotu, kterou přiřadilo, takže můžeme psát (podobně jako třeba v Céčku) `x←y←5`.

Můžete si také *definovat vlastní operace.* Nejjednodušší je to pro monadické: `f x: x⋆2` zavede monadickou operaci `f`, která vrátí druhou mocninu svého parametru. Kdybychom chtěli definovat dyadickou operaci `g`, řekněme pro součet druhých mocnin, napíšeme `x g y: (f x) + (f y)`. Funkce více parametrů dostávají parametry ve složených závorkách oddělené středníky: `max{a;b;c}: a⌈b⌈c`.

Úkol 1 (1 bod): Nadefinujte operaci `×x` (signum) pomocí ostatních operací.

APL je vektorové – v podstatě všechno, co umí provádět s čísly, dokáže i s posloupnostmi čísel (čili vektory). Například `1 2 4 + 3 1 2` sečte vektory `1 2 4` a `3 1 2` po prvcích, takže vyjde `4 3 6`. Pokud k vektoru přičítáme konstantu, přičte se ke každému prvku: `1 2 4 + 1 → 2 3 5`. (Zde raději místo rovnítka značíme výsledek šípkou, protože `=` by se jako každá jiná operace na vektory aplikovalo po prvcích.) Obecně to funguje takto: Pokud použijeme na vektor monadickou operaci, provede se s každým prvkem zvlášť. Použijeme-li dyadickou na dva stejně dlouhé vektory, provede se po prvcích (první s prvním, druhý s druhým atd.). Pokud dyadickou na vektor a číslo, z čísla se vyrobí vektor tím, že se číslo zopakuje.

Často potřebujeme vyrobit vektor čísel $0, \ldots, n-1$. Tehdy se hodí operátor `ιn` (*iota*), který přesně takové vektory vytváří.

Mimo vektorů APL dokáže pracovat i s vícerozměrnými poli: `ι a b c` vytvoří trojrozměrné pole velikosti `a` × `b` × `c` a vyplní ho čísly od 0 do `(a×b×c)-1`. Obecně můžete operátoru `ι` dát jako parametr libovolný vektor a on vás obdaří polem, které má tolik rozměrů, kolik je délka vektoru, přičemž každá složka určuje, jak je pole v daném rozměru velké. Počtu rozměrů se říká *rank* pole. Dvojrozměrným polím budeme říkat *matice.*

Úkol 2 (1 bod): Co udělá `ι5+1` a co `1+ι5`? (Nezapomeňte, v jakém pořadí se vyhodnocují operace.)

Úkol 3 (1 bod): Co udělá `ι(2+ι3)`?

Pokud chceme zjistit, jak je nějaké pole velké, stačí použít monadický operátor `ρ` (*ró*). Ten vrátí vektor, jehož jednotlivé složky jsou velikosti v jednotlivých rozměrech. Tedy `ρ⍳2 3 4` vrátí `2 3 4`. Je-li `x` vektor, `ρx` musí tedy být jedno číslo, které udává počet prvků vektoru. Proto `ρρp` vrátí rank pole `p`.

Ještě jsme ale nepřišli na to, jak vícerozměrné pole zadat. K tomu se hodí dyadická forma operace `ρ`. Ta slouží k *přeformátování* pole na dané rozměry: `2 3 ρ 4 5 6 7 8 9` například vezme vektor `4...9` a přerovná ho na matici o dvou řádcích `4 5 6` a `7 8 9`. Pokud má původní pole příliš málo prvků, začnou se opakovat: `2 3 ρ 1 2` vyrobí matici s řádky `1 2 1` a `2 1 2`. Přerovnávání přitom zachovává standardní pořadí prvků: vektor se čte zleva doprava, matice se čte po řádcích, vícerozměrné tak, že se první rozměr mění nejpomaleji a poslední nejrychleji.

Pole můžeme také indexovat (od nuly): pokud je `x` vektor, pak `x[0]` je jeho nultý prvek, `x[1]` první atd. Jako index můžeme také použít vektor, v tom případě vybereme více prvků najednou: `x[0 2 3]` dá vektor skládající se z prvků `x[0] x[2] x[3]`. U vícerozměrných polí se indexy oddělují středníkem: `y[1;3]` je třetí prvek na prvním řádku matice `y`, `y[2 3;2 3]` je čtvercová podmatice 2×2 „vykousnutá" z matice `y`. Na pole ranku k se také můžeme dívat jako na vektor, jehož prvky jsou pole ranku $k-1$, tedy pro matici `y` je `y[0]` její první řádek, `y[1]` druhý atd.

Existuje řada dalších operací, které zacházejí s vektory. Zajímavé je třeba spojování vektorů za sebe nebo pod sebe. Spojení za sebe se zapisuje čárkou: `x,y` je vektor, který obsahuje nejprve všechny prvky vektoru `x` a pak všechny prvky z `y`. Spojení pod sebe neboli *laminace* `x~y` má za výsledek dvojřádkovou matici, jejiž prvním řádkem je vektor `x` a druhým vektor `y` (řádky přitom musí být stejně dlouhé). Obě operace samozřejmě fungují i na vícerozměrná pole: čárka spojuje v prvním z rozměrů (matice tedy slepuje pod sebe), laminace přidá na začátek nový rozměr, takže `(x~y)[0]→x` a `(x~y)[1]→y`.

Hodit se nám bude též *redukce* `+/x`. Ta spočte pro vektor $\mathtt{x} = x_0 x_1 \ldots x_{n-1}$ výraz $x_0 + x_1 + \ldots + x_{n-1}$, tedy součet všech prvků. Podobně můžete pomocí lomítka vyrobit z nějaké operace její redukční verzi zpracovávající postupně prvky vektoru: například `⌈/x` vrátí maximum z vektoru.

U vícerozměrných polí se redukce chová trochu záludněji. Pokud použijeme `+/` například na matici, což, jak už víme, je vektor řádkových vektorů, sečte nám jednotlivé řádky, takže vznikne vektor, jehož i-tá složka vyjadřuje součet i-tého sloupce matice. Třeba pro matici

$$\mathtt{x} = \mathtt{⍳3\ 4} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

spočítáme `+/x → (0 1 2 3)+(4 5 6 7)+(8 9 10 11) → 12 15 18 21`.

Co kdybychom naopak chtěli sčítat prvky v každém řádku? Tehdy je nejjednodušší použít operaci transpozice ⍉, která prohodí obě souřadnice – řádky se nyní stanou sloupci a naopak. Pro řádkové součty tedy stačí `+/⍉x`.

Úkol 4 (1 bod): Jak spočítat maximum ze všech prvků matice?

Na lomítko se můžeme dívat jako na něco, co dostane dyadickou operaci s čísly a vyrobí z ní monadickou operaci s vektory (nebo obecněji z dyadické operace s poli ranku k vyrobí monadickou operaci s polem ranku $k+1$). Takových konstrukcí, které vyrábějí z operací jiné operace, má APL víc a říká se jim *operátory.*

Dalším důležitým operátorem je *direktní součin* `∘.f`, kde `f` je nějaká dyadická operace. Pokud ho použijeme na vektory `x` a `y` (tedy napíšeme `x∘.fy`), dostaneme matici, jejíž prvek na pozici (i,j) obsahuje výsledek operace `f` aplikované na i-tý prvek vektoru `x` a j-tý prvek vektoru `y`. Výraz `(⍳10)∘.×(⍳10)` nám tedy vytiskne malou násobilku. Direktní součiny samozřejmě fungují i s poli vyšších ranků; snadno si domyslíte, jak, když napovíme, že součinem pole ranku k s polem ranku ℓ bude pole ranku $k+\ell$.

Potkali jsme tedy zatím tyto operace pracující s poli:

`⍳n`	iota: generátor přirozených čísel
`⍴x`	zjištění rozměrů pole
`x⍴y`	přeformátování pole `y` na rozměry `x`
`x[...]`	indexování pole
`⍉x`	transpozice (překlopení)
`x,y`	spojení za sebe
`x~y`	laminace (spojení pod sebe)
`f/x`	operátor redukce
`x∘.fy`	operátor direktního součinu

S prohlídkou dalších operací a operátorů počkáme do příště, i s těmi, co už známe, se dá naprogramovat ledacos:

Úkol 5 (2 body): Napište funkci, která vytvoří matici $n\times n$ z nul a jedniček, která bude mít na i-tém řádku i jedniček a za nimi $n-i$ nul.

Úkol 6 (3 body): Jak pro dané sudé n vyrobit vektor $0, n-1, 1, n-2, 2, n-3, \ldots, n-1, 0$?

Úkol 7 (3 body): Vymyslete, jak v APL spočítat největšího společného dělitele dvou čísel.

Ještě dodejme, že jde vytvářet i delší programy – stačí napsat více řádků a na každém jeden výraz, nebo případně více výrazů na řádek oddělených středníkem. Pokud byste si chtěli nainstalovat opravdový překladač APL, podívejte se na odkazy ve webové verzi tohoto zadání. Fonty a TEXová makra na sázení APL najdete tamtéž. Pokud se vám nechce nic instalovat, můžete samozřejmě psát programy na papír a nemajíce varhany, psát místo podivných znaků jejich názvy.

APL a dnešní doba. Masového rozšíření se APL nikdy nedočkalo (snad proto, že jen opravdovým programátorům vyhovuje programovat tak, že hodinu se zavřenýma očima přemýšlejí, načež si sednou k počítači a napíší jednořádkový program). Přesto ale není pouze mrtvým jazykem vystaveným v muzeu počítačové prehistorie – stále vznikají nové implementace APL (například `A+` nebo dokonce `APL.Net`) a jazyky od APL odvozené (třeba jazyk `J`, který se snaží vystačit si bez varhan, tedy se znaky na běžné klávesnici). Navíc se s nástupem víceprocesorových počítačů ukazuje, že programy skládané pomocí vektorových operací a operátorů, jako je třeba naše redukce a direktní součin, jsou velice dobře paralelizovatelné. Kdo ví, co se ještě o APL naučíme.

22-5-7 ArcheoPaleoLingua 14 bodů

V této sérii si ještě jednou vyzkoušíme APL, programovací jazyk z dob našich prababiček, pradědečků a pralidí vůbec. K základním operacím a operátorům, které jsme si zavedli v úloze 22-4-7, doplníme pár dalších a naprogramujeme něco krapet složitějšího. A opět bez namáčení, ... totiž bez cyklů.

Dyadický operátor *replikace* `x/y` dostane dva vektory stejné délky a vytvoří vektor, ve kterém se nejprve vyskytuje `x[0]` kopií prvku `y[0]`, pak `x[1]` kopií prvku `y[1]`, atd. Tedy například `3 0 2/4 5 6 → 4 4 4 6 6`. Pokud se ve vektoru x vyskytují jen nuly a jedničky, replikace vlastně jen vybere ty prvky z y, na jejichž pozici se v x vyskytuje jednička, tedy provede jakousi *kompresi* vektoru y.

Inverzní operací k této kompresi je dyadický operátor *expanze* `x\y`. Ten na vstupu potřebuje vektor x složený z nul a jedniček a vektor y, jehož délka je rovna počtu jedniček v x. Výsledkem je pak vektor vzniklý z x nahrazením každé jedničky odpovídajícím prvkem z y. Kupříkladu tedy platí `0 1 1 0 0 1\4 5 6 → 0 4 5 0 0 6` a také `x/x\y → y`.

Konečně přidáme operátor *scanování* `f\x`, kde f je dyadická operace a x vektor. Dá nám vektor, jehož i-tá složka je redukce `f/` prvních $i+1$ složek vektoru x. Nechme raději hovořit příklad: `+\1 2 3 → 1 (1+2) (1+2+3) → 1 3 6`.

A nyní slíbené úkoly:

Úkol 1 (5 bodů): Napište funkci, která najde všechna prvočísla menší než dané N.

Úkol 2 (4 body): Jak v APL uspořádat daný vektor čísel vzestupně? Slibujeme, že se žádné číslo nebude opakovat. (Zde prosíme používejte pouze podmnožinu APL, kterou jsme nadefinovali v našem seriálu. Operátory ⍒ a ⍋, jakkoliv krásné, se nepočítají.)

Úkol 3 (5 bodů): Je dán vektor nul a jedniček. Jak zjistit délku nejdelšího souvislého úseku tvořeného jedničkami?

Připomínáme, že u této úlohy na časové ani paměťové složitosti nezáleží, rozhodující je krátkost a elegance vašeho programu. Archeologii zdar.

Programátorské kuchařky

20-4-K Kuchařka čtvrté série – vyhledávání v textu

V dnešním vydání kuchařky se podíváme na vyhledávání slov v textu. Náš úkol tentokrát zní: Máte seznam slov a *hodně dlouhý* text, vypište všechny výskyty těchto slov v textu. Ukážeme si řešení, kterému stačí jeden průchod textem a lineární čas na předzpracování slovníku.

Pro začátek si zavedeme několik pojmů:

- Mějme nějakou konečnou *abecedu* Σ, tedy množinu všech *znaků.* Klidně si představujte klasickou latinskou abecedu, ale může to být např. i množina $\{0, 1\}$.
- Σ^* je množina všech *slov*, která lze z naší abecedy utvořit. To jsou všechny konečné posloupnosti znaků z Σ. Takové slovo může tudíž být i posloupnost 01101. Slova budeme značit řeckými písmenky a zvláštní postavení mezi nimi má *prázdné slovo* ε.
- $|\alpha|$ pro $\alpha \in \Sigma^*$ je *délka* slova, tedy počet jeho znaků.
- $\alpha\beta$ pro $\alpha, \beta \in \Sigma^*$ je *zřetězení* slov α a β, tedy slovo, které vznikne zapsáním slov α a β za sebe.
- γ^k je slovo vzniklé k-násobným zopakováním slova γ. Tedy $\gamma^0 = \varepsilon$, $\gamma^{k+1} = \gamma^k\gamma$.
- Slovo α nazveme *podslovem* slova β, pokud je α obsaženo v β, čili pokud $\beta = \gamma\alpha\delta$ pro nějaká slova γ a δ.
- Řekneme, že slovo α je *prefixem* slova β, pokud slovo β začíná slovem α, čili $\beta = \alpha\delta$ pro nějaké slovo δ.
- Podobně α je *suffixem* slova β, pokud β končí slovem α, tedy $\beta = \delta\alpha$ pro nějaké slovo δ.
- Každé slovo je prefixem i suffixem sebe sama, takovému pre-/suffixu říkáme *nevlastní;* všem ostatním *vlastní.*
- Všimněte si, že prázdné slovo je podslovem, prefixem i suffixem každého slova včetně prázdného slova.

Po tomto teoretickém úvodu se konečně zamyslíme nad vlastním vyhledáváním. Ponejprv si úlohu trochu zjednodušme a zkoumejme případ, kdy hledáme všechny výskyty jednoho slova $\alpha \in \Sigma^*$ o délce $|\alpha| = p$ v textu $\beta \in \Sigma^*$, $|\beta| = n$. (Hledanému slovu se často říká jehla, textu kupka sena.)

Asi první algoritmus, který nás napadne, je procházet text β od začátku až do konce a pro každou pozici i v textu zkontrolovat, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provedeme až p porovnání znaků, čili celkem až np porovnání. To není nic pěkného, zkusme to lépe.

Všimněme si, že porovnávání slova s textem může skončit dvěma způsoby. Buď zjistíme, že se slovo s textem shoduje celé, nebo najdeme v textu znak, který ve slově není. Tehdy nestačí pokračovat novým vyhledáváním od místa, kde jsme skončili: např. pro slovo `instinkt` a text `instinstinkt` by algoritmus u druhého `s` zjistil, že se text liší, a pokud by pokračoval dále, již by nenalezl skutečný výskyt slova. Proto se vždy musíme vrátit o kousek zpět, v předchozím algoritmu jsme se vraceli vždy těsně za místo, kde se text začal se slovem shodovat.

Na druhou stranu, když se takto vrátíme, začneme znovu zpracovávat text, který už jsme jednou četli, takže je vlastně předem dáno, jak to dopadne. Pojďme toho využít. Říkejme *stavy* prefixům slova α. Pro každou pozici i v textu si označme $r[i]$ nejdelší stav, který je obsažen v textu tak, že v něm končí na pozici i (nebo vezměme nejdelší suffix prvních i znaků textu, který je stavem – to je totéž). Posuneme-li se v textu o pozici dále, další znak $\beta[i+1]$ buď prodlouží prefix $r[i]$, a tím určitě získáme nový nejdelší stav $r[i+1]$ (rozmyslete si, že nemůže existovat delší), nebo už prefix není možné prodloužit, a tehdy budeme muset najít jiný. Nahlédněme ale, že useknutím posledního písmenka stavu získáme zase stav, takže useknutím posledního písmenka stavu $r[i+1]$ získáme nějaký suffix stavu $r[i]$. Naše $r[i+1]$ tedy vznikne prodloužením co možná nejdelšího suffixu stavu $r[i]$ o písmenko $\beta[i+1]$ (některé suffixy prodloužit nejdou, vezměme nejdelší, který jde). Pro předchozí příklad a prefix `instin` to bude suffix `in`.

Jelikož nový stav získáme ze suffixů předchozího stavu, nemusíme vědět vůbec nic o předcházejících písmenech textu. Postačí nám předpočítat si pro každý stav σ jeho nejdelší vlastní suffix, který je také stavem – ten si označíme $f(\sigma)$ a funkci f budeme říkat *zpětná funkce*. Přechod od $r[i]$ k $r[i+1]$ budeme provádět tak, že zkusíme $r[i]$ prodloužit o znak $\beta[i+1]$ a když to nepůjde, zkrátíme si $r[i]$ pomocí zpětné funkce a opět zkusíme přidat tentýž znak, pokud to stále nejde, zkracujeme dál opětovným zavoláním zpětné funkce, dokud se nám prodloužení nezdaří nebo dokud nedostaneme prázdné slovo.

Když navíc během výpočtu narazíme na i, pro které je $r[i] = \alpha$, ohlásíme výskyt slova α.

Aby se nám se stavy v programu pohodlně pracovalo, očíslujeme si je – j-tý stav bude prefixem slova α o délce j. Zpětná funkce pak bude přiřazovat číslům čísla, takže si ji můžeme pamatovat v obyčejném jednorozměrném poli.

Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? Jak spočítat zpětnou funkci? Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až p-krát. Při každém volání však klesne délka aktuálního stavu alespoň o jedna, zatímco

kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků lineární v délce textu.

Konstrukci zpětné funkce provedeme malým trikem. Všimněte si, že $f(i)$ je přesně stav, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\alpha[2\ldots i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco $r[i]$ označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku. Takže f získáme tak, že spustíme vyhledávání na část samotného slova w. Jenže k vyhledávání zase potřebujeme funkci f. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $f(1) = \varepsilon$. Pokud již máme $f(i)$, pak výpočet $f(i+1)$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celý řetězec $\alpha[2\ldots p]$ a sledovat, jakými stavy bude procházet, a to budou přesně hodnoty zpětné funkce. Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $p-1$, a proto poběží v čase $\mathcal{O}(p)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(n+p)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP.

Tento algoritmus můžeme také formálně popsat pomocí automatů:

Konečný automat nad abecedou Σ si můžeme představit jako stroj, kterému dáme slovo ze Σ^* a on ho buď odmítne nebo přijme. V průběhu práce je vždy v právě jednom *stavu* z nějaké pevné množiny stavů. Slovo zpracovává po jednotlivých znacích a podle přečteného znaku se rozhodne, do jakého stavu přejde. K tomu slouží *přechodová* funkce g, která dvojicím $(\textit{aktuální stav}, \textit{nový znak})$ přiřazuje nové stavy. Pokud vstupní slovo dojde, automat podle toho, v jakém stavu se právě nachází, odpoví, že je slovo přijato nebo odmítnuto.

Konečný automat můžeme formálně nadefinovat jako čtveřici (Q, g, q_0, F), kde:

- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která pro daný stav automatu a znak na vstupu řekne, do jakého stavu má automat přejít;
- $q_0 \in Q$ je *počáteční stav,* v němž je automat na počátku výpočtu;
- $F \subset Q$ je množina *přijímacích stavů.*

Výpočet konečného automatu pak probíhá následovně:

1. Nastav aktuální stav s_0 na počáteční stav q_0.
2. Postupně čti znaky $x[i]$ ze vstupu a po přečtení každého přejdi ze stavu s_{i-1} do stavu $s_i = g(s_{i-1}, x[i])$.
3. Pokud skončíš v přijímacím stavu ($s_n \in F$), pak slovo přijmi.

Příklad: Mějme automat nad abecedou $\Sigma = \{\texttt{0}, \texttt{1}\}$ se třemi stavy $s_1 \ldots s_3$, počátečním stavem $q_0 = s_1$, jedním přijímacím stavem $F = \{s_1\}$ a přechodovou funkcí g dle tabulky:

$$\begin{array}{lll} g(s_1, \texttt{0}) = s_3 & g(s_2, \texttt{0}) = s_1 & g(s_3, \texttt{0}) = s_3 \\ g(s_1, \texttt{1}) = s_2 & g(s_2, \texttt{1}) = s_3 & g(s_3, \texttt{1}) = s_3. \end{array}$$

Tento automat přijímá právě slova ve tvaru $(\texttt{10})^k$, $k \geq 0$, tedy např. `101010` a prázdné slovo přijme, zatímco `1010101` odmítne.

Konečné automaty docela dobře popisují chod našeho algoritmu – ten také zpracovává text po znacích a přechází podle právě přečteného znaku mezi stavy. Jsou zde ale ještě některé rozdíly: předně KMP neodpovídá ano/ne, ale hlásí jednotlivé výskyty. K tomu můžeme automat upravit například tak, že množinu přijímacích stavů bude používat nejen na konci vstupu, ale v každém kroku. Druhá odlišnost tkví v tom, že přechodová funkce KMP (ta odpovídá prodlužování prefixu o další písmeno) není definována všude. Tam, kde definována není, nastupuje místo ní zpětná funkce, která nás přesouvá mezi stavy tak dlouho, než přechodová funkce definována je.

Tomuto rozšíření se obvykle říká *vyhledávací automat* a definuje se jako pětice (Q, g, f, q_0, out), kde:

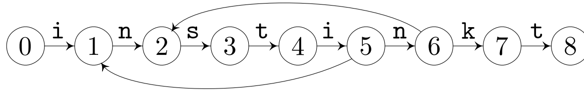
- Q je konečná množina stavů automatu;
- $g : Q \times \Sigma \to Q$ je *přechodová funkce,* která je definovaná pouze pro některé dvojice $(stav, znak)$;
- $f : Q \to Q$ je *zpětná funkce,* která říká, do jakého stavu se má automat přesunout, pokud přechodová funkce není definována;
- $q_0 \in Q$ je *počáteční stav,* v němž se automat nachází na začátku výpočtu;
- $out : Q \to \mathcal{P}(\Sigma^*)$ je *výstupní funkce,* která každému stavu přiřazuje, jaký se v něm má ohlásit výstup, což bude množina nalezených slov. (V případě KMP byla vždy buďto prázdná nebo jednoslovná, až budeme za chvíli hledat více slov, bude bohatší.)

Výpočet vyhledávacího automatu pak probíhá následovně:

1. Nastav aktuální stav s na počáteční stav q_0.
2. Pro každý znak $c = x[i]$ vstupního textu proveď:
3. Dokud je $g(s, c)$ nedefinovaná, přejdi zpět do stavu $s \leftarrow f(s)$.
4. Přejdi do nového stavu $s \leftarrow g(s, c)$.
5. Vypiš všechna slova z $out(s)$.

Ještě doplňme, že aby se algoritmus vždy zastavil, musí být $g(q_0, c)$ definováno pro každý znak $c \in \Sigma$, obvykle opět jako q_0.

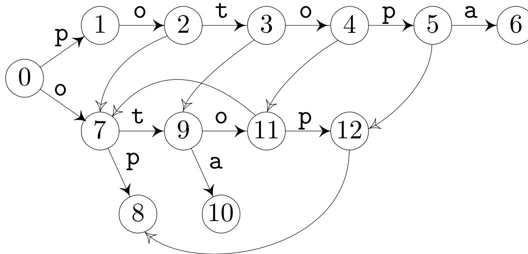
Příklad: Pro slovo `instinkt` by vyhledávací automat vypadal takto (zpětnou funkci jsme kreslili pouze tam, kde nevede do stavu 0):



Nyní algoritmus KMP rozšíříme, aby uměl hledat více slov. Mějme slovník K, což je konečná množina slov nad abecedou Σ, a prohledávaný text β. Vytvoříme vyhledávací automat, jehož výstupem bude výpis nalezených slov a jejich pozic v textu. Jeho stavy budou odpovídat prefixům všech slov ze slovníku a očíslujeme si je přirozenými čísly, počáteční stav $q_0 = 0$ bude odpovídat prázdnému prefixu. Výstupní funkce *out* pro prefix α ohlásí všechna slova ze slovníku, která jsou suffixem slova α.

Příklad: Jak takový vyhledávací automat může vypadat, si ukážeme pro latinskou abecedu a slovník

$$K = \{\texttt{potopa}, \texttt{op}, \texttt{ota}, \texttt{otop}\}.$$



Rovnými čarami je zobrazena přechodová funkce, kroucenými zpětná funkce. Nejsou zakresleny šipky do 0 u přechodové ani u zpětné funkce. Výstupní funkce je dána následující tabulkou:

$$\begin{aligned} &out(5) = \{\texttt{otop}, \texttt{op}\} && out(10) = \{\texttt{ota}\} \\ &out(6) = \{\texttt{potopa}\} && out(12) = \{\texttt{otop}, \texttt{op}\} \\ &out(8) = \{\texttt{op}\} && out(\text{ostatní}) = \emptyset. \end{aligned}$$

Vyhledávání pomocí tohoto automatu bude probíhat stejně jako u KMP, $r[i]$ opět bude nejdelší stav, na který končí právě přečtená část textu, složitost vyhledávání bude opět $\mathcal{O}(n)$ až na vypisování výskytů, které poběží v čase $\mathcal{O}(\textit{počet výskytů})$, což může být více než lineárně, ale lépe to určitě nejde.

Pro pořádek dokážeme, že automat doopravdy vyhledává všechny výskyty: (i) Každé slovo, které oznámíme jako nalezené, se v textu opravdu vyskytuje

($r[i]$ se v textu vyskytuje podle své definice a všechna oznámená slova jsou suffixy $r[i]$). (ii) Všechny výskyty opravdu oznámíme. Pokud se na pozici i vyskytuje slovo $\alpha \in K$, pak je zajisté α jedním ze stavů, na něž $\beta[1 \ldots K]$ končí a $r[i]$ musí být buďto tento stav nebo nějaký ještě delší, jehož je α suffixem.

Teď se podíváme na to, jak vyhledávací automat pro daný slovník sestrojit. Provedeme to ve dvou krocích. Nejprve sestrojíme množinu stavů Q, přechodovou funkci g a částečnou výstupní funkci o. Ve druhém kroku vytvoříme zpětnou funkci f a rozšíříme o na výstupní funkci *out*.

V prvním kroku založíme počáteční stav 0, postupně projdeme celý slovník K a každé slovo σ ze slovníku do automatu přidáme. To provedeme tak, že začneme ve stavu 0 a pustíme automat na σ. Jakmile ale v některém stavu s pro znak $\sigma[i]$ nebude přechodová funkce definována, přidáme nový stav q, nastavíme přechodovou funkci $g(s, \sigma[i]) = q$, přejdeme do stavu q a pokračujeme. Tím v lineárním čase vytvoříme strom stavů. Pokaždé, když dojdeme na konec slova, nastavíme také částečnou výstupní funkci $o(q)$ na $\{\sigma\}$.

Popíšeme tuto část formálně:

1. Začni s množinou stavů $Q \leftarrow \{0\}$.
2. Pro každé slovo σ ze slovníku K proveď kroky 3–7:
3. Nastav aktuální stav s na 0.
4. Pro každé písmeno $\sigma[i]$ slova σ proveď 5–6:
5. Pokud je $g(s, \sigma[i])$ nedefinované, založ nový stav q, nastav $Q \leftarrow Q \cup \{q\}$ a polož $g(s, \sigma[i]) \leftarrow q$.
6. Přejdi do nového stavu: $s \leftarrow g(s, \sigma[i])$.
7. Nadefinuj částečnou výstupní funkci: $o(s) \leftarrow \{\sigma\}$.

Zpětnou funkci vytvoříme podobně jako pro jedno slovo tak, že pustíme ještě nehotový automat na část vyhledávaného slova. Opět chceme využít toho, že je funkce definovaná pro všechna kratší slova. Vezměme si náš příklad. Při přidávání slova `potopa` bychom nastavili $f(1) = 0$, $f(2) = 7$, $f(3) = 9$, ale u druhého `o` bychom chtěli použít zpětnou funkci $f(9)$, která ještě není definovaná. Proto budeme postupovat pro všechna slova ze slovníku současně v pořadí podle rostoucí vzdálenosti od stavu 0.

Ještě vyřešíme výstupní funkci. Označme $\sigma(s)$ slovo, jehož cesta vede do stavu s. Pokud pro stav s platí $f(s) = 0$, znamená to, že neexistuje žádný nevlastní (neprázdný) suffix, který by byl prefixem některého ze slov ve slovníku. Proto v tomto stavu může skončit pouze slovo $\sigma(s)$. Nastavíme $out(s) = o(s)$. Pokud $f(s) \neq 0$ končí v tomto stavu také všechna slova, které jsou suffixem slova $\sigma(s)$. Tehdy je $out(s) = o(s) \cup out(f(s))$.

Opět formálně:

1. Založ frontu F, zatím prázdnou.
2. Nastav $f(0) \leftarrow 0$ a $out(0) \leftarrow \emptyset$.
3. Pro každý znak $c \in \Sigma$ proveď následující krok:
4. Pokud je stav $s \leftarrow g(0, c) \neq 0$ pak nastav $f(s) \leftarrow 0$, $out(s) \leftarrow o(s)$ a zařaď s na konec fronty F.
5. Dokud je nějaký stav ve frontě, prováděj následující:
6. Odeber první stav r z fronty F.
7. Pro každý znak $c \in \Sigma^*$, pokud je $g(r, c) \neq 0$, proveď:
8. Označ $s \leftarrow f(r)$. Dokud $g(s, c) = 0$, zvol $s \leftarrow f(s)$.
9. Nastav $f(s) \leftarrow g(s, c)$.
10. Nastav $out(s) \leftarrow o(s) \cup out(f(s))$.
11. Zařaď s na konec fronty F.

Aby algoritmus fungoval rychle, musíme zvolit šikovnou reprezentaci výstupní funkce. Kdyby si každý stav pamatoval svou vlastní množinu, mohly by tyto množiny dohromady být víc než lineárně velké (zkuste vymyslet příklad slovníku, pro který tomu tak je) a museli bychom se vzdát naděje, že stihneme automat zkonstruovat v lineárním čase. Proto použijeme trik: všimneme si, že $out(s)$ je pro každý stav buďto rovna $out(f(s))$ nebo se od ní liší přidáním slova $o(s)$. Stačí si proto pamatovat $o(s)$ a ještě nějakou funkci $z(s)$, která řekne, ve kterém stavu máme najít zbytek množiny $out(s)$. Krok 9 proto upravíme takto:

9. Pokud je $o(f(s)) = \emptyset$, polož $z(s) \leftarrow z(f(s))$, jinak $z(s) \leftarrow f(s)$.

Podobně upravíme vypisování nalezených slov: vypíšeme $o(s)$ a pokud je $z(s) \neq 0$, pokračujeme ve vypisování ve stavu $z(s)$.

Ještě se zamysleme nad časovou složitostí. Označme P velikost celého slovníku. První část algoritmu provede maximálně $\mathcal{O}(P)$ kroků, pokud považujeme velikost abecedy za konstantu. Ve druhé fázi se každý stav dostane do fronty právě jednou, takže vše je lineární až na průchody zpětnou funkcí. Můžeme si ale všimnout, že podobně jako u KMP i zde vlastně spouštíme vyhledávací automat na všechna hledaná slova bez prvního písmene, až na to, že místo jedno po druhém je zpracováváme na přeskáčku a že společné části výpočtů (než se strom rozvětví) počítáme jen jednou. Celkem to tedy bude trvat nejvýše tolik, kolik vyhledání všech slov dohromady, což je $\mathcal{O}(P)$.

Celkové tedy vyhledávací algoritmus běží v čase $O(P + n + v)$, kde n je délka textu, P celková velikost slovníku a v počet nalezených výskytů. Na závěr dodejme, že tento algoritmus vymysleli pan Aho a paní Corasicková.

Dnešní menu Vám servírovali
Martin Mareš a Petr Škoda

Vzorová řešení

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 $\mathcal{O}(n+m)$.[4] 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

[4] V grafových úlohách se často používá n pro počet vrcholů a m pro počet hran; tak je tomu i tentokrát.

22-1-2 Sad

Fareyovy posloupnosti

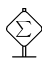
„Však se podívej na druhou úlohu – Catalanova čísla a nic jiného!"

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 generovaých 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 přiznivých 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?

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

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

Tohle vypadá obtížně, viďte? Musíme na to oklikou.

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:

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

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

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 pro všechna $j \in \{1, 2, \ldots, N\}$ 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 $\lfloor N^2 \cdot a/b \rfloor$. 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 ze zadaného 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 $\Phi = (c+d)(bx - ay) + (a+b)(cy - dx)$ 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. 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 $(x + y) \equiv$ $\equiv i \pmod n$ (obsahuje tedy body $(i, 0), (i-1, 1), \ldots$), vedlejší diagonála v_i má rovnici $(x - y) \equiv i \pmod n$ (tedy $(i, 0), (i+1, 1), \ldots$).

Zkusme na to jít jednoduše, budeme dávat dámy postupně na řádky (tedy 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 třeba 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 $(x_i + y_i) \equiv$ $\equiv (y_i k + y_i) \equiv y_i(k+1) \pmod n$, 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 $x_i \equiv ky_i$ 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 ($x_i \equiv ky_i \pmod n$), nešly by třeba pro zlá n rozestavit dámy jinak? Marná snaha, pro sudá n doká-

žeme 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 $\sum y_i + \sum x_i \equiv n \cdot (n-1)/2 + n \cdot (n-1)/2 \pmod n$ (každá dáma má svou vlastní horizontální i vertikální přímku) – tedy $n \cdot (n-1)/2 \equiv 0 \pmod n$.

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 $n(n-1)/2 \equiv 0 \pmod n$ 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 druhých 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.

Martin Mareš & Vojtěch Tůma

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[5], 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

[5] v reedici vydána se 4. sérií

nemohli jít do tohoto vrcholu. Použijeme při tom určitě zpětnou hranu rodiče a možná i nějakých dalších vrcholů, ale o těch víme, 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.

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`.

Vzorový program následuje na další straně.

```
-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, []).
```

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 čísti sekvenčně.

Vojtěch 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 $f(v) = \max\{f(u)|u$ 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,[6] 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 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.

[6] `http://en.wikipedia.org/wiki/Strongly_connected_component`

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 $(0,\ 1,\ 2,\ \ldots,\ B-1)$.

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 $k-p \equiv$ $\equiv k \pmod B$[7]. 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 $0\le C\le x_1x_2\ldots x_k-1$.

---

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

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 $\mathcal{O}(HS)$.

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 $S \bmod H$, 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:

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

('≈' 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 anglicky – 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 šírky 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[8]) 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!

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.

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

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.

```
-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]).
```

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, která, 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 přidělíme.

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.

```
-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} ->
    % Přidáme 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}.
```

22-3-1 Falešná mince

Nad úlohou budeme uvažovat trochu pozpátku. Co nám asi může říct lékárník? Z každého vážení mohou přijít tři různé výstupy. Buďto se váhy naklonily vlevo, nebo vpravo, nebo zůstaly v rovnováze. Tedy máme celkem 3^K možných odpovědí pro K vážení. Kdybychom věděli, že mince je BÚNO (bez újmy na obecnosti) lehčí, pak máme naprosto triviální situaci. Podíváme se, která mince byla na všech miskách, které byly prohlášeny za „lehčí“, a nebyla na žádných jiných. Umíme zvážit 3^K mincí a finito.

Nicméně my víme jen to, že mince je různé hmotnosti, a tedy musíme uvážit obě možnosti. Nalezneme minci, která byla buďto na všech lehčích, nebo na všech těžších miskách a při rovnovážných váženích ležela stranou. Tedy sestavíme takovou sadu vážení, aby se toto dalo jednoznačně určit. Uvědomme si však, jak se liší výsledky pro stejnou minci falešnou, ovšem jednou lehčí a jednou těžší – výsledek každého vážení se prostě obrátí (a vznikne *inverzní výsledek*) s výjimkou případu, kdy byly váhy vždy v rovnováze, ten je inverzní sám sobě – a tedy můžeme určit maximální počet mincí, které umíme zvážit K váženími, na $(3^K + 1)/2$.

Každé minci tedy můžeme přiřadit jeden řetězec znaků `<=>`, znamenajících „levá strana je lehčí“, „váhy jsou v rovnováze“, „pravá strana je lehčí“. Z každého můžeme získat jemu inverzní vzájemným nahrazením znaků `<>`.

Takový řetězec také říká, jak ve kterém vážení mince figuruje. Jsou-li váhy v rovnováze, zjevně na nich falešná mince zrovna není, jinak se nachází na jedné z misek. Uvažme tedy i-té vážení: Existuje právě $2 \cdot 3^{K-1}$ různých řetězců délky K majících na i-tém místě `<` nebo `>`. Z nich ale právě polovinu vyškrtáme – jsou v nich totiž samé dvojice *duálních řetězců.* Takže na dvě misky v i-tém vážení potřebujeme rozmístit 3^{K-1} mincí, to ale není možné (neumíme nedestruktivně rozdělit lichý počet mincí na poloviny) – jeden řetězec zjevně nevyužijeme, a tedy neumíme zvážit $(3^K + 1)/2$ mincí, ale jen $(3^K - 1)/2$ mincí.

Nyní přichází nejtěžší úkol – jak zkonstruovat rozložení mincí na váhy. Pro $N = 1$ si můžeme být jisti, že ta mince, kterou držíme v ruce, je falešná. Pro $N = 2$ to naopak určit vůbec nelze. Pro ostatní N vytvoříme N řetězců délky K ze znaků `<=>` tak, že pro každou pozici $i \in \{1 \ldots K\}$ bude platit, že existuje stejný počet řetězců majících na i-té pozici `<` jako počet řetězců majících na i-té pozici `>` (dále označuji jako *Podmínka*). Pak jednoduše rozložím mince na váhy při jednotlivých váženích tak, že při `<` položím minci na levou misku, při `>` na pravou a při `=` odložím stranou.

Příklad: vážení pro $N = 4$ se dá zapsat jako `1--2, 2--3`, ale také jako $M_N = \{$`<=, ><, =>, ==`$\}$, což pak přečteme jako předpis: „První minci polož při prvním vážení na levou misku a při druhém ji odlož stranou; druhou minci polož při prvním vážení na pravou misku a při druhém na levou; třetí minci

nejdříve odlož stranou a při druhém vážení polož na pravou misku a čtvrtá mince nechť se vah ani nedotkne.“

Zkonstruujme nejprve rozložení pro $N = N_K = (3^K - 1)/2$ a z nich potom všechna ostatní rozložení. To uděláme rekurentně – konstrukcí z předchozího. Pro $K = 2$, $N_2 = 4$ máme předchozí příklad. Všimněme si, že v něm není řetězec <<. Lze jednoduše ukázat, že $N_{K+1} = 3N_K + 1$. Vezměme tedy množinu M_{N_K}, odstraníme z ní řetězec ==...= a do množiny $M_{N_{K+1}}$ ji vložíme třikrát – jednou ke všem řetězcům přidáme na začátek <, jednou > a jednou =. Chybí nám ještě 4 řetězce: trojice =>>>...>, ><<<...<, <===...= a „odložená mince“ ===...==. Je jednoduše ověřitelné, že množina $M_{N_{K+1}}$ splňuje *Podmínku* a zároveň neobsahuje řetězec <<<<...<<.

Nakonec si ještě rozmyslete, že z každé takto zkonstruované množiny lze vyškrtnout správný počet řetězců, abychom získali M_N, pro všechna N s výjimkou $N = 2$. Jde totiž rozdělit M_{N_K} na trojice splňující *Podmínku*: Označíme-li v $M_{N_{K-1}}$ nějakou existující trojici řetězců jako A, B, C, tak v M_{N_K} máme následující trojice: (<A, =B, >C), (=A, >B, <C), (>A, <B, =C). Zbytek je jedna trojice a „odložená mince“.

Tedy pokud potřebuju M_N pro $N = N_K - 3\varphi$, tak odstraním φ trojic, pro $N = N_K - (3\varphi + 1)$ odstraním φ trojic a „odloženou minci“. Zbývá podlý trik podle Mirka Olšáka (díky!) pro $N = N_K - (3\varphi + 2)$: Z řetězců =<<<<...<<=, <>>>...>>=, =<<<...<<< udělám řetězec <<<<...<<, který v M_{N_K} nebyl, čímž jsem se zbavil dvou řetězců, a následně ještě odstraním φ dalších trojic.

A tedy počet vážení, které potřebujeme k určení falešné mince z množiny N mincí, je roven $K = \lceil \log_3(2N) \rceil$.

Jan „Moskyt“ Matějka

22-3-2 Dětská hra

Jednalo se o grafovou úlohu. Děti představují vrcholy, a pokud má dítě u chytat dítě v, pak existuje hrana (u, v). Dítě d se může dostat do pozice, kdy by muselo chytat samo sebe, pouze v případě, že vrchol d leží na kružnici. Z každého vrcholu vede právě jedna hrana, takže počet vrcholů (n) je roven počtu hran (m). Díky tomu víme, že v každé komponentě je právě jeden cyklus. Pokud $m = n - 1$, tak je graf stromem (každý vrchol kromě kořene je zavěšen jednou hranou ke svému předkovi). Pokud přidáme n-tou hranu, tak nám vznikne kružnice.

Kružnice budeme hledat prohledáváním do hloubky. V každé fázi si vybereme vrchol x, který jsme ještě neprošli, a spouštíme prohledávání z něj. U každého vrcholu si značíme „čas“ prvního příchodu (např. číslo fáze) $in(vrchol)$. Postupujeme po hranách, dokud nenarazíme na hranu (u, v), která vede do prozkoumaného vrcholu v. Pokud je čas příchodu do v menší než čas příchodu

do x, tak jsme se jen dostali k již prozkoumané části grafu, v opačném případě jsme nalezli kružnici délky $in(u) + 1 - in(v)$. Poznačíme si délku nalezeného cyklu a pokračujeme další fází, dokud existují neprozkoumané vrcholy. Nakonec jsou všechny vrcholy navštívené a výsledný počet dětí, které mohou chytat samy sebe, je součtem nalezených cyklů.

Při procházení navštívíme každý vrchol právě jednou a ještě jedenkrát se do něj můžeme podívat, pokud leží na cyklu, anebo na cestě, kterou jsme začali procházet až od něj. Časová složitost tedy je $\mathcal{O}(n)$, paměťová složitost je taktéž $\mathcal{O}(n)$ – pro každý vrchol si pamatujeme jeho následníka.

David Marek

22-3-3 Kurýrní služba

Na první pohled je vidět, že úloha je grafová, města tvoří vrcholy a kurýři orientované grafy (komu nic tyto pojmy neříkají, doporučuji přečíst kuchařku o grafech na našich stránkách). Počet měst (vrcholů) si označíme N a počet kurýrů (hran) M. O husitském orientovaném grafu chceme zjistit, zdali existuje cesta mezi každými dvěma vrcholy alespoň jedním směrem, tedy z A do B nebo z B do A pro každé dva vrcholy A a B. Takový graf se nazývá polosouvislý.

To bude určitě Floyd-Warshallův algoritmus, zazní v hlavě první návrh. Ten přece počítá nejkratší cestu v orientovaném grafu mezi všemi vrcholy. Používá dvourozměrné pole velikosti $N \times N$, přičemž prvek na pozici $[i, j]$ obsahuje nejkratší cestu mezi vrcholy i a j. Na začátku jsou všechny prvky inicializovány „nekonečnem“ nebo ohodnocením hrany a v N krocích se vylepšuje odhad na délku nejkratší cesty. Pro podrobnější popis odkazuji opět na naše kuchařky na webu, konkrétně na dynamické programování. Jen dodám, že tento algoritmus má časovou složitost $\mathcal{O}(N^3)$ a paměťovou $\mathcal{O}(N^2)$.

S časem $\mathcal{O}(N^3)$ by však mohli husiti prohrát válku, než by zjistili, že se mezi městy nedají posílat zprávy – mají pomalé dřevěné počítače a spoustu dobytých měst. Navíc nepotřebujeme nutně počítat nejkratší cestu, ani neznáme ohodnocení hran (vzdálenost mezi městy). A zatřetí, dávali bychom pouze za použití algoritmu z kuchařky 13 bodů? Zkusíme tedy postupovat lépe.

Nejprve si dokážeme, že v polosouvislém grafu musí existovat sled procházející všemi vrcholy (sled je cesta, ve které se mohou opakovat vrcholy i hrany). Kdyby totiž neobsahoval všechny vrcholy, vezmeme vrchol V, jenž v něm neleží. Aby byl graf polosouvislý, musí pro každý vrchol U ze sledu existovat buď cesta z U do V nebo z V do U. Podle toho, jestli z vrcholu existuje cesta z nebo do V, si rozdělíme vrcholy na dvě skupiny (tyto skupiny mohou mít nějaký překryv) a seřadíme je podle toho, jak leží na sledu. Potom ovšem můžeme V přidat do sledu mezi poslední vrchol, z něhož se lze dostat do V, a první vrchol, do něhož vede cesta z V, anebo kamkoliv do překryvu těchto dvou skupin.

Aby se nám sled lépe hledal, odstraníme si z grafu orientované cykly, respektive sloučíme je do jednoho vrcholu. Jinými slovy, najdeme silně souvislé komponenty (SSK), což jsou maximální podgrafy, ve kterých mezi každými dvěma vrcholy vede orientovaná cesta oběma směry. Pro představu, SSK mohou tvořit jednotlivé cykly, soustavy více spojených cyklů, ale i samotné vrcholy. Jak je vidět, v nich skutečně nemusíme zjišťovat, jestli existuje cesta mezi každými dvěma vrcholy. Netřeba vymýšlet kolo, na nalezení silně souvislých komponent se používá Kusarajův či Tarjanův algoritmus. My si zde popíšeme Tarjanův, jenž najdete i na anglické Wikipedii pod heslem *Tarjan's strongly connected components algorithm*.

Tarjanův algoritmus je modifikací prohledávání do hloubky. Oproti standardnímu prohledávání si navíc budeme vrcholy číslovat podle toho, kdy do nich vstoupíme (seřadíme je prohledáváním do hloubky), a při zpětném průchodu hledáme, do jakého vrcholu s co nejnižším číslem se můžeme dostat. Je-li to nejnižší nalezené číslo rovno číslu vrcholu, našli jsme cyklus, a tedy SSK. Všimněte si, že každou SSK zaznamenáme právě jednou, protože všechny její vrcholy obdrží stejné číslo – to nejnižší z celé SSK – a rovnost tedy nastane jen u jednoho vrcholu. Tento vrchol si nazveme kořenem SSK.

Jak zjistit, jaký potomek vrcholu má nejnižší číslo? Jednoduše, stačí se podívat, do jakého nejnižšího čísla se dostali potomci vrcholu, do nichž z něj vede hrana. Pokud narazíme na cyklus, a tedy na vrchol, u něhož dosud tuto hodnotu neznáme, vezmeme prostě jeho číslo (což se dá zařídit lehce – na začátku bude mít každý vrchol inicializován nejnižšího potomka svým číslem).

Díky nalezení SSK v kořeni (vrcholu s nejnižším číslem v SSK) máme jistotu, že jsme získali maximální SSK, tedy že k ní už nelze žádný vrchol přidat, aby zůstala silně souvislá. Pro určení, které vrcholy náleží do nalezené SSK, použijeme zásobník, do nehož přidáváme vrcholy při vstupu do nich. Po objevení SSK se vrcholy ze zásobníku odebírají, dokud se nenarazí na kořen.

Máme tedy silně souvislé komponenty. Co s nimi? Sloučíme je do jednoho vrcholu a vytvoříme si nový, acyklický graf, tzv. kondenzaci původního grafu. V tomto grafu již stačí otestovat, jestli v něm existuje cesta obsahující všechny vrcholy (důvod je podobný tomu, že v původním grafu je sled se všemi vrcholy). Nový graf se ale nemusí skládat jen z cesty, může obsahovat tzv. dopředné hrany, jež vedou z nějakého vrcholu A do jiného vrcholu, jenž je na cestě dále než A. Test, který se pokusí takovou cestu najít a rozhodne o výsledku, může být například nalezení vrcholu, do nějž nevede hrana (ten existuje, máme acyklický graf a musí být právě jeden), a průchod do šířky, při kterém si budeme pamatovat, přes kolik vrcholů jsme již přešli.

Jaká je časová a paměťová složitost algoritmu? Jak již bylo řečeno, Tarjanův algoritmus je modifikované prohledávání do hloubky. Má-li tedy operace zjištění, jestli je prvek v zásobníku, konstantní časovou složitost (což se dá za-

řídit polem, jehož prvek na místě i určuje, jestli je i-té město v zásobníku), složitost Tarjanova algoritmu vyjde $\mathcal{O}(N+M)$. Na následné nalezení vrcholu, do kterého nevede hrana, spotřebujeme opět řádově $N+M$ operací. Složitost prohledávání do šířky nám už hezkou lineární časovou složitost $\mathcal{O}(N+M)$ nezkazí. Asymptoticky rychleji úlohu určitě nevyřešíme, nepřečetli bychom ani celý vstup. Algoritmus zabere také jen $\mathcal{O}(N+M)$ paměti, takže jsme zachránili husity s pomalými dřevěnými počítači.

Jiné řešení

Celkem elegantní řešení poslal Mirek Olšák. Všiml si, že stačí modifikovat Tarjanův algoritmus, takže si vystačíme pouze s jedním prohledáváním do hloubky. Použijeme následující pozorování: když se vracíme při prohledávání acyklického grafu, jejž nám vyrobil Tarjanův algoritmus, z nějakého vrcholu, musí z něj vést hrana do vrcholu, z něhož jsme se vraceli předtím (pro lepší představu si zkuste na chvíli vypustit z grafu cykly). Při návratu z vrcholu tedy uložíme vrchol do nějaké globální proměnné a v každém vrcholu zkontrolujeme, jestli do něj vede hrana. Že graf není polosouvislý, objeví tento algoritmus ve vrcholech, z nichž nevede hrana do žádného dosud neprošlého vrcholu, zkuste si rozmyslet proč.

Nejlépe půjde algoritmus pochopit asi z následujícího pseudokódu vycházejícího z implementace Tarjanova algoritmu na Wikipedii:

```
vstup: graf G = (V, E)
// V je množina vrcholů a E množina hran
index = 0
// proměnná sloužící k číslování vrcholů
S = empty   // prázdný zásobník pro vrcholy
posledni = undefined
// poslední vrchol, z něhož jsme se vrátili
forall v in V do
  if (v.index == undefined)
    // prohledej do hloubky vrcholy,
    // které ještě nebyly navštíveny
    tarjan(v)
print "Hurá! Graf je polosouvislý."

procedure tarjan(v)
  v.index = index  // očíslování vrcholu
  // inicializace nejnižšího dosaženého vrcholu
  v.lowlink = index
  index = index + 1
  nalezen_posledni = false
  if (posledni == undefined)
    //zatím jsme se ze žádného vrcholu nevraceli
    nalezen_posledni = true
  S.push(v)  // přidej vrchol do zásobníku
  // projdi všechny následníky vrcholu
```

```
    forall (v, v') in E do
      // pokud není vrchol navštíven
      if (v'.index == undefined)
          tarjan(v') // prohledej ho do hloubky
          // urči vrchol s nejnižším číslem
          v.lowlink = min(v.lowlink, v'.lowlink)
      else if (v' is in S)
          // vrchol je v zásobníku
          v.lowlink = min(v.lowlink, v'.index)
      // vede hrana do vrcholu, z něhož
      // jsme se naposled vraceli?
      if (posledni == v')
         nalezen_posledni = true
    // konec for cyklu
    if (nalezen_posledni == false)
      print "Graf není polosouvislý."
      exit
    if (v.lowlink == v.index) // nalezena SSK
      // odeber všechny vrcholy SSK ze zásobníku
      repeat
        v' = S.pop
        // nejhlouběji v zásobníku je kořen SSK
      until (v' == v)
    posledni = v
```

Pavel „Paulie" Veselý

22-3-4 Rukavice

Trošku nás mrzelo, jak málo jste se snažili dokazovat správnost svých řešení. Není vůbec těžké na nějakou strategii přijít, ale vrací taková opravdu požadované, tj. vzhledem k L a P minimální výsledky? K nočním můrám opravovatelů KSP patří divná, složitá a odvážná řešení, která skoro určitě nefungují, ale je třeba přijít na potřebný protipříklad – v případě této úlohy byla ale většina špatných řešení vyvratitelná krátkým kritickým náhledem, kterého byste měli být schopni i sami. Nebojte se nám napsat, že slabinu svého postupu znáte: chápeme, že na dobré řešení není snadné přijít.

Jednoduchý, nesprávný, ale slibný pohled vypadá takto: pokud bych měl jistotu, že z levé truhly vytáhnu od každého páru alespoň jednu rukavici, mohlo by mi stačit z pravé bedny vytáhnout libovolnou. Takovou jistotu však nemohu získat nikdy, protože mi nic nezaručuje, že neexistují pravé rukavice bez levého ekvivalentu (bezlevé): proto budu chtít zajistit, abych z levé bedny vytáhnul alespoň jednu rukavici od každé zastoupené barvy a z pravé vytáhnu tolik rukavic, abych měl jistotu, že mezi nimi bude nějaká, která má v levé bedně souputníka.

Kolik tedy? Inu, pokud necháme Tomáše z pravé bedny vytáhnout tolik rukavic, kolik je tam bezlevých a ještě jednu navíc, nemůže se ani v tom nejhorším případě stát, že by Tomášův výběr obsahoval samé bezlevé pravé rukavice. Podobně pokud v levé bedně vybereme tolik rukavic, kolik jich je tam celkem bez počtu nejméně zastoupeného druhu plus jednu navíc, určitě se nám nemůže stát, že by se v takovém výběru nevyskytovala libovolná varianta. (Rozmyslete si! Proč to musí být „nejméně zastoupeného druhu“?)

Nyní nás může napadnout, že při nahrazení levé bedny za pravou a naopak by nám tento postup mohl vrátit lepší výsledek, kupříkladu kdyby napravo bylo velmi mnoho bezlevých rukavic. Je propočítání obou variant a navrácení té lepší správné řešení?

Ne.

(Přestaňme nyní uvažovat bezlevé a bezpravé rukavice – obecně jejich výskyty stejně nemůžeme řešit jinak, než že jejich počet přičteme k počtu rukavic k vytáhnutí.)

Rozhodli jsme se z jedné bedny vytáhnout zaručeně všechny barvy a z druhé zaručeně jednu, z čehož jsme si logicky odvodili, že získáme jednobarevnou dvojici. Co kdybychom chtěli z levé bedny zaručeně vytáhnout k různých barev a z pravé $n-k+1$? Dirichletův princip praví, že i pak bychom měli zaručeno, že vytáhneme alespoň jeden stejný pár. Může se pro nějaké k stát, že dosáhneme lepšího výsledku než v krajních případech $k = 0$, $k = n$? (Uvědomte si, že to jsou přesně dvě možnosti zvažované v odstavci před „Ne.“)

Nejdříve: jak zajistíme vytáhnutí k barev? Stačí vytáhnout počet všech rukavic bez $n - k + 1$ nejméně častých plus jednu navíc, podobně jako jsme to dělali výše. Teď si je třeba rozmyslet, jaký rozdíl v počtu tažených rukavic způsobí přechod od nějakého k ke $k + 1$: z levé bedny budeme muset vytáhnout navíc rukavice $(n - k + 1)$-ní nejméně zastoupené barvy, z pravé naopak nebudeme muset táhnout rukavice k-té nejméně zastoupené barvy. V obecném případě nevíme nic o tom, jestli si tím polepšíme nebo pohoršíme, musíme tedy získat minimum přes všechna k.

No a triviální postup, jak ho získat, je seřadit si počty rukavic jednotlivých barev a pak k projít cyklem. Trvat to bude $\mathcal{O}(n \log n)$, místa zabereme $\mathcal{O}(n)$. Detaily najdete v autorském zdrojáku.

Je to tak správně? Jde si snadno představit, že nastavili-li bychom pro určité množství L rukavic P menší, než jak to děláme, tj. takové, pro které nám Dirichlet už nezaručuje, že vytáhneme dvě opačné rukavice stejné barvy, šel by sestavit výběr rukavic, který skutečně takovou dvojici neobsahuje. Teď je otázka, zda neexistuje L, které jsme nezkoušeli a které dává lepší řešení: počet všech levých rukavic je ostře větší než n, takže jsme těch propočtů přešli docela hodně.

A vtip je v tom, že pro L, které $L_k < L < L_{k+1}$ (kde L_k odpovídá počtu levých rukavic, které musíme vytáhnout pro dané k), bude $P = P_k$, protože jsme oněch $L - L_k$ rukavic vytáhli nadarmo: nezaručili jsme jimi vytažení většího množství barev.

Mária Vámošová & Lukáš Lánský

22-3-5 Reklama

Na začiatku riešenia si dopomôžeme menším trikom. Celú situáciu bodov v rovine otočíme o 45 stupňov proti smeru hodinových ručičiek. To spravíme jednoducho tak, že nahradíme $x' = x + y$ a $y' = x - y$, a ďalej budeme pracovať už len s týmito transformovanými súradnicami. Teraz platí, že všetky body (x_1, y_1), ktoré môžu byť spojené s nejakým bodom na súradniciach (x_2, y_2), musia mať $x_1 \leq x_2$ a $y_1 \leq y_2$ alebo $x_1 \geq x_2$ a $y_1 \geq y_2$.

Najskôr sa zamyslime, čo to znamená, nakresliť čo najmenej čiar, aby obsahovali všetky body. Ak si predstavíme čiaru ako postupnosť úsekov spájajúcich dva body, tak to znamená, že minimum sa dosiahne práve vtedy, keď je celkový počet úsekov všetkých čiar dohromady maximálny.

Jeden úsek čiary je teda spojenie dvoch rôznych bodov $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ také, že $x_1 \leq x_2$ a $y_1 \leq y_2$. Takéto usporiadané dvojice (p_1, p_2) bodov budeme ďalej volať pár a množinu párov, v ktorých každý bod vystupuje maximálne 2 krát (raz ako p_1 a raz ako p_2), budeme volať párovanie, ktorého veľkosť sa snažíme maximalizovať.

Na ďalšie uvažovanie si úlohu trocha preformulujme: Sú dané dve množiny bodov S a P, pričom S obsahuje kópie bodov, ktoré môžu vystupovať v párovaní ako body p_1 a P obsahuje kópie bodov, ktoré môžu vystupovať v párovaní ako body p_2.

Pochopiteľne, naša pôvodná úloha je špeciálnym typom tejto preformulovanej úlohy, keď $S = P = \{$ pôvodné body $\}$.

Ak si množiny S a P predstavíme ako partity bipartitného grafu, kde každý korektný pár reprezentujeme hranou medzi príslušnými dvoma vrcholmi v S a P, potom veľkosť najväčšieho párovania vieme nájsť pomocou obecného algoritmu na hľadanie maximálneho bipartitného párovania. O tomto algoritme si môžete prečítať viac na anglickej Wikipedii[9] alebo na stránkach Martina Mareše[10] a za jeho použitie ste mohli získať maximálne 10 bodov.

K lepšiemu algoritmu bolo nutné učiniť dôležité pozorovanie: označme bod $p = (x_0, y_0) \in P$ ako bod množiny P s najväčšou y-ovou súradnicou (a spomedzi tých s najmenšou x-ovou súradnicou). A skúmajme, s ktorými bodmi v S môže tvoriť pár v nejakom optimálnom (maximálnom) párovaní.

[9] `http://en.wikipedia.org/wiki/Maximum_bipartite_matching`

[10] `http://mj.ucw.cz/vyuka/ga/`

Pre všetky body, s ktorými môže tvoriť pár, platí $x \leq x_0$ a $y \leq y_0$. Keďže p je bod s maximálnou y-ovou súradnicou, dostávame, že to môže byť ľubovoľný bod, spĺňajúci podmienku $x \leq x_0$. Ak žiaden takýto bod neexistuje, potom je samozrejmé, že v žiadnom optimálnom riešení nie je tento bod spárovaný. Inak označme bodom $q = (x_1, x_2) \in S$ bod s maximálnou y-ovou súradnicou spĺňajúci $x_1 \leq x_0$, a ak je takých viac, tak spomedzi tých bod s najväčšou x-ovou súradnicou.

Ukážeme, že existuje optimálne riešenie, keď je bod p spárovaný s bodom q.

Nech existuje ľubovoľné optimálne riešenie, v ktorom to neplatí. Potom môžu v tomto párovaní nastať len tieto situácie:

- Bod p je spárovaný s nejakým iným bodom $q' \neq q \in S$ a bod q je bez páru alebo q je spárovaný s nejakým iným bodom $(p' \in P) \neq p$ a bod p je bez páru. To však znamená, že môžeme spárovať p s q a q' (resp. p') nechať bez páru, čím párovanie nezmenšíme.
- Bod p je spárovaný s nejakým iným bodom $q' \neq q$ a bod q je spárovaný s nejakým iným bodom $p' \neq p$. Keďže však platí, že bod q je bod s najväčšou y-ovou súradnicou (a prípadne najväčšou x-ovou), pre ktorý platí $x_1 \leq x_0$, potom pre bod $p' = (x_2, y_2)$ platí $x_2 > x_0$ a $y_2 \geq y1$. Rovnako pre bod $q' = (x_3, y_3)$ platí $x_3 \leq x_1$ a $y_3 \leq y_1$. Teda môžeme spárovať bod p s q a bod p' s q', čím párovanie nezmenšíme.
- Prípad, keď v optimálnom riešení je bod p aj q bez páru, nemôže nastať, lebo spárovaním vieme dostať párovanie o 1 väčšie.

Vidíme, že nič nepokazíme, keď bod p spárujeme s bodom q. Vykonaním tohto kroku sme si vlastne zredukovali náš problém veľkosti N na problém veľkosti $N-1$. Pretože bod p už nemôže vystupovať v žiadnom párovaní, môžeme ho z množiny P odstrániť a rovnako bod q odstrániť z množiny S. A na nový problém použijeme rovnaký algoritmus.

Jednoduchým aplikovaním uvedeného postupu, keď N-krát opakovane nájdeme bod s maximálnou súradnicou y v čase $\mathcal{O}(N)$ a k nemu príslušný bod q tiež v $\mathcal{O}(N)$, dostávame kvadratický algoritmus, za ktorý ste mohli získať 12 bodov.

Na finálny vzorový algoritmus bolo nutné trocha zmeniť pohľad na úlohu.

Začneme body v množine P prechádzať v poradí rastúcej x-ovej súradnice (a body s rovnakým x podľa y-ovej súradnice). Pričom vždy, keď navštívime bod, určíme, s ktorým bodom v množine S bude spárovaný. Za týmto účelom si budeme pamätať množinu $Q \subseteq S$ doteraz nespárovaných bodov.

Budeme sa pri tom riadiť pravidlom, že aktuálny bod $p = (x_0, y_0) \in P$ spárujeme s bodom $q = (x_1, y_1) \in Q$, ktorý má zo všetkých bodov v Q najväčšiu y-súradnicu, avšak menšiu ako y_0. Ak takýto bod neexistuje, tak bod p zostane bez páru. Následne odstránime z množiny Q bod q a zaradíme tam bod p.

Treba však ukázať, že takýto algoritmus vedie k optimálnemu riešeniu, teda najmenšiemu počtu nutných čiar.

Pri tomto spracovávaní platí invariant, že v každom kroku existuje optimálne párovanie, ktoré páruje spracované body P rovnako, ako sme to spravili my.

Ak takýto invariant platí v kroku n, potom spárovanie bodu p s bodom q nám takýto invariant nepokazí a bude platiť aj v kroku $n+1$. Pretože v tomto optimálnom riešení, ktoré spárovalo prvých n bodov množiny rovnako ako my, môžu nastať len situácie rovnaké ako rozoberané situácie v našom kvadratickom algoritme. Teda spárovaním p s q zachováme invariant o existencii optimálneho riešenia.

Algoritmus sa dá efektívne implementovať tak, že na začiatku si utriedime body P v čase $\mathcal{O}(N \log N)$ a pri spracúvaní si budeme množinu Q udržiavať v utriedenom poradí podľa súradnice y, napr. pomocou vyvažovaného binárneho stromu, čím dokážeme v čase $\mathcal{O}(\log N)$ hľadať v tejto množine bod, ktorý má najväčšiu súradnicu menšiu ako dané y_0 a v tomto čase tiež vymazať bod q s Q a vložiť tam bod p.

Šikovnejšiu implementáciu dostaneme, ak si všimneme, že bod p sa vkladá na rovnaké miesto v utriedenej množine Q, z ktorého bol vymazaný bod q (ak existoval). Alebo ak neexistoval, tak vloženie sa uskutočňuje vždy na koniec, čím sa nám ponúka jednoduchá možnosť udržiavať si množinu Q v poli a na ňom hľadať požadovaný prvok binárnym vyhľadávaním.

Pamäťová zložitosť je $\mathcal{O}(N)$, stačí si nám pamätať len prvky P a množinu Q.

Poučenie na záver: Táto úloha spadá do kategórie „Greedy algoritmy“, alebo tiež „hladové“. V týchto úlohách platí, že v každom kroku máme možnosť vykonať operáciu, ktorá nám zaručene nezabráni nájsť optimálne riešenie, čo implikuje, že ak takú operáciu spravíme v každom kroku, dostaneme optimálne riešenie,

V tomto prípade to bola operácia spárovania bodu s maximálnou y-ovou súradnicou.

Pri týchto úlohách je však vždy nutné overiť, či náš krok je skutočne tým, ktorý nám našu cestu k optimálnemu riešeniu neodstrihne a či máme možnosť nejakú takúto operáciu spraviť v každom kroku.

Peter Ondrúška

22-3-6 Kolejní výtahy

V této sérii byla leckterá úloha pořádně vypečená, jednoduchost jsme skryli právě do praktické úložky.

Ochotu jednotlivých studentů, kteří chtějí vyjet nahoru výtahem, můžeme chápat jako intervaly na celočíselné ose od 1 do n, kde n je výška kolejí. Některé

intervaly se mohou překrývat a naším cílem je vybrat takovou množínu čísel, že pokryjeme všechny intervaly. Počet intervalů označme k.

Pojďme na to od lesa. Tedy ... od začátku. První student (tedy ten, který je ochoten vystoupit v nejnižším patře) bude muset někde vystoupit, ale může zastavit tak, aby pomohl více lidem. Moc nového jsme se nedozvěděli. Co když se ale podíváme na prvního studenta, který *musí vystoupit* – jehož konec intervalu je ze všech konců nejblíže 1. Víme, že tohoto studenta musíme někde z výtahu vystrčit, ale navíc víme, že to klidně můžeme učinit právě na tomto místě.

Můžeme to udělat proto, neboť jsme ho chytře vybrali – může se stát, že jeho interval protíná intervaly jiných studentů, avšak žádný z těch, kdo s jeho intervalem ochoty mají průnik, ještě nemusí vystupovat – jinak řečeno, všichni takoví mohou vystoupit právě v tomto patře.

Necháme tedy z výtahu odejít všechny studenty, kteří jsou na konci intervalu prvního studenta ochotni. A máme základ algoritmu! Zbytek dořešíme obdobně – nalezneme dosud neprobraného studenta, jehož konec intervalu je nejblíže poslednímu místu, a necháme s ním vystoupit všechny ostatní, kteří jsou v tom patře ochotni.

Pokud na začátku algoritmu data setřídíme podle konců intervalů (v čase $\mathcal{O}(k \log k)$), zbytek dořešíme se složitostí, $\mathcal{O}(k)$ a celkově to tedy stihneme v čase $\mathcal{O}(k \log k)$.

Máme za sebou složitost, ale je algoritmus správně? Postupem výše zmíněným určitě dojdeme k nějaké korektní posloupnosti zastávek, nemusí nám být ale zcela jasné, že taková posloupnost je minimální. Podívejme se na posloupnost intervalů, které jsme vždy vybrali jako další „koncové“, a označme její velikost p. Protože jsme na každém konci poslali z výtahu všechny studenty, kteří vystoupit mohli, tyto intervaly jsou disjunktní. Tím pádem každé přípustné rozložení zastávek musí zastavit alespoň p-krát – na každém intervalu alespoň jednou. Nícméně p je právě počet zastávek, které vykonal náš algoritmus, a výsledek je tedy vskutku optimální. Howgh.

Tím skončila indiánská část řešení. Ještě zmíníme, že pokud si nakreslíme jednotlivé intervaly na reálnou osu a budeme se na tuto strukturu dívat jako na graf (intervaly = vrcholy a dva vrcholy jsou spojeny hranou, protínají-li se dva intervaly), dostaneme speciální typ grafu, jménem „průnikový graf“. Takovéto grafy (nejen přímkové) jsou často studovány v teorii grafů a již se o nich leccos ví, například že leckteré „těžké“ problémy pro obecné grafy na nich lze vyřešit v polynomiálním čase.

Martin Böhm

22-3-7 Pavouci internetu

Přehled

V našem vzorovém řešení je situace mírně komplikovaná – dohadují se mezi sebou 3 druhy procesů. Podívejme se tedy napřed na jednodušší situaci, kdy nic nepadá. Jak bude vypadat pracovní cyklus?

Klient si vybere server a zadá mu práci, počká na potvrzení o jejím přijetí. Někdy mezi tím se někde jinde objeví pracující proces a připojí se také na server. Když se tedy na serveru sejdou oba, server je seznámí – předá práci pracujícímu procesu, společně s PID toho, kdo ji zadal. Více se o ně nestará. Pracující provede zadaný úkol, odešle výsledek (přímo zadávajícímu, nikoliv přes server) a znovu se přihlásí na server. Mezitím server seznamuje další pracující s prací. Zatím jednoduché?

Jak je ale řešené, když se pracující přihlásí na jiný server, než na který je uložen úkol? Servery mezi sebou komunikují také. Ve chvíli, kdy na některém serveru dojde k přebytku libovolného druhu (přebývají buď pracující a nebo úkoly), oznámí to server všem ostatním. Ti si buď řeknou o přesun pracujícího, nebo nabídnou svého pracujícího. Tomu je poté sděleno, aby se připojil na onen jiný server.

Jak budeme řešit havárie? Podívejme se na jednotlivé části podrobněji.

Zadávání

Když chceme zadat úkol, spustíme nový proces. Jeho úkolem bude sledovat, co se s úkolem kde děje.

Ten se pokusí odeslat práci prvnímu serveru z nastavení. Bohužel, ten nemusí běžet, proto čekáme na odpověď jen nějakou dobu. Pokud ale běží, linkne si náš proces a příjem potvrdí.

Ve chvíli, kdy máme potvrzení o přijetí, čekáme na přiřazení. Až server najde vhodného pracujícího, dá nám o tom vědět (a také jeho PID). My si ho linkneme a budeme čekat na odpověď od něj (a serveru už si nebudeme všímat, co se s ním děje, je nyní nezajímavé).

Když přijde výsledek, vše proběhlo, jak mělo, my můžeme poslat výsledek do původního procesu a spokojeně skončit.

Pokud se nedočkáme odpovědi od serveru, zkusíme další server v seznamu a s ním provádíme totéž. Jestliže selže cokoliv dalšího (server či pracující umře, když čekáme na něco od něj), tak začneme zcela od začátku – od prvního serveru a v novém procesu. (Kdo si má pamatovat, co všechno by bylo potřeba unlinknout? Navíc, pokud by některý server žil, ale nestihl odpovědět včas, pak by měl uloženou naši práci – skončením ji u něj zrušíme.)

Pracující

Každý pracující má dvě vývojová stádia. První stádium je čekající. Podobným způsobem jako klient se pokusíme spojit se serverem (tedy, pošleme

mu své PID a on buď odpoví, že nás bere, nebo, když se nedočkáme odpovědi, zkusíme další server). Až nás některý přijme, tak si nás linkne a my jen budeme poklidně čekat, až nám přidělí nějakou práci.

Až nějakou dostaneme, tak si nás unlinkne server, ale linkne si nás klient. My práci pustíme, ale to uděláme v novém procesu (samozřejmě, také s námi slinkovaném) a čekáme na výsledek. Až skončí, pošleme výsledek klientovi, sami se pokusíme znovu přihlásit k serveru, ale to opět v novém procesu (abychom zničili všechny stará spojení).

Nyní, co se může stát? Když spadne proces s prací, který ale běží na stejném stroji, jako my, znamená to, že je v něm chyba. Nahlásíme to tedy klientovi a práci považujeme za splněnou. Obdobně když se nedočkáme výsledku v požadovaném čase (ale předtím proces s prací ukončíme).

Pokud spadne buď server nebo klient, pak jen ukončíme případnou probíhající práci a zkusíme se opět na některý server připojit. To je zcela v pořádku – pokud spadl server (a my se o tom dozvěděli), pak ještě nemáme žádnou práci a nic se neztratí. Jediný případ, kdy zničíme nějakou práci, je, když umře klient, ale v tom případě není komu poslat výsledek, je tedy zbytečná.

A co když umřeme my? Buď se tak stalo ještě před připojením na server a v tom případě to nezpůsobí žádnou škodu. A poté až do dokončení práce je s námi vždy někdo slinkovaný, takže se o tom dozví a může provést nápravná opatření (v případě serveru si nás jen smazat, v případě klienta zadat práci znovu, někomu živějšímu).

Server

Nyní, co dělá server? Na chvíli si odmysleme poněkud krkolomné předávání pracujících procesů. V tom případě je celý server velmi jednoduchý. Sbírá úkoly a pracující procesy, ve chvíli, kdy se sejde od každého jeden, tak je spáruje a dál se o ně nestará.

Po dobu uložení úkolu i pracujícího procesu je s ním prolinkován. Ve chvíli, kdy druhý konec spadne, smaže si jej ze seznamu (nefungující pracující je k ničemu, práce pro neexistující proces je zbytečná). Když spáruje požadavek s pracujícím, tak je oba unlinkne – pro server jsou již nezajímaví, vyřídí si vše mezi sebou.

Co když spadne server? Potom se o tom dozvědí všichni klienti i všichni pracující. Pracující se jen přepojí na jiný server a klienti pošlou práci znovu, někam jinam.

Předávání pracujících

Není problém se dohodnout, že někdo jiný potřebuje pracujícího – stačí si navzájem posílat zprávy o tom, že se jedna z množin stala neprázdnou. Problémem je, jak pracujícího předat bezpečným způsobem. Aby se někde „neztratil cestou“, je potřeba, aby s ním vždy byl někdo prolinkovaný, a pokud cestou umře, říct si o nového.

Napřed se tedy dva servery, které si ho předávají, nalinkují spolu a dohodnou si předání. Přijímající server si nalinkuje pracujícího a potvrdí předávajícímu, ten ho v tuto chvíli může unlinkovat a říci mu, aby se přesunul. Pracující se připojí na nový server a přesun je hotový.

Co může spadnout? Inu, cokoliv. Když spadne pracující, dozví se o tom přijímající server (linkne si ho) a může požádat o nového. Když umře přijímající server při dohadování, dozví se o tom předávající. V tom případě nic neodešle (není komu). Nakonec, může umřít i předávající, o tom se ovšem dozví přijímající a nebude tedy čekat na pracujícího. (Kdyby náhodou přišel, tak se nic zlého nestane, přijmeme ho tak jako tak. Pokud nepřijde, stejně si od mrtvého serveru nemůžeme vyžádat nového.)

```
-module(klient).
-export([zadej/1, cekej/2]).

% Odešle požadavek na server a počká, jestli ho přijme.
% Pokud ne, zkusí další server.
% Po přijetí počká na první odpověď (je možné, že se původní
% přiřazení někde zdrželo).
cekejInterni(Funkce, Zadavatel, [Server|Nahradni]) ->
  % Řekne serveru, že chceme něco provést
  {server, Server} ! {prace, self(), Funkce},
  receive
    % Server práci přijal
    prijato -> receive
      % Server nám při tom umřel, zkusíme to jinde
      {'EXIT', Server, _} -> restart;
      % Bylo to přiřazeno nějakému pracantovi
      {prirazeno, PID} -> link(PID), receive
        % Umřel pracant (server už nás nezajímá)
        {'EXIT', PID, _} -> restart;
        % Hurá, máme výsledek, pošleme to majiteli a končíme
        {vysledek, Stav, Hodnota} -> Zadavatel !
                                     {vysledek, Stav, Hodnota}, PID ! ok
      end
    end
    % Server neodpovídá, zkusíme jiný
    after nastaveni:timeout_serveru() -> cekejInterni(Funkce, Zadavatel, Nahradni)
  end.

cekej(Funkce, Zadavatel) ->
  % Kdyz umře server (který si nás při přijetí linkne), tak o tom chceme vědět
  process_flag(trap_exit, true),
  % Zkus to vyřídit
  case cekejInterni(Funkce, Zadavatel, nastaveni:servery()) of
    % Umřel server, nový pokus
    restart -> cekej(Funkce, Zadavatel);
    % Vše v pořádku
    _ -> ok
```

```
  end.

% Zadá funkci ke zpracování.
zadej(Funkce) -> spawn(klient, cekej, [Funkce, self()]).
-module(nastaveni).
-export([servery/0, timeout_serveru/0, timeout/0]).

% Kde běží servery?
servery() -> [server@localhost, nahradni@localhost].
% Jak dlouho budeme čekat, než prohlásíme server za mrtvý a zkusíme jiný?
timeout_serveru() -> 1000.
% 5 minut bude trvat nejdelší povolený úkol
timeout() -> 300000.
-module(prace).
-export([start/0, start/1, pracuj/2, proved/2]).

servery() -> lists:map(fun (S) -> {server, S} end, nastaveni:servery()).

proved(Majitel, Fun) -> Majitel ! {vysledek, ok, Fun()}.

% Počká, až nám zadavatel dovolí skončit nebo skončí sám
pockej(Zadavatel) -> receive
  ok -> ok;
  {'EXIT', Zadavatel, _} -> ko
end.

pracuj(Zadavatel, Fun) ->
  process_flag(trap_exit, true),
  link(Zadavatel),
  % Spustíme výpočet
  Pracujici = spawn_link(prace, proved, [self(), Fun]),
  % A počkáme, jestli to vyjde, nebo spadne
  receive
    % Vyšlo, pošleme výsledek
    {vysledek, Stav, Hodnota} -> Zadavatel !
                               {vysledek, Stav, Hodnota}, pockej(Zadavatel);
    % Umřelo, pošleme hlášení o chybě
    {'EXIT', Pracujici, Chyba} -> Zadavatel !
                               {vysledek, chyba, Chyba}, pockej(Zadavatel);
    % Umřel zadavatel, zabij práci
    {'EXIT', Zadavatel, _} -> exit(Pracujici, kill)
    % Už běží moc dlouho, zabít
    after nastaveni:timeout() -> Zadavatel !
          {vysledek, timeout, nic}, exit(Pracujici, kill), pockej(Zadavatel)
  end,
  % Zase se přihlásíme o práci
  start().

pracuj() ->
  % Neumřeme, když umře server
  process_flag(trap_exit, true),
```

```
  receive
    % Chce se po nás, abychom pracovali někde jinde,
    % tak tedy se přesuneme tam a dostaneme práci
    % Je stejné jako přihlášení, jen je ten server,
    % který chce práci, první v seznamu.
    {presun, Odkud, Kam} ->
      unlink(Odkud), % S ním už nemáme nic společného
      link(Kam), % Sem se nastěhujeme
      Kam ! {prihlasit, self()}, % Přihlašme se
      receive
        prijato -> pracuj(); % Přijal nás
        {'EXIT', Kam, _} -> start() % Umřel, než nás přijal, zkusíme nový start
      end;
    % Přišla práce. Skončíme (tím se odpojíme ze serveru),
    % ale předtím ještě spustíme vlastní zpracování.
    {prace, Zadavatel, Fun} -> pracuj(Zadavatel, Fun);
    % Umřel server, přihlásíme se jinam o práci
    {'EXIT', _, _} -> start()
  end.

start([Server|Nahradni]) ->
  % Zkusme se přihlásit na server
  Server ! {prihlasit, self()},
  receive
    % Super, chce nás
    prijato -> pracuj()
    % Neozývá se, zkusíme jiný
    after nastaveni:timeout_serveru() -> start(Nahradni)
  end.

start() -> spawn(prace, start, [servery()]).
-module(server).
-export([start/0, startInterni/0]).

rozesli(Co) -> lists:foreach(fun (Server) -> {server, Server} !
                                   {Co, self()} end, nastaveni:servery()).

% Když máme jak úkoly, tak pracující, tak něco z toho zpracujeme
zpracuj([{Zadavatel, Fun}|Ukoly], [Pracant|Pracujici]) ->
  % Předáme práci
  Pracant ! {prace, Zadavatel, Fun},
  % Už se o něj nemusíme starat, oni si to vyřídí mezi sebou
  unlink(Pracant),
  % Zadavateli o tom řekneme a dál nás nezajímá
  Zadavatel ! {prirazeno, Pracant},
  unlink(Zadavatel),
  % A nyní ten zbytek úkolů
  zpracuj(Ukoly, Pracujici);
zpracuj(Ukoly, Pracujici) -> receive
    % Chce se po nás vykonávat nějaká práce
    {prace, Zadavatel, Fun} ->
```

```
      % Vezmeme si to na starost a uložíme
      link(Zadavatel),
      Zadavatel ! prijato,
      case Pracujici of
        [_|_] -> ok;
        % Nemáme žádného pracanta, řekneme si o nějaké
        true -> rozesli(chciPracanta)
      end,
      % A podíváme se, co se dá dělat nyní
      zpracuj(lists:append(Ukoly, [{Zadavatel, Fun}]), Pracujici);
    % Přišel nový pracant.
    {prihlasit, Pracant} ->
      % OK, bereme ho
      link(Pracant),
      Pracant ! prijato,
      case Ukoly of
        [_|_] -> ok;
        true ->  rozesli(mamPracanta)
      end,
      zpracuj(Ukoly, [Pracant | Pracujici]);
    % Někdo chce pracanta
    {chciPracanta, Server} -> case Pracujici of
      % Máme ho, tak mu řekneme, ať se přestěhuje
      [Pracant | Zbytek] ->
        % Dohodneme se s druhým serverem, aby ho čekal
        link(Server),
        Server ! {cekej, Pracant, self()},
        receive
          % Čeká na něj, pošleme mu ho
          cekam ->
            unlink(Server),
            Pracant ! {presun, self(), Server},
            zpracuj(Ukoly, Zbytek);
          % Umřel, tak si ho necháme
          {'EXIT', Server, _} ->
            zpracuj(Ukoly, [Zbytek])
        end;
      % Nemáme, ignorujeme požadavek
      [] -> zpracuj(Ukoly, Pracujici)
    end;
    % Máme očekávat pracanta
    {cekej, Pid, Od} ->
      link(Pid),
      Od ! cekam,
      receive
        % Přišel
        {prihlasit, Pid} ->
          % Přijmout, zařadit a pokračovat
          Pid ! prijato,
          zpracuj(Ukoly, [Pid | Pracujici]);
        % Ten už nepřijde, řekneme si o jiného
```

```
        {'EXIT', Pid, _} ->
          Od ! {chciPracanta, self()},
          zpracuj(Ukoly, Pracujici);
        % Umřel server, který ho měl poslat, smůla
        {'EXIT', Od, _} -> zpracuj(Ukoly, Pracujici)
      end;
    % Někdo nabízí pracanta
    {mamPracanta, Server} ->
      case Ukoly of
        % Máme pro něj využití, řekneme si o něj
        [_|_] -> Server ! {chciPracanta, self()};
        [] -> ok
      end,
      zpracuj(Ukoly, Pracujici);
    % Něco skončilo. Ať to byl pracant nebo zadavatel,
    % odebereme všechny, které tomu tady odpovídají
    % Pokud to byl zadavatel, tak se nic neděje
    % Pokud to byl pracant, zadavatel si požadavek zadá znovu
    {'EXIT', PID, _} -> zpracuj(lists:keydelete(PID, 1, Ukoly),
                                lists:delete(PID, Pracujici))
  end.

startInterni() ->
  % Určitě nechceme umírat, když nepřežije některý klient, jen ho vyřadíme
  process_flag(trap_exit, true),
  % Začneme pracovat, nejsou žádné úkoly ani pracanti
  zpracuj([], []).

start() -> register(server, spawn(server, startInterni, [])).
```

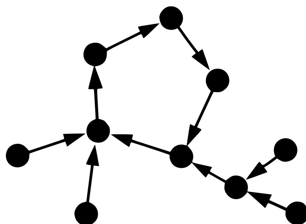
Michal „vorner“ Vaner

22-4-1 Zaheslované stránky

Ano, je to úloha na grafy, dokonce orientované![11] Vrcholy jsou stránky, hrany si nastavíme, aby vedly z odemknutelného dokumentu na dokument s příslušným heslem. Potom má graf tu zvláštní vlastnost, že počet hran, které vychází z libovolného vrcholu, je roven 0, nebo 1.

To se bude jednak příjemně implementovat, druhak to silně omezuje strukturu takových grafů. Ptejme se na počet hran u každé komponenty slabé souvislosti (to je taková, která nehledí na orientaci hran) v závislosti na počtu vrcholů komponenty (n).

1) Nejméně to může být $n - 1$: v případě, že jde o strom. Odebrání libovolné hrany způsobí rozpad komponenty.
2) Nejvíce to může být n, protože každému vrcholu přísluší nejvýše jeden výstupní konec hrany. Taková komponenta může být tvořena orientovanou kružnicí, vlastnost ale neporušíme ani tím, přilepíme-li k takové kružnici cestu vedoucí do nějakého jejího vrcholu. Situace dokonce může vypadat jako na obrázku:



V obou případech stačí prolomit jedinou stránku. Našemu algoritmu proto stačí spočítat komponenty slabé souvislosti daného orientovaného grafu a určitě bychom to mohli v lineárním čase a prostoru stihnout spočítat tak, že bychom si situaci odorientovali a spouštěli do nových a nových vrcholů průchody do hloubky / šířky.

Můžeme to ale udělat jednodušeji. Uložíme-li graf do pole délky n, kde na i-tém místě uložíme, do kolikátého vrcholu míří hrana vycházející z i-tého vrcholu (nulu, pokud žádná nevychází), máme k dispozici šikovnou reprezentaci, kterou se snadno projdeme bez použití front a zásobníků: stačí si prstem (proměnnou) ukazovat, kde právě jsme, a jít „rovnou za nosem“.

Začneme-li si ukazovat v libovolném vrcholu komponenty prvního druhu, dostaneme se do kořene stromu, v komponentě druhého druhu se zacyklíme v jejím cyklu. Pokud se nám tyto případy podaří detekovat a zaznamenat do kořene / celé kružnice, že už jsme tam byli, můžeme při takové příležitosti přičíst k počtu nutně prolomených stránek jedničku.

---

[11] Kuchařka o grafech: *http://ksp.mff.cuni.cz/tasks/20/cook3.html*

Toto poznamenávání může probíhat ve zvláštním poli, kde si budeme při průchodu přepisovat tři značky: *prolomeno*, *právě lámáno* a *netknuto*. Na počátku je všude poznamenáno *netknuto*, při procházení za sebou pokládáme *právě lámáno*, což po dokončení průchodu zaměníme za *prolomeno*. Cyklus pak snadno detekujeme tím, že narazíme na značku *právě lámáno*, dojdeme-li někdy do situace *prolomeno*, můžeme skončit bez navýšení počtu nutných prolomení.

Ještě jednoduší je přepisovat uložený graf. Pěknou implementaci na tomto základě napsal Dominik Smrž – autorský program[12] využívá jeho nápadu se zápornými čísly vrcholů coby identifikátory průchodů. Určitě si ho projděte, je krátký.

Lukáš Lánský

22-4-2 Rozvoz zásilek

Nejprve pár slov o tom, jak úlohu neřešit. Ne vždy je dobrý nápad zkontrolovat úsek, kterým prochází největší počet dosud nezkontrolovaného zboží – mohlo by se to totiž v budoucnu vymstít. Ukažme si to na protipříkladu (ve formátu zadání úlohy): `1:1->2, 2:1->4, 2:3->6, 1:5->6` se dvěma skenery k dispozici. V prvním kroku je sice nejvýnosnější zkontrolovat úsek `3-4`, ale tím dostanu řešení o nejvýše 5 zkontrolovaných kusech, zatímco optimální řešení kontrolující úseky `1-2` a `5-6` dodá luxusních 6. Aneb důkazy nejsou pro blaho opravovatelů, ale pro to, abyste si mohli být jisti funkčností Vašich řešení.

Když tedy na první pohled nevíme, které řešení je nejlepší, zkusíme preventivně všechny. A jak říká známe přísloví známé už z dob počítání na prstech, „od backtracku k dynamice krůček.“ Budeme si tedy počítat, kolik nejvýše zásilek zvládneme zkontrolovat pomocí `k` skenerů na stanicích `1..i`, a to v (dvourozměrném) poli `soucet`; pak optimální řešení úlohy zkontroluje právě `soucet[N][S]` kusů zboží.

Jak spočísti hodnotu `soucet[i][j]`? Inu, pro `i=1`, tedy jeden skener, je otázka jednoduchá, prohlédneme všechny stanice v intervalu `1..j` a podíváme se, ve které užitím skeneru zkontrolujeme nejvíce zboží – přesně to je hodnota `soucet[1][j]`. Pro více skenerů už to ale takto jednoduše nejde, viz první odstavec. Každopádně si můžeme říci – buďto oskenujeme úsek `j-1:j` (poslední úsek intervalu), nebo ne; spočítáme nejlepší řešení pro obě situace a pamatujeme si jen to lepší. V druhém případě vlastně skenujeme jen na intervalech `1..j-1`, tedy počet kusů je `soucet[i][j-1]`. Prvý případ je drobet složitější, na počtu zboží, které oskenujeme v `j`-té stanici, se totiž promítne, jaké předchozí stanice jsme již oskenovali. Každopádně to opět můžeme vzít hrubou silou,

[12] Autorské programy kvůli šetření papírem nepřikládáme, naleznete je na našich webových stránkách.

tedy postupně uvážit, že před `j`-tou stanicí jsme jako poslední oskenovali `k`-tou pro všechna `k` menší než `j`, a z těchto si vzít nejlepší řešení.

Nechť `cena[k][j]` říká, kolik zboží se proskenuje úsekem `j-1:j`, byl-li nejlevější proskenovaný úsek `k-1:k`; pak by předchozí myšlenka vypadala v pseudokódu následovně:

```
l = 0;
for(k=0;k<j;k++)
  l = max(l,soucet[i-1][k] + cena[k][j]);
soucet[i][j] = max(soucet[i-1][j],l);
```

Bystrý čtenář si jistě klade otázku, jak víme, že v optimálním řešení pro `i-1` skenerů na stanicích `1..k` bude poslední skener právě na pozici `k-1:k`, to jest, že můžeme počítat `cena[k][j]` a ne nějakou nižší. Inu, my to nevíme, ale je nám to jedno. Ono se bude stávat, že skener nebude na poslední pozici a my tak vlastně v tomto kroku dostaneme horší řešení; jenže z nějakého předchozího kroku už máme lepší.

Zbývá vysvětlit, jak efektivně spočteme `cena[i][j]`. Dělat to v průběhu počítání není ono, protože spoustu věcí bychom dělali mockrát. Raději si hodnoty spočítáme dopředu, všechny najednou. Nejprve si intervaly dopravy zboží setřídíme vzestupně dle jejich koncových zastávek. Pak postupně pro všechny úseky `j-1:j` provedeme následující: do fronty si naházíme všechny intervaly dopravy zboží, které procházejí přes `j-1:j`, a při tom si spočteme celkový počet zboží `m` v těchto intervalech. Pak budeme postupně brát úseky `i-1:i > j-1:j` a spočteme `cena[j][i]`. Z fronty vyházíme ty intervaly, které už do `i-1:i` nezasáhnou (takové se nacházejí jen a pouze na začátku fronty), a o počet zboží v nich snížíme `m`, `cena[j][i]` je pak celková suma zboží v intervalech procházejících přes `i-1:i` minus `m`. Pro podrobnosti viz zdrojový kód na našem webu.

Jak se přesvědčit, že algoritmus je správně? Inu, indukcí podle počtu skenerů a zastávek. Pro jeden skener a libovolně zastávek náš algoritmus určitě funguje. Udělejme tedy indukční krok pro `N` skenerů a `S` zastávek, za předpokladu, že pro menší počet skenerů a zastávek algoritmus funguje. Pokud optimální řešení neskenovalo poslední zastávku, tak tak neučiní ani náš algoritmus, neboť si všimne, že `soucet[N][S-1]` je větší než libovolné řešení skenující poslední zastávku (z indukčního předpokladu), a vítězíme. Pokud optimální řešení poslední zastávku skenovalo, podíváme se, jakou zastávku skenovalo jako předposlední, nechť je to `k`-tá. Pak ono optimální řešení zkontroluje nejvýše `soucet[N-1][k] + cena[k][j]` zásilek (z indukčního předpokladu), ale přesně tohle náš algoritmus vezme v úvahu. Celý důkaz je v podstatě jen přečtením toho, co algoritmus dělá, tak už to u (fungujících) dynamik bývá :).

Ještě se stručně zamysleme nad tolik omílanou časovou složitostí. Předpočet pole `cena` nás stojí $\mathcal{O}(S \cdot (S + P))$ – kde S je počet stanic a P počet zásilek

zboží – neboť pro každý úsek `j-1:j` postupně do fronty naházím a vyházím až P prvků a při tom kouknu na nejvýše S zastávek. A počítání pole `soucet` je s časem na $\mathcal{O}(N \cdot S^2)$, neboť pro každý počet skenerů počítám každý úsek `i-1:i` kouknutím na až S předchozích úseků. S pamětí se vejdeme do sympatických $\mathcal{O}(S^2 + P)$. Za domácí úkol si zkuste řešení upravit tak, aby dělalo to co má, tedy neodpovídalo jen váhou optimálního řešení, ale ono řešení přímo vypsalo.

Vojtěch Tůma

22-4-3 Muzeum

Úloha se dala řešit vylepšeným algoritmem na hledání mostů z grafové kuchařky. Tady si ukážeme jeden trošku elegantnější postup.

Všimneme si, že centrální kamera nemůže ležet na kružnici. To je vidět z toho, že má stupeň rovný počtu komponent a mezi komponentami nemůže vést hrana. Další pozorovací cvičení: Když se jedná o biologické oddělení, tak nám stačí znát kostru grafu (na hranách, které tvoří kružnice, nezáleží, protože ty jsou jen uvnitř komponent). Do třetice si všimneme, že jen z centrálního vrcholu můžeme prohledat $(N - 1/K) \cdot (K - 1) + 1$ kamer,[13] a to tehdy, když tam nezačínáme s hledáním.

Při řešení budeme nejprve předpokládat, že jsme správně u biologů, pak najdeme centrální kameru a nakonec ověříme, že jsme tam skutečně správně byli. Začneme tedy hledat třeba do hloubky a při návratu počítat navštívené vrcholy. Pokud narazíme na takový, který je podezřelý z centrálnosti, poznamenáme si ho. Teď už jen ověření. Kupodivu stačí to stejné prohledávání. Centrální vrchol si označíme za navštívený a pro každý z jeho sousedů zkontrolujeme, že z něj jde prohledat právě tolik vrcholů, kolik se na slušnou komponentu patří.

Jitka Novotná

22-4-4 Ořez zárodků

Označme si mateřský strom A, odvozený B. Začneme drobným pozorováním: Pokud ve stromě A najdeme posloupnost bratrských podstromů, která odpovídá podstromům synů kořene B, potvrdili jsme odvození B od A. Je-li x kořenem stromu X, jeho bratrským podstromem přirozeně rozumíme podstrom s kořenem y, kde y je bratrem x. Jaký strom zvolit jako mateřský? Zřejmě ten, který obsahuje více vrcholů. Každé „osekání“ pouze vrcholy odebírá. Pokud jich mají po „osekání“ stejně, musí být stromy identické a uvedené pozorování nadále platí.

[13] to je počet všech kamer bez jedné komponenty

Jak efektivně hledat posloupnost podstromů synů kořene B v A? Uděláme cimrmanovský krok stranou, vyhneme se znovuobjevování kola a převedeme problém na hledání podřetězce v řetězci. Ano, kuchařku jste si měli přečíst ...

Zbývá najít vhodnou reprezentaci stromu pomocí řetězce. Odpověď je triviální – použijeme uzávorkované výrazy. List je reprezentovaný pomocí `()`. Každý jiný vrchol (včetně kořene) pak jako `(` *=reprezentace 1. syna= =reprezentace 2. syna= ... =reprezentace posledního syna=* `)`. Dva malé stromečky ze zadání této úlohy jsou pak reprezentovány například takto: `((()())())` a `(()()())`.

Zřejmě každý strom má nějakou reprezentaci. Platí také, že je reprezentací strom jednoznačně určen? To snadno dokážete pomocí indukce. Pro list to platí a dále postupně podle složitosti vrcholu ... Zkuste si to rozmyslet. Také platí, že každý správně uzávorkovaný výraz (v běžném slova smyslu) reprezentuje nějaký strom. Pokud tedy vezmeme několik správně uzávorkovaných výrazů a „slepíme“ je za sebe do řetězce q, reprezentují posloupnost nějakých stromů $Y_1, Y_2, \ldots, Y_n$. Pokud se navíc q vyskytuje v reprezentaci nějakého stromu X, našli jsme uvnitř X interval sousedících bratrských podstromů $Y_1, \ldots, Y_n$.

Ať to tedy uzavřeme: Vezměme reprezentaci B a odštípněme vnější závorky (tj. získáme „slepenec“ reprezentací podstromů jeho synů), označme jako q. Pokud nalezneme q v reprezentaci stromu A, platí, že B je odvozený od A, v opačném případě nemůže být B od A odvozen.

Cože? Ještě jste si tu kuchařku nepřečetli a nevíte jak najít q v reprezentaci A? Přece pomocí vynálezu pánů Knutha, Morrise a Pratta ... algoritmem KMP.

Čas, paměť? Trvání výroby řetězcové reprezentace stromu a její velikost jsou lineární vzhledem k počtu vrcholů stromu. KMP běží v lineárním čase se součtem délek řetězců (jehly i kupky sena :o)). Časová i prostorová složitost algoritmu je tedy $\mathcal{O}(N)$, kde N budiž součtem počtu vrcholů obou stromů.

Josef Pihera

22-4-5 Energetické články

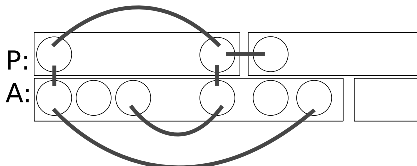
Úloha měla přísné limity v zadání, ale samotná data byla většinou přísná jen v jednom parametru, a tak jistě prošla i asymptoticky ne tak dobrá, leč vynalézavá řešení. A tak to u praktických úloh má být!

Pojďme na řešení: nejprve bylo důležité si všimnout, že pokud zřetězíme dva palindromy za sebe a vznikne další palindrom, musí se stát, že oba palindromy mají společný „základ“. Pokud je kratším palindromem `ABA`, pak delší palindrom (aby bylo zřetězení palindromem) musí začínat `ABA` (má tvar `ABAx`), a protože delší palindrom je také palindrom, bude i končit `ABA` (tvar `ABAxABA`) a dále pokračujeme stejným argumentem (indukcí). Není pravda, že jeden je mocninou druhého (například pro `ABAABA` a `ABAABAABA`), ale společný kořen jistě mají.

Hodilo by se nám tedy počítat **kořen** palindromu, tedy nejkratší řetězec takový, že jeho vhodným umocněním získám celý palindrom. Takový kořen je sám palindromem. Umíme-li kořen najít, pak stačí pro vstupní palindromy vždy najít kořeny, tyto kořeny jednoduše zpracovat (ať už pomocí hashování nebo pomocí setřízení, jako je to předvedeno ve vzorovém řešení) a správně napočítat.

Teď na tu pořádnou práci: jak co nejrychleji spočítat kořen palindromu? Půjdeme na to stejně, jak popisoval Vojta na KSPáckém fóru: budeme postupně procházet řetězec a průběžně si přepočítávat, jak by takový kořen mohl být dlouhý. Zpět se už ohlížet nebudeme – čteme-li Z-té písmenko a myslíme si, že kořen je P znaků dlouhý, podíváme na ($Z \bmod P$)-té písmenko a porovnáme. (Vězte, že je to takové písmeno, jež musí být stejné se Z-tým písmenkem, aby opravdu platilo, že kořen má délku P.)

Když se písmenka neshodují, můžeme prodloužit periodu P na $Z-1$ nebo Z. Nemůže se nám stát, že se nám schovává perioda někde mezi P a $Z-1$. To se nahlédne podobně jako první pozorování: byla-li by tam taková perioda A, pak v periodě A je 1., A-té, $A+1$-ní a P-té písmeno shodné. Protože A je menší než $Z-1$, víme, že A-té písmenko není P-té ani 1., a to nám dá další shodnou dvojici … a nakonec dostaneme, že by všechna písmenka musela být shodná.



Co se týče časové složitosti, tak pokud využijeme chytrou hashovací funkci se zhruba konstantní časovou složitostí na přístup (a pamatujeme si neprázdná políčka), pak se (nikoli v nejhorším případě) dostaneme k $\mathcal{O}(V+N)$, kde N je počet řetězců a V je velikost vstupu (sice mohla být až $N \cdot K$, ale rychleji by to stejně nešlo). Vzorové řešení používá mnoho asymptoticky záludných situací (například porovnávání řetězců nebo jejich třídění zabudovanou knihovnou), ale přesto jsme se jej rozhodli takto zveřejnit. Ukazuje totiž vlastnost mnoha praktických a soutěžních úloh – občas si můžete dovolit zkusit vyměnit nejlepší konstantu či asymptoticky horší složitost za čitelnost a jednoduchost kódu, pokud je hlavní podmínkou „vejít se“ do časového limitu.

Martin Böhm & CodEx

22-4-6 Umisťování panelů

Najskôr sa zamyslime nad tým, ako vyzerajú jednotlivé obdĺžniky, ktoré vystupujú v nejakom riešení, a ako ich rýchlo všetkých nájsť.

Tieto obdĺžniky sú najmenšie obdĺžniky, ktoré obsahujú množinu nejakých K bodov. Teda určite platí, že ohraničujúce x-ové a y-ové súradnice každého obdĺžnika sú rovnaké ako príslušné súradnice nejakých bodov, ktoré daný obdĺžnik obsahuje. Pretože inak by sme mohli jednoducho obdĺžnik v príslušnom smere zmenšiť a dostali by sme menší obdĺžnik, obsahujúci tie isté body.

Teda všetkých obdĺžnikov je maximálne $\mathcal{O}(N^4)$, pretože toľko je kombinácií, ako vybrať ohraničujúce body obdĺžnika. A pre každý obdĺžnik vieme v čase $\mathcal{O}(N)$ skontrolovať, či obsahuje práve K bodov.

V skutočnosti však počet všetkých možných obdĺžnikov obsahujúcich K bodov je rádovo menší. Ak totiž máme danú spodnú a hornú y-ovú súradnicu obdĺžnika a tiež ľavú x-ovú súradnicu obdĺžnika, potom existuje nanajvýš jedna možná hodnota x' pre pravú x-ovú súradnicu obdĺžnika tak, že obsahuje K bodov. Pretože obdĺžniky s menšou pravou x-ovou súradnicou ako x' obsahujú menej ako K bodov a obdĺžniky s väčšou pravou x-ovou súradnicou ako x' obsahujú viac bodov ako K.

Teda počet všetkých korektných obdĺžnikov je len $\mathcal{O}(N^3)$. Ak si naviac na začiatku usporiadame body podľa x-ovej súradnice, tak pre každú kombináciu hornej a dolnej súradnice y a ľavej súradnice x obdĺžnika vieme nájsť odpovedajúcu pravú stranu obdĺžnika v čase $\mathcal{O}(N)$. To spravíme jednoducho tak, že budeme postupne pridávať body do obdĺžnika, pokiaľ nedosiahneme počet K.

Ku vzorovému riešeniu však potrebujeme nájsť všetky korektné obdĺžniky rýchlejšie, a to v čase $\mathcal{O}(N^3)$. To vyriešime metódou „okienka“ (sliding window) nasledovne:

Najskôr si body znova usporiadame podľa x-ovej súradnice. Zafixujeme si hornú a dolnú súradnicu y a následne jedným prechodom nájdeme pravý okraj ku všetkým obdĺžnikom, ktoré majú príslušnú hornú a dolnú y-ovú súradnicu a ďalej budeme teda pracovať len s bodmi ležiacimi v tomto páse.

Najskôr nájdeme štandardným spôsobom (pridávaním bodov, pokiaľ nedosiahneme K) pravý okraj k obdĺžniku, ktorý má ľavý okraj na prvej x-ovej súradnici. Následne sa presunieme na ďalšiu ľavú x-ovú súradnicu a vieme, že príslušný pravý okraj leží napravo od pravého okraja posledného nájdeného obdĺžnika. Teda ho nájdeme znova pridávaním bodov, ale začíname od posledného nájdeného okraju. Všimnime si, že súradnice všetkých ľavých a pravých okrajov nájdených obdĺžnikov navzájom tvoria neklesajúcu postupnosť, a teda pri tomto hľadaní spravíme spolu lineárne veľa práce.

Teraz môžeme pristúpiť k vyriešeniu celej úlohy a zamyslieť sa, čo to znamená nájsť dva disjunktné osovo-paralelné obdĺžniky, pričom každý obsahuje práve K bodov.

V podstate to znamená to, že v každom takomto riešení existuje deliaca horizontálna alebo vertikálna priamka taká, že jeden z obdĺžnikov je na jednej strane tejto priamky a druhý je na opačnej strane priamky.

Bez ujmy na všeobecnosti teda predpokladajme, že pre optimálne riešenie je táto priamka horizontálna. (Vyriešením rovnakej úlohy pre vertikálnu priamku a vybratím lepšieho riešenia dostaneme celkové optimum.)

Teda pre hľadané dva obdĺžniky musí platiť, že horná y-ová súradnica jedného je menšia ako dolná y-ová súradnica druhého.

Pre každú y-ovú súradnicu si teda spočítame najmenší obdĺžnik, ktorý má na tejto pozícii dolný okraj, a taktiež najmenší obdĺžnik majúci na tejto súradnici horný okraj. Následným vyskúšaním všetkých dvojíc dolného a horného okraju nájdeme optimálne riešenie.

Časová zložitosť spočíva z $\mathcal{O}(N \log N)$ na utriedenie bodov na začiatku, $O(N^3)$ na nájdenie všetkých korektných obdĺžnikov a nakoniec $O(N^2)$ na vyskúšanie všetkých kombinácií dolného a horného okraja dvoch obdĺžnikov. Teda spolu $O(N^3)$.

Pamäťová zložitosť nám vystačí lineárna, ak si pri hľadaní obdĺžnikov súčasne vytvárame aj tabuľku obvodu najmenšieho obdĺžnika majúceho daný horný a dolný okraj, pričom túto tabuľku ihneď po nájdení obdĺžnika zaktualizujeme a samotné obdĺžniky si teda nemusíme pamätať.

Peter Ondrúška

22-4-7 Pozdrav z pravěku

Úkol 1: Nabízí se `x÷|x`, tedy dělit číslo jeho absolutní hodnotou, ale to selže pro $x = 0$. Lepší je využít toho, že porovnávání vrací 0 nebo 1 a použít `(x>0)-(x<0)`.

Úkol 2: Jelikož APL vyhodnocuje zprava doleva, `⍳5+1` je totéž co `⍳6`, tedy vektor `0 1 2 3 4 5`. Naproti tomu `1+⍳5` je totéž co `1+0 1 2 3 4`, což je podle pravidel o počítání s vektory `1 2 3 4 5`.

Úkol 3: `⍳(2+⍳3) → ⍳(2+0 1 2) → ⍳(2 3 4)`, což vytvoří trojrozměrné pole tvaru $2 \times 3 \times 4$ vyplněné čísly od 0 do 23.

Úkol 4: Stačí `⌈/⌈/x` – nejdříve nám `⌈/x` dá vektor maxim sloupečků a z něj si pak druhou redukcí vybereme maximum.

Úkol 5: Všimněme si, že jednička má být právě tam, kde je první souřadnice (číslo řádku) menší než druhá. Stačí tedy použít direktní součin `(⍳n)∘.<(⍳n)`.

Úkol 6: Nejprve vytvoříme matici

$$\begin{pmatrix} 0 & 1 & 2 & 3 & \dots & n-2 & n-1 \\ n-1 & n-2 & n-3 & n-4 & \dots & 1 & 0 \end{pmatrix}$$

laminací vektorů `⍳n` a `(n-1)-⍳n`. Pak ji stačí transponovat a operátorem `⍴` přeformátovat na vektor délky $2n$. Celý program tedy zní

```
(2×n)⍴⍉(⍳n)~((n-1)-⍳n).
```

Úkol 7: Nebudeme troškaři, najdeme rovnou všechny společné dělitele zadaných čísel x, y a pak z nich vybereme toho největšího:

```
a ← 1+⍳n
p ← 0=a|x
q ← 0=a|y
r ← a×p×q
d ← /⌈r
```

Jak to funguje? Nejprve sestrojíme vektor a obsahující čísla $1, \ldots, n$. Další vektor p obsahuje jen nuly a jedničky, přičemž jedničky jsou přesně na místech dělitelů čísla x (spočítáme zbytky a porovnáme je s nulou). Podobně q indikuje dělitele čísla y. A vektor r vznikne z a vynulováním těch čísel, která nejsou společnými děliteli x a y, takže už stačí najít maximum z jeho prvků.

Program ještě můžeme trochu zkrátit:

```
/⌈(0=a|x)×(0=a|y)×a←1+⍳n.
```

Filip Hlásek vymyslel ještě magičtější řešení:

```
+/0=y|x×1+⍳y,
```

zkuste přijít na to, jak funguje. Poradíme vám, že se k tomu hodí rovnost $xy = nd$, kde d je největší společný dělitel a n nejmenší společný násobek.

Martin Mareš

22-5-1 Turnaj

Vaše řešení (která byla tentokrát takřka výhradně správná) vykazovala velké odchylky v délce: zatímco si někteří vystačili s pěti větami, jiní popsali stránku. Vzhledem k obtížnosti úlohy si vážíme obou přístupů, ale odbytí těžšího příkladu krátkým textem vidíme velmi neradi.

Zadání klade návodné otázky, které nyní zodpovíme:

Jací jsou kandidáti na druhé místo? Evidentně právě ti draci, kteří prohráli s vítězem. Všichni ostatní totiž byli (třebas nepřímo) poraženi některým z těchto poražených draků a nejlepší tabulkové místo, na které mohou dosáhnout, je třetí. Z této množiny pak zároveň nemůžeme bez dalšího zkoumání žádného draka vyřadit, protože spolu zápas jistě nehráli, ani se libovolný z nich nemůže nacházet v podstromu libovolného jiného, takže si je nemůžeme nijak uspořádat.

Kolik takových kandidátů je? Jej, to je záludné. Zadání se explicitně nezmiňuje o tom, že by byl pavouk hry úplný binární strom, obrázek však k takovému pojetí vede. Při opravování jsem tedy akceptoval jak názor, že je těchto kandidátů logaritmicky vůči počtu zúčastněných draků, tak názor, že se to nedá moc dobře říct, jelikož strom může vypadat všelijak.

Jak mezi nimi co nejefektivněji vybrat druhého draka? To naopak záludné není vůbec: prostě sestavíme herní strom pro draky poražené výhercem. Vzhledem k jejich počtu N bude potřeba sehrát $N - 1$ utkání (každé vyřadí právě jednoho draka), vzhledem k počátečnímu počtu draků tedy v případě úplného stromu máme logaritmický počet nutných dohrávek.

Lukáš Lánský

22-5-2 Stráže údolí

V tejto úlohe sme mali zadané body na priamke, vedeli sme medzeru medzi každými dvoma susednými a chceli sme odstrániť maximálne K z nich, aby sme maximalizovali najkratšiu medzeru medzi tými bodmi, ktoré zostanú.

Táto úloha rovnako ako mnoho iných úloh má jednoduché, rýchle, ale pritom nesprávne greedy riešenie, ktoré je založené na postupnom odstraňovaní bodov susediacich s najkratšou medzerou (skúste si nájsť protipríklad).

Jednoduché korektné riešenie vieme naprogramovať pomocou dynamického programovania, kde stav výpočtu je dvojica (n, k) a pre každú dvojicu chceme spočítať optimálne riešenie, ak sme spracovali prvých n bodov a vyhodili sme práve k z nich. Takáto úvaha vedie na riešenie so zložitosťou $\mathcal{O}(N^2K)$, ale stále má ďaleko od vzorového riešenia.

Naše vzorové riešenie využíva myšlienku, ktorá sa používa vo veľa problémoch, kde spočítať samotné riešenie problému je pomerne zložité, zato však overiť, či existuje riešenie s požadovanou vlastnosťou, je pomerne jednoduché.

V našom príklade vieme ľahko overiť, či existuje riešenie, ktoré odstráni maximálne K bodov z daných a má minimálnu vzdialenosť aspoň takú ako pevne dané M. Zodpovedanie tejto otázky vieme previesť na iný známy problém „plánovania intervalov“: Máme zadaných N intervalov v čase, pričom každý začína v čase a_i a má dĺžku b_i. Pričom z týchto intervalov chceme vybrať maximálny počet tak, že žiadne dva vybraté intervaly sa neprekrývajú.

Prevod je nasledovný: všetky čísla a_i sú rovné pozíciam bodov na priamke a všetky b_i sú rovné M. Ak nájdeme riešenie tohto problému, potom sme našli maximálnu množinu bodov (počiatky intervalov), ktoré sú od seba vzdialené aspoň M. Pričom keď sme našli takéto riešenie, ktoré maximalizovalo počet vybratých intervalov (bodov), potom súčasne toto riešenie minimalizuje počet intervalov (bodov), ktoré sme nevybrali. Teda po nájdení riešenia, vieme zodpovedať otázku, či existuje riešenie, ktoré má najmenšiu vzdialenosť aspoň M, podľa toho, či naše riešenie „plánovania intervalov“ nevybralo maximálne K intervalov.

Treba však vedieť riešiť samotný problém plánovania intervalov. Na tento problém je však známy jednoduchý greedy algoritmus: Na začiatku si utriedime body podľa času konca intervalu, následne začneme tieto intervaly prechádzať v tomto poradí a súčasne si budujeme riešenie (množinu vybratých intervalov) použitím jednoduchého pravidla: pri prechádzaní, vždy keď môžeme práve spracovávaný interval pridať k budovanému riešeniu, tak ho tam pridáme.

Toto sa dá po usporiadaní intervalov vykonať v lineárnom čase od počtu intervalov, stačí si vždy len pamätať čas konca posledného intervalu v našom budovanom riešení. Naviac v našom špeciálnom prípade majú všetky intervaly rovnakú dĺžku, takže stačí usporiadať intervaly podľa začiatku (na vstupe však už máme pozície utriedené a v našej úlohe nemusíme triedenie vôbec riešiť).

Teraz už vieme zodpovedať otázku, či existuje riešenie s danou minimálnou vzdialenosťou. K čomu nám to poslúži? Treba si všimnúť, že ak existuje riešenie, ktoré má minimálnu vzdialenosť aspoň M, potom existuje riešenie, ktoré má minimálnu vzdialenosť M' pre každé $M' \leq M$ (jednoducho ponecháme rovnakú množinu bodov). Inak povedané, existuje číslo M^* také, že pre všetky $M \leq M^*$ riešenie existuje a pre všetky $M > M^*$ riešenie neexistuje.

A práve čislo M^* hľadáme. Teda riešenie by sme mohli nájsť tak, že ak máme rozsah súradníc bodov z nejakého intervalu R, potom vieme postupným skúšaním existencie riešenia, ktoré má minimálnu vzdialenosť R, $R-1$, $R-2$, …, nájsť číslo R^* v čase $\mathcal{O}(RM)$. Avšak z vlastnosti hľadaného čísla M^* môžeme použiť binárne vyhľadávanie na intervale R. Keď si pre medián prehľadávaného intervalu riešení zistíme, či existuje riešenie, pak sa na základe toho vieme rozhodnúť, v ktorej polovici prehľadávaného intervalu leží číslo M^*.

Takto vieme nájsť maximálnu minimálnu vzdialenosť medzi dvojicou bodov a body, ktoré máme odstrániť, sú počiatky nevybratých intervalov pri riešení

príslušného podproblému plánovania intervalov.

Celková časová zložitosť je $\mathcal{O}(N \log R)$, kde pri binárnom vyhľadávaní na intervale dĺžky R vieme v lineárnom čase overiť existenciu riešenia. Pamäťová zložitosť je $\mathcal{O}(N)$.

Peter Ondrúška

22-5-3 Zrcadla

Máme čtvercovou síť a hledáme v jistém smyslu nejkratší cestu, respektive cestu s co nejméně „zatáčkami" tvořenými zrcadly. Že by prohledávání do šířky? Tak se podívejme, jak ho realizovat v tomto případě. Pokud jste ještě žádné prohledávání do šířky nikdy nepotkali, podívejte se do grafové kuchařky na našich stránkách.[14]

Jak se dá čekat, ve frontě, jíž používá prohledávání do šířky, budou jednotlivá políčka čtvercové sítě a každé se tam dostane maximálně jednou, fronta tedy může narůst do velikosti až $\mathcal{O}(M \times N)$. Vždy, když odebereme políčko z fronty, pustíme z něj světlo do všech čtyř směrů (do některých políček může přijít světlo z různých směrů přes stejný počet zrcadel a ukládat ho do fronty dvakrát se nevyplácí). Pro každý směr postupně procházíme políčka, dokud nenarazíme na překážející dům, a zařazujeme je do fronty, jestliže v ní ještě nebyly. Když narazíme na dům, jejž chceme osvítit, vypíšeme počet zrcadel a skončíme. Vyprázdní-li se fronta a cíl je nedosažen, nejde na něj dosvítit.

Pro evidenci, kde se nacházejí překážky a přes kolik zrcadel došel algoritmus na konkrétní políčko, si zavedeme dvourozměrné pole o velikosti $M \times N$. Hodnota -3 na políčku i, j znamená, že je tam překážka, hodnota -2, že do políčka ještě nedorazilo světlo, a hodnoty větší nebo rovné nule, přes kolik nejméně zrcadel se tam světlo dostane.

Proč toto řešení funguje? Stačí, když si všimneme, že políčka ve frontě jsou uspořádána dle minimálního počtu zrcadel, která musíme použít, aby se do nich dostalo světlo. Pokud jsme se na políčko A dostali nejprve z políčka B a dostaneme-li se do něj později z jiného bodu, určitě k tomu použijeme nejméně tolik zrcadel jako z políčka B.

Jaká je časová složitost tohoto algoritmu? Každé políčko se sice objeví ve frontě maximálně jednou, ale světlo se z něj může dostat až do $\mathcal{O}(M + N)$ dalších políček. Navíc na každé políčko může doletět světlo až z $\mathcal{O}(M + N)$ jiných, celkově tedy vyjde ne moc pěkná složitost $\mathcal{O}(MN(M + N))$.

Jak algoritmus zrychlit? Hlavní problém, proč algoritmus pracuje v nejhorším případě tak pomalu, je, že se na některá políčka podíváme dokonce až $\mathcal{O}(M + N)$-krát, avšak do fronty je zařadíme jen jednou. Přitom je zbytečné se na ně dívat ze stejného směru vícekrát (např. z políčka, jež je nad ním, pak

[14] Kuchařka o grafech: *http://ksp.mff.cuni.cz/tasks/20/cook3.html*

z toho, co je o 2 nad ním…). Proto si budeme u každého políčka navíc ukládat, jakými všemi směry už jím letělo světlo.

Můžete si všimnout, že stačí ukládat pouze dva bity informace: jestli políčkem letělo světlo horizontálním směrem a jestli vertikálním směrem. Pokud totiž poletí světlo z políčka v souvislém úsek bez překážek na nějakém řádku či sloupci, tak ho proletí celý bez ohledu na to, odkud se naposledy mohlo odrazit.

S tímto vylepšením se po vytažení políčka z fronty podíváme, jestli už jím proletělo světlo horizontálním i vertikálním směrem a případně projdeme políčka tím či oním směrem (oběma zároveň určitě ne kromě zdroje, jelikož světlo muselo do políčka nějakým směrem doputovat). Díky vlastnostem prohledávání do šířky (políčka jsou ve frontě seřazena dle počtu zrcadel, přes která se do nich dostalo světlo) jsme si touto úpravou určitě nepokazili řešení.

Nyní už do každého políčka doputuje světlo nejvýše dvakrát, takže časová složitost vyjde $\mathcal{O}(MN)$. V nejhorším případě stejně projdeme skoro celou čtvercovou síť a ostatně i velikost vstupu je nejvýše $\mathcal{O}(MN)$ (bude-li řádově tolik překážek), takže asymptoticky lepší algoritmus vymyslíme jen těžko, pomineme-li nějaké heuristiky (triky, které v určitých případech zrychlí program), jež však obecně nefungují.

Pavel „Paulie“ Veselý

22-5-4 Davy lidí

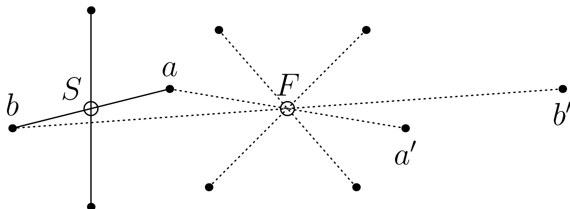
Úloha byla věru těžká. Vyřešíme tedy nejdříve několik podproblémů a z nich pak složíme celé řešení. Výklad okořeníme tímto značením: jsou-li A a S body v rovině, pak A^S značí obraz bodu A ve středové souměrnosti se středem S.

1. *Je množina bodů symetrická podle zadaného středu S?* To můžeme zjistit snadno: uložíme body množiny do nějaké datové struktury (třeba do vyhledávacího stromu). Pak je budeme postupně procházet body a pro každý bod A se podíváme, je-li ve struktuře i bod A^S. Pokud ano, oba smažeme a pokračujeme dál. To pro n bodů zvládneme v čase $\mathcal{O}(n \log n)$.

Můžeme to provést i jednodušeji: Setřídíme body lexikograficky (tzn. nejdříve podle x-ové souřadnice a kde je x stejné, tam podle y) a všimneme si, že pokud je bod A lexikograficky před B, pak je B^S lexikograficky před A^S. Jinými slovy v setříděném pořadí platí, že obraz prvního bodu je poslední bod, obraz druhého předposlední a tak dále. Tříděním strávíme čas $\mathcal{O}(n \log n)$, kontrolou pak $\mathcal{O}(n)$. To je výhodnější v případě, že chceme postupně vyzkoušet několik různých kandidátů na střed S.

2. *Je množina bodů symetrická?* To bude snadné – pokud je množina symetrická, musí její těžiště ležet ve středu symetrie. Stačí tedy spočítat těžiště (jeho x-ová souřadnice je průměrem x-ových souřadnic všech bodů a podobně y-ová souřadnice) a spustit na něj předchozí algoritmus.

3. *Známe polohu sochy S a fontány F, lze body rozdělit na část souměrnou podle S a část souměrnou podle F?* Zde naprostá většina řešitelů zkusila hladový algoritmus – testovat už známým způsobem souměrnost podle S, body, které souměrné nejsou, si dávat stranou a nakonec vyzkoušet, jestli jsou souměrné podle F. To ale bohužel nefunguje, elá hop, protipříklad z klobouku ven:



Body a, b se účastní dvou symetrií – jednak spolu podle S, jednak s a', b' podle F. Pokud tedy při zkoumání středu S body a, b spárujeme, zbudou pak a' a b' na ocet. Kdybychom je ovšem odložili oba stranou, spárovali bychom následně podle středu F dvojice a–a' a b–b'. (Zde by samozřejmě pomohlo zkoumat nejdřív F a pak S; takový algoritmus ale nachytáme, pakliže k našemu protipříkladu přidáme ještě jeho kopii překlopenou podle osy úsečky SF.)

Jak z téhle arcipatálie ven? Inu, za vším hledej grafy… body prohlásíme za vrcholy, dvojice symetrické podle S spojíme jedním typem hran (na obrázku plné čáry), dvojice symetrické podle F druhým (na obrázku tečkovaně). V tomto grafu chceme najít *perfektní párování,* čili rozdělit vrcholy na dvojice tak, aby každá dvojice byla spojena hranou.

Žádný problém, každý matfyzák ví už od narození, že na hledání perfektního párování tu je Edmondsův „zahradní“ algoritmus. My ale tak mocné kouzlo ani nebudeme potřebovat. Místo toho si zkusíme představit, jak náš graf vypadá. Kterýkoliv vrchol může sousedit s nejvýše jednou hranou prvního druhu a nejvýše jednou druhého. Stupeň vrcholu tedy může být buď 0, nebo 1, nebo 2 a pokud je 2, jsou obě hrany různých druhů. To nám nedává moc možností – každá komponenta souvislosti musí být buďto izolovaný vrchol nebo cesta, případně kružnice. Na cestě i na kružnici se navíc musejí střídat hrany obou druhů, takže ihned víme, že kružnice mají sudou délku a tím pádem si na nich stačí vybrat buď jeden nebo druhý druh hran a je spárováno. Cesty o sudém počtu hran a izolované vrcholy (to jsou vlastně cesty o nula hranách) spárovat určitě nejdou. Na cestě o lichém počtu hran stačí použít ten typ hrany, kterým cesta začíná i končí.

K vyřešení tohoto podproblému tedy postačí sestrojit pomocný graf (třeba dvojím spuštěním algoritmu **1.**), rozložit ho na komponenty souvislosti a ověřit, jestli se mezi nimi nevyskytne sudá cesta. To vše zvládneme v čase $\mathcal{O}(n)$, pokud už máme všechny body setříděné lexikograficky.

4. *Polohy S, F neznáme.* Co teď? Budeme pokorně zkoušet všechny kandidáty na polohu sochy a fontány a spouštět pro ně předchozí ověřovací algoritmus. Není jich nekonečně mnoho? Ne ne, střed přeci musí ležet buďto v nějakém zadaném bodě nebo ve středu úsečky určené dvěma zadanými body. Takových míst je $\mathcal{O}(n^2)$, takže dvojic kandidátů na S, F je $\mathcal{O}(n^4)$, ověřováním každé strávíme $\mathcal{O}(n)$. Celková časová složitost je tedy $\mathcal{O}(n \log n + n^5) = \mathcal{O}(n^5)$, paměti nám stačí lineárně.

5. *Zrychlujeme.* Jak se zbavit obludné páté mocniny, jež nám škodolibým chechtotem kazí radost z vítězství? Trochu množinu kandidátů na středy omezíme. Předně – zvolíme si nějaký pevný bod A a prohlásíme, že socha je to, podle čeho je tento bod souměrný. Stačí tedy při hledání poloh sochy vyzkoušet jen středy úseček, kterých se bod A účastní. Pro každou polohu sochy pak nalezneme nějaký bod, který podle ní není s ničím symetrický (kdyby žádný takový nebyl, už jsme úlohu vyřešili). Tento bod jistě patří do druhé množiny, takže fontána se vyskytuje na nějaké úsečce vedoucí z tohoto bodu. Celkem tedy $\mathcal{O}(n)$ možností pro sochu, $\mathcal{O}(n)$ pro fontánu a čas $\mathcal{O}(n)$ na ověření. To dává dohromady $\mathcal{O}(n \log n + n^3) = \mathcal{O}(n^3)$ s lineární pamětí. Umíte to lépe? My zatím ne.

opravila Jitka Novotná, řešení sepsal Martin Mareš

22-5-5 Čokolámání

Předtím, než určíme, kolik vlastně rozlámání celé čokolády nejméně stojí, musíme vymyslet, jak takové rozlámání provést. Velice jednoduché řešení je jít na čokoládu „hladově“. Vzhledem k tomu, že zlomy, které provedeme dříve, se započítají méněkrát než ty, co provedeme později, tak čokoládu rozlomíme vždy podle nejdražšího zatím nepoužitého zlomu. Pokud zlom prochází přes víc kusů čokolády, rozlomíme každý z nich. Jenže takovýhle jednoduchý postup přece nemůže fungovat, ne? Ukazuje se, že může, jenom to musíme dokázat.

Nějaký konkrétní postup rozlámání si můžeme představit dvěma způsoby: buď jako binární strom, kde každý vrchol je kus čokolády, synové vrcholu jsou ty kusy, které z něj vzniknou jedním rozlomením, a listy jsou kusy, které už nejdou rozlomit (tzn. velikosti 1×1). Druhou reprezentací postupu rozlámání je posloupnost zlomů, ve které se každý zlom vyskytuje právě jednou.

Převedení posloupnosti zlomů na strom je jednoduché: lámeme čokoládu podle zlomů v posloupnosti a všímáme si, které kusy jsme rozlomili na jaké. Opačný směr je ovšem složitější: posloupnost zlomů určíme ze stromu tak, že rekurzivně vypočítáme posloupnosti zlomů podstromů synů kořene, ty spojíme a na začátek ještě přidáme zlom z kořene. Spojení je definované tak, že obě posloupnosti musí být podposloupnostmi (ne nutně souvislými) výsledku s tím, že se v něm žádný zlom nesmí opakovat, ale na druhou stranu můžeme změnit pořadí v rámci souvislých skupin zadaných posloupností, které obsahují pouze

zlomy jedné orientace. Když vodorovné zlomy budu značit písmeny a svislé čísly, tak například posloupnosti `DCA1B` a `C3ABD` přeuspořadám na `CAD1B` a `C3ADB` a výsledkem je `C3AD1B`, posloupnosti `A1B` a `B3A` spojit nejdou a strom, který obsahuje takové podstromy, nejde reprezentovat jako posloupnost zlomů. Když postup rozlámání reprezentovaný stromem převedu na posloupnost zlomů a pak zpět na strom, výsledkem může být jiný strom. Jejich ceny ale budou stejné, protože jsme jenom změnili pořadí v rámci skupin zlomů se stejnou orientací. (Když lámu zleva doprava, tak výsledek má stejnou cenu, jako když lámu zprava doleva. Když ale nejdřív lámu vodorovně a pak svisle, tak výsledek může mít různou cenu, než když lámu v opačném pořadí.)

Náš postup rozlámání se reprezentuje jako posloupnost zlomů jednoduše: jsou to všechny zlomy setříděné od nejdražšího. Nyní potřebujeme dokázat, že tato posloupnost zlomů je nejlevnější možná a také, že žádný postup rozlámání, který se nedá vyjádřit jako posloupnost zlomů, nejlevnější být nemůže.

Všimneme si, že jakoukoliv posloupnost můžeme setřídit tak, že vezmeme prvek s nevyšší hodnotou a přesuneme jej na první místo, pak vezmeme prvek s druhou nejvyšší hodnotou a dáme ho na druhé místo a tak dále. Při každém takovém přesunutí prvek přeskakuje jen prvky, které mají menší hodnotu, než on sám. Pokud jsou prvky posloupnosti zlomy, tak platí, že přeskočení zlomu, který má stejnou orientaci cenu nezmění. Na druhou stranu přeskočení zlomu s opačnou orientací způsobí, že počet výskytů přeskakujícího zlomu ve stromové reprezentaci se zmenší o jedna, naopak počet výskytů přeskakovaného zlomu se o jedna zvýší. A protože počet výskytů ve stromě odpovídá tomu, kolikrát se zlom započítá do výsledné ceny, určitě jsme takovýmto setříděním posloupnosti zlomů její cenu nezvýšili. Jako výsledek jsme dostali naši posloupnost, ta je tedy určitě nejlevnější.

To, že postup rozlámání, který nejde reprezentovat posloupností zlomů, nemůže být nejlevnější, dokážeme tak, že si ve stromě, který reprezentuje takovýto postup, najdeme vrchol, jehož podstrom reprezentovat posloupností nejde, ale podstromy obou jeho synů jdou (takový určitě existuje). Alespoň jedna z posloupností synů není setříděná od nejdražšího (kdyby obě byly, tak jdou spojit) a navíc se ani nedá setřídit přehazováním v rámci souvislých skupin se stejnou orientací. To znamená, že tam buď existuje dvojice po sobě jdoucích zlomů s opačnou orientací, jejíž první prvek má menší cenu, nebo se taková dvojice dá vytvořit přehazováním ve skupině se stejnou orientací. Když tuto dvojici prohodím, zmenším tím cenu rozlámání, a tento postup tedy nemohl být nejlevnějším.

Dokázali jsme tedy, že náš postup je nejlevnější, teď už zbývá jenom vymyslet, jak spočítat tuto cenu. Stačí si uvědomit, že každý zlom v posloupnosti se započítá o jedna víckrát, než kolik je před ním zlomů s opačnou orientací. Algoritmus bude postupovat tak, že si všechny zlomy setřídí podle ceny a postupně

je od nejdražšího započítává, každý tolikrát, kolik zlomů podle opačné osy, než má aktuální zlom, jsme už započítali. Pokud má čokoláda rozměry $M \times N$, tak časová složitost je $\mathcal{O}((N+M)\log(N+M))$ a paměťová $\mathcal{O}(N+M)$.

Poznámka: Setřídění zlomů podle ceny jsme docela odbyli, jaký třídící algoritmus je nejlepší? Vzhledem k tomu, že tato úloha je praktická, tak odpověď je velice jednoduchá: obvykle ten, který programovací jazyk, který používáme, sám obsahuje. Třeba v Céčku je to `qsort()`, v C# `Array.Sort()`.

Petr Onderka

22-5-6 Hlídači princezny

Začneme, jako každý líný člověk, od toho nejjednoduššího. Představme si, že máme hlídače, řekněme A, kterého nekryje vůbec nikdo. Ten určitě do útoku jít nemůže. Tak tam ale do útoku pošleme hlídače B, který je kryt hlídačem A (pokud již v útoku není). Oba ze vstupu odstraníme, protože jsou již vyřešení a pokračujeme s menší úlohou stejného druhu.

Co ale v případě, že žádného nekrytého nemáme? Pak si všimneme, že takový graf musí být několik nepropojených orientovaných cyklů. Vezmeme tedy každý z cyklů zvlášť (můžeme, neovlivňují se). Když je cyklus sudé délky, pak dokážeme poslat do útoku právě polovinu z jeho hlídačů (každého druhého) – lépe to zřejmě nejde, za každého v útoku musí být alespoň jeden, který kryje. A u lichého? Tam nám, bohužel, jeden zbude, ale ať párujeme jakkoliv, jeden zbýt musí, tedy to také nejde lépe.

Že mi ještě nevěříte? No, tak malinko důkazů. Napřed si dokážeme, že pokud v grafu není žádný vrchol vstupního stupně 0 a všechny mají výstupní stupeň 1, pak se jedná o cykly. Vyberme si libovolný vrchol. Z něho vede právě jedna hrana ven. Vydejme se po ní a dojdeme do dalšího. A tak dále. Jednou musíme potkat vrchol, ve kterém jsme již byli. A proč je to ten první? Kdyby nebyl, tak do toho, který jsme potkali podruhé, vedou alespoň dvě různé hrany (jedna, po které jsme přišli poprvé a druhá, kterou jsme přišli teď). Protože ale z každého vrcholu vychází právě jedna hrana, průměrně do každého musí také vstupovat jedna. A neexistuje vrchol, který by měl méně než jednu vstupní hranu, nemůže tedy existovat ani takový, který má více než jednu.

A nyní to tvrzení hned na začátku. Proč můžeme vzít hlídače B? Hledáme nejmenší protipříklad – vstup s nejmenším počtem hlídačů, kde náš program vybere špatné řešení. Hlídače B jsme vybrali a zkazili jsme to tím – to ale znamená, že byl potřeba v záloze na krytí hlídače C.

No dobrá, ale tím, že místo C vezmeme B, si přeci neuškodíme. Hlídačů máme stejně a po nasazení B nám v grafu zbude jeden vrchol navíc (což nám, zřejmě, neuškodí, protože ho můžeme jednoduše nevyužít).

To je celé hrozně hezké, víme, že to funguje. Jak to ale napsat? A to ještě tak, aby to běželo rychle? Samozřejmě, mohli bychom pokaždé projít celý vstup,

pokusit se najít vrchol stupně 0, ale to by trvalo dlouho. Proto je na to potřeba jít chytřeji.

Předpočítáme si, hned na začátku, vstupní stupeň každého vrcholu. Poté si rozházíme vrcholy na dvě hromádky – v jedné budou ti nekrytí a v druhé ti ostatní.

Potom zkusíme vzít vždy jednoho nekrytého. Toho dáme do zálohy (to je náš A). Pokud je ten, kterého kryje, ještě nezpracovaný, nasadíme ho do útoku (to je B). A tomu, kterého kryje B, odečteme jedničku od vstupního stupně, pokud mu klesne na nulu, přehodíme z jedné hromádky do druhé. Celý tento jeden krok lze stihnout v konstantním čase.

Jakmile není na nekryté hromádce nikdo, máme cykly. Je jedno, od kterého začneme cyklus „rozmotávat“, tak si prostě jeden vrchol vezmeme a uděláme s ním to samé – řekneme, že je v záloze, toho, koho kryje, pošleme do útoku. Tím nám vznikne nekrytý hlídač (pokud měl cyklus délku alespoň 3) a pokračujeme dál obvyklým způsobem.

Dále, hromádka krytých hlídačů může být čistě virtuální – ve chvíli, kdy z ní odebíráme, tak je totožná se všemi ještě nepoužitými. A nepoužitelnost si můžeme značit přímo v hlídači a pamatovat si, kde jsme naposledy skončili s vyhledáváním.

Celkově nám z toho tedy vychází pěkná lineární složitost časová a stejně tak paměťová.

Michal „vorner“ Vaner

22-5-7 ArcheoPaleoLingua

Úkol 1: Prvočísla můžeme hledat například takto:

```
p N: (2=+/~Z∘.|Z)/Z←1+⍳N.
```

Jak toto kouzlo funguje? Nejprve si do proměnné Z uložíme čísla od 1 do N. Pak pomocí vnějšího součinu `Z∘.|Z` vytvoříme tabulku všech zbytků po dělení a operátorem `~` ji znegujeme – výsledkem je tedy matice, která má na pozici i, j jedničku právě tehdy, když je číslo j dělitelné číslem i, jinak nulu. Redukcí `+/` z toho vytvoříme vektor, jehož j-tá složka udává počet dělitelů čísla j. Ten následně porovnáme s dvojkou a dostaneme vektor, jehož j-tá složka je 1 právě tehdy, je-li j prvočíslo. Pak už stačí použít operátor komprese, abychom z vektoru Z získali seznam prvočísel.

Úkol 2: Úlohu si rozdělíme na dvě části: nejprve zjistíme, v jakém pořadí se prvky mají nacházet, a pak je do něj přeházíme. Pořadí popíšeme permutací p, což bude vektor, jehož i-tý prvek bude říkat, na jakém místě se má objevit `x[i]`.

Hledanou permutaci sestrojíme takto: vezmeme direktní součin `x∘.<x`. Ten nám vytvoří matici nul a jedniček, jejíž i-tý sloupec prozradí, které prvky jsou

menší než `x[i]`. Jejich počet (zjistíme redukcí) je samozřejmě roven místu, na kterém se má `x[i]` ocitnout.

Asi nejjednodušší způsob, jak pak prvky prohazovat, je využít toho, že vektor lze indexovat vektorem, a co víc, do takto indexovaného vektoru lze i přiřadit. Stačí tedy použít `x[p]←x` a je prohozeno. Celý program vypadá takto:

```
x[+/x∘.<x]←x.
```

Úkol 3: I zde, tentokrát inspirováni řešením Jirky Eichlera, přidáme jeden rozměr. Vytvoříme matici, která bude mít v každém sloupci kopii vstupního vektoru x:

```
y←x∘.+x×0.
```

Také vyrobíme matici stejné velikosti s jedničkami nad diagonálou:

```
m←(⍳⍴x)∘.<⍳⍴x.
```

Nyní tyto jedničky přeneseme do y (`m∨y`) a použijeme scanování logickým součinem (`∧\`) – tím pádem v i-tém sloupci zbude úsek jedniček, který se zastaví o první nulu následující po i-tém řádku, a za ním už samé nuly. Teď naopak všechna políčka nad diagonálou vynulujeme (`(~m)∧`...). Co jsme dostali? V i-tém sloupci bude nejprve $i-1$ nul, pak souvislý úsek jedniček začínající ve vstupu na pozici i (může být i prázdný, pokud $x[i]=0$), a za ním nuly. Každý maximální úsek jedniček v x se tedy vyskytuje v alespoň jednom sloupci. Teď už stačí jedničky v každém sloupci pomocí redukce `+/` spočítat a druhou redukcí `⌈/` najít maximum:

```
⌈/+/(~m)∧∧\m∨y.
```

Tím se naše okénko do prehistorie uzavírá. Co si z něj odnést do současnosti? Asi hlavně povědomí o tom, že programy můžeme budovat i z jiných základních konstrukcí než podmínek a cyklů, třeba právě z direktních součinů, redukcí a scanování. Právě tyto operace v dnešní době tvoří základ mnoha jazyků pro paralelní programování, protože se jejich provádění dá velice snadno rozdělit mezi více procesorů. Ale o tom třeba zase někdy příště.

Martin Mareš

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Jiří Eichler	SlovanGOL	2	35	224.2
2.	Vojtěch Kolář	GNeratov	4	31	188.5
3.	Vojtěch Hlávka	GŠlapanice	1	29	185.7
4.	Filip Hlásek	GMikulášPL	3	20	167.0
5.	Pavol Rohár	GMRŠKošice	4	27	154.3
6.	Vlastimil Dort	GŠpitálsPH	4	18	134.0
7.	Štěpán Šimsa	GJungmanLT	1	16	114.9
8.	Ondřej Hübsch	ZŠJílovsPH	0	15	112.7
9.	Miroslav Olšák	GBuďánkaPH	4	14	108.4
10.	Karel Král	GMost	4	15	98.5
11.	Karel Tesař	SPŠEPlzeň	4	12	80.1
12.	Jiří Setnička	G25březnPH	3	10	71.5
13.	Daniel Stahr	GJungmanLT	3	10	70.2
14.	Ondřej Cífka	GNAlejíPH	1	8	68.1
15.	Pavel Taufer	ArcibisGPH	4	11	58.3
16.	Ondřej Mička	GJírovcČB	1	7	55.4
17.	Petr Hudeček	GCoubTábor	2	7	48.5
18.	Petr Pecha	SPŠsVsetín	3	8	48.1
19.	Jakub Diatel	GSlavičín	2	8	42.3
20.	Mária Mrocková	GJHroncaBA	3	6	41.5
21.	Anna Dresslerová	GJHroncaBA	3	4	40.0
22.	Jan Bok	GJungmanLT	3	4	38.4
23.	Martin Zikmund	GTurnov	2	6	37.1
24.	Ondřej Fiedler	GJungmanLT	3	4	35.1
25.	Petr Čermák	GEBenešeKL	4	5	34.8
26.	Dominik Smrž	GOhradníPH	0	4	34.5
27.	Filip Štědronský	GMikulášPL	3	4	34.3
28.	Lukáš Folwarczný	GKomHavíř	2	3	30.0
29.	Jonatan Matějka	GJírovcČB	0	4	29.5
30.	Martin Holec	GSlavičín	3	4	29.3
31.	Alena Bušáková	GŠpitálsPH	3	5	28.2
32.	Matěj Kocián	GLesníZlín	3	3	25.5
33.	Daniel Šafka	GKepleraPH	3	4	25.1
34.	Martin Mach	GJírovcČB	2	4	24.9
35.	Petra Vahalová	GPlasy	4	3	24.8
36.	Tomáš Novella	GAlejKošic	4	6	23.9
37.	Kateřina Lorenzová	GČeskáČB	3	3	22.9
38.	Jirka Kučera	GZborovPH	3	2	22.5
39.	Lukáš Kripner	G_Litvínov	4	2	22.0

40.	Radim Cajzl	GNoMěsNMor	3	4	18.6
41.	Juda Kaleta	GKlatovy	1	2	18.4
42.	Petr Zvoníček	GSlavičín	4	3	18.2
43.	Jindřich Pilař	GBroumov	2	3	14.9
44.–45.	Dana Marečková	GPatočkyPH	4	2	12.8
	Filip Matzner	GJirsíkaČB	3	2	12.8
46.–47.	Michaela Kochmanová	GMikulášPL	3	2	12.0
	Jakub Červenka	GŠpitálsPH	4	1	12.0
48.	Tomáš Masák	GJirsíkaČB	3	3	10.7
49.	Hynek Jemelík	GJarošeBO	3	2	10.0
50.–51.	Michal Katuščák	SOŠHluboká	2	1	9.5
	Matěj Židek	GBroumov	2	1	9.5
52.–53.	Jerguš Greššák	GRaymanaPV	1	1	9.1
	Tomáš Maleček	GEBenešeKL	4	1	9.1
54.	Pavel Kratochvíl	VOŠGSvětlá	2	2	7.8
55.	Michal Bilanský	GLepařovJČ	4	2	6.6
56.	Karel Hulec	GJirsíkaČB	3	1	6.0

Obsah

Úvod . . . 3
Zadání úloh . . . 3
První série . . . 4
Druhá série . . . 12
Třetí série . . . 18
Čtvrtá série . . . 23
Pátá série . . . 32
Seriál: Erlang . . . 38
Seriál: APL . . . 57
Programátorské kuchařky . . . 62
Kuchařka čtvrté série – vyhledávání v textu . . . 62
Vzorová řešení . . . 69
První série . . . 69
Druhá série . . . 86
Třetí série . . . 98
Čtvrtá série . . . 116
Pátá série . . . 125
Pořadí řešitelů . . . 136
Obsah . . . 139

Martin Böhm a kolektiv

Korespondenční seminář z programování
XXII. ročník

Autoři a opravující úloh:
Martin B̈obříkK̈ruliš, Martin Böhm, Martin Kruliš, Lukáš Lánský,
David Marek, Martin Mareš, Jan Matějka, Jan M̈oskytM̈atějka,
Jitka Novotná, Petr Onderka, Peter Ondrúška, Pavel P̈aulieV̈eselý,
Josef Pihera, Vojta Tůma, Vojtěch Tůma, Mária Vámošová,
Michal Vaner, Pavel Veselý, Michal v̈ornerV̈aner

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 318. publikaci.

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš a Jan Matějka.

S jejich pomocí ročenku vysázel Jan Matějka.

Obrázek na obálce nakreslila Lucie Mohelníková.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 140 stran
Náklad 200 výtisků
Praha 2010

Vydáno pro vnitřní potřebu fakulty.
Publikace není určena k prodeji.

ISBN 978-80-7378-124-8

ISBN 978-80-7378-124-8