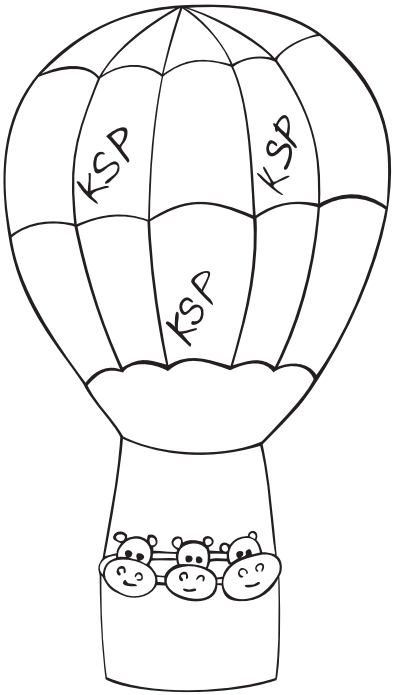
MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXIII. ročník – 2010/2011



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

MARTIN BÖHM A KOLEKTIV

Korespondenční seminář z programování

XXIII. ročník – 2010/2011

matfyzpress

Praha 2011

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

ISBN 978-80-7378-195-8

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý třetí 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 okořeněné 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, 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 o 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 několik 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 před tím, než je donese pošta. V předchozím ročníku 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
První série

Milý deníčku,

oproti minulému týdnu, kdy jsem slavila osmé narozeniny a dostala hříbě, byl tenhle strašný. Matka mi sehnala nového učitele matematiky, který je ještě ošklivější než ten předchozí. Navíc se jí prý mám věnovat o dvě hodiny týdně víc.

Už takhle skoro nedělám nic jiného. Moc ráda bych teď, když umím flétnu, začala hrát na klavír a slečna Jane, naše komorná, mi nabídla, že mě to naučí, ale matce se to nelíbilo, že prý mi sežene pořádného učitele, ale že mám na hudbu dost času.

Včera se mi povedlo utéct z domu a zašít se s Charliem do dílny. Měl tam schovanou žábu. Povídal mi, že od svého táty, našeho zahradníka, slyšel, že můj táta válčí v Řecku. Přišlo mi to divný, ale prý to ví fakt určitě.

Zítra mě čekají nerovnice. Netěším se.

23-1-1 Básníkův deník — 9 bodů

Zahradníkův syn měl pravdu: otec naší hrdinky, slavný romantický básník lord Byron, se na začátku roku 1824 skutečně účastnil Řecké války za nezávislost na Otomanské říši. Jako záminku (či, chcete-li, motivaci) pro první úlohu nového ročníku našeho semináře si vezmeme jeho deník, ve kterém popisoval své výlety a cesty Řeckem předtím, než 19. dubna umřel.

Řekněme, že si každý večer před spaním zaznamenal, o kolik metrů níže či výše za uběhlý den sestoupil či vystoupil. Váš program dostane seznam těchto údajů (jedno *celé číslo* za každý den) na vstupu a vaším úkolem je vypsat dvě čísla:

- V jaké výšce se lord Byron na konci každého dne nacházel nejčastěji? Je-li více možných odpovědí, vypište libovolnou z nich.
- V jaké největší nadmořské výšce jeho lordstvo přenocovalo?

Předpokládejte, že putování začalo u mořské hladiny (této výšce přiřadíme, jak je zvykem, nulu), a nikdy pod mořskou hladinu nesestoupilo.

Příklad vstupu: `103 20 -20 50 -82`
Odpovídající výstup: `nejčastěji: 103, nejvýše: 153`

Vážený pane de Morgane,

jsem vám velice vděčna, že jste si i na cestách našel čas a sepsal mi obsáhlý dopis plný úloh, jejichž řešením se poslední dva týdny těším. Věžte prosím, že matematiku studuji stejně pilně jako pod Vaším laskavým dozorem a že až na den mých patnáctých narozenin nebyla má pozornost odvedena od počtů ničím zásadním.

V příloze vám zasílám některé výsledky a upřímně doufám, že v nich nejsou ony trapné numerické chyby, které mne poslední dobou tak pronásledují.

S netrpělivostí v srdci vyhlížím váš návrat,

Ada

23-1-2 Jedna geometrická — 10 bodů

Z množství fiktivních úloh od pana de Morgana vybíráme dvě takové, které dává smysl zpracovávat moderní výpočetní technikou.

Mějme v kartézské soustavě souřadnic zadány body, jejichž souřadnice obecně nemusí být celočíselné. Ptáme se na nejmenší kruh, který je všechny obsahuje – tedy kde je jeho střed a jaký má poloměr.

Příklad vstupu: `[0;0] [0;1] [1;0] [2;2]`
Odpovídající výstup: `S=[1;1], r=1.4142135`

23-1-3 Jedna maticová — 11 bodů

Na vstupu dostaneme matici, tj. dvojrozměrné pole celých čísel, která má navíc tu zvláštní vlastnost, že jsou čísla v každém jejím řádku a sloupci ostře rostoucí. Potřebovali bychom rychle zjistit, zdali v ní neexistuje nějaké políčko v i-tém řádku a j-tém sloupci, které by mělo hodnotu přesně $i + j$.

Pokud hledaných políček existuje víc, můžete vypsat libovolné z nich. Pár bodíků navíc si můžete vysloužit, pokud vymyslíte, jak rychle spočítat, kolik takových políček je.

Při zvažování časové složitosti nepočítejte dobu načítání: představujte si, že už máte matici v paměti. Zkuste zdůvodnit, proč nelze dosáhnout rychlejšího řešení.

Příklad vstupu:

```
-3  1  4
 4  5  6
 7  9 11
```

Odpovídající výstup: `1. řádek, 3. sloupec`

Příklad vstupu:

```
 3  4  5
 4  5  6
 5  6  7
```

Odpovídající výstup: `žádné takové políčko není`

23-1-4 Ale co trapné numerické chyby? — 10 bodů

Ada však evidentně s matematikou pomoct nepotřebuje: trápí ji trapné numerické chyby. Mohli byste jí pomoci s tím? Třeba… s dělením?

Na vstupu dostanete dělence a dělitele a váš program by měl na výstup vypsat celý desetinný rozvoj podílu. Pokud je rozvoj nekonečný, vyznačte nejkratší možnou periodu.

Příklad vstupu: `1 2`
Odpovídající výstup: `0.5`

Jiný příklad: `1 3`
Výstup: `0.(3)`

Ještě jeden příklad: `143 56`
Výstup: `2.553(571428)`

Vážený pane Babbagei,

mrzí mne, že Vás zdržuji od důležité práce, ale naše setkání na zahradní slavnosti u pana Dickense mi nedá spát. Byla jsem Vám představena paní Somervillovou jako Ada, plným jménem jsem Augusta Byronová.

Je mi teprve sedmnáct let, ale má matka mi zajistila dobré matematické vzdělání a Vaše myšlenka Diferenciálního stroje, který by zautomatizoval výpočty matematických a technických tabulek, mi přijde úchvatná a podivně samozřejmá.

V příloze Vám zasílám seznam dokumentů a knih, o kterých od paní Somervillové vím, že je vlastníte, a o kterých doufám, že byste mi je, samozřejmě za patřičnou protislužbu, mohl zapůjčit.

Velice ráda bych se s Vámi též opět setkala osobně.

Zdraví

Augusta „Ada“ Byronová

23-1-5 Adina knihovna — 10 bodů

Velká zásilka knih od pana Babbageho co nevidět přijde, Ada si proto musí udělat pořádek v knihovně a uvolnit pro ni zvláštní police. Trvá jí to samozřejmě předlouho, každé zařazení publikace do správné poličky se neobejde bez zběžného projití až pročtení jejího obsahu.

Jakmile to Ada dodělá, napadne ji následující logická hříčka, nad kterou stráví zbytek večera:

Mějme řadu N knih správně seřazenou podle svého názvu (který má každá kniha různý). Nyní ji přeskládáme tak, aby každá knížka byla na pozici právě o K větší nebo menší, než byla po seřazení. Pro jaká K v závislosti na N jde něco takového udělat a kolika způsoby?

(adresováno Charlesi Babbageovi)

Drahý příteli,

v příloze vám zasílám finální text překladu Menabreaova textu společně s poznámkami, na kterých jsme se dohodli. Věnujte prosím pozornost znění popisu výpočtu Bernoulliových čísel, opět jsem do toho hrábla, doufám, že nyní už bez újmy na matematické přesnosti.

Mým dětem se daří dobře, děkuji za optání. Byron už dorůstá do věku, kdy se musím rozhodnout, zdali ho hodlám obtěžovat matematikou, nebo jeho vzdělání nechám obvyklejší humanitní tvar. Je to těžké rozhodování a budu potřebovat Vaši radu.

Ráda bych Vás tu v Ockhamu zase někdy viděla. S mou finanční podporou samozřejmě můžete nadále počítat. Oba víme, kde by peníze, které bych Vám upřela, skončily.

Vaše Ada

23-1-6 Babbageova cesta — 10 bodů

Z posledního dopisu vidíme, že Babbage nemá peněz nazbyt. Samozřejmě se chce za paní Adou, hraběnkou z Lovelace, dostat v co nejkratším čase, ale ze všech možností, ze všech tras, které mu dosažení tohoto nejkratšího času nabízejí, potřebuje vybrat takovou, která je nejlevnější.

Na vstupu dostaneme dopravní mapu Anglie zadanou jako seznam spojení (železničních tras), které vedou mezi různými městy (vždy oběma směry). Pro jednoduchost předpokládejme, že použití takového spojení trvá jednotkový čas. U každého spojení je na vstupu také napsáno přirozené číslo, kolik pana Babbage jeho použití stojí. Také dostanete napsáno, ve kterém městě Babbage začíná a kde bydlí Ada.

Na výstupu vypište posloupnost spojení (neboli *cestu*), která ze všech cest z Babbageho stanoviště do Adina bydliště, které používají nejméně spojení, stojí nejméně peněz, tj. součet ohodnocení všech spojení na cestě je nejmenší. Pokud je takových cest víc, vypište libovolnou z nich.

Ada zemřela v 36 letech na rakovinu dělohy. Nechala za sebou tři děti, které už jsou samozřejmě také dávno mrtvé, a první počítačový algoritmus, který by prý na Analytickém stroji, Babaggeově vylepšené verzi stroje Diferenciálního, skutečně běžel a generoval Bernoulliho čísla.

Vedou se neplodné diskuse o tom, nakolik byl program jejím dílem a nakolik šlo o práci Babbage, který toliko obdivoval její slohové schopnosti. Ada sice jeho snažení podporovala značnými částkami, stejně nemalé peníze ale prohýřila v sázkách na koně. Ke svým dětem prý měla ambivalentní vztah – Diferenciální stroj však považovala za „přítele“.

Sterling s Gibsonem kolem její postavy sepsali steampunkový román Mašina zázraků. Po Adě se jmenuje relativně použitelný programovací jazyk. Ada je skvělá. Ada je krásná. Ada je děvče všech matfyzáků bez děvčat.

Druhá série

Alan Mathison Turing se narodil 23. června 1912 a zemřel o 42 (!) let později. Nevíte-li o něm nic dalšího, než že si nějakým způsobem musel zasloužit, abychom tu o něm psali, můžete začít přemýšlet nad tím, proč tak brzo. Ale radši čtěte dál. Do sebevraždy se možná trefíte, její okolnosti jsou však krajně zajímavé.

Gordon Brown se omluvil 10. září 2009. Vyjádřil se v tom smyslu, že je mu to celé moc líto a že za všechny, kteří díky Turingově práci mohou žít ve svobodě, říká, že... je mu celá věc moc líto. Učinil tak díky petici, kterou zařídil jistý britský programátor.

23-2-1 Balíčky balíčků — 10 bodů

Petice byla samozřejmě elektronická, ale aby ji pan premiér nemohl *zamést pod koberec*, rozhodne se ji její iniciátor John Graham-Cumming vytisknout a zaslat poštou. Poprvé po mnoha letech studuje poštovné a co nevidí?

Výhodné nabídky balíčků! Můžete poslat jeden o váze N kg, dva o váze $N-1$ kg, tři o váze $N-2$ kg, ..., N o váze 1 kg, kde N závisí na ročním období, denní hodině a sjízdnosti silnic. To vás nemusí trápit, N dostane váš program na vstupu.

Dále dostanete váhu H petice v celých kilogramech. Vaším úkolem bude vymyslet, které nabídky balíčků je třeba vybrat, aby se do nich dohromady vešlo H kg petice, ale zároveň aby jejich kapacita byla co nejblíže tomuto H.

Je třeba zdůraznit, že „3 balíčky, každý o váze $N-2$ kg“ je jedna nabídka, kterou jako celek buď přijmete, nebo nepřijmete. Chcete-li poslat $3N-6$ kg, je to ideální volba.

Chcete-li poslat $N-2$ kg a N není úplně malé (třeba $N=100$), je lepší zvolit nabídku „1 balíček o váze N kg“, přestože dva kilogramy nevyužijete. Stejně dobré řešení by pak bylo vybrat „N balíčků o váze 1 kg“ a nám je jedno, které z takových dvou stejně dobrých řešení vypíšete.

Chcete-li poslat 100 kg a $N=12$, můžete vybrat třeba kombinaci $3\times(N-2)+3\times(N-2)+5\times(N-4)=3\times 10+3\times 10+5\times 8$.

Turing byl „zakladatel moderní informatiky“. Co myslíte, dařilo se mu na něco takového balit holky?

Zavedl nejvýznamější teoretický model počítače, kterému se dnes říká Turingův stroj. Můžeme si ho představit jako nekonečnou pásku popsanou symboly z nějaké konečné množiny. Nad páskou se pohybuje hlava stroje a v každém kroku výpočtu podle jednoduché tabulky pravidel přečte symbol, nahradí ho jiným, a přesune se doleva nebo doprava. Existuje teze, že práce každého rozumného (teoretického i skutečného) počítače se dá na práci Turingova stroje převést.

Na základě tohoto modelu dokázal, že neexistuje zaručeně konečný algoritmus, který by dokázal posoudit, zdali se jiný daný algoritmus na daných datech zastaví.

Tím rozhodnul Hilbertův problém z roku 1928, který se ptal po existenci zaručeně konečného algoritmu, který dostane matematické axiomy a domněnku a rozhodne, je-li domněnka z těchto axiomů odvoditelná. Zamítnul existenci takového algoritmu, protože díky formalizaci Turingova stroje uměl vyjádřit „zastaví se algoritmus na daných datech?“ jako matematickou domněnku.

Vymyslel též „Turingův test“. Jde v podstatě o člověkostřednou definici inteligence, kdy je objekt inteligentní právě tehdy, nerozezná-li lidský pozorovatel jeho lingvistický výstup od lingvistického výstupu člověka. Takové Turingovské testování provádí každý z nás vždy, když mu píše neznámá entita po IM a nabízí výrobek.

Co na tom, že ho v mnoha (i zmíněných) věcech asi o rok předběhl Alonzo Church; Turingův přístup se ukázal být stravitelnější než Churchův lambda kalkulus.

23-2-2 Zastavení — 10 bodů

Když už jsme u toho zastavování… Máme-li spravedlivou šestistěnnou kostku, umíme na ní generovat (celá) náhodná čísla mezi 1 a 6 (včetně) se stejnou pravděpodobností 1/6. Představme si, že máme po ruce 4-, 6-, 8-, 12- a 20stěnnou kostku. Pro které hodnoty n umíme pomocí těchto kostek generovat (celá náhodná) čísla mezi 1 a n (včetně) tak, aby všechna padala se stejnou pravděpodobností a trvalo nám to zaručeně konečný počet kroků?

Pokud píšeme generovat, myslíme tím prostě, že nějaký váš algoritmus dostane kostku „zapůjčenou“ a může si s její pomocí generovat náhodná čísla, na jejichž základě nakonec spočítá požadované výsledné náhodné číslo. Jakékoli jiné náhodné generátory jsou zakázány. Nepůjde-li vám to ve vší obecnosti, zkuste ověřit, jestli (a jak) jdou vygenerovat čísla od 1 do 120.

Třeba náhodné číslo v intervalu 1 až 32 snadno získáme na dva hody výrazem $(8(d_4 - 1) + d_8)$, kde d_4 je číslo hozené na čtyřstěnné kostce a d_8 číslo hozené na osmistěnné kostce.

O Turingovi se povídá spousta „geekovských“ drbů. Často se zmiňuje jeho kolo, byl to náruživý cyklista. Začal mu prý jednou padat řetěz a on nedbal opravy, místo toho si při jízdě počítal otočky a pokaždé, když se to mělo stát, seskočil z kola a opatrně posunul řetěz o pár pozic dál.

To zní strašlivě neprakticky, dokud si neosvětlíme, že řetěz padal právě a jen tehdy, sešel-li se jeden ohnutý zoubek na kotouči s jednou nedokonalou pozicí na řetězu. Je pak otázka dělitelnosti, kdy se při jízdě takové dvě chyby setkají: klidně to mohlo nastávat „jen" každých deset minut.

23-2-3 Projížďka — 12 bodů

Představme si Turinga mířícího vlakem do krajiny, kterou si chce na kole projet. Dostaneme na vstupu seznam rozcestí a cest, které vedou mezi nimi. Jeho kolo je tentokrát bezvadné, leč terén je obtížný a hlavně nevyrovnaný – některé silnice jsou krátké a klidné, dokonce vedou z kopce; jiné jsou dlouhé, klikaté a strmé, takže velmi unavují.

Turing každé z nich při pohledu do mapy přidělil celé číslo vyjadřující tuhle subjektivní obtížnost – na kladně ohodnocených cestách si bude odpočívat a na záporně ohodnocených tuhle nashromážděnou energii vydá.

Teď by od vás chtěl, abyste napsali program, který mu najde takovou cestu (včetně začátku – a může začínat na libovolném rozcestí), po které jednak projede všechny silnice *právě jednou*, druhak bude na každém rozcestí součet všech Turingem do té chvíle projetých silnic *nezáporný* a navíc se vrátí na rozcestí, na kterém začal. Chcete-li a myslíte-li si, že vám to pomůže, předpokládejte klidně, že z každého rozcestí vychází sudý počet silnic.

Důležité je, že Turingovy vědecké výsledky blednou ve srovnání s jeho srdnatostí za druhé světové války. Účastnil se dešifrování německého šifrovacího stroje Enigma v anglickém Bletchley Parku, postavil při této příležitosti jednoúčelový počítač Bombe, který podstatně urychlil procházení možností. Díky tomu, že byly kódy Německa zlomeny, nabrala válka poněkud jiný rozměr, který hezky zachytil Neal Stephenson ve své knížce Kryptonomikon (ve které mimochodem Turing skutečně vystupuje):

„[Ve filmech] se praví, že Patton a MacArthur jsou odvážní generálové. Svět bez dechu očekává jejich další neohrožené eskapády za nepřátelskou linií. Waterhouse ví, že Patton a MacArthur jsou víc než cokoliv jiného inteligentní konzumenti [prolomených šifer] Ultra/Magic. Používají je k tomu, aby zjistili, kde nepřítel soustředil síly, pak ho obejdou a udeří na místo, kde je nejslabší. To je všechno."

23-2-4 Plánování — 10 bodů

Pořád by ale bylo poněkud nespravedlivé upřít všem spojeneckým generálům jakoukoliv tvořivost. I když vám cizí zprávy diktují, na která místa potřebujete kdy zaútočit, pořád máte omezené zdroje.

V této úloze dostanete na vstupu seznam časových intervalů zadaných přirozenými čísly, ve kterých je potřeba likvidovat nějaký výhodný cíl ploužící se za frontovou linií. Seznam je uspořádaný podle počátků těchto intervalů.

Chceme od vás, abyste našli minimální počet bombardérů, který stačí k likvidaci všech cílů, a jejich časový rozvrh.

Neuvažujte doby přilétání a odlétání, tankování, údržby a podobně, to už je započítáno v intervalech. Letadlo je vždy plně využito celý požadovaný interval, takže se nesnažte o nějaké triky s předčasným návratem, jedním letadlem na dvou místech apod.

Příklad vstupu:

```
5-8 7-12 11-13 12-15
```

Příklad výstupu:

```
2
1: 5-8 11-13
2: 7-12 12-15
```

23-2-5 Zaměřování — 8 bodů

Historicky se matematika ve válkách používala vedle šifrování také při všemožné balistice. Nabízíme vám touto historií velmi vzdáleně inspirovanou úlohu:

Dostanete na vstupu pozici dvou kanónů v kartézské soustavě souřadnic a po řadě vrcholy i nekonvexního mnohoúhelníka (ale žádné dvě jeho nesousedící hrany se neprotínají), na jehož obvod šílený velitel přikázal střílet. Souřadnice nemusí být celá čísla.

Navíc vyžaduje, aby oba kanóny střílely na takový bod na obvodu, pro který platí, že je obsah trojúhelníka určeného oběma kanóny a tímto bodem co nejbližší zadanému číslu. Pokud je jich více, vypište libovolný z nich.

Příklad vstupu (kanóny, pevnost, obsah):

```
[1, 1] [1, 2]
[2, 1] [2, 2.5] [3.71, 2.5] [3.71, 6] [7, 1]
1
```

Odpovídající výstup může být třeba `[3, 1]`.

Po válce pracoval Turing na stavbě počítačů, tentokrát už ne jednoúčelových, a nějakou dobu se mu dařilo konkurovat podstatně lépe financovanému americkému výzkumu.

23-2-6 Testovací — 10 bodů

Potřebujeme-li u takového jednoduchého počítače otestovat bezchybnost, chceme nějakou dosti jednoduchou úlohu, po jejímž vyřešení a naprogramování si budeme moci být jisti, že pokud dává počítač špatné výsledky, není to naším naprogramováním.

Co třeba takovou?

Máte zadanou setříděnou posloupnost přirozených čísel a chcete vypsat všechny trojice ve tvaru a, $a + k$, $a + 2k$ (pro všechna možná přirozená a, k), které se v ní nacházejí.

Příklady (vstup → výstup):

`1 2 3 5 8 9` → `1-2-3 1-3-5 1-5-9 2-5-8`

`1 2 4 5 10 11` → nic (zde není žádná taková trojice)

Turing byl také mimochodem homosexuál, v Británii to bylo do roku 1968 trestné (u nás „jen“ do roku 1960), a tak mu soud, když se na to přišlo, zakázal pracovat na vládních projektech na výstavbu počítačů a nařídil hormonální „léčbu.“ O dva roky později si Turing kousl do jablka, které předtím naplnil kyanidem. Bizarní? Měl moc rád Sněhurku a sedm trpaslíků od Disneyho – inspiraci tedy možná našel v tomto filmu.

Každopádně se všeobecně soudí, že ho k tomu dohnaly dosti nepěkné vedlejší účinky prováděného léčení a to je oním důvodem, proč se mu britský premiér po 55 letech omluvil.

Mezi informatiky je Turing oblíbený, protože jeho životní příběh dokumentuje, jak může být takový teoretik užitečný, když vyvstanou velké praktické problémy. Existují odhady, podle kterých analytici z Bletchley Parku zkrátili válku o rok a zachránili milion lidí, a je samozřejmé nemožné říct, jestli tomu tak je. Je rozhodně dojemné si uvědomit, že po válce samozřejmě jako hrdinové oslavováni nebyli, protože britská vláda nechtěla, aby se o prolomení daných šifer vědělo. Mnoho jich tedy, stejně jako Turing, zemřelo bez jakéhokoliv uznání.

V češtině vyšla v edici Aliter popularizační knížka „Muž, který věděl příliš mnoho,“ která se celá věnuje Turingově životu a snaží se jemně vysvětlovat jeho výsledky. Pokud vás zajímá teoretická informatika a chcete být drsní, Charles Petzold nedávno sepsal „Annotated Turing,“ což je přetisk Turingovova ústředního článku s poznámkami.

Existuje docela známá beletristická knížka o kryptoanalyticích z Bletchley Parku a špionážních tanečcích kolem, která se jmenuje „Enigma“ – dokonce podle ní natočili film. Tam už ale našeho hrdinu nenajdete.

Třetí série

Edsger Dijkstra byl slavný holandský myslitel. Byl to samotářský, konzervativní člověk, kterému se jen velmi těžko určuje obor, jímž se zabýval. Narodil se roku 1930 v Rotterdamu, kde zůstal až do svých vysokoškolských studií teoretické fyziky. Později žil a učil v Eindhovenu, kde se věnoval praktické i teoretické matematice a informatice.

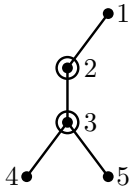
Zabýval se mimo jiné i grafy. Jedním z nejznámějších algoritmů, které vymyslel, je hledání nejkratší cesty v ohodnoceném grafu, jenž po něm nese jméno a můžete si ho přečíst třeba v naší kuchařce.[1] *My bychom po vás teď chtěli vymyslet jeden trochu jednodušší, ale zdánlivě podobný algoritmus.*

23-3-1 Úsporný kořen — 9 bodů

Na vstupu dostanete neohodnocený strom[2] zadaný počtem vrcholů a seznamem hran. Vrchol s hloubkou x bude takový vrchol, od kterého je každý jiný vrchol vzdálený maximálně x. Úsporný kořen je vrchol s nejmenší možnou hloubkou. Nalezněte všechny úsporné kořeny stromu.

Příklad vstupu:

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



Výstup: `2 3`

Jako správný samotář bydlel na vesnici, takže do školy jezdil jen v úterý. Vedl tam i seminář, kterému se příhodně říkalo Tuesday Afternoon Club, kde se řešily úlohy podobné těm z KSP. Kdykoliv tam či jinde studenti příliš hlasitě šuškali (to asi dobře znáte), nekřičel, ale naopak začal šeptat. To mělo ohromný efekt – všichni ztichli. Takový měl respekt.

Když se později stěhoval do Ameriky a cesta mu připadala dlouhá, vymyslel úlohu, kterou pak zadal americkým studentům při jejich prvním semináři Tuesday Afternoon Club.

23-3-2 Nejkratší cesta přes oceán — 14 bodů

⚠ Pro zjednodušení si severní Ameriku i Evropu představme jako dva konvexní n-úhelníky, které se neprotínají. Chcete nalézt nejkratší cestu mezi nimi. Na vstupu nejdříve dostanete souřadnice Ameriky a potom souřadnice Evropy jako vrcholy v pořadí, v jakém leží na obvodu.

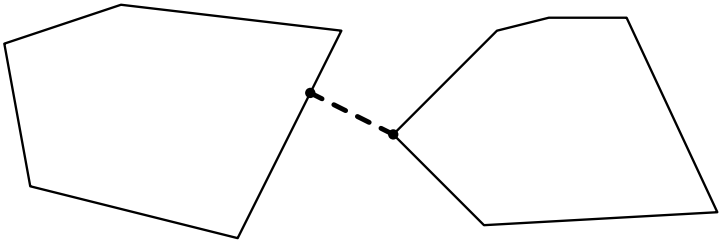
[1] `http://ksp.mff.cuni.cz/viz/kucharky/halda-a-cesty`
[2] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

Vaším úkolem je najít takové dva body, jeden na obvodu Ameriky a druhý na obvodu Evropy, aby jejich vzdálenost byla co nejmenší. Pokud je víc možností, stačí vypsat libovolnou z nich.

Vstup (znázorněný obrázkem):

```
5
0 75
10 20
90 0
130 80
45 90

6
185 5
275 10
240 85
210 85
190 80
150 40
```



Výstup (čárkovaně):

```
118 56
150 40
```

Krom tisíce jiných věcí přemýšlel, jak počítače naučit počítat, třeba odpovědět na zadání `1+2*3`. K tomu účelu znovu objevil postfixový zápis a objevil, jak na něj běžný infixový zápis rychle převést. Pokud to chcete umět taky, můžete se podívat třeba do Wikipedie.[3]

Abychom nezůstali u teorie, podílel se na vývoji programovacího jazyka ALGOL 60 a později i jeho prvního překladače. Tento jazyk vznikl pro snadný zápis algoritmů a jako konkurence tehdy mohutně nasazovaného BASICu.

Edsger Dijkstra vůbec velmi brojil proti příkazu `goto` a zasazoval se o strukturované programování. Příkaz `goto` považoval za nepřehledný. Samozřejmě, že na úrovni procesoru se stále používá, ale programátor by od něj měl být odstíněn, pokud to jen jde.

Měl rád programování rovnou na čisto, nejdřív si program rozmyslet a pak jej plynule psát. Lepší je chyby nedělat, než je hledat.

Následující úlohu vymyslete rovnou bez chyb a tak, aby šla zapsat bez `goto`. Pokud nevíte, co to `goto` je, máte to snazší, buďte jen rádi.

[3] `http://cs.wikipedia.org/wiki/Shunting-yard_%28algoritmus%29`

23-3-3 Skok bez padáku — 13 bodů

Z letadla vyskočil Američan, leč až po výskoku si uvědomil, že místo padáku si vzal batoh spolucestujícího Čecha. Naštěstí pro něj je na stráni pod ním rozmístěno N trampolín.

Představme si stráň jako rovinu postavenou svisle, tedy souřadnice x určuje horizontální pozici a souřadnice y je výška.

Každá trampolína je určena dvojicí souřadnic (x, y). Parašutista má nějakou počáteční pozici (x_0, y_0). Padá ve směru osy y, dokud nenarazí na trampolínu, která je o d níže než on. Od ní se odrazí a vyletí o $d/2$ výše, pak si může vybrat, jestli se posune o 1 vlevo, nebo vpravo (posunout se musí) a posune se podle toho.

Toto se opakuje, dokud nedopadne na zem. Na vstupu také dostanete výšku h; pokud spadne na zem z větší výšky, způsobí si zranění neslučitelná se životem a přivolaný lékař konstatuje smrt.

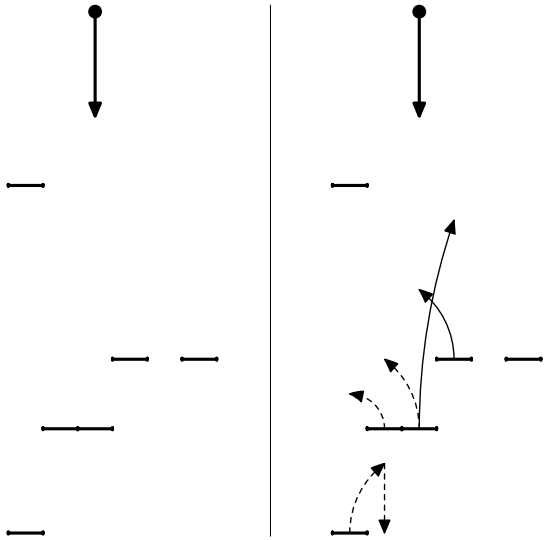
a) Určete, jestli má šanci přežít, a pokud ano, jak má postupovat. Nalezněte nejkratší možné řešení (nejméně odrazů). (5 bodů)

b) Nalezněte všechna možná $y \leq y_0$, pro která přežije pád s počáteční pozicí (x_0, y). (8 bodů)

Příklad vstupu: Na prvním řádku jeho počáteční souřadnice a h_0, na druhém řádku K, na dalších K souřadnice trampolín.

Na obrázku vlevo náčrt zadání, vpravo možné řešení (nejprve skáče po plných, následně po čárkovaných šipkách).

```
2 15 3
6
0 0
0 10
1 3
2 3
3 5
5 5
```



Zasazoval se o eleganci nejen zdrojových kódů, ale i matematických důkazů. Ty jeho byly zvlášť pěkné. Málokdy přesáhly 16 stran a každou větu pečlivě vybíral, aby nebyla zbytečná, nudná ani nepochopitelná. Důkazy psal jako pohádky. Neměl rád dlouhé formální řady implikací ani důkazy sporem, zato zvládl i třeba v geometrii nebo algebře použít algoritmické důkazy a jiné překvapivé finty z programátorského světa.

Jeho konzervativní přístup k vědě se projevoval třeba tím, že nerad jezdil na konference, ale raději vykládal v menší skupině lidí. Většinu své práce psal na stroji. Osobní počítač si pořídil až ke konci života, ale i tehdy ho používal minimálně a preferoval psaní rukou. Měl krásné technické tiskací písmo, které se používá i jako počítačový font.

23-3-4 Psaní písmen — 10 bodů

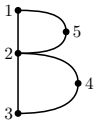
Každé písmeno se skládá z bodů a linií, které je spojují. V jednom bodě může začínat i končit více linií. Při psaní perem lze psát víc navazujících linií jedním tahem, nejde-li to, musí se pero zvednout a začít jinde. Kolikrát nejméně je potřeba pero zvednout?

Na vstupu dostanete neorientovaný graf o N vrcholech a M hranách a vypište, kolika nejméně tahy lze nakreslit.

Samozřejmě šetříme, takže je zakázáno jakoukoli hranu nakreslit více než jednou. Jinak řečeno, nesmíte se vracet po již nakreslených liniích.

Příklad vstupu:

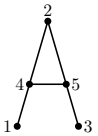
```
5 6
1 2
2 3
3 4
4 2
2 5
5 1
```



výstup: `1`

Jiný příklad:

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



výstup: `2`

Jeden z mnoha jeho textů (EWD 1250) – známý problém dvou párů, loďky a řeky. K řece přišly dva páry a potřebují překonat řeku tak, aby nikdy nezůstal sám pán z jednoho páru s dámou z druhého páru. Loďka unese maximálně dva lidi.

The couples, the river, and the little boat

Here is the problem:

"Two husbands and two wives have to cross a river in a boat which can hold only two people. How can they cross so that no woman is in the company of a man unless her husband is also present?"

(Copyright © 1967 by Morris Kline)

Pamatoval i na ty, kteří už úlohu znali, nebo hned vyřešili. „Žádost: Pokud vám připadá tento problém příliš jednoduchý na to, abyste na něj plýtvali svým časem, hledejte prosím místo toho počet různých řešení. Děkuji. ⟨Konec žádosti.⟩“

Request If you think this problem too trivial to waste your time on, please state instantaneously the number of different solutions. Thank you. (End of Request.)

Mimochodem, někteří organizátoři KSP mají velmi podobné písmo… také vám připadá výrazně čitelnější než klasické psací písmo?

Někdy mu bylo vyčítáno, že neuváděl žádné nebo málo zdrojů. Ale co měl dělat, když něco vymyslel jen tak? Navíc většinu své práce nepublikoval v časopisech, ale posílal přátelům a známým. Tyto články jsou označovány EWD (jeho iniciálami) a číslem, třeba EWD 1250, a dají se stáhnout volně na internetu.[4]

Analogii můžeme najít v hudbě, kde se takhle označují díla slavných skladatelů, asi nejznámější jsou BWV (Bach-Werke-Verzeichnis) a HWV (Händel-Werke-Verzeichnis). Zásadní rozdíl je ovšem, že Dijkstra si to čísloval sám, kdežto hudebníci to neřešili a udělal to za ně někdo jiný o mnoho let později.

EWD shromažďuje pan Ham Richards. Jednou, když je nesl, zakopl a rozsypaly se mu po podlaze.

[4] `http://www.cs.utexas.edu/users/EWD/welcome.html`

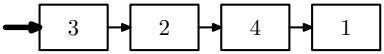
23-3-5 Rozházené EWD — 7 bodů

Chudák pan Richards má jen svou zapomětlivou hlavu a pár papírů, tak budete muset vymyslet, jak setřídit EWD v konstantní paměti. To jest, že si může udělat třeba 1000 záznamů, ale ne pro každou z N EWD jeden. Vámi spotřebovaná paměť prostě na N vůbec nesmí záviset (a N může být libovolně velké – argument, že EWD je konečně mnoho, vám neprojde). Dávejte si pozor na rekurzi, spotřebovává tolik paměti, jak hluboko je zanořená.

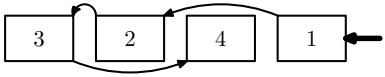
Přeházená EWD budeme reprezentovat jako spojový seznam. V programu dostanete ukazatel na první prvek spojového seznamu, kde je číslo EWD a ukazatel na další. Vaším úkolem je ho setřídit a vrátit ukazatel na první prvek (nejstarší EWD).

Spojový seznam už máte v paměti, vaším úkolem je přepojit jej do setříděného stavu.

Příklad před setříděním:



A po setřídění:



Z úplně jiného soudku je jeho návrh operačního systému THE multiprogramming system, zaměřeného na sekvenční zpracování úloh a s podporou multitaskingu a paralelizace. Velkého rozšíření se sice nedočkal, nicméně myšlenky vytvořené pro něj se ujaly. Jestli vás paralelní svět zajímá, měli jsme o něm kdysi seriál[5] *a také o něm byla série úloh před pár lety v Matematické olympiádě.*

Jako už starý se vrátil do Holandska do svého původního domu ve vesnici Nuen, kde roku 2002 zemřel na rakovinu.

Pozůstalí pak přemýšleli, jestli mu na hrob napsat, že byl matematik, nebo informatik. Hodilo by se jim k rozhodování vědět, kolik procent lidí si nejčastěji myslelo, že byl informatik.

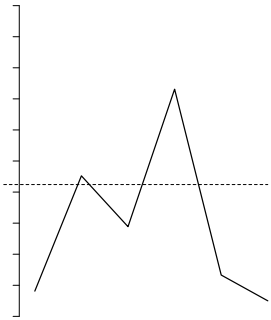
[5] `http://ksp.mff.cuni.cz/viz/12`

23-3-6 Výzkum veřejného mínění — 9 bodů

Za jeho života se dělalo N výzkumů veřejného mínění, jestli byl informatik. Výsledkem bylo vždy číslo v procentech s přesností na M desetinných míst. N je řádově tolik jako 2^M. Můžeme předpokládat, že výzkumy odpovídaly realitě a mezi jednotlivými výzkumy procento lidí, kteří si myslí, že ano, buď jen stoupalo, nebo jen klesalo. Vaším úkolem je zjistit, jaké procento bylo během jeho života nejčastější. Případně určit libovolné z nejčastějších. Nezapomeňte, že to nutně nemuselo být v době, kdy se konal výzkum.

Příklad vstupu:

```
6 5
8.124 45.223 28.8723 73.117 13.3 5.0
```



Výstup (vyznačen čárkovanou čarou):

`42.42` (4×)

Čtvrtá série

Jen výjimečně se najdou lidé píšící programy s dokumentací obsáhlejší než samotný kód a provádějící veškerou činnost s obdivuhodnou pečlivostí. V této sérii si představíme osobnost, pro niž je psaní dobře okomentovaných programů takřka denním chlebem a která povýšila programování na umění programování.

Možná už tušíte, že se jedná o Donalda Ervina Knutha, a vybavilo se vám jeho dílo Umění programování (v originále The Art of Computer Programming), obsahující mnohé znalosti informatiky, popisy základních algoritmů a jejich matematickou analýzu. Výjimečnost tohoto muže potvrzuje i titul emeritního profesora (plným názvem Professor Emeritus of The Art of Computer Programming) na Stanfordově univerzitě v USA.

Emeritní znamená, že odešel z profesionálního života, nicméně mu zůstal čestný profesorský titul. Knuth se totiž rozhodl opustit univerzitu, aby se mohl plně věnovat práci na Umění programování. Chce-li totiž napsat o nějakém tématu, hluboko se do něho ponoří, přečte o něm spoustu článků, z nichž vybere pro čtenáře nejpřínosnější poznatky, každý algoritmus si naprogramuje...

Proto také jeho práce na Umění programování trvá již od roku 1962, nyní jsou napsány tři svazky (Základní algoritmy, Seminumerické algoritmy, Vyhledávání a třídění), čtvrtý (Kombinatorické algoritmy) se dokončuje, ale v plánu jsou ještě další tři.Celkem by tedy mělo vyjít sedm svazků.

23-4-1 Studenti a profesoři — 12 bodů

Jedna z věcí, jež ho na univerzitě zaměstnávala (a tedy mu ubírala čas od psaní), bylo vedení vědeckých prací studentů (např. psaní článků do odborných časopisů).

Studenti na Stanfordově univerzitě se chtějí prosadit a napsat co nejvíce článků, přičemž každý z nich má vytipováno několik profesorů, pod jejichž vedením by chtěl článek psát. S jinými profesory spolupracovat nechce a nebude.

Studenti jsou schopni psát maximálně K článků najednou. Leč čas profesorů je omezený, každý z nich je totiž ochoten spolupracovat maximálně s K studenty, přičemž je jim jedno, kteří to budou.

Vaším úkolem je najít algoritmus, který zjistí, jestli je možné, aby každý student psal právě K článků a každý profesor spolupracoval právě s K studenty, a pokud ano, tak vypsat, který student bude spolupracovat s kterým profesorem.

Můžete předpokládat, že profesorů i studentů je stejně, totiž N, a nemusíte uvažovat situaci, že by student chtěl psát u jednoho profesora více článků.

Příklady: pro vstup $N = 4$, $K = 2$, student S1 chce psát článek s profesory P1 a P2, student S2 s P1, P2, P3, P4, student S3 s P2, P3, P4 a student S4 s profesory P3, P4, jsou řešením tyto páry student–profesor: S1–P1, S1–P2, S2–P1, S2–P3, S3–P2, S3–P4, S4–P3, S4–P4.

Pro vstup $N = 5$, $K = 2$, studenti S1 a S2 chtějí psát u profesorů P1, P2, P3, student S3 u P3, P4, P5 a studenti S4, S5 u P4, P5, řešení neexistuje. Existovalo by, kdyby bylo $K = 1$, ale to už je zase jiný vstup.

Aby byl Knuth při své práci co nejméně vyrušován, zrušil 1. ledna 1990 e-mailovou adresu. Jak píše na svém webu, cítí se nyní šťastnější, protože mu už nechodí nevyžádaná pošta. I přesto výhod elektronické komunikace a internetu stále využívá, ale většinu e-mailů za něj vyřizuje jeho sekretářka.

23-4-2 Paralelní profesoři — 10 bodů

Na chvíli si představme, že neexistuje nic jako e-mail, internet, ba dokonce obyčejná „šnečí“ pošta. Jak by si takoví profesoři sdělovali své poznatky?

Na univerzitě pracuje N profesorů. Na začátku dne má každý svůj unikátní nový objev, který chce sdělit všem ostatním.

Jelikož však ve skupince třech a více profesorů je vzájemné sdělování poznatků nemožné kvůli překřikování, pořádají profesoři během dne občas sezení. Během nich utvoří dvojice (ne nutně všichni jsou ve dvojici) a ve dvojicích si vymění všechny objevy, které mají (tedy i objevy získané dříve od jiných profesorů). Během sezení se dvojice nemění.

Jaký je pro různá N minimální počet sezení, která musí proběhnout, aby každý profesor znal objevy všech svých kolegů?

Třeba pro $N = 2$ stačí jedno sezení, pro $N = 4$ jsou potřeba dvě (nejdřív A–B a C–D a potom A–C a B–D).

Jak D. E. Knuth nedávno v jednom rozhovoru poznamenal, mnohdy naráží na odborné texty, jejichž jediným cílem je překonat tajuplnými metodami jednodušší algoritmy, ale pouze, když bude velikost vstupu větší než počet protonů ve vesmíru. Ve svých knihách proto předkládá jednodušší, ale stále efektivní metody.

Jak však ukáže následující úloha, ne vždy jsou jednoduché a rychlé metody dobré.

23-4-3 Zabugovaný program — 8 bodů

Dva zlatokopové objevili bohaté ložisko zlata a společně vykopali velké množství nugetů. Chtějí se o ně rozdělit rovným dílem, na což si napsali program. Každý nuget má svoji zadanou cenu (přirozené číslo), seznam nugetů program dostane na vstupu, na výstup vypíše, jak si je mají rozdělit, nebo oznámí, že řešení neexistuje.

Například pro nugety o hodnotách 3, 3, 5, 5 dostanou oba nugety s cenou 3, 5; pro sadu nugetů 3, 3, 5 řešení neexistuje.

Co čert (nebo spíš zlatokop) nechtěl, v programu je chyba a ne malá, nehledejte tedy zapomenutý středník, chybu v použití knihovní funkce jako `qsort` nebo neošetření čtení vstupu. Zlatokopové špatně vymysleli celý algoritmus a program by si zasloužil od základu přepsat.

To však nechte na nich, ať se pocvičí v algoritmizaci, vaším úkolem bude pouze přesvědčit je, že program napsali špatně – najít jim vstup, na němž vypíše špatný výsledek, a určit pro tento vstup správný výsledek. Bonusové body neminou řešitele, kteří najdou nejmenší takový vstup co do počtu předmětů nebo celkové ceny.

Zdrojový kód (C):

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

#define MAX 1000
int N;                  // počet nugetů
int ceny[MAX];          // ceny nugetů
int la[MAX], lb[MAX]; // co dostane kdo

// Vypíše zadané pole o N prvcích
void vystup(int *pole, int N) {
  for (int i=0; i<N; i++) {
    if (i>0)
      printf(" ");
    printf("%d", pole[i]);
  }
  printf("\n");
}

// Porovnávací funkce pro qsort()
int cmp(const void *a, const void *b) {
  return (*((int *)b) - *((int *)a));
}

int main(void) {
  // Přečteme vstup
  scanf("%d", &N);
  for (int i=0; i<N; i++)
    scanf("%d", &ceny[i]);

  // Setřídíme vstup
  qsort(ceny, N, sizeof(int), cmp);

  int a=0, ai=0, b=0, bi=0;
  for (int i=0; i<N; i++) {
    // Který zlatokop má zatím méně?
    if (a < b) { // A -> přidáme A
      la[ai] = ceny[i];
      ai++;
      a += ceny[i];
    } else {     // B -> přidáme B
      lb[bi] = ceny[i];
      bi++;
      b += ceny[i];
    }
  }

  if (b == a) { // Povedlo se rozdělit
    // Tak to vypíšeme
    vystup(la,ai);
    vystup(lb,bi);
  } else        // Nepovedlo se rozdělit
    printf("Nelze spravedlivě rozdělit.\n");
  exit(0);
}
```

Zdrojový kód (Python):

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Čtení vstupu (zde není chyba)
# N je počet nugetů
# ceny je pole s jejich cenami
N = long(raw_input())
s = raw_input()
ceny = map(long,s.split(" "))
if len(ceny) != N:
        print "(Po)Chybný vstup"
        exit(1)
# Vstup úspěšně přečten

# Setřídíme ceny
ceny.sort()

A = 0
B = 0
LA = []
LB = []

# a rozházeme
while len(ceny) > 0:
        c = ceny.pop()
        if A > B:
                LB.append(c)
                B += c
        else:
                LA.append(c)
                A += c

# Výpis (zde není chyba)
if A == B:
        print " ".join(map(str,LA))
        print " ".join(map(str,LB))
else:
        print "Nelze spravedlivě rozdělit."
```

Zmiňme ještě trochu biografických údajů. Donald Ervin Knuth se narodil v roce 1938 ve Wisconsinu. Už od mládí vykazoval vysokou inteligenci, když v 8 letech dokázal složit z písmen „Ziegler's Giant Bar" 4500 anglických slov do jedné soutěže, přestože porota měla jen 2500 slov. V roce 1956 nastoupil na Case Institute of Technology na fyziku, ale byl záhy přesvědčen, aby se věnoval matematice.

K informatice se dostal v podstatě náhodou při své letní práci, když narazil na počítač IBM 650. Zaujal ho natolik, že u něj strávil dlouhé hodiny jeho zkoumáním. Ve 20 letech napsal program na analýzu výkonnosti univerzitního basketbalového týmu, který mu vynesl trochu slávy. Tomuto počítači dokonce později věnoval jednu svoji práci slovy „na památku mnohých příjemných odpolední".

S pracemi na Umění programování začal již v roce 1962 (tedy 6 let po nástupu na vysokou školu) a první tři svazky vyšly v letech 1968, 1969 a 1973. Poté byl však zklamán změnou techniky sazby jeho knih, protože se změnilo písmo a snížila kvalita. Rozhodl se proto vyvinout vlastní systém pro sazbu textů, a tak vznikly jeho dva další známé počiny: TEX jako nový systém pro sázení textů a METAFONT *pro tvorbu fontů.*

Dotáhl sazbu a propracovanost svých knih dokonce tak daleko, že se rozhodl vyplatit 0x$1.00 (jeden hexadecimální dolar, tedy $2.56) každému, kdo najde v nějaké jeho knize chybu. Na svých stránkách má seznam odměněných, který dnes čítá bezmála 400 jmen.

23-4-4 Závorky v TEXu — 10 bodů

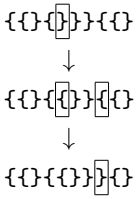
V TEXu se hojně využívají složené závorky (např. v definicích či voláních maker) a lze je i libovolně vnořovat. Občas se ale stane, že se člověk překlepne a místo `{` napíše `}` nebo naopak.

Vymyslete algoritmus, který na vstupu dostane posloupnost (řetězec) N složených otevíracích a zavíracích závorek (bez dalších jiných znaků) a nalezne minimální počet změn znaku `{` na znak `}` nebo naopak, aby byl řetězec správně uzávorkovaný. Žádné jiné změny, kromě přepsání jednoho znaku na druhý, nejsou povoleny.

Správně uzávorkovaný znamená, že ke každé otevírací závorce existuje odpovídající uzavírací, která je od ní napravo, a podobně ke každé uzavírací existuje odpovídající otevírací, jež je od ní nalevo.

Nepůjde-li posloupnost závorek žádným způsobem změnit na správně uzávorkovaný řetězec, měl by to být váš algoritmus schopen rozpoznat.

Příklad: pro vstup `{{}{}}}{{}` je jedním z možných nejkratších postupů ke správnému uzávorkování tento:



Výsledek je tedy 2.

Jak je uvedeno na začátku, Knuth píše dobře dokumentované programy. V 70. letech, kdy vznikal TEX, si vymyslel dokonce vlastní styl programování, v originále nazvaný literate programming (česky by se dalo přeložit jako „kultivované programování“), který se snaží programování více přiblížit myšlení člověka a ne potřebám strojů.

Ve zdrojovém kódu se míchá dokumentace a samotný kód (např. v Pascalu či C), přičemž Knuth si naprogramoval utility na vyextrahování čistého zdrojového kódu pro účely kompilace a programátorské dokumentace v TEXu.

O tom, že Knuth není žádný suchar a rád si hraje s čísly, vypovídá několik věcí. Už jako středoškolský student odeslal do soutěže vědeckých talentů svůj první matematický článek o číselných soustavách se záporným či dokonce komplexním základem. Vymyslel soustavu o základu $2i$, jejíž speciální vlastností je, že každé komplexní číslo může být reprezentováno pouze číslicemi 0, 1, 2 a 3 a bez znaménka.

Zvláštní je také systém číslování verzí TEXu a METAFONTu. Jak se TEX stává dokonalejším, jeho verze se stále více blíží číslu π. Prohlásil, že po jeho smrti se číslo verze programu TEX s definitivní platností ustálí na π a všechny zbývající bugy se stanou vlastnostmi programu. Podobně se verze METAFONTu blíží základu přirozeného logaritmu, číslu e, a po jeho smrti bude provedena podobná změna jako u TEXu.

23-4-5 Palindromnásobky — 11 bodů

Když už jsme u těch čísel, naprogramujeme si jednu zajímavou úlohu z této oblasti. Víte, co je zajímavé na roce 1991? Když ho vydělíte 11, získáte 181, přičemž všechna tato tři čísla jsou palindromy. (Palindrom je takový řetězec, který se stejně čte zleva i zprava.)

Vaším úkolem bude napsat program, který na vstupu dostane čísla K a D a na výstup vypíše počet násobků čísla K, jež mají délku D a jsou palindromy (všechno v desítkovém zápise). $1 < K < 1\,000$ a $D < 20$.

Zdá-li se vám, že takových čísel musí být hrozně málo, tak vězte, že pro každé K nedělitelné 10 existuje nekonečně mnoho jeho palindromických násobků. Číslo dělitelné 10 takový násobek nemá, protože se nuly na začátku nepíší.

Formát vstupu a výstupu: v souboru `vstup.in` jsou na prvním řádku dvě přirozená čísla K a D oddělená mezerou. Do souboru `pocet.out` vypište na první řádku počet násobků K délky D, které jsou palindromy.

Příklady:

vstup.in	pocet.out
3 1	3
25 3	2
12 4	7
60 4	0
81 6	0

Veškerý svůj čas však Don Knuth nevěnuje jen informatice. Se svou ženou Jill mají doma například vlastní varhany s 812 píšťalami.

Další zajímavou zálibou je focení kosočtverečných dopravních značek, které se vyskytují např. v USA, Kanadě a Austrálii. Na svých stránkách má slušnou sbírku 1069 fotek různých dopravních značek,[6] *včetně kuriozit jako značky s nápisem „Anti-icing spray system.“*

23-4-6 Knuthovy cesty po státech — 9 bodů

Knuth si během jedné cesty po státech, při níž fotil značky, při průjezdu křižovatkou vždy zapsal její číslo. On si je totiž předem očísloval. Zajímalo by ho, jakou největší souvislou část cesty (podle počtu křižovatek) neprošel žádnou křižovatkou více než jednou.

Například pro posloupnost křižovatek

$$3, 4, 1, 2, 4, 8, 7, 2, 3, 8, 2, 9, 1, 4$$

je správným řešením posloupnost

$$3, 8, 2, 9, 1, 4.$$

Jako třešničku na informatickém dortu si uvedeme citát z jednoho jeho dopisu: „Beware of bugs in the above code; I have only proved it correct, not tried it.“ („Pozor na chyby ve výše uvedeném kódu; pouze jsem dokázal jeho správnost, ale nezkoušel jsem ho.“)

I přes stáří 73 let používá Knuth moderní prostředky. Své internetové stránky[7] *často aktualizuje a pracuje na dvou počítačích: má Mac na vytváření grafiky a přístup k internetu a notebook s Linuxem na práci. Je tedy možné, že používá i některé z utilit zmíněných v dalším díle seriálu (ačkoliv píše v editoru Emacs a ne ve Vimu).*

[6] `http://www-cs-faculty.stanford.edu/~uno/diamondsigns/diam.html`

[7] `http://www-cs-faculty.stanford.edu/~uno/`

Pátá série

John von Neumann se narodil v Maďarsku (vlastně v Rakousku-Uhersku) v roce 1903 a zemřel v roce 1957 v USA. Byl to velmi univerzální vědec – pracoval na matematice čisté i aplikované a významnou měrou přispěl k rozvoji počítačů.

S jeho jménem se jde občas potkat dokonce i ve středoškolské výuce informatiky, kde je často jmenována „von Neumannova architektura", jejíž hlavní rys tkví v jediné paměti pro program i data.

Něco takového je skutečně dobrý nápad, který stojí za dnešní univerzálností počítačů, ale tehdy to byl velmi pokrokový koncept, neboť první výpočetní stroje se programovaly přepojováním drátů.

Vymyslel také jednoduchý a ne zcela nepoužitelný způsob generování pseudonáhodných čísel, který funguje tak, že předchozí vygenerované náhodné číslo umocníme na druhou a vezmeme z jeho desítkového zápisu prostřední část, kterou ohlásíme jako nové „náhodné" číslo.

Von Neumann si byl vědom omezení podobných (pseudonáhodných) metod pro generování čísel a známý je jeho výrok „každý, kdo chce generovat náhodné číslice aritmetickými prostředky, je hříšník." On sám jich však potřeboval neobvykle hodně pro náhodné simulace vodíkové bomby, a tak vzal zavděk takovýmto hříšným způsobem.

Společně se Stanisławem Ulamem je uváděn jako průkopník buněčných automatů, zjednodušených modelů vývoje fyzikálních systémů. Podařilo se mu v jednom takovém modelu vytvořit sebereplikující stroj a napsal na toto téma celou knihu. Navrhoval podobné sebesestavující stroje použít pro rozsáhlé těžební operace, kde by obvyklá tovární výroba potřebných strojů stála lidstvo přílišné úsilí.

Podobné myšlenky posledních několik desítek let budí hrůzu technologických nadšenců, kteří předvídají, že nanotechnologie umožní stavbu tak dobrých sebereplikujících jednotek, že na sebe přemění celou planetu. Ostatně o tom byl jeden z posledních proužků na xkcd.[8]

23-5-1 Boj s nanoboty — 11 bodů

Přeneseme se teď do fiktivního světa knihy Diamantový věk, ve které nejrůznější nanoboti vesele poletují po světě a zkázu světa to nevyvolá.

Existují totiž certifikační organizace, které pomocí svých bojových nanobotů vynucují, aby měl každý stroj, který se chce pohybovat v ovzduší, jisté nanorazítko, které zaručuje jeho bezpečnost.

Pokud se zloduch rozhodne, že zaplaví svět necertifikovanými nanoboty, nastane „tonerová válka" – nanoboti se do sebe pustí, lidé z jejich zbytků dostanou rakovinu plic a vymřou.

Mějme kousek světa zadaný jako čtvercovou síť. Na prvním řádku dostaneme M; na každém z následujících M řádků dostaneme souřadnice města x_i a y_i, počet obyvatel C_i a čas příchodu padoucha T_i.

[8] `http://xkcd.com/865/`

Náš hrdina chce předejít tonerové válce a z ní vyplývajícím zdravotním rizikům pro obyvatele daného města. Proto cestuje po mapě a snaží se v tom padouchovi zabránit. Povede se mu to vždy tehdy, když je ve správný čas ve správném městě.

Hrdina začíná na políčku $(0, 0)$ v čase 0 a za jednotku času se může přemístit na sousední políčko, nebo zůstat stát.

Vaším úkolem je najít posloupnost měst, které má navštívit, aby zachránil co nejvíce lidí.

Například pro vstup

```
4
1 6 1000 18
2 7 300 16
3 3 100 11
6 5 500 11
```

program odpoví `4 1` (zachrání 1500 lidí).

Podílel se na konstrukci atomové bomby. Na rozdíl od většiny ostatních významných vědců projektu Manhattan se dokonce zapojil do navazujícího programu pro vývoj bomby vodíkové.

Označoval své názory za „militantnější, než je obvyklé“ a prosazoval kupříkladu preventivní jaderný úder na Sovětský svaz předtím, než si obstará vlastní jaderné zbraně.

Postupně se tak zapojovoval do různých armádních poradních sborů. Přišel na to, že je efektivnější nechat detonovat atomovou nálož vysoko nad zemí, než při dopadu.

Zúčastnil se tedy třeba komise pro výběr japonských měst, nad kterými bude bomba použita, aby pomohl spočítat případné japonské ztráty.

23-5-2 Zjednodušení situace — 13 bodů

Když je mapa bitevního pole moc nepřehledná, odstraní se méně významné jednotky, popř. se sdruží pod souhrnnou vlaječku. Mohli bychom ale v takové situaci chtít zazoomovat na část bitevního pole tak, aby se nezměnil zobrazený poměr sil.

Na vstupu dostaneme $2N$ pozic vojáků naší strany a $2M$ pozic vojáků nepřítele; $2N, 2M \in [2, 600]$. Pozice budou dvojice nezáporných celých čísel v intervalu $[0, 100\,000]$.

Úkolem bude najít takovou přímku, která rozdělí bojiště na dvě části, kde v každé z části leží právě N našich vojáků a M nepřátel. Přímku vypište jako trojici desetinných čísel (a, b, c) (ve výstupu oddělených mezerou), což jsou koeficienty v rovnici přímky $ax + by + c = 0$.

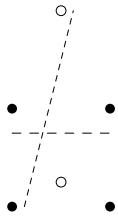
Můžete předpokládat, že se žádné tři body na vstupu nenacházejí na jedné přímce. Na dělicí přímce nesmí ležet žádný ze zadaných bodů. Pokud přímek bude několik, stačí vypsat libovolnou z nich.

Na vstupu jsou na prvním řádku čísla $2N$ a $2M$, následuje $2N + 2M$ řádků, na každém jsou dvě celá čísla oddělená mezerou – souřadnice vrcholů. Nejdříve jsou uvedeny naše pozice, poté pozice nepřátel.

Příklad vstupu:

```
4 2
0 0
0 4
4 0
4 4
2 1
2 8
```

Dvě možná řešení jsou na obrázku vpravo. `4 -1 -2` (krátké čárky) nebo `0 1 -3` (dlouhé čárky).



V oblasti čisté matematiky pracoval na jejích základech – dnešní způsob množinové definice přirozených čísel pochází právě od něj.

Možná znáte algoritmus Minimax, kterým se dá naučit počítač hrát piškvorky, ale i mnohé další hry. Von Neumann stál u zrodu teorie her a formuloval a dokázal tzv. minimaxovou větu.

Mimimaxová věta využívá principu „své tahy volím jako nejlepší, u tahů nepřítele počítám s nejhorším pro mě“ k tomu, aby u her dvou hráčů (s nulovým součtem)[9] *prohlásila, že jeden hráč může nejlépe vyhrát stejnou měrou, jako může druhý hráč nejlépe (tj. nejméně) prohrát.*

23-5-3 Hra pro jednoho hráče — 9 bodů

Hry pro dva hráče známe z běžné zkušenosti – piškvorky jsou jejich nejběžnějším zástupcem. Řekneme-li „hra pro jednoho hráče“, napadne nás nejspíš něco jako solitaire.

Zajímavá věc – existuje i pojem „hra pro žádné hráče“ a myslí se tím třeba již zmíněný buněčný automat (například Game of Life Johna Conwaye), kdy je celý průběh jeho vývoje určen počátečním stavem a jde se na něj jen koukat.

Hra, kterou budeme v této úloze uvažovat, počítá s hráčem jedním. Jmenuje se Hanojské věže a spočívá v tom, že dostanete tři tyče A, B a C a na tyči A máte navlečeny disky se zmenšujícím se průměrem (nahoře je nejmenší).

Vaším úkolem je přemístit při zachování pořadí disky z tyče A na tyč C. Jediný tah, který máte povolen, je přemístění svrchního disku z libovolné tyče na jinou, ale pouze tehdy, pokud je disk menší, než svrchní disk na cílové tyči.

Pro tuto známou hru existuje jedinečná vyhrávající strategie, která má $2^n - 1$ tahů a na kterou není těžké přijít. Pro tři disky vypadá kupříkladu tak,[10] že se ten nejmenší přesune na tyč C, střední na tyč B, nejmenší na tyč B, největší na tyč C... a pak už je to jasné.

[9] `http://cs.wikipedia.org/wiki/Hra_s_nulov%C3%BDm_sou%C4%8Dtem`

[10] `http://commons.wikimedia.org/wiki/File%3ATower_of_Hanoi.gif`

Vaším úkolem v této úloze není tuto strategii zahrát. Dostanete na vstupu počet disků D a číslo N a vypíšete, jak bude vypadat stav hry s D disky po odehrání N kroků této optimální strategie.

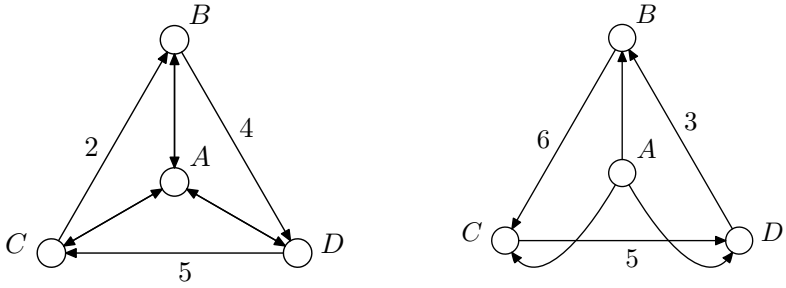
Třeba pro $D = 3$ a $N = 3$ je na tyči A disk největší, na tyči B disk střední a nejmenší a na tyči C není disk žádný.

I o von Neumannovi se vypráví řada veselých příhod. Byl prý špatný, ale vášnivý řidič a často si za volantem četl.

23-5-4 Model čtoucího řidiče — 11 bodů

Mějme řidiče, jenž kvůli kvalitní beletrii nemá dost pozornosti na sledování informačních tabulí, které by mu pověděly, kam která odbočka vede. Jezdí po silniční síti, která je zadaná ohodnoceným orientovaným grafem (tj. každá silnice je jednosměrka), kde budeme každý vrchol chápat jako kruhový objezd a dostaneme s ním na vstupu i cyklické pořadí hran.

Protože řidič nesleduje cedule, vyjede vždy na nejbližším možném následujícím výjezdu. Vaším úkolem je najít pro něj orientovanou cestu s nejnižším možným součtem ohodnocení, která prochází každým vrcholem právě jednou.



Pro případ vlevo řešení neexistuje, pro případ vpravo je řešením A–C–D–B. Hrany bez čísel chápejte jako ohodnocené 1.

Stephen Wolfram o von Neumannovi k stému výročí narození napsal,[11] *že se oddaně držel aktuálních trendů vývoje matematiky a že je překvapivé, že člověk tak inteligentní jako on nikdy nepřišel s žádným skutečně originálním a nečekaným výsledkem, ke kterým měl ve své době opravdu blízko.*

Gödelovy věty či Turingův stroj mohly velmi dobře pocházet i od něj. „Von Neumannova architektura“ se sice stala velmi vlivným konceptem, jistě však nešlo o první myšlenku svého druhu, o několik let ho s ní předběhnul kupříkladu Turing.

Vysvětluje si to jednak tím, že mu současné metody šly aplikovat tak hladce, že k odvážným výletům za hranice obvyklého necítil potřebu, druhak tím, že byl konformní člověk, který ctil autority a stejně jako byl zadobře s americkou vládou a ar-

[11] `http://www.stephenwolfram.com/publications/recent/neumann/`

mádou, tak nechtěl zpochybňovat velká Hilbertova přesvědčení, proti kterým zmíněné originální výsledky šly.

Podle pamětníků to byl velmi společenský a přátelský člověk. Každý týden uspořádal dva večírky a měl rád děti. Rád vyprávěl obhroublé vtipy. Jeho relativně brzká smrt nebyla tak tragická (dá-li se srovnávat) jako v případě zmíněného Gödela nebo Turinga – měl rakovinu a do poslední chvíle pracoval.

23-5-5 Kuchařková — 12 bodů

Následující problém si pojmenujeme Metr a vaším úkolem je dokázat, že je NP-úplný.

Jistě znáte skládací metry. Mají typicky pět článků po dvaceti centimetrech. Mějme metr nepravidelný, jehož jednotlivé články jsou různě dlouhé. Tyto délky dostaneme v pořadí na vstupu, stejně tak délku pouzdra, do kterého bychom chtěli metr uložit.

Podaří se nám to? Pro články délek 6, 3, 3 a pouzdro délky 6 odpověď jistě zní ANO, pro vstup 6, 3, 4 a stejně dlouhé pouzdro to už ale NEpůjde.

23-5-6 Předposlední — 13 bodů

Dostanete na vstupu orientovaný graf s kladně celočíselně ohodnocenými hranami. Dále tam bude pro každý vrchol dvojice kýžených limitů – minimální součet vstupních hran a maximální součet výstupních hran.

Vaším úkolem je najít nové nezáporné celočíselné ohodnocení každé hrany, které nebude větší než to původní a které bude dohromady se všemi ostatními novými ohodnoceními respektovat dané limity.

Seriál: Regulární výrazy

Jan „Moskyto“ Matějka

23-1-7 Regulární výrazy — 14 bodů

V letošním ročníku si budeme povídat o regulárních výrazech. Už se vám jistě někdy stalo, že jste potřebovali nějak zběsile přejmenovat soubory, nahradit v textu všechny výskyty jména Markéta za jméno Dominika nebo jednoduše najít všechna slova začínající velkým písmenem. Dokud jsem nevěděl o regulárních výrazech, dělal jsem veškerou takovou práci ručně, což není od deseti stran nic příjemného, nehledě na to, že lidské oko často nějaký výskyt opomene...

K nalezení všech pádů jména Markéta v jednotném čísle by tedy například sloužil výraz `Markét(a|y|ě|u|o|ou)`.

Regulární výraz umožňuje definovat *množinu řetězců*, které mu vyhovují. Když pak pomocí něj vyhledáváte, najdete všechny řetězce z té množiny.

Jak takový regulární výraz vypadá? Je to řetězec poskládaný z obyčejných a speciálních znaků. Typickým obyčejným znakem je písmeno: výrazu `ab` vyhovuje jen řetezec `ab`. Obyčejný znak je obecně „všechno ostatní“, tedy všechny znaky, o kterých si neřekneme nic zvláštního.

První důležitá skupina speciálních znaků, kterou si uvedeme, jsou znaky, které určují, kolikrát se předchozí znak bude opakovat:

`*`	libovolný počet ($0 \ldots \infty$)
`+`	alespoň jednou ($1 \ldots \infty$)
`?`	jednou nebo vůbec
`{`n`}`	právě n-krát, kdy n je přirozené číslo
`{`a`,`b`}`	a- až b-krát, přičemž musí platit, že $a \leq b < K$, kde K je číslo závislé na systému, obvykle 256, ale může být i víc. Pravá mez (b) může být i vynechaná. Pak je považována pravá mez za nekonečnou, nebo se to chová, jako by tam bylo $K - 1$: to také záleží na systému.

Takže výrazu `ab*` vyhovuje řetězec, který začíná `a` a následuje libovolné množství `b`; výrazu `a+b?c+` vyhovuje řetězec, který začíná alespoň jedním `a`, pak v něm možná je `b` a končí alespoň jedním `c`. Výraz `d{3,5}` je ekvivalentní s výrazem `dddd?d?`, neboť oběma vyhovují právě řetězce obsahující 3, 4, nebo 5 `d`.

Opakovat jen jeden konkrétní znak by však bylo hloupé. Můžeme tedy nějaký kus výrazu uzávorkovat do kulatých závorek a k němu celému se pak tenhle *opakovací operátor* bude vztahovat: výrazu `(aa)*` vyhovuje řetězec obsahující sudý počet `a` (ve stejném smyslu se dá použít `a{2}*`). Výrazu `b?(ab)*a?` vyhovuje řetězec, ve kterém se `a` a `b` pravidelně střídají. Výrazu `b?(a+b)*a*` pak vyhovuje řetězec složený z `a` a `b`, ve kterém se nikde nevyskytuje dvojice `b` vedle sebe.

Do závorek můžeme dát více variant za použití znaku `|` – výrazu `(bagr|kombajn)` vyhovuje jak řetězec `kombajn`, tak řetězec `bagr` (a nic jiného). Pozor, pokud napíšete `(a|b){2}`, budou výrazu vyhovovat řetězce `aa`, `bb`, ale i `ab` a `ba`. Opakovací operátory musíme brát jen jako zkratku za nakopírování toho, co opakují, vedle sebe do výrazu.

Úkol 1 [2b]: Napište jiný výraz, kterému budou vyhovovat přesně stejné řetězce jako výrazu `b?(a+b)*`.

Úkol 2 [2b]: Určete, které všechny řetězce vyhovují výrazu `((ab?)+|(ba?)+)*`, a případně nalezněte kratší ekvivalentní výraz.

Další trik, který si v tomto díle ukážeme, jsou hranaté závorky. Do nich uvedeme množinu znaků, které se na tomto místě mohou objevit. Tedy výrazu `[aZ!]*` vyhovují řetězce jakékoli délky, které jsou složené pouze ze znaků `a`, `Z` a `!`. Do závorek je možno uvést i rozsah: Výrazu `[a-z]` vyhovuje jakékoli malé písmeno. Také je dovnitř možno uvést *třídu znaků*: třeba `[[:digit:]]` jsou všechny číslice – standardní je následující dvanáctice:

`alnum`	písmeno nebo číslice
`alpha`	písmeno
`blank`	prázdný znak (obvykle mezera nebo tabulátor)
`cntrl`	řídící znak (znaky s ASCII kódem menším než 32 a ještě pár dalších)
`digit`	číslice (0–9)
`graph`	tisknutelný znak kromě mezery (písmena, čísla, interpunkce, ...)
`lower`	malé písmeno
`print`	tisknutelný znak včetně mezery
`punct`	tisknutelný znak mimo písmena, čísla a mezeru
`space`	bílý znak (mezera, nový řádek, tabulátor, ...)
`upper`	velké písmeno
`xdigit`	hexadecimální číslice (`[0-9a-fA-F]`)

Třídy znaků je možno kombinovat: Výrazu `[[:digit:][:lower:]XYZ.]` vyhovují všechny číslice, malá písmena, `X`, `Y`, `Z` a tečka. Třídu můžeme také znegovat, uvedeme-li před výčet znaků stříšku `^`: výrazu `[^abc]` vyhovují všechny znaky kromě `a`, `b`, `c`.

Takovou perličkou je pak výraz, kterému vyhovují znaky `]`, `^` a `-`. Pomlčka totiž musí být na začátku nebo na konci, jinak signalizuje výčet; stříška nesmí být na začátku, jinak signalizuje negaci, a `]` musí být na začátku, jinak signalizuje konec závorky: `[]^-]` je správné řešení problému.

Může se hodit ještě jedna zkratka: tečka `.` symbolizuje *jakýkoli znak*. Výraz `.*` pak symbolizuje *libovolný řetězec*.

Úkol 3 [4b]: Napište výraz, kterému budou vyhovovat právě desítkové zápisy čísel dělitelných osmi (takže 0, 8, 256, 344 vyhovují, ale 42 ani 37 ne).

Úkol 4 [2b]: Určete, které řetězce vyhovují výrazu `(00)*[01](11)*`.

Úkol 5 [4b]: Tomuto výrazu měly původně vyhovovat všechny řetězce, které mají sudý počet nul i sudý počet jedniček (a neobsahují jiné znaky). Nalezněte protipříklad a zkuste výraz opravit : `(00|11)*(0(11|00)*1(11|00)*){2}`

Tip: Existují dva typy protipříkladů na regulární výrazy: *false positive* je řetězec, který vyhovuje, i když by neměl; *false negative* je řetězec, který nevyhovuje, ale měl by. Druhý z nich je obvykle závažnější, první z nich se špatně hledá, zvláště v rozsáhlejších výrazech.

Pokud chcete použít jakýkoli speciální znak v jeho obyčejném významu, předřaďte před něj `\` – výrazu `\.\*\\` vyhovuje tečka následovaná hvězdičkou a pak zpětným lomítkem.

Ještě by se hodila poznámka, že tohle všechno zde vyložené jsou *POSIX extended regular expressions*, neboli rozšířené regulární výrazy. Existuje totiž mnoho dalších dialektů regulárních výrazů, něco víc si o nich povíme příště.

Mnoho programů vyhledává v textu po řádkách, typicky `grep` vypisuje všechny řádky, na nichž našel vyhovující řetězec. Existují tedy speciální znaky pro *přišpendlení* řetězce na začátek (`^`) nebo konec (`$`) řádku, takže výrazu `^...$` vyhovují právě tříznakové řádky a výrazu `^a` vyhovují všechna `a` na začátku řádku. Tyto znaky mohou být použity jen na úplném začátku, resp. konci výrazu, kdekoli jinde jsou použity chybně. Takže výraz `(a|^b$|c)` je chybný…

Nakonec, kde se vlastně s regexy (což je zkratka z *regular expressions*) setkáme? S jejich pomocí vyhledávají slavné editory Vim i Emacs, na Windowsech třeba PSPad. Denním chlebem jsou pro programátora v Perlu a taktéž program `grep` a další využívají regexy ke svojí běžné činnosti. Konkrétně pak `egrep` používá přesně ten formát regexů, kterým jsme se tu zabývali.

Regexy můžete používat i v drtivé většině programovacích jazyků od C přes C# až k Perlu a Pythonu – někde jsou přímo součástí jazyka, jindy k dispozici jako knihovna…

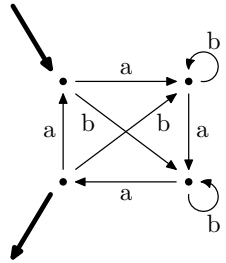
23-2-7 Regulomaty — 12 bodů

Po představení syntaxe regulárních výrazů se podíváme na zoubek tomu, co se s nimi děje uvnitř počítače.

Proč se vlastně oněm výrazům říká regulární? Popisují totiž regulární jazyky, tedy jazyky rozpoznávané konečnými stavovými automaty. Že nevíte, o čem píšu?

Konečný automat je množina (Q, A, δ, q_0, F) … formální definici nechme na seriál 17. série a úlohu 17-2-5.

Konečný automat si představme jako množinu stavů, mezi kterými můžeme různě přecházet tím, že přečteme znak ze vstupu. Ilustrativní obrázek napoví víc než suchá teorie. Tlusté šipky symbolizují vstupní stav a výstupní stavy.



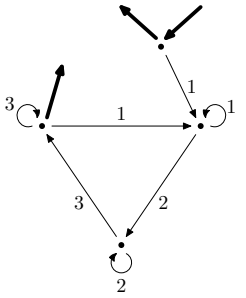
Výstupem našeho automatu pak bude přijetí, nebo odmítnutí, podle toho, jestli řetězec vyhovuje, či nikoli.

Na začátku jsme v počátečním stavu. Přečteme písmenko ze vstupu a vybereme si příslušnou hranu a po ní přejdeme do dalšího stavu. A zase přečteme písmenko ze vstupu, vybereme si hranu, přejdeme…

Když nemáme co přečíst, stačí nám jednoduše zjistit, jestli jsme zrovna ve výstupním stavu, nebo nikoli. Pokud jsme ve výstupním stavu, tak jsme přijali vstup, jinak ho odmítneme. Vstup taktéž odmítneme, pokud při běhu zjistíme, že z aktuálního stavu žádná vhodná hrana nevede.

Automat na obrázku tedy přijímá taková slova jako `bba`, `bababa`, `aaaaaaa`, ale odmítne například slova `abba`, `baba` (skončí vpravo dole), `ababab` (skončí vpravo nahoře), `aaaa` nebo λ[13] (skončí vlevo nahoře).

Existuje hezká věta, která říká, že každý konečný automat lze převést na regulární výraz. To znamená, že umíme najít regulární výraz, který matchuje[14] právě ty vstupy, které přijme konečný automat (a žádné jiné). Důkaz té věty se dá udělat třeba ukázáním univerzálního postupu, který ale bude až v autorském řešení, jinak byste přišli o tu zábavu vymýšlet, jak na to.



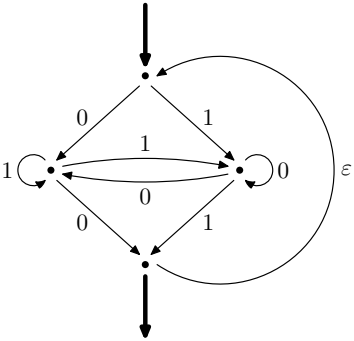
Úkol 1 [4b]: Převeďte automat na obrázku na regulární výraz:

Nebylo to tak hrozné, ne? Co si to vzít obráceně? Existuje docela hezká věta, která dokazuje, že každý regulární výraz lze převést na konečný automat.

Některé jdou i ručně.

Úkol 2 [3b]: Převeďte zadaný výraz na konečný automat:
`(1(23|32)|2(31|13)|3(12|21))*`

Čím méně hran a stavů, tím více bodů máte šanci získat (za automat mající více než 10 stavů nebudou ani 2 body).



Drtivou většinu výrazů ale hned tak jednoduše převést nepůjde, nebo to alespoň není na první pohled vidět. Zavedeme proto nedeterministické konečné automaty (NKA).

NKA je automat, u kterého může z jednoho stavu vést více hran pro jedno písmenko. Program si tedy může vybrat, kterou cestou půjde. Vstup je pak přijat, pokud existuje možnost, jak ho přijmout, neboli pokud existuje vhodná cesta (tedy po které program mohl jít), která končí v nějakém z výstupních stavů.

Navíc si dovolíme takzvané ε-přechody. To jsou hrany, po kterých je možno přejít, aniž přečteme znak ze vstupu. Aby bylo poznat, že jsme k hraně nezapomněli připsat její znak, píšeme k ní ε – symbol pro prázdný řetězec.

Tento automat přijímá třeba řetězce `10111`, `1111`, ale třeba ne `000` nebo `111`. Vyzkoušejte si sami, jak.

Úkol 3 [5b]: Převeďte výraz `(10(1(10)*1)*01)*` na NKA s ε-přechody. Bonus 2b pro ty, kdo vymyslí jednodušší (tedy kratší) ekvivalentní výraz.

[13] λ je obvykle používaná zkratka pro prázdné slovo.

[14] regex *něco* matchuje = regexu vyhovuje *něco*

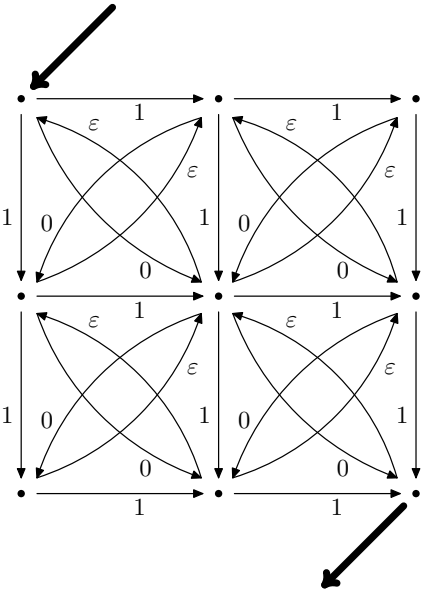
23-3-7 Automaty stokrát jinak — 12 bodů

Minule jsme si popsali nedeterministický konečný automat (NKA) s ε-přechody. Definovali jsme, že vstup přijme, pokud v něm existuje alespoň jedna možnost, jak při čtení onoho vstupu skončit ve výstupním stavu.

Běžný program ale neumožňuje paralelně zkoumat všechny větve, kterými by mohl procházet. To bychom si museli pořídit třeba paralelizátor jako v jednom ze starých ročníků olympiády.

Mohli bychom však zkusit převést NKA na DKA, který programem simulovat velmi jednoduše umíme. Dá se dokázat, že převod je možné provést vždycky (i když výsledný automat může mít až exponenciální velikost), ale obvykle se to dělá ukázáním obecného postupu a ten si ukážeme až v řešení.

Úkol 1 [6b]: Vymyslete, jak simulovat NKA a jak převést NKA na DKA. Pokud vám to nepůjde ve vší obecnosti, máte šanci získat 2 body za převod tohoto konkrétního automatu:



Samozřejmě, když vyřešíte úlohu obecně, tak ji nemusíte řešit konkrétně na tomto automatu, nicméně snad nic nebrání jeho použití třeba k názornému ilustrování vašeho postupu. Za pomoci mašinérie, kterou jsme si zatím ukázali, by pro vás neměl být problém zjistit následující (ale můžete to dělat i jinak, jestli chcete):

Úkol 2 [5b]: Zjistěte, jestli tyto dva regexy popisují stejný jazyk (tedy vyhovují jim právě stejné řetězce):

```
(AB)*(AA(BA)*(A|BB)(AB)*)*
(A(AB)*(B|AA))*
```

Úkol 3 [1b]: Nalezněte nejmenší násobek devíti, jehož desítkový zápis vyhovuje tomuto výrazu:

```
(102)*101(201)*((0|202)(102)*101(201)*)*
```

Nezapomeňte, že součástí každého úkolu je přesvědčit opravujícího, že vaše řešení je správně. Regex bez jakéhokoli vysvětlení není úplné řešení a nemá nárok na plný počet bodů. Stejně tak konstatování „NE“, „10202010102“ nebo „nelze“...

Tím jsme uzavřeli kapitolu konečných automatů. Příště se vrátíme zpátky k regexům, ukážeme si, jak se programem `sed` nahrazují řetězce za jiné, co dělat, když regex pro požadovaný řetězec prostě neexistuje a jak s tím vším souvisí Chuck Norris.

23-4-7 Bratrstvo Seda a Grepa — 16 bodů

Ukážeme si, jak regulární výrazy používáme v praxi. Programy, které nás budou zajímat, jsou UNIXové nástroje `sed`, `grep` a případně světoznámý editor Vim. Jsou snad v každé distribuci Linuxu, na Windows je možné použít balík Cygwin.[15]

Na vyhledávání v textu se používá program `grep`. Není to žádná aplikace s klikacím frontendem, používá se trapně v terminálu. O to je však rychlejší. Základní použití je `egrep <regex>`, kdy program čte standardní vstup (to, co mu píšete na klávesnici) a vypisuje na standardní výstup (terminál). Vypisuje ty řádky, na kterých našel podřetězec vyhovující zadanému regexu.

Možná si říkáte – proč `egrep`? Pohled do manuálu (`man grep`) napoví, že existovaly starší regexy, které toho uměly méně a mají trochu jinou syntaxi. Kvůli zpětné kompatibilitě starých skriptů byla tedy přidána tato varianta programu `grep`. Ze stejného důvodu budeme později u programu `sed` používat přepínač `-r`.

Když si tedy pustíme `egrep 'ba*gr+'`, tak na vstup

```
grbagr
baagrr
rgba
bbarg
rbgrbgg
```

dostaneme výstup

```
grbagr
baagrr
rbgrbgg
```

Můžeme chtít, aby `grep` četl vstup ze souboru. Za zadaný regex můžeme napsat libovolně mnoho jmen souborů, které má číst (oddělené mezerami).

Pokud čte jeden, vypíše prostě vhodné řádky. Když jich je víc, vypíše na začátku každého řádku ještě jméno souboru, ve kterém jej našel. Toto chování se dá regulovat volbami `-h` (bez jmen) a `-H` (vypiš jméno, i když je soubor sám), které můžete napsat před výraz.

Takže například `egrep -h Kája jmena prijmeni archiv` vyhledá všechny řádky ze souborů `jmena`, `prijmeni` a `archiv`, které obsahují výraz `Kája`. Volba `-h` potlačila výpis jmen souborů.

Ještě existuje zajímavá volba `-v`, která logicky obrátí výpis (vypisuje jen ty řádky, které hledaný výraz neobsahují). Například když hledáte, jak často k vám na web chodí lidé z jiných prohlížečů než IE a FF, tak ze záznamů prostě vyřadíte řádky odpovídající IE a FF...

[15] `http://www.cygwin.com/`

Programy za sebe můžete řetězit znakem `|` – vezme standardní výstup prvního programu a nevypisuje ho na terminál, ale předhodí ho druhému programu na standardním vstupu. Například `egrep ba+gr|egrep -v baaagr`. Někdy je jednodušší programy zřetězit než psát jeden dlouhý výraz, který pojme všechno.

Tento znak se také označuje jako „roura“ (*pipe*), protože to dávným programátorům připadalo, jako by mezi programy prostě natahovali potrubí.

Složitější výrazy se hodí psát mezi apostrofy: `'(\(|\))'`Jinak je může interpretovat ještě terminál samotný a z `a*` udělat seznam souborů, které začínají `a`, což rozhodně nechcete, a to je to nejmenší, můžou se stát daleko horší věci...

Dosud jsme regulárními výrazy pouze vyhledávali. Jde však o daleko mocnější zbraň, výrazně větší kladivo. Jak bylo zmíněno už v prvním dílu, regexy často používáme k systematickému nahrazování v textu.

Na nahrazování se hodí program `sed`. Jeho základní použití je podobné jako u programu `grep`.

Jak ale vypadá nahrazovací výraz? Začíná písmenkem `s`, pak následuje oddělovač (typicky `/` nebo `#`, ale může to být libovolný znak, klidně písmenko), pak hledaný řetězec, potom znovu stejný oddělovač, potom řetězec k nahrazení, a nakonec zase oddělovač. Například nahrazovací výraz `s/ahoj/nazdar/` nahrazuje slovo `ahoj` za `nazdar`.

Oddělovač se potom stává speciálním znakem a pokud jej chcete použít v původním významu, je potřeba to říct – obackslashovat jej. (`\/`, `\#`)

Jak to funguje jako příkaz? Když pustíte příkaz

```
sed -r 's/ahoj/nazdar/',
```

čte standardní vstup po řádkách, na každém řádku hledá výraz `ahoj`, jeho první nalezený výskyt změní na `nazdar` a změněný řádek vypíše na výstup.

Takže pokud vstup

```
moskyto@atrey.karlin.mff.cuni.cz
bagr
ksp@mff.cuni.cz bait@uu.ucw.cz
kombajn
```

proženeme třeba příkazem `sed -r 's/@/ (at) /'`, dostaneme výstup

```
moskyto (at) atrey.karlin.mff.cuni.cz
bagr
ksp (at) mff.cuni.cz bait@uu.ucw.cz
kombajn
```

Všimněte si na třetím řádku zbylého zavináče. Kdybyste potřebovali nahrazovat všechny výskyty, ne jen ten první, tak musíte za výraz ještě připsat `g` (od „globally“):

```
sed -r 's/@/ (at) /g'
```

Pokud potřebujete „přišpendlit“ celý výraz na začátek nebo na konec řetězce, uveďte stříšku `^` na jeho úplném začátku nebo dolar `$` na jeho konci. To funguje jak pro `sed`, tak pro `grep`.

Úkol 1 [1b]: Napište nahrazovací výraz, který smaže všechny „trailing spaces“ – bílé znaky na koncích řádků (stačí asi mezera, `\t` a `\r`).

Jenom takhle by to ale bylo trapné. Co když potřebujeme z řádku jenom něco vytáhnout? Pak opravdu můžeme „najít“ třeba celý řádek jedním výrazem a vybrat si z něj jenom tu část, kterou potřebujeme.

`sed -r 's/^.*"(.*)".*$/\1/'` z celého řádku nechá jen řetězec v uvozovkách.

Takže vstup vlevo se přepíše na výstup vpravo.

```
"                                   "
bagr                                bagr
b"ag"r                              ag
b"a"g"r                             g
```

Když tedy kus řetězce uzávorkujete, můžete se na tyto závorky pak odkazovat pomocí znaku `\` následovaného číslicí (`1` – `9`). Závorky jsou číslovány zleva doprava podle své otevírací závorky. Takže třeba `sed -r 's/(.)(.)/\2\1/'` na každém řádku prohodí první dva znaky.

Pamatujte si, že pokud `sed` na řádku nenajde žádný výskyt hledaného výrazu, vypíše řádek beze změny.

Také je potřeba si uvědomit „žravost“ některých operací. Znaky `*` i `+` jsou „nenažrané“. To znamená, že pokud by si `sed` měl vybrat mezi více podřetězci, které by přiřadil do oné hvězdičky, tak vybere ten nejdelší z nich. Přednost ve žravosti má dřívější `*`/`+`. Toho si můžete všimnout na čtvrtém řádku, kde byly troje uvozovky, že ty první byly polknuty do `.*` na začátku.

Podobně je žravý i výběr z více možností – bere první variantu, která se na dané místo hodí, takže `(.|..)` vždy uloží do `\1` jeden znak (a druhá půlka závorky je tedy zbytečná), kdežto `(..|.)` uloží do `\1` dva znaky, pokud to jenom trochu jde.

Pravidlo číslování u otevírací závorky se hodí u vnořených závorek:

```
sed -r 's/^(ab*cd(ef|gh)i+j)$/\1: \2/'
```

připíše za každý řádek, pokud je v onom poměrně složitém tvaru, dvojtečku a `ef` nebo `gh`...

Úkol 2 [4b]: V češtině je typografickou chybou napsat na konec řádku jednopísmennou předložku nebo spojku, výjimkou je spojka a, pokud se před ní nevyskytuje nějaké interpunkční znaménko (tečka, čárka, závorka, ...).

Různé programy to řeší různě, v TEXu se za předložku místo mezery vkládá vlnka (`~`). Napište výraz pro `sed`, který zařídí korektní „ovlnkování textu“ (případ, kdy je předložka/spojka přímo na začátku nebo na konci řádku, řešit nemusíte).

Tyhle „odkazy zpátky“ (*backreferences*) můžeme ale používat i ve vyhledávaném výrazu. `s/^(.*)\1$/\1/` tedy najde všechny řádky, na kterých je řetězec dvakrát za sebou, a nahradí tyto výskyty jen jedním opakováním řetězce. Na vstup

```
aa
ab
abeceda
bagrbagr
```

odpoví

```
a
ab
abeceda
bagr
```

Mimochodem, například také slova `sentence`, `sequence` nebo `statement` jsou nahrazovacími výrazy…

Úkol 3 [4b]: Napište vyhledávací výraz, který ze slovníku „vygrepuje“ všechna slova, která jsou validními nahrazovacími výrazy pro `sed`. I `grep` umí backreference (stejně jako `sed`). Bonusové 4 body dostane ten, kdo vyrobí takový výraz bez backreferencí.

Výraz musí být samozřejmě nezávislý na použité abecedě, takže není správným řešením vygenerovat pro každý možný oddělovač separátní výraz…

Ve slovníku je na každém řádku právě jedno slovo (a vlevo a vpravo od něj nejsou žádné mezery).

Místo jednoho výrazu můžete použít až tři volání `grep` spojená za sebou rourou.

Nyní si zavedeme fiktivní programovací jazyk, budeme mu říkat třeba „ReProg“. Programem v ReProgu bude posloupnost nahrazovacích výrazů pro `sed`.

Nad vstupními daty se postupně spustí každý z výrazů. Když program doběhne na konec, zjistí ReProg, jestli nějaký z výrazů měnil data. Pokud ano, spustí se celý program od začátku znovu; pokud ne, data se prohlásí za výstup a program skončí.

Takový vzorový program je třeba

```
statement
sentence
samaria
```

Když mu předhodíme sebe sama jako vstup, pak po prvním průběhu budeme mít

```
steriencement
sencence
serienmaria
```

Po druhém průchodu máme

```
steriencerienc
sencence
serienriemenria
```

a tak dále, až pátý průchod data nezmění a výstupem je

```
steriencerienc
sencence
serienrierienrierien
```

Úkol 4 [7b]: Napište program pro ReProg, který na vstupu na každém řádku dostane dvě přirozená čísla ve dvojkovém zápisu oddělená mezerou a na výstupu bude na každém řádku to větší z nich. Například pro jednořádkový vstup `1011 110` bude výstup `1011`.

Za zmínku ještě stojí, že i `sed` podobné iterování umí, byť na jiném principu a s trochu jinou syntaxí. To už je ale trochu jiný příběh, třeba na nějakou soustředkovou přednášku.

A kdybyste se báli, že se na soustředko nedostanete, přečtěte si třeba manuálovou stránku (`man sed`), nebo dosti obsáhlou dokumentaci na internetu.[16]

To je pro tentokrát vše, v závěrečném dílu si ukážeme temnou a mocnou sílu jedné zajímavé mutace regexů – těch v Perlu. Prý se v nich dá napsat výraz, který nahradí dvojici čísel na vstupu za jejich podíl...

23-5-7 Perlím, Perlíš, Perlíme — 15 bodů

V závěrečném díle seriálu si ukážeme, kam došlo všemožné rozšiřování regexů v Perlu – jazyce určeném zvláště na zpracování textu. Nebudeme probírat kompletní PerlRe, kdybyste se o nich chtěli dozvědět více, přečtěte si přehlednou dokumentaci.[17]

První kroky v Perlu

Nejprve je potřeba Perl nějak získat. Pokud máte Linux, máte jej už pravděpodobně dávno nainstalovaný, jen o něm nevíte. Kdybyste jej náhodou neměli, nainstalujte si jej z balíčkovače, mají jej snad všechny rozumné distribuce.

Na Windows si nainstalujte Cygwin,[18] někteří jej asi máte už od minulé série. V něm by měl Perl jít nainstalovat.

Do Cygwinu se balíčky doinstalovávají spuštěním setupu, člověk si vybere nějaký server a dostane se na seznam balíčků. Tam dá vyhledat, co potřebuje, zaškrtne, odklikne a při příštím spuštění Cygwinu může nové balíčky používat.

Perl je programovací jazyk mnoha zajímavých vlastností, zájemci o více informací si přečtou rozsáhlou dokumentaci,[19] případně se zeptají na fóru.

[16] `http://www.gnu.org/software/sed/manual/sed.html`
[17] `http://perldoc.perl.org/perlre.html`
[18] `http://www.cygwin.com/`
[19] `http://perldoc.perl.org/perlintro.html`

My si ukážeme jenom minimální program, který můžete používat k testování svých pokusů s PerlRe.

```
#!/usr/bin/perl
use strict; use warnings; # hlídací pes

# cyklus přes všechny řádky vstupu
while (<>) {
    # sem pište své příkazy
    # nahraď všechny bagry za kombajny
    s/bagr/kombajn/g;

    # sem už své příkazy nepište
    # výpis
    print;
}
```

Tenhle program čte vstup po řádkách, pro každý řádek provede zadané příkazy a pak jej vypíše. Každý příkaz končí středníkem; komentáře jsou uvozeny znakem `#` (kriminál), od něj dál se všechno ignoruje.

Jak spustit program? V terminálu dojděte do příslušné složky (pomocí `cd`) a napište `perl minimal.pl`, pokud jste pojmenovali svůj minimální program `minimal.pl` (obvyklá přípona programu v Perlu je `.pl`).

Pak na vstup píšete podobně jako u `sed`u; ukončit vstup je možné stiskem Ctrl+D. Uvedený program by tedy převedl vstup (vlevo) na výstup (vpravo):

```
Žlutý bagr                    Žlutý kombajn
Modrý kombajn                 Modrý kombajn
Mám dva bagry.                Mám dva kombajny.
bagrbagrbagr                  kombajnkombajnkombajn
```

Nuže, po technickém úvodu přijde konečně něco o regexech (které také budeme v tomto textu označovat jako PerlRe).

Základní regexy

Existují dva typy regexových příkazů. Prvním z nich je regex ve stylu `grep`, který zjišťuje, jestli je na vstupním řádku vyhovující podřetězec. Vypadá třeba takhle: `m#cosi#`.

První písmenko je vždycky `m`, druhé písmenko je oddělovač, pak následuje hledaný regex (ve kterém je případně oddělovač nutno escapovat, viz níže) a pak zase oddělovač. Používá se třeba v podmínkách:

```
print "obsahuje bagr\n" if m/bagr/;
```

Další z mnoha využití je při parsování vstupu:

```
# $1 = hodiny, $2 = minuty, $3 = vteřiny
m/(..):(..):(..)/; print "Je $3. vteřina";
```

Takovéhle programy už ale psát nebudeme, zůstaneme jen u samotných regexů.

Druhý typ je nahrazovací – `s#bagr#kombajn#g`. Písmenko `s`, oddělovač, regex, oddělovač, čím nahradit, oddělovač, modifikátory (ty se ostatně mohou objevit i u prvního typu). Příkaz nalezne první výskyt regexu a nahradí jej zadaným řetězcem. Je-li uveden modifikátor `g`, nalezne příkaz všechny nepřekrývající se výskyty a pak je najednou nahradí.

Příkaz `s/rr/tr/g;` tedy převede vstup (vlevo) na výstup (vpravo):

`rr`	`tr`
`rrr`	`trr`
`rrrr`	`trtr`
`rrrrr`	`trtrr`

Jak vypadá oddělovač? Může to být nějaký ze znaků

```
!"#$&%)*+,-./:;=>?@\]^‘|}~,
```

případně je možné použít konstrukci `s(a)(b)g` (a analogicky s `{}`, `<>` a `[]`).

Jak vypadá výraz? Regexy, které jsme si definovali v prvním díle, by měl Perl přijmout bez řečí. Backreference fungují také stejně, jen jich může být libovolně mnoho. Navíc pokud není reference použita přímo ve výrazu, neodkazuje se na ni `\4`, ale `$4` (nebo třeba `$2078`). Takže například takhle: `s/(.*)(.*)\2\1/$2$1$1$2/`

Rozhlížecí předpoklady

Anglický název je „Look-around assertions“, kdybyste si o tom chtěli přečíst v manuálu.

Taková typická konstrukce je `s/bagr(?=b)/rýč/g`. Té vyhovuje `bagr`, pokud za ním je písmeno `b`. To však není zahrnuto do onoho řetězce, takže z řetězce `bagr-bagrbagry` udělá `rýčrýčbagry`.

Jak vypadají tyto konstrukce formálně? `(?=`*`Výraz`*`)` vyhovuje, pokud na tomto místě začíná kus řetězce, který mu vyhovuje. Pak se Perl **vrátí na jeho začátek** a tváří se, jako by tím kusem řetězce ještě nikdy neprošel. Třeba výrazu `^(?=(..)*$)(...)*$` vyhovují všechny řádky, na kterých je počet znaků dělitelný šesti.

Konstrukce `(?!`*`výraz`*`)` dělá téměř totéž, akorát v negaci. Vyhovuje, pokud se Perlu nepodaří najít žádný vyhovující řetězec začínající na tomto místě. Po tomto zjištění se Perl **vrátí na jeho začátek** a pokračuje, jako by se nechumelilo.

Ještě existují podobné konstrukce v obráceném směru. Předešlé dvě byly „koukni dopředu“, následující budou „koukni zpátky“.

Chcete-li tedy Perlu říct „Tady zastav a zjisti, jestli na tomto místě končí řetězec vyhovující výrazu,“ použijete konstrukci `(?<=`*`výraz`*`)`; pokud chcete zajistit, aby na tomto místě **nekončil** žádný vyhovující řetězec, napíšete `(?<!`*`výraz`*`)` – například výrazu `.*(?<!bagr)` vyhovují všechny řetězce, které nekončí „bagr“.

Výrazy typu „koukni dozadu“ mají jedno nepříjemné omezení. Musí mít fixní délku. To znamená, že v nich nejen nesmí být kvantifikátory, ale ani třeba `(a|bb)`, jde o dvě varianty různé délky.

Rekurze

Držte si klobouky, jedeme s kopce. Jak vyrobit výraz, kterému by vyhovoval palindromický řádek? Tenhle to je: `^((.)(?1)\2|.?)$`

Co je na něm tak zajímavého? Ten malinký kousek `(?1)`, který říká něco takového: „Najdi závorku, na kterou by ses normálně odkazoval pomocí `$1`. Ten výraz, který je uvnitř, jako bys spustil na tomto místě…“

Nejlepší bude asi rekurzi předvést na příkladech. Onen první je velice jednoduchý. Celý se skládá ze dvou alternativních větví – v první probíhá rekurzivní krok, druhá funguje jako ukončovací podmínka.

První větev vezme první znak, spustí rekurzi a zbytek pak zase musí být první znak. Rekurze končí ve chvíli, kdy dojde doprostřed testovaného řetězce – zbývá tam prázdný řetězec (v palindromu sudé délky) nebo prostřední znak, který je sám sobě párovým. Přesně o to se stará druhá větev.

Za připomenutí stojí, že závorky se číslují podle pozice jejich **otevírací** závorky zleva doprava.

Ještě si uvedeme jeden rekurzivní příklad – regex, kterému vyhovují všechna správná uzávorkování. Prozkoumejte si jej sami.

```
(\((?1)*\))*
```

Úkol 1 [9b]: Na řádku jsou vždy dvě různá kladná celá čísla zapsaná ve dvojkové soustavě oddělená mezerou. Napište (jeden) substituční výraz, který na řádku zanechá to větší z nich.

Tedy na vstup (vlevo) dostanete výstup (vpravo):

`1001 1101`	`1101`
`11100 11`	`11100`
`100 1111`	`1111`

Úkol 2 [6b]: Na každém řádku vstupu je pseudoXML. Jsou povolené jen párové tagy bez atributů a prostý text libovolně mezi nimi a okolo. Název tagu (řetězec nenulové délky mezi `<` a `>`) smí být složen pouze z `[[:alnum:]]` znaků. Prostý text nesmí obsahovat znaky `<` a `>`.

Váš (jeden) substituční výraz zkontroluje validitu každého řádku na vstupu, tedy jestli má každý tag příslušný uzavírací a jestli se tagy nekříží. Pokud je řádek validní, nechá jej být, v opačném případě jej smaže (nahradí prázdným řetězcem).

Ze vstupu

```
xx<a>ehm<b>sgfd</b>dfsd</a>
xx<a>ehm<b>sgfd</a>dfsd</b>
```

nechá program jen první řádek.

Připomínám, že k řešení patří zdůvodnění. Zvláště u této série si dejte záležet na popisu jednotlivých částí regexů, stejně jako na zdůvodnění správnosti.

Používejte k řešení jen tu část PerlRe, kterou jsme si zde vyložili, ať mají všichni stejné podmínky.

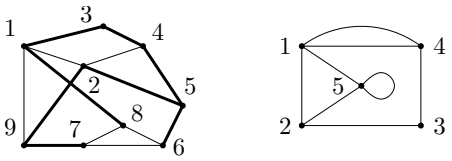
Programátorské kuchařky

Kuchařka první série – grafy

V dnešním vydání známého bestselleru budeme péci grafy souvislé i nesouvislé, orientované i neorientované. Řekneme si o základním procházení grafem, komponentách souvislosti, topologickém uspořádání a dalších grafových algoritmech. Abychom ale mohli začít, musíme si nejprve říci, s čím budeme pracovat.

Ingredience

Neorientovaný graf je určen množinou vrcholů V a množinou hran E, což jsou neuspořádané dvojice vrcholů. Hrana $e = \{x, y\}$ spojuje vrcholy x a y. Většinou požadujeme, aby hrany nespojovaly vrchol se sebou samým (takovým hranám říkáme *smyčky*) a aby mezi dvěma vrcholy nevedla více než jedna hrana (pokud toto neplatí, mluvíme o *multigrafech*). Obvykle také předpokládáme, že vrcholů je konečně mnoho. Neorientovaný graf většinou zobrazujeme jako body pospojované čarami.



Neorientovaný graf a multigraf

Podgrafem grafu G rozumíme graf G', který vznikl z grafu G vynecháním některých (a nebo žádných) hran a vrcholů.

Často nás zajímá, zda se dá z vrcholu x dojít po hranách do vrcholu y. Ovšem slovo „dojít“ by mohlo být trochu zavádějící, proto si zavedeme pár pojmů:

- *sled* budeme říkat posloupnosti vrcholů a hran ve tvaru v_1, e_1, v_2, e_2, …, e_{n-1}, v_n, že $e_i = \{v_i, v_{i+1}\}$ pro každé i. Sled je tedy nějaká procházka po grafu. Délku sledu měříme počtem hran v této posloupnosti.
- *tah* je sled, ve kterém se neopakují hrany, tedy $e_i \neq e_j$ pro $i \neq j$.
- *cesta* je sled, ve kterém se neopakují vrcholy, čili $v_i \neq v_j$ pro $i \neq j$. Všimněte si, že se nemohou opakovat ani hrany.

Lehce nahlédneme, že pokud existuje sled z vrcholu x do y ($v_1 = x$, $v_n = y$), pak také existuje cesta z vrcholu x do vrcholu y. Každý sled, který není cestou, totiž obsahuje nějaký vrchol u dvakrát. Existuje tedy $i < j$ takové, že $u = v_i = v_j$. Pak ale můžeme z našeho sledu vypustit posloupnost $e_i, v_{i+1}, \ldots, e_{j-1}, v_j$ a dostaneme také sled spojující v_1 a v_n, který je určitě kratší než původní sled. Tak můžeme po konečném počtu úprav dospět až ke sledu, který neobsahuje žádný vrchol dvakrát, tedy k cestě.

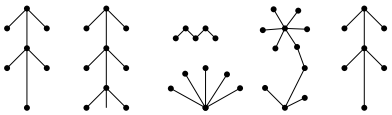
Kružnicí neboli *cyklem* nazýváme cestu délky alespoň 3, ve které oproti definici cesty platí $v_1 = v_n$. Někdy se na cesty, tahy a kružnice v grafu také díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

Ještě si ukážeme, že pokud existuje cesta z vrcholu a do vrcholu b a z vrcholu b do vrcholu c, pak také existuje cesta z vrcholu a do vrcholu c. To vyplývá z faktu, že existuje sled z vrcholu a do vrcholu c, který můžeme dostat například tak, že spojíme za sebe cesty z a do b a z b do c. A jak jsme si ukázali, když existuje sled z a do c, existuje i cesta z a do c.

V mnoha grafech (například v těch na předchozím obrázku) je každý vrchol dosažitelný cestou z každého. Takovým grafům budeme říkat *souvislé.* Pokud je graf nesouvislý, můžeme ho rozložit na části, které již souvislé jsou a mezi kterými nevedou žádné další hrany. Takové podgrafy nazýváme *komponentami souvislosti.*

Teď se podívejme na pár pojmů z přírody: *Strom* je souvislý graf, který neobsahuje kružnici. *List* je vrchol, ze kterého vede pouze jedna hrana. Ukážeme, že každý strom s alespoň dvěma vrcholy má nejméně dva listy. Proč to? Stačí si najít nejdelší cestu (pokud je takových cest více, zvolíme libovolnou z nich). Oba koncové vrcholy této cesty musí být nutně listy: kdyby z některého z nich vedla hrana, musela by vést do vrcholu, který na cestě ještě neleží (jinak by ve stromu byla kružnice), ale o takovou hranu bychom cestu mohli prodloužit, takže by původní cesta nebyla nejdelší.

Grafům bez kružnic budeme obecně říkat *lesy,* jelikož každá komponenta souvislosti takového grafu je strom.



Les, jak ho vidí matematici

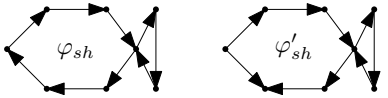
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

Kostra souvislého grafu říkáme každému jeho podgrafu, který je stromem a spojuje všechny vrcholy grafu. Můžeme ji například získat tak, že dokud jsou v grafu kružnice, odebíráme hrany ležící na nějaké kružnici. Pro nesouvislé grafy nazveme kostrou les tvořený kostrami jednotlivých komponent. Na prvním obrázku je jedna z koster levého grafu znázorněna silnými hranami.

Cvičení: Zkuste si dokázat, že stromy jsou právě grafy, které jsou souvislé a mají o jedna méně hran než vrcholů.

Orientované grafy

Často potřebujeme, aby hrany byly pouze jednosměrné. Takovému grafu říkáme *orientovaný graf*. Hrany jsou nyní uspořádané dvojice vrcholů (x, y) a říkáme, že hrana vede z vrcholu x do vrcholu y. Hrany (x, y) a (y, x) jsou tedy dvě různé hrany. Orientovaný graf většinou zobrazujeme jako body spojené šipkami. Většina pojmů, které jsme definovali pro neorientované grafy, dává smysl i pro grafy orientované, jen si musíme dát pozor na směr hran.



Silně a slabě souvislý orientovaný graf

Se souvislostí orientovaných grafů je to trochu složitější. Rozlišujeme slabou a silnou souvislost: *slabě souvislý* je graf tehdy, pokud se z něj zapomenutím orientace hran stane souvislý neorientovaný graf. *Silně souvislým* ho nazveme tehdy, vede-li mezi každými dvěma vrcholy x, y orientovaná cesta v obou směrech. Pokud je graf silně souvislý, je i slabě souvislý, ale jak ukazuje náš obrázek, opačně to platit nemusí.

Komponenta silné souvislosti orientovaného grafu G je takový podgraf G', který je silně souvislý a není podgrafem žádného většího silně souvislého podgrafu grafu G. Komponenty silné souvislosti tedy mohou být mezi sebou propojeny, ale žádné dvě nemohou ležet na společném cyklu.

Ohodnocené grafy

Další možností, jak si graf „vyzdobit“ je ohodnotit jeho hrany čísly. Například v grafu silniční sítě (vrcholy jsou města, hrany představují silnice mezi nimi) je zcela přirozené ohodnotit hrany délkami silnic nebo třeba mýtným vybíraným za průjezd silnicí. Přiřazeným číslům se proto často říká *délky* hran nebo jejich *ceny*. Pojmy, které jsme si před chvílí nadefinovali pro obyčejné grafy, můžeme opět snadno rozšířit pro grafy ohodnocené – např. délku sledu budeme namísto počtu hran sledu počítat jako součet jejich ohodnocení. Neohodnocený graf pak odpovídá grafu, v němž mají všechny hrany jednotkovou délku.

Podobně můžeme přiřazovat ohodnocení i vrcholům, ale raději si všechny operace s ohodnocenými grafy necháme na některé z dalších dílů kuchařky. I tak budeme mít práce dost a dost.

Reprezentace grafů

Nyní už víme o grafech hodně, ale ještě jsme si neřekli, jak graf reprezentovat v paměti počítače. To můžeme udělat například tak, že vrcholy očíslujeme přirozenými čísly od 1 do N, hrany od 1 do M a odkud kam vedou hrany, popíšeme jedním z následujících tří způsobů:

- *matice sousednosti* – to je pole A velikosti $N \times N$. Na pozici $A[i, j]$ uložíme hodnotu 0 nebo 1 podle toho, zda z vrcholu i do vrcholu j vede hrana (1) nebo nevede (0). S maticí sousednosti se zachází velmi snadno, ale má tu nevýhodu, že je vždy kvadraticky velká bez ohledu na to, kolik je hran. Výhodou naopak je, že místo jedniček můžeme ukládat nějaké další informace o hranách, třeba jejich délky. Vpravo od tohoto odstavce najdete matici sousednosti grafu z prvního obrázku.

```
  123456789
1 011000011
2 100110001
3 100100000
4 011010000
5 010101000
6 000010110
7 000001011
8 100001100
9 110000100
```

- *seznam sousedů* je obvykle tvořen dvěma poli, konkrétně polem sousedů $S[1 \ldots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \ldots N]$, v němž se pro každý vrchol dozvíme začátek odpovídajícího úseku v poli S. Pokud navíc do $Z[N+1]$ uložíme $M+1$, bude platit, že sousedé vrcholu i jsou uloženi v $S[Z[i]], \ldots, S[Z[i+1]-1]$. Tato reprezentace má tu výhodu, že zabírá pouze prostor $\mathcal{O}(N+M)$ a sousedy každého vrcholu máme pěkně pohromadě a nemusíme je hledat. Pro graf z 1. obrázku:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2	4	6	5	7	8	6	8	9	1	6	7	1	2	7

i	1	2	3	4	5	6	7	8	9	10
$Z[i]$	1	5	9	11	14	17	20	23	26	29

Reprezentace grafu seznamem sousedů

- *půlhranami* – tato reprezentace se používá tehdy, pokud potřebujeme během výpočtu graf složitě upravovat. Je univerzální, ale dost pracná na naprogramování. Spočívá v tom, že si každou hranu uložíme jako dvě půlhrany (začátek a konec hrany), každý vrchol bude obsahovat spojové seznamy přicházejících a odcházejících půlhran a každá půlhrana bude ukazovat na svou druhou polovici a na vrchol, ze kterého vychází.

V následujících receptech budeme vždy používat seznamy sousedů, poli S budeme říkat `Sousedi`, poli Z `Zacatky` a nadeklarujeme si je takto:

```
var N, M: Integer;  { počet vrcholů a hran }
    Zacatky: array[1..MaxN+1] of Integer;
    Sousedi: array[1..MaxM] of Integer;
```

Prohledávání do hloubky

Naše povídání o grafových algoritmech začneme dvěma základními způsoby procházení grafem. K tomu budeme potřebovat dvě podobné jednoduché datové struktury: *Fronta* je konečná posloupnost prvků, která má označený začátek a konec. Když do ní přidáváme nový prvek, přidáme ho na konec posloupnosti. Když z ní prvek odebíráme, odebereme ten na začátku. Proto se tato struktura anglicky nazývá *first in, first out*, zkráceně *FIFO*. *Zásobník* je také konečná posloupnost prvků se začátkem a koncem, ale zatímco prvky přidáváme také na konec, odebíráme je z téhož konce. Anglický název je (překvapivě) *last in, first out*, čili *LIFO*.

Algoritmus prohledávání grafu do hloubky:

1. Na začátku máme v zásobníku pouze vstupní vrchol w. Dále si u každého vrcholu v pamatujeme značku z_v, která říká, zda jsme vrchol již navštívili. Vstupní vrchol je označený, ostatní vrcholy nikoliv.
2. Odebereme vrchol ze zásobníku, nazvěme ho u.
3. Každý neoznačený vrchol, do kterého vede hrana z u, přidáme do zásobníku a označíme.
4. Kroky 2 a 3 opakujeme, dokud není zásobník prázdný.

Na konci algoritmu budou označeny všechny vrcholy dosažitelné z vrcholu w, tedy v případě neorientovaného grafu celá komponenta souvislosti obsahující w. To můžeme snadno dokázat sporem: Předpokládáme, že existuje vrchol x, který není označen, ale do kterého vede cesta z w. Pokud je takových vrcholů více, vezmeme si ten nejbližší k w. Označme si y předchůdce vrcholu x na nejkratší cestě z w; y je určitě označený (jinak by x nebyl nejbližší neoznačený). Vrchol y se tedy musel někdy objevit na zásobníku, tím pádem jsme ho také museli ze zásobníku odebrat a v kroku 3 označit všechny jeho sousedy, tedy i vrchol x, což je ovšem spor.

To, že algoritmus někdy skončí, nahlédneme snadno: v kroku 3 na zásobník přidáváme pouze vrcholy, které dosud nejsou označeny, a hned je značíme. Proto se každý vrchol může na zásobníku objevit nejvýše jednou, a jelikož ve 2. kroku pokaždé odebereme jeden vrchol ze zásobníku, musí vrcholy někdy (konkrétně po nejvýše N opakováních cyklu) dojít. Ve 3. kroku probereme každou hranu grafu nejvýše dvakrát (v každém směru jednou). Časová složitost celého algoritmu je tedy lineární vzhledem k počtu vrcholů N a počtu hran M, čili $\mathcal{O}(N + M)$. Paměťová složitost je stejná, protože si tak jako tak musíme hrany a vrcholy pamatovat a zásobník není větší než paměť na vrcholy.

Prohledávání do hloubky implementujeme nejsnáze rekurzivní funkcí. Jako zásobník v tom případě používáme přímo zásobník programu, kde si program ukládá návratové adresy funkcí. Může to vypadat třeba následovně:

```
var Oznacen: array[1..MaxN] of Boolean;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Oznacen[V] := True;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if not Oznacen[Sousedi[I]] then
      Projdi(Sousedi[I]);
end;
```

Rozdělit neorientovaný graf na komponenty souvislosti je pak už jednoduché. Projdeme postupně všechny vrcholy grafu a pokud nejsou v žádné z dosud označených komponent grafu, přidáme novou komponentu tak, že graf z tohoto vrcholu prohledáme do hloubky. Vrcholy značíme přímo číslem komponenty, do které patří. Protože prohledáváme do hloubky několik oddělených částí grafu, každou se složitostí $\mathcal{O}(N_i + M_i)$, kde N_i a M_i je počet vrcholů a hran komponenty, vyjde dohromady

složitost $\mathcal{O}(N+M)$. Nic nového si ukládat nemusíme, a proto je paměťová složitost stále $\mathcal{O}(N+M)$.

```
var Komponenta: array[1..MaxN] of Integer;
    NovaKomponenta: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Komponenta[V] := NovaKomponenta;
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Komponenta[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
end;

var I: Integer;
begin
  ...
  for I := 1 to N do Komponenta[I] := -1;
  NovaKomponenta := 1;
  for I := 1 to N do
    if Komponenta[I] = -1 then
      begin
        Projdi(I);
        Inc(NovaKomponenta);
      end;
  ...
end.
```

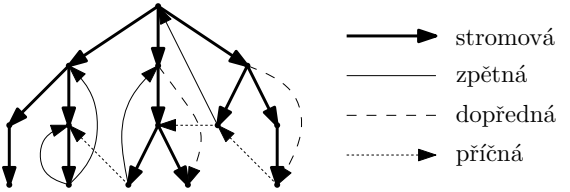
Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom – podle anglického názvu Depth-First Search pro prohledávání do hloubky). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny každého vrcholu si uspořádáme v pořadí, v němž jsme je navštívili; tomuto pořadí budeme říkat *zleva doprava* a také ho tak budeme kreslit. Hranám mezi otci a syny budeme říkat *stromové hrany*. Protože jsme do žádného vrcholu nešli dvakrát, budou opravdu tvořit strom. Hrany, které vedou do již navštívených vrcholů na cestě, kterou jsme přišli z kořene, nazveme *zpětné hrany. Dopředné hrany* vedou naopak z vrcholu blíže kořeni do už označeného vrcholu dále od kořene. A konečně *příčné hrany* vedou mezi dvěma různými podstromy grafu.

Všimněte si, že při prohledávání neorientovaného grafu objevíme každou hranu dvakrát: buďto poprvé jako stromovou a podruhé jako zpětnou, a nebo jednou jako zpětnou a podruhé jako dopřednou. Příčné hrany se objevit nemohou – pokud by příčná hrana vedla doprava, vedla by do dosud neoznačeného vrcholu, takže by se prohledávání vydalo touto hranou a nevznikl by oddělený podstrom; doleva rovněž vést nemůže: představme si stav prohledávání v okamžiku, kdy jsme opouštěli levý vrchol této hrany. Tehdy by naše hrana musela být příčnou vedoucí doprava, ale

o té už víme, že neexistuje.

Prohledávání do hloubky lze tedy také využít k nalezení kostry neorientovaného grafu, což je strom, který jsme prošli. Rovnou při tom také zjistíme, zda graf neobsahuje cyklus: to poznáme tak, že nalezneme zpětnou hranu různou od té stromové, po níž jsme do vrcholu přišli.

Pro orientované grafy je situace opět trochu složitější: stromové a dopředné hrany jsou orientované vždy ve stromě shora dolů, zpětné zdola nahoru a příčné hrany mohou existovat, ovšem vždy vedou zprava doleva, čili pouze do podstromů, které jsme již prošli (nahlédneme opět stejně).



Strom prohledávání do hloubky a typy hran

Prohledávání do šířky

Algoritmus prohledávání do šířky je založený na podobné myšlence jako prohledávání do hloubky, pouze místo zásobníku používá frontu:

1. Na začátku máme ve frontě pouze jeden prvek, a to zadaný vrchol w. Dále si u každého vrcholu x pamatujeme číslo $H[x]$. Všechny vrcholy budou mít na začátku $H[x] = -1$, jen $H[w] = 0$.
2. Odebereme vrchol z fronty, označme ho u.
3. Každý vrchol v, do kterého vede hrana z u a jeho $H[v] = -1$, přidáme do fronty a nastavíme jeho $H[v]$ na $H[u] + 1$.
4. Kroky 2 a 3 opakujeme, dokud není fronta prázdná.

Podobně jako u prohledávání do hloubky jsme se dostali právě do těch vrcholů, do kterých vede cesta z w (a označili jsme je nezápornými čísly). Rovněž je každému vrcholu přiřazeno nezáporné číslo maximálně jednou. To vše se dokazuje podobně, jako jsme dokázali správnost prohledávání do hloubky.

Vrcholy se stejným číslem tvoří ve frontě jeden souvislý celek, protože nejprve odebereme z fronty všechny vrcholy s číslem n, než začneme odebírat vrcholy s číslem $n + 1$. Navíc platí, že $H[v]$ udává délku nejkratší cesty z vrcholu w do v. Že neexistuje kratší cesta, dokážeme sporem: Pokud existuje nějaký vrchol v, pro který $H[v]$ neodpovídá délce nejkratší cesty z w do v, čili vzdálenosti $D[v]$, vybereme si z takových v to, jehož $D[v]$ je nejmenší. Pak nalezneme nejkratší cestu z w do v a její předposlední vrchol z. Vrchol z je bližší než v, takže pro něj už musí být $D[z] = H[z]$. Ovšem když jsme z fronty vrchol z odebírali, museli jsme objevit i jeho souseda v, který ještě nemohl být označený, tudíž jsme mu museli přidělit $H[v] = H[z] + 1 = D[v]$, a to je spor.

Prohledávání do šířky má časovou složitost taktéž lineární s počtem hran a vrcholů. Na každou hranu se také ptáme dvakrát. Fronta má lineární velikost k počtu vrcholů, takže jsme si oproti prohledávání do hloubky nepohoršili a i paměťová složitost je $\mathcal{O}(N + M)$. Algoritmus implementujeme nejsnáze cyklem, který bude pracovat s vrcholy v poli představujícím frontu.

```
var Fronta, H: array[1..MaxN] of Integer;
    I, V, Prvni, Posledni: Integer;
    PocatecniVrchol: Integer;
begin
  ...
  for I := 1 to N do H[I] := -1;
  Prvni := 1;
  Posledni := 1;
  Fronta[Prvni] := PocatecniVrchol;
  H[PocatecniVrchol] := 0;

  repeat
    V := Fronta[Prvni];
    for I := Zacatky[V] to Zacatky[V+1]-1 do
      if H[Sousedi[I]] < 0 then begin
        H[Sousedi[I]] := H[V]+1;
        Inc(Posledni);
        Fronta[Posledni] := Sousedi[I];
      end;
    Inc(Prvni);
  until Prvni > Posledni;  { Fronta je prázdná }
  ...
end.
```

Prohledávání do šířky lze také použít na hledání komponent souvislosti a hledání kostry grafu.

Topologické uspořádání

Teď si vysvětlíme, co je *topologické uspořádání* grafu. Máme orientovaný graf G s N vrcholy a chceme očíslovat vrcholy čísly 1 až N tak, aby všechny hrany vedly z vrcholu s větším číslem do vrcholu s menším číslem, tedy aby pro každou hranu $e = (v_i, v_j)$ bylo $i > j$. Představme si to jako srovnání vrcholů grafu na přímku tak, aby „šipky“ vedly pouze zprava doleva.

Nejprve si ukážeme, že pro žádný orientovaný graf, který obsahuje cyklus, nelze takovéto topologické pořadí vytvořit. Označme vrcholy cyklu $v_1, \ldots, v_n$, takže hrana vede z vrcholu v_i do vrcholu v_{i-1}, resp. z v_1 do v_n. Pak vrchol v_2 musí dostat vyšší číslo než vrchol v_1, v_3 než v_2, $\ldots$, v_n než v_{n-1}. Ale vrchol v_1 musí mít zároveň vyšší číslo než v_n, což nelze splnit.

Cyklus je ovšem to jediné, co může existenci topologického uspořádání zabránit. Libovolný acyklický graf lze uspořádat následujícím algoritmem:

1. Na začátku máme orientovaný graf G a nastavíme proměnnou p na 1.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana (budeme mu říkat *stok*). Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Proměnnou p zvýšíme o 1.
6. Opakujeme kroky 2 až 5, dokud graf obsahuje alespoň jeden vrchol.

Proč tento algoritmus funguje? Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok: Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Co se při tom může stát?

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Vyhráli jsme, máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále a nové a nové vrcholy. V konečném grafu nemožno.

Algoritmus můžeme navíc snadno upravit tak, aby netratil příliš času hledáním vrcholů, z nichž nic nevede – stačí si takové vrcholy pamatovat ve frontě a kdykoliv nějaký takový vrchol odstraňujeme, zkontrolovat si, zda jsme nějakému jinému vrcholu nezrušili poslední hranu, která z něj vedla, a pokud ano, přidat takový vrchol na konec fronty. Celé topologické třídění pak zvládneme v čase $\mathcal{O}(N + M)$.

Jiná možnost je prohledat graf do hloubky a všimnout si, že pořadí, ve kterém jsme se z vrcholů vraceli, je právě topologické pořadí. Pokud zrovna opouštíme nějaký vrchol a číslujeme ho dalším číslem v pořadí, rozmysleme si, jaké druhy hran z něj mohou vést: stromová nebo dopředná hrana vede do vrcholu, kterému jsme již přiřadili nižší číslo, zpětná existovat nemůže (v grafu by byl cyklus) a příčné hrany vedou pouze zprava doleva, takže také do již očíslovaných vrcholů. Časová složitost je opět $\mathcal{O}(N + M)$.

```
var Ocislovani: array[1..MaxN] of Integer;
    Posledni: Integer;
    I: Integer;

procedure Projdi(V: Integer);
var I: Integer;
begin
  Ocislovani[V] := 0; { zatím V jen označíme }
  for I := Zacatky[V] to Zacatky[V+1]-1 do
    if Ocislovani[Sousedi[I]] = -1 then
      Projdi(Sousedi[I]);
  Inc(Posledni);
  Ocislovani[V] := Posledni;
end;

begin
  ...
  for I := 1 to N do
    Ocislovani[I] := -1;
  Posledni := 0;
  for I := 1 to N do
    if Ocislovani[I] = -1 then Projdi(I);
  ...
end.
```

Hranová a vrcholová 2-souvislost

Nyní se podíváme na trochu komplikovanější formu souvislosti. Říkáme, že neorientovaný graf je *hranově 2-souvislý*, když platí, že:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolné hrany.

Hranu, jejíž odebrání by způsobilo zvýšení počtu komponent souvislosti grafu, nazýváme *most.*

Na hledání mostů nám poslouží opět upravené prohledávání do hloubky a DFS strom. Všimněme si, že mostem může být jedině stromová hrana – každá jiná hrana totiž leží na nějaké kružnici. Odebráním mostu se graf rozpadne na část obsahující kořen DFS stromu a podstrom „visící“ pod touto hranou. Jediné, co tomu může zabránit, je existence nějaké další hrany mezi podstromem a hlavní částí, což musí být zpětná hrana, navíc taková, která není jenom stromovou hranou viděnou z druhé strany. Takovým hranám budeme říkat *ryzí zpětné hrany.*

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky,

protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

Algoritmus je tedy pouhou modifikací procházení do hloubky a má i stejnou časovou a paměťovou složitost $\mathcal{O}(N + M)$. Zde jsou důležité části programu:

```
var Hladina, Spojeno: array[1..MaxN] of Integer;
    DvojSouvisle: Boolean;
    I: Integer;

procedure Projdi(V, NovaHladina: Integer);
var I, W: Integer;
begin
  Hladina[V] := NovaHladina;
  Spojeno[V] := Hladina[V];

  for I := Zacatky[V] to Zacatky[V+1]-1 do
  begin
    W := Sousedi[I];
    if Hladina[W] = -1 then
    begin { stromová hrana }
      Projdi(W, NovaHladina + 1);
      if Spojeno[W] < Spojeno[V] then
        Spojeno[V] := Spojeno[W];
      if Spojeno[W] > Hladina[V] then
        DvojSouvisle := False; { máme most }
    end
    else { zpětná nebo dopředná hrana }
    if (Hladina[W] < NovaHladina-1) and
       (Hladina[W] < Spojeno[V]) then
      Spojeno[V] := Hladina[W];
  end;
end;

begin
  ...
  for I := 1 to N do
    Hladina[I] := -1;
  DvojSouvisle := True;
  Projdi(1, 0);
  ...
end.
```

Další formou souvislosti je *vrcholová souvislost*. Graf je *vrcholově 2-souvislý*, právě když:

- má alespoň 3 vrcholy,
- je souvislý,
- zůstane souvislý po odebrání libovolného vrcholu.

Artikulace je takový vrchol, který když odebereme, zvýší se počet komponent souvislosti grafu.

Algoritmus pro zjištění vrcholové 2-souvislosti grafu je velmi podobný algoritmu na zjišťování hranové 2-souvislosti. Jen si musíme uvědomit, že odebíráme celý vrchol. Ze stromu procházení do hloubky může odebráním vrcholu vzniknout až několik podstromů, které všechny musí být spojeny zpětnou hranou s hlavním stromem. Proto musí zpětné hrany z podstromu určeného vrcholem v vést až *nad* vrchol v. Speciálně pro kořen nám vychází, že může mít pouze jednoho syna, jinak bychom ho mohli odebrat a vytvořit tak dvě nebo více komponent souvislosti. Algoritmus se od hledání hranové 2-souvislosti liší jedinou změnou ostré nerovnosti na neostrou, sami zkuste najít, které nerovnosti.

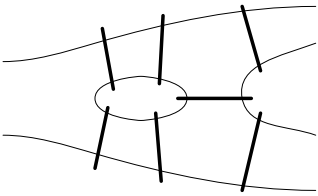
Dnešní menu Vám servírovali
Martin Mareš, David Matoušek a Petr Škoda

Kuchařka druhé série – procházky po grafech

Tento spisek hojně používá jazyka teorie grafů. Pokud ještě termíny jako „hrana“, „cesta“ nebo „stupeň vrcholu“ neznáte, přečtěte si prosím nejprve úvodní kuchařku o grafech.

Historický problém

V roce 1735 se švýcarskému matematikovi Leonhardu Eulerovi na stůl dostal na první pohled jednoduchý problém, který mu předložil starosta města Královec (dnešní Kaliningrad). Královcem teče řeka Pregola, na ní je několik ostrovů a ostrovy byly spojeny se zbytkem města mosty. Dobová ilustrace situaci vystihla takto (schématická kresba):



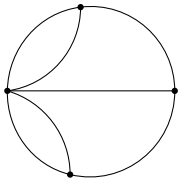
Pan starosta se pana matematika v dopise tázal, jestli je možné začít z některého z břehů (nebo ostrovů) a udělat si vycházku po městě tak, že každým mostem projdeme právě jednou.

Navíc chtěl procházku skončit na kusu suché země, ze kterého jsme vyšli. Euler jej nejprve chtěl poslat k šípku – problém jde snadno vyřešit rozborem případů, což by zvládli i tehdejší studenti střední školy (natož pak ti dnešní).

Profesor Euler se ovšem zachoval jako pravý mamematik – přišel na to, jak problém zobecnit, a mistrně vyřešil hádanku i pro všechna možná města, která kdy budou chtít pořádat podobné procházky.

Eulerovský tah

Pojďme si nyní problém popsat abstraktně a tím si připomenout grafovou terminologii. Vrcholy našeho grafu jsou kusy pevniny, ať už to budou části města nebo ostrovy. Mezi dvěma vrcholy povede hrana, pokud jsou spojeny mostem, a onen most odpovídá hraně. V tomto zadání má smysl uvážit, že mezi dvěma kusy pevniny povede mostů více – například v Praze jich vede tolik, že se na to ptají v leckteré zeměpisné olympiádě. Graf, kde mezi vrcholy vede více hran, nazýváme *multigraf*, a pokud dvě hrany vedou mezi stejnými vrcholy, mluvíme o nich jako o *paralelních* hranách.



Obecná procházka v grafu z vrcholu A do vrcholu B (posloupnost hran taková, že cílový vrchol předchozí hrany je počáteční vrchol hrany následující) se nazývá *sled* z A do B. Ve sledu se mohou opakovat jak hrany, tak vrcholy; sled tedy není řešením našeho problému (ve sledu je možné se vrátit po hraně, ze které jsme právě přišli). Pro naši úlohu se hodí posloupnost hran taková, že vrcholy se opakovat mohou, ale hrany nikoli. Této posloupnosti se říká *tah* z A do B. Kdyby se neopakovaly ani vrcholy, pak posloupnost označujeme jako *cestu*. Tah (respektive sled) je *uzavřený*, pokud začíná v A a končí také v A.

Podíváme-li se tedy na mapu Královce jako na multigraf, ptáme se, zdali existuje uzavřený tah takový, že každou hranu navštíví právě jednou. Takovému tahu pak říkáme *uzavřený eulerovský*.

Mimochodem, tahu se „tah“ neříká jen tak náhodou. Děti se často ve školce překonávají v umění nakreslit obrázek jedním tahem, aby se tužkou nemuselo vracet po už nakreslené čáře. Pokud si obrázek představíme jako graf (čáry jsou hrany, místa jejich setkání vrcholy), pak eulerovský tah nalezneme jen v tom obrázku, který lze nakreslit jedním tahem. V uzavřeném eulerovském tahu se pak vrátíme i do místa, kde jsme začali.

Podmínky tahu

Je na čase poodhalit řešení našeho problému s eulerovským tahem. Půjdeme na to jako matematici – nejprve ukážeme *nutnou* a hned nato *postačující* podmínku. Nutná vlastnost grafu je taková, že bez ní eulerovský tah není možné najít; postačující vlastnost je ta, se kterou vždy eulerovský tah najít umíme. Je-li podmínka nutná a postačující zároveň, pak se jedná o ekvivalenci, a tak tomu bude i nyní.

Představme si, že jsme kouzlem nějaký uzavřený eulerovský tah našli, ať už je jakýkoli. Vždy, když se dostaneme do jednoho vrcholu (a není důležité, jestli už jsme v něm byli, nebo ne), tak abychom tah uzavřeli, musíme z něj hned také odejít. A protože tah je eulerovský, každou hranou projdeme jen jednou, takže tyto dvě hrany (tu příchozí a odchozí) už nepoužijeme. U každého vrcholu mimo výchozí tedy platí, že hrany tvoří dvojice – jedna, co vedla dovnitř, a jedna, která z něj vedla ven.

Podobná věc platí i pro startovní vrchol. Sice do něj nevstoupíme poprvé pomocí hrany, takže počet navštívených hran u něj bude stále lichý – ale jen do chvíle, než se do něj naposledy vrátíme a skončíme, protože skončením jsme použili poslední hranu, která bude tvořit dvojici s hranou první.

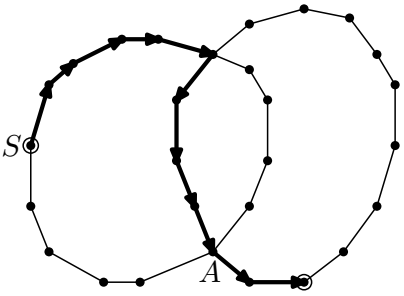
Jakou vlastnost grafu jsme odhalili? Neplatí, že graf má sudý počet hran (protože trojúhelník jedním tahem nakreslíme a přesto má 3 hrany), ale platí, že do každého vrcholu vede sudý počet hran, tedy že graf má **všechny stupně sudé**. Nezapomeňme také na to, že graf musí být souvislý – dva oddělené obrázky jedním tahem bez zvednutí tužky nenakreslíme. Máme nutné podmínky!

Nalezení tahu

Zbývá tedy ověřit, že podmínky jsou i postačující. Mějme souvislý graf, který má všechny stupně sudé. Umíme v něm vždy najít uzavřený eulerovský tah? Ověřme to, jak se na informatiky patří – algoritmem.

Předložený algoritmus je založený na vylepšeném prohledáváni do hloubky, tedy DFS. To patří do základního arzenálu každého programátora, jeho popis naleznete třeba v předchozí kuchařce.

Vyberme si vrchol, v něm začneme. Náš algoritmus musí umět označovat hrany jako „probrané“ jako to dělá DFS. Vyberme si tedy jednu hranu, a pokračujme dále, zatím bez vypisování.



Po nějakém tom procházení se jistě stane, že jsme se zastavili – vrchol už nemá žádné nepoužité hrany. Nutně to znamená, že to je ten vrchol, u kterého jsme začínali. V procházení do hloubky se vracíme zpět, ale my k tomu přidáme vypisování cesty – postupně pozpátku vypisujeme hrany, kterými se vracíme zpět v prohledávání.

Na obrázku vlevo je příklad právě probíhajícího algoritmu. Začal ve zvýrazněném vrcholu vlevo, procházel po šipkách až do bodu A, kde volil hrany tak, že hned skončil na začátku. Dále pokračoval vypisováním hran pozpátku, až došel zase do bodu A. Zde si vybral jednu ještě nepoužitou hranu a po ní prošel celou druhou kružnici – zbytek hran – zpět do bodu A. Nyní vypisuje hrany pozpátku od bodu A.

Buď tímto výpisem dojdeme až na začátek, nebo se dostaneme do vrcholu, který má ještě nějaké nepoužité hrany (situace může vypadat třeba jako na obrázku). Potom vypisování zastavíme a pokračujeme v prohledávání DFS přes nepoužitou hranu. I tam se to může zastavit (a zastaví), i tam začneme vypisovat pozpátku. Nakonec dojdeme do původního místa rozbočení, a budeme opět pozpátku vypisovat hrany, které nás nakonec dostanou až na počátek, kde skončíme.

Najde tento algoritmus opravdu korektní uzavřený eulerovský tah? Graf byl souvislý a o algoritmu DFS se ví, že v takovém případě navštíví každou hranu právě jednou.

Algoritmus opravdu vypisuje cyklus – jen je u něj trochu zvláštní způsob, jak ho vypisuje. Když dojde na křižovatku s ještě nepoužitými hranami, tak výpis zastaví, tiše po nich kráčí, označuje si je a vypisuje, až když se po nich vrací. Ověřme si, že hrany opravdu navazují.

V duchu argumentů z předcházející části víme, že jediný vrchol grafu s lichým počtem nepoužitých hran je právě ona křižovatka – a algoritmus DFS prochází graf podobně, jako jsme ho procházeli v minulé sekci, takže právě do tohoto vrcholu algoritmus dojde, až se průchod touto částí grafu zastaví. Jakmile sem program dojde (a nezbudou mu volné hrany), začne cestovat zpět a hrany vypisovat – a opravdu, pokračuje se tedy z místa, kde naposledy přestal, a program vskutku vypíše tah přes všechny hrany v grafu – uzavřený eulerovský tah.

Věta o eulerovském tahu v celé své kráse tedy zní:

(Multi)graf obsahuje uzavřený eulerovský tah právě tehdy, když má všechny stupně sudé a je souvislý.

Je třeba podotknout, že složitost našeho algoritmu na bázi DFS je lineární vůči velikosti grafu (počtu vrcholů a hran). Existují i jiné algoritmy pro hledání eulerovského tahu, jedna varianta například prochází grafem a vybírá si na křižovatkách takové hrany, které souvislost grafu pokud možno nepoškodí. Tyto algoritmy už nemusí mít nutně lineární časovou složitost.

Jiné druhy procházek

Nejen kreslením obrázků ze stejného bodu živ je člověk. Co kdybychom mohli začít a skončit v jiném místě, tedy ptali se po neuzavřených eulerovských tazích, změnilo by se něco? Není tomu tak, pouze nutné a postačující podmínky si vyžádají, aby všechny vrcholy měly sudý stupeň až na právě dva vrcholy, které mají lichý stupeň. Pokud nám to nevěříte, zkuste si to rozmyslet sami, opravdu to není těžké.

Smysl také dává zkusit najít ne uzavřený tah, ale uzavřenou cestu – uzavřenou cestu přes všechny vrcholy, která navštíví každý vrchol právě jednou. Bohužel, ačkoli jsou problémy příbuzné, musíme vás zklamat – není znám žádný efektivní (polynomiální) algoritmus na tento problém, a kdyby jej někdo z vás nalezl, vyřešil by otázku „P vs. NP[20]“ a získal alespoň milion dolarů. Chcete-li si o tomto problému přečíst něco dalšího, napište do vyhledávače „Hamiltonovská cesta“ – tak se ona úloha jmenuje.

V matematice se také někdy zmiňují „náhodné procházky“ po grafech – můžete si je představit tak, že se po mostech města Královce motá opilec, který si hází (opilou nebo spravedlivou) mincí a podle toho se rozhoduje, přes který most jít dál. Použití mají tyto modely hlavně v matematické teorii grafů a teorii pravděpodobnosti. O tom si můžeme povědět zase někdy jindy.

Dnešní menu uvařil a servíruje
Martin Böhm

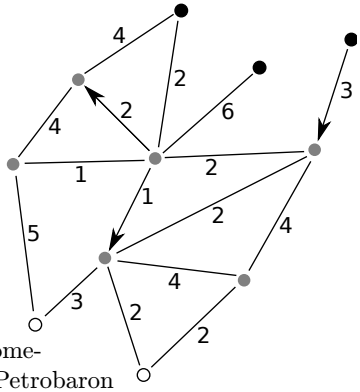
[20] viz kuchařku páté série, str. 65

Kuchařka čtvrté série – toky v grafech

Ukážeme si uměle znějící úlohu, kterou posléze zmatematizujeme, vyřešíme a dokážeme vlastnosti řešení. Nakonec přijdou četná užití, která ozřejmí, proč jsme se snažili.

Látka je lehce pokročilá, takže věžte, že budete potřebovat znát grafy.

Uměle znějící úloha



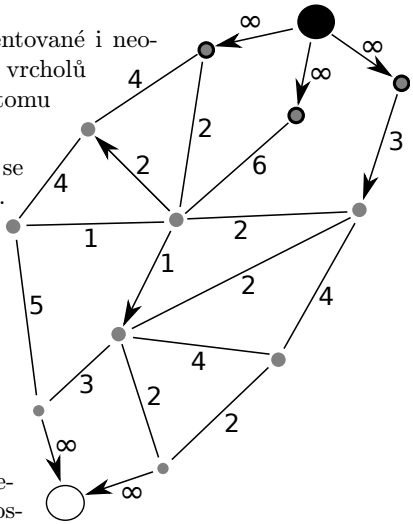
Ruský petrobaron vlastní ropná naleziště na Sibiři a trubky vedoucí do Evropy. Trubky vedou mezi nalezišti, uzlovými body a koncovými body, kde ropu přebírají odběratelé.

Každá trubka může a nemusí mít definováno, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

Naleziště jsou bezedná a mohou posílat neomezená množství ropy. Odběratelé také dokáží neomezená množství ropy z koncových bodů odebírat. Petrobaron čelí problému, jak protlačit danou distribuční sítí co nejvíce ropy za hodinu ze zdrojů k odběratelům.

Zapeklité je to zejména kvůli tomu, že v uzlových bodech nelze ropu hromadit, ani pálit – rozhodně tedy nejde bez rozmyslu přikázat, ať každou trubkou teče maximum, protože bychom poškodili cenná zařízení a v hnusu labutě zahubili.

Zmatematizování



V zadání vidíme graf, který obsahuje orientované i neorientované hrany, kde je nějaká podmnožina vrcholů označená jako zdroje a jiná jako… říkejme tomu třeba stoky.

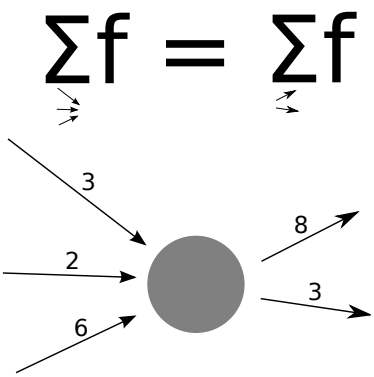
Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.

Teď nám stačí vymyslet algoritmus, který řeší problém s právě jedním zdrojem a právě jedním stokem.

Každý vstup totiž popsaným způsobem převedeme, pošleme ho algoritmu a z výstupu prostě jen odstraníme dva přidané vrcholy a připojené hrany. Podobně se zbavíme neorientovaných hran.

Každou takovou hranu v každém zadání změníme na dvojici protisměrných orientovaných hran se stejnou kapacitou. V algoritmu pak už můžeme počítat jen s hranami orientovanými.

Dostáváme se nyní k nejdůležitějšímu – podmínkám na hledaný tok.



Na vstupu dostáváme ohodnocení hran nezápornými čísly a naším úkolem je sestavit jiné ohodnocení těch samých (všech) hran.

Je důležité, aby se nám to nepletlo – ohodnocení ze vstupu se říká kapacita a značí se $c(e)$, konstruované ohodnocení se jmenuje tok a říkáme mu $f(e)$.

Konstruované ohodnocení se snažíme maximalizovat, ale omezuje nás kapacita a Kirchhoffův zákon.

Tak budeme říkat podmínce na to, že součet toku na hranách, které do vrcholu vstupují, musí být stejný jako součet toku na hranách, které z vrcholu vystupují. Máte-li rádi fyziku nebo berete-li školu vážně, důvod k takovému pojmenování jistě chápete.

Formálně ony dvě podmínky vypadají takto:

$$\forall e \in E : f(e) \leq c(e)$$

$$\forall v \in V \setminus \{z, s\} : \sum_{u\vec{v} \in E} f(u\vec{v}) = \sum_{v\vec{u} \in E} f(v\vec{u})$$

Kirchhoffova podmínka se samozřejmě netýká ani zdroje, ani stoku – tam nám naopak jde o to ji co nejvíce porušit. Velikost toku je nejsnazší měřit na nich. Budeme ji definovat jako rozdíl mezi součtem odtoků a součtem přítoků ve zdroji.

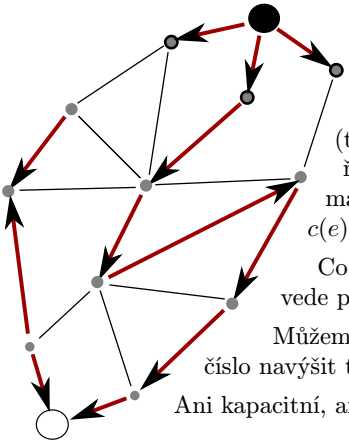
K zamyšlení:

- Nastavit ohodnocení hrany (kapacitu) na skutečné nekonečno v našem programovacím jazyce nemusí jít. Pak se to řeší tím, že se zvolí dostatečně velké číslo. Jak co nejmenší, ale stále bezpečné, rychle ze zadání určit? Stejný problém se řeší třeba v Dijkstrově algoritmu, ale i ve spoustě dalších.
- Neorientované hrany, neboli obousměrné trubky, si zaslouží podrobnější rozbor, než jaký jsme jim věnovali v textu. Jak spolehlivě převedeme řešení algoritmu do původní sítě?
- Vymysleli jsme, jak vyřešit více zdrojů a stoků a jak ošetřit obousměrné trubky. Co kdyby bylo v zadání omezení na průtok vrcholy?
- Umíte dokázat, že je absolutní hodnota rozdílu přítoků a odtoků stejná na zdroji i na stoku? Tedy že bychom mohli velikost toku stejně tak dobře měřit i na stoku?

Řešení

Problém je velmi studovaný a k jeho řešení existují dva velké přístupy, které jsou humorně protikladné. Ten první vezme nulový tok a opatrně ho zlepšuje. Druhý si napíská veliké ohodnocení hran, které ani tokem není, a pak ho opravuje.

Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, jako že mezi každými dvěma vrcholy vede oběma směry hrana. Tam, kde ze vstupu nepřišla, si domyslíme jednu s nulovou kapacitou.



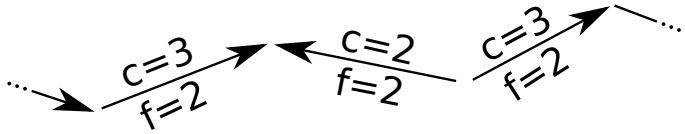
Představme si graf, na kterém počítáme tok a dejme tomu, že už nějaký tok máme – třeba prázdný. Představme si, že jsme ropný magnát a každý rozdíl mezi kapacitou potrubí a jejím využitím (tokem) nás stojí miliony dolarů. Už jsme se smířili s tím, že každá trubka nemůže být využita na maximum, ale… zkusme si vyznačit ty hrany, kde $c(e) \neq f(e)$.

Co když existuje cesta z nadzdroje do nadstoku, která vede pouze po takových hranách?

Můžeme vzít minimum z rozdílů na každé hraně a o toto číslo navýšit tok na každé z nich!

Ani kapacitní, ani Kirchhoffovu podmínku to jistě nepoškodí.

Pokud žádnou takovou cestu nevidíme, znamená to, že tok vylepšit nejde? Ne úplně. Představte si následující situaci:



Copak nejde zlepšit? Jde! Není na to první pohled úplně jasné, ale můžeme zlepšovat výsledný tok i tím, že ho na protisměrné části cesty snížíme. Samozřejmě však nesmíme nastavovat tok záporný.

(Je smutné, že si teď trochu kazíme grafovou terminologii – co je to za cestu v orientovaném grafu, která nemusí respektovat orientaci hran?)

Takže jaká je přesně podmínka pro „vyznačení“ hrany $\vec{uv}$? Nastává $f(\vec{uv}) < c(\vec{uv})$ nebo $f(\vec{vu}) > 0$. Potom ji lze zlepšit o $c(\vec{uv}) - f(\vec{uv}) + f(\vec{vu})$.

Hledání všech vhodných („zlepšujících“) cest tedy můžeme dělat prostým prohledáváním do šířky přes vyznačené hrany. Budeme to dělat opakovaně znovu a znovu, až žádnou takovou nenajdeme, a pak vrátíme získaný tok jako výsledek.

Analýza algoritmu

Správnost

Zavolali jsme algoritmus na prázdný tok, ten ho zlepšil do situace, ve které neexistuje zlepšující cesta. Znamená to, že je výsledný tok maximální? Opačná implikace je jasná – maximální tok zlepšit žádným způsobem nepůjde, takže ani přes zlepšující cestičky.

Když zkusíme algoritmus pustit na graf, kde už žádná taková cesta není, můžeme si poznamenat všechny vrcholy, kam jsme se pomocí prohledávání zlepšitelných hran ještě dostali. Tato množina bude jistě obsahovat zdroj (tam jsme začali) a jistě nebude obsahovat stok (to by existovala zlepšující cesta).

Na hranách mezi touto množinou a jejím doplňkem nemůžeme zlepšovat, jinak by se po nich náš program pustil dál a množinu vrcholů, kam se dostal, by rozšířil. Všechny hrany směřující ven tedy mají $f(e) = c(e)$, pro všechny hrany směřující dovnitř platí $f(e) = 0$.

Tyto hrany tvoří *řez* naším grafem. Dovolám se v tuto chvíli na vaši intuici – tok nemůže být větší než libovolný řez. Z toho už dostáváme, že náš algoritmus našel tok maximální, protože našel také řez, který zaručuje, že nemůže existovat tok větší. Formálnější předvedení najdete ve skriptíčkách z kombinatoriky.[21]

Časová složitost

Je možné dobu běhu omezit počtem vrcholů a hran? Výše uvedeným postupem na grafu s celočíselnými kapacitami každou nalezenou cestou zvýšíme tok alespoň o jednotku, takže program nebude běžet déle, než je součet všech kapacit. Ale to není moc uspokojivý odhad, protože záleží na ohodnocení.

Pokud budeme hledat cesty skutečně prohledáváním do šířky, bude počet kroků v $\mathcal{O}(nm^2)$, protože se dá ukázat, že se hrany, které při zlepšování cesty tvoří minimum, postupně vzdalují od zdroje. Pak máme $\mathcal{O}(m)$ času k nalezení cesty a m hran, které se nejvýše n-krát mohou vzdálit. Že to tak skutečně je, je lehce zdlouhavé intelektuální cvičení. Nechat si prozradit postup můžete třeba v druhém vydání Introduction to Algorithms na straně 662.

O vylepšení daného postupu si můžete přečíst v záznamu[22] z jedné Medvědovy přednášky předmětu ADS2, ukázka druhého přístupu k řešení hledání maximálního toku je na záznamu[23] jejího pokračování.

K zamyšlení:

- Důležitou vlastností algoritmu je, že když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Dokážete to?
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela velký, a proto se tomu druhému občas říká Edmondsův-Karpův. Najděte malý graf a nevhodnou

[21] `http://kam.mff.cuni.cz/~valla/kg.html`

[22] `http://mj.ucw.cz/vyuka/0910/ads2/2-dinic.pdf`

[23] `http://mj.ucw.cz/vyuka/0910/ads2/3-goldberg.pdf`

posloupnost cest, která způsobí, že F-F poběží skutečně v závislosti na velikosti kapacit.

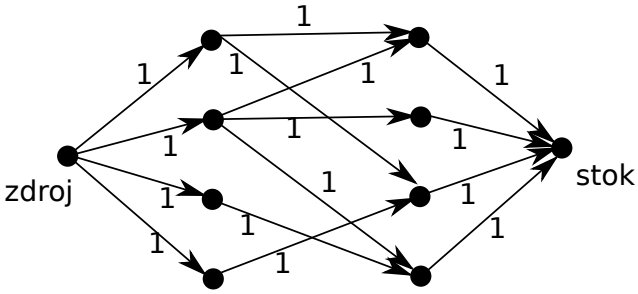
- Můžete dokonce zkusit využít zlatého řezu k nalezení grafu s reálnými kapacitami, na kterém F-F pro danou (nešikovnou) posloupnost cest nikdy neskončí.
- Skončí algoritmus v konečném čase, jsou-li kapacity čísla racionální?

Užití

Párování v bipartitních grafech

Máme-li za úkol najít na plese co nejvíce tanečnicím tanečníka, kterého znají, stojíme před zásadním a nelehkým úkolem.

Co třeba postavit na základě známosti bipartitní graf mezi partitou tanečníků a partitou tanečnic, přidat zdroj za kluky a stok za holky, tyto k nim připojit hranami s jednotkovou kapacitou, hranám v bipartitním grafu také nastavit jednotkové kapacity a nakonec všechno zorientovat směrem do stoku?



Maximální celočíselný tok, který na tomto grafu získáme, nám hrany bipartitního grafu rozdělí na nevybrané s tokem 0 a vybrané s tokem 1. Můžou vybrané hrany sdílet tanečníka? Těžko, když do něj teče nejvýše jednotkový tok a musí platit Kirchhoffův zákon. A podobně s tanečnicemi.

Vybrané hrany nám proto vytvoří párování. A protože jsme našli maximální tok, jde o párování největší. Kdyby existovalo párování větší, dokázali bychom z něj zvětšit tok.

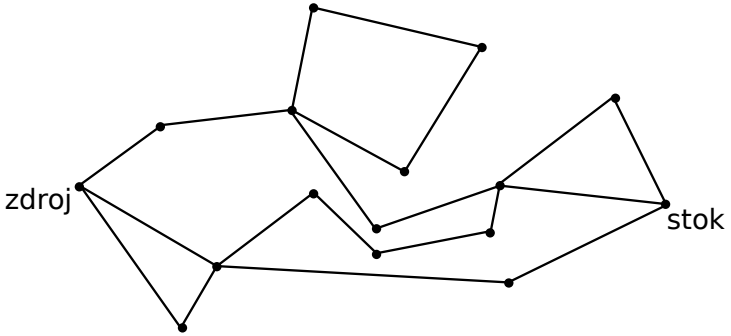
Hledání hranově a vrcholově disjunktních cest

Chceme-li se v grafu G dostat z vrcholu u do vrcholu v, může nás zajímat (třeba kvůli spolehlivosti, s jakou se umíme dostat do cíle), kolik mezi nimi vede cest, které

- nesdílí hrany, nebo
- nesdílí vrcholy. (Tato podmínka je silnější. Když dvě cesty nesdílí vrcholy, nesdílí hrany.)

Oba tyto problémy lze převést na hledání maximálního toku. V obou případech nastavíme u jako zdroj a v jako stok. V prvním případě nastavíme jednotkové kapacity všem hranám, v druhém navíc všem vrcholům.

Ford-Fulkerson nastavil některým hranám jednotkový tok, některým nulový. Nulové nyní z grafu vyhodíme. Pokud jsme hledali hranově disjunktní cesty, můžeme nyní získat třeba takovýto graf:



Jak z něj vykřesat kýžený výsledek? Začneme procházet ze zdroje zbylé hrany. Vždy, když se dostaneme do vrcholu, ve kterém už jsme v tom samém průchodu byli, vyhodíme z grafu všechny hrany cyklu, který jsme tímto objevili. (Hodnota toku se tím nezmění.)

Průchodem grafu se vždy můžeme dostat až do stoku (všude jinde budeme moci podle Kirchhoffova zákona jít dál – dost to připomíná úvahu o eulerovských tazích)[24] a protože jsme mezitím agilně odstraňovali cykly, dostali jsme cestu. Vrátíme ji jako jeden výsledek, smažeme její hrany a pokud ještě tok není nulový, pokračujeme dál.

Počet cest je tedy velikost toku. Podle Mengerovy věty je počet hranově/vrcholově disjunktních cest roven stupni hranové/vrcholové souvislosti grafu – máme tedy nyní algoritmus, který ji najde.

K zamyšlení:

- Úvaha nebyla naprosto přímočará kvůli cyklům v nalezeném toku. Říká se jim cirkulace. Je jasné, že v případě hledání hranově disjunktních cest vzniknout mohou. Co v případě vrcholově disjunktních, tedy v situaci, kdy jsme omezili tok vrcholy?
- Nepracuje neupravený Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

[24] `http://ksp.mff.cuni.cz/viz/kucharky/eulerovske-tahy`

Kuchařka páté série – těžké problémy

V této kuchařce konečně vysvětlíme, cože je to onen velmi známý problém za milion dolarů – *Je P rovno NP?*. Než se k němu dostaneme, budeme si muset ujasnit, které problémy jsou v informatice vlastně „těžké“.

Kuchařka není nijak komplikovaná, ale doporučujeme si aspoň oprášit, co to znamená lineární a exponenciální časová složitost.

S mapou v bludišti

Představme si, že jsme v bludišti a hledáme (náš algoritmus hledá) nejkratší cestu ven. Rychle nás napadne, že bychom mohli použít prohledávání do šířky[25] a cestu najít v čase lineárním ku velikosti bludiště. To je asymptoticky nejlepší možné řešení, v nejhorším případě bude totiž bludiště jedna dlouhá nudle a i nejkratší cesta bude dlouhá lineárně vůči velikosti bludiště.

Ve skutečném životě však „kulišáci“ znají lepší řešení – podvádět! Prostě si od kamaráda půjčíme mapu bludiště s vyznačenou nejkratší cestou a pak poběžíme hned tou nejkratší cestou, aniž bychom kdekoli ztráceli čas.

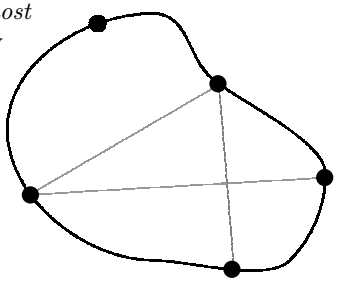
V nudlovém bludišti (nejkratší cesta má zhruba stejně vrcholů jako celý graf) jsme si vůbec nepomohli (takže je řešení asymptoticky stejně dobré). V alespoň trochu spletitém bludišti už budeme v cíli dříve než náš kamarád, který bloudí (prohledává) do šířky.

Existují tedy problémy, kde by se i v nejhorším možném případě vyplatilo podvádět pomocí taháku? Ano, zde je příklad – opět jsme v labyrintu, ale tentokrát jsou na všech stanovištích umístěny koláčky. Labyrint je to zvláštní, cesty se v něm nekříží, ale je tam plno nadchodů a podchodů.

Naším cílem je najít okružní cestu ze startovního místa zpátky na start, abychom každé stanoviště s koláčkem prošli právě jednou (protože víc než jeden koláček nám nedají).

Kdybychom tady chtěli použít procházení do šířky, bylo by to opět možné – ale tentokrát bychom se museli mnohokrát vracet, protože posloupnost stanovišť (začátek, první, druhé) může být špatná, zatímco posloupnost (začátek, druhé, první) už může být dobrá.

Přesněji řečeno, už by neplatilo, že při prohledávání do šířky každé stanoviště navštívíme nejvýše jednou, ale každou *posloupnost* stanovišť navštívíme nejvýše jednou. Projít všechny nám potrvá, matematicky řečeno, exponenciálně mnoho času vůči velikosti bludiště.



Kamarád s tahákem je na tom pořád dobře – prostě si pořídí jiný plán, na kterém bude mít vyznačenou cestu, po které má jít, aby vyhrál.

[25] `http://ksp.mff.cuni.cz/tasks/20/cook3.html`

Ta cesta má stejně křižovatek jako bludiště samo, a tak bude jeho nápověda lineárně velká vůči velikosti bludiště a průchod napovězenou cestou bude trvat také lineárně. Podvodník tedy vyhrává i asymptoticky. Bídák!

Všechny by nás zajímalo, jestli by bylo možné najít tu nejlepší cestu bez podvádění v rozumně krátkém (řekněme polynomiálním) čase. Tato otázka je ekvivalentní otázce P vs. NP. Pojďme ta tajemná písmena přesně definovat.

Podvádíme s certifikáty

V teorii složitosti se často omezujeme jen na jeden typ problému, takzvaný *rozhodovací problém*. To je vlastně otázka, na kterou existují dvě možné odpovědi: ANO, nebo NE. Například „Existuje cesta z bludiště délky k?“ nebo „Je součet čísel $8 + 3$ roven 5?“

Ve zbytku kuchařky už budeme pracovat jen s nimi – skoro vždy se rychlé řešení rozhodovacího problému dá převést na rychlé řešení příslušného vyhledávacího problému, jako *Nalezněte nejkratší cestu z bludiště*.

Rozhodovací problém (dále už jen problém) bude náležet do *třídy problémů P* (třída je zde jen pomocné označení pro nekonečnou množinu), pokud existuje polynomiální algoritmus, který pro zadaný vstup odpoví korektně ANO nebo NE na výstupu.

Taháku z předchozí kapitolky se v literatuře říká *certifikát*. Formálně to je jen jakási polynomiálně velká informace. Můžeme si jej představit jako data, která náš program nalezne v „našeptávajícím“ vstupním souboru, ke kterému program z třídy P nemá přístup.

Problém bude náležet do *třídy problémů NP* (nepoctivci), pokud existuje algoritmus a ke každé odpovědi ANO vhodný certifikát tak, že algoritmus je schopen pomocí certifikátu ověřit, že odpověď je skutečně ANO. Čili má-li ten program správný tahák, musí být schopen bludištěm projít rychle.

Zde si dejme pozor na to, že definice nedovoluje „podvádět na druhou“ – nemůžeme si do pomocného souboru prostě uložit ANO a pak jej vypsat. Tak by se pak dal řešit libovolně složitý problém, i problémy mimo třídu NP! Jen tak na okraj – takové opravdu existují.

Onen algoritmus musí být schopen řešení ověřit, tedy odpovědět ANO tehdy a jen tehdy, pokud mu to napověděl certifikát a odpověď je správná. Kdyby byla skutečná odpověď NE a certifikát chybně tvrdil, že ANO, algoritmus musí být napsán tak, aby oznámil NE.

Co přesně bude certifikát, záleží na zadané úloze – často to bývá právě ono nejlepší možné řešení, kterého se stačí držet a najdeme hledanou odpověď (nebo zjistíme, že úloha nemá řešení).

Asi vám bylo hned jasné, že každý program z P patří také do NP – jakmile známe polynomiální řešení bez nápovědy, certifikátem může být i třeba prázdný soubor! Horší je to s problémy, pro které potřebujeme pro polynomiální vyřešení nějaký certifikát a zatím to lépe neumíme.

Příkladem buď problém z povídání o bludišti. Říká se mu *Hamiltonovská kružnice*.

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Certifikát: Posloupnost vrcholů hamiltonovské kružnice.

Ověření v polynomiálním čase s certifikátem: Projdeme postupně vrcholy a ověříme, že jsou opravdu zapojeny do kružnice a kružnice je správné délky. Vrátíme NE, pokud tomu tak není.

Zatím nikdo nepřišel s řešením, které by nepoužívalo vůbec žádný certifikát. Dokonce zatím nikdo nenalezl problém, který by byl v NP, ale bez certifikátu už jej nelze řešit v polynomiálním čase. Kdyby takový neexistoval, třídy P a NP by se rovnaly. To je jádro otevřeného problému P vs. NP.

Převoditelnost a NP-úplnost

Když řešíme nějakou algoritmickou úlohu, obvykle přijdeme na nějaké řešení využívající základních technik (prohledávání do šířky, dynamické programování, zametací přímka). Vzácně se může i stát, že v problému rozpoznáme problém jiný – občas lze geometrický problém převést na třídění čísel nebo umíme popsat situaci nějakým vhodným grafem.

Ukazuje se, že se ve třídě NP často vyplatí problémy převádět, neboť přímá řešení jsou zde vzácná. Dokonce tak můžeme i zjistit, do které z probíraných tříd problém patří.

Převodem budeme rozumět polynomiální algoritmus, který upraví vstup jednoho problému na vstup jiného problému. Musí navíc problémy převést tak, aby správná odpověď (ANO nebo NE) na vstup prvního problému byla tatáž, jako správná odpověď na vstup druhého problému.

Jednoduchým převodem je úprava problému *Existuje cesta z bludiště ze zadaného políčka délky d?* na *Existuje cesta v grafu délky c začínající v zadaném vrcholu?*.

Do výstupního grafu za každou křižovatku dáme vrchol, za každou cestu mezi křižovatkami hranu a ke hraně si poznamenáme, jak dlouhá byla. Hodnotu c pak můžeme nechat stejně velkou, jako d.

Pokud najdu správnou cestu v tomto grafu, pak nutně podobná cesta je i v bludišti, a pokud cesta v grafu není, pak není ani v bludišti. Převod je tedy korektní.

Zadefinujme si nyní pojem, který nám bude sloužit jako zkratka za to, že problém je ve třídě NP, ale není zároveň lehký (v P). Nemůžeme jen tak ledabyle říci „je v NP a není v P“ protože to nevíme. To je právě ta slavná otázka.

Uděláme tedy krok stranou – budeme říkat, že problém je *NP-úplný*, pokud onen problém je v NP a zároveň jdou všechny ostatní problémy v NP převést na tento problém.

Všechny problémy v NP na něj jdou převést? Pokud tuto definici vidíte poprvé, asi to působí dost zvláštně – je těžké si představit, že všechny grafové, geometrické, počítací problémy, o kterých víte, že jsou v P (a tedy i v NP) jdou převést na nějaký NP-úplný superproblém.

Ale je to správně, ba co víc, Cookova věta[26] říká, že existuje alespoň jeden takový problém. (Samotná definice NP-úplného problému nezaručuje, že takový problém vůbec existuje.) Ukazuje se však, že není sám, jsou jich stovky. Dokazovat existenci dalších NP-úplných problémů je však o dost lehčí, než dokázat Cookovu větu! Stačí totiž jen najít následující dva kroky:

- Dokázat, že problém je v NP – najít certifikát a polynomiální algoritmus, co jej využívá.
- Převést zadání libovolného NP-úplného problému na zadání našeho problému tak, že náš algoritmus vlastně vyřeší onen NP-úplný problém.

To postačí, protože pak libovolný jiný problém v NP nejprve převedeme na zvolený NP-úplný problém a pak pustíme námi vymyšlený převod. Zřetězení dvou polynomiálních algoritmů (převodů) je opět polynomiální algoritmus, takže podmínka převoditelnosti je splněna.

Ukážeme si důkaz NP-úplnosti jednoho problému na příkladu, pokud nám uvěříte, že již probíraný problém *Hamiltonovská kružnice* je NP-úplný. Nejprve zadefinujme jiný problém:

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

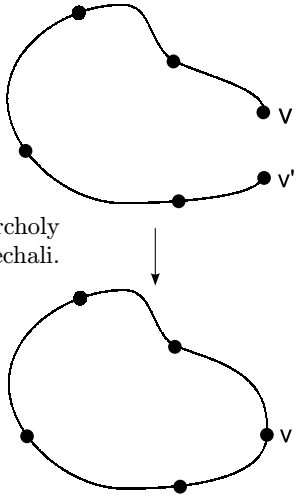
Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Certifikát: Posloupnost vrcholů tvořící správnou cestu.

Řešení v NP: Projděme cestu z certifikátu a ověřme, že vrcholy jdou za sebou, je jich správný počet a žádný jsme nevynechali.

Důkaz NP-úplnosti: Převedeme předchozí problém (hamiltonovskou kružnici) na hledání hamiltonovské cesty. Uvažme graf G, ve kterém chceme najít hamiltonovskou kružnici.

Vyberme si libovolný vrchol v a vytvořme vrchol v', který bude kopií vrcholu v – do grafu přidáme hranu mezi u a v', pokud už v něm je hrana mezi u a v.



Na upravený graf zavoláme řešení problému *Hamiltonovská cesta* mezi vrcholy v a v'. Pokud taková cesta existuje, tak nutně v původním grafu G existuje hamiltonovská kružnice. Cesta z vrcholu v' přesně odpovídá pokračováni kružnice poté, co přijde do vrcholu v.

[26] `http://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem`

Pseudopolynomiální algoritmy

Znáte problém batohu? Jeho varianty jsou oblíbené na programovacích soutěžích. Zadat se může třeba takto: mějme na vstupu seznam n dvojic kladných přirozených čísel, kde každá dvojice označuje váhu a cenu nějakého předmětu. Nakonec dostaneme na vstupu ještě číslo b, které udává nosnost našeho batohu.

Otázka zní: Jaký je nejcennější možný náklad, který přesto nepřesahuje váhový limit batohu?

Možná víte, že úloha jde řešit dynamickým programováním – vytvořím si pole `podbatoh[]` od 1 do b, kde `podbatoh[`i`]` je maximální hodnota, kterou bych si odnesl v batohu o nosnosti i. Postupně od první věci do poslední pak projdu celé pole `podbatoh[]` „zprava doleva“ od b do 1 a zkusím, jestli je výhodnější do batohu vložit novou věc a volné místo doplnit starými (optimální volné místo pro předchozí věci máme napočítané), nebo si nechat jen ty staré. Tuto hodnotu pak zapíšeme jako aktuální pro váhu i na místo `podbatoh[`i`]`.

Po n průchodech tohoto pole dostaneme řešení pro všechny věci dohromady na políčku `podbatoh[`b`]`. Celková složitost je $\mathcal{O}(nb)$, to je polynom, algoritmus je tedy polynomiální.

Světe div se, toto řešení je ve skutečnosti exponenciální. Kde jsme v řešení udělali chybu? Nikde – naše složitost závisela na b, ovšem když se podíváme do vstupních dat, tak pokud jsou zapsána v binárním (nebo ternárním a vyšším) tvaru, tak zápis čísla b byl veliký $\mathcal{O}(\log_2 b)$, ale naše složitost závisela na $b = 2^{\log_2 b}$, tedy exponenciálně vůči velikosti vstupu.

Problém batohu, respektive jeho rozhodovací verze, je dokonce NP-úplný problém.

Algoritmům, které řeší nějakou úlohu a jsou polynomiální oproti *hodnotě* čísel na vstupu, ale exponenciální ve *velikosti zápisu* těchto čísel, říkáme *pseudopolynomiální algoritmy*. Některé další NP-úplné problémy mají pseudopolynomiální řešení (jako například *Dva loupežníci* níže), ale dá se dokázat, že na jiné problémy pseudopolynomiální algoritmus neexistuje (pokud P $\neq$ NP).

Mimochodem: pokud bychom na vstupu zapisovali čísla v unárním zápisu, každý pseudopolynomiální problém by ležel v P.

Poznámky na závěr

Otázku „Je třída P rovna NP?“ se již snažilo rozlousknout mnoho matematiků a informatiků. Tato teorie přinesla spoustu zajímavých výsledků, například už se podařilo dokázat, že některými technikami tuto domněnku nelze nikdy dokázat, ani vyvrátit.

Kdyby platilo P = NP, pak by mnoho lidi zajásalo – mnoho přirozených problémů, které nastávají i v reálném životě, by najednou byla řešitelná rychle. Navíc by krachlo dosavadní šifrování a bylo by možné najít rychle důkaz ke každému pravdivému tvrzení výrokové logiky.

Tato rovnost by se dala hypoteticky ukázat velice snadno – stačilo by najít jeden polynomiální algoritmus pro libovolný NP-úplný problém! Většina informatiků studujících složitost se však domnívá, že se třídy nerovnají.

To ale neznamená, že si to nemáte zkusit dokázat! Naopak, bojovat s NP-úplnými problémy je užitečné i v reálném světě – například jde mnohdy vymyslet dobrá aproximace NP-úplného problému.

Například nenajdeme hamiltonovskou kružnici v polynomiálním čase, ale nalezneme nějakou relativně dlouhou kružnici, která nám v praxi může stačit, pokud podle ní třeba chceme vést náročný cyklistický závod.

O aproximacích je toho v literatuře napsáno mnoho zajímavého, pokud byste si o nich chtěli přečíst více v češtině, zkuste třeba výpisky z předmětu na Matfyzu.[27]

O NP-složitosti můžete něco najít na stejné adrese, nebo zkuste vynikající anglicky psanou knížku *Algorithms* od profesorů exotických jmen Dasgupta, Papadimitriou a Vazirani.

Existují i problémy, které jsou mimo P i NP, a dokonce existuje spousta různých dalších tříd problémů. Je jich celá zoologická zahrada – můžete ji najít na internetu.[28]

Seznam NP-úplných problémů

Sedíte-li nad zatím nevyřešenou úlohou, kterou jste nalezli jinde než v KSP, pak se klidně může stát, že bude NP-úplná. Abyste mohli mezi NP-úplnými úlohami převádět, tak je dobré znát jich aspoň hrstku, podle toho, je-li problém grafový, rovnicový, a tak dále.

V následujícím seznamu najdete několik úloh, které jsou zaručeně NP-úplné. Převody se nám sem sice nevešly, ale mnoho z nich (ne-li všechny) zvládnete vymyslet sami – zkuste si to!

Název problému: Hamiltonovská kružnice

Vstup: Neorientovaný graf.

Problém: Existuje v zadaném grafu kružnice procházející všemi vrcholy právě jednou?

Název problému: Hamiltonovská cesta.

Vstup: Neorientovaný graf, dva speciální vrcholy x a y.

Problém: Existuje cesta z x do y (posloupnost vrcholů, ve které se žádné dva neopakují), která prochází každým vrcholem právě jednou?

Název problému: Splnitelnost

Vstup: Logická formule. Tu tvoří proměnné a logické spojky negace $\neg$, konjunkce $\wedge$ a disjunkce $\vee$. Například

$$(x \wedge (\neg y)) \vee z.$$

Problém: Můžeme proměnným přiřadit hodnoty 0 nebo 1 tak, že výsledná vyhodnocená formule má hodnotu 1?

[27] `http://mj.ucw.cz/vyuka/0910/ads2/12-apx.pdf`

[28] `http://qwiki.stanford.edu/index.php/Complexity_Zoo`

Název problému: Součet podmnožiny

Vstup: Seznam nezáporných celých čísel, speciální číslo k.

Problém: Existuje podmnožina čísel, jejíž součet je přesně k?

Název problému: Batoh

Vstup: Seznam dvojic nezáporných čísel, kde dvojice označuje hodnotu a váhu předmětu. Přirozené číslo b – nosnost batohu, přirozené číslo k.

Problém: Umíme vložit do batohu předměty o hodnotě alespoň k, aniž bychom přešli přes limit váhy b?

Název problému: Dva loupežníci

Vstup: Seznam nezáporných celých čísel.

Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Název problému: Klika

Vstup: Neorientovaný graf, číslo k.

Problém: Existuje v grafu úplný podgraf o velikosti k, tedy k vrcholů takových, že mezi každými dvěma z nich vede hrana?

Název problému: Nezávislá množina

Vstup: Neorientovaný graf, číslo k.

Problém: Existuje v grafu prázdný podgraf o velikosti k, tedy k vrcholů, že žádné dva z nich nejsou spojeny hranou?

Název problému: Trojbarevnost grafu

Vstup: Neorientovaný graf.

Problém: Lze vrcholy tohoto grafu obarvit třemi barvami tak, že každá hrana sousedí s vrcholy dvou různých barev?

Název problému: Rozparcelování roviny

Vstup: Seznam bodů v rovině, kde každý má přiřazenu jednu z b barev, číslo k.

Problém: Umíme rozdělit rovinu pomocí k přímek tak, že v každé oblasti jsou jen body té samé barvy?

Název problému: 3D párování

Vstup: Seznam mužů, žen a zvířátek, následovaný seznamem kompatibilních trojic tvaru $\{\text{muž}, \text{žena}, \text{zvířátko}\}$. Tyto trojice říkají, která trojice muž, žena a zvířátko by se dohromady snesla.

Problém: Můžeme všechny muže, ženy a zvířátka z prvního seznamu rozdělit do trojic tak, že každá trojice je kompatibilní a každá bytost je právě v jedné trojici?

Kuchařku sepsal Martin Böhm.

Vzorová řešení

23-1-1 Básníkův deník

Podúloha najít nejvyšší místo, kde spal, splnila svůj účel: vyřešili jste ji všichni. Stačilo si jen počítat nadmořské výšky přičtením toho, co za den ušel, k výsledku včerejšího dne a při tom si v proměnné aktualizovat nejvyšší místo, kam došel. Času to zabere $\mathcal{O}(n)$ a stejně tak paměti (n je počet dnů, po které psal deník).

Nalezení nejčastějšího místa, kde přespal, šlo řešit různými způsoby. Nejoblíbenější byl pomocí třídění.

V setříděných nadmořských výškách už stačí najít tu, která je nejdelší. To jste zvládli při jednom projití. Průběžně si pamatovat, kolik stejných čísel za sebou bylo viděno, a srovnávat to se zatím nejdelším viděným úsekem. Třídění trvá lepšími algoritmy $\mathcal{O}(n \log n)$ a projití $\mathcal{O}(n)$. Celkově tedy $\mathcal{O}(n \log n)$.

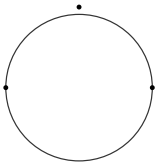
Hodně z vás taky využívalo vyhledávacího stromu, někdy převlečeného za mapu či slovník, ale bylo podstatné si při tom uvědomit, že se o vyhledávací strom jedná. Složitosti při tom zůstávají stejné.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-1-1.c`

Jitka Novotná

23-1-2 Jedna geometrická

Nejprve pár slov k došlým řešením. Mnozí z vás se u této úlohy pokoušeli hledat dva nejvzdálenější body a sestrojit nad nimi Thaletovu kružnici. To však obecně nefunguje, viz obrázek 1. Hledaný střed kruhu dokonce ani nemusí ležet na ose dvou nejvzdálenějších bodů (není tedy pravda, že oba tyto body leží na jeho obvodu) – protipříklad si zkuste vymyslet sami.



Obrázek 1: protipříklad na hledání 2 nejvzdálenějších bodů. Mezi bodem nahoře a oběma body na obvodu je menší vzdálenost než mezi samotnými body na obvodu.

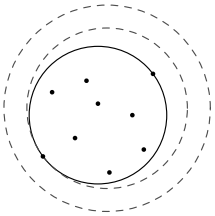
Pouze pár řešení fungovalo a z nich jen jedno pracovalo optimálně. Gratulace putuje k Jakubu Zíkovi, jenž nastudoval lineární algoritmus nezaložený na pravděpodobnosti (těžší na pochopení než zde prezentované řešení pracující lineárně jen v průměru) a pak ho popsal. Jak vidno, ne každá úloha s krátkým zadáním má i krátké a jednoduché řešení.

Jedná se však o problém starý (poprvé se jím zabýval anglický matematik Sylvester v roce 1857) a v praxi využitelný. Vezměte si například firmu, která má po zemi rozmístěné klienty a hledá místo pro své středisko tak, aby k němu žádný klient neměl moc daleko. Říká se mu také „problém bomby“ (chceme zjistit, kde odpálit výbušninu a jak má být velká, abychom zničili všechny cíle).

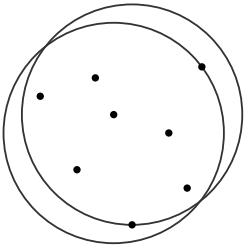
První pozorování

Nyní k tomu, jak se úloha řeší. Nejmenší kruh obsahující všechny body si označíme K. Při řešení úlohy se budou hodit následující pozorování:

- Dostaneme-li na vstupu jen jeden bod, má kruh nulovou velikost, tento případ tedy nebudeme uvažovat.
- Na obvodu kruhu K leží minimálně dva body, jinak ho můžeme zmenšit (viz obrázek 2).

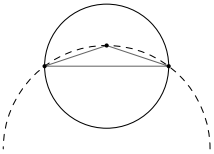


Obrázek 2(vlevo): čárkované kruhy lze zmenšit, protože se nedotýkají dvou bodů.



Obrázek 3(vpravo): nechť jsou dva kruhy obsahující všechny body nejmenší, ale pak jejich průnik, jenž se dá obklopit ještě menším kruhem, obsahuje všechny body.

- Kruh K je pro danou množinu bodů unikátní, tj. neexistují dva nejmenší kruhy obsahující všechny body (pokud by byly dva různé, jejich průnik obsahuje všechny body a zároveň se musí vejít do kruhu s menším poloměrem, takže tyto dva kruhy nejsou nejmenší, viz obrázek 3)
- Na určení kruhu nám stačí maximálně 3 body na jeho obvodu. Pokud na jeho obvodu leží pouze dva body, průměr kruhu je vzdálenost mezi nimi. Jestliže je kruh určen 3 body, musí tvořit ostroúhlý nebo pravoúhlý trojúhelník, jinak by mohl mít kruh průměr rovný vzdálenosti strany proti tupému úhlu a bod u tupého úhlu by neležel na obvodu (viz obrázek 4). Jinak řečeno, jsou-li na obvodu kruhu alespoň 3 body, musí se mezi nimi vyskytovat 3 netvořící tupoúhlý trojúhelník.



Obrázek 4: pokud 3 body tvoří tupoúhlý trojúhelník, není řešením opsat jim kružnici.

Pohled do ZOO algoritmů

Algoritmů řešících takovouto úlohu je více, zde si letmo představíme ty jednodušší a letmo ten „nejhustější," ale kdo chce, ať rovnou přeskočí na sekci randomizovaný algoritmus, kde bude pořádně vysvětleno v průměru lineární řešení.

O kousek výše je v pozorováních zmíněno, že kruh K je určen 2 nebo 3 body. Co prostě vzít všechny dvojice a trojice bodů, opsat jim kružnici, zkontrolovat, jestli v ní leží všechny body, a vybrat nejmenší? To bude určite fungovat, jen časová složitost je nepěkných $\mathcal{O}(N^4)$.

Existuje celkem přímočaré (geometricky myšleno) řešení bežící v čase $\mathcal{O}(N^2)$. Skládá se z následujících kroků:

1. Na začátku vezměte nějaký kruh, který bude určitě obsahovat všechny body (je jedno jaký).

2. Najděte nejvzdálenější bod A od středu kruhu a zmenšete poloměr na vzdálenost mezi A a středem. Kruh se očividně zmenší a stále bude obsahovat všechny body.
3. Pokud na obvodu leží jen bod A, posunujte střed po přímce mezi A a středem směrem k A a zároveň zmenšujte jeho poloměr, aby A stále ležel na obvodu. Pokračujte, dokud se obvod kruhu nedotkne jiného bodu B.
4. Nyní tedy leží na obvodu minimálně 2 body. Dle našich pozorování potřebujeme zjistit, jestli jsou mezi nimi 3 tvořící ostroúhlý trojúhelník. Lze nahlédnout, že takové 3 body neexistují právě tehdy, když na obvodu lze najít část neobsahující body, která je delší než půlka obvodu (podívejte se na poslední obrázek u pozorování). Ta také může být vždy maximálně jedna.
5. Pokud tam taková část není, můžeme skončit. Jinak vezměme dva body na okrajích této části bez bodů (nazveme je D a E), zmenšujme poloměr kruhu a posouvejme střed tak, že D i E jsou stále na jeho obvodu. Mohou nastat dva případy:
   a) Průměr kruhu je vzdálenost mezi D a E: pak jsme nalezli nejmenší kruh obsahující všechny body.
   b) Na obvod kruhu se dostane bod F, máme tedy alespoň 3 body na obvodu a můžeme opět přejít na bod 4 (tj. zkusit najít část bez bodů delší než půlka obvodu a případně opět zmenšovat kruh).

Implementace tohoto geometrického postupu je trochu obtížná. Například zmíněné zmenšování kruhu bude opět hledání jistým způsobem nejvzdálenějšího bodu (přesněji řečeno třeba pro krok 2, pokud jsou dány dva body na obvodu a přímka, po níž se pohybuje střed, je třeba vypočítat, kde bude ležet střed, z toho se získají poloměry a vybere se ten největší).

Kroky 1, 2 a 3 zaberou lineární čas, samotný krok 4 také, ale může se stát až $(N-2)$-krát, že se bude krok 4 opakovat. Proto je časová složitost v nejhorším případě $\mathcal{O}(N^2)$.

Toto řešení bylo objeveno až v roce 1972 pány Elzingou a Hearnem, potom následovaly těsně po sobě nápady na první $\mathcal{O}(N \log N)$ algoritmy (Shamos a Hoey v r. 1975, Preparata v r. 1977 a Shamos v r. 1978).

Velmi zajímavý algoritmus vychází z pozorování, že konvexní obal určuje hledaný nejmenší kruh. V čase $\mathcal{O}(N \log N)$ (resp. $\mathcal{O}(N)$, máme-li body seřazené), najdeme konvexní obal, jeho velikost budiž H, a na něj prostě pustíme kvadratický algoritmus, což dává složitost $\mathcal{O}(N \log N + H^2)$.

Až v roce 1983 vymyslel Nimrod Megiddo k překvapení všech lineární algoritmus založený na metodě prořezávej a hledej (anglicky *prune and search*). Podstatou algoritmu je na základě několika geometrických triků odstranit v lineárním čase $n/16$ bodů bez změny nejmenšího kruhu obsahujícího všechny body.

Na zbylých $15n/16$ bodů je pušten algoritmus znovu a tak dál, dokud nezbyde jen celkem málo bodů (např. 15), pro než lze úlohu rychle vyřešit i kvadratickým algoritmem.

Vtip je v tom, že složitost jednotlivých kroků algoritmu se posčítá díky vlastnostem geometrické řady na lineární složitost, přesněji řečeno: $n + 15n/16 + 225n/256 + \ldots = 16n$. Jelikož úplné vysvětlení by zabralo pěkných pár stránek, raději si přečtěte původní anglický článek. [29]

Randomizovaný algoritmus

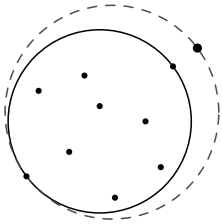
Jak jsme slíbili, teď předvedeme randomizovaný algoritmus (randomizovaný znamená založený na náhodě, v tomto případě náhoda ovlivňuje časovou složitost), běžící v průměru lineárně. Vymyslel ho Welzl v roce 1991. Ten začne s 2 body, jimž opíše kružnici, a poté postupně přidává bod po bodu a upravuje nejmenší kruh K obsahující všechny dosud přidané body, je-li to nutné. Náhodné pořadí přidávaných bodů zajistí onu lineární složitost, jak později ukážeme.

Na začátku je tedy vhodné náhodně uspořádat body v čase $\mathcal{O}(N)$, aby „zlý“ uživatel nezadal pořadí, na němž program poběží pomalu (třeba i body setříděné dle souřadnice x způsobí pomalý průběh, jak se za chvíli ukáže). Tohle můžeme udělat například tak, že vybereme náhodný prvek z pole (tedy vygenerujeme číslo od 1 do N), ten prohodíme s posledním, pak vezmeme náhodný prvek, ale už jen od 1 do $N-1$, prohodíme s předposledním... A takto postupujeme, dokud nedojdeme na začátek.

Nyní přijde trocha geometrických hrátek s body. Začněme tedy prvními dvěma a opišme jim kruh, jež nazveme K_2. Obecně pak K_i bude nejmenší kruh obsahující body $1 \ldots i$. Co dělat, když přidáme i-tý bod a máme kružnici K_{i-1}? Pokud bod náhodou padne do kruhu K_{i-1} (nebo na jeho obvod), pak $K_i = K_{i-1}$, tedy kruh se nezměnil a můžeme pokračovat vesele dál.

Mnohem zajímavější je případ, kdy přidávaný bod leží mimo kruh K_{i-1}. Označme tento bod B_i. Je zřejmé, že B_i musí ležet na obvodu kruhu K_i, jinak by už ležel uvnitř K_{i-1} (neurčuje kruh, můžeme ho tedy vynechat beze změny kruhu). Takže nyní máme za úkol spočítat nejmenší kruh pro $i-1$ bodů s B_i na obvodu. A jak? Zavoláním stejné funkce pro $i-1$ bodů jen navíc s informací, že jistý bod má být na obvodu.

Obrázek 5: bod B_i leží mimo K_{i-1}, takže je třeba zvětšit kruh.



Naším řešením bude funkce, která v parametrech dostane množinu bodů M (ty, pro něž počítá nejmenší kruh) a seznam bodů, jež musí ležet na obvodu (body na obvodu nemusí být v množině M). Funkce nejprve zkontroluje, jestli už na obvodu nemusí ležet 3 body (pak je kruh jednoznačně

[29] `http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/Center/centercli.htm`
N. Megiddo, Linear-Time Algorithms for Linear Programming in $\mathbb{R}^3$ and Related Problems, SIAM Journal on Computing, Vol. 12, 759–776, dostupné na: `http://www-ma2.upc.es/~geoc/m-lalparp-83.pdf`.

určen a dopočítá se) nebo není M prázdná (v tom případě se kruh opíše bodům na obvodu, jsou-li nějaké).

Poté se rekurzivně zavolá s množinou M o jeden bod B menší (to je ten přidávaný bod), uloží si vrácený kruh K a následně zjistí, zdali bod B leží v kruhu K nebo ne. Pokud ano, vrátí kruh K, jinak se rekurzivně zavolá s množinou M o bod B menší a s B na obvodu.

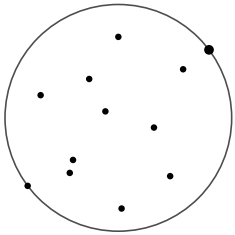
Kdo se v tomto odstavci ztratil, může se najít v následujícím pseudokódu (O je množina bodů na obvodu):

```
function nejmensiKruh(M, O) {
    if (|M| == 0 nebo |O| == 3) {
        Vrať kruh spočtený přímo z množiny O
    }
    Bod B = Vezmi náhodný bod z M
    Kruh K = nejmensiKruh(M - B, O)
    if (B neleží v K) {
        Přidej B do O
        Vrať nejmensiKruh(M - B, O)
    }
}
```

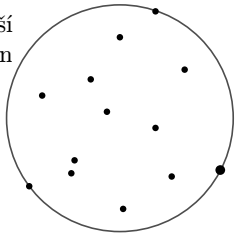
Náhodný výber z množiny, díky němuž už za chvíli získáme průměrnou lineární složitost, zajistíme náhodným seřazením pole. Pak prostě budeme brát poslední prvek.

Tak a nyní k **časové složitosti**. Prostým pohledem na pseudokód by člověk řekl, že bude $\mathcal{O}(N^3)$ (pro každý přidaný bod spustíme rekurzivně tu samou funkci s jedním bodem na obvodu navíc), což je také nejhorší možný případ. Jenže nás teď zajímá průměrná časová složitost, k níž nám dopomůže náhodné seřazení bodů.

První rekurzivní volání `minimalniKruh(M - B, O)` teď budeme tiše ignorovat (ono se totiž provede vždy) a budeme předpokládat, že body postupně přidáváme. Zajímá nás tedy, jaká je pravděpodobnost, že se s novým přidaným bodem B zavolá funkce rekurzivně s B na obvodu navíc. Uvažujme nejmenší kruh K obsahující už bod B. Všimněte si, že rekurzivní volání při přidávání bodu B je podobné zmenšení kruhu K po odebrání bodu B, tedy pravděpodobnost „drahého“ rekurzivního volání je stejná jako pravděpodobnost, že se kruh K po odebrání bodu B zmenší.



Obrázek 6 (vlevo): kruh se zmenší právě tehdy, když odebereme jeden ze dvou konkrétních bodů.



Obrázek 7 (vpravo): To samé pro 3 body na obvodu, jež tvoří ostroúhlý trojúhelník.

Někde vysoko nahoře v tomto textu jsem zmínil, že nejmenší kruh je určen 2 nebo 3 body. Hledaná pravděpodobnost při přidávání i-tého bodu tak vyjde $2/i$ nebo $3/i$ (pro jednoduchost budeme dále uvažovat jen $3/i$, není težké si rozmyslet, že pro $2/i$ vše vyjde stejně).

Dále budeme rozebírat časovou složitost podle počtu bodů, jež musí být na obvodu. Pro 3 je to triviálně $\mathcal{O}(1)$, takže začněme se 2 body na obvodu. Rekurzivní volání algoritmu už nás stojí pouze $\mathcal{O}(1)$ (na obvodu musí být 3 body) a počet bodů na obvodu se nikdy nezmenší, takže N bodů se dvěma danými body na obvodu zvládne algoritmus vždy v $\mathcal{O}(N)$.

Zajímavější je situace, je-li dán jen jeden bod na obvodu. Nyní využijeme před chvílí spočtenou pravděpodobnost (zde $2/i$ a ne $3/i$, protože máme jeden bod předem daný na obvodu), a tak můžeme napsat časovou složitost po přidání i-tého bodu takto:

$$\frac{i-2}{i}\,\mathcal{O}(1) + \frac{2}{i}\,\mathcal{O}(i) = \mathcal{O}(1)$$

Pro N bodů s jedním daným na obvodu se časová složitost posčítá na $\mathcal{O}(N)$. A co N daných bodů a žádný, který by byl určitě na obvodu? To je přeci naše původní úloha. Znovu využijeme pravděpodobnost, takže na přidání i-tého bodu spotřebujeme:

$$\frac{i-3}{i}\,\mathcal{O}(1) + \frac{3}{i}\,\mathcal{O}(i) = \mathcal{O}(1)$$

Při součtu ještě přidáme počáteční náhodné zamíchání pole bodů:

$$\mathcal{O}(N) + \sum_{i=1}^{N} \mathcal{O}(1) = \mathcal{O}(N).$$

Sláva! Tak máme dokázánu i časovou složitost. Paměťová je zjevně vždy lineární.

A poučení do příště? U některých úloh, jestliže si s nimi lámete hlavu už docela dlouho, se vyplatí zeptat se vyhledávače, zdali nezná řešení, jež pak případně přečtete, pochopíte a popíšete vlastními slovy.

Program (C++):
`http://ksp.mff.cuni.cz/viz/23-1-2.cpp`

Pavel Veselý

23-1-3 Jedna maticová

K této úloze nám došla spousta řešení, téměř každé fungovalo, ale problém byl ve složitosti. Některá řešení byla příliš pomalá, u jiných byl problém se špatně určenou složitostí. Nezapomínejte, že pro dobré hodnocení je potřeba mít správný a srozumitelný popis vašeho algoritmu a také správnou časovou a prostorovou složitost.

První řešení spočívá v prohledání celé matice řádek po řádku a kontrole každého prvku. Takové řešení samozřejmě funguje, dokonce funguje i pro obecné matice. A to je právě kámen úrazu.

Protože toto řešení nevyužívá vlastností matice, musí se podívat na každý prvek. Jeho složitost je tedy $\mathcal{O}(n \cdot m)$ pro matici velikosti $n \times m$. To ani zdaleka není to, co bychom chtěli a za co bychom byli ochotni dát celých 11 bodů.

Někteří si uvědomili, že když je posloupnost čísel v řádku ostře rostoucí, dalo by se využít binární vyhledávání. A tak jde zlepšit složitost z $\mathcal{O}(n \cdot m)$ na $\mathcal{O}(n \log m)$. Ale věřte tomu nebo ne, ani to nám nestačí.

Když nestačí použít na každém řádku binární vyhledávání, co ještě provést? Správné řešení používá binární vyhledávání na hlavní diagonále matice (tak se říká úhlopříčce vedoucí doprava dolů). Před uvedením algoritmu si musíme uvědomit, že platí dvě důležité věci:

- Pokud je v matici A na indexech i, j (označíme jako $A_{i,j}$) prvek, jehož hodnota je menší než $i + j$ ($A_{i,j} < i + j$), víme z uspořádání prvků v řádcích a sloupcích, že jsou menší i všechny prvky v matici, jejichž souřadnice jsou menší než i a j ($\forall k \leq i, l \leq j : A_{k,l} < k + l$).
  $A_{i,j}$ je alespoň o jedna menší než $i + j$, tedy i např. $A_{i-1,j}$ musí být alespoň o jedna menší než $A_{i,j}$, což znamená, že je alespoň o jedna menší než $i - 1 + j$. A takto tranzitivně dále.
- Pokud platí $A_{i,j} > i + j$, pak $\forall k \geq i, l \geq j : A_{k,l} > k + l$. Opět platí obdobně, $A_{i,j}$ je alespoň o jedna větší než $i + j$, takže i všechny následující prvky musí být alespoň o jedna vychýleny.

Z těchto dvou pozorování plyne, že pokud se podíváme na prvek uprostřed matice, tak mohou nastat tři možnosti. Mohli jsme narazit na správný prvek. To znamená, že můžeme skončit. Nebo je nalezený prvek větší než součet jeho souřadnic, pak můžeme zapomenout pravou dolní čtvrtinu matice, případně je prvek menší a zapomeneme levou horní čtvrtinu matice.

Takže budeme provádět binární vyhledávání na hlavní diagonále, buď najdeme správné řešení, nebo nám nakonec zůstane jen pravá horní a levá dolní čtvrtina matice. Na ty zavoláme rekurzivně stejný algoritmus. Právě tento způsob je použit ve vzorovém kódu.

Toto řešení nám přišlo několikrát, ovšem pouze jednou u něj byla uvedena správná časová složitost. Pojďme si ji tedy rozebrat detailně. Čas potřebný pro nalezení řešení je definován rekurzivně: $T(n^2) = 2T(n^2/4) + \log_2 n$ (pro jednoduchost předpokládáme čtvercovou matici).

Každý správný programátor je hlavně hrozný lenoch, využijeme tedy kuchařkovou metodu pro počítání složitosti rekurzivních algoritmů. Ta se jmenuje Master Theorem a řeší rekurzivní vztahy ve tvaru $T(N) = aT(N/b)+f(N)$, kde $a \geq 1, b > 1$. Dále tvrdí, že pokud $f(N) = \mathcal{O}(N^{\log_b(a)-\varepsilon})$ pro nějaké $\varepsilon > 0$, tak $T(N) = \Theta(N^{\log_b a})$.

Pro naši rekurenci tohle všechno platí:

$$a = 2, b = 4, \log_2 n = \mathcal{O}(n^{2\log_4 2-\varepsilon}),$$

takže výsledná složitost je $\Theta(n)$. Prostorová složitost je logaritmická, protože používáme zásobník.

Existuje i jednodušší řešení, které také vede k cíli. Pro něj si stačí uvědomit, že pokud se podíváme na prvek v levém dolním rohu, tak buď jsme našli správné řešení, nebo je větší než součet souřadnic, pak můžeme zahodit celý poslední řádek, nebo je menší než součet souřadnic a můžeme zahodit celý první sloupec. Nakonec se posuneme buď nahoru nebo doprava, podle toho, čeho jsme se zbavili, a pokračujeme stejně.

Takto se v každém kroku zbavíme buď celého sloupce, nebo řádku. V nejhorším případě tedy provedeme $\mathcal{O}(n + m)$ operací. Prostorová složitost je zde konstantní.

Pokud bychom chtěli najít všechny prvky matice, které odpovídají zadání, tak je snadné uvedené dva algoritmy upravit, víme totiž, že pokud najdeme jedno řešení, budou s ním další sousedit, nebo budou v zatím neprozkoumané části matice.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-1-3.c` *David Marek & Karel Tesař*

Vida, to je zvláštní druh algoritmu – rychlejší než lineární ve velikosti vstupu (ta je $m \times n$), protože si ani celý vstup nemusí přečíst. Také vám vrtá hlavou, jestli by nestačilo si ze vstupu přečíst ještě méně? Pojďme dokázat, že nestačilo.

Nejdřív si úlohu převedeme na jinou, ekvivalentní, aby se nám o ní snáze přemýšlelo. Místo zadané matice budeme uvažovat stejně velkou matici $B_{i,j} = A_{i,j} - i - j$. Jelikož A byla v řádcích i sloupcích rostoucí, B bude alespoň neklesající (rozmyslete si, proč). A hledané políčko $A_{i,j} = i + j$ odpovídá políčku $B_{i,j} = 0$. Pokud tedy umíme vyřešit původní úlohu, dokážeme vyřešit i tuto, a naopak.

Nyní uvažujme matici B, která bude mít na hlavní diagonále a nad ní hodnoty $+1$ a pod diagonálou samé -1. To je neklesající matice, v níž žádné nulové políčko neexistuje. Kdykoliv ale změníme některou z $+1$ na diagonále na 0, matice bude pořád neklesající, ale řešení v ní už bude existovat:

$$\begin{pmatrix} +1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & +1 & +1 \\ -1 & -1 & +1 & +1 & +1 \\ -1 & -1 & -1 & 0 & +1 \\ -1 & -1 & -1 & -1 & +1 \end{pmatrix}$$

Pokud tedy libovolný algoritmus řešící úlohu spustíme na naši matici B, musí přečíst alespoň všechna políčka na diagonále, aby si ověřil, že v matici žádné nulové políčko není.

Martin Mareš

23-1-4 Ale co trapné numerické chyby?

Na této úloze nebylo mnoho těžkého, a tak spousta řešitelů dostala zasloužených 10 bodů. Blahopřejeme, příště už to tak snadné nebude!

Přejděme k úloze samotné. Periodu racionálního čísla nelze poznat jen tak, že se v desetinném zápisu opakuje řetězec (začátek 0.88 neznamená periodu 8, například u 15/17). Když dělíme čitatel a jmenovatel na papíře, poznáme periodu tak, že se „zacyklíme“ – dělíme už jednou to samé číslo, ten samý zbytek. Tak proč to tak neimplementovat? Zbytky po dělení jmenovatele nám budou sloužit jako odkazy do pole, uvnitř pole si zapamatujeme první výskyt odkazovaného zbytku – abychom věděli, kde zapsat závorku. Hotovo!

Poznamenejme, že paměťová složitost je $\mathcal{O}(N)$, kde N je velikost jmenovatele, tedy počet možných zbytků, a časová je lineární vůči velikosti výstupu. (V některých případech je pro dokázání optimality užitečnější měřit časovou složitost nikoli podle vstupu, ale podle velikosti výstupu. Nakonec, i kdybychom uměli dělit rychleji než na papíře, stejně musíme výstup vypsat.)

Program (C):
`http://ksp.mff.cuni.cz/viz/23-1-4.c`

Martin Böhm & CodEx

23-1-5 Adina knihovna

Očíslujme si N knih po řadě zleva doprava 1 až N. Podívejme se na knihu s číslem 1, která je jistě na kraji. Lze ji přesunout na jediné místo, a to na pozici $1 + K$. Dalším pohledem zjistíme, že knihu z pozice $1 + K$ musíme přesunout na pozici 1, protože jinou tam dát nesmíme.

Podívejme se na knihu s číslem $č \leq K$. Lze ji přesunout na jediné místo, a to na pozici $č + K$, odkud přesuneme knihu na pozici $č$.

Tedy prvních $2K$ knih povyměňujeme mezi sebou a zbyde nám $N - 2K$ knih, na které můžu použít stejný argument. Tohle opakujeme i kroků, až nám zbyde $0 \leq N - 2iK < 2K$. Buďto platí, že $N - 2iK = 0$, pak jsme hotovi (a tedy platí, že $2K$ dělí N, protože $N/2K = i \in \mathbb{N}$). Nebo máme nenulový zbytek, ale v tom jistě umíme najít knihu, kterou neumíme přesunout ani vlevo, ani vpravo (třeba tu úplně uprostřed), tedy knihovnu s takovým K nelze přeskládat.

Zbývá tedy první varianta, a tedy bereme pouze taková K, která dělí $N/2$ (pro lichá N úloha nemá řešení). Zjevně je jen jeden způsob, jak knihy přeskládat, což byla druhá věc, na kterou se zadání ptalo.

Někteří řešitelé ještě uvažovali triviální případ, kdy $K = 0$, to funguje pro všechna N (i lichá). Někdo také řešil možnost $K < 0$. To bylo možné, i když jsme to nijak extra nehodnotili.

Jan „Moskyto“ Matějka & Pali Rohár

23-1-6 Babbageova cesta

Píšeme-li v zadání „*Pro jednoduchost* předpokládejme, že použití takového spojení trvá jednotkový čas“ můžeme tím myslet různé věci. Takové omezení zjednodušuje popisování zadání, zjednodušuje načítání vstupu...

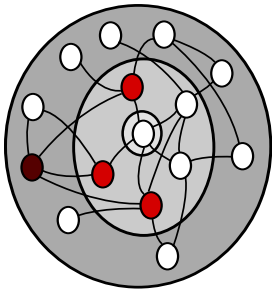
Může se stát, že bychom zjednodušovali práci procesoru, že by asymptotická časová složitost řešení s takto omezeným zadáním byla nižší, než složitost řešení, kde by použití spojení zabralo zadaně vteřin?

Je to tak a většinu z vás jsme na to nachytali. Použití Dijkstrova algoritmu je v daném případě triviálně možné: stačí měřit vzdálenost v uspořádané dvojici (počet použitých hran, součet cen na použitých hranách), kde při porovnávání klademe důraz na druhé složky pouze v případě rovnosti prvních složek.

Do časové složitosti takového řešení se však nevyhnutelně vloudí logaritmy, které tam zaneslo použití haldy coby rozumné implementace prioritní fronty, kterou Dijkstrův algoritmus prostě potřebuje.

Poodstoupíme o krok zpátky: dokud jsme nevěděli, co to Dijkstrův algoritmus je, uměli jsme měřit nejkratší cesty pouze co se počtu hran týče, a to prohledáváním do šířky.

To nám přirozeně rozdělí vrcholy do vrstev podle vzdálenosti od vrcholu, ze kterého jsme prohledávat začali, stačí si k vrcholu tuto vzdálenost připsat (výchozímu vrcholu nastavit nulu) a při vkládání nezpracovaných vrcholů do fronty jim ji přidělovat o jednotku zvýšenou.



Naše úloha je složitější o to, že druhotné kritérium v zadání mluví o ohodnocení hran. S tím se ale vyrovnáme snadno lehkou úpravou prohledávání do šířky: kdykoliv dostaneme z fronty vrchol v s přiřazenou hranovou vzdáleností n, rozhlédneme se po sousedních vrcholech (tj. těch, se kterými v spojuje hrana), vybereme jen ty, které jsou ve vrstvě vzdálené $n-1$ (od výchozího bodu), a vrcholu v nastavíme coby minimální cenu minimum ze součtu cen vrcholů z této vrstvy a příslušných cen přepravy (ohodnocení hran) z těchto vrcholů do našeho v. A samozřejmě, abychom mohli posléze zrekonstruovat cestu, si uložíme, který že to vrchol z vrstvy vzdálené $n-1$ byl pro náš v takto výhodný.

Bude to fungovat? Do daného vrcholu prostě musíme přijít z vrcholu ve vrstvě vzdálené nejvýše $n-1$, chceme-li dodržovat hranovou vzdálenost coby úhlavní kritérium. A z vrstvy s menším pořadovým číslem nám do vrcholu ve vrstvě n samozřejmě žádná hrana vést nemůže. Pokud tedy věříme, že máme ceny v nižších vrstvách spočítány správně, budeme je mít dobře i ve spočítaném vrcholu. No a protože cena v počátečním vrcholu je dobře (0), roznese nám matematická indukce tuto správnost po všech vrcholech v grafu.

Samozřejmě nepotřebujeme všechny cesty, ale to už je ta potíž s algoritmy pro hledání nejkratší cesty z bodu A do bodu B, že toho většinou musí mimoděk spočítat o hodně víc. Časová složitost našeho řešení je každopádně $\mathcal{O}(n + m)$ a paměťová stejně tak.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-1-6.c`

Lukáš Lánský

23-1-7 Regulární výrazy

Sešlo se nám přes 30 řešení různé kvality a přístupu. Bylo nelehkým úkolem je opravit a alespoň pseudospravedlivě obodovat, takže pokud vám bude připadat, že jsme zrovna k vám byli nespravedliví, tak se ozvěte e-mailem opravujícím, nebo třeba na fóru. Prostoru pro dotazy je dost, ty nejvíce očekávané se zde pokusíme zodpovědět rovnou.

Autorským řešením **úkolu 1** byl výraz `((b?a)*b)?` – ten přijímá opravdu stejné řetězce jako zadaný `b?(a+b)*` – za každým `b` musí nutně následovat alespoň jedno `a`, pokud tedy není na konci řetězce. Musí vyhovovat i prázdný řetězec, což bylo často opomínáno.

Řešení spočívající v náhradě `a+` za `aa*` nebo `b?` za `b{0,1}` jsme hodnotili stylově desetinou bodu. On je to totiž vlastně stejný výraz.

V řešení **úkolu 2** jste se mohli odvážit dál. Mnoho z vás zůstalo u výrazu `(a+|b+)*`, který šlo po krátkém rozmyslu zredukovat na `[ab]*`, což je také autorské řešení.

Úkol 3 byl poněkud šílený. Na něm jste si mohli vyzkoušet tvorbu rozsáhlých regexů, na kterých je poznat každá nesystematičnost, každá výjimka. Zde jsme strhávali body i za používání `(0|2|4|6|8)` místo `[02468]`. Ono to má stejný význam, akorát to první se čte výrazně hůř.

Mnoho řešitelů jednoduše vypsalo všechna koncová trojčíslí dělitelná 8. To je sice hezké, ale pomalé. Každý znak navíc je zpomalení. Porovnejte s autorským řešením (mezery a konce řádků ignorujte):

```
(0|-?(8|[48][08]|[159]6|[26]4|[37]2|[2468]([048][08]|[159]6|
 [26]4|[37]2)| ([1-9][0-9]*)?[13579]([048]4|[159]2|[26][08]|
 [37]6)|[1-9][0-9]*[02468]([048][08]|[159]6|[26]4|[37]2)))
```

Nelíbila se nám čísla, která začínala řadou nul, stejně tak drobné chyby jako neuvažování nuly nebo záporných čísel, nicméně jsme za ně strhávali výrazně méně než za false positives nebo false negatives.

Následovalo cvičení z exaktního vyjadřování. V **úkolu 4** bylo za úkol popsat, co danému výrazu vyhovuje. Obyčejný popis stylu „sudý počet nul, pak nula, nebo jednička, a potom sudý počet jedniček“ vyfasoval bod. Nápaditější řešitelé, kteří napsali „sudý počet nul, pak lichý počet jedniček, nebo lichý počet nul a pak sudý počet jedniček“ získali body dva. Hodí se zmínit, že nula je také sudé číslo. Mnoho z vás si to neuvědomilo a řešili nulu zvlášť.

Nakonec trochu přiblížení reality. **Úkol 5** vyžadoval opravu zadaného výrazu, což je nejčastější problém, se kterým se při práci s regexy setkáte. Zadanému regexu měly vyhovovat právě ty řetězce, ve kterých je sudý počet jedniček, sudý počet nul a nic jiného.

Zadaný výraz byl dost mimo. Jedna z možností, jak ho opravit, spočívala ve zhruba dvojnásobném natažení výrazu, neboť kromě bloku `0(00|11)*1(00|11)*` bylo potřeba ještě zahrnout blok `1(00|11)*0(00|11)*`. Lepší variantou bylo zadaný výraz zahodit a vymyslet úplně nový.

Má-li řetězec sestávat ze sudého počtu nul a sudého počtu jedniček, pak musí mít také celkem sudý počet znaků, tedy nám rozhodně nebude vadit, že jej budeme kontrolovat po dvojicích.

Prázdný řetězec rozhodně vyhovuje. Pokud nyní přečteme dvojici `(00|11)`, bude rozhodně vyhovovat taky. Naopak kdybychom měli řetězec, který nevyhovuje, tak po přečtení dvojice `(00|11)` vyhovovat také nebude. Tedy nás nezajímá, kdy, kde a v kolika exemplářích se nějaká tato dvojice objeví.

Přesně obráceně to platí pro dvojici `(01|10)`. Ta vždy přepne mezi vyhovujícím a nevyhovujícím řetězcem. Té tedy potřebujeme sudý počet. Po poskládání všech požadavků máme výraz `(00|11)*(((01|10)(00|11)*){2})*`

První část spolkne začátek sestávající z `(00|11)*`, další část vždycky přejde do stavu „nevyhovující řetězec“, spolkne `(00|11)*`, přejde do stavu „vyhovující řetězec“ a zase spolkne `(00|11)*`. Jednoduché a účinné.

Jozef Gandžala & Jan „Moskyto“ Matějka

23-2-1 Balíčky balíčků

Naše úloha se docela podobá problému batohu (viz kuchařka);[30] takže by nás mohlo napadnout použít modifikovanou verzi algoritmu, kterým se řeší.

Postupně procházíme celá čísla od nuly vzhůru a pokud jsme právě na hodnotě, kam se umíme dostat, tak projdeme všechny nabídky a pro každou z nich si poznačíme, že se umíme dostat na hodnotu, která je součtem této nabídky a hodnoty, na které právě jsme. Na začátku víme jenom to, že se umíme dostat do čísla nula. Takhle postupujeme, dokud se nedostaneme do čísla, které je větší nebo rovno H, a máme řešení.

Tenhle postup sice funguje, ale dosti pomalu. K rychlejšímu algoritmu dojdeme, když si uvědomíme, co to znamená, že každou nabídku můžeme použít, kolikrát chceme – to, že kdykoliv umíme poslat x kg, tak umíme poslat i $x + kN$ kg pro jakékoliv nezáporné celé číslo k (N kg je totiž hmotnost nejmenší nabídky).

Díky tomu si můžeme pole hmotností přeuspořádat do tabulky o N sloupcích. Políčko na i-tém řádku j-tého sloupce pak představuje $(i \cdot N + j)$ kg.

K vyplňování této tabulky bychom mohli použít stejný postup jako před chvílí, ale my si ho upravíme tak, že když jsme na nějakém políčku a umíme se dostat do nějakého políčka nad ním (číslo sloupce je stejné, číslo řádku menší), tak si poznačíme, že se umíme dostat i do aktuálního políčka, ale už nemusíme zjišťovat, kam se odsud můžeme dostat s použitím různých nabídek.

To proto, že pokud se na nějaké políčko umíme dostat z aktuálního použitím nabídky x kg, tak se tam umíme dostat i ze zmíněného políčka nad ním. A to nejdříve použitím nabídky x kg a následně několikanásobným použitím nabídky N kg.

Tuto tabulku si ale nemusíme pamatovat celou. Stačí si pro každý sloupec pamatovat, který je první řádek v tomto sloupci, na který se umíme dostat.

Tento seznam sloupců pak procházíme dokola podobně, jako jsme předtím procházeli celou tabulku – jeden průchod seznamem odpovídá průchodu jedním řádkem v tabulce.

Navíc ani nemusíme procházet seznamem slouců tolikrát, kolik řádků bychom prošli v tabulce. Jakmile se jednou umíme dostat do sloupce, který obsahuje cílové políčko, tak víme, že se umíme dostat až tam.

Pro určení výsledné kombinace balíčků si musíme pro každý sloupec zapamatovat, s použitím jakého balíčku jsme se tam dostali.

Samotnou výslednou kombinaci určíme tak, že nejdříve započítáme nabídku N kg tolikrát, kolik řádků by činil rozdíl v tabulce mezi cílovým políčkem a políčkem, kam se umíme dostat. Následně procházíme sloupce podle toho, pomocí kterého balíčku jsme se do něj dostali, dokud se nedostaneme do nultého sloupce. Všechny balíčky, které jsme na této cestě použili, započítáme také a máme kýžený výsledek.

[30] `http://ksp.mff.cuni.cz/tasks/21/cook5.html`

Jakou má tento algoritmus složitost? Paměťová je $\mathcal{O}(N)$ – nejvíce zabírá seznam sloupců a těch je N.

S časovou složitostí je to složitější. Procházení nabídek provádíme nejvýše jedenkrát pro každý sloupec, což nám dává $\mathcal{O}(N^2)$. Protože se ale může stát, že budeme procházet seznamem opakovaně, dokud se neumíme dostat do všech sloupců, potřebujeme zjistit, kolikrát nejvýše to uděláme.

Stačí se podívat na jedinou nabídku: $2 \cdot (N-1)$. Pokud budeme používat jenom tuto nabídku, tak se v případě lichého N po N krocích dostaneme do každého sloupce. Došli jsme tedy až do čísla $N \cdot 2 \cdot (N-1)$, a počet průchodů seznamem je tedy $2 \cdot (N-1) = \mathcal{O}(N)$.

V případě sudého N se do lichých sloupců nedá dostat žádným způsobem a použitím stejné nabídky jako v předchozím případě se po $N/2$ krocích dostaneme do všech dostupných sloupců. Prošli jsme tedy seznamem opět $\mathcal{O}(N)$-krát.

V obou případech tedy musíme projít v nejhorším případě $\mathcal{O}(N^2)$ políček. Zpětný průchod pro zjištění výsledku projde každým sloupcem nejvýše jednou a složitost nám tedy nezhorší. Celková časová složitost tedy je $\mathcal{O}(N^2 + N^2 + N) = \mathcal{O}(N^2)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-2-1.c`

Petr Onderka & CodEx

23-2-2 Zastavení

Zkusme nejprve generovat čísla 1 až 120. Hodíme jednou šestistěnkou a jednou dvacetistěnkou, máme tedy 6 možností, jak dopadne hod první kostkou a 20 možností, jak dopadne hod druhou kostkou. Celkem tedy máme 120 různých možností a pro každou možnost odpovíme jiným číslem.

Kdybychom chtěli generovat čísla od 1 do 50, hodíme dvakrát desetistěnkou a dostaneme 100 různých možných výsledků ($(3,1)$ a $(1,3)$ jsou rozdílné výsledky). Všechny výsledky jsou stejně pravděpodobné, každá kostka je dokonale náhodná a jednotlivé hody se neovlivňují.

Pokud tedy program odpoví jedničkou pro první dva výsledky (pro libovolné uspořádání), dvojkou pro další dva atd., umí správně generovat požadovaná čísla, protože generuje každé se stejnou pravděpodobností a potřebuje konečný počet hodů.

Nyní obecnější případ, chceme generovat N čísel a N dělí nějaký násobek počtů stěn našich kostek P – to je ve skutečnosti počet možných výsledků, které můžou nastat po hodech těmito kostkami. Rozdělíme všechny možné výsledky na N (disjunktních) částí o P/N prvcích a použijeme předchozí postup.

Zbývá ukázat, že pro jiná N nedokážeme na zaručeně konečný počet hodů vždy vygenerovat správný výsledek. Nejmenší N takové, že nedělí žádné možné P, je 7. V prvočíselném rozkladu žádného počtu stěn našich kostek totiž není 7.

Napřed rozeberme špatné postupy. Zahození některých výsledků – hodím osmistěnkou, pokud padne 8, hodím znova. Takovému algoritmu by mohla padat pořád 8 a nezastavil by se, leda by nám stačil průměrně konečný počet hodů, viz úloha 16-1-5.[31]

Když nemůžeme dostat vhodné P násobením, zkusíme sčítat – sečtu dvě padlá čísla po hodu čtyřstěnkou a od toho odečtu 1. Tento postup ale nedává stejné pravděpodobnosti všech čísel.

Jednička může vzniknout jen poté, co padne $(1,1)$, ale trojka může vzniknout po pádu $(1,3)$, $(2,2)$ nebo $(3,1)$, takže trojkou by algoritmus odpověděl s třikrát větší pravděpodobností. Některým číslem odpovím i jindy – pokud padne 8, odpovím jedničkou, ale toto triviálně nedává stejnou pravděpodobnost všem číslům.

Jak tedy dokázat, že žádný algoritmus si nemůže vystačit s konečným počtem hodů?

Pro spor budeme předpokládat, že existuje nějaký algoritmus, který správně generuje pro N, která nedělí žádné možné P. Po nějakém konečném počtu hodů program proběhne jedním z P různých způsobů (všechny jsou stejně pravděpodobné) a na konci každého odpoví nějakým z N požadovaných čísel.

Kdyby ale všechny odpovědi měly stejnou pravděpodobnost, znamenalo by to, že jsme dokázali P celočíselně a bez zbytku vydělit číslem N, což je spor s předpokladem.

Umíme tedy generovat jen pro taková N, která dělí nějaké P.

Martin Böhm & Karel Král

23-2-3 Projížďka

Trocha magie

Milý čtenář mi jistě pro jednou odpustí, pokud si zahraji na kouzelníka a vytáhnu jednoho králíka z klobouku.

Napřed, zadání šlo chápat různými způsoby, avšak příliš neměnilo podstatu řešení. Předpokládejme tedy například, že všechny cesty jsou jednosměrky a že „z rozcestí vychází sudý počet cest“ znamená, že právě polovina tohoto sudého počtu je v příchozím a právě polovina v odchozím směru.

Opravdu nám stačí taková podmínka pro orientovaný graf. V neorientovaném jsme potřebovali sudý počet, protože kdykoliv jsme vešli do vrcholu, také z něj někudy musíme odejít. Stejně to funguje pro orientovaný, jen musíme přijít po vstupní hraně a odejít po výstupní. Že jde o podmínku postačující, lze nahlédnout také zcela stejně jako v neorientovaném grafu. Jediné, na co si musíme dát pozor, je, že při vypisování dostáváme hrany pozpátku.

Na grafu na vstupu (rozcestí jsou vrcholy a cesty jsou hrany) si najdeme uzavřený eulerovský tah (to již za nás vyřešila kuchařka). Nyní jej projdeme a budeme si udržovat průběžný součet prošlých hran (říkejme tomu součtu odpočatost). Rozeberme dva případy.

[31] `http://ksp.mff.cuni.cz/viz/16-1-5`

Jako první případ vezmeme situaci, kdy po projití celého tahu dostaneme záporné číslo. Potom je součet všech hran záporný a takový zůstane, ať je vezmeme v libovolném pořadí. Proto úloha nemá řešení.

Pokud průšvih popsaný v minulém případu nenastane, vezmeme místo v tahu, kde se nachází minimum ze všech odpočatostí (místem v tahu není myšlen jen vrchol, ale i který průchod tímto vrcholem máme na mysli, neboť při různých průchodech můžeme mít různé hodnoty odpočatosti). V tomto místě v tahu začneme (jakoby jej pootočíme).

Složitost

Máme hezké lineární řešení (jak pamětí, tak časem), neboť již kuchařka nám ukázala, že eulerovský tah v dané složitosti zvládneme najít, a přidali jsme jen dva průchody vzniklým cyklem (jeden na průběžné počítání, druhý na výpis „pootočené“ verze).

Proč to funguje

Nyní už jen zbývá zdůvodnit, proč tento algoritmus vlastně počítá, co má. První případ je nezajímavý (neboť jsme jej již zdůvodnili výše). Dále tedy předpokládejme, že nám nastal druhý případ. Protože máme uzavřený eulerovský tah, projedeme každou cestou právě jednou. Zbývá dokázat, že odpočatost v pootočeném tahu nikde neklesne do záporných čísel.

Předpokládejme tedy, že v místě $š$ na tahu máme zápornou odpočatost. Minimum máme v místě m. Pokud by v původním neotočeném tahu bylo $š$ až za m, pak by muselo být také s menším číslem než m a m by tedy nebylo minimum. Tento případ tedy nenastal.

Takže $š$ je před m. Představme si, že jsme prošli tahem dvakrát místo jednou, tedy při druhém průchodu $š$ jsme na nižším čísle, než při prvním průchodu m (proto nám po pootočení v $š$ vyšlo něco záporného). Ale protože druhý průchod nezačíná od nuly, ale od něčeho nezáporného, odpočatost druhého průchodu $š$ je alespoň tak velká, jako první. Tedy i při prvním průchodu $š$ jsme měli nižší číslo než u m, což je opět ve sporu s výběrem minima.

Jak na to přijít

Jednak, kdyby na to bylo jednoduché přijít, nebyla by úloha za 12 bodů. Ale přesto si řekneme způsob, jak na to přijít.

Můžeme si představit, že jsme řešení již našli a koukat na jeho vlastnosti. To, že je to uzavřený eulerovský tah, je vidět celkem jednoduše. Dále si všimneme, že vybráním jiného začátku se nám všechna čísla posouvají jen nahoru a dolů, rozdíly zůstávají stejné (s výjimkou rozpojeného konce – začátku). No a dále víme, že nejmenší číslo je 0 a to je na počátku.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-2-3.c`

Michal „Vorner“ Vaner

23-2-4 Plánování

Budeme hladově přiřazovat letadla událostem tak, jak nám ze vstupu (seřazené dle počátků) přijdou pod ruku.

Podrobněji řečeno: v každé chvíli běhu programu si budeme udržovat hypotézu „stačí nám L letadel“, kde L navýšíme jen tehdy, ukáže-li se býti flagrantně špatná tím, že nebudeme mít při zpracování počátku události žádné volné letadlo.

Pokud volné letadlo mít budeme, prostě ho dané události přidělíme – k uchovávání volných letadel můžeme mít zásobník, do kterého budeme házet jejich pořadová čísla. Nebo frontu, pokud toužíme vytvářet zdání spravedlivého rozvrhování vůči pilotům. (Rozmyslete si.)

Tímto jistě dojdeme ke správnému minimu počtu letadel, protože pokud nám po poslední události zbyla hypotéza „stačí nám L letadel“, jistě se někdy stalo, že $L - 1$ letadel nedokázalo pokrýt probíhající události. Zároveň je jasné, že tak umíme vytvořit správné rozvrhy, protože jsme si celou situaci de facto odsimulovali.

Z tohoto popisu to vypadá, že nám stačí lineární čas, ale celá věc má jeden háček: potřebujeme zpracovávat konce událostí (uvolňovat letadla), ale kdy?

Musí to být před dalšími odlety, abychom zbytečně nezvýšili L, musí to být po předchozích odletech, abychom nenabyli zdání, že letadel potřebujeme méně – asi nám nezbyde nic jiného, než tyto konce v $\mathcal{O}(N \log N)$ zatřídit do vstupní posloupnosti, jejíž počáteční setřízení podle počátků nám nakonec z hlediska časové složitosti k ničemu nebylo.

Paměťová složitost je samozřejmě lineární.

Úloha souvisí se specifickými grafy, kterým se říká intervalové, ale jejich přímé použití by program nevyhnutelně zpomalilo a vzhledem k jednoduchosti algoritmu by nám nijak nepomohly ani ve výše provedené úvaze.

Program (Python):
`http://ksp.mff.cuni.cz/viz/23-2-4.py`

Lukáš Lánský

23-2-5 Zaměřování

Obsah trojúhelníku lze spočíst jako $S = av_a/2$, kde a je základna a v_a odpovídající výška. Délka základny je známa, a tedy pokud existuje bod, který spolu s kanóny tvoří trojúhelník s obsahem právě S, bude ležet na průsečíku mnohoúhelníku a rovnoběžek spojnice kanónů ve vzdálenosti v_a od nich.

V programu si můžete všimnout, že jsou ignorovány hrany rovnoběžné se spojnicí kanónů. To si můžeme dovolit, neboť u nich záleží jen na okrajových bodech a ty se uváží nejpozději v kroku, kdy nenalezneme průsečík.

Pokud takový průsečík nenalezneme, tak bod, který by spolu s kanóny tvořil trojúhelník s obsahem právě S, neexistuje. Potom je třeba hledat bod, který spolu s kanóny vytvářel trojúhelník s obsahem co nejbližším k zadání. Jeden z takových bodů bude určitě vrchol mnohoúhelníku.

To se snadno nahlédne sporem. Pokud by existoval takový bod X uprostřed hrany AB, tak mohou nastat dvě možnosti. Buď je hrana AB rovnoběžná se spojnicí kanónů (a pak je jedno, který bod z této hrany uvážíme – všechny budou vytvářet trojúhelník se stejným obsahem), nebo není rovnoběžná, a pak pokud s X hneme, tak na jednu stranu bude obsah trojúhelníku růst, na druhou klesat. A jelikož víme, že trojúhelník s obsahem přesně S neexistuje, tímto posunutím jsme našli bod s menším rozdílem obsahů a máme spor.

Projdeme tedy všechny vrcholy mnohoúhelníku a najdeme ten, který odpovídá úloze.

Časová složitost je $\Theta(N)$ – seznam vrcholů projdeme právě $2\times$ – a paměťová také $\Theta(N)$ – někde musí být uložen vstup. Pokud bychom uvážili, že vstup nám bude někdo zadávat postupně, dal by se program upravit, aby potřeboval jen další $\Theta(1)$ paměti.

Následuje pár poznámek k užité analytické geometrii, kterou jsme hojně využívali ve zdrojovém kódu.

Skalární součin vektorů $\vec{a}$ a $\vec{b}$ se určí jako

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y.$$

Platí pro něj $\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos\theta$, kde θ je úhel, který spolu vektory $\vec{a}$ a $\vec{b}$ svírají a $|\vec{a}|$, resp. $|\vec{b}|$ jsou velikosti vektorů $\vec{a}$ a $\vec{b}$. Speciálně platí $\sqrt{\vec{a} \cdot \vec{a}} = |\vec{a}|$ a $\vec{a} \cdot \vec{b} = 0$, pokud jsou na sebe vektory $\vec{a}$ a $\vec{b}$ kolmé.

Dále potřebujeme popsat úsečku a přímku. Nejjednodušší je parametrický popis. Uvažme, že úsečka je mezi body, jejich polohu zapíšeme jako vektor od počátku souřadnic. Pro body $\vec{X}$ ležící na ní platí

$$\vec{X} = \vec{A} + t\left(\vec{B} - \vec{A}\right),$$

kde t je reálný parametr nabývající hodnot mezi nulou a jedničkou. Zřejmě nula odpovídá bodu $\vec{A}$, jednička bodu $\vec{B}$ a ostatní hodnoty bodům mezi okraji. Pokud bychom z tohoto chtěli přímku, stačí vynechat omezení $t \in \langle 0, 1\rangle$.

Pro přímku však existuje i jiný způsob popisu. Uvažme, že známe vektor $\vec{n}$ kolmý na $\vec{B} - \vec{A}$. Pokud jím skalárně vynásobíme parametrický zápis přímky, dostaneme rovnici $\vec{X} \cdot \vec{n} + c = 0$, kde $c = -\vec{A} \cdot \vec{n}$ (tedy nějaká konstanta) a $\vec{X}$ obecný bod. Této rovnici se říká implicitní zápis přímky. Lze také ukázat, že každé řešení této rovnice je popsáno odpovídajícím parametrickým zápisem.

U implicitního zápisu ještě chvíli zůstaneme. Označme $\vec{s}$ vektor spojující body A a B, kterými prochází přímka, $\vec{s} = \vec{B} - \vec{A}$. Normálový vektor k němu zvolme $\vec{n} = (-s_y, s_x)$. Snadno nahlédneme, že opravdu $\vec{n} \cdot \vec{s} = 0$. Implicitní tvar rovnice přímky procházející body A a B tak může být zapsán jako $\vec{n} \cdot \vec{x} + c = 0$.

Nyní však uvažme, co se stane, pokud do ní dosadíme bod, který na přímce neleží. Podívejme se podrobněji, co dostaneme na pravé straně. Bod $\vec{X}$ lze zapsat jako $\vec{X} = \vec{A} + \alpha\vec{n} + \beta\vec{s}$, kde α a β jsou nějaká jednoznačně určená čísla (α určuje

posun po přímce od bodu A a β posun kolmo k ní). Po dosazení do implicitní rovnice dostaneme

$$\vec{n} \cdot \left(\vec{A} + \alpha\vec{n} + \beta\vec{s}\right) - \vec{n} \cdot \vec{A} = \beta\vec{n} \cdot \vec{n} = \beta|\vec{n}|^2.$$

Vzhledem k výše popsané konstrukci $\vec{n}$ si snadno čtenář ověří, že $|\vec{n}| = |\vec{s}|$, tedy že velikost normálového vektoru je rovna vzdálenosti bodů A a B. Kromě toho víme, že $\beta\vec{n}$ je takový posun směrem kolmým na přímku, abychom se z přímky dostali do bodu X. Tedy velikost tohoto vektoru (rovná $|\beta| \cdot |\vec{n}|$) je výška trojúhelníku ABX kolmá na stranu AB.

Proto výraz $|\beta| \cdot |\vec{n}|^2$ popisuje dvojnásobek obsahu trojúhelníku ABX. Ten v programu budeme určovat vzorcem $2S = |\vec{X} \cdot \vec{n} + c|$. Funguje jak pro určení obsahu trojúhelníku ABX, tak i pro určení rovnoběžky - zjevně stačí upravit konstantu c o $\pm 2S$.

Nakonec budeme potřebovat určit průsečík přímky a úsečky. Předpokládejme, že přímku máme implicitně zadanou ve tvaru $\vec{n} \cdot \vec{X} + c = 0$ a úsečku mezi body P a Q zadanou parametricky $\vec{X} = \vec{P} + t(\vec{Q} - \vec{P})$. Dosazením těchto rovnic do sebe a vyjádřením t dostaneme

$$t = -\frac{c + \vec{n} \cdot \vec{P}}{\vec{n} \cdot \left(\vec{Q} - \vec{P}\right)}.$$

Pokud platí, že takto spočtené $t \in \langle 0, 1\rangle$, průsečík existuje, jinak ne.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/23-2-5.c`

Pavel Čížek

23-2-6 Testovací

Nejjednduší řešení, které se nám na první pohled nabídne, je vyzkoušet všechny možné trojice (a_i, a_j, a_k), pro které platí $i < j < k$, a pro každou takovou trojici otestovat, zda platí rovnost $a_j - a_i = a_k - a_j$. Tím získáme jednoduché řešení pracující v $\mathcal{O}(N^3)$.

Jak si spousta z vás všimla, tento jednoduchý algoritmus můžeme urychlit tím, že využijeme setříděnosti posloupnosti a použijeme binární vyhledávání (o kterém se můžete dočíst v jedné z našich kuchařek). V naší úloze binární hledání využijeme k nalezení třetího prvku.

Tedy pro všechny dvojice (a_i, a_j) si spočítáme

$$a_k = a_j + (a_j - a_i)$$

a pokusíme se a_k vyhledat v intervalu a_{j+1} až a_{N-1}. Tím dostaneme řešení se složitostí $\mathcal{O}(N^2 \log N)$. Ale ani to ještě není optimálním řešením.

Optimální řešení pracuje v čase $\mathcal{O}(N^2)$ a využívá jak setříděnosti posloupnosti, tak toho, že ke každé dvojici (a_i, a_j) existuje nejvýše jedno a_k splňující podmínku. Jak na to?

Nejdříve si všimneme, že pokud $a_{k_0} - a_j < a_j - a_i$ platí pro nějaké k_0, tak tato nerovnost bude platit i pro všechna $k < k_0$. Naopak pokud $a_{j_0} - a_i < a_k - a_{j_0}$ platí pro nějaké j_0, tak stejná nerovnost platí i pro všechna $j < j_0$. Není těžké si na papíře rozmyset, proč.

A jak toho využijeme v našem řešení? Pro všechna možná i zvolíme $j = i + 1$ a $k = j + 1$ (následující prvky), pokud tedy $i + 2 < N$ (musí existovat), a dále opakujeme následující postup.

Pokud $a_j - a_i = a_k - a_j$, nalezli jsme řešení, vypíšeme jej a k a j zvýšíme o jedna (pro jedno a_j nemůže existovat více a_k).

Pokud $a_j - a_i > a_k - a_j$, zvýšíme k o jedna. Je důležité si uvědomit, že tuto operaci můžeme udělat a nepřijdeme tak o žádné řešení, protože pro všechna nižší k řešení už také neexistuje, nebo jsme jej už vypsali.

Zbývá nám možnost $a_j - a_i < a_k - a_j$. V tomto případě zvýšíme j o jedna, protože pro tohle j už žádné řešení nebude.

Celý postup opakujeme, dokud $k < n$.

Nyní si jen stačí uvědomit, že u neklesající posloupnosti, kde můžou být bloky stejných čísel, se nám nic hrozného nestane – jen když po zvýšení nějakého indexu $x \in \{i, j, k\}$ zjistíme, že $a_x = a_{x-1}$, zvýšíme jej ještě jednou.

Složitost je $\mathcal{O}(N^2)$, protože pro každé i maximálně N-krát iterujeme j i k o jedna. Toto řešení si můžete přečíst i jako zdrojový kód.

Nyní ještě dokážeme, že lepší časové složitosti v nejhorším případě nemůžeme dosáhnout. Uvažme jednoduše posloupnost $1, 2, \ldots, N$. V takové posloupnosti existuje $N - 2$ trojic s diferencí 1, $N - 4$ s diferencí 2, $N - 6$ s diferencí 3 atd., až nakonec 1 trojice s diferencí $(N - 1)/3$.

Počet všech trojic je tedy $(N^2 - 1)/4$, což je vzhledem k N kvadraticky mnoho, takže algoritmus může mít až kvadraticky velký výstup a nemůžeme dosáhnout lepší složitosti v nejhorším případě než $\Theta(N^2)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-2-6.c`

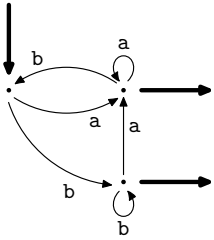
Karel Tesař

23-2-7 Regulomaty

Převeďte automat na obrázku na regulární výraz. To se drtivé většině z vás podařilo, strhával jsem body za chybějící vysvětlení.

`(1+2+3+)*` bylo správné řešení, někteří z vás zapomněli, že existuje operátor `+` a zapsali to jako `(11*22*33*)*`, za což jsem strhával řádově desetiny bodu.

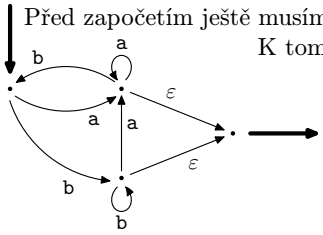
Jak se ovšem **úkol 1** řeší obecně? Jak dostanete z každého automatu regex, když jsem se v zadání chvástal, že to umím pro všechny? Existuje univerzální postup, který si tu předvedeme.



Postupně se budeme zbavovat vrcholů automatu, až nám jich zbyde jen pár, konkrétně ty vstupní a výstupní. Budeme na to pořád dokola používat tři operace:

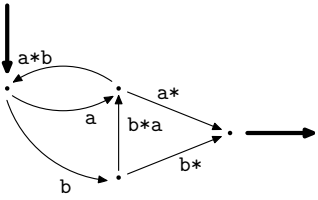
1) spojení paralelních hran
2) odstranění smyček
3) odstranění vrcholu

Celou věc si budeme ilustrovat na jiném, názornějším automatu, zde na obrázku.

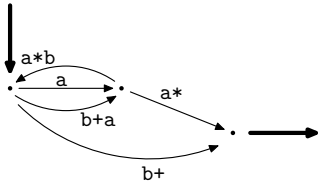


Před započetím ještě musíme automat upravit, aby měl jen jeden výstupní stav. K tomu použijeme ε-hrany, které si teď na chvíli povolíme.

Vytvoříme tedy jeden stav navíc, který bude oním jediným výstupním, a ze všech bývalých výstupních stavů do něj natáhneme ε-hrany.

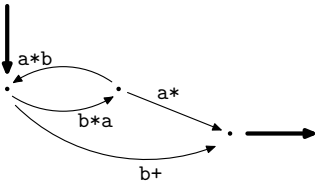


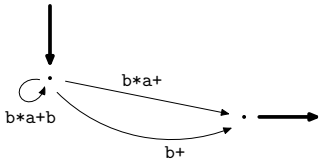
Nyní budeme cyklit pořád dokola naše tři body. Paralelní hrany zatím nemáme, ale smyčky se nějaké vyskytují, tak je zrušíme. Obalíme je hvězdičkou a připojíme na začátek výstupních hran. Teď už nebudou hrany označeny znakem, ale regexem. Nakonec zbydou dva stavy – vstupní a výstupní – a jediná hrana mezi nimi, která bude označena výsledným regexem.



Vybereme si nějaký vrchol a ten odstraníme. Jednoduše vytvoříme všechny možné kombinace hran, jež bylo možné použít, abychom prošli tímto vrcholem. Samozřejmě neodstraňujeme vstupní a výstupní vrchol! Všimněte si, že na obrázku už jsou spojené výrazy `bb*a` do `b+a` a `bb*` do `b+`.

A zase od začátku. Spojení paralelních hran, tentokrát tady jeden případ máme, tak pryč s nimi! Výrazy `a` a `b+a` se mi spojí do `(a|b+a)`, což můžeme upravovat postupně na `(b+)?a` a `b*a`, což je výsledný výraz.



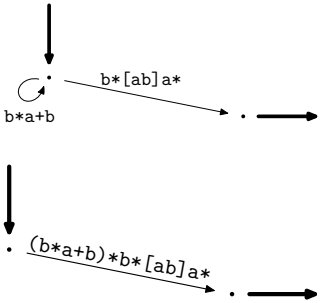


Eliminaci smyček pro tentokrát vynecháme, žádné v automatu zrovna nemáme, znovu budeme odstraňovat vrchol, teď už jediný odstranitelný.

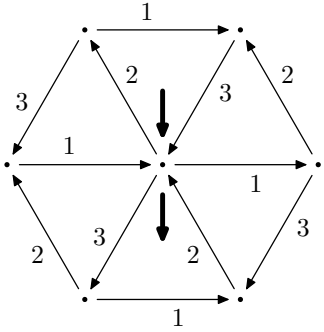
Zase jsme mohli nově vzniklé výrazy zjednodušit. Z výrazu `b*aa*` máme `b*a+` a na smyčce z původního `b*aa*b` vznikl `b*a+b`.

Přichází na řadu spojení hran, při kterém vznikne ze dvojice výrazů `b*a+` a `b+` postupně `(b+|b*a+)`, `b*(b|a+)` a `b*[ab]a*`. Poslední přeměna už nebyla úplně mechanická – výrazu totiž odpovídá libovolný řetězec nenulové délky, který nejdřív obsahuje jen `b` a potom jen `a`.

A po eliminaci smyček jsme u konce, na jediné hraně mezi vstupním a výstupním stavem máme výsledek.

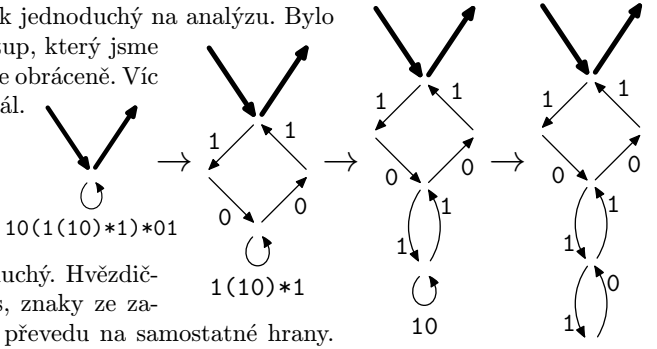


Správné řešení druhého úkolu bylo velice jednoduché, drtivá většina řešitelů za něj dostala plný počet bodů (s občasným stržením nějakých bodů za chybějící slovní popis, co že to je zač). Za nakreslení tohoto přehledného tvaru jsem uděloval malý bodový bonus.

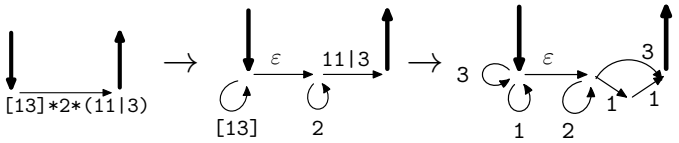


Uvedenému výrazu po krátkém zkoumání vyhovují všechny řetězce sestavené z permutací `123`, na obrázku jedničky symbolizují posun doprava, dvojky vlevo dolů a trojky vlevo nahoru. Abych tedy po každém třetím znaku byl v počátečním stavu, musím projít nějakou permutaci `123`...

Poslední úkol nebyl tak jednoduchý na analýzu. Bylo potřeba aplikovat postup, který jsme si právě předvedli, jenže obráceně. Víc řekne obrazový materiál.



Postup je tedy jednoduchý. Hvězdičku rozbalím na cyklus, znaky ze začátku a konce výrazu převedu na samostatné hrany. Více hvězdičkovaných výrazů oddělím ε-hranou. Kdyby se objevilo více možností, udělám z nich paralelní hrany. Předvedu ještě na příkladu `[13]*2*(11|3)`:



Můžete si stáhnout zdrojové kódy obrázků (Metapost).[32]

Jan „Moskyto“ Matějka

[32] `http://ksp.mff.cuni.cz/tasks/23/s2327.mp`

23-3-1 Úsporný kořen

Řešitelé, kteří mají dobrou grafovou intuici nebo dostatečně naposloucháno, si uvědomili, že jde dokázat, že kýžené vrcholy najdou uprostřed nejdelší cesty stromu. Jan Bok si dobře všimnul, že v dávné úloze 18-1-3 Keřík už jsme dokonce obecnější variantu problému nejdelší cesty ve stromu řešili.

Vezmeme zavděk algoritmem, který takové pozorování nevyužívá. Bude se zakládat na opakovaném obírání stromu o listy. Nejdřív ale několik otázek:

Může být list stromu na alespoň třech vrcholech úsporný kořen? Nemůže, protože soused takového listu je ke všem ostatním vrcholům o jednotku bližší (každá cesta z listu k dalšímu vrcholu totiž vedla přes něj), takže bude mít o jednotku menší hlobuku.

Změní se množina úsporných kořenů odstraněním všech listů stromu na alespoň třech vrcholech? Ne, protože takovou operací zmenšíme hloubku všech zbylých vrcholů právě o jedničku – vrcholy s minimální hloubkou zůstanou tytéž.

Proč právě o jedničku? Hloubka každého vrcholu je dána vrcholy, které jsou od něj nejdál. To ale musí být listy, jinak by šlo onu vzdálenost měřící cestu protáhnout a hloubku zvětšit.

Tím, že odstraníme všechny listy, tedy odstraníme všechny důvody, proč by nemohla být hloubka o jednotku menší. O víc to být nemůže, protože sousedi odstraněných nejvzdálenějších listů svědčí o existenci cesty o jednotku kratší.

Je dobré si rozmyslet, kde argumentace selhává na stromech, které ani tři vrcholy nemají.

Teď už je zřejmá správnost algoritmu, který vrací výsledek sama sebe pro strom obraný o všechny své listy, je-li spouštěn na stromu s třemi a více vrcholy. Pro strom na jednom či dvou vrcholech je množina úsporných kořenů rovna množině vrcholů.

Algoritmus skončí, protože každý strom na alespoň dvou vrcholech obsahuje alespoň dva listy (jsou to třeba konce nejdelší cesty).

Abychom se vešli do lineární časové složitosti, předpočítáme si stupeň (počet sousedů) každého vrcholu a při každém odtrhávání listů si jej zaktualizujeme. Budeme si také udržovat seznam listů grafu – vrcholy z něj zanikají odtrháváním a přibývají snížením stupně na jednotku.

Odůvodněním lineárnosti pak budiž to, že odstranění každého vrcholu nám trvá konstantně času – nezapomeňme, že odstraňujeme listy, takže aktualizace seznamu sousedů a stejně tak stupně se týká jen jediného souseda tohoto odstraňovaného.

Program (Python):
`http://ksp.mff.cuni.cz/viz/23-3-1.py`

Lukáš Lánský

23-3-2 Nejkratší cesta přes oceán

Nejprve se podíváme na to, jak vypadá ona nejkratší úsečka, kterou hledáme. Její krajní body leží na obvodech zadaných mnohoúhelníků, takže buďto na nějaké straně, nebo v nějakém vrcholu mnohoúhelníka.

Navíc, když si po chvíli uvědomíme, že řešením určitě bude kombinace vrchol-strana, nebo vrchol-vrchol, tak už není žádný problém vyzkoušet všechny takové kombinace v čase $\mathcal{O}(N^2)$. Může existovat i řešení strana-strana, ale pak existuje i jiné stejně dobré...

S trochou štěstí, třídění a binárního vyhledávání se dá takový algoritmus zrychlit až na $\mathcal{O}(N \log N)$, ale to stále není žádná sláva.

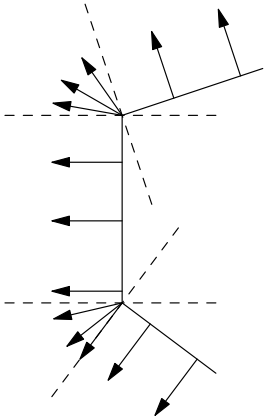
Naservírujeme si tedy trochu geometrických důkazů a vykoukáme z nich algoritmus ještě výrazně rychlejší.

První případ. Pokud je řešením kombinace strana-vrchol, pak ona hledaná úsečka bude na příslušnou stranu kolmá. Důkaz sporem. Uvažme stranu $\overline{XY}$ jednoho mnohoúhelníka, uvnitř které leží bod B (různý od X i Y); bod A je vrcholem druhého mnohoúhelníka. Hledanou úsečkou budiž $\overline{AB}$ a úhel ABY nechť není pravý.

Pak nechť B' je pata kolmice z bodu A na přímku XY. Pokud B' leží mezi X a Y, pak rozhodně $|AB'| < |AB|$, a tedy máme kratší úsečku a $\overline{AB}$ nebyla řešením.

Pokud by bod B' padl mimo $\overline{XY}$, pak na přímce XY leží body určitě v pořadí B', X, B, Y, nebo B', Y, B, X, přičemž vzdálenost od bodu A v tomto pořadí roste ($\overline{AB'}$ je nejkratší a $\overline{AY}$, resp. $\overline{AX}$ je nejdelší). Speciálně tedy jedna z $\overline{AY}$ a $\overline{AX}$ musí být kratší než úsečka $\overline{AB}$, která tedy není řešením. Spor. *QED*

Podobnou úvahou zjistíme, že pokud je řešením kombinace vrchol-vrchol, pak hledaná úsečka musí s oběma přilehlými stranami svírat alespoň pravý úhel, jinak na ni můžeme aplikovat argument z předchozích odstavců.



Takže pro každou hranu máme jen jeden směr, ve kterém z ní může vést hledaná úsečka, a pro každý vrchol interval směrů.

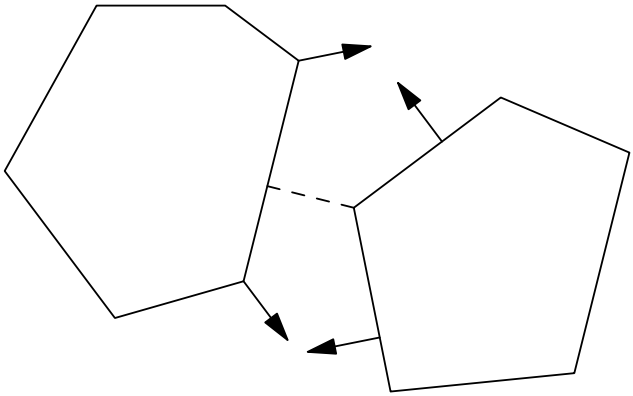
Co víc, mnohoúhelníky jsou konvexní, takže když si řekneme libovolný směr (úhel), tak nalezneme jen jedno místo na každém mnohoúhelníku, pro které tento směr připadá v úvahu.

Navíc jsme dostali ony mnohoúhelníky zadané jako body v pořadí na obvodu, takže můžeme jednoduše v lineárním čase postupně projít všechny možné směry.

Lze si to také představit tak, že máme dvě rovnoběžky, které otáčíme každou okolo jednoho z mnohoúhelníků stejným směrem (tak, abychom nepřeskočili žádný vrchol), a vždycky si ukládáme, kterého vrcholu se zrovna která ze přímek dotýká. Tak je také implementován vzorový program.

Jak zjistíme, že právě procházíme okolo řešení? Pokud je správným řešením kombinace vrchol-strana (A-BC), rozhodně se v jednu chvíli stane, že na jednom mnohoúhelníku máme zrovna vybraný bod A a na druhém přecházíme z B do C.

Navíc pro správné řešení jako jediné platí, že ABC je ostroúhlý trojúhelník, který se nepřekrývá se zadanými mnohoúhelníky. Důkaz je jednoduchý – od správného řešení se rozchází odpovídající si vrcholy na různé strany, viz obrázek.



Pro případ, že řešení je vrchol-vrchol, si ještě ukládáme vzdálenosti mezi projitými dvojicemi vrcholů. Pokud tedy doběhne cyklus bez toho, abychom vypsali výsledek a skončili, je správným řešením nalezené minimum vrchol-vrchol.

Lineární řešení (C):
`http://ksp.mff.cuni.cz/viz/23-3-2-1.c`

Čas je tedy $\mathcal{O}(N)$, paměť taktéž. Vyřešili jsme tedy úlohu tak rychle, jak rychle umíme načíst vstup.

Existuje drsnější řešení, které využívá modifikaci půlení intervalu. Jeho popis by vystačil na samostatný článek a jeho časová složitost je $\mathcal{O}((\log A)(\log E))$, kde A a E jsou počty vrcholů mnohoúhelníků. Nám však bohatě stačilo řešení lineární.

V úloze se masivně používá analytická geometrie a vektorový počet. Za zmínku stojí několik použitých faktů:

- Bod se dá považovat za vektor.
- Skalární součin dvou vektorů a, b je roven $|a||b|\cos\varphi$, tedy je kladný, pokud svírají ostrý úhel, záporný pro tupý úhel a nula pro pravý úhel φ.
- Normálový vektor a_N je kolmý na vektor a
- Skalární součin vektoru a a normálového vektoru b_N je kladný, je-li vektor b „na jedné straně“ od vektoru a, jinak záporný (a pro vektory opačného směru nulový). Kladná a záporná strana závisí na definici normálového vektoru (je-li to „ten kolmý vlevo“, nebo „ten kolmý vpravo“).

Jan „Moskyto“ Matějka & Jitka Novotná

23-3-3 Skok bez padáku

Úloha má přehršel parametrů a u takových se obvykle stává, že složitost různých řešení závisí na různých parametrech. Tak si je pojďme pojmenovat:

(x_0, y_0)	počáteční pozice
h_0	výška, ze které smíme spadnout
T	počet trampolín
W	šířka (pozice nejpravější trampolíny)

Ujasnění zadání. Zadání zarytě mlčí o dvou důležitých věcech:

- Jsou souřadnice celočíselné? Nikoho z řešitelů naštěstí nenapadlo, že by nemusely být, tak to předpokládejme také. (Jinak by totiž úloha byla mnohem zákeřnější – byla by vůbec řešitelná v konečném čase?)
- Co se stane, když padáme z výšky 1? Pak by měl následovat odraz do výšky 0. A pokud spadneme na jednu z několika sousedních trampolín, můžeme po nich pak volně chodit a na kraji seskočit dolů? Raději nulové odrazy zakážeme. (Kdybychom je opravdu chtěli, náš algoritmus půjde snadno upravit, aby s nimi počítal.)

Pár pozorování pro začátek. Předně, pokud spadneme z bodu (x, y) na trampolínu (x, t), odrazíme se do výšky $y' = \lfloor (y + t)/2 \rfloor$ a odtamtud se posuneme buďto do $(x - 1, y')$, nebo do $(x + 1, y')$. Jelikož $t < y$ (trampolína leží pod námi) a nulové odrazy jsme zakázali, musí být i $y' < y$. Takže postupně padáme z čím dál tím nižších bodů.

Proto ať už se odrážíme jakkoliv, po konečně mnoha odrazech spadneme na zem (živí či mrtví; se schrödingerovsky kočkovitými parašutisty nepočítáme). Dokonce víme, že odrazů je vždy nejvýše y_0.

Rekurzivní řešení. Nejprve se podíváme na první podúlohu. Chceme tedy naprogramovat funkci, která dostane počáteční polohu (x_0, y_0) a oznámí, jaký je minimální počet odrazů, chceme-li přežít (nebo $+\infty$, pokud nemáme šanci). Tato funkce si může spočítat, která trampolína leží pod zadaným bodem, odrazit se od ní, a vyzkoušet jak posunutí doleva, tak doprava.

Každá z těchto možností zase dává nějaký bod, ze kterého budeme padat. Který si vybrat? Nevíme. Tak zkusíme oba. Pro každý se zavoláme rekurzivně a zjistíme, která možnost dává menší počet odrazů. O 1 větší počet pak prohlásíme za svůj výsledek. Jak už víme, stále klesáme, takže výpočet se nemůže zacyklit.

Zbývá ošetřit triviální případ, totiž ten, že už pod námi žádná trampolína neleží. Pak podle toho, zda už jsme v bezpečné výšce, vrátíme buď 0 nebo $+\infty$.

Toto je jistě funkční řešení, bohužel ale poněkud hlemýždí – pro každý odraz se dvakrát rekurzivně voláme, takže pro nejvýše y_0 odrazů dostáváme exponenciální časovou složitost $\mathcal{O}(2^{y_0})$. (Náš odhad počtu odrazů je poněkud přemrštěný, ale i s tím správným, který časem dokážeme, vyjde exponenciála.)

Jak neopakovat výpočty. Čím všechen ten čas trávíme? Inu, počítáme pořád dokola totéž. Vstupem naší funkce je totiž dvojice souřadnic a různých dvojic existuje pouze $W \times y_0$.

Algoritmus tedy můžeme vylepšit tím, že si pořídíme pole („blbenku“) a budeme si v něm pamatovat, pro které počáteční polohy už známe výsledek a jaký je. Před každým voláním funkce se tam podíváme a pokud už hodnotu známe, použijeme ji. Jinak volání provedeme a výsledek si poznamenáme. Tím celkový počet volání snížíme na $\mathcal{O}(Wy_0)$.

Jak najít trampolínu. V předchozím rozboru jsme poněkud zamluvili, že potřebuje pro zadanou polohu zjistit, jaká je nejbližší nižší trampolína. Na to by se dalo jít všelijak chytře, třeba si souřadnice trampolín setřídit lexikograficky a pak v nich půlením intervalu hledat.

My na to ale půjdeme jinak: předpočítáme si „navigační tabulku“ tvaru $W \times y_0$, která nám pro každý bod řekne, jak hluboko pod ním je trampolína.

Nejprve tabulku vyplníme nulami, jen na pozice trampolín napíšeme jedničky. Pak pole projdeme zespoda nahoru a doplňujeme hodnoty. Jedničky zůstanou jedničkami, pro každou nulu se podíváme, co je pod ní. Pokud nula, ponecháme naši nulu. Pokud něco jiného, naše hodnota bude o 1 větší. Výpočet tabulky tedy bude trvat čas $\mathcal{O}(Wy_0 + T)$.

Každý krok našeho rekurzivního algoritmu s blbenkou teď už umíme provést v konstantním čase, celý algoritmus tedy poběží v čase $\mathcal{O}(Wy_0 + T)$.

Zespoda nahoru. Rekurzi s blbenkou obvykle můžeme zjednodušit na dynamické programování. Tím myslíme, že budeme blbenku rovnou počítat zespoda nahoru – pro výpočet každé hodnoty potřebujeme jenom hodnoty z nižších řádků, které už budeme mít spočítané.

Přesněji řečeno, označíme si $P[x, y]$ minimální počet odrazů při pádu z bodu (x, y) a budeme zespoda nahoru provádět toto:

- Pokud pod (x, y) neleží žádná trampolína, položíme
$$P[x, y] = \begin{cases} 0 & \text{pro } y \leq h_0, \\ +\infty & \text{pro } y > h_0. \end{cases}$$
- Pokud pod leží (x, y) trampolína (x, t), spočítáme výšku po odrazu $y' = \lfloor y + t \rfloor$ a položíme
$$P[x, y] = \min(P[x - 1, y'], P[x + 1, y']) + 1.$$
Ptáme-li se na hodnotu mimo tabulku, použijeme $+\infty$.

S předvýpočtem navigační tabulky seběhne i tento algoritmus v čase $\mathcal{O}(Wy_0 + T)$, ale je daleko jednodušší. Proto jsme ukázkový program psali podle něj.

Podúloha b. Druhou podúlohu, totiž stanovení všech výšek, ze kterých spadnuvše bychom přežili, získáme jako vedlejší produkt právě popsaného algoritmu. Stačí se totiž do tabulky P podívat na x_0-tý sloupec a vypsat ta y, pro něž je $P[x_0, y]$ konečné. To stihneme v čase $\mathcal{O}(y_0)$, takže nám to časovou složitost nezhorší.

Pseudopolynomiální složitost. Algoritmus, který jsme si ukázali, má takzvaně pseudopolynomiální složitost. Tím se myslí, že složitost není polynom ve velikosti vstupu, nýbrž v hodnotách čísel obsažených na vstupu. U opravdového polynomiálního algoritmu by tedy směla záviset pouze na T, nikoliv na y_0, h_0 nebo W. Případně pokud bychom (jak se často činí) měřili velikost vstupu v bitech, byla by vzhledem k velikosti vstupu polynomiální také čísla $\log y_0$, $\log h_0$ a $\log W$. Neuměli bychom najít poctivé polynomiální řešení?

Lepší odhad na počet odrazů. Především si všimneme, že naše omezení počtu odrazů číslem y_0 bylo naprosto přemrštěné. Zaměřme se na jednu trampolínu a sledujme výšky, do kterých se dostaneme po jednotlivých odrazech. Kdyby žádné jiné trampolíny neexistovaly (a dovolili bychom si na chvíli po odrazu neuhnout doleva ani doprava), dělila by se po každém odrazu výška dvěma, takže po řádově $\log y_0$ odrazech by byla nulová. Teď vrátíme ostatní trampolíny do hry a všimneme si, že tím, že jsme si na ně odskočili (doslova), jsme si při dalším návratu na naši trampolínu mohli výšku jedině zmenšit. Takže i tehdy je počet odrazů o jednu trampolínu nejvýše $\log y_0$ a celkem se proto můžeme odrazit nejvýše $(T \log y_0)$-krát.

Odstranění závislosti na W. Závislosti na parametru W (šířce mapy) se můžeme zbavit snadno. Všimneme si totiž, že se ve vodorovném směru nikdy nedostaneme dál než o T kroků od počátku. Do vzdálenosti $T + 1$ musí přeci ležet aspoň jeden sloupec bez trampolíny a ten nemáme jak přeskočit. Stačí tedy pole P v našem algoritmu omezit na velikost $(2T + 1) \times y_0$ (sloupec odpovídající souřadnici x_0 bude uprostřed) a trampolíny ležící mimo ignorovat. Tím časovou složitost zlepšíme na $\mathcal{O}(Ty_0)$.

Závislost na y_0. Ve svislém směru to nedopadne tak skvěle. Nabízí se využít toho, že během jednoho seskoku spadneme na jednu trampolínu nejvýše $(\log y_0)$-krát, takže bychom políčka nad touto trampolínou mohli rozdělit na nějaké intervaly, uvnitř kterých je $P[x, y]$ konstantní, a pamatovat si pouze hranice intervalů a jednu hodnotu pro každý z nich. Takových algoritmů se dá vymyslet vícero, ale všechny selžou na tom, že v různých seskocích může být toto rozdělení na intervaly různé, takže intervaly se mohou množit a množit, až jich nakonec bude řádově y_0.

Je tato hrozba reálná? Bohužel ano – ukážeme konstrukci vstupu, který se v těchto ohledech chová značně ošklivě. Předem varujeme, že to nebude úplně snadné; čtenář neprahnoucí po dobrodružství nechť raději přeskočí k podpisu autora na konci řešení.

Ještě tu jste? Dobrá, jdeme na to. Nejdříve si uvědomíme, jak se mění souřadnice, když se během jednoho seskoku odrážíme postupně od trampolín ve výškách $t_1, t_2, \ldots, t_n$. Už víme, že po prvním odrazu vyskočíme do výšky $y_1 = (y_0 + t_1)/2$ (zaokrouhlení s dovolením zanedbáme a pak budeme volit výšky tak, aby vždy vy-

šlo celé číslo). Obecně $y_i = (y_{i-1} + t_i)/2$. Pokud tyto vztahy složíme dohromady, dostaneme:

$$y_n = \frac{y_0}{2^{n+1}} + \frac{t_1}{2^n} + \frac{t_2}{2^{n-1}} + \ldots + \frac{t_n}{2^1}. \qquad (*)$$

Naše konstrukce bude vypadat tak, že si zvolíme nějaká čísla $x_1, \ldots, x_T$ a rozmístíme T trampolín na souřadnice $(n - i, x_i)$. Uvažujme, do jakých výšek nad nejpravější trampolínou se můžeme dostat při různých způsobech seskoku. Ukážeme, že možných výšek je spousta, a to dokonce i tehdy, když se omezíme na některé speciální druhy seskoků.

Kterýkoliv seskok můžeme jednoznačně popsat posloupností rozhodnutí o směru doleva/doprava po jednotlivých odrazech. Nás budou zajímat pouze seskoky složené z úseků tvaru `PPLP` nebo `PLPP`. Všimněte si, že každý takový úsek nás posune přesně o 2 trampolíny doprava, takže po $T/2$ úsecích proskáčeme celou posloupnost trampolín; celkem se při tom odrazíme $2T$-krát.

Nyní použijeme vzoreček $(*)$ a uvážíme, jak k finální výšce přispějí trampolíny v i-tém úseku. Úseky přitom očíslujeme od nultého úplně vpravo, takže i-tý úsek bude složený z trampolín $(n - 2i - 2, x_{2i+2})$ a $(n - 2i - 1, x_{2i+1})$ a navštívíme ho ve skocích s vahami (to jsou ty mocniny dvojky ve vzorečku) 2^{4i+4} až 2^{4i+1}.

Pokud ho proskáčeme způsobem `PPLP`, přispěje k součtu hodnotou

$$A_i = \frac{x_{2i+2}}{2^{4i+4}} + \frac{x_{2i+1}}{2^{4i+3}} + \frac{x_{2i}}{2^{4i+2}} + \frac{x_{2i+1}}{2^{4i+1}}.$$

Při `PLPP`:

$$B_i = \frac{x_{2i+2}}{2^{4i+4}} + \frac{x_{2i+1}}{2^{4i+3}} + \frac{x_{2i+2}}{2^{4i+2}} + \frac{x_{2i+1}}{2^{4i+1}}.$$

Rozdíl těchto dvou hodnot označíme

$$C_i = B_i - A_i = \frac{x_{2i+2} - x_{2i}}{2^{4i+2}}.$$

Finální výšku tedy můžeme vyjádřit jako součet všech A_i, ke kterému přičteme ta C_j, která odpovídají úsekům typu `PLPP`.

Uvažujme nyní nějakou obecnou posloupnost přirozených čísel $z_0, \ldots, z_K$ ($K = T/2 - 2$). V naší konstrukci nastavíme $x_i = 0$ pro všechna lichá i, dále položíme $x_0 = 0$ a $x_{2j+2} = x_{2j} + z_j \cdot 2^{4j+2}$ pro všechna j. Navíc zvolíme počáteční výšku y_0 tak, aby byla větší než $2^{2T+1} \cdot \max_i x_i$ – tím zařídíme, že se během seskoku délky $2T$ nemůžeme dostat pod žádnou z navržených trampolín.

Touto volbou hodnot x_i jsme zařídili, že rozdíly C_i z předchozího výpočtu jsou rovny právě z_i. Jinými slovy, výšky dosažitelné zkoumanými druhy seskoků se dají napsat konstanta plus součet nějaké podmnožiny čísel z_i.

K dokončení stačí klasický trik: z mocnin dvojky $2^0, \ldots, 2^K$ se dají nasčítat všechna čísla od 0 do $2^{K+1} - 1$ (tak funguje dvojková soustava). Pro volbu $z_i = 2^i$ tedy existuje alespoň 2^{K+1} dosažitelných výšek, což je exponenciální vzhledem k T.

Navíc počty použitých trampolín odpovídají počtu jedniček v binárním zápisu čísla, což se mění příliš rychle na to, aby intervalů mohlo být řádově méně. EPA.[33]

Program (C):
`http://ksp.mff.cuni.cz/viz/23-3-3.c` *Martin „Medvěd“ Mareš*

23-3-4 Psaní písmen

Poznámka redakce: Zadavatel této úlohy do CodExu ji pozměnil. Oproti zadání v letáku a na webu byl na vstupu zadán graf explicitně rozsekaný na komponenty. Navíc zadání v CodExu vyžadovalo optimalizaci na paměť. Tomu odpovídá i zdrojový kód.

Abychom mohli úlohu vyřešit, měli bychom vědět, co jsou to eulerovské tahy a jaké podmínky splňují grafy, které je obsahují (nahlédnout můžete do našich grafových kuchařek). To, že jde obrázek nakreslit jedním tahem, znamená, že obsahuje uzavřený či otevřený eulerovský tah.

Pokud souvislý graf obsahuje pouze vrcholy sudého stupně, je v něm možno nalézt uzavřený eulerovský tah.

Co se stane, pokud neobsahuje pouze vrcholy sudého stupně? Mezi dvojici lichých vrcholů přidáme hranu (opakujeme, dokud máme vrcholy lichého stupně), takto postupně dostaneme graf, ve kterém jsou všechny vrcholy sudého stupně, tedy obsahuje uzavřený eulerovský tah.

Nyní odebereme hrany, které jsme přidali, a tento eulerovský tah se nám rozpadne na několik hranově disjunktních tahů, které vždy začínají a končí v nějakém vrcholu lichého stupně (jeden počáteční lichý vrchol a jeden koncový lichý vrchol pro každý tah), tudíž celkový počet těchto tahů je počet lichých vrcholů děleno dvěma.

Žádný vrchol lichého stupně nemůže být uprostřed tahu, tudíž tahů nemůže být méně, než jsme našli.Stačí nám vědět, kolik takových tahů potřebujeme, není tedy potřeba je konstruovat, stačí nám určit počet lichých vrcholů (a dát si pozor na grafy bez lichých vrcholů).

Samotné řešení úlohy (provedeme pro každou komponentu samostatně):

Potřebujeme pole délky n (počet vrcholů), při načítání si v něm udržujeme stupně jednotlivých vrcholů. Po načtení projdeme toto pole a určíme počet lichých vrcholů, který vydělíme 2. Dostaneme, kolikrát musíme zvednout pero při kreslení grafu.

Paměťová složitost je $\mathcal{O}(n)$, časová složitost je $\mathcal{O}(m+n)$, m je počet hran grafu.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-3-4.c`

Martin Böhm & Lucie Mohelníková & CodEx

[33] Est post aves. To je něco jako „Quod erat demonstrandum“ ale znamená to „A je po ptákách.“

23-3-5 Rozházené EWD

Úkolem bylo setřídit zadaný jednosměrný spojový seznam co nejrychleji, ale v konstantní paměti, což znamená jen s předem daným počtem proměnných, bez rekurze a dalších pomocných polí, tedy pouze přepojováním původního spojového seznamu.

Určitě bylo dobrým nápadem podívat se do naší kuchařky o třídění.[34] A co s tak malou pamětí? Bublinkové třídění (bubble sort) bude zcela jistě fungovat, protože v průběhu algoritmu prohazujeme jen dva sousední prvky, což lze udělat jednoduše.

Bublinkové třídění má navíc pěknou vlastnost, že třídění již setříděných dat trvá pouze $\mathcal{O}(N)$. Jenže nejhůře a dokonce i průměrně vyjde asymptotická složitost $\mathcal{O}(N^2)$. Je to nejrychlejší možný výsledek za daných podmínek, nebo ne?

Než si řekneme řešení, uveďme si dolní odhad složitosti. Jelikož stáří záznamů EWD můžeme akorát tak porovnávat (nic o nich nevíme), platí důkaz uvedený na konci kuchařky o třídění, a tedy určitě nevymyslíme algoritmus s průměrnou složitostí lepší než $\mathcal{O}(N \log N)$.

Takový algoritmus existuje. My si ukážeme, jak modifikovat třídění sléváním (Mergesort) se zachováním složitosti v nejhorším případě i v průměru $\mathcal{O}(N \log N)$, na což přišlo i několik řešitelů. Nevylučuji však, že nepůjde upravit jiný algoritmus, i když třídění haldou ani Quicksort nejspíš převést na řešení úlohy nelze.

Jak funguje takový běžný Mergesort na třídění pole? Ten si nejprve rozdělí pole na dvě půlky, ty setřídí stejným algoritmem (zavolá se na každou rekurzivně) a pak je „slije“ tedy odebírá vždy menší z prvků na začátku obou setříděných půlek pole a vkládá je do nového pole. Podrobnější popis opět v kuchařce.

Nyní upravíme Mergesort pro potřeby naší úlohy. Jelikož nesmíme použít rekurzi, nebudeme postupovat „odshora dolů“ (postupně půlíme data na co nejmenší části), ale „odspoda nahoru“ (spoustu malých setříděných částí sléváme postupně do jedné).

V prvním kroku se podíváme na všechny dvojice sousedních prvků (každý prvek je nejvýše v jedné dvojici), porovnáme prvky dvojice a případně je prohodíme, což v případě spojového seznamu znamená přepojení odkazů. V druhém kroku sléváme vždy dvě sousední dvojice prvků do setříděné čtveřice, v třetím dvě čtveřice do osmice…

Obecně v k-tém kroku slijeme dvě sousední části o 2^k prvcích. Až slijeme všechny prvky do jedné setříděné posloupnosti, máme vyhráno.

Často se může stát, že poslední slévaný úsek v k-tém kroku nemusí mít 2^k prvků, ale to vůbec nevadí (jeden slévaný úsek bude menší). Podobně lichý počet slévaných úseků (nemůžeme je spárovat do dvojic) ošetříme prostým ignorováním posledního úseku. V nějakém pozdějším kroku musí být tento úsek slit se zbytkem, třeba pro $2^n + 1$ prvků se bude poslední prvek slévat až v posledním kroku.

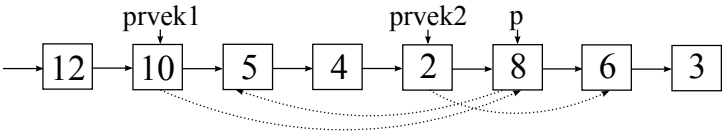
[34] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Nyní pojďme na implementaci slévání dvou setříděných úseků ve spojovém seznamu (ne nutně stejné délky) s konstantní pomocnou pamětí. Budeme si pamatovat odkaz na prvek před prvním úsekem (tedy poslední prvek již slité části) v proměnné `prvek1` a odkaz na prvek před druhým úsekem v proměnné `prvek2`.

Na začátku slévání dvou úseků nejprve posuneme odkaz `prvek2` o délku prvního úseku za odkaz `prvek1`. Abychom mohli kontrolovat, jestli v nějakém úseku nedošly prvky, vytvoříme si dvě proměnné `delka1` a `delka2`, v nichž budou počty zbývajících prvků v úsecích.

Pak postupně bereme prvky ze začátku obou úseků (následníky prvku `prvek1` a `prvek2`) a menší z nich přepojíme za prvek `prvek1`. Je-li to prvek z prvního seznamu, stačí posunout odkaz `prvek1` o jeden prvek dopředu, jinak je to následník `prvek2` (označme ho `p`), který přepojíme za `prvek1` takto: následníkem `p` bude následník `prvek1`, následníkem `prvek1` bude `p`, následníkem `prvek2` bude původní následník `p`.

Jestli vás předchozí odstavec zmátl, vůbec se nedivím a raději předkládám obrázek (tečkované šipky ukazují přepojení prvku `p`):



Je vidět, že potřebujeme jen konstantně mnoho pomocné paměti. Co se týče časové složitosti, bude pro jakákoliv data $\mathcal{O}(n \log n)$, kde n je počet prvků. V k-tém kroku totiž sléváme úseky o 2^k prvcích, a bude-li $2^k > n/2$, získáme po tomto kroku celý setříděný spojový seznam. Odtud zlogaritmováním dostaneme, že stačí $\log_2 n$ kroků, přičemž v každém provedeme $\mathcal{O}(n)$ operací.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-3-5-full.c`

Pavel Veselý

23-3-6 Výzkum veřejného mínění

Tato úloha měla spoustu možností, jak ji řešit. My si ukážeme jedno kvadratické řešení a pak řešení v čase $\mathcal{O}(N \log N)$. Nejdříve se podíváme na kvadratické řešení.

Vstupní posloupnost si načteme do dvou polí. V poli X budeme mít posloupnost, tak jak přišla na vstupu, a do pole Y uložíme posloupnost setříděnou podle velikosti.

Nyní si všimneme, že každá dvě po sobě jdoucí čísla v poli X nám určují intervaly mezi vstupními období (kde popularita klesá/stoupá) a dvě po sobě jdoucí čísla v poli Y určují intervaly hodnot, které budou mít stejnou četnost výskytů.

My tedy z každého intervalu v poli Y vezmeme libovolnou hodnotu, (například prostřední), a spočítáme, kolikrát se vyskytuje v intervalech pole X.

Nyní k řešení pracující v čase $\mathcal{O}(N \log N)$. Existuje spousta způsobů, jak na úlohu jít. My si ukážeme techniku zvanou Zametání přímkou (line sweep), pomocí které se mimo jiné dají řešit i některé geometrické úlohy.

Představme si, že se ke grafu popularity blíží přímka rovnoběžná s osou x. Tato přímka začne v minus nekonečnu, projde grafem od zdola nahoru a skončí v plus nekonečnu. Nás v každém okamžiku bude zajímat, kolikrát přímka protíná graf.

Všechny okamžiky ale testovat nemůžeme, tak se budeme věnovat jen těm, ve kterých se počet průsečíků s přímkou mění. Takovým okamžikům budeme říkat události a tyto události budeme zpracovávat v pořadí, v jakém nastanou při průchodu od zdola nahoru.

V našem případě jsou události všechny body, ve kterých se mění počet průsečíků s přímkou. Všimneme si, že tento počet se nám bude měnit pouze v lokálních maximech a minimech (tam, kde je špička). V maximu nastanou dvě události: nejdříve se počet průsečíků zmenší o jedna (došli jsme do špičky) a poté špičku opustíme a počet průsečíků se znova zmenší o jedna. Podobné budou i události u minima.

My si tedy pro každý bod vytvoříme příslušné události (pozor, u krajních bodů je pouze událost opuštění/přidání špičky) a tyto události si setřídíme primárně podle výšky a sekundárně podle jejich priority.

Priorita událostí je:

1. změna na špičku maxima
2. přidání špičky minima
3. opuštění špičky maxima
4. rozdvojení špičky u minima

Zkuste si rozmyslet, proč jsou priority událostí právě takto a v jakém případě může nastat problém, kdyby žádné priority nebyly. Teď už jen postupně zpracováváme všechny události a po každém zpracování zkontrolujeme, jestli nejsme v maximálním počtu průsečíků. Po zpracování všech událostí vypíšeme výsledek. Na první pohled to vypadá docela složitě, ale vlastně je to jednoduché. Viz zdrojový kód.

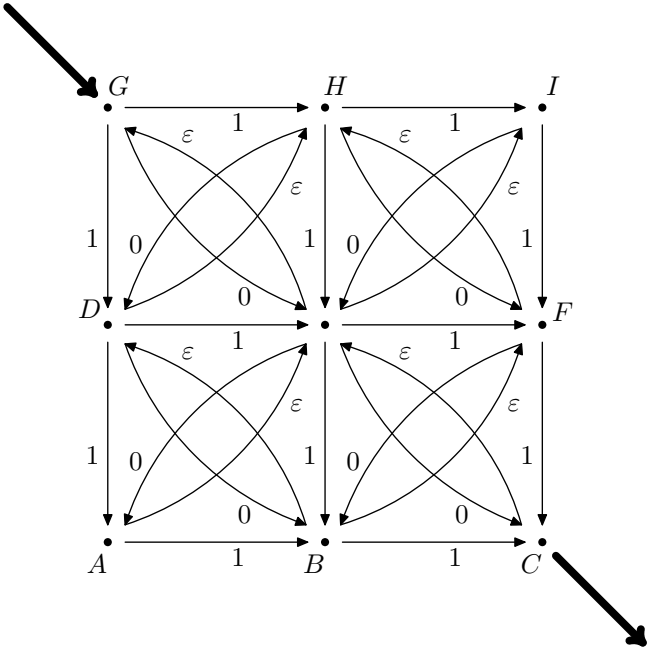
Program (C++):
`http://ksp.mff.cuni.cz/viz/23-3-6.cpp`

Karel Tesař

23-3-7 Automaty stokrát jinak

Třetí sérií uzavíráme seriálovou odbočku k automatům. Ještě nám chybí vysvětlit převod NKA na DKA a redukci automatu.

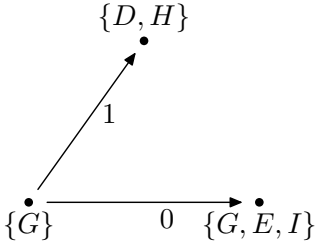
Nejprve si ukážeme převod NKA na DKA třeba na zadání **úkolu 1**. Označíme si jednotlivé stavy třeba písmeny A až I jako na obrázku (ten stav uprostřed je E, jen se to tam nevešlo).



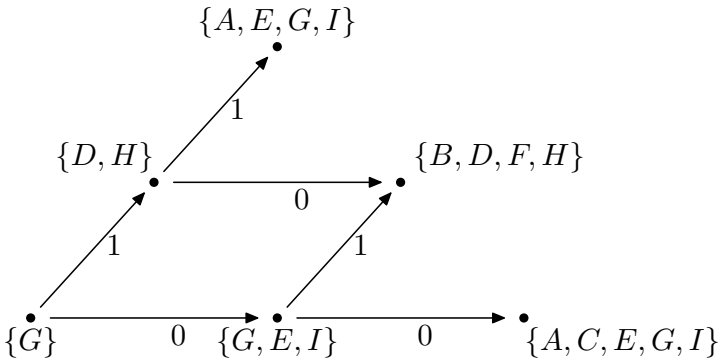
Nyní budeme konstruovat DKA, kde budou jako stavy množiny stavů původního NKA. Vstupní stav je G. Z něj se můžeme dostat přečtením znaku 1 do $\{D, H\}$ a přečtením 0 do $\{G, E, I\}$.

Kam se nyní můžeme dostat z $\{D, H\}$ přečtením 0? Ze stavu D jde jít do B (a pak po ε hranách do D, F a H), z H jde jít do D a F (a po ε hranách do H), tedy z $\{D, H\}$ vede hrana popsaná 0 do stavu $\{B, D, F, H\}$.

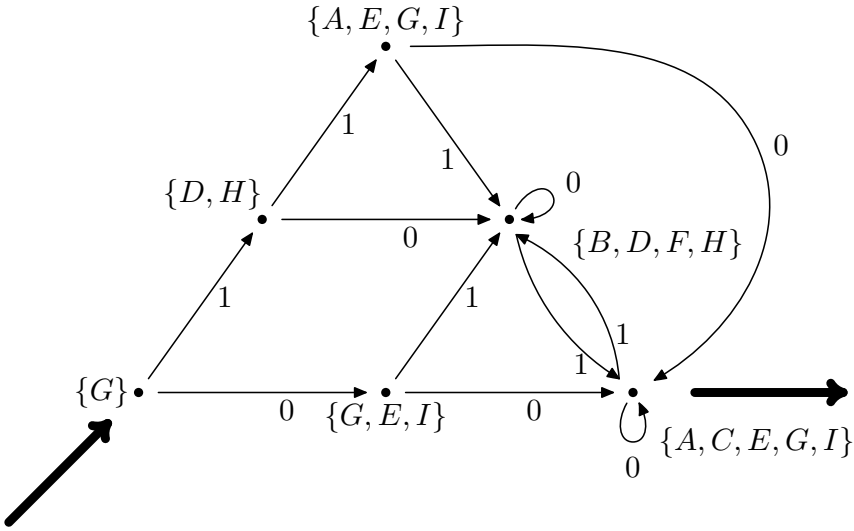
Analogicky z $\{D, H\}$ přečtením znaku 1 dojdeme do $\{A, E, G, I\}$. Z $\{G, E, I\}$ pak vedou hrany 0 a 1 do $\{A, C, E, G, I\}$ a $\{B, D, F, H\}$.



Stejným způsobem pak ještě doplníme hrany z nově vzniklých tří stavů (další už nevzniknou, ale teoreticky by mohly – výsledný DKA může mít až 2^N stavů oproti NKA s N stavy).

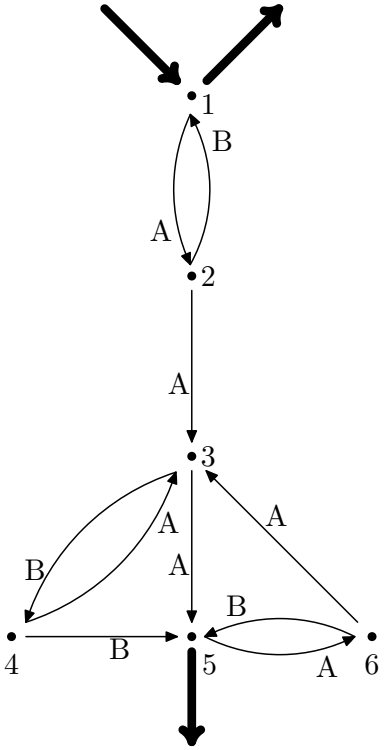


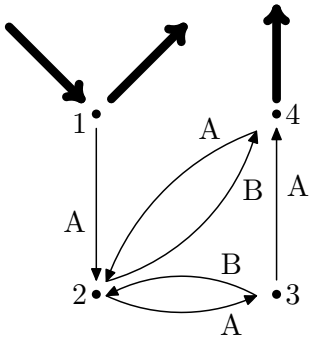
Výstupní stavy jsou pak všechny ty, v jejichž množinách se vyskytuje alespoň jeden výstupní stav původního NKA. V tom byl v našem případě výstupním stavem jen C, který se i ve výsledném automatu vyskytuje v jediném stavu. Ten je tedy výstupním.



Na obrázku vidíte kompletní zkonstruovaný DKA a zároveň řešení úkolu 1.

Když jste převedli oba dva výrazy z **úkolu 2** na NKA (postupem z minulé série) a touto metodou na DKA, dostali jste přibližně automaty, které uvidíte na následující stránce.





Věřili byste, že jsou ekvivalentní, tedy že přijímají stejný jazyk? Na první pohled to tak rozhodně nevypadá, ale jsou. Jak na to přijdeme?

Ukážeme si postup zvaný „redukce automatu“, kdy nalezneme všechny stavy, které jsou ekvivalentní, a sloučíme je. Například si můžeme všimnout, že u řešení úkolu 1 by šlo sloučit (nerozlišitelné) stavy $\{A, E, G, I\}$ a $\{G, E, I\}$. Ať přečtu cokoli, skončím na stejném místě.

	A	B	
→ <u>1</u>	2	–	ω
2	3	1	α
3	5	4	α
4	3	5	α
<u>5</u>	6	–	ω
6	3	5	α

Zapíšeme si levý automat tabulkou. Šipka značí vstupní stav, podtržení výstupní. Ve sloupci vpravo jsou zapsány kategorie stavů – jak by automat z tohoto stavu pokračoval, kdyby na vstupu už nebyl žádný znak.

Tabulku budeme dále rozšiřovat. Předpokládejme, že je na vstupu o znak víc:

	A	B		A	B	
→ <u>1</u>	2	–	ω	α	–	ω
2	3	1	α	α	ω	α
3	5	4	α	ω	α	β
4	3	5	α	α	ω	α
<u>5</u>	6	–	ω	α	–	ω
6	3	5	α	α	ω	α

Zjistili jsme, jak se automat chová po přečtení jednoho znaku. Vidíme, že stavy 2, 4 a 6 jsou nerozlišitelné, pokud přečteme maximálně jeden znak ze vstupu. Taktéž stavy 1 a 5.

Třetí, separátní kategorii jsme museli zavést pro stav 3, který se začal lišit od stavů 2, 4 a 6.

	A	B		A	B		A	B	
→ 1	2	–	ω	α	–	ω	α	–	ω
2	3	1	α	α	ω	α	β	ω	α
3	5	4	α	ω	α	β	ω	α	β
4	3	5	α	α	ω	α	β	ω	α
5	6	–	ω	α	–	ω	α	–	ω
6	3	5	α	α	ω	α	β	ω	α

Provedeme ještě jeden krok a zjistíme, že se už kategorie stavů nezměnily.

Jedna hezká věta říká, že jakmile se jednou nezmění kategorie stavů, nezmění se nikdy. To je docela jasně vidět, když si uvědomíte, že by se vlastně pořád dokola opakovaly stejné trojice sloupečků.

Další hezká věta říká, že poslední trojice sloupečků nám popisuje tzv. redukovaný automat, který je ekvivalentní s tím původním. Duplicitní řádky vynecháme.

	A	B
→ ω	α	–
α	β	ω
β	ω	α

Když pak provedeme totéž pro druhý automat, dostaneme podobnou tabulku.

	A	B		A	B		A	B	
→ 1	2	–	α	β	–	α	β	–	α
2	3	4	β	β	α	β	γ	α	β
3	4	2	β	α	β	γ	α	β	γ
4	2	–	α	β	–	α	β	–	α

$\longrightarrow$

	A	B
→ α	β	–
β	γ	α
γ	α	β

Automaty tedy jsou ekvivalentní, neboť jejich redukované verze jsou ekvivalentní (stačí tabulky přepísmenkovat). Proto i dva zadané regexy jsou ekvivalentní, tedy popisují stejný jazyk.

Jedno obtížně dokazatelné tvrzení říká, že pokud jsou dva automaty ekvivalentní, pak je lze zredukovat tímto postupem na stejný DKA, až na isomorfismus.

Isomorfní DKA jsou takové, že pokud správně přečíslujeme stavy jednoho z nich, tak dostaneme druhý automat. Ač se to nezdá, na problém neznáme polynomiální algoritmus, ale ani nevíme, jestli je NP-úplný.

A jaké bylo správné řešení **úkolu 3**? `10101010101201` je nejmenším násobkem devíti vyhovujícím zadanému výrazu. Bylo potřeba si všimnout, že všechny ohvězdičkované trojice jsou násobky 3, takže přinejhorším nějakou vložíme na konec.

Na začátku byla povinně trojice `101` s ciferným součtem 2. V každé iteraci velké závorky musela být zase trojice `101` a navíc buďto `0` nebo `202`. První varianta měla ciferný součet 2, druhá 6.

Krátkým rozborem případů pak došlo na to, že nejkratší násobek 3 vyhovující regexu je `10101010101`. Přilepením `201` za něj pak vypadl kýžený nejmenší násobek 9.

Většina řešitelů obdržela téměř plný počet bodů, nejčastější chybou bylo opomenutí popisu, které stavy budou vstupní a které výstupní po převodu DKA na NKA. Obecně však byla vaše řešení hezká a bylo mi potěšením je opravovat.

Jan „Moskyto“ Matějka

23-4-1 Studenti a profesoři

Všichni, kdo se odvážili odevzdat řešení, argumentovali převoditelností na problém maximálního toku – kuchařka v tomhle směru napovídala dost jasně. Nikdo pod devět bodů nedostal (vždyť také nikdo celé řešení pouhým poukazem do mého textu neodbyl), ve zbývajícím rozsahu jsem tak hodnotil úvahy o časové složitosti a nuance.

Je docela jasné, že si budeme uzpůsobovat první ze dvou zmíněných aplikací, která mluví o tom, jak pomocí toku najít maximální párování. Postavíme si ze zadání bipartitní graf, zorientujeme v něm hrany k profesorům, vrcholy studentů a profesorů pak napojíme na studentský zdroj a profesorský stok.

Protože chceme, aby měl student právě K profesorů, nastavíme váhu každé z hran ze studentského zdroje na K – to samé uděláme hranám do profesorského stoku, to aby měl každý profesor právě K studentů. Hranám uvnitř někdejšího bipartitního grafu nastavíme jedničky.

Povšimněme si tu, že kdyby zadání nezakazovalo, aby si některý student vybral profesora pro několik svých prací, vyrovnali bychom se s tím jednoduše – hraně, která by mezi příslušnými vrcholy vedla, bychom nastavili kapacitu na povolenou maximální násobnost.

Samozřejmě by ani nebyl problém mít rozdílný počet profesorů a studentů, či dokonce zavést individuální požadavky na počet vedených prací. Zadání bylo tak jednoduché předně proto, aby neděsilo.

Vraťme se k původní úloze. Na popsaný graf pustíme tokový algoritmus zachovávající celočíselnost a získáme z něj výsledek. Pokud není nalezený tok velký právě NK, řešení, které by každého plně uspokojilo, není. Pokud ano, vypíšeme páry profesor-student, jejichž hrana má jednotkový tok.

Důvod, že postup funguje, můžeme načrtnout třeba skrze fakt, že tok větší než NK v grafu existovat nemůže. Svědčí o tom řez na hranách mezi studentským zdrojem a studentskými vrcholy, kde je N hran, každá o kapacitě K.

Z toho vidíme, že pokud nám algoritmus vrátí takto velký tok, musí vést z každého studentského vrcholu k profesorům K jednotkových hran (a podobně ze strany profesorů), tedy jde o skutečné řešení našeho původního problému.

Zároveň se nemůže stát, aby postup řešení (maximální tok) nenašel a ono by existovalo – vždyť z každého řešení sestavíme tok o maximální velikosti.

Co časová složitost? Smířit se s tím, že má Edmondsův-Karpův algoritmus složitost $\mathcal{O}(M^2N)$, je přístup lenivý. Nicméně si můžeme všimnout, že zlepší-li každá cesta výsledek alespoň o jednotku, nenajdeme takových cest víc než KN.

Z toho plyne složitost $\mathcal{O}(KMN)$, což je lepší, protože pro $K > N$ úloha zřejmě není zajímavá.

Vysloveně akční přístup je začít se poohlížet po nekuchařkovém algoritmu. (To ale k získání maximálního počtu bodů potřeba nebylo.) Můžeme buď přemýšlet o tom, jestli není možné vzít Dinice či Goldberga a vzhledem k jisté speciálnosti našeho grafu vylepšit odhady časové složitosti, nebo zkusit najít specializovaný postup.

Vtip tkví v tom, že při zkoumání druhé možnosti nejspíše narazíme na Hopcroftův-Karpův algoritmus pro nalezení maximálního párování v bipartitním grafu běžící v čase $\mathcal{O}(M\sqrt{N})$, který je však jen dobře odhadnutý a přeříkaný Dinic.

My tu sice nechceme bipartitní párování, leč každé naše řešení (K-regulární bipartitní podgraf) se skládá z K takových disjunktních množin hran (1-regulárních bipartitních podgrafů). To není úplně vidět, ale je to hezká a užitečná pravda.

Můžeme tedy K-krát spustit Hopcrofta-Karpa a pokud nějaké řešení existuje, získáme ho v čase $\mathcal{O}(KM\sqrt{N})$. Pořád tak netrumfneme škálu rozličných moderních algoritmů pro hledání maximálního toku na obecném grafu, jde však o celkem srozumitelné a snadno naprogramovatelné řešení.

Lukáš Lánský

23-4-2 Paralelní profesoři

Tuto úlohu se pokoušelo vyřešit jen 8 z vás a k mému zklamání jen jedno řešení bylo úplně správně. Gratulace patří Vojtěchu Hlávkovi.

Nejčastější chybou bylo, že jste úlohu vyřešili pro $N = 2^k$ a zobecnili pro všechna N. Proč je tato úvaha špatná, je dobře vidět například pro $N = 3$.

Jak to tedy mělo být? Pokud $N = 2^k$, tak v prvním kroku profesory rozdělíme do dvojic a tím získáme dvojice profesorů se stejnými informacemi. Ve druhém kroku k sobě posadíme různé dvojice profesorů a tím získáme čtveřice profesorů se stejnými informacemi atd.

Až se dostaneme k jedné skupince o velikosti 2^k. Bude nám tedy stačit $\log_2 N$ sezení. Problém s dělením nikde nenastane, počet skupinek bude vždy sudý.

Pro jiné počty profesorů ale tento algoritmus aplikovat nemůžeme, protože v nějakém kroku dostaneme lichý počet skupinek a ten už neumíme jednoduše spárovat.

Úlohu vyřešíme zvlášť pro sudá a lichá čísla. U lichých čísel si můžeme všimnout, že při každém sezení bude alespoň jeden z profesorů lichý. Spočítáme si tedy, za kolik nejméně sezení se můžou všichni profesoři dozvědět informaci od toho, který byl lichý při prvním sezení.

Po prvním sezení ví onu informaci pouze on sám a po každém dalším sezení se množství profesorů se znalostí této informace může maximálně zdvojnásobit. Z toho vyplývá, že všichni profesoři můžou tuto informaci znát nejdřív po $\lceil \log_2 N \rceil + 1$ sezeních.[35]

Když najdeme obecný algoritmus pro lichá čísla, který řeší úlohu v $\lceil \log_2 N \rceil + 1$ krocích, tak máme vyhráno. Každé číslo si můžeme napsat jako $2^k + l$, kde k a l jsou přirozená čísla a k je nejvyšší možné.

[35] $\lceil x \rceil$ je takzvaná horní celá část, nejmenší celé číslo větší nebo rovné x.

Pak při prvním sezení sprárujeme l profesorů s některými z 2^k, těchto 2^k už umíme vyřešit v k krocích a nakonec opět zbylých l profesorů spárujeme s některými z 2^k.

Situaci se nám tedy povedlo vyřešit na

$$k + 2 = \lfloor \log_2 N \rfloor + 2 = \lceil \log_2 N \rceil + 1 \text{ kroků},$$

a to jsme chtěli.

Zbývají jen sudá čísla. Obdobně jako u lichých čísel ukážeme, že minimální nutný počet sezení je $\lceil \log_2 N \rceil$.

Profesory očíslujeme $0, \ldots, N-1$. První sezení spárujeme $(0, 1), (2, 3), \ldots, (N - 2, N - 1)$, tím každý zná dvě informace.

Pro druhé sezení vytvoříme dvojice $(0, 3), (2, 5), (4, 7), \ldots$ Tím všichni profesoři se sudým číslem s mají informace $s \ldots (s + 3) \bmod N$, po k-tém sezení mají analogicky informace $s \ldots (s + 2^k - 1) \bmod N$.

Lichá čísla jsou k sudým párována symetricky, takže až sudí budou znát vše, tak i liší. Celkem nám tedy bude stačit $\lceil \log_2 N \rceil$ sezení.

Obecně N je tedy optimální počet sezení

$$\lceil \log_2 N \rceil + (N \bmod 2).$$

Jedinou výjimku tvoří $N = 1$, kde nepotřebujeme žádné sezení.

Karel Tesař

23-4-3 Zabugovaný program

Dva zlatokopové, neboli ve známější verzi loupežníci, zvolili hladový algoritmus. Předložený program setřídil vstupní hodnoty a potom je hladově rozdělil mezi zlatokopy.

Hladově, to znamená tak, že se podíval, který z nich má zrovna méně, a tomu nuget přidělil. Začínal od největšího, skončil nejmenším.

Rychle jste odhalili, že potřebujete najít *false negative*, tedy vstup, u kterého program nenalezne správné řešení, byť by existovalo. Když totiž program ohlásí řešení, je zjevně správně.

Nejmenší vstup, na kterém se program zachoval chybně, byl `3 3 2 2 2`, kde bylo správným řešením dát jednomu ze zlatokopů `3 3` a druhému `2 2 2`. Program si nicméně tvrdošíjně mlel svou a po rozdělení `3 2 2` a `3 2` prohlásil, že řešení neexistuje.

Vstup byl nejmenší co do počtu nugetů. Řešitelé, kteří to dokázali, získali body navíc.

Důkaz byl docela jednoduchý rozbor případů. Vstup s jedním nugetem nemá řešení. Vstup se dvěma nugety a_1, a_2 může mít řešení jen pro $a_1 = a_2$, což náš program najde. Vstup se třemi nugety $a_1 \leq a_2 \leq a_3$ může mít řešení jedině pro $a_1 + a_2 = a_3$, což náš program zase bez problému najde.

V případě čtyř nugetů na vstupu ($a_1 \leq a_2 \leq a_3 \leq a_4$) bylo potřeba vyřešit několik možných případů. Program vždycky rozdělil nugety a_1 a a_2 na dvě různé hromádky. Kdyby platilo $a_1 = a_2$, muselo by také platit $a_3 = a_4$, jinak by řešení neexistovalo (dokažte za domácí úkol). V takovém případě ale náš program funguje.

Tudíž $a_1 > a_2$, a tedy náš program dá a_3 na hromádku k a_2. Nakonec odloží a_4 na menší z obou hromádek. Pokud platí $a_4 = |a_1 - a_2 - a_3|$, program vydá správné řešení; rozmyslete si, že to platí vždy.

Základem důkazu je na tomto místě úvaha, za jakých podmínek by ve všech správných řešeních musely být a_1 a a_2 nebo a_1 a a_3 na společných hromádkách, nebo a_2 a a_3 na různých.

Vstup byl i nejmenší co do celkové hodnoty. Za důkaz jsme taktéž udělovali bonusové body. Taktéž byl nejrozumnějším přístupem rozbor případů.

Někteří řešitelé si nevšimli, že program bral vstupní hodnoty sestupně. Pythoní kód to řešil metodou `pop`, která odebírá z konce seznamu, program v C třídil obráceně a procházel pole od začátku.

Za řešení s touto chybou jsme udělovali 2 body. Jeden za správnost, druhý za nápad, jak si úlohu výrazně zjednodušit (nejmenším protipříkladem by byl vstup `1 1 2`).

Martin „Medvěd“ Mareš & Jan „Moskyto“ Matějka

23-4-4 Závorky v TEXu

Nejprve se podívejme na rozpoznávání správného uzávorkování. Řetězec se závorkami `{` a `}` budeme procházet zleva doprava a počítat si, kolik neuzavřených levých závorek nám zbývá (tento počet označme k). Mohou nastat pouze dva případy znamenající, že řetězec není správně uzávorkovaný:

- k je 0 a přečteme uzavírací závorku (počet neuzavřených klesne pod nulu),
- k bude na konci řetězce větší než 0.

Jak poznat, že lze řetězec změnit na správně uzávorkovaný pouhými změnami znaku? Je zřejmé, že pro lichý počet to učinit nelze a pro sudý naopak vždy lze, protože můžeme jednoduše změnit všechny znaky na řetězec `{}{}{}{}`...

Nyní přejděme k algoritmu, který řeší naši úlohu a zajišťuje minimální počet změn znaků. Stejně jako při rozpoznávání, jestli je uzávorkování správné, budeme procházet závorky zleva doprava a počítat si neuzavřené levé.

Když k klesne pod 0 na pozici i, musíme změnit nějakou uzavírací závorku na pozici menší nebo rovno i (na pozici větší než i už to nepomůže). Je celkem jedno kterou, jde nám jen o počet změn. Také nesmíme zapomenout aktualizovat počet neuzavřených závorek (z -1 na 1).

Po přečtení poslední závorky mohou nastat 3 případy:

- k je 0 – pak už máme správně uzávorkovaný řetězec a vypíšeme počet dosud provedených změn,
- k je liché – pak i celkový počet závorek je lichý a řetězec nelze správně uzávorkovat,
- k je sudé – musíme tedy nějaké otevírací závorky změnit na uzavírací a nesmí to být libovolné, protože bychom mohli dostat špatně uzávorkovaný řetězec (při čtení zleva by klesl počet neuzavřených levých závorek pod 0).

Určitě nic nepokazíme, pokud budeme otevírací závorky měnit zprava. Počet změn je $k/2$ (každou změnou klesne počet neuzavřených levých závorek o 2).

Časová složitost je zjevně lineární a lépe to nejde (musíme se podívat na každou závorku). Část řešitelů prohlásila i paměťovou složitost za lineární, což kupodivu jde zlepšit. Kdo četl zadání pozorně, všiml si, že úkolem bylo najít pouze počet změn.

Stačilo tedy číst znaky ze vstupu (např. z obrovského souboru) a vůbec je neukládat, což dává konstantní paměťovou složitost. Kdo chtěl vracet správně uzávorkovaný řetězec, musel si pamatovat alespoň část řetězce od poslední změny znaku `}` na `{` nebo od posledního nulového počtu neuzavřených levých závorek, takže nejhůře celý řetězec.

Možná se zcela správně ptáte, proč náš algoritmus dává minimální počet změn. Je zřejmé, že pro správně uzávorkovaný řetězec vypíše 0. Pro špatně uzávorkovaný žádnou ze změn provedených při kontrole, jestli k nekleslo pod 0, nemůžeme vrátit. Na konci také musíme změnit nějakých $k/2$ otevíracích závorek.

Navíc nelze na žádné pozici provést změnu z `{` na `}` a potom zpět na `{` (tj. obě změny by byly zbytečné), protože by nám opět kleslo k na té pozici pod 0. Algoritmus tedy dává minimální počet změn.

Program (Python):
`http://ksp.mff.cuni.cz/viz/23-4-4.py`

Pavel „Paulie“ Veselý

23-4-5 Palindromnásobky

Zkusme řešit jednoduše – projdeme všechna čísla délky D dělitelná K a započítáme ta z nich, která jsou palindromem. Časová složitost tohoto řešení je $\mathcal{O}(D{\cdot}10^D/K)$, protože čísel, která testujeme, je $\mathcal{O}(10^D/K)$ a pro otestování, zda je číslo palindromem, musíme projít všech jeho D číslic.

Co takhle zkusit to naopak, procházet všechny palindromy a určit, které z nich jsou dělitelné K? Palindromy projdeme tak, že začneme nejmenším z nich (jeho první a poslední číslice jsou 1, všechny ostatní 0) a vezmeme první polovinu jeho číslic, začínajíce od největšího řádu a včetně prostřední číslice v případě lichého D.

Toto číslo zvětšíme o jedna a zrcadlíme zpět, abychom získali palindrom odpovídající délky. Tedy například $13931 \to 139 \to 140 \to 14041$. Stejný postup opakujeme, dokud se nedostaneme k číslu obsahujícímu samé devítky, čímž jsme u konce.

Abychom nemuseli v každém kroku palindrom půlit a pak zase zrcadlit zpátky, můžeme pracovat přímo s palindromem, jenom začneme uprostřed a případný přenos šíříme na obě strany. Takto dostaneme časovou složitost $\mathcal{O}(10^{D/2})$.

Exponenciální časové složitosti jsou ale hodně ošklivé. Copak tahle úloha nejde vyřešit v (pseudo-)polynomiálním čase?

Jistěže jde, jenom je potřeba se trochu zamyslet. Každý palindrom můžeme jednoznačně rozložit na $\lceil D/2 \rceil$ podpalindromů stejné délky jako celý palindrom tak, že i-tý podpalindrom má nenulové cifry pouze na i-té pozici od začátku a i-té pozici od konce. Tyto podpalindromy mohou mít, na rozdíl od běžných palindromů, nuly na začátku. Například 10301 rozložíme na 10001, 00000 a 00300.

Jak tohoto rozkladu využijeme? Vytvoříme tabulku zbytků – pro každou možnou hodnotu zbytku po dělení číslem K (tedy pro čísla 0 až $K-1$) si budeme pamatovat, kolika různými způsoby umíme vytvořit palindrom s daným zbytkem.

V prvním kroku projdeme podpalindromy, které mají nenulovou číslici na prvním (a tedy i na posledním) místě. Pro každý z nich určíme jejich zbytek a tyto počty si poznamenáme.

Ve druhém (a obdobně v každém dalším) kroku postupujeme tak, že nejdříve vytvoříme novou tabulku zbytků zkopírováním té staré, protože všechny palindromy, které jsme uměli vytvořit v předchozím kroku, umíme vytvořit stále (rozklad takového palindromu by měl na odpovídajícím místě podpalindrom ze samých nul).

Dále projdeme podpalindromy, které mají nenulovou číslici na druhém (a tedy i na předposledním) místě. Pokud má podpalindrom zbytek r, přičteme do nové tabulky hodnoty ze staré, cyklicky posunuté o r míst. To proto, že pokud jsme v předchozím kroku uměli vytvořit n palindromů se zbytkem q, umíme s využitím aktuálního podpalindromu vytvořit n nových palindromů se zbytkem $(q+r) \bmod K$.

Po posledním kroku takto získáme v závěrečné tabulce zbytků na pozici 0 počet palindromů délky D, které mají zbytek po dělení číslem K rovný nule, což je přesně to, co jsme chtěli. Vzhledem k tomu, jak s palindromy a podpalindromy pracujeme (a s využitím předpočítaných zbytků mocnin desítky) si je dokonce ani nemusíme pamatovat celé, stačí vždy jejich zbytek po dělení K.

Celková časová složitost tohoto algoritmu je $\mathcal{O}(D \cdot 10 \cdot K)$, protože pro každý podpalindrom, kterých je $10-1$ v každé z $\lceil D/2 \rceil$ skupin, přičítáme K hodnot do tabulky zbytků (kromě podpalindromů s první číslicí nenulovou, které jsou jednodušší). Možná by vás mohlo zarazit použití konstanty 10 v časové složitosti. Pokud bychom chtěli stejnou úlohu řešit v jiné soustavě, než je desítková, nahradili bychom toto číslo základem dané soustavy. Pokud ale nad takovou možností neuvažujeme, můžeme časovou složitost zapsat jako $\mathcal{O}(DK)$.

Vzhledem k tomu, že používáme předpočítané zbytky mocnin desítky, a vzhledem k tomu, že v každém kroku nám stačí dvě tabulky zbytků (aktuální a z předešlého kroku), je paměťová složitost $\mathcal{O}(D+K)$.

A ještě jeden dodatek na konec – zadané limity byly takové, že výsledek mohl být tak velký, že se nevešel do 32-bitového integeru, takže pro získání plného počtu

bodů bylo potřeba použít 64-bitový integer.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-4-5.c`

Petr Onderka

23-4-6 Knuthovy cesty po státech

Zkusme postupovat tak, že budeme následovat Knuthovu cestu a na každé křižovatce si pamatovat, kam až sahá nejdelší úsek cesty, který se nekříží a zároveň končí tam, kde zrovna jsme. Z takovýchto úseků pak vezmeme nejdelší a jsme hotovi.

Nyní předpokládejme, že známe nejdelší nekřížící se úsek končící i-tou křižovatkou (jeho začátek označme Z) a chceme najít takovou část cesty pro další křižovatku (nechť je to křižovatka K). Tam nám mohou nastat 2 případy:

1) Křižovatku K jsme navštívili před Z (popř. jsme ji nenavštívili vůbec). Pak můžeme nejdelší nekřížící se úsek cesty prodloužit o křižovatku K.
2) Křižovatku K jsme navštívili během nejdelší nekřížící se cesty pro i-tou křižovatku. Pak nastavíme nový začátek Z hned za minulou návštěvu křižovatky K a pokračujeme dál.

Zřejmě pokud bychom prodloužili aktuální cestu o jednu křižovatku zpět, dostali bychom se na nějakou podruhé, a tedy v každém kroku je nalezený úsek nejdelší nekřížící se.

Na to, aby výše uvedený postup fungoval efektivně, budeme potřebovat vědět, kdy jsme naposledy jakou křižovatku navštívili. To se udělá snadno pomocí pole o velikosti počtu křižovatek, kde si budeme příslušnou informaci udržovat.

Časová složitost je lineární a paměťová také.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/23-4-6.pas`

Pavel Čížek

23-4-7 Bratrstvo Seda a Grepa

Omluva na začátek. Formulace některých úkolů byly vágní a umožňovaly různé interpretace. Přesto jste je pochopili převážně tak, jak jsem je původně myslel.

Řešení **úkolu 1** bylo správně u všech, kdo jej poslali.

```
s/[ \t\r]+$//
```

Jeden výtečník zapomněl na dolar a místo něj použil chybné `\n`, za což byl nepatrně ztrestán (`sed` čte vstup po řádcích, takže `\n` na vstupu defaultně nikdy není). Hezké bonusové řešení předvedl Vojta Hlávka:

```
s/[[:space:]]*$//
```

Druhý úkol byl potvorný. Jedním ze správných řešení bylo napsat regex

```
s/(([ ~][kkOoSsUuVvZzIiA])|([[:punct:]][[:space:]~]?a)) /\1~/g
```

a prohlásit, že jej spustíme dvakrát. Proč? Poprvé ovlnkuje liché, podruhé sudé výskyty. Vstup „`..., a i s nimi`“ se tedy nejprve změní na „`..., a~i s~nimi`“, aby po druhém průchodu přibyla i prostřední vlnka a vznikl kýžený výsledek „`..., a~i~s~nimi`“.

Druhá správná varianta se dala vymyslet s manuálem GNU `sed`u v ruce, kde jste se mohli dočíst o `\b`, což je *předpoklad nulové délky* s významem „hranice slova“.

Jinak řečeno, když `sed` přijde k `\b`, tak se podívá, jestli je na hranici slova. Pokud ano, pokračuje dál, jinak se vrátí a zkusí jinou variantu. Druhé možné řešení tedy bylo

```
s/((\b[kkOoSsUuVvZzIiA])|([[:punct:]][[:space:]~]?a)) /\1~/g
```

které už stačilo spustit jednou.

Většina řešitelů si nevšimla buďto problému s ovlnkováním řetězu předložek, nebo zapomněli na variantu se spojkou `a` a interpunkcí apod. Toleroval jsem neúplný výčet interpunkce i chybějící předložky v seznamu, nicméně v praktickém použití si na to dejte pozor.

Úkol 3 byl zadaný vágně. Nebyla totiž definována abeceda, nad kterou se problém řeší. Většina řešitelů předpokládala, že bude rozumná a nebude obsahovat speciální znaky. Za takových podmínek byl problém řešitelný.

Nabízí se řešení přímočaré, leč chybné:

```
egrep '^s(.).+\1.*\1g?$'
```

Vyhovuje mu totiž například i řetězec `saaaaaa`. Jak tomu předejít? Vykutálený trik některých řešitelů `[^\1]` nefunguje (`\1` se totiž uvnitř hranatic interpretuje jako dvojice znaků `\` a `1`).

Nuže, přilepíme k prvnímu regexu filtr, který zahodí nevyhovující řetězce (obsahují moc oddělovačů).

```
| egrep -v '^s(.).*\1.*\1.*\1.*$'
```

To je sice hezké, ale tentokrát neprojde `sgaggg`. Musíme oddělit `g`. Zde je kompletní správné řešení.

```
egrep '^s(.).+\1.*\1g?$' |
egrep -v '^s(([^g]).*\1.*\1.*\1.*|g[^g]*g[^g]*g([^g]+|g.+))' |
egrep -v '^s\1\1.*'
```

První regex vytáhne kandidáty, druhý z nich vyháže ty, které mají nadbytečný počet oddělovačů (dobře si prohlédněte druhou větev, ve které se řeší `g`) a třetí ještě smaže ty, které mají prázdný regex k vyhledání, neboť ty také nejsou validní.

Ještě by se dalo připustit v abecedě zpětná lomítka. S tím se úspěšně porval jeden člověk.

Jakub Zíka se pustil do složitějšího rozboru případu, kdy uvažoval obecnější abecedu, která by mohla obsahovat speciální znaky. Právem mu náleží dva bonusové body.

Pokud by abeceda obsahovala kulaté závorky, nemůžeme ani zkontrolovat jejich správné vnoření, ani zkontrolovat validitu backreferencí a v tom případě je úkol zadanými prostředky neřešitelný.

Třešničkou na dortu pak byl **úkol 4**. Objevil se nápad počítat si po jedné, dokud nedojdeme k menšímu ze zadaných čísel (a vypsat pak to druhé). Má to však jeden háček – napsat sčítačku je možná o něco těžší než tento úkol samotný...

Autorské řešení spočívalo v porovnání řetězců nejprve podle délky, pak už bylo přímočaré.

```
s/([01]+) ([01]+)/\1#\2#\1#\2/
s/#[01]([01]*)#[01]([01]*)$/#\1#\2/
s/([01]+)#([01]+)##[01]+/\2/
s/([01]+)#([01]+)#[01]+#$/\1/
s/([01]+)1([01]*)#\1[0]([01]*)##$/\11\2/
s/([01]+)0([01]*)#\1[1]([01]*)##$/\11\3/
```

První řádek zamezí vícenásobnému startu (změna oddělovače) a připraví půdu pro porovnání podle délky – vytvoří kopie, ze kterých budeme usekávat číslice.

Druhý řádek usekne z kopie vstupu po číslici. Třetí a čtvrtý řádek ošetřují případ, kdy je jedno číslo kratší než druhé.

Na pátém a šestém řádku jsme zjistili, že jsou čísla stejně dlouhá, takže z nich vybereme to větší a vypíšeme.

Kdyby mohly být na začátku čísel nuly, stačilo by doplnit na vhodná místa `0*`.

Ještě nabízím variantní řešení se sčítačkou.

```
s/^([01]+) ([01]+)$/\1-\2@0/
s/@([01]*)0$/#\11/
s/@([01]*)01(1*)$/:\110\2/
s/:([01]*0+)1(1*)$/:\10\2/
s/:([01]*0+)$/#\1/
s/^([01]+)-([01]+)#\1$/\2/
s/^([01]+)-([01]+)#\2$/\1/
s/#/@/
```

První řádek je vstupní, druhý řádek přičítá k sudému číslu, třetí až pátý k lichému. Významy oddělovačů jsou snad jasné. Pro ještě nezvýšené číslo používáme `@`, pro právě inkrementované číslo používáme `:` a pro hotové číslo k porovnání máme `#`.

Zde by se už hodila analýza časové složitosti. Předpokládejme, že vyhodnocení regexu trvá jednotkový čas. Reálný odhad to není a asi nikdy nebude, leč pro představu to stačí.

První řešení nejdřív postupně odsekává po číslici, což trvá N kroků (N číslic na vstupu). Porovnání stejně dlouhých čísel už proběhne v konstantním čase, takže složitost odsekávacího řešení je $\mathcal{O}(N)$ cyklů.

Druhé řešení počítá po jedné. Byť stráví sčítáním od 1 do K jen $\mathcal{O}(K)$ cyklů, je to pořád $\mathcal{O}(2^N)$, neboť K může být s $\mathcal{O}(N)$ číslicemi na vstupu až exponenciálně veliké.

Není všechno zlato, co se třpytí, aneb přičítání jedničky vypadá lákavě, leč jeho rychlost není závratná. Naopak zdánlivě chlupaté řešení se sekáním číslic je výrazně rychlejší a efektivnější.

Jan „Moskyto“ Matějka

23-5-1 Boj s nanoboty

Napřed si představíme jednodušší řešení. Podívejme se na problém jako na trojrozměrný svět (dva prostorové rozměry a jeden čas). Nebo pokud nemáte časoprostorovou představivost, zkuste si představit hromadu 2D-světů nad sebou (dole je v čase 0, nad ním v čase 1 atd.).

A v těchto světech budeme ukládat, na která všechna políčka se hrdina mohl dostat a kolik živých lidí již mohl mít na svědomí, pokud by nyní pobýval na tomto políčku. Tedy v nultém světě (v tom přímo zadaném) se může nacházet pouze na jednom políčku ($(0,0)$) a nemá na kontě nikoho (předpokládejme, že padouch začíná hrát nejdříve v čase 1).

Jak spočítáme novější verzi našeho světa? Z každého políčka, kde se mohl nacházet, ho zkopírujeme do stejného a všech sousedních políček. Je-li na nově obsazeném políčku zrovna na potvoru padouch, tak ho zamorduje a my si přičteme skóre.

Pokud máme možnost nakopírovat hrdinu z více políček, samozřejmě si vybereme to s nejlepším skóre (to, jak se dostal na toto políčko, již neovlivní budoucnost a záchranou více lidí si celkově pomůže, nikdy si nemůže uškodit).

Ke zrekonstruování výsledku si ke každému možnému výskytu hrdiny také potřebujeme poznamenat, které bitvy s padouchem se účastnil předtím.

Po spočítání všech pater stačí jen najít výskyt s nejvyšším skóre a prohrabat se zpětně bitvami, které podstoupil.

Toto by samozřejmě fungovalo, ale je to pomalé. Můžeme si ale všimnout, že většinu času trávíme sledováním bloumání hrdiny po okolí. Nás však zajímá, jen jestli se včas dostaví na rande, ne kterou cestu k tomu zvolil. Taktéž, není zajímavé, kde tráví přebytečný čas (čekat může kdekoliv).

Takže se omezíme pouze na zajímavé události. Všimněme si také, že pro zjištění, jak dlouho bude cesta trvat, stačí jen sečíst vzdálenosti míst v obou souřadnicích, tedy pokud jsou sousední jen do stran, nahoru a dolů. Kdybychom uvažovali i diagonální sousedy, pak by to bylo maximum z těchto vzdáleností.

Tak tedy, seřaďme si vypuštění nanobotů chronologicky, od nejbližšího v budoucnosti po nejvzdálenější. Pro každou událost se podíváme, ze kterých všech střetů se to sem dá stihnout. Z nich vybereme ten, který má nejlepší skóre, a uložíme si jej.

Pro jednoduchost považujme narození hrdiny také za střet. Nakonec vybereme událost, po které měl největší skóre, a stejným způsobem jako v předchozím řešení ji odmotáme k začátku.

Složitosti jsou jednoduché – pamatujeme si všechny události, tedy paměťová je lineární. A pro každou událost si prohlížíme všechny předchozí, což je $1+2+3+\ldots+n$, z čehož nám vyjde složitost kvadratická.

A proč to vlastně funguje? Využíváme pozorování zmíněné v prvním řešení – že pokud se už hrdina někde vyskytuje, tak na budoucnost již nemá vliv, jak se

tam dostal, proto je pro nás nejvýhodnější, aby se na takovém místě vyskytoval s nejvyšším možným skóre.[36]

Z toho indukcí dokážeme, že po každé události by měl maximální možné skóre, kdyby se jí účastnil. U narození je to jasné a u každé další to odvodíme z toho, že jsme si vybrali tu nejlepší předchozí událost.

Program (C++):
`http://ksp.mff.cuni.cz/viz/23-5-1.cpp` *Michal „Vorner“ Vaner*

23-5-2 Zjednodušení situace

Tuto úlohu – dělení množin bodů jednou přímkou na poloviny – bylo těžké nejen vyřešit, ale také zadat do CodExu.

Stojí za zmínku, že v mnoha (i náhodných) případech se dá použít rozumných heuristik (setřídit podle nějaké osy, zkusit najít řešení, případně zvolit jinou osu a iterovat) a řešení je pak stejně rychlé jako varianta optimálního řešení. Děkujeme všem, kteří upozornili na tuto slabinu původního zadání.

Klíčové pozorování k řešení naší úlohy je, že si vlastně můžeme představit, že ona oddělující přímka prochází dvěma body vstupu – pokud bychom našli nějakou, která toto nesplňuje, můžeme ji nejdřív posunout a pak pootočit tak, aby již tento invariant splňovala.

Obtíž máme jen s tím, že zadání příkladu tuto situaci zakazuje – vyřešíme to tedy tak, že najdeme řešení s body na dělící přímce a pak jen přímku o kousek pootočíme správným směrem a posuneme.

Určitě umíme vyřešit problém v čase $\mathcal{O}(n^3)$ – pro každou dvojici bodů ze zadání existuje právě jedna přímka, která jimi prochází, a pro tuto přímku v lineárním čase snadno zkontrolujeme, je-li to ta hledaná, či nikoli.

Pro řešení v lepším čase než kubickém použijeme klasický geometrický trik – kubické řešení často zahazuje mezivýsledky, avšak my si pro nějakou množinu bodů umíme existenci řešení najít rychleji než pro každý zvlášť. Nejen v tomto případě budeme hledat všechny možné dělící přimky, které prochází jedním bodem.

Vezmeme si bod a setřídíme si okolní body podle směrnice. Představit si to můžeme tak, že máme náš bod uprostřed a kolem něj točíme postupně onu dělící přímku.

Pak procházejme ostatní body podle pořadí, které nám určilo setřídění. Pro první bod si spočítáme počet vojáků na obou stranách klasicky, v lineárním čase. Každý další bod už ale umíme zpracovat v konstantním čase – vždy přechází buďto „zprava doleva“ nebo „zleva doprava“ a podle toho přičteme a odečteme jedničku.

Takto umíme v čase $\mathcal{O}(n \log n)$ zkontrolovat všechny přímky, které mají jeden společný bod. Protože už víme, že naše hledané řešení obsahuje dva body ze vstupu, musíme po lineárním počtu kroků najít správné řešení. Časově jsme se dostali na $\mathcal{O}(n^2 \log n)$ a paměťově na $\mathcal{O}(n)$.

[36] Odborně se této vlastnosti problému – že menší kousek optimálního řešení je optimální řešení menšího problému – říká submodularita.

Pro úplnost vzorového řešení se ujistíme, že přímku lze vždy správně posunout, tedy že nejbližší celočíselný bod je od ní dostatečně daleko.

Mějme tedy přímku určenou dvěma body, můžeme předpokládat, že není vodorovná ani svislá, pro ně to platí jistě. Jeden z bodů si můžeme (posunutím osy) zadefinovat jako $(0, 0)$, ten druhý jako (k, l). Navíc o souřadnicích (k, l) můžeme tvrdit, že jsou nesoudělné, jinak bychom bez újmy na obecnosti volili jiný bod.

Představme si nyní naši přímku jako graf funkce $k \cdot x/l$. Vkládejme za x celá čísla, dostáváme hodnoty y-ové souřadnice. Pokud výsledek nebude celé číslo, jak daleko může být? Alespoň $1/l$, protože neumíme zvýšit jmenovatel. Pokud bychom otočili osy, získali bychom, že to musí být alespoň $1/k$.

Toto však není úplně přesné. Máme pravoúhlý trojúhelník, jehož odvěsny jsou dlouhé alespoň $1/k$ a $1/l$. Spočítáme-li nyní jeho přeponu Pythagorovou větou a následně výšku na přeponu z vzorce $S = ab = cv_c$, získáváme skutečnou vzdálenost mřížového bodu od přímky.

Pokud za k i l dosadím 100 000, výška vyjde alespoň $\frac{\sqrt{2}}{200\,000}$, což je stále bohatě v mezích přesnosti.

Zbývá otázka – je to optimální řešení? Ale kdepak! Tato úloha je celkem slavná, je to konkrétní varianta problému *sendviče se šunkou*, anglicky Ham sandwich problem.[37]

Pro náš rovinný případ jej lze řešit dokonce v lineárním čase, můžeme jej řešit dokonce i ve více dimenzích (tam bychom pak hledali nadroviny). Optimální řešení využívá principu „Rozděl a panuj“ tak, že v každem kroku vyhodí lineárně mnoho kandidátů na dělící přímku a pokračuje dále.

Algoritmus je to však poněkud netriviální a pracuje s duální verzí problému (tedy hledá bod, který leží nad i pod právě polovinou přímek), takže jej tady neuvádíme. Možná se k němu dostaneme někdy příště.

Jste-li netrpěliví, můžete si oprášit angličtinu a najít si odborný článek „Algorithms for Ham-Sandwich Cuts“ od autorů Lo, Matoušek a Steiger. Nepodaří-li se vám jej získat, napište nám, zařídíme.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-5-2.c`

Martin Böhm & CodEx

23-5-3 Hra pro jednoho hráče

Hanojské věže jsou klasickým příkladem na rekurzi. Máme dané kotouče $n \ldots 1$, $n \geq 2$ a chceme je přesunout z tyče A na tyč C za pomoci tyče B. Postup vypadá takto:

1. kotouče $(n-1) \ldots 1$ přesuneme z tyče A na tyč B,
2. nic nám teď nebrání přesunout kotouč n z tyče A na tyč C,
3. kotouče $(n-1) \ldots 1$ z tyče B položíme na tyč C.

[37] `http://en.wikipedia.org/wiki/Ham_sandwich_theorem`

Programem to samé počítači vysvětlíme skoro stejně:

```
def hanoj(n, zdroj, pom, cil):
    if n != 1:
        hanoj(n-1, zdroj, cil, pom)
    print(str(n) + ": " + zdroj + "->" + cil)
    if n != 1:
        hanoj(n-1, pom, zdroj, cil)

hanoj(3, "A", "B", "C")
```

Časová složitost je exponenciální vůči n a lineární vzhledem k velikosti výstupu, což je to nejlepší, v co jsme mohli doufat. Pokynů k přesunu kotouče bude $2^n - 1$ – to lze z algoritmu dokázat indukcí, vždyť $2(2^{n-1} - 1) + 1 = 2^n - 1$.

Kód si pro řešení našeho zadání můžeme docela snadno upravit tak, aby sledoval stav hry a odpočítával tahy. Až zjistíme, že jsme v kýženém tahu, prostě jen stav vytiskneme.

```
def hanoj(n, zdroj, pom, cil, k, kyzeneK, stav):
    if n != 1 and kyzeneK < k + 2**(n-1) - 1:
        hanoj(n-1, zdroj, cil, pom, k, kyzeneK, stav)
    k += 2**(n-1) - 1

    if k == kyzeneK: print(stav)
    stav[cil].append(stav[zdroj].pop())

    k += 1

    if n != 1 and kyzeneK > k - 1:
        hanoj(n-1, pom, zdroj, cil, k, kyzeneK, stav)
```

Algoritmus funguje, ale má stále časovou složitost $\mathcal{O}(2^n)$, přičemž v tomto případě to už výstupem neomluvíme – ten je lineární.

Lineární algoritmus existuje – stačí si uvědomit, že při rekurzivním procházení nikdy nepotřebujeme volat funkci `hanoj` dvakrát. Když totiž přesně víme, kolik tahů které volání udělá, umíme určit, jestli se k-tá pozice vyskytuje až po přesunu kotouče, nebo ještě před ním.

Pak už si jen zjednodušíme práci tím, že stav hry nebudeme udržovat, ale budeme ho rovnou průběžně tisknout. Zde je výsledný algoritmus řešící problém v lineárním čase i prostoru.

```
def hanoj(n, zdroj, pom, cil, k, kyzeneK):
    if n == 0 : return

    if kyzeneK < k + 2**(n-1) :
        print(str(n) + " je na tyci " + zdroj)
        hanoj(n-1, zdroj, cil, pom, k, kyzeneK)

    if kyzeneK >= k + 2**(n-1) :
        print(str(n) + " je na tyci " + cil)
        hanoj(n-1, pom, zdroj, cil, k + 2**(n-1), kyzeneK)

hanoj(3, "A", "B", "C", 0, 3)
```

Rozmyslete si, nečiní-li nám problém užití pythonovského mocnění. Má program, jak je napsán, opravdu lineární složitost? Pokud ne, proč? Bylo by těžké to opravit? Další námět k zamyšlení – paměťová náročnost algoritmu, jak je implementován, je lineární. Jak ji srazit na konstantní?

Víceméně toho samého programu se šlo dobrat i zapřemýšlením nad tím, co má náš problém společného s počítáním n-té permutace.

Velká část řešitelů naměřila, že výraz

$$\left(\pm\frac{M}{2^N}+\frac{1}{2}\right) \bmod 3$$

při vhodném nastavení znaménka a zaokrouhlení (podle N) dává kýžené číslo tyče pro N-tý disk; M pak určuje číslo tahu. S různým úspěchem se řešitelé snažili tuto skutečnost využít ve svém řešení, leč kamenem úrazu byla absence důkazu správnosti a časté drobné chyby.

Několik řešitelů používalo poznatek z Wikipedie o vztahu dvojkového zápisu čísla tahu a situace hry. Nic jsem proti tomu neměl, pokud autor prokázal kvalitním zdůvodněním, že rozumí, co se v postupu děje.

Lukáš Lánský

23-5-4 Model čtoucího řidiče

Priamo zo zadania vyplýva, že pre každú hranu, ktorou na križovatku prídeme, je jasne určená hrana, ktorou zase odídeme. Naopak, pre jednu výstupnú hranu môže byť viacero vstupných.

Na začiatku si očíslujeme hrany a z pôvodného grafu zostrojíme štruktúru, v ktorej budeme hľadať cesty. Táto štuktúra bude reprezentovaná ako pole hrán, kde každá hrana si pamätá svojho následovníka, a zoznam svojich predchodcov.

Štruktúru vytvoríme v čase $\mathcal{O}(M)$. Pre každý vrchol v pôvodnom grafe postupne prechádzame zoznam jeho hrán odzadu. Ak narazíme na výstupnú hranu, poznamenáme si ju ako hranu H. Ak narazíme na vstupnú hranu, nastavíme jej následovníka hranu H, a hrane H pridáme do zoznamu predchodcov spracovávanú vstupnú hranu.

Je zaujímavé si uvedomiť, že kým skoro každá hrana má nutne následovníka (okrem hrán vedúcich do vrcholu bez výstupných hrán), viacero hrán môže nemať predchodcu. Napríklad ak má vrchol tri výstupné hrany za sebou, na druhú a tretiu sa šofér nikdy z iného vrcholu nedostane. Ale môže nimi svoju cestu začať.

Po vytvorení štruktúry nastavíme každej hrane príznak, že ešte nebola spracovaná. Hrana si tiež pamätá, aký vrchol je jej začiatočný a aký je koncový (keďže je orientovaná).

Teraz začneme hľadať cesty. Na začiatku si vezmeme prvú hranu a prechádzaním dopredu si vytvárame spojový zoznam hrán tejto cesty. Keďže v pôvodnom grafe hľadáme cestu s N vrcholmi, a teda $N-1$ hranami, v našej štruktúre hrán to odpovedá ceste dĺžky $N-1$.

Cesta je teda reprezentovaná spojovým zoznamom hrán, ktoré používa, a držíme si ukazateľ na jej začiatok a koniec.

Vrcholy, ktorými cesta prechádza, si pamätáme pomocou poľa o veľkosti N, kde hodnota v poli na mieste i vyjadruje, koľkokrát daná cesta prechádza vrcholom i. Okrem toho si tiež musíme pamätať, koľko rôznych vrcholov sme navštívili.

Pri vytváraní cesty si vždy pri pridaní hrany zvýšime hodnotu v poli navštívených vrcholov. Dôležité je, že ak sa hodnota zmenila z 0 na 1, zvýšime počet rozdielnych navštívených vrcholov.

Po pridaní hrany si túto hranu označíme ako spracovanú. Takisto pridám jej cenu do celkovej ceny cesty.

Po vytvorení cesty skontrolujeme, koľko rôznych vrcholov sme navštívili. Ak ich je N a cena cesty je lepšia ako doteraz najlepšia nájdená, uložíme si cestu ako najlepšiu doteraz nájdenú. Keďže cesta je jednoznačne určená svojou začínajúcou hranou, stačí si uložiť len prvú hranu cesty.

Takže máme nájdenú nejakú prvú cestu. Je dôležité si uvedomiť, že táto cesta nemusí mať práve $N-1$ vrcholov. Mohla skončiť predčasne, ak jej koncová hrana už nemala následovníka. To však nevadí, ako uvidíme neskôr. Teraz pomocou tejto cesty nájdeme ďalšie cesty, a to prehľadávaním do hĺbky.

V každom kroku sa najprv pozrieme, či má začiatočná hrana cesty nejakého predchodcu, po ktorom sme sa ešte nevracali (berieme ich postupne v tom poradí, ako ich máme uložených).

Ak áno, posunieme cestu o jeden vrchol dozadu – pridáme predchodcu súčasnej začiatočnej hrany na začiatok cesty, pridáme hodnotu tejto hrany do celkovej ceny cesty, a zvýšime čítač v poli navštívených vrcholov u začiatočného vrcholu práve pridanej hrany. Taktiež upravíme hodnotu, koľko rôznych vrcholov sme navštívili (ak sa nám zmenila 0 na 1).

Ak je cesta dlhá $N-1$ hrán (čo nie je vždy, môže byť aj kratšia), zrušíme poslednú hranu na konci (na ktorú si držíme ukazateľ). Odčítame jej hodnotu z celkovej hodnoty cesty.

Tiež znížime čítač v poli navštívených vrcholov u koncového vrcholu hrany. Ak sa nám zmenila hodnota z 1 na 0, znížime počet rôznych vrcholov, ktoré sme navštívili. Ak je cesta kratšia ako $N-1$, koncovú hranu necháme, čím cestu o jedna predĺžime.

Naopak, pokiaľ sa cesta nemôže posunúť smerom dozadu, pokúsime sa ju posunúť smerom dopredu – posunieme začiatočnú hranu na jej následovníka, pričom upravíme počty navštívených vrcholov. Ak má koncová hrana následovníka, posunieme aj ju. Ak nemá, koniec neposúvame, a cesta sa nám proste skráti.

Pri posune dopredu si musíme dať pozor na cykly, po ktorých by sa cesta mohla posúvať teoreticky donekonečna.

Po posune cesty si také nové hrany, cez ktoré prejdeme, poznačíme ako spracované.

Na konci posunu sa pozrieme, koľko rôznych vrcholov sme navštívili. Ak je ich práve N a cena cesty je lepšia ako doteraz najlepšia, upravíme hodnotu najlepšej doteraz nájdenej cesty (konkrétne len začiatočnej hrany cesty) a jej cenu.

Ak sa nie je kam pohnúť, tak sme skončili spracovanie jedného súvislého úseku hrán. Úsekov ale môže byť viacero. Takže nájdeme úvodnú cestu v novom úseku.

To urobíme tak, že v zozname hrán nájdeme ešte nespracovanú hranu a z nej spustíme hľadanie úvodnej cesty. Zoznam však neprechádzame od začiatku, ale od miesta, kde sme skončili posledne, preto je zložitosť nájdenia všetkých začiatkov $\mathcal{O}(M)$.

Analýza celkovej zložitosti je následovná – vstup načítame v zložitosti $\mathcal{O}(M+N)$. Následovníkov a predchodcov hrán spočítame v $\mathcal{O}(M)$. Nájdenie všetkých začiatkov ciest je dokopy tiež $\mathcal{O}(M)$. Ostáva spočítať zložitosť hľadania ciest.

Posun cesty zvládneme v čase $\mathcal{O}(1)$. Koľko takýchto posunov bude? Keďže následovník každej hrany je maximálne jeden, je každá hrana predchodcom pre najviac jednu hranu. Celkový počet predchodcov je teda maximálne M.

Pri pohybe dozadu sa každá hrana vyskytne na začiatku cesty práve raz, pričom poradie určuje prechádzanie do hĺbky. Pohyb dopredu je jednoznačne určený pohybom dozadu – dopredu sa hýbeme len keď sa nedá hýbať dozadu, čím simulujem práve prehľadávanie do hĺbky.

Keď si to predstavíme, vidíme, že pri pohybe dopredu sa začiatok vracia po hranách, ktorými predtým prešiel dozadu. Každou hranou teda prejde maximálne dvakrát – raz dopredu a raz dozadu. Pri koncových hranách úseku je možné, že nimi prejde len raz, a to dopredu.

Keďže jeden posun cesty zaberie $\mathcal{O}(1)$, celková zložitosť nájdenia všetkých ciest bude $\mathcal{O}(M)$. Časová zložitosť algoritmu je teda $\mathcal{O}(M+N)$, pamäťová tiež $\mathcal{O}(M+N)$.

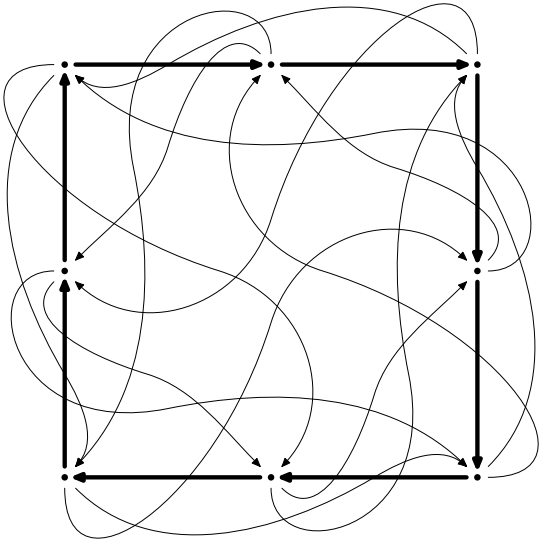
Väčšina riešiteľov úlohu vyriešila v čase $\mathcal{O}(MN)$. Táto jednoduchšia varianta spočíva v tom, že postupne berieme všetky hrany ako začiatky ciest a sledujeme, či cesta dlhá N vrcholov prechádza všetkými vrcholmi.

Pritom však nesmieme zabudnúť na to, že si následovníkov hrán treba predpočítať dopredu, aby sme ich dokázali určiť v konštantnom čase. Ak by sme pre nájdenie následovníka hrany zakaždým museli prechádzať celý zoznam hrán vo vrchole, časová zložitosť nám narastie na $\mathcal{O}(M^2)$.

Skonštruujeme si graf následovne. Na začiatku pridáme hrany tak, aby vznikla Hamiltonovská kružnica. Označme si tieto hrany „tučne“ V každom vrchole je teraz jedna vstupná a jedna výstupná hrana.

Medzi tieto hrany teraz v smere jazdy šoféra vložíme $(M-N)/2N$ vstupných hrán, a v opačnom smere $(M-N)/2N$ výstupných. Teraz je teda pôvodná tučná výstupná hrana následovník pre všetky nové vstupné hrany, a naopak, práve vložené výstupné hrany nemajú predchodcu.

Druhé konce vložených hrán zvolíme tak, aby vznikol korektný graf.



Analýza zložitosti hľadania ciest v tomto grafe je následovná. Pre každú hranu platí, že môže byť na ceste prvá až N-tá. Tučná hrana bude raz prvá a $M/2N$-krát druhá až N-tá, teda pre ňu budeme hľadať následovníka dokopy $\mathcal{O}(M)$-krát.

Nájdenie následovníka pre tučnú hranu trvá $\mathcal{O}(M/N)$ (musíme prejsť všetky umelo pridané vstupné hrany medzi ňou a výstupnou hranou). Takže na každej tučnej hrane spotrebujeme dokopy $\mathcal{O}(M^2/N)$ času.

Tučných hrán je $\mathcal{O}(N)$, teda na všetkých tučných hranách spotrebujeme celkovo $\mathcal{O}(M^2)$ času. Novopridané hrany už výsledok neovplyvnia.

Program (C):
`http://ksp.mff.cuni.cz/viz/23-5-4.c`

Mária Vámošová

23-5-5 Kuchařková

Našim úkolem je dokázat, že úloha Metr je NP-úplná. Jak nám kuchařka radila, je příliš pracné dokazovat úplnost tak, že převedeme na Metr všechny úlohy z NP. Raději tedy dokážeme, že lze jednu NP-úplnou úlohu vyřešit pomocí Metru.

Nejtěžší v NP-úplnostních převodech bývá rozpoznat, která úloha se nám bude převádět nejsnáz.

Na Metru stojí za všimnutí, že překládání samotného metru do pouzdra nám v jistém smyslu rozděluje úseky na dva typy – pokud jde metr uložit, tak jeden typ úseku je přeložen na jednu stranu (řekněme zprava doleva) a druhý je přeložený nazpátek (zleva doprava). Navíc je metr zadán jako posloupnost čísel.

Když se podíváme do seznamu NP-úplných úloh, najdeme tam úlohu Dva loupežníci, která také rozděluje čísla na dvě hromádky. Zkusme tedy pomocí Metru řešit Loupežníky.

Připomeňme si zadání Dvou loupežníků z kuchařky:

Název problému: Dva loupežníci

Vstup: Seznam nezáporných celých čísel.

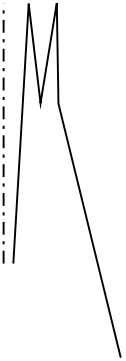
Problém: Existuje rozdělení seznamu na dvě hromádky tak, že každé číslo bude v právě jedné hromádce a v každé hromádce bude stejný součet čísel?

Začněme tedy převádět vstup Dvou loupežníků na vstup Metru. Vstup Loupežníků nám nijak neurčuje, jak velké má být pouzdro metru – to si tedy můžeme zvolit sami, aby se nám snáz převádělo.

Dopředu není úplně jasné, jaká velikost by se nám hodila. Bude nám stačit součet všech předmětů (označujme ho σ), nebo velikost jednoho lupu, $\sigma/2$? Méně než $\sigma/2$ nedává příliš smysl, ale více by mohlo…

Jak jsme diskutovali výše, mohlo by nám stačit označit ty části metru (tedy tu část kořisti), které jdou zleva doprava, jako lup pro loupežníka A a ty, které jdou zprava doleva, přiřadíme loupežníku B.

Nyní se zamysleme nad vstupy, které by nám mohly dělat neplechu. Například seznam předmětů `1 1 1` by se do pouzdra velikosti alespoň 1.5 snadno vešel, ale my musíme odpovědět `NE`, protože jej rozdělit pro dva loupežníky nelze.

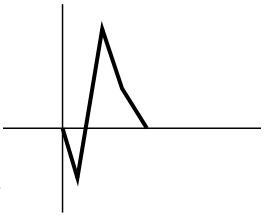


Mohli bychom tedy zkusit nastavit, aby začátek i konec lupu končil ve stejném bodě metru – například tak, že na začátek i konec přidáme úsek dlouhý jako celé pouzdro.

Tím by určitě odpadl případ `1 1 1`. Jak by taková úprava vstupu vypadala, vidíte na obrázku. Bohužel nám po chvíli úvah dojde, že by nám také odpadl případ `1 3 1 1`, který ovšem rozdělit jde.

Podívejme se na vstup `1 3 1 1` a zamysleme se, jak naši úvahu vylepšit. Na dalším obrázku jsme jej zakreslili tak, aby se uložení metru podobalo grafu funkce, který začíná a končí v nule.

Každé rozdělitelné zadání Dvou loupežníků jde takto nakreslit – prostě jednu část kresleme jako rostoucí úsečky a druhou jako klesající.



Můžeme tedy vhodnou úpravou našeho vstupu pro Loupežníky zajistit, aby řešení Metru přesně odpovídalo grafu takovéto funkce?

Ano, stačí jen trochu upravit nápad, který jsme měli před pár odstavci. Potřebujeme totiž v Metru povolit, abychom mohli vstoupit na grafu i do „záporných hodnot.“

Na začátek metru tedy vložme úsek o velikosti k, což bude také velikost pouzdra. Ten se dá do pouzdra vložit jen tak, že jeho konec bude na okraji pouzdra. Další

úsek si tedy také zvolme – tentokrát jako $k/2$. Z okraje pouzdra jsme se tedy dostali přesně doprostřed. To bude náš počátek grafu.

Dále už pokládejme úseky o velikosti stejné, jako byly hodnoty na vstupu Dvou loupežníků, a ve stejném pořadí. Abychom se ujistili, že na konci opravdu naše funkce skončí v nule, přidejme ještě jeden úsek délky $k/2$ a za něj úsek délky k.

Nyní už víme, co od k chceme – abychom neřekli zbytečné NE, pokud bychom neměli dostatečný rozsah na jejich poskládání. Bude nám stačit nastavit $k = \sigma$, ale klidně bychom mohli mít pouzdro i větší.

Převod je dokonán, pojďme si tedy ukázat, že je korektní.

Už během rozboru jsme si rozmysleli, že řešení Dvou loupežníků existuje právě tehdy, když existuje nakreslení lupu jako grafu funkce tak, že graf začíná i končí v nule.

V naší konstrukci platí, že metr lze vložit právě tehdy, když část odpovídající lupu loupežníků začíná a končí uprostřed pouzdra – a to platí právě tehdy, když existuje onen graf funkce začínající a končící v počátku.

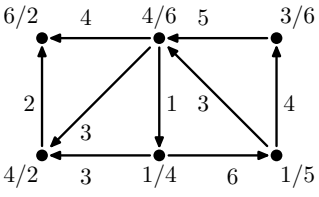
Složením těchto ekvivalencí dostaneme, že náš převod odpoví ANO na Metr právě tehdy, když problém Dva loupežníci šel vyřešit, a tedy je vše v pořádku – Metr je NP-těžký.

Pro formální správnost si ještě povězme, že rozdělení metru (informace o tom, kde metr začíná a v jakém směru jej zlomit) je polynomiálně velkým certifikátem k našemu problému, a Metr je tedy v NP. Obě tvrzení spojíme dohromady a dostáváme, že Metr je NP-úplný.

Martin Böhm

23-5-6 Předposlední

Největší problém celé úlohy je poznat, že se jedná o toky v sítích (něco o tocích si můžete přečíst v kuchařce ke 4. sérii). My si nyní tipneme, že se jedná o nějaký tok, a budeme se jej tam snažit najít. Jak na to?



Vstupní hrany a výstupní hrany jsou na sobě nezávislé v rámci vrcholu. Tak si každý vrchol rozdělíme na 2 nové vrcholy, levý a pravý.

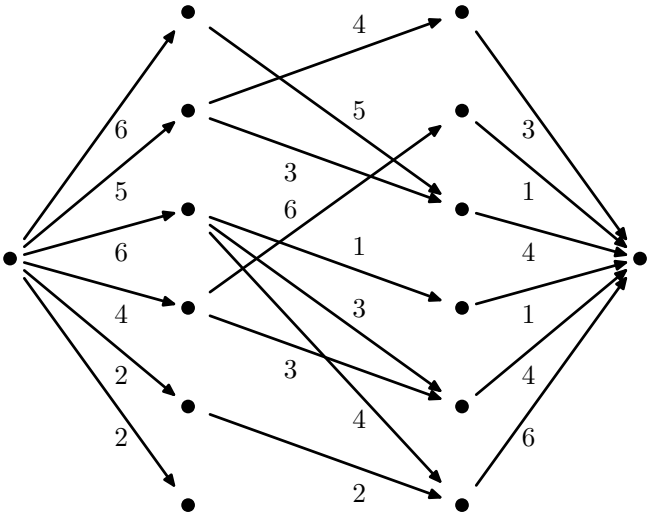
Levý nám bude reprezentovat výstupní část (z této části povedou všechny hrany) a pravý bude reprezentovat vstupní část (do tohoto vrcholu naopak povedou všechny hrany).

Není těžké nahlédnout, že jsme takto vytvořili orientovaný bipartitní graf, kde všechny hrany vedou z levé partity do pravé.

Nyní ještě potřebujeme zohlednit maximální vstupní součet a minimální výstupní součet. To uděláme tak, že do grafu přidáme další 2 vrcholy.

Jeden pojmenujeme zdroj a povede z něj hrana do každého vrcholu levé partity. Tyto hrany budou ohodnoceny maximálním výstupním součtem příslušných vrcholů.

Druhý pojmenujeme stok a z každého vrcholu pravé partity do něj povede hrana. Tyto hrany budou ohodnoceny minimálním vstupním součtem příslušných vrcholů.



Nyní máme ohodnocený orientovaný graf se zdrojem, stokem a celočíselnými kapacitami hran. Zavoláme tedy některý z algoritmů na hledání maximálního (celočíselného) toku, například Fordův-Fulkersonův algoritmus s hledáním zlepšujících cest pomocí prohledávání do šířky (viz kuchařku).

Pokud se velikost maximálního toku bude rovnat sumě minimálních výstupních součtů, tak jsme našli příslušné ohodnocení. Pokud ne, tak neexistuje žádné řešení.

Proč to funguje? Hrany ze zdroje do levé partity nám zajišťují, že se do grafu nikdy nedostanou takové hrany, které by porušovaly podmínku maximálního výstupního součtu. Hrany mezi partitami jsou přesně ty samé hrany jako hrany v původním grafu.

Hrany vedoucí z pravé partity do stoku nám obstarávají minimální vstupní součty a jejich kapacity jsou právě tyto hodnoty. Kdybychom totiž měli řešení, ve kterém by některý vstupní součet byl větší než daný minimální, tak pak můžeme tok hran vedoucích dovnitř libovolně snížit tak, aby jejich součet byl roven minimálnímu vstupnímu součtu a všechny podmínky zůstanou zachovány.

Hrany z pravé partity do stoku nám tvoří v grafu řez. Problém má řešení, právě když tyto hrany jsou naplněny na maximum. Hrany mezi partitami nám také tvoří řez, takže vše, co proteče ze zdroje do stoku, proteče i hranami mezi partitami. A hrany mezi partitami reprezentují hrany původního grafu, takže tok na nich je naším řešením.

Nyní k časové složitosti. Časová složitost převodu na nový graf je $\mathcal{O}(n+m)$, kde n je počet vrcholů v původním grafu a m je počet hran.

Každý vrchol zdvojíme, na každou hranu se podíváme jen jednou a přidáváme jen 2 nové vrcholy a s nimi dohromady $2n$ hran. Zbytek časové složitosti závisí na použitém algoritmu pro zjištění maximálního toku. V našem případě, kdy jsme použili Forda-Fulkersona s procházením do šířky, je to $\mathcal{O}(nm^2)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/23-5-6.cpp`

Karel Tesař

23-5-7 Perlím, Perlíš, Perlíme

V úloze byly zadány dva úkoly. První těžší, druhý lehčí. S oběma jste se dokázali velmi dobře vyrovnat.

První úkol bylo možno řešit dvěma způsoby. Buďto smažu menší ze zadaných čísel, nebo matchnu celý výraz a správnou konstrukcí z backreferencí vyberu to větší. Oba postupy využívaly rekurzi a rozhlížení (bez toho to pravděpodobně nešlo).

Mnoho z vás si všimlo, že z rozhlížecích předpokladů se dá sestavit jakási podmínka. `((?=A)B|C)` znamená jednoduše to, že pokud řetězec splňuje regex `A`, tak použij `B`, jinak použij `C`.

Rozeberme si autorské řešení. Předpokládáme v něm, že čísla nejsou uvozena nulami.

```
s#^((?=(.((?2)| ).).+).* (.*)|(?=.+(?2)(?!.))(.*) .*|
  (.*)0.* (\6[1].*)|((.*)1.*) \9[0].*)$#$4$5$7$8#
```

Na prvním řádku jsme nejprve rekurzí zjistili, že druhé z čísel je delší, takže si vybereme to druhé z nich. Poté jsme analogicky zjistili, že první z čísel je delší. Mimochodem, domácí cvičení – jaký je rozdíl mezi `(?!.)` a `$`?

Na druhém řádku jsou tedy čísla nutně stejně dlouhá. Využijeme žravosti hvězdičky (`.*` polkne co nejvíc) a předloženým výrazem zjistíme rozdíl dvou stejně dlouhých čísel – porovnáváme první cifru, která se liší.

Poslední část je nahrazovací výraz. Jsou to reference na „ty správné závorky“, které buďto nematchly, a tedy neobsahují nic, a nebo obsahují to větší z čísel.

Úkol se jak zadáním, tak stylem řešení dosti podobá poslednímu úkolu z předchozího dílu. Řešitelé, kteří si toto uvědomili, si ušetřili trochu přemýšlení.

Úkol 2 byl jednodušší. Stačilo upravit příklad ze zadání na správné uzávorkování a uvědomit si, jak se dá výraz negovat. Jedna z variant byla typu `s/(?!A).*//`. Druhá nepoužívala rozhlížení, ale referenci – `s/(A)|.*/$1/`.

V prvním případě pokud `A` vyhovuje, tak se nepokračuje dál, tedy `.*` nic nepožere a nic se nesmaže, jinak se smaže všechno. Ve druhém případě se matchne buď `A` a nahradí se za sebe sama, nebo se matchne všechno a `$1` bude prázdná.

I zde si rozebereme autorské řešení, řešení účastníků se prakticky nelišila.

```
s#^(?!([^<>]*(<([[:alnum:]]+)>(?1)</\3>[^<>]*)*)$).*$##
```

Regex nejprve přečte veškerý prostý text. Pak je zde velká ohvězdičkovaná závorka, ve které se matchne otevírací tag, uzavírací tag, další následný prostý text a spustí se celý regex rekurzivně na vnitřek tagu. Rekurze se zastavuje tím, že ona velká ohvězdičkovaná závorka nematchne ani jednou.

Negace regexu je prvního uvedeného typu. Má to tu výhodu, že se zbytečně nenahrazuje za `$1`, takže se řetězec celý nepřepisuje, pokud to není potřeba.

Na efektivitu je vůbec u složitějších regexů potřeba dát pozor. ať už z hlediska spotřebovaného času, nebo paměti. Vezměme si například regex

```
s/^(a{0,10000}){0,10000}$//,
```

který vypadá na první pohled nevinně. Nicméně Perl jej nedokáže zoptimalizovat a na vstupu `aaaaaaaaaaaaaaaaaaaaaaaaaab` už běží několik vteřin...

Nicméně takové problémy jsem při opravování nehledal a neřešil.

Jan „Moskyto“ Matějka

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
1.	Jakub Zíka	GNAlejíPH	4	24	236.1
2.	Vojtěch Hlávka	GŠlapanice	2	35	217.9
3.	Lukáš Folwarczný	GKomHavíř	3	22	196.9
4.	Juda Kaleta	GKlatovy	2	23	192.4
5.	Martin Raszyk	G_Karvina	1	22	189.4
6.	Matěj Kocián	GLesníZlín	4	17	173.4
7.	Vojtěch Sejkora	SPSE_Pard	2	23	153.4
8.	Jan Bok	GJungmanLT	4	17	150.9
9.	Peter Zeman	GAV	4	18	149.3
10.	Filip Hlásek	GMikulášPL	4	15	143.2
11.	Michal Pokorný	SŠkybernHK	3	18	134.5
12.	Ondřej Fiedler	GJungmanLT	4	16	129.4
13.	Jindřich Pilař	GBroumov	3	22	128.7
14.	Ondřej Hübsch	GArabskáPH	1	15	126.4
15.	Štěpán Šimsa	GJungmanLT	2	14	121.6
16.	Jerguš Greššák	GRaymanaPV	2	12	118.9
17.	Michal Anderle	GTim_Lučen	4	12	114.8
18.	David Bernhauer	GZborovPH	3	17	109.9
19.	Ondřej Cífka	GNAlejíPH	2	12	102.0
20.	Ondřej Mička	GJírovcČB	2	14	101.2
21.	Jan Hadrava	GZborovPH	3	10	99.8
22.	Jiří Setnička	G25březnPH	4	10	89.3
23.	Matouš Kozma	BiGyBBHK	4	10	88.6
24.	Vojtěch Kletečka	GHavlBrod	3	14	80.2
25.	Filip Štědronský	GMikulášPL	4	6	71.1
26.	Daniel Stahr	GJungmanLT	4	9	69.4
27.	Matěj Židek	GBroumov	3	11	51.5
28.	Jonatan Matějka	GJírovcČB	1	7	48.3
29.	David Krška	GJirsíkaČB	4	6	47.4
30.	Jan Škoda	GMikulášPL	4	7	45.3
31.	Andrej Mariš	PriorPC	3	6	45.2
32.	Jiří Eichler	SlovanGOL	3	4	43.9
33.	Dominik Smrž	GOhradníPH	1	5	43.4
34.	Jitka Fürbacherová	GKlatovy	2	7	40.7
35.	Jan Paštyka	SPSKutHora	2	5	39.8
36.	Rastislav Rabatin	GJHroncaBA	2	6	37.4
37.	Tereza Hulcová	GKlatovy	2	6	34.8
38.	Filip Matzner	GJirsíkaČB	4	5	34.5
39.	Robin Mana	GValašKlob	4	6	32.3
40.	Milan Berka	G_Krumlov	4	4	29.8
41.	Mária Mrocková	GJHroncaBA	4	3	29.0
42.	Daniel Švec	SPŠERožnov	3	5	27.9

Pořadí řešitelů

43.	Jakub Kulhan	G_Kralupy	3	4	27.3
44.	Michal Punčochář	GJírovcČB	1	3	24.2
45.	Tomáš Varga	GMost	-1	4	22.0
46.	Tomáš Jareš	PORGPha	0	3	20.8
47.–48.	Anna Dresslerová	GJHroncaBA	4	2	19.0
	Milan Mikuš	GĽŠtúraTN	3	2	19.0
49.	Pavel Kratochvíl	VOŠGSvětlá	3	4	18.8
50.	Tomáš Velecký	GBezručeFM	0	2	17.3
51.	Jiří Šebele	GArabskáPH	1	2	14.3
52.	Roman Beránek	PrumChemBO	2	3	13.4
53.–54.	Martin Mach	GJírovcČB	3	1	10.0
	Alexander Mansurov	GNVPlániPH	2	1	10.0
55.	Josef Klesa	GKlatovy	3	1	9.5
56.	Martin Holec	GSlavičín	4	2	8.7
57.	Jan Lejnar	GKlatovy	1	1	8.6
58.	Tomáš Turlík	GRaymanaPV	2	3	8.4
59.	Jan Knížek	ZŠDukStrak	0	1	7.5
60.	Petr Pecha	SPŠsVsetín	4	1	7.2
61.	Barbora Hourová	G_Brandýs	4	1	5.7
62.	Patrik Jung	GKlatovy	1	1	4.5
63.	Radim Cajzl	GNoMěsNMor	4	1	1.7

Obsah

Úvod ... 3
Zadání úloh ... 4
- První série ... 4
- Druhá série ... 8
- Třetí série ... 13
- Čtvrtá série ... 20
- Pátá série ... 26

Seriál: Regulární výrazy ... 31
Programátorské kuchařky ... 44
- Kuchařka první série – grafy ... 44
- Kuchařka druhé série – procházky po grafech ... 55
- Kuchařka čtvrté série – toky v grafech ... 59
- Kuchařka páté série – těžké problémy ... 65

Vzorová řešení ... 72
- První série ... 72
- Druhá série ... 84
- Třetí série ... 94
- Čtvrtá série ... 109
- Pátá série ... 119

Pořadí řešitelů ... 132
Obsah ... 134

Martin Böhm a kolektiv

Korespondenční seminář z programování XXIII. ročník

Autoři a opravující úloh:
Martin Böhm, Pavel Čížek, Jozef Gandžala, Karel Král
Lukáš Lánský, David Marek, Martin Mareš, Jan Matějka
Lucie Mohelníková, Jitka Novotná, Petr Onderka, Pali Rohár
Karel Tesař, Mária Vámošová, Michal Vaner, Pavel Veselý

Autoři příběhů v zadání:
Lukáš Lánský, Jitka Novotná, Pavel Veselý

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 389. publikaci.

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

S jejich pomocí ročenku vysázel Radim Cajzl.

Obrázek na obálce nakreslila Lucie Mohelníková.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 136 stran
Náklad 200 výtisků
Praha 2011

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

ISBN 978-80-7378-195-8

ISBN 978-80-7378-195-8