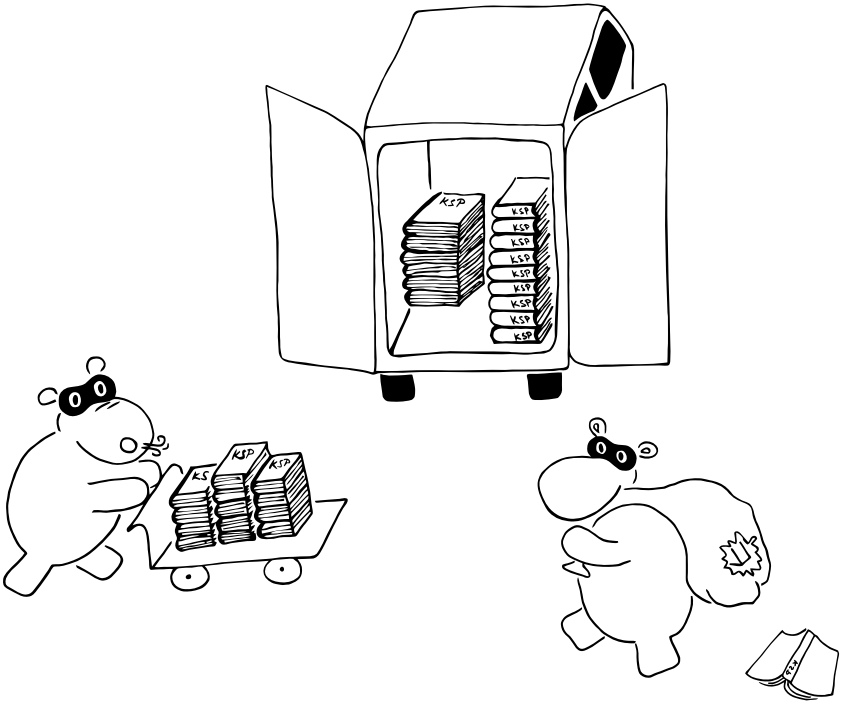
JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXV. ročník – 2012/2013



matfyzpress

VYDAVATELSTVÍ
MATEMATICKO-FYZIKÁLNÍ FAKULTY
UNIVERZITY KARLOVY V PRAZE

JIŘÍ SETNIČKA A KOLEKTIV

Korespondenční seminář z programování

XXV. ročník – 2012/2013

matfyzpress

Praha 2013

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

ISBN 978-80-7378-247-4

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož dvacátý pátý 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 většinou osmi úloh okořeněné příběhem. Poslední úloha je tvořena tzv. *seriálem*, což je povídání o nějakém zajímavém informatickém tématu prolínající se celým ročníkem. V této ročence je seriál vyčleněn z ostatních úloh a uveden pohromadě.

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 následujícího ročníku. Účastníci soustředění zažijí bohatý program – aktivity ryze odborné (především přednášky na různá zajímavá témata) i 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.

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
KAM MFF UK
Malostranské náměstí 25
118 00 **Praha 1**

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

(Nejen) u úloh v této knize lze zahlédnout tyto značky označující typ úlohy:


Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě zvládnou levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.


Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.


Této úloze říkáme *praktická*, jelikož není potřeba popsat algoritmus, jen ho naprogramovat a odevzdat přes Internet. Bližší informace naleznete přímo v jejím zadání.


V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se můžete dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.


Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku,* ze které se můžete takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdete na našich webových stránkách.


Dále tímto symbolem označujeme místa, jejichž pochopení může vyžadovat větší zamyšlení, případně nějaké předchozí znalosti.

Zadání úloh

První série

(volně přeloženo z japonského originálu)

Vážený strýčku,

jsem Vám velice vděčen za Vaši pomoc při přípravách naší cesty do České republiky. Je to velice zajímavá země s mnoha pravzláštními obyčeji, které se Vám pokusím aspoň trochu přiblížit.

Nafotil jsem spoustu fotografií, ukázat Vám je však nemohu – jak píši dále, o všechny jsem přišel.

Už náš příjezd byl takový zvláštní – čekal jsem, že nás na letišti vyzvedne průvodce, pojedeme autobusem do centra, vysedneme na nějaké rušné ulici plné turistů a vydáme se obdivovat památky.

Místo toho jsme však nasedli do dodávky, která trčela v dopravní zácpě snad hodinu. Pak jsme se dlouho proplétali uzounkými uličkami, až jsme se zastavili v jedné velice zapadlé, nikde nikdo a skoro nebylo vidět na slunce.

Chvíli to trvalo, než jsme se vymotali z uliček a dostali do míst, kde byli i jiní lidé. Vydali jsme se s rodiči přes takový starý most do centra. Cestou jsme samozřejmě všichni fotili.

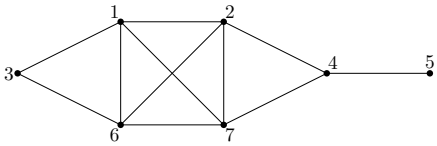
25-1-1 Fotografování — 12 bodů

Máme N japonských turistů, kteří se mezi sebou navzájem fotí. Typicky, když jeden Japonec vyfotí druhého, ten druhý mu to musí ze slušnosti oplatit.

Japonci mají v oblibě jednu hru: v určitou chvíli někdo oznámí přirozené číslo K, načež si všichni spočítají, kolik ostatních mají nafoceno (což je zjevně počet lidí, kteří fotili je samotné). Každý, kdo má nafoceno méně než K kamarádů, vypadává ze hry.

V druhém kole se počítají pouze ti, kteří nevypadli. Opět ti, kteří mají nafoceno méně než K nevypadnuvších, vypadnou ze hry. Takto hra pokračuje, dokud se stav mezi dvěma koly mění. Všichni zbylí hru vyhrají.

Víte, které dvojice Japonců se navzájem vyfotily. Určete pro každého z nich, pro jaké maximální K hru vyhraje. 7 bodů dostanete, pokud najdete vyhrávající skupinku pro $K = 3$.



Příklad: Uvažujme situaci pro $N = 7$, přičemž se vyfotily dvojice: 1–2, 2–4, 4–5, 3–1, 3–6, 6–1, 6–2, 6–7, 7–1, 7–2 a 7–4. Pro $K > 3$ nevyhraje nikdo. Pro $K = 3$ vyhrají hráči $1, 2, 6, 7$, pro $K = 2$ vyhrají všichni až na hráče 5 a pro $K < 2$ vyhrají všichni.

Úzkými uličkami jsme se dostali na náměstí. Měli zde moc pěkné hodiny s figurkami. Také jsme se podívali do místní tržnice. Nevěřil byste, jak jsou místní trhovci hádaví.

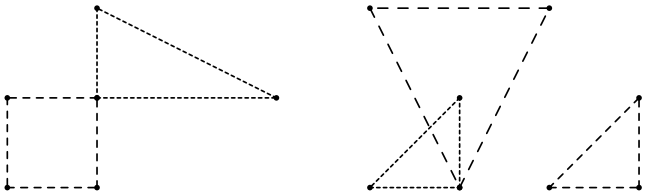
25-1-2 Stánky na náměstí — 12 bodů

Máme T trhovců. Každý trhovec má vlastní stánek. Při jeho stavbě zkušeným okem určil oblast, odkud na něj turisté nejlépe uvidí, tato oblast má tvar konvexního mnohoúhelníka.

Problém nastává, pokud se dvě takovéto oblasti protínají – potom se trhovci neustále hádají a přetahují si navzájem turisty.

Radní o problému ví a snaží se mu zabránit. Oblasti vám zadají jako T seznamů bodů, kde body jsou vždy zadané podél obvodu. Každý seznam bodů tvoří konvexní mnohoúhelník. Chtěli by vědět, jestli se nějaké dvě oblasti překrývají.

Příklad: Pro $T = 2$ a oblasti určené seznamy bodů $\{[1,1],[2,1],[2,2],[1,2]\}$ a $\{[2,2],[4,2],[2,3]\}$ se tyto dvě neprotínají (vlevo), ale pro $T = 3$ a oblasti zadané pomocí bodů $\{[1,1],[2,1],[2,2]\}$, $\{[3,1],[4,1],[4,2]\}$ a $\{[1,3],[2,1],[3,3]\}$ se první a třetí oblast protínají (vpravo).



Pak jsme se vrátili po stejném mostě, prý se podíváme na místo, kde žije český prezident. Těšil jsem se hlavně na pražskou hradní stráž, slyšel jsem totiž, že její řazení při výměně je pověstné.

25-1-3 Řazení hradní stráže — 7 bodů

Při výměně dvou gard stráže ta odchozí nastoupí na nádvoří do řady. Příchozí stráž nastoupí vedle ní.

Obě gardy jsou přirozeně stejně velké – každá má právě N vojáků. Vojáci mají dopředu určeno místo, kde hlídají, každý z gardy hlídá na jiném místě.

Pro kontrolu, že jsou všichni a že žádné z míst nezůstane nehlídáno, se příchozí garda musí seřadit stejně jako ta odchozí.

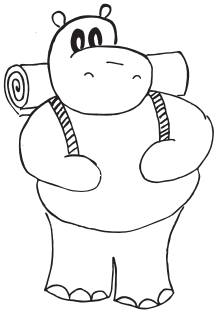
Řazení podle protokolu probíhá tak, že se vždy odpojí poslední v řadě příchozí gardy a zařadí se na libovolné místo. Kolik takovýchto přesunů je nejméně potřeba, aby příchozí garda byla seřazena stejně jako odchozí?

Příklad: Když si vojáky očíslujeme $1 \ldots N$ dle místa, kde budou hlídat, tak pro odchozí řadu $4,2,1,3,5$ a příchozí řadu $1,3,5,2,4$ jsou potřeba dva přesuny.

To jsem bohužel už neviděl. Na tom mostě jsem se snažil vyfotit hlavu ve zdi, než se mi to však podařilo, všichni byli dávno pryč. Snažil jsem se je dohnat, ale nějak jsem se ztratil.

Dostal jsem se do krásného parku, jsou tady i nějaká obrovská mimina.

„Dobrý den, pane, promiňte, že obtěžuji, ale neviděl jste tu skupinku turistů z Japonska?“



Pán vypadal velmi vyjeveně. Pak jen zadrmolil „Sorry, don't speak english,“ a zmizel. Copak já na něj mluvím anglicky? Vždyť to ani neumím... Zkusil jsem se ptát ještě pár dalších, vždy s podobným výsledkem. Koukali na mě, jako bych spadl z Marsu. Jeden z nich mi zdviženým prostředníčkem naznačil, že jsem jedničkа, ale stejně mi nepomohl.

Nikdo mi nerozumí. Sedl jsem si na lavičku, sklopil hlavu a přemýšlel, jak se dostanu zpátky. Taková ostuda! Takhle zahanbit své rodiče!

„Ahoj, co tu děláš tak sám?“

Zvedl jsem hlavu. Tak přece někdo! Stála nade mnou menší hnědovlasá slečna, na sobě měla hranaté kovové brýle a přes rameno brašnu.

„Kde máš rodiče?“

„Já nevím. Byli jsme támhle na mostě, fotil jsem, ostatní najednou zmizeli.“

Když už jsem měl v ruce foťák, tak jsem si slečnu vyfotil.

„A kam šli?“

„Říkali něco o hradu, ale těžko říct.“

„A víš aspoň, kde se máte sejít?“

„Tady za rohem máme dodávku, nejspíš tam.“

„Tak pojď.“

Chvíli jsme se proplétali uličkami, než jsme našli tu jednu liduprázdnou, kde jsme parkovali. Teď už moc liduprázdná nebyla. Bylo tam asi dvacet mužů, všichni stáli kolem naší dodávky. Někteří se jen dívali okolo, někteří vykládali z naší dodávky nějaké zboží. Asi čtyři z nich stáli opodál a živě spolu diskutovali.

Samozřejmě jsem začal fotit, tohle se jen tak nevidí. Při páté fotce fotoaparát usoudil, že scéna je moc tmavá, a zapnul blesk. V tu chvíli se všichni zarazili a podívali se na mě. Spoušť jsem zmáčknul ještě jednou.

V tu chvíli mě ona slečna chňapla za ruku a táhla pryč. Rozběhli jsme se a utíkali, co nám síly stačily. Proč, proboha? Na otázky však nebyl čas. Pochopil jsem, že z nějakého důvodu nás nesmí chytit. Nejspíš tu slečnu znají a hrají nějakou společenskou hru.

Běželi za námi dva. Když se rozdělíme, budou nás hledat mnohem hůře. Vytrhl jsem slečně svoji ruku a rychle zahnul doprava. Nejspíš pochopila můj záměr a zahnula doleva.

V úzkých uličkách je snadno ztratíme.

25-1-4 Útěk — 12 bodů

Úzké uličky tvoří bludiště. V tomto bludišti jsou osoby, které nesmíte potkat. Tyto osoby neběhají chaoticky, mají jistý okruh, po kterém chodí stále dokola.

Bludiště je zadáno jako čtvercová síť o rozměrech $W \times H$, kde $1 \leq W, H \leq 70$. V bludišti jsou čtyři typy polí: zeď `#`, volno `.`, start `S` a cíl `C`. Start je v bludišti právě jeden.

V bludišti je N osob, kterým utíkáte, přičemž $0 \leq N \leq 2$. Pohyb každé z osob je dán seznamem souřadnic délky D, jehož konec navazuje na začátek. Osoba se po nich cyklicky pohybuje. Seznam je dlouhý $1 \leq D \leq 50$ a je vždy platný (osoby neprocházejí zdmi, místa na sebe navazují). Je-li $N = 2$, pak mohou obě osoby stát na stejném místě.

Pohyb v bludišti probíhá po tazích, v tahu se vždy můžete posunout o jedno políčko vodorovně nebo svisle, nebo stát na místě. Nejprve se pohnete vy, poté chytající osoby.

Najděte nejrychlejší cestu ven z bludiště, aniž byste potkali libovolnou z osob.

6 bodů získáte za řešení fungující pro $N = 0$, 3 body pro $N = 1$ a 3 body pro $N = 2$.

Už mně dýchal na krk. Najednou se však zastavil a nešel dále. Došel jsem na větší obdélníkové náměstí, uprostřed vedly tramvajové koleje.

Uf. To bylo o fous.

Najednou jsem na chodníku uviděl peněženku. Jak mě to mí ctění rodiče vždy učili, sebral jsem ji, neotevíral a začal se poohlížet po policistech, kterým bych ji mohl odevzdat.

Jeden šel kousek ode mě. Vydal jsem se mu naproti a natáhl ruku s peněženkou. Policista došel ke mně, pořádně se na mě ani nepodíval, ignoroval peněženku a sebral mi foťák.

Stál jsem jako opařený, stále s nataženou rukou. Najednou byla prázdná. Nějak se tam objevila ta slečna, která se mnou předtím utíkala.

S peněženkou v ruce se o něčem s policistou dohadovala. Po chvíli však bez jediného slova zmizela.

Policista se na mě podíval a snažil se mi něco říct, já mu však nerozuměl. Pak mě odvedl s sebou na služebnu. Ach, strýčku! Připadal jsem si jako nějaký sprostý zloděj!

Policista mě posadil naproti svému stolu, chvíli na mě koukal, pak zmizel. Koukal jsem na práci jejich sekretářky. Pořád vytahovala z kartotéky nějaké papíry, občas na ně něco načmárala a pak je zase vložila zpátky. Vypadalo to však velmi neefektivně.

25-1-5 Algoritmus sekretářky — 7 bodů

Představte si, že sekretářka je naprogramovaná a provádí následující kód:

```
for i in range (0, N):
  for j in range (0, M):
    if (a[i]==c[j]):
      print b[i]+" "+d[j]+"\n";
```

Zkuste si rozmyslet, co kód (sekretářka) dělá a s jakou časovou složitostí. Potom zkuste vymyslet vlastní program, který dělá totéž efektivněji, nebo dokažte, že (asymptoticky) to už efektivněji nejde.

Vzpomněl jsem si, že mám v kapse kartičku, kterou mi rodiče napsali pro podobné případy. Vyndal jsem ji a podal policistovi, když se vrátil.

Ten si ji přečetl a zvedl telefon. Těm slovům jsem nerozuměl, přesto se mi vryla do paměti.

„Dobrý den, máme tu jedno ztracené dítě, má s sebou kartičku s vaším číslem. Jmenuje se Tanaka Mashiro, má krátké černé vlasy, velké kulaté sluneční brýle, na sobě černé tričko a modré rifle, nosí s sebou černý fotobatoh... Vážně? Výborně, dovedu ho k vám.“

Pak se zvedl, něco si zamumlal a naznačil, že mám jít s ním. Chvíli jsme se proplétali uličkami, než jsem uviděl naši krásnou vlajku.

Vešli jsme do budovy, nad kterou visela. Policista se chvíli bavil s vrátným, pak odešel a zmizel.

Vrátný se na mě usmál a ukázal na stůl vedle.

„Posaď se a chvíli počkej. Máš hlad? Donesu ti něco k jídlu.“

Teprve teď jsem se trošku uklidnil.

Při čekání jsem se díval z okna na zahradu. Je na ní krátký zelený trávník a spousta nádherných květin. Zajímalo by mě, jak ji sekají.

25-1-6 Sekání trávy — 10 bodů

Představte si trávník $N \times M$, který chceme posekat. Máme sekačku, kterou se můžeme pohybovat pouze vodorovně nebo svisle.

Začínáme v levém horním rohu a chceme každé políčko projet právě jednou (start je výjimka) a vrátit se na začátek. Na trávníku rostou květiny, které nechceme posekat.

Pro jaké N, M a umístění květin umíme trávník posekat? Pro zjednodušení předpokládejte, že na trávníku rostou květiny jen na jednom políčku.

4 body dostanete, pokud vyřešíte sekání trávníku bez květin.

Seděl jsem tam asi půl hodiny, než přišel pán v obleku.

„Vítej na japonské ambasádě. Už jsem volal tvým rodičům, vyzvednou si tě tady. Máš velké štěstí, že ses sem dostal. Ztrácí se tady spousta dětí. Mohl bys mi říct, co se vlastně stalo?“

Všechno jsem pánovi vyložil, celou dobu pozorně poslouchal. Zmínil jsem se mimo jiné i o zmizelém fotoaparátu.

Lehce se uklonil a natáhl ruce s fotoaparátem. Též jsem se uklonil a převzal jej.

„Datovou kartu bohužel nemáme, je mi líto.“

„Přesto vám co nejsrdečněji děkuji.“

„Teď mě omluv. Musím zmizet na schůzi.“

Přemýšlel jsem, jak se budou rodiče tvářit, až mě uvidí. Nejspíš na mě budou naštvaní, že jsem jim takhle zkazil dovolenou. Ani fotky nemám. . .

Nakonec ale měli spíš radost, že mě našli. Došli jsme zpátky k dodávce (tentokrát už kolem ní nikdo nestál), nasedli a jeli zpátky na letiště. Mezitím jsem od rodičů dostal aspoň GPS s logem, abych se mohl podívat, kam došli.

Při procházení logu jsem si všímal hlavně nadmořských výšek. Cestou po Praze dosti kolísaly. Zajímavý byl zejména počet kopečků a jeho změny s přiblížením mapy.

Po chvíli jsem si všimnul, že při vhodném přiblížení to vypadá tak, že se nejprve hodně dlouho stoupá, pak se dosáhne vrcholu a za ním se už jen klesá. Dal by se podobný výběr udělat i „ručně“?

25-1-7 GPS log — 7 bodů

Máte zadanou posloupnost nadmořských výšek tak, jak je turisté postupně procházeli. Zvládnete z nich vyškrtat co nejméně výšek tak, aby zbylá posloupnost obsahovala maximum, výšky před maximem pouze stoupaly a ty za ním klesaly?

Příklad: Pro posloupnost výšek $1, 3, 2, 2, 3, 4, 6, 7, 7, 5, 6$ je jedním ze správných řešení posloupnost $1, 2, 3, 4, 6, 7, 5$ (poslední prvek lze nahradit šestkou).

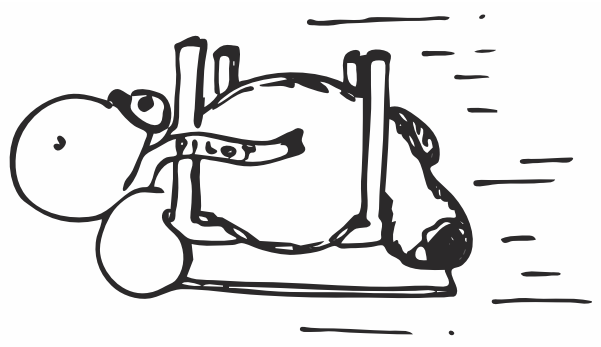
Jistě uznáte, že tato cesta byla velice zvláštní. Doteď pořádně nechápu, co se vlastně stalo. Možná mi to někdo někdy vysvětlí.

Tanaka Mashiro

Z japonštiny přeložil:

Radim „Rumcajz“ Cajzl

Druhá série



„Újezd. Příští zastávka: Hellichova.“ No jo, tahle hlášení bych mohla odříkávat nazpaměť. Ne že bych si za těch pár let, co v Praze žiju, stihla zapamatovat všechny linky MHD, ale úseky některých z nich ano.

Dav turistů, mířících nejspíš na petřínskou lanovku, se vyhrne z tramvaje. Hned se tu aspoň dá trochu dýchat. Někdy by se hodilo vědět, kde se má člověk připravit na největší davy.

25-2-1 Vytíženost dopravy — 13 bodů

Pokud si představíme, že zastávky jsou vrcholy grafu a spoje představují hrany mezi nimi, potom dopravní síť tvoří strom. Hrany jsou ohodnocené očekávaným počtem cestujících.

Navrhněte datovou strukturu, která se vybuduje pro zadaný strom a následně bude umět co nejrychleji odpovídat, ve kterém úseku (na které hraně) cesty z vrcholu X do vrcholu Y pocestuje nejvíce lidí. Počítejte s tím, že počet dotazů bude řádově odpovídat počtu vrcholů.

⇧ **Lehčí varianta (za 7 bodů):** Řešte stejnou úlohu za předpokladu, že grafem představujícím dopravní síť je cesta.

Konečně dorážíme na Hellichovu. Ani nečekám na další hlášení a vystupuju. Teď už jen pár uliček a japonský velvyslanec pan Yamada se může těšit na milou návštěvu. Jemu možná tak milá připadat nebude, ale… vaše články se nedostanou na titulní stránky novin proto, že se lidí ptáte jen na milé věci.

Cestou kolem muzea hudby si všímám hezky upravené zahrady, a hlavně chlapíků, co ji právě sečou. Pořád se střídají u sekačky. Skoro to vypadá, že hrají nějakou hru.

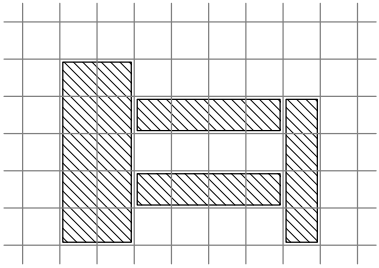
25-2-2 Sekání trávy podruhé — 11 bodů

Mějme zahradu tvořenou několika obdélníky ve čtvercové síti. Zahrada je souvislá. Jednotlivé obdélníky spolu sousedí, ale nepřekrývají se. Každý obdélník má sudý obsah.

Na začátku stojí sekačka na libovolném políčku. Dva hráči se pravidelně střídají, každý vždy popojede se sekačkou o jedno políčko. Benzín je v dnešní době drahý, a proto nesmí být žádné políčko posekáno vícekrát. Ten hráč, který jako první nemá kam sekačkou pohnout, prohrál a musí ji po dosekání uklidit.

Rozhodněte, pro kterého hráče existuje výherní strategie, a popište ji. Tedy zjistěte, jestli vyhraje ten, kdo pohne se sekačkou jako první, nebo ten, kdo s ní pohne jako druhý. Pro tohoto hráče popište, jak má táhnout, aby mu žádné protitahy jeho soupeře nezkazily výhru.

Na obrázku je jeden z možných tvarů zahrady. Úlohu řešte obecně pro všechny tvary zahrady splňující podmínky zadání.



Na chvilku jsem se u sledování sekání zapomněla, ale opět vyrážím dál. Po chvilce se dostávám na místo určení. A koho to nevidím, to bych si snad ani nemohla naplánovat – pan Yamada osobně. Přidávám do kroku.

„Ohayou gozaimasu, Yamada-sama!“ zastupuju muži cestu, zatímco si nenápadně zapínám ukrytý diktafon. Prvotní překvapení ve tváři pana Yamady střídá jasný výraz nespokojenosti, tím se ovšem nenechávám zastrašit. Mluvím o nedávných případech, ptám se ho na jeho názor. Mluvím o mafii a chci po něm vyjádření.

Odpovědi jsou všechny jenom takové ty diplomatické frázičky, ale jsem si jistá, že tenhle člověk ví víc, než přiznává. Najednou pan Yamada sahá po telefonu a s omluvou odchází o kus dál. Zaslechnu jen slova „. . . zase tady“ a „udělej s ní něco“, kupodivu mluví česky. Začínám tušit problémy.

A taky že jo! Neuplynulo snad ani pět minut a přihnal se sem nějaký policista. Že tohle nesmím, že musím odejít, blá blá.

Je čas použít mou úžasnou výmluvu. „Promiňte, já si jen chtěla nechat poradit s touhle japonskou kalkulačkou,“ vytahuju svoji starou kalkulačku značky Yamaha. „Má jeden zajímavý mód.“

25-2-3 Doplňování operátorů — 6 bodů

Máme zadaná celá čísla a chceme mezi ně doplnit znaménka plus + nebo krát $*$ tak, aby výsledek vzniklého výrazu byl co největší. Jak to udělat?

Příklad: Pro čísla 6 2 1 3 0 je nejvýhodnější doplnění

$$6 * 2 * 1 * 3 + 0 = 36.$$

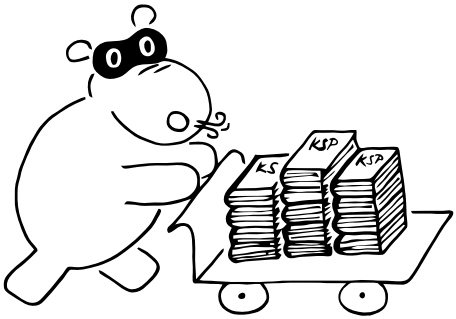
Evidentně byl úkol příliš jednoduchý. Hlavně mě ten policista zdržel natolik, že milý pan velvyslanec pláchl. Pro teď bude asi rozumnější zmizet a vrátit se sem jindy. Raději se půjdu někam projít.

Moje kroky mě dovedly až na Kampu. A o kus dál, k menšímu japonskému chlapci. Působí ztraceně, radši ověřím situaci.

„Ooi. Hitoribocchi desu. Naze.“ ptám se. Chlapec se na mě podíval a úplně se mu rozzářily oči.

Po chvilce už vím, že se jmenuje Tanaka, zatoulal se své skupince a že snad trefí tam, kde se mají sejít. Samozřejmě ho doprovodím, potřebuje to… a kdo ví, třeba se od jeho skupinky ještě dozvím něco zajímavého.

Z Tanakova výrazu vyčtu, že bychom měli být na místě, a vzápětí lapám po dechu. Mafiáni! Lidi před námi jsou určitě mafiáni uprostřed akce. Když děláte novinařinu dost dlouho, na některé věci prostě máte čuch, a tohle k nim patří. Chvíli sleduju, jak jsou zorganizovaní.

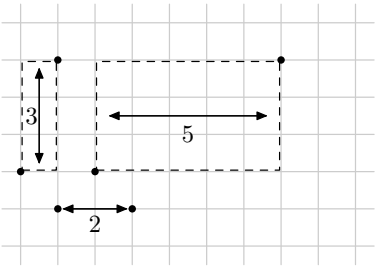


25-2-4 Organizace vykládky — 13 bodů

Základem organizovanosti je vybrání dobrého místa pro zastavení dodávky. Mafiáni se kolem ní pak rozestaví v pomyslné čtvercové síti a předávají si zboží, které se má dopravit na jednotlivá místa. Je žádoucí, aby počet mafiánů potřebných na vyložení všeho zboží byl co nejmenší. Pohyb mafiánů po čtvercové síti odpovídá pohybu krále po šachovnici.

Formálněji řečeno, mějme N bodů v rovině, používejme maximovou metriku (právě ta odpovídá minimálnímu počtu kroků šachového krále mezi dvěma poli šachovnice). Hledáme bod ze zadaných, pro který platí, že součet vzdáleností od všech ostatních bodů je pro něj nejmenší.

Maximová metrika funguje v rovině tak, že vzdálenosti bodů odpovídá větší z rozdílů jejich souřadnic. Tedy $d((x_1, y_1), (x_2, y_2)) = \max\{|x_1 - x_2|, |y_1 - y_2|\}$



Najednou se objeví blesk z Tanakova foťáku. A do háje! Všichni si nás samozřejmě všimli.

Chytám Tanaku za ruku a rozbíhám se s ním pryč. Kdyby nás chytili, mohlo by to být hodně špatné. Najednou se mi Tanaka vytrhl. Běží doprava. Nejsem si jistá, co má v plánu, ale odbočuju doleva. Máme tak větší šance a on se snad znovu neztratí. Ani nenechá chytit.

Z nějakého důvodu, možná jak jsme se od sebe tak odtrhli, mi ale vyletěl z brašny blok s poznámkami a vysypaly se z něj jednotlivé papíry!

To mi tak chybělo... potřebuju je posbírat, a to hezky rychle.

25-2-5 Sbírání papírů — 8 bodů

Novinářce se na cestu rozsypaly papíry. Představme si cestu jako čtvercovou síť $M \times N$, kde přesun mezi dvěma políčky odpovídá jednomu kroku a není povoleno přesouvání šikmo. Na některá pole se vysypaly jednotlivé papíry.

Novinářka se nemůže vracet zpět (řekněme dolů), může jen vpřed (nahoru), doprava a doleva. Zároveň potřebuje posbírat všechny papíry na co nejmenší počet kroků. Navrhněte algoritmus, který jí poradí, jak se má pohybovat. Na začátku stojí novinářka v levém dolním rohu.

Příklad: (políčko s papírem je 1, bez papíru 0)

```
0 1 0 0
0 1 0 1
0 0 1 0
```

Optimální řešení je například `RRURLLU`.

⇧ **Lehčí varianta (za 3 body):** Řešte úlohu pro oblast širokou právě 3 políčka, tedy pro čtvercovou síť rozměru $3 \times N$.

S papíry v náručí utíkám dál, dokud se mi nepovede setřást i posledního mafiána. Těžce oddechuju. Vůbec jsem nevnímala, kam běžím, ale to vyřeším později. Tohle bude úžasný článek. Škoda jenom, že nemám ty fotky, co nafotil Tanaka. Ale žiju, to je možná hlavní.

Sahám do brašny pro mobil, abych zavolala Jitce. Moment, tady je něco špatně!

Chvíli zoufale přehrabuju brašnu, než si připustím, že má peněženka v ní prostě není. Jenže ráno jsem ji určitě měla. Musela jsem ji vytratit při tom zběsilém útěku. Co teď?

Zpátky, přímo do náruče mafiánů, se mi tedy nechce. Raději vyrážím po okolí se zoufalou nadějí, že se peněženka někde zázračně objeví.

A dneska se zázraky podle všeho dějí. Kluk, kterého vidím, totiž není nikdo jiný než Tanaka, a věc, kterou drží ve své ruce, není nic jiného než má peněženka! A . . . , počkat, to je ten otrava z rána, co mě odháněl od ambasády!

Co se to tam vlastně děje? Tanaka mává mou peněženkou a Otrava mu bere foťák. . . Ha, foťák! Kdybych se dostala k fotkám, byl by materiál pro článek už naprosto dokonalý.

Dojdu k těm dvěma blíž. Tanaka je tak zaražený, že si vůbec nevšímá, když mu z ruky vytáhnu svou peněženku. Otrava si toho ale všiml a hned po mně vyjel. Bráním se, že peněženka je moje, že v ní jsou doklady, podle kterých mě může zkontrolovat.

S nedůvěřivým pohledem mi bere peněženku. Nijak zásadně se nebráním.

„Nechcete podržet ten foťák, ať to můžete líp zkontrolovat?“ ptám se.

„Hm. . . jo, díky,“ zabručí Otrava a podá mi foťák.

Jo! Když děláte novinařinu dost dlouho, naučíte se taky dost rychle vytahovat paměťovky z foťáků. A dost nenápadně.

Takže když mi pan policista o chvíli později s náležitými omluvami a domluvami podává peněženku zpět, bydlí už paměťovka z Tanakova foťáku v mé brašně.

Spokojeně mizím přímo do redakce. Tam mě vítají zprávy o optimalizaci, či co to má být.

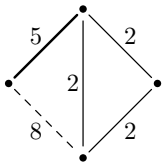
25-2-6 Optimalizace v redakci — 9 bodů

Novinový článek se může nacházet v mnoha různých stavech a mezi každými dvěma z nich ho může přesouvat nejvýše jeden redaktor. Každému takovému redaktorovi se ale platí, a šéfredaktoři se rozhodli snížit výdaje. Chtějí tedy některé redaktory propustit, aby součet platů těch zbylých byl co nejmenší.

Je ovšem třeba zajistit, že se článek stále bude moci dostat z libovolného stavu do libovolného jiného. Zjistěte, kteří redaktoři se nemusí trápit, kteří si mají začít hledat novou práci a kteří by měli vyrazit koupit svým nadřízeným dobrou bonboniéru.

Formálněji řečeno, nalezněte algoritmus, který pro každou hranu neorientovaného ohodnoceného grafu (váhy více hran mohou být stejné) rozhodne, zda ta hrana leží v každé minimální kostře grafu, žádné minimální kostře, nebo v některých minimálních kostrách. Připomeňme, že o minimálních kostrách píšeme v kuchařce.

Příklad: V následujícím grafu jsou tučně vyznačeny hrany, které leží ve všech minimálních kostrách, tence hrany, které leží v některých, a čárkovaně hrany, které neleží v žádné minimální kostře.



S potěšením zjišťuju, že já jsem v bezpečí. A v ještě větším bezpečí budu, až konečně dopíšu ten článek.

Investigativně novinařila

Karolína „Karry“ Burešová

Třetí série

10. 12. 2042

Odkládat věci je snadné, proklatě snadné. Někdy kolem třicátých narozenin jsem si řekl, že jednou sepíšu některé ze svých zážitků a zanechám v nich otisk svých pocitů z tohoto světa. Každý rok jsem si říkal, že ještě není ta pravá chvíle, že to přece má svůj čas. A najednou... ani nevím jak, jsem starcem a tuším, že čas se krátí.

Když se tak zpětně ohlížím, asi jsem nikdy příliš nepřivykl tempu života. V mládí jsem usiloval o spoustu věcí, ale vždy se mi nakonec podařilo nechat si je proplout mezi prsty. Jednou jsem na přechodnou dobu přijal místo u policie. Z přechodné doby se stala záležitost na celý život. Asi jsem objevil klid, který jsem hledal. Práci jsem trávil pochůzkami, většinou jsem lidem pomáhal s různými výtržníky a chuligány, dělal jsem to velmi rád a po práci měl konečně klid na všechny ty věci, které mi dříve unikaly...

Chci vyprávět o mnohém, ale začnu historkou, která mi do dnešního dne občas nedá spát.

⋆ ⋆ ⋆

I když se odehrál před desítkami let, pamatuji si ten den velice dobře. Dopoledne zajímavé nebylo, začalo velkou poradou, před kterou si náš velitel, poručík Hamáček, neodpustil monolog o stavu disciplíny v policejním sboru, který korunoval okázalou kontrolou toho, zda se všichni dostavili.

25-3-1 Kontrola docházky — 12 bodů

Pro řádnou kontrolu docházky je nutno své podřízené přepočítat. Policejní poručík Hamáček na to má svůj systém osvědčený léty služby – ve svém notesu má N dvojic (M_i, K_i). Všechna čísla M_i jsou po dvou nesoudělná a platí $0 \leq K_i < M_i$. Celkový počet policistů je menší než součin všech M_i.

Samotná kontrola probíhá v N krocích, v i-tém kroku se příslušníci srovnají do řad po M_i osobách a poručík Hamáček následně zkontroluje, zda odpovídá počet policistů, kteří už nemohli vytvořit celou řadu, hodnotě K_i.

Vrchní referent, strážmistr Borůvka, se stěhuje. Pomozte mu určit k takové, že vygumováním dvojice (M_k, K_k) z poručíkova notesu umožní co největšímu počtu kolegů mu místo porady pomoci se stěhováním.

Poručík nesmí nic poznat, tedy počty nezařazených policistů pro zbývajících $N - 1$ dvojic musí stále odpovídat.

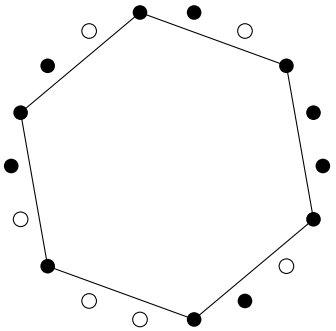
Příklad: Mějme 7157 policistů, kteří se postupně řadí do řad po 12, 13 a 49 osobách. Dvojice (M_i, K_i) tedy jsou $(12, 5)$, $(13, 7)$ a $(49, 3)$. Optimálním řešením je vygumovat dvojici $(13, 7)$. Na stanici pak musí zůstat 101 policistů.

Po úmorném přepočítávání, připomínajícím vojenské cvičení, jsme konečně mohli usednout k poradě.

25-3-2 Zasedání u kulatého stolu — 10 bodů

Na policejní schůzi se sešlo N příslušníků policejního sboru sedících u kulatého stolu, vzdálenost mezi každými dvěma sousedními policisty je shodná. Schůze je dlouhá, policisté v jejím průběhu všelijak odcházejí a přicházejí. Přítomnost policistů v pravé poledne je zadána jakožto posloupnost N nul a jedniček, kde i-tá jednička znamená, že policista na i-tém místě je na schůzi právě přítomen. Vaším úkolem je zjistit, zda existuje $K \geq 3$ takové, že lze vytvořit pravidelný K-úhelník, jehož vrcholy tvoří přítomní policisté.

Příklad: Pro 12 policistů a posloupnost 111010111011 je odpověď kladná, lze sestrojit trojúhelník nebo šestiúhelník. Pro 5 policistů a posloupnost 10111 pravidelný K-úhelník nesestrojíme. Další příklad je na obrázku (plné tečky představují přítomné policisty):



Po schůzi jsem se z naší tehdejší služebny ve Vlašské ulici vydal na pochůzku. Propletl jsem se spoustou aut před Schönbornským palácem, asi se na americké ambasádě konala důležitá recepce, a pak zahnul do spleti úzkých uliček.

V jedné velmi úzké uličce stála dost svérázně zaparkovaná černá dodávka s japonským osazenstvem. Po zdlouhavé komunikaci, která spíš než pomocí několika málo anglických slůvek probíhala především gestikulací, se mi podařilo domluvit, ať mi zavolají někoho, kdo se alespoň trochu dorozumí anglicky (možná německy, jistý jsem si naší domluvou příliš nebyl). Zatímco jsem čekal, začalo mě velmi silně zajímat, co se nachází v těch černých pytlích v dodávce.

V tom okamžiku se ale z mé vysílačky ozval rozkaz k okamžitému přesunu k japonské ambasádě. Má námitka, že zrovna něco zajímavého mám, byla smetena. Prý je nutné se okamžitě postarat o bezpečnost japonského konzula. Propletl jsem se tedy několika úzkými uličkami a za zvuku sekaček z nedalekého parku uháněl k ambasádě.

25-3-3 Do třetice sekání — 13 bodů

V této sérii pro změnu trávník vypadá jako jeden řádek čtvercové sítě a je již posekán, nyní je potřeba posvážet posekanou trávu. Vaším úkolem je zanalyzovat, kolik by různé možnosti svozu stály námahy.

Máte zadáno N přirozených čísel udávajících hmotnost trávy na jednotlivých polích a D intervalů $[a..b]$. Takový interval znamená, že se bude svážet z políček a až b. Námaha pro převoz L trávy z políčka k na políčko l je dána jako $L \cdot |k - l|$. Pro každý interval určete políčko, na které se dá všechna tráva posvážet s nejmenší celkovou námahou. Celková námaha je součet veškeré námahy, jež byla potřeba pro svezení trávy z celého intervalu na toto políčko.

Pokud existuje více správných políček, vypište libovolné z nich. Výpočty pro jednotlivé intervaly jsou nezávislé, tedy množství trávy na jednotlivých políčcích se mezi intervaly nemění. Složitost algoritmu by měla být optimalizována pro případy, kdy je D řádově stejně velké jako N.

Příklad: (První řádek obsahuje N a D, následují hodnoty L a pak intervaly.)

```
10 3
10 3 1 3 9 8 5 4 12 9
1 4
3 6
5 9
```

Odpovědi pro jednotlivé intervaly jsou `1 5 7`.

Před ambasádou postávalo několik lidí. Nejvíce pozornosti zde poutala žena hovořící s jedním Japoncem, pravděpodobně oním konzulem, v jeho rodné řeči. Jemu to zřejmě nebylo příliš příjemné.

Oslovil jsem je, abych zjistil, co se děje. Na to spustila přítomná žena dlouhý monolog o tom, že je novinářkou, že se zabývá důležitou mezinárodní zločinnou kauzou, že je ve veřejném zájmu, aby položila několik otázek konzulovi, že jí japonská ambasáda odpírá právo na informace a že v této zemi obecně není dostatečně ctěna svoboda tisku. Zatímco mi to vše tak emotivně sdělovala, jí ale pan konzul utekl.

V okamžiku, kdy si toho všimla, trochu znejistěla, řekla něco o tom, že mi vlastně vůbec nic není do toho, co dělá. Ona si prý jen tak postává před ambasádou a zrovna potřebuje něco spočítat na jakési speciální kalkulačce. A skutečně vytáhla z kabelky kalkulačku.

Ohlásil jsem stanici, že konzul je mimo nebezpečí. Trochu vzrušený hlas poručíka Hamáčka mi sdělil, že se mám okamžitě přesunout na místo na druhém konci Malé Strany. Prý naše jednotka bude provádět zásah.

Podařilo se mi tam dorazit v okamžiku, kdy probíhaly poslední přípravy. Zatímco chlapi z jednotky kontrolovali zbraně, vyprávěl poručík Hamáček svým skoro otcovským hlasem o tom, jak mu jeho pečlivá příprava jednou zachránila život při střetu se členy jednoho nebezpečného mezinárodního gangu.

25-3-4 Zločinná záležitost — 10 bodů

Ve špinavých vodách mezinárodního zločinu je obzvláště důležitá organizace, typickým příkladem zločinu je výroba něčeho ilegálního. Výroba probíhá ve fázích a obvykle na více než jednom místě. Převoz z jednoho místa na druhé je nejkritičtější částí výrobního procesu, proto je potřeba počet převozů mezi výrobními místy minimalizovat.

V této úloze budete mít na vstupu popsán výrobní proces ve dvou továrnách. První řádek obsahuje čísla N a M, kde N udává počet výrobních fází a M počet závislostí mezi nimi. Druhý řádek vstupu obsahuje N hodnot, kde i-tá hodnota je 1, pokud má i-tá fáze probíhat v první ilegální továrně, a 2 v případě, kdy má probíhat ve druhé továrně. Následuje M řádků obsahující dvojice a, b, které značí, že fáze b může proběhnout až tehdy, když byla provedena fáze a.

Určete pořadí fází výrobního procesu tak, aby každá fáze proběhla až poté, co jsou provedeny všechny fáze, na kterých je závislá, a aby počet převozů mezi výrobními místy byl minimální. Převoz je nutný vždy, když po fázi probíhající v továrně 1 bezprostředně následuje fáze v továrně 2, nebo naopak. (Můžete předpokládat, že řešení existuje.)

Lehčí varianta (za 6 bodů): Nabízíme k řešení i jednodušší variantu úlohy, kde všechny fáze probíhají v jediné továrně.

Příklad:

```
7 9
2 2 1 1 1 2 1
2 1
2 5
3 2
3 4
4 1
4 5
4 7
5 6
6 7
```

Potřebujeme alespoň 4 převozy. Výroba může proběhnout takto – začneme v první továrně, provedeme v ní fáze 3 a 4, pak se přesuneme do druhé továrny a provedeme fáze 2 a 1. Následují fáze 5 v první továrně a 6 v druhé továrně, končíme po čtvrtém převozu v první továrně fází 7.

Už se nepamatuji, jaké zločince tam naše jednotka očekávala. Každopádně, po pompézním vyražení dveří a sražení k zemi všech přítomných osob zjistili ozbrojení kolegové, že se jim podařilo zneškodnit všehovšudy tři zaměstnankyně jakéhosi asijského bufetu. Vydal jsem se tedy zase zpátky k dodávce, třeba tam ještě něco zajímavého bude.

Nešel jsem ani pět minut a opět mě volali vysílačkou. K dodávce se asi jen tak nedostanu. Měl jsem přivést japonského chlapce v černém tričku a modrých riflích s velkým fotoaparátem. Spatřen prý byl na nedalekém náměstí.

Na náměstí skutečně ještě byl. Působil nesmírně zmateně. V okamžiku, kdy mě zahlédl, ke mně natáhl ruku s peněženkou. Netušil jsem, co tím zamýšlí, zda to jsou jeho doklady, či nějaká lest k tomu, aby mohl následně utéci. Každopádně jsem k němu přistoupil, pokusil se na něj usmát, něco mu říct a raději jej chytil jemně za rameno a pro jistotu chytil i onen důležitý foťák, aby jej v té nervozitě ještě nerozbil.

V tom se zezadu přiřítila opět ona milá novinářka a vzala z jeho ruky peněženku. Povídala, že to je její peněženka a že si to klidně mohu zkontrolovat. Učinil jsem tak a dal jí podržet chlapcův fotoaparát. Dřív, než jsem stihl zareagovat, z něj vyndala paměťovou kartu. V duchu jsem si zanadával, věděl jsem, že tyhle novinářky jsou dost mazané a šikovné na to, abych tu kartu už nikdy neviděl a radši se tvářil, že jsem si toho nevšimnul. (Krátce před tím jsem udělal ještě jeden průšvih, a kdyby se k tomu přidalo, že jsem si nechal před nosem vzít paměťovou kartu, opravdu by mi to neprospělo.)

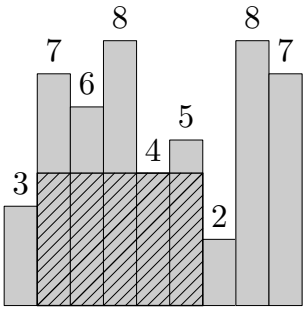
Pak jsem chlapce odvedl k nám na stanici. Na chodbě zrovna postával jeden z vyšších velitelů pražské policie, u nás na stanici jsem jej viděl asi podruhé. Vzal si ode mě chlapcův fotoaparát a řekl mi, ať se postarám o chlapce a najdu jeho rodiče.

Vzal jsem jej k nám do kanceláře, zdál se být hodně zaujat prací naší sekretářky. Zrovna přerovnávala přílohy ke spisům, především grafy.

25-3-5 Histogram — 9 bodů

Jedním z četně používaných typů grafů je histogram. Histogram je, jak praví Wikipedie, grafické znázornění distribuce dat pomocí sloupcového grafu se sloupci stejné šířky. Máte zadán histogram jakožto posloupnost N přirozených čísel udávajících výšky sloupců, šířka sloupců je jednotková. Určete obsah největšího obdélníka rovnoběžného s osami, který lze do grafu umístit tak, že celá jeho plocha leží na sloupcích histogramu.

Příklad: Pro 7 sloupců a výšky 3 6 7 4 2 3 1 je výsledný obsah 12. Existují hned 4 různé obdélníky s tímto optimálním obsahem. Další příklad s jednoznačným řešením je na obrázku:



Chlapec mi naštěstí dal kartičku pro podobné případy. Mimo jiné se na ní nacházelo číslo na japonskou ambasádu. Během telefonátu s ambasádou přišel velitel, jemuž jsem předával foťák. Byl dost naštván, že ve fotoaparátu nebyla paměťová karta a jestli o tom něco nevím, samozřejmě jsem odpověděl, že nikoliv. Sekretářka ambasády se při našem telefonátu musela dobře bavit.

Domluvili jsme se, že jim chlapce přivedu. Předal jsem jej vrátnému, ten člověk to asi s dětmi uměl lépe. Už když se pozdravili, objevil se na chlapcově tváři úsměv. Na jeho stole jsem dokonce zahlédl nějakou knihu s kresbou draka a princezny. Chlapec byl určitě v dobrých rukou.

25-3-6 Rytíř a princezny — 10 bodů

Rytíř v dalekém království se vydal na hrdinnou výpravu. Trasa jeho výpravy vypadá jako N polí uspořádaných do řádku. Na každém poli se nachází buď drak, který má u sebe H zlatých mincí, nebo princezna, která má koeficient krásy K. Rytíř se pohybuje při své výpravě z prvního políčka na poslední a je dostatečně silný na to, aby zabil kteréhokoliv draka po cestě. Je jeho volbou, zda draka zabije a získá mince, či jej obejde a ponechá drakovi život i mince.

V okamžiku, kdy přijde k princezně o kráse K, nastávají dvě možnosti. Pokud již rytíř zabil alespoň K draků, princezna se do něj zamiluje a chce si jej

vzít. Rytíř není schopen odmítnout a jeho výprava končí. Pokud rytíř zabil menší počet draků, prohodí pár zdvořilých slov s princeznou a pokračuje ve výpravě.

Na posledním políčku sídlí princezna, kterou rytíř miluje. Pro zadanou trasu výpravy určete, zda je možné získat princeznu na posledním poli. Pokud ano, vypište seznam zabitých draků tak, aby rytíř získal nejen princeznu na posledním poli, ale i co nejvíce zlatých mincí.

Konečně jsem se mohl vrátit na místo dodávky, ta už tam přirozeně nebyla. Místo jsem prohledal. V nedalekém průchodu jsem objevil svého starého kolegu a přítele, říkejme mu Jan. Jan se tam bavil s dalším mužem, pravděpodobně Japoncem. K mé smůle si mne všimli a Japonec se dal na útěk. Rozběhl jsem se za ním, ale Jan se mi postavil do cesty.

Měl s sebou obří brašnu na spisy. Vypadal opravdu vyděšeně, říkal jen: „Prosím, nech mě odejít. Nikdy jsme se tady neviděli.“ Tak jsem to i udělal. Být to kdokoli jiný, okamžitě jej zatýkám a zabavuji tyto spisy. Toto ale byl Jan – můj nejlepší přítel, kterému jsem vděčil opravdu za mnoho. V dalších částech svého vypravování se k tomu snad ještě dostanu.

⋆ ⋆ ⋆

O kauze toho později proniklo mnoho ven. Došlo i k sérii vražd, asi 2 měsíce po dni, o němž jsem vyprávěl. O 20 let později se mi podařilo dokonce dostat k odtajněným spisům GIBS a ÚOOZ. Vyšetřovalo se velmi důkladně, ale nakonec byl případ uzavřen pro nedostatek důkazů.

25-3-7 Zkratky — 7 bodů

Svět je plný zkratek, nejen těch policejních. V této úloze máte zadáno nejvýše 10 zkratek o nejvýše pěti písmenech a slovo délky L. Vaším úkolem je rozhodnout, zda lze slovo sestavit ze zadaných zkratek. Zkratky je možno spojovat za sebe a jednu zkratku lze použít i opakovaně.

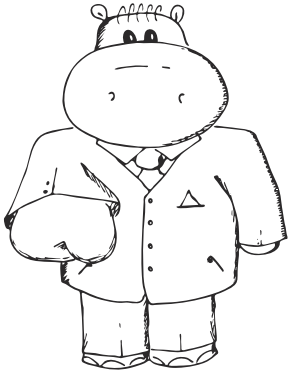
Příklad: Pokud bychom jako zkratky měli oblíbenou čtveřici *sus*, *usu*, *sss*, *Ssu* a ještě pro obohacení *su*. Pak slovo *Ssusss* postavit půjde, slovo *sususu* také a rovnou dvěma způsoby, ale slova *ssss* nebo *susSus* už nepostavíme.

S Janem jsme překvapivě zůstali v kontaktu. Říkal, že se svou nerozvážností dostal do obřího průšvihu a šlo mu o život. Domluvili jsme se, že další informace si raději nechá pro sebe. Mnoho času jsem přemítal nad tím, jaká je cena přátelství. Trápila mě otázka, zda právě tyto spisy nemohly případ rozuzlovat a třeba i zabránit dalšímu krveprolití. . .

Ke zpovědi policisty se dostal

Lukáš Folwarczný

Čtvrtá série



Deníky japonského velvyslance v ČR p. Yamady. Dešifrováno a přeloženo v NSA 2023-11-15.

To byl zase den. Nakonec všechno dobře dopadlo, ale jsem opravdu vyčerpaný. Tak vyčerpaný, že bych snad odložil dnešní zápis až na zítřek. Ale to bych měl špatné spaní a vůbec bych si neodpočinul. Na tom doporučení psychologa něco bude, vypsat se ze svých starostí. Tak tedy do toho.

Den začal jako každý jiný. Nějaké nepodstatné schůzky, podepisování, uklánění a potřásání. Ti Evropané jsou divní. Tohle musí přispívat k šíření nemocí.

Pak ale přišlo první vytržení ze stereotypu. Kódovaná zpráva od jednoho kontaktu u místní policie. Budu si muset promluvit se svými lidmi. Taková jednoduchá šifra, primitivní prohazování písmen. Takto mi ty kontakty dlouho nevydrží, někdo je určitě objeví.

25-4-1 Přesmyčky — 10 bodů

Každé slovo je zašifrované jako posloupnost písmen a číslo k. Písmena jsou ze zašifrovaného slova, jen přeházená. Nalezněte původní slovo, což je k-tá přesmyčka v lexikografickém pořadí z daných písmen. Například pro vstup

```
acb 3
```

je výsledkem `bac`, neboť přesmyčky v lexikografickém pořadí jsou:

```
abc acb bac bca cab cba
```

Pozor, písmena se mohou opakovat. V takovém případě jsou stejná písmena nerozlišitelná, tedy třeba slovo `aaa` má jedinou přesmyčku, slovo `baa` má přesmyčky tři, konkrétně `aab aba baa`.

⇧ **Lehčí varianta (za 7 bodů):** Vyřešte úlohu za předpokladu, že se písmena opakovat nemohou.

Po dekódování se o mě však pokusil infarkt. Naštěstí jsem jej kratičkou meditací zažehnal. Na víc než 5 minut jsem však čas neměl. Ve zprávě totiž stálo, že policie má tip na jeden z mých skladů zboží a chystají dnes razii. Díky varování mám naštěstí několik hodin náskok, začnu tedy plánovat, jak zachránit jak sklad, tak své lidi.

Plán byl jednoduchý. Je to totiž prodejní sklad, takže má i místnost pro styk s veřejností. A ta je maskovaná jako levné čínské bistro. Stačí tedy zboží naložit a odvézt. Až dorazí policie, najde jen kuchařku a číšnice. Jediný háček byl tedy v odvozu.

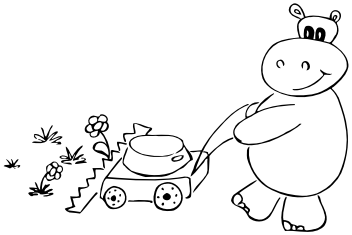
Má organizace má, samozřejmě, k dispozici dostatečné množství rychlých vozů. Pokud jsou ale naložené, musí v zatáčce přibrzdit. A to zdržuje. A čím déle budou vozy na cestě, tím větší je šance, že je někdo odhalí. Vzal jsem tedy mapu Prahy a začal plánovat trasu s co nejméně zatáčkami.

25-4-2 Plánování trasy — 9 bodů

Představme si Prahu jako čtvercovou síť. Na některých políčkách stojí překážky (budovy, policisté a podobně), těmi ostatními jde projíždět. Dále máme na mapě vyznačeno startovní a cílové políčko. Obě tato políčka jsou průjezdná.

Vůz se vydá ze startovního políčka některým ze čtyř směrů a pokračuje stále rovně, až kam to jde (tedy, kdyby pokračoval ještě jedno políčko, najel by na překážku, případně by vyjel z mapy). Zde si opět vybere jeden ze zbylých tří směrů a pokračuje, kam až to jde. Tedy, nikdy není ochotný zatočit, pokud před sebou má volné místo. Pokud se ocitne na cílovém políčku, zastaví se.

Nalezněte trasu, která obsahuje co nejméně zatáček. Z takových, pokud jich bude víc, vyberte tu nejkratší.



Po tak náročné činnosti jsem se šel projít na zahradu. Ale klid mi to nepřineslo. Napřed tam dělali hluk ti zahradníci, co sekali zahradu. A všude nechávali hrozně moc posekané trávy. Doufám, že se zítra nadřou při jejím svážení. Oni si určitě budou chtít práci usnadnit, takže bych jim měl dát za úkol posvážet trávu z nějaké části, kde to ani tak nebudou mít příliš jednoduché. Pozorovat je při práci mi totiž zítra jistě zvedne náladu.

25-4-3 Rozpis svozu — 13 bodů

Trávník je obdélníkový a rozdělený na čtverce o straně 1 metr. Víme, kolik posekané trávy se nachází na každém ze čtverců.

Jsou dány rozměry trávníku N, M. Dále dostaneme K oblastí (určených svým levým horním a pravým dolním rohem). Chceme pro každou z těchto oblastí určit nejmenší nutnou práci pro svoz trávy z celé oblasti na nějaké jedno políčko.

Práce se spočítá jako hmotnost trávy na čtverci vynásobená vzdáleností, kam se veze, s tím, že vozit smíme jen vodorovně nebo svisle. Tedy vzdálenost mezi čtverci $(0, 0)$ a $(3, 4)$ je 7, nikoliv 5.

Pro každou oblast určete políčko, kam trávu svézt, aby celková práce byla nejmenší možná, a příslušné množství práce. Pokud existuje takových nejlepších políček více, vypište libovolné z nich. Složitost algoritmu optimalizujte pro případy, kdy K je řádově stejně velké jako N.

Příklad: (První řádek obsahuje rozměry trávníku, dalších M řádků popisuje hmotnosti trávy na jednotlivých čtvercích, pak následuje číslo K a K řádků popisujících oblasti.)

```
3 5
8  2  1  5  3
6  9  1  2  7
1  1  3  2  10
3
1 1 3 3
3 2 5 3
2 1 4 2
```

Nejvýhodnější políčka s příslušnou prací jsou následující:

```
2 2 36
5 3 22
2 2 24
```

Poznámka: Pokud vám to připadá téměř jako zadání úlohy 25-3-3,[1] jen na dvojrozměrném trávníku, pak máte zcela správný pocit.

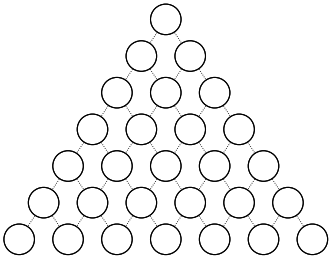
Než jsem stihl rozmyslet, kterou část jim vyberu, objevila se mi tu nějaká ženská. Nepamatuji se, že bych takové kdy poskytl audienci a rozhodně jsem to ani neplánoval. Ona však měla tolik drzosti, že se mě hned začala vyptávat na mé soukromé obchody. A, považte, dokonce japonsky. Takové neúcty bych se ve svém rodném Japonsku rozhodně nedočkal.

[1] `http://ksp.mff.cuni.cz/viz/25-3-3`

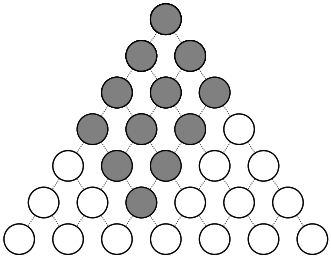
Bohužel, ve zdejších končinách není přípustné nosit samurajský meč a zastrašovat jím drzé zvědavce. Nezbylo mi tedy než sáhnout po telefonu a požádat o laskavost jednoho z mých věrných lidí u policie. Když jsem se tenkrát sázel se svým čínským kolegou, kdo dokáže podplatit i posledního policistu, nikdy jsem netušil, jak moc se to bude hodit.

25-4-4 Podplácení — 8 bodů

Policejní hierarchie je jakási pyramida. Úplně nahoře se nachází jeden kapitán. Ten má k dispozici dva nadporučíky. Dále existují tři poručíci, čtyři podporučíci, atd. až úplně dolů k řadovým policistům. Celá hierarchie o výšce 7 je schematicky znázorněna jako příklad níže. Nedivte se, že někteří policisté mají dva přímé nadřízené, v naší zemi je možné cokoliv.



V podplácení soutěží dva velvyslanci. Ten, který je na řadě, si vybere jednoho policistu, který ještě nebyl podplacen, a předá mu zavazadlo naplněné penězi. Tento policista obejde všechny své ještě nepodplacené nadřízené (přímé i nepřímé) a peníze jim spravedlivě rozdělí. Tím je podplatí. Takto by tedy vypadala hierarchie po jednom podplacení:



Vyhrává ten velvyslanec, který podplatí posledního policistu.

Pokud je zadaná výška celé hierarchie, určete, který z velvyslanců má vyhrávající strategii. Začíná Japonec.

A opravdu, po kratičké chvilce se objevil policista a zbavil mě jí. Asi jí začal docela zatápět, což je jedině dobře. Po chviličce se zcela zřejmě pokusila přechytračit policistu nějakým trikem s kalkulačkou. Nemínil jsem čekat na to, až se jí to podaří, a raději jsem zmizel opět dovnitř. Už mi v kanceláři stačili připravit čaj.

Jak jsem si tak sedal, uvědomil jsem si, že to byl úplně stejný typ kalkulačky, na jaké jsem se učil vyššímu účetnictví. V tomto účetnictví bylo mnoho kliček a triků, jak v něm schovat výdělky, do kterých nikomu nic není. Můj nejoblíbenější byl ten, že úředníci neuměli počítat s velkými čísly, proto pracovali jen se zbytky po vydělení svým inteligenčním maximem. To skýtalo opravdu mnoho možností, jak je přimět si myslet, že vlastně nikdo nevydělal nic, a tedy že ani nemá platit žádné daně.

25-4-5 Účetnictví — 12 bodů

Na začátku nemáme na účtu nic (svítí na něm tedy hezká 0). Budeme provádět transakce po dobu k dní. V i-tém dni provedeme transakci v hodnotě i. Umíme ovlivnit, jestli peníze přijdou na účet, nebo z něj odejdou.

Úředníci berňáku počítají mod n. Zvolme například $n = 50$. Máme-li na účtu 20 Kč, můžeme si na něj nechat převést od kamaráda 30 Kč a berňák si bude myslet, že nemáme nic. Kdybychom naopak z prázdného účtu kamarádovi 30 Kč odvedli, skončíme s 20 korunami.

Zajímalo by nás, kolika způsoby můžeme rozvrhnout všechny transakce tak, abychom na konci měli opět „prázdný“ účet. Počet způsobů ale roste velmi rychle, stačí tedy počet „cest z nuly do nuly“ počítat modulo $10^9 + 7$.

Odpoledne se opět neslo ve znamení nepodstatných potřásání a uklánění. Tedy, s výjimkou dvou telefonátů. Jeden mi sděloval, samozřejmě smluveným kódem, že se přesun skladiště zdařil.

Druhý telefonát oznamoval radostnou zprávu, že nové zboží z domoviny zdárně dorazilo. Samozřejmě, maskované jako skupinka turistů s foťáky. Můj nevlastní bratranec z třetího kolena Mashiro je opravdu třída. Turisté nic netušili, a dokonce jako maskování poslal i svého vlastního synovce. Jak jsem se nasmál, když mi předevčírem vyprávěl na videohovoru, že se malý Tanaka tak moc těší.

Večer byl ale opět namáhavý. Obchodní jednání s jedním z klientů. Dožadoval se množstevní slevy. Nakonec se mi takovou katastrofu podařilo zažehnat, ale jen díky tomu, že jsem ho obehrál v prastarých japonských Triádách.

25-4-6 Triády — 12 bodů

Triády jsou karetní hra. Celá pravidla jsou komplikovaná, nám bude stačit jen základní princip. Na stůl se vždy vyloží n karet. Každá karta nese k různých vlastností (kde k může být velké) a každá vlastnost může mít jednu ze 3 různých hodnot.

Trojici vyložených karet nazveme triádou, pokud se v každé vlastnosti všechny tři karty shodují, nebo se v ní navzájem liší. Pokud bude $n = 4$, $k = 3$ a vyložené karty $(1, 2, 3)$, $(1, 2, 1)$, $(1, 3, 2)$ a $(1, 1, 3)$, tak potom druhá, třetí a čtvrtá karta tvoří triádu. V první vlastnosti se shodují a ve druhých dvou se liší.

Vaším úkolem je mezi zadanými kartami najít triádu, nebo zjistit, že mezi nimi žádná není (samozřejmě rychleji než soupeř).

Tak to by byl další namáhavý den. Pro jistotu musím ještě svůj deník zašifrovat, aby se k němu nedostal někdo, kdo by jej mohl zneužít ve svůj prospěch. Jak tak prohlížím to šifrovací zařízení s knoflíky, přemýšlím, jak dlouho by někomu trvalo, než by vyzkoušel všechna možná hesla. Heslo se zadává nastavením knoflíků do správných pozic. Na mé verzi je celých 20 knoflíků, každý s 12 pozicemi. To by mohlo i takové NSA trvat aspoň 100 let, a tou dobou už to nebude můj problém.

25-4-7 Šifrovací knoflíky — 10 bodů

Šifrovací zařízení na sobě má k otočných knoflíků a každý jde nastavit do n různých pozic (knoflíky se chovají cyklicky, tedy je možné je protočit). Těmito knoflíky se nastavuje šifrovací klíč, a to jak k zašifrování, tak poté k dešifrování.

My klíč neznáme, proto bychom rádi vyzkoušeli všechny možnosti nastavení knoflíků. Nechceme se však zdržovat, a proto chceme každý klíč nastavit právě jednou. V jednom kroku umíme otočit jedním knoflíkem o jednu pozici.

Popište způsob, jak knoflíky otáčet, abychom nastavili každý klíč právě jednou. Zároveň požadujeme, aby na konci byly knoflíky ve výchozích pozicích.

⇧ **Lehčí varianta (za 5 bodů):** Vyřešte úlohu bez požadavku, aby se knoflíky vrátily do výchozích pozic.

V archivu NSA nalezl

Michal „vorner“ Vaner

Pátá série

Při tlačenici v metru se muž už po několikáté podíval na hodinky. Byl to pobočník vysoce postaveného důstojníka policie a spěchal do práce. Tedy ne že by byl pobočníkem již nějak dlouho, na tohle místo byl přidělený asi před měsícem, ale už stačil zjistit, že šéf nemá rád nedochvilnost.

Konečně dojel na Malostranskou a rychle vyběhl po eskalátorech. Dělníci stále kopali tramvajové koleje, takže i dnes se musel svézt náhradním autobusem. „Tak co, jednou přijdu pozdě. Snad jen, kdyby ten řidič byl dneska obzvláště rychlý. . . " pomyslel si při nastupování do nezvykle vypadajícího vozidla.

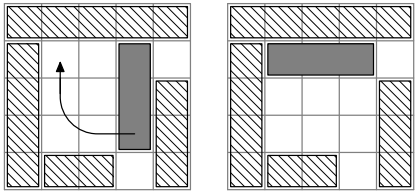
25-5-1 Cesta autobusem — 11 bodů

Dopravní podnik testuje nový druh autobusu – autobus s roztažitelnou karosérií. Bohužel vytočit se s ním v úzkých uličkách není vždy snadné a vyžaduje to velké řidičské umění.

Představte si plán města jako klasickou čtvercovou síť, volná místa představují ulice a náměstí, na zaplněných políčkách jsou domy, parky, fontány a jiné věci, přes které by autobus projíždět neměl. Autobus obsazuje několik políček za sebou (tedy je to jakýsi obdélník o šířce jedna a délce k) a může jet buď vodorovně, nebo svisle, a to oběma směry (buď jede dopředu, nebo couvá).

Na plánu města je start, cíl a navíc jsou zde nástupní a výstupní zastávky. Pokud autobus projede přes nástupní zastávku, tak se o jedno políčko do délky natáhne (proti směru, ze kterého na políčko přijel), a pokud naopak přes výstupní, tak se zkrátí. Nemůže se však zkrátit na nulovou délku. Zastávkou nelze projet dvakrát těsně za sebou, je nutné mezitím navštívit alespoň jednu jinou.

Aby se autobus mohl otočit, potřebuje dostatek místa. Otáčí se kolem některého ze svých konců, a to tak, že pokud má délku k, musí stát tímto koncem v rohu volného prostoru rozměru $k \times k$. Pak se otočí jako na obrázku a stojí ho to právě jeden krok. Přirozeně, druhý konec autobusu se také musí nacházet v onom volném prostoru.



Vaším úkolem je nalézt nejkratší cestu ze startu do cíle (nemusíte projet všemi zastávkami) takovou, aby autobus projel a měl dostatek místa na otáčení.

⇧ **Lehčí varianta (za 6 bodů):** Vyřešte to samé, ale bez zastávek (tedy autobus má jen pevnou délku k a během cesty se jeho délka už nemění).

Po klikaté jízdě skrz uličky vyběhl pobočník z autobusu a rychle běžel na stanici, kde se měl se šéfem setkat. Cestou minul na dvoře nějaký policejní nástup, místní poručík si asi přepočítával přítomné policisty. To už ale pobočník vešel do budovy a u dveří kanceláře zaslechl hlas šéfa. Zrovna domlouval nějaké odvolání nepohodlného pochůzkáře na jiné místo, jak pobočník pochopil.

„Tak na to se podívejme!“ zamumlal si potichu, aby to nikdo neslyšel. To by zapadalo do obrázku, který si o svém šéfovi udělal během toho měsíce, který u něj zatím strávil. Podivná setkání, divné zprávy, rozkazy a náhody. Začínal mít vážné podezření, že na něj šéf nehraje čistou hru. Zaslechl od kolegů nějaké zvěsti o rozrůstající se japonské mafii.

Chvíli váhal, ale pak se rozhodl. Tuhle nahrávku musel mít celou. Šéf udělal chybu a tenhle rozhovor vedl přes telefon na stanici, který byl samozřejmě nahráván. Opatrně tedy vešel do vedlejší místnosti, zavřel za sebou dveře a posadil se k počítači.

25-5-2 Telefonní ústředna — 9 bodů

Program telefonní ústředny na policejní stanici zaznamenává všechny provedené hovory. Bohužel věcí, co zaznamenává, je spousta, a proto se musí ukládat komprimovaným způsobem.

Všechny záznamy tvoří dohromady posloupnost 0 a 1 dlouhou n. Hovor je identifikovaný konkrétním vzorcem 0 a 1 o délce s, který se v posloupnosti může vyskytovat jako vybraná podposloupnost.[2]

Máme zadaný vzorec námi hledaného hovoru a ptáme se, kolik hovorů si budeme muset přinejhorším poslechnout, abychom nalezli ten pravý. Neboli kolik je možností, jak vybrat z celé posloupnosti daný vzorec.

Příklad: Pro posloupnost 010111 existuje 9 způsobů, jak v ní nalézt vzorec 011.

„Konečně, už to mám!“ zaradoval se v duchu pobočník. V tom ale málem dostal infarkt. Těžká ruka mu dopadla na rameno. Za ním se ozval šéfův hlas.

„Jane, co to tu děláte?“

„Já... pane... já...“ rychle se pokoušel schovat okno se záznamy hovorů.

Šéf si výpisu záznamů však všiml. Chvíli stál mlčky s rukou na Janově rameni. Jan skoro cítil, jak šéf v hlavě prochází spoustu možností – nedopadne Jan stejně jako předchozí pobočník, o kterém se povídá, že už ho nikdo nikdy neviděl? Pak si to šéf zjevně rozmyslel a jen suše řekl: „Pojďte Jane, půjdeme se někam najíst.“

Jan, nevěda co očekávat, ho následoval. Skoro mlčky došli do blízké restaurace. Jan přemýšlel, jestli nemá utéct, ale nějaký tajemný pocit mu říkal, že teď

[2] Vybraná podposloupnost vznikne z původní posloupnosti čísel tak, že vynecháme některé její prvky. Pořadí zbylých prvků zůstane zachováno.

šéfovi může věřit. Usadili se v osamělém rohu a objednali si jídlo. Mezitím, co Jan opatrně uždiboval špagety, začal mu šéf líčit spoustu věcí.

25-5-3 Špagety — 11 bodů

Představte si špagety v typické italské restauraci v centru Prahy. Je to jakási směs zamotaných těstovin a člověk musí hodně dávat pozor, jak je nabírat, aby si je na sebe při jídle neplácl. Proto je nejlepší odebírat špagety jen z vrchu talíře a netahat je zespoda.

Talíř špaget si můžeme představit jako trojrozměrnou mřížku. Špageta je vždy nějaký souvislý „had“ složený z trojrozměrných jednotkových krychliček, který se může libovolně kroutit. Jednotlivé špagety se v trojrozměrné mřížce vzájemně neprotínají.

Navíc máme daný směr gravitace, tedy osu, ve které budeme špagety postupně jíst. V každém kroku můžeme vzít právě ty špagety, na kterých ve směru této osy neleží žádná jiná špageta (sama na sobě však ležet může, to nám nevadí). Tím se nám uvolní některé další špagety, které zase můžeme odebrat při dalším kroku, a tak dále. Skončíme ve chvíli, kdy buď sníme celý talíř, nebo už nebudeme mít žádnou špagetu volnou.

Vaším úkolem pro zadaný talíř špaget je tedy spočítat minimální počet kroků pro snězení celého talíře a vypsat špagety odebírané v každém kroku, nebo určit, že špagety sníst nelze. Jako vstup můžete předpokládat popis celého talíře po jednotlivých souřadnicích (tedy buď se na souřadnici nachází kus nějaké určité špagety, nebo je zde volné místo).

⇧ **Lehčí varianta (za 5 bodů):** Uvažujte jen rovinnou situaci, tedy když se špagety budou proplétat jen ve dvou rozměrech a odebírat je budeme ve směru jedné z os.

„To snad nemyslíte vážně, pane!“ řekl Jan, když konečně dojedl špagety. Během uplynulých dvaceti minut se mu úplně převrátil pohled na šéfa. Žádný mafián, agent speciálního útvaru policie to byl!

„Dobře jste všechny okolo vodil za nos. A proč jste si vlastně vybral mě?“

„Inu Jane, to byla součást plánu. Mafiány nejlépe dostanete zevnitř. A proč jsem si vybral vás? Nejste z Prahy, takže vás nemůžou znát, vaše hodnocení ze služby v Brně je přímo ukázkové a můj kamarád, šéf vašeho okrsku, mi vás doporučil. A navíc jste byl rok v Japonsku a prý umíte trochu japonsky, což se možná bude hodit. Ale teď –“ náhle ho přerušil telefon.

Šéf rychle prohodil několik slov do telefonu a pak se na Jana podíval. „Ale teď provedeme pár výslechů, poručíkovi zdejšího oddělení Hamáčkovi se právě povedlo udělat razii v nějakém čínském bistru,“ zlověstně se usmál.

25-5-4 Výslechy — 11 bodů

Policii se povedlo při razii v čínském bistru zadržet několik osob. Bohužel nevíme, kdo z nich je spořádaný zaměstnanec bistra a kdo z nich je mafián. Jediné, co víme, je, že mafiáni vždy lžou a zaměstnanci bistra vždy mluví pravdu.

Máme množinu výroků dvou typů: „A tvrdí, že B je mafián“ a „A tvrdí, že B není mafián“. Protože policisté chtějí mít při rozklíčování této situace alespoň nějaká vodítka, chtějí po vás zjistit počet možných řešení, neboli počet různých způsobů, kterými lze podezřelé označit za mafiány nebo zaměstnance bistra. Počet chceme spočítat modulo nějakou konstantou K (tedy tato úloha není myšlená jako úloha na velká čísla).

Navíc byste měli poznat, pokud si výpovědi nějakým způsobem protiřečí. Přesně řečeno že neexistuje žádné možné rozdělení na mafiány a zaměstnance, které by při daných výrocích dávalo smysl (v tom případě se pak už policisté nějak zařídí).

Příklad: Pro množinu tří osob A, B a C a pro výroky: „A tvrdí, že B je mafián“ a „A tvrdí, že C není mafián“ máme jen dvě možnosti: A a C jsou mafiáni a B zaměstnanec bistra, nebo přesně naopak. Kdybychom k nim však přidali ještě osobu D, tak se nám počet možností zdvojnásobí (protože D může být v obou případech jak mafián, tak zaměstnanec bistra).

Výslechy byly zdlouhavé a táhly se až do večera. Nakonec ale Jan se šéfem zjistili něco, co se jim vůbec nelíbilo. Vypadá to, že právě teď se chystá velká dodávka nelegálního zboží.

Aby toho nebylo málo, tak hned venku přinesl nějaký rychlý posel šéfovi zprávu. Šéf si ji přečetl a pak zaklel. To bylo poprvé, co ho Jan slyšel mluvit sprostě.

„Právě dostali mého člověka. Nevím, jak se o něm doslechli, ale leží ve vážném stavu v nemocnici. Dnes večer měl domluvenou tajnou schůzku s konzulem, měl hrát prostředníka jistému bohatému podnikateli.“

Šéf chvíli přemýšlel. Pak ho něco napadlo. Vysvětlil Janovi svůj plán.

Jan chvíli přemýšlel. To, co po něm šéf chtěl, nebylo lehké. Jestli tohle vyjde, můžou nachytat celou japonskou mafii i s konzulem. Ale pokud ne... Ale co na tom, rodinu nemá, o rybičky se mu doma už někdo postará – „Jdu do toho, pane.“

* * *

V drahém obleku se Jan cítil trochu nesvůj, ale už si na něj zvykal. Jen se bez zbraně na boku a odznaku v kapse cítil jako nahý. Před chvílí navíc minul svého starého známého, jednoho z pochůzkářů. Jen tak tak, že ho nepředvedl na služebnu, když přebíral aktovku s dokumenty od jednoho kontaktu.

Teď šel rozvážným krokem k japonské ambasádě. Když už byl skoro u ní, všiml si znaveně vypadajících zahradníků. „Zajímavé, jako by celý den vozili sem

a tam trávu," podivil se. Teď už to vypadalo, že zahradu u ambasády konečně uklízí.

25-5-5 Úklid trávníku — 9 bodů

Zahradníci starající se o trávník u japonské ambasády jsou po celém náročném dni už silně unavení, ale ještě na ně čeká poslední úkol. Musí z posekané trávy vybrat reprezentativní vzorek, který pošlou do laboratoře na rozbor, jestli je trávník zdravý.

Na sběr trávy používají zmenšenou verzi balíkovacího stroje, kterým se tráva svazuje do malých krychlových úhledných balíků. Do laboratoře chtějí zaslat právě k náhodně vybraných balíků z celého trávníku, ale neví, na kolik balíků sběr vší posekané trávy vyjde.

Chtějí tedy od vás nějaký postup, jak z posloupnosti balíků neznámé délky vybrat právě k balíků. Každá k-tice balíků musí mít stejnou pravděpodobnost, že bude vybrána. Již prošlé balíky nelze vracet (nakládají se na valník a ten je odváží na kompost), tedy nelze si počet balíků nejdříve spočítat a pak teprve vybírat. Vše je nutné udělat během jednoho průchodu.

Lehčí varianta (za 4 body): Řešte úlohu pro $k = 1$, tedy pokud chceme vybrat jen jeden náhodný balík.

To už ale Jan došel na ambasádu a nechal se uvést ke konzulovi. Ještě než mohl konzul cokoliv říct, tak Jan spustil.

„Předně bych vám chtěl poděkovat, pane Yamado. Nevím, jak ten špeh ke mně proklouzl, ale příště budu své lidi mnohem více prověřovat. Jsem vám zavázán. A teď bychom se mohli věnovat započatému obchodu," uctivě se uklonil. Pokoušel se držet si sebejistou tvář, ale srdce měl strachem až v krku.

„Takže pan Kebner osobně, jsem rád, že konečně vidím vaši tvář." Luskl prsty a rázem přiskočili dva bodyguardi, kteří ho během pár sekund prohledali a pak rychle kývli na konzula. „V pořádku, chtěl jsem si být jistý. Posadíte se a dáte si se mnou partičku Triád? Můžeme nad nimi prodiskutovat tu množstevní slevu, kterou jste navrhoval."

Jan, potěšen, že mu konzul zatím věří, se s ním posadil nad herní stolek. Bohužel konzul byl příliš dobrý a Jan stále vystrašený, takže konzul snadno vyhrál. Během toho mluvili o obchodu a konzulovi nedalo příliš práce požadovanou slevu srazit na minimum. Však Janovi o peníze vlastně ani nešlo. Navzájem si také potvrdili vše, co už dříve domlouval nyní raněný agent, jen změnili data a časy. Mělo to proběhnout již dnes v noci.

Ještě než se však dostali k samotnému naplánování, přerušila je konzulova žena. „Drahý pane, můj milý muži..." pokoušela se mluvit česky, ale bylo na ní znát, že se musí hodně soustředit. „Dnes já upekla tento... dort se tomu říká u vás?" Jan přikývl. „Prosím, dejte si."

Pak potichu odešla. Jan se na dort podíval. Byl celkem malý a již nakrájený, ale docela nepravidelně. Etiketa sice vyžadovala, aby ho celý nesnědl sám, ale měl už děsný hlad, a tak se ho chtěl najíst co nejvíc.

25-5-6 Dělení dortu — 11 bodů

Kulatý dort je nakrájený na jednotlivé kousky různé velikosti, kousky mají tvar kruhové výseče. První strávník si vybere jakýkoliv kousek dortu a ten sní. Pak se postupně střídají s druhým strávníkem, dokud nesnědí celý dort. Poté, co už je odebrán první dílek, je možné odebírat pouze z okraje odebrané výseče (tedy vždy jsou na výběr maximálně dva dílky).

Uvažujte, že oba strávníci chtějí sníst co největší množství dortu a že druhý strávník vždy odebírá optimálně. Jaké největší množství dortu může první strávník sníst při použití optimální strategie?

Poté, co dojedli dort – Janovi se povedlo sníst více, což ho trochu zasytilo a dodalo mu sebejistoty – sáhl konzul někam za sebe a spustil umně ukrytý projektor. Na zdi se za chvíli objevila celkem podrobná mapa Prahy.

„Tohle už asi znáte, pane Kebnere?“ zeptal se. Jan opatrně přikývl, i když mapu v životě neviděl. Na okrajích bylo pár poznámek v japonštině, které s vypětím sil přelouskal. Popisovaly něco o rozmístění policejních hlídek.

„Teď se ale trochu mění situace. Nevím, proč to plutonium a další věci chcete, ale už si po dnešku nemůžu dovolit nechat si to ve svých skladech. Musíme to provést ještě dnes.“

„Povedlo se mi z jistého zdroje získat aktuální rozestavění policie.“ Po konzulových slovech se Jan opět podíval na mapu. V duchu si oddechl, to důležité tam scházelo. Musel naplánovat trasu předání nelegálního zboží tak, aby to konzulovi nebylo podezřelé, ale tak, aby ho dostal tam, kam potřebuje...

25-5-7 Policejní koridor — 13 bodů

Máme zadanou mapu města jako neohodnocený neorientovaný graf (křižovatky pospojované ulicemi), na některých křižovatkách stojí policejní kontroly. Dále máme zadaný start, sklad a cíl jako nějaké křižovatky a chceme vyjet ze startu, naložit věci ve skladu a dojet do cíle.

Okolo každé policejní kontroly můžeme projet za celou cestu maximálně jednou. Kdybychom okolo ní projeli vícekrát, tak už jí to přijde podezřelé, zastaví nás a podrobí nás prohlídce – a to přesně nechceme. Současně chceme cestu absolvovat co nejrychleji.

Najděte tedy pro zadanou mapu s policejními hlídkami co nejkratší cestu mezi startem, skladem a cílem tak, aby každou křižovatkou s kontrolou procházela nejvýše jednou.

Lehčí varianta (za 7 bodů): Zjistěte jen, jestli taková cesta existuje.

Janovi se konečně povedlo vymyslet trasu, která vypadala alespoň trochu rozumně. Ukázal ji konzulovi. Ten nad ní chvíli taky váhal, ale pak přikývl. „Tohle vypadá dobře. Ano. Ale pojedete s námi, ať máme jistotu.“ S tímhle Jan nepočítal, chtěl odejít. Ale teď tu operaci přece nemůže pokazit, teď už je to nutné dohrát až do konce, ať bude jakýkoliv. „Dobře, kdy vyrážíme?“

Auto se pomalu blížilo k místu setkání. Celý nákladový prostor byl zaskládán nelegálním zbožím a uprostřed něj trůnila zlověstná bedna se znaky radioaktivity na boku. Jelo pomalu, bez světel.

Na smluveném místě z něj vystoupil jeden muž. Došel doprostřed plácku ohraničeného starými průmyslovými budovami. Tam čekalo druhé auto s jedním osamoceným mužem. První muž se rozhlédl, něco mu tady nehrálo. Najednou se ozvalo několik kovových cinknutí.

Jan věděl, co čekat. Vyskočil z auta, pevně zavřel oči a přitiskl si ruce na uši. Okolí najednou zaplavilo nesnesitelné světlo a zvuk o omračující síle. Šokové granáty. Světlo zmizelo stejně rychle, jako se objevilo. Jan otevřel oči, kopem skolil jednoho z japonských bodyguardů a vrhl se do bezpečí mezi průmyslové budovy.

Plácek mezitím zaplavilo světlo z mnoha reflektorů a výkřiky policie. Mafiáni byli natolik zaskočeni, že se nikdo nezmohl na žádný odpor, nikomu nebylo ublíženo. Během několika sekund složili zbraně a policisté je odvedli.

„Dobrá práce Jane,“ ozvalo se nad ním. Byl to šéf a natahoval ruku, aby mu pomohl se zvednout. „Nechcete pro mě pracovat i dál?“

Závěr příběhu vyprávěného z různých pohledů sepsal

Jirka Setnička

Seriál o TEXu

Jan Matějka

25-1-8 Sázíme v TEXu — 13 bodů

Bylo nebylo, 30. března 1977 obdržel Donald Ervin Knuth testovací výtisk jedné ze svých knih (druhé edice druhého dílu série *The Art of Computer Programming*). Prohlásil: „Strávil jsem 15 let psaním knih, ale pokud budou vypadat takhle nechutně, tak už žádnou nenapíšu.“ Pak stáhnul knihu z tisku a napsal TEX – sázecí systém a programovací jazyk.

První a zároveň poslední stabilní verzi TEXu vydal v roce 1989. Od té doby se již celý systém prakticky nezměnil. Název se čte „tech“ nebo „tek“, neboť ono X je ve skutečnosti velké řecké písmeno chí. Pokud byste náhodou nezvládli napsat název TEX se sníženým E, tak můžete napsat TeX. Nikdy ne však TEX.

Pojďme si tedy představit program a jazyk, díky kterým letáky KSP skvěle vypadají a dobře se čtou. Naučíme se používat TEX od úplných základů, začneme obyčejnými texty, probereme se matematickými vzorci a nakonec si ukážeme i zběsilé triky. Stanete se TEXniky, jaxepatří.

Instalace

Jak začít? Máte-li Linux, je to jednoduché. Nainstalujte si „TEXovou distribuci“ TEXlive (stačí minimalistická verze) plus československá rozšíření. V Debianu a Ubuntu se jedná o balíky `texlive-base` a `texlive-lang-czechslovak`.

Ve Windows je to o něco složitější. Stáhněte si instalační balík z CTANu[3] a rozbalte jej. Máte-li dost místa na disku, můžete spustit `install-tl.bat`, odklikáte Next a nainstaluje se prakticky všechno, co byste kdy mohli i nemohli potřebovat. Zabere to přes 3 GB.

Pokud nechcete ucpat tolik místa na disku, případně nechcete stahovat tolik dat (to, co teď máte, je jen instalátor), spusťte `install-tl-advanced.bat` a v otevřeném okně si naklikejte menší instalaci. Pokud nevíte, případně se vám nad tím nechce přemýšlet, použijte tento návod:

1. Jako „Selected scheme“ vyberte „basic scheme“.
2. Z „Language collections“ vyberte československý balík.
3. TEXDIR upravte, pokud chcete změnit místo, kam bude TEX nainstalován.
4. Ujistěte se, že „Default page size“ je A4.
5. „Install TeXworks front end“ přepněte na Yes (poslední bod).
6. Klikněte na „Install TeX Live“. Instalátor stáhne z internetu všechno, co je potřeba, a nainstaluje. Celkem zabere okolo 250 MB.

[3] `http://mirror.ctan.org/systems/texlive/tlnet/install-tl.zip`

Ahoj, světe!

Vyzkoušíme si, jak se TEX spouští. Použijeme k tomu testovací vstup:

```
Ahoj světe
\bye
```

Na Linuxu si prostě otevřete nějaký textový editor (autor má rád Vim, ale klidně použijte třeba GEdit), vyrobíte soubor `ahoj.tex`, otevřete si terminál, dokráčíte do příslušné složky příkazem `cd` a spustíte `pdfcsplain ahoj.tex`, což vyrobí soubor `ahoj.pdf`, který si prohlédnete.

Pod Windows bych doporučil použít TeXworks, které jste si před chvílí nainstalovali. Nejdříve trocha nastavování:

1. V menu Edit vyberte Preferences.
2. Na kartě Editor vyberte Encoding: ISO-8859-2.
3. Na kartě Typesetting v rámečku Processing tools klikněte na tlačítko +.
4. Vyplňte `pdfCSplain` jako Name a `pdfcsplain.exe` jako Program.
5. Do pole Arguments přidejte `$synctexoption` a `$fullname`.
6. Klikněte na OK, nastavte `pdfCSplain` jako Default a zavřete Preferences klikem na OK.
7. Zavřete TeXworks a otevřete je znovu.

Nyní můžete vepsat testovací vstup. Stiskem zeleného tlačítka se šipkou vlevo nahoře jej přeložíte a zobrazíte.

Poznámky:

- TeXworks existují i pro Linux, ale já mám radši Vim. Ovšem vyberte si dle chuti sami. De gustibus non est disputandum.
- TEX se dá spustit z příkazové řádky pod Windows, ale TeXworks jsou asi o něco příjemnější.
- S případnými problémy s instalací můžeme pomoct, pokud se nám svěříte na fóru na našem webu.
- Znak \ najdete na české klávesnici na klávese Q při stisknutém pravém Altu.

Sázíme texty do odstavců

Napíšete-li do vstupního souboru libovolný text, TEX jej vysází. Vyzkoušejte si to dle libosti. Chcete-li přejít na nový odstavec, vynechejte prázdný řádek. TEX považuje libovolné nenulové množství mezer a tabulátorů za jednu mezeru.

Taktéž se polykají konce řádků, nejedná-li se tedy o dva konce řádků za sebou (ty značí konec odstavce). TEX také spolyká veškeré mezery a tabulátory na začátku a na konci odstavce.

Některé znaky není možné zadat přímo do vstupního textu, neboť je TEX interpretuje jako speciální. Jsou to znaky `#$\%&^_{}~`. Pokud chcete napsat #, %, $, &, a _, stačí předřadit zpětné lomítko: `\#`, `\%`, `\$`, `\&`, a `\_`. Stříška je považována za diakritické znaménko, takže je potřeba ji nakreslit samostatně: `\^\relax` [4] se vykreslí jako ^.

TEX používá znak \ pro tzv. řídící sekvence. Ty můžou být jednoznakové jako v minulém odstavci, nebo víceznakové složené z písmen. Za písmennými se polykají mezery; kdybyste za nimi potřebovali mezeru vynutit, použijte `\␣`.

Ostatní (\, {, } a ∼) jsou znaky, které se používají prakticky výhradně v matematickém zápise, ten nás čeká za chvíli.

Česká písmena s háčky a čárkami, stejně jako slovenská ľ, ŕ, Í, ä, ô apod. je možno napsat přímo do textu. TEX ale umí i ne-úplně-běžná písmena, třeba můžete napsat žiltovka nebo gyűrű. Existují totiž příkazy, které přidají vhodné diakritické znaménko nad/pod následující písmeno.

Vstup	*Výstup*		*Vstup*	*Výstup*	
`\‘o`	ò	*čárka dozadu*	`\.o`	ȯ	*tečka nad*
`\’o`	ó	*čárka dopředu*	`\u o`	ŏ	*půlkolečko nad*
`\^o`	ô	*stříška*	`\v o`	ǒ	*háček nad*
`\"o`	ö	*přehláska*	`\c o`	o̧	*cedilla*
`\~o`	õ	*vlnka*	`\d o`	ọ	*tečka pod*
`\=o`	ō	*čára nad*	`\b o`	o̲	*čára pod*
`\accent23 o`	o̊	*kroužek nad*			
`\H o`	ő	*dlouhá přehláska (v maďarštině)*			
`\t oo`	o͡o	*oblouček spojující 2 písmena*			

TEX automaticky dělí slova, která se nevejdou na konec řádku. Používá k tomu slovník, který je specifický pro každý jazyk. Základní je angličtina; pokud chcete nastavit češtinu nebo slovenčinu, použijte `\language\czech`, resp. `\language\slovak`.

Úkol 1 [2b]: Vysázejte tento text:

Poznatky získané cílevědomě v preadolescentním věku jsou adekvátní poznatkům pořízeným náhodně ve věku seniorském. (Co se v mládí naučíš, ve stáří jako když najdeš.)

Pokud se TEXu nepovedlo vysázet text tak, aby se vešel na řádek, objeví se za ním černý obdélník (slimák). Pak je potřeba řádek zlomit ručně, buď poradit TEXu, kde může lámat slovo, značkou `\-` (`slo\-víč\-ko`), nebo třeba přeformulovat text.

[4] `\relax` je prázdný příkaz

Chcete-li vysázet něco tučně nebo kurzívou, použijete `\bf` nebo `\it` ve skupině. Skupina je kus vstupu uzavřený mezi složené závorky `{` a `}`. Všechno, co se nastaví ve skupině, platí jen v ní, a po ukončení skupiny zase platí původní nastavení.

Takto tedy sázíme **tučně** a *kurzívou*:

```
Takto tedy sázíme {\bf tučně} a {\it kurzívou}:
```

Úkol 2 [2b]: Vymyslete (vyzkoušejte), co dělá `\rm`, a dodejte ukázkový kód.

Za zmínku stojí ještě několik českých typografických pravidel. V češtině nesmí zůstat na konci řádku jednopísmenná předložka. Pokud by ji tam TEX nechal, je potřeba mu naznačit, že následující mezera je nedělitelná. K tomu slouží vlnka: `K~tomu`, `u~lesa`.

Když píšete české „uvozovky“, použijte řídící sekvenci `\uv`: `\uv{uvozovky}`. Pozor, není možné přesahovat uvozovkami přes hranici odstavce. V takovém případě je ale obvykle lepší vyznačit dlouhou přímou řeč třeba jiným řezem písma nebo třeba zúžením okrajů.

Také stojí za zmínku pomlčky a podobné znaky. Spojujeme-li dvě slova, například „česko-slovenský“ nebo „byl-li“, napíšeme to TEXu jako jednu pomlčku.

Pokud používáme pomlčku jako interpunkční znaménko – třeba zde, zapíšeme ji jako dvě pomlčky vedle sebe a oddělíme ji na obou stranách mezerou. Totéž platí pro rozsah, například „pondělí – středa“: `pondělí -- středa`.

Pro úplnost ještě zmíníme dlouhou pomlčku `---`, která se používá zřídka – obvykle na vyznačení náhle ukončené přímé řeči.

„Prosím pozor —“ Nádražní rozhlas zmlknul, světlo zhaslo, notebook se přepnul na baterii. Vypadla elektřina.

Matematický mód

Velké přednosti TEXu jsou v sazbě matematiky. Vyzkoušejme si základní věci. Matematika se v TEXu obaluje mezi dolary (`$`), neboť v dřevních[5] dobách typografie byla její sazba velmi drahá.

Jednoduchou matematiku stačí psát. Mezery se v matematickém módu ignorují zcela, TEX je počítá podle poměrně složitého algoritmu a v drtivé většině případů vypadají vysázené vzorce hezky.

[5] Ty doby byly ve skutečnosti nejen dřevní/dřevěné, ale i olověné. Kdysi se totiž sázelo ručně, uchycovala se olověná písmenka do dřevěných rámů. Vysázet tehdy kvalitně matematiku uměl málokdo a obvykle si za to nechal mastně zaplatit. Kvalita ovšem odpovídala ceně. Tehdejšímu umění ručních sazečů se dnešní počítačová sazba ani zdaleka nevyrovná, natož pak třeba Word nebo LibreOffice.

`$a+b$`	$a+b$
`$a+b^2$`	$a+b^2$
`$a(b + c)$`	$a(b+c)$
`$abc + def$`	$abc+def$
`$a_1+a_2$`	a_1+a_2
`$a+b^{2c}$`	$a+b^{2c}$
`$a+b^2c$`	$a+b^2c$

Povšimněte si rozdílu mezi posledními dvěma řádky. Je zde využita skupina, která slouží k označení, co všechno má být horním indexem. Stejně se chová dolní index.

Horní a dolní index se vejde pod sebe, jsou-li uvedeny oba.

`$a_1^2 = a^2_1$`	$a_1^2 = a^2_1$
`$P_x^y \ne P{}_x^y$`	$P_x^y \ne P{}_x^y$
`$2^{2^{2^x}}$`	$2^{2^{2^x}}$
`$x_{y^a_b}^{z_c^d}$`	$x_{y_b^a}^{z_c^d}$
`$a', a'', a''', \dots$`	$a', a'', a''', \dots$

Zde se objevila sekvence `\dots`, která vykreslí trojtečku se správnými rozestupy teček. Je možno ji používat i mimo matematický mód … Prohlédněte si rozdíl oproti třem samostatným tečkám …

`$\sqrt{2}$`	$\sqrt{2}$
`$\sqrt{x^3-1}$`	$\sqrt{x^3-1}$
`$\overline{x+y}$`	$\overline{x+y}$
`$\overline{\overline{x}+y}$`	$\overline{\overline{x}+y}$
`$\root 3 \of 2$`	$\sqrt[3]{2}$
`$\root n^2+n+1 \of {n^2-n+1}$`	$\sqrt[n^2+n+1]{n^2-n+1}$

TEX umí v matematickém módu mnoho různých značek a symbolů. V prvé řadě umí řeckou abecedu, některá písmena dokonce ve více variantách. Vyzkoušejte:

`\alpha`, `\beta`, `\gamma`, `\delta`, `\epsilon/\varepsilon`, `\zeta`, `\eta`, `\theta/\vartheta`, `\iota`, `\kappa`, `\lambda`, `\mu`, `\nu`, `\xi`, `o`, `\pi/\varpi`, `\rho/\varrho`, `\sigma/\varsigma`, `\tau`, `\upsilon`, `\phi/\varphi`, `\chi`, `\psi`, `\omega`.

Další zajímavé značky jsou například různé relace nebo operace: $=$, $>$, $<$, $\neq$ (`\ne`), $\leq$ (`\le`), $\geq$ (`\ge`), $\sim$ (`\sim`), $\in$ (`\in`), $\subseteq$ (`\subseteq`), $\times$ (`\times`), $\setminus$ (`\setminus`), $\cap$ (`\cap`),$\cup$ (`\cup`), $\vee$ (`\vee`, `\lor`), $\wedge$ (`\wedge`, `\land`), …

Závorky se píšou svými standardními znaky s výjimkou složených závorek, zapisovaných `\{` a `\}`. Se závorkami jde dělat spousta zajímavých věcí, ale to necháme na nějakou příští sérii, pokud bude místo.

Taktéž některé zajímavé funkce mají své řídící sekvence, například sin, cos, log, min nebo max: `\sin`, ... Porovnejte: $\sin x \neq sinx$ (`$\sin x \ne sin x$`).

Poslední, co se dnes naučíme, budou zlomky. Práce se zlomky je jednoduchá, slouží k tomu sekvence `\over`, kterou zapíšeme mezi čitatele a jmenovatele. Všechno, co je vlevo, je čitatel; napravo je jmenovatel.

```
$a + b \over c + d$                                  a+b / c+d
$a + {b \over c} + d$                                a + b/c + d
${a \over b} + {c \over d} \over {e \over f + g}$    (a/b + c/d) / (e/(f+g))
```

Pokud potřebujete vysázet vzorec na samostatnou řádku, obalte jej mezi dvojité dolary: `$$a + b + c$$` se vysází takto:

$$a + b + c$$

V tomto módu se některé konstrukce sází jinak, protože mají víc místa. Vyzkoušejte si například odmocniny, zlomky nebo $\sum$ (`\sum`) a $\prod$ (`\prod`).

Poznámky:

- V matematickém módu píšeme i samostatně stojící proměnné apod. Například „x je nezávislé na funkci f při libovolné volbě parametrů a, b, c“.
- Vlnovky využijeme i zde: `proměnná~$x$` nebo `funkce~$f$`, případně `$y$~je`. Jednopísmenné vzorce, čísla apod., to všechno by mělo být přivlnovkované k nejtěsněji souvisejícímu slovu.
- Obsáhlý seznam všech značek najdete v TEXbooku [6] na stranách 434 až 439.
- Název programu (TEX) se vysází jako `\TeX`.

Úkol 3 [5b]: Vymyslete, jak vysázet tento vzorec, a dodejte zdrojový kód:

$$\frac{a^{(b-2d_c)^3}}{\sqrt{2a^{3b_1}}} + \begin{matrix}8\\7\\6\\5\\4\\3\\2\\1\end{matrix} - \sqrt{1+\sqrt{1-\sqrt{1+\sqrt{1-\sqrt{1+\frac{1-\frac{1-\frac{1-x}{1+x}}{1+\frac{1-x}{1+x}}}{1+\frac{1-\frac{1-x}{1+x}}{1+\frac{1-x}{1+x}}}}}}}}$$

Pokud se vám nebude dařit jej zkonstruovat celý, vymyslete aspoň část, budou za to také nějaké body.

[6] Donald Ervin Knuth: The TEXbook (Reading, Massachusetts: Addison-Wesley, 1984), ISBN 978-0-201-13448-9

Úkol 4 [4b]: Vysázejte TEXem řešení jiné úlohy v této sérii a dodejte zdrojový kód. Měli byste znát dost na to, aby z TEXu vypadnul rozumně hezký výstup. Pokud si nebudete vědět s něčím rady, zeptejte se na fóru na našem webu a orgové vám rádi poradí.

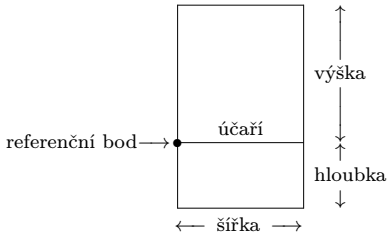
25-2-7 Zaléváme dokument 13 bodů

Seriál o TEXu pokračuje svým druhým dílem. Minule jsme se naučili základy sazby, přeložili první dokumenty a vyzkoušeli matematický mód. Tentokrát se ponoříme hlouběji do vnitřností TEXu, naučíme se psát makra a zavřeme dokument do krabičky. Matematika přijde zkrátka, v této sérii se neobjeví.

Připomínám, že preferovaný formát řešení je komentovaný zdrojový kód odevzdaný jako prostý text. Nemusíte se snažit zdroják hezky vysázet apod., jen bychom s tím měli zbytečně víc práce.

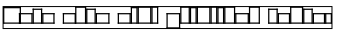
Sazba do boxů

TEX sází na stránku nepřeberné množství různých objektů. Aby se v tom vyznal, každý z nich zabalí do krabičky, a dál už pracuje jenom s ní. Všechno, co se sází, je tedy krabička – obdélníkový box, který má definovanou výšku, šířku a hloubku.



TEX vidí jednotlivá písmenka jako boxy, z nich (a výplní mezi boxy) pak staví řádky a celé stránky.

Tuto větu vidí přibližně takto:



Jednotlivé boxy se mohou skládat do horizontálních a vertikálních boxů – věty v odstavci sestávají z písmen, která se naskládají do horizontálních boxů – řádků. Řádky se pak nasypou do vertikálního boxu a z toho vznikne odstavec.

Do horizontálního boxu se tedy skládají objekty vedle sebe, kdežto do vertikálního boxu pod sebe.

TEX řeší skládání do boxů automaticky, ale přesto je občas potřeba vyrábět boxy explicitně. K tomu se můžou hodit následující primitiva:[7]

`\hbox{něco}` a `\vbox{něco}` explicitně vyrobí horizontální nebo vertikální box a do nich vloží příslušný obsah.

`\hbox to 10cm{}` je hbox široký přesně 10 cm. To znamená, že jeho obsah se natáhne nebo smrskne přesně na zadanou velikost. Pokud to TEX nezvládne, bude si stěžovat hláškou *Overfull hbox* nebo *Underfull hbox*.

Další možné jednotky délky jsou například `mm`, `in`, `pt`, případně `em` a `ex`. První čtyři jsou absolutní – jsou prostě dlouhé centimetr, milimetr, palec nebo americký typografický bod (TEX point; $\frac{1}{72{,}27}$ palce). Jednotky `em` a `ex` závisí na nastavené velikosti písma. První z nich odpovídá šířce velkého písmene M, druhá z nich odpovídá výšce malého písmene x.

Chcete-li hbox široký přesně stejně jako stránka (nezasahující do okrajů), použijte `\line{obsah}` nebo `\hbox to \hsize{obsah}`. Obojí udělá totéž. Rozměr `\hsize` určuje šířku, na kterou se sází odstavec.

`\vbox to 10cm{}` je vbox vysoký přesně 10 cm. Ještě existuje `\vtop`, který má referenční bod v referenčním bodu prvního z objektů uvnitř, kdežto `\vbox` má referenční bod v referenčním bodě posledního z objektů uvnitř.

Pozor, jakmile se uvnitř vboxu objeví odstavec, tak má vbox automaticky šířku `\hsize`.

Proč tolik řeším referenční bod, respektive účaří? Protože v drtivé většině případů se objekty skládají do horizontálního boxu tak, aby jejich účaří lícovala s účařími toho horizontálního boxu.

A proč má vlastně každý box výšku a hloubku? Sázíme-li písmena na řádek, pak některé znaky přesahují pod řádek: gjpqy – ty pak mají nenulovou hloubku. Stejně jsou na tom například závorky: ()

Módy, čáry a prázdné místo

TEX operuje v různých módech. Na začátku je ve vertikálním módu, tedy skládá boxy pod sebe. Jakmile začnete odstavec, přejde do horizontálního módu a skládá boxy vedle sebe. Když skončí odstavec (`\par`), způsobí zalámání a přejde zpět do vertikálního módu.

Taktéž uvnitř vboxu je ve vertikálním módu a uvnitř hboxu v horizontálním, jen s tou výjimkou, že uvnitř boxů jste jistým způsobem ohraničeni, například hbox se vám nezaláme.

[7] Primitivum je základní řídící příkaz TEXu. Je vhodné jej nepředefinovat, protože by se pak TEX mohl chovat podivně.

Když chcete nakreslit vodorovnou nebo svislou čáru, použijte `\hrule` nebo `\vrule`. Pozor, `\hrule` smí být použita jen ve vertikálním módu a `\vrule` jen v horizontálním.

Vodorovná čára je standardně vysoká 0.4 pt a široká stejně jako box, který ji ohraničuje (což je buď příslušný vbox, nebo celá stránka). Pokud se nám to nelíbí, můžeme to změnit: `\hrule width 2cm height 1pt depth 1pt`. Analogicky funguje svislá čára v horizontálním boxu.

Doporučuji za takovýhle příkaz napsat `\relax`, čímž zamezíte tomu, aby se TEX pokoušel interpretovat nějaký následující text jako rozměry čáry.

Nakonec, když chcete bílé místo, použijte příkaz `\vskip` nebo `\hskip` podle módu, ve kterém jste: `\vskip 15mm` vytvoří 15 mm vertikální mezeru.

Pokud potřebujete roztažitelnou mezeru, použijte `\vfil` nebo `\hfil`. Taková výplň se může roztahovat do nekonečna. Takže například centrovaný řádek vypadá takto: `\line{\hfil obsah\hfil}`. Nebo použijte konstrukci `\centerline{obsah}`, ta funguje stejně.

K boxům a výplním se ještě vrátíme ve čtvrté sérii a vysvětlíme si, jak fungují doopravdy uvnitř různých procedur TEXu.

Makra

Máte-li pocit, že nějaký kus textu nebo kódu píšete vícekrát, jsou makra přesně pro vás. Fungují podobně jako funkce v běžných programovacích jazycích.

Základní variantou je makro bez parametrů. Definuje se primitivem `\def`:

```
\def\nazevmakra{obsah {\bf makra}}
```

Na takto definované makro se pak můžete kdekoli dál odvolat pomocí `\nazevmakra`. Typické použití může být třeba takovéto:

```
\def\podpis{Karel Povolný, MFF UK\par}
Text dopisu 1
\podpis
\vfil\eject %% další stránka
Text dopisu 2
\podpis
\vfil\eject
\bye
```

Primitivum `\par` způsobí přechod na nový odstavec, stejně jako prázdný řádek.

S makry jste se už potkali v minulé sérii. Například `\TeX` je makro, které vysází název programu TEX se správně posunutými písmeny.

Každé makro je definované jen v rámci své skupiny. Vyzkoušejte, jak se přeloží například následující zdroják:

```
\def\makro{ABC}{\def\makro{DEF}\makro}\makro
```

Definicí již existujícího makra předefinujete stávající. Tím si můžete absolutně rozbít prostředí, takže je potřeba si vybírat jména, která zatím neexistují. Spolehlivý test vypadá například takto:

```
\def\isdefined#1{\ifx\undefined#1N\else Y\fi}
\isdefined{\macro}
\isdefined{\wtf} ...
```

Takové testování proveďte jednou. Ve chvíli, kdy makro definujete, si ověřte, že tím nic nezkazíte. Pak takový test zrušte, nemá smysl, zbytečně by akorát zaplevelil zdroják. Žádná budoucí verze plainu vám vaše makro nerozbije.[8]

Makra s parametry

Funkce obvykle můžou mít parametry, stejně na tom jsou makra.

```
\def\makro#1#2#3{Tohle je makro se třemi
parametry. Parametr 1 je #1, parametr 2 je #2
a parametr 3 je #3.}
\makro{kombajn}{bagr}{traktor}
```

Tohle je makro se třemi parametry. Parametr 1 je kombajn, parametr 2 je bagr a parametr 3 je traktor.

Každé makro může mít až 9 parametrů, číslovaných od 1. Na `n`-tý parametr se odkazujeme pomocí `#n`. Na každý parametr se můžeme odkázat klidně vícekrát, nebo také vůbec.

```
\def\rekl#1{Karel řekl: \uv{#1}
            Opravdu řekl: \uv{#1}}
\def\ignoruj#1#2#3{}
```

Makra s oddělenými parametry

Makra jsou mocnější zbraň, než by se mohlo zdát. Parametry totiž nemusí být jenom uzavřené ve složených závorkách. Mohou být odděleny prakticky čímkoli. Taková definice vypadá například takto:

```
\def\uloha#1: #2b{Úloha za #2 bodů: #1\par}
\uloha Třídění: 4b
\uloha Grafy: 5b
```

Parametr `#1` v uvedeném příkladu tedy požere všechno až do dvojtečky a následující mezery. A parametr `#2` požere všechno od toho místa dál až do nejbližšího b.

[8] D. E. Knuth prohlásil, že plain nebude měnit, zůstane navěky stejný.

Mějme následující definici a zkoumejme chování makra `\mc`:

```
\def\mc#1::#2:#3::#4:{(#1)(#2)(#3)(#4)}
\it
\mc::::::
\mc:a::b:c::d:
\mc a::b::c::d::
\mc::::a::
\mc::a::::
::::::
```

Výstup vypadá takto:

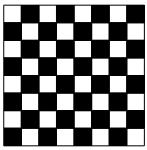
()()()() (:a)(b)(c)(d) (a)(b)(:c)(d): ()()(:a)(():a)()(:)()::::

Ještě stojí za poznámku, že je možno míchat oddělené a neoddělené parametry. Platí, že za neoddělenýmparametrem není žádný oddělovač, v definici `\def\mc#1:#2#3#4::` jsou to parametry 2 a 3, kdežto 1 a 4 jsou oddělené dvojtečkou, resp. čtyřtečkou.

Značka `#n` je zjednodušeně odkaz na příslušný parametr. Pokud byste například ale chtěli napsat „makro, které definuje makro“, možná budete potřebovat `##`, což se expanduje na jediný znak `#`. Vyzkoušejte následující kód:

```
\def\obalmakro#1#2{\def#1##1{##1#2##1}}
```

Úkol 1 [3b]: Nakreslete TEXem šachovnici 8×8. Hrana čtvercového políčka nechť je přesně 2 cm. Bodujeme hlavně preciznost a čistotu kódu. Měla by vypadat takhle, jen větší:



Úkol 2 [2b]: Definujte makro `\uloha`, které se bude volat takto:

```
\uloha 25-2-7: Zaléváme dokument (13)
```

a vysází se tak, aby výsledek vypadal co nejvíce jako hlavička úlohy v letácích KSP. Skloňování u počtu bodů můžete ignorovat, za výraz „3 bodů“ vám žádné body nestrhneme.

Kategorie znaků

TEX rozlišuje 16 kategorií znaků:

Číslo	Název	Znaky
0	Escape char	`\`
1	Open group	`{`
2	Close group	`}`
3	Math	`$`
4	Alignment	`&`
5	End of line	CR (znak s ASCII kódem 13)
6	Parameter	`#`
7	Superscript	`^`
8	Subscript	`_`
9	Ignored	znak s ASCII kódem 0
10	Space	␣
11	Letter	`a-z`, `A-Z`
12	Other	cokoli jiného neuvedeného
13	Active	`~`
14	Comment	`%`
15	Invalid	znak s ASCII kódem 127

Znaky se mezi kategoriemi dají přehazovat užitím primitiva `\catcode`. Ve skutečnosti je to 256 nezávislých 4-bitových čísel. Prostým uvedením ASCII kódu znaku za `\catcode` vybíráte jeho kategorii: `\catcode 71` odpovídá kategorii znaku G, tedy 11.

Chceme-li číslo vysázet, použijeme primitivum `\the`: `\the\catcode 64` by mělo vysázet 12 (což je kategorie znaku @). Když chceme číselnou hodnotu nastavit, použijeme následující konstrukci:

```
\catcode 64 = 13 %% Přehlednější varianta
\catcode 64 13 %% Totéž jako předchozí
```

Číslo je také možno zapsat jinak než decimálně: `'xyz` je oktalový zápis, `"xy` je hexadecimální zápis a `` `\x `` je ASCII kód znaku `x`. Tedy `'107`, `71`, `"47` a `` `\G `` znamenají totéž číslo.

Kategorie znaků mají různý význam. Escape char uvozuje řídící sekvenci, avšak nezapočítává se do ní: Je-li `\catcode `\@=0`, pak `@par` a `\par` mají úplně stejný význam.

Open group a close group slouží k uzávorkování všeho možného. Stejně jako u escape charu, nezáleží na ASCII kódu znaku, otevřít resp. uzavřít skupinu může kterýkoli znak kategorie 1 resp. 2.

Uvozovací znak matematiky už znáte z minula. Alignment se používá v tabulkových konstrukcích, to nás čeká v nějaké z dalších sérií.

Znaky end-of-line se chovají stejně jako mezera, až na to, že za EOL se ignoruje zbytek řádky. Zkuste si to v praxi sami. Je-li navíc znak EOL na začátku řádky, přeloží se na `\par`.

Parameter slouží k označení parametrů maker, také se s ním potkáme v tabulkách. Subscript a superscript se používají v matematice na horní a dolní index.

Ignored a invalid se chovají téměř stejně – jsou ignorovány. V případě invalidního znaku si ještě navíc TEX stěžuje.

U mezery se ignoruje ASCII kód a nahrazuje se bílým místem podle parametrů fontu. Mezera je totiž divný znak – všimněte si, že jako jediná může mít různou šířku podle potřeby.

Písmena a ostatní znaky se liší prakticky jedinou věcí – tím, jak se chovají za escape charem. Řídící sekvence je totiž escape char + všechny následující znaky kategorie 11, nebo escape char + jeden následující znak libovolné kategorie.

Aktivní znaky se chovají stejně jako řídící sekvence. Můžete je použít za `\def` a definovat.

A konečně znak komentáře. Od toho se ignoruje vše až ke konci řádky.

Řádky

Možná vás zarazilo, že jenom znak CR (13) je end-of-line, když na Linuxu je konec řádky znak LF (10).

TEX čte vstup po řádkách tak, jak je dostává od systému. Na Linuxu je to tedy znak LF (kód 10), na Windows dvojice znaků CR+LF (13 a 10). Systémový znak konce řádku se uřízne, ořežou se bílé znaky a na konec řádku se vloží znak CR.

Na vstupu může také probíhat netriviální překódování, obvykle se tím ale není třeba zabývat.

Tokeny

Je důležité vědět, jak se TEX vlastně chová ke svému vstupu. Každý znak, který se objeví, je tokenizován. Je určena jeho kategorie a jeho ASCII kód společně s kategorií je uložen jako token. Tokeny budeme značit takto: (znak, kategorie).

Dlužno podotknouti, že řídící sekvence se považuje za jeden token, tedy například `\par` je jen jeden token (`par`, 0).

Tento kód nefunguje tak, jak byste čekali. Zkuste si to:

```
\catcode `\@ 11 %% @ je letter
\def\mac #1@{parametr: (#1)}
\catcode `\@ 12 %% @ je other
\mac něco @
\catcode `\@ 11 %% @ je letter
\mac něco @
```

Makro totiž očekává token (`@`, 11), nikoli (`@`, 12). A tak čte tokeny jeden za druhým a čeká, jestli se neobjeví ten správný. A on se neobjeví, protože i když postupně přečte `\catcode `\@ 11` na pátém řádku, tak jsou to pro něj stále jen tokeny, které přijdou do prvního parametru...

Expanze maker

Když se na nějakém místě objeví řídící sekvence, která je definována jako makro, tak se TEX podívá, jak se má volat a jak mají vypadat parametry. Pak čte tokeny jeden za druhým, dokud nenačte všechny parametry makra.

Přečtené tokeny odstraní a místo nich vloží definici makra. V ní nahradí všechny výskyty `#n` příslušnými parametry a všechny dvojice tokenů (`#`, 6) zredukuje na jednotlivé výskyty. A pak se na to pustí hledání maker znova a znova, dokud tam nezůstanou jenom znaky a primitiva.

TEX navíc obsahuje omezení pro případ běžných překlepů (zapomenutých pravých závorek) – standardně není povoleno, aby parametr makra obsahoval `\par`. Když to potřebujete, předřaďte před definici makra `\long`:

```
\long\def\a#1{}
\def\b#1{}
\a{\par} %% projde
\b{\par} %% vyhodí chybu
```

Úkol 3 [8b]: Vymyslete, jak přepnout TEX do módu, kdy vysází na výstup (téměř) přesně to, co má ve vstupu. Hodnotí se funkčnost, čistota kódu a nápad. Nebojte se zeptat ve fóru, rádi poradíme a pomůžeme, také se tam můžete dozvědět různá doporučení a upřesnění úlohy.

Spolu s hotovým makrem dodejte také ukázkové použití – vysázené řešení nějaké jiné úlohy z této série se zdrojovým kódem. Toto řešení dodejte jako součást řešení **této úlohy**, jinak nebude hodnoceno.

Pokud by vás náhodou napadlo prozkoumat zdrojové kódy LATEXu, nedělejte to, byť by se v nich jedno možné řešení dalo najít. Jsou spletité a akorát se v nich zamotáte. Radši to zkuste vymyslet sami.

Během řešení úloh se vám ještě může hodit primitivum `\let`: Po provedení `\let\xyz\abc` má `\xyz` identický význam jako `\abc`. Používá se například na

uložení původního významu řídící sekvence, případně na „nakopírování“ makra. I když se pak změní význam původní sekvence (v uvedeném příkladě `\abc`), význam nové sekvence zůstává. Vyzkoušejte:

```
\def\abc{ABC} \abc
\let\xyz\abc \xyz
\def\abc{DEF} \abc \xyz
```

Ve skutečnosti `\let` nepřiřazuje význam řídící sekvence, ale význam tokenu, takže například můžete použít konstrukci `\let\zavinac @` a pak bude mít `\zavinac` stejný význam jako token (`@`, 12).

Také se vám při definování maker můžou hodit primitiva `\begingroup` a `\endgroup`. Hodí se ve chvíli, kdy potřebujete například jedním makrem otevřít a jiným pak zavřít skupinu, neboť definice makra musí být dobře uzávorkovaná.

Pozor, toto je jiný typ skupiny než ta, která je ohraničena tokeny (*, 1) a (*, 2).[9] Takže skupina otevřená primitivem `\begingroup` musí být uzavřená pomocí `\endgroup`, jinak si TEX stěžuje (a obráceně skupina otevřená tokenem kategorie 1 musí být uzavřená tokenem kategorie 2).

Toť protentokrát vše. Přeji mnoho štěstí při definování maker. Dotazy a doplnění posílejte do fóra, stejně jako v první sérii.

[9] U tokenů kategorie 1 a 2 nezáleží na kódu znaku, proto jsou uvedeny hvězdičky.

25-3-8 Tabulatika — 13 bodů

Třetí díl seriálu o TEXu bude snad méně děsivý než druhý. Nebojte, budeme se věnovat „jenom“ tabulkám, pokročilé matematice a podrobnostem sazby odstavců.

Sazba na tabulátory

Nejjednodušší tabulky můžeme sázet jednoduchým způsobem. Následující konstrukcí rozdělíme stránku na 3 stejně široké sloupce a do nich sázíme řádky:

```
\settabs 3\columns
\+Sloupec 1&Sloupec 2&Sloupec 3\cr
\+&Jen druhý\cr
\+První&&a třetí\cr
\+Vykřičník je ve čtvrtém sloupci~-- mimo&&&!\cr
\+První sloupec je příliš dlouhý&
  a druhý se přesází přes něj.\cr
\+Mezery za \&&     se ignorují.\cr
\+\hfill Tento řádek&\hfill je zarovnán
  &\hfill na pravý okraj.&\cr

Ukončení tabulky se nijak zvlášť neřeší.
```

Sloupec 1	Sloupec 2	Sloupec 3
	Jen druhý	
První		a třetí
Vykřičník je ve čtvrtém sloupci – mimo		!
První sloupec je příliš dlouhý	a druhý	přesází přes něj.
Mezery za &	se ignorují.	
Tento řádek	je zarovnán	na pravý okraj.

Ukončení tabulky se nijak zvlášť neřeší.

Poznámka: Důvod, proč zde má `\hfill` najednou dvě L, vysvětlíme v tomto díle seriálu v kapitole o různých typech lepidla.

Nelíbí se vám stejně široké sloupce? Vymyslete si vzorový řádek, podle kterého TEX nastaví šířky sloupců:

```
\settabs\+První sloupec &Druhý sloupec
         &Třetí sloupec &Zbytek&\cr
\+A&B&C&D&E\cr
\+První sloupec &Druhý sloupec
 &Třetí sloupec &Zbytek&!\cr
```

A	B	C	D	E
První sloupec	Druhý sloupec	Třetí sloupec	Zbytek!	

Vzorový řádek se nezobrazí, jen se podle něj nastaví šířka sloupců.

Úkol 1 [2b]: Vysázejte pomocí tabulátorů tuto jednoduchou tabulku z knihy jízd:

Odkud	*Kam*	*Kdy*	*Kolik km*
Praha	Olomouc	21. 12.	250
Olomouc	Uherské Hradiště	30. 12.	130
Uherské Hradiště	Vyšší Brod	5. 1.	350
Vyšší Brod	Jablonec nad Nisou	17. 2.	324

Správně odsazený zdrojový kód

Tabulátory vůbec nemusí být extra pevné. Můžeme zrušit všechny tabulátorové pozice vpravo od aktuální „buňky“ příkazem `\cleartabs`. A pokud použijeme mezi `\+` a `\cr` znak `&` na místě, kde ještě není definovaná tabulátorová pozice, tak se ta pozice jednoduše nadefinuje právě na ono místo.

Víc asi ukáže příklad:

```
\cleartabs
\+{\bf if} $x<0$: &{\bf if} $x<-1000$:
  &{\it print} \uv{$x$ je echt záporné}\cr
\+&{\bf else}:
  &{\it print} \uv{$x$ je trochu záporné}\cr
\+{\bf else}: &\cleartabs{\bf if} $x>0$:
  &{\it print} \uv{$x$ je kladné}\cr
\+&{\bf else}: &{\it print} \uv{$x$ je nula}\cr
```

if $x < 0$: **if** $x < -1000$: *print* „x je echt záporné“
else: *print* „x je trochu záporné“
else: **if** $x > 0$: *print* „x je kladné“
else: *print* „x je nula“

Tabulky

Při sazbě na tabulátory je potřeba odhadnout, který sloupec bude jak dlouhý, a podle toho nastavit šířku sloupců. Také pokud chcete tabulku s orámováním, nemáte moc rozumných možností. TEX však nabízí mocnější nástroj než sazbu na tabulátory – primitivum `\halign`.

Tabulku z **úkolu 1** vysázíme primitivem `\halign` takto:

```
\halign{
# \hfil&# \hfil&# \hfil& \hfil#\cr
\it Odkud&\it Kam&\it Kdy&\it Kolik km\cr
Praha&Olomouc&21. 12.&250\cr
Olomouc&Uherské Hradiště&30. 12.&130\cr
Uherské Hradiště&Vyšší Brod&5. 1.&350\cr
Vyšší Brod&Jablonec nad Nisou&17. 2.&324\cr
}
```

Tabulka se skládá z jednotlivých řádků oddělených od sebe značkou `\cr`. Buňky se od sebe oddělují znakem `&` (nebo jiným znakem kategorie 4 – alignment).

První řádek obsahuje vzor. V každé buňce musí být znak `#` (kategorie 6 – parameter) právě jednou (jinak vám TEX vynadá), jinak může být vzorem prakticky libovolný kus TEXu, který se dá použít uvnitř hboxu (třeba `\bye` není povolený příkaz). Buňky se pak sází tak, že se v příslušném vzoru nahradí výskyt znaku `#` za obsah buňky.

Dejte si pozor na mezery – mezery za `&` se ignorují, mezery před `&` ale ne.

Tabulky a boxy

Při sazbě tabulky si TEX zjistí pro každý sloupec, jak bude široký, a podle toho nastaví šířku všem jeho buňkám. Každá buňka tabulky je separátní hbox široký právě tak jako celý sloupec. Pokud nechcete mít ošklivě roztažené mezery v textu, vložte na vhodné místo `\hfil`-y, vizte příklad výše.

Očárovaná tabulka

Potřebujete-li do tabulky vložit vodorovnou čáru (nebo libovolný jiný materiál), využijte prostředí `\noalign`. Tím začne vertikální box šířky přesně takové, jak je široká celá tabulka:

```
\halign{
\strut# \hfil&# \hfil&# \hfil& \hfil#\cr
\it Odkud&\it Kam&\it Kdy&\it Kolik km\cr
\noalign{\hrule\smallskip\line{\hfil
    2012 \hfil}\smallskip\hrule\smallskip}
Praha&Olomouc&21. 12.&250\cr
Olomouc&Uherské Hradiště&30. 12.&130\cr
\noalign{\hrule\smallskip\line{\hfil
    2013 \hfil}\smallskip\hrule\smallskip}
Uherské Hradiště&Vyšší Brod&5. 1.&350\cr
Vyšší Brod&Jablonec nad Nisou&17. 2.&324\cr
}
```

Odkud	*Kam*	*Kdy*	*Kolik km*
	2012		
Praha	Olomouc	21. 12.	250
Olomouc	Uherské Hradiště	30. 12.	130
	2013		
Uherské Hradiště	Vyšší Brod	5. 1.	350
Vyšší Brod	Jablonec nad Nisou	17. 2.	324

Potřebujeme-li i svislé čáry, je náš úkol o poznání složitější. TEX totiž vkládá každý řádek tabulky samostatně do stránkového vboxu jako jednotlivé hboxy. Takže mezi ně vloží `\lineskip` nebo `\baselineskip`. Proto je musíme vypnout a výšky řádků nastavit explicitně. Na vypnutí lineskipů „pořádně a důkladně“ použijte makro `\offinterlineskip`, které myslí i na zběsilé okrajové případy.

Aby byl každý řádek stejně vysoký, je potřeba do něj vložit vzpěru. Jinak by sazba vypadala ošklivě. K tomu se hodí makro `\strut`, které je v Plain TEXu definováno přibližně takto:

```
\def\strut{\vrule height 8.5pt
           depth 3.5pt width 0pt\relax}
```

Odkud	*Kam*	*Kdy*	*Kolik km*
	2012		
Praha	Olomouc	21. 12.	250
Olomouc	Uherské Hradiště	30. 12.	130
	2013		
Uherské Hradiště	Vyšší Brod	5. 1.	350
Vyšší Brod	Jablonec nad Nisou	17. 2.	324

```
{\offinterlineskip
\def\higher{\vrule height 11pt
            depth 3.5pt width 0pt\relax}
\halign{%
\strut# \hfil&# \hfil\vrule\ &# \hfil& \hfil#\cr
\it Odkud&\it Kam&\it Kdy&\it Kolik km\cr
\noalign{\hrule\smallskip\line{\hfil
    2012 \hfil}\smallskip\hrule}
\higher Praha&Olomouc&21. 12.&250\cr
Olomouc&Uherské Hradiště&30. 12.&130\cr
\noalign{\hrule\smallskip\line{\hfil
    2013 \hfil}\smallskip\hrule}
\higher Uherské Hradiště&Vyšší Brod&5. 1.&350\cr
Vyšší Brod&Jablonec nad Nisou&17. 2.&324\cr
}}
```

Ještě se vám můžou hodit dvě tabulkové operace: `\omit` na začátku buňky dočasně nahradí vzor pro tuto buňku za prosté `#`.

Místo `&` v běžném řádku můžete použít `\span`. V tu chvíli se příslušné dvě buňky spojí. To, co bylo před `\span`-em, se vloží na místo prvního `#`, a to, co je za `\span`-em, se vloží na místo druhého `#`.

```
{\offinterlineskip
\halign{
\strut1A#1B\hfil\vrule&\ 2A#2B\hfil\vrule
   &\ 3A#3B\hfill\cr
x&y&z\cr
\omit\strut x x&y\span z\cr
\omit\strut\hfil x\span\omit y&z\cr
xxx& yyy& zzz\cr
x\span y\span z\cr
\omit\span\omit\span\omit
   \strut\hfil abcde\hfil\cr
}}
```

```
1Ax1B   | 2Ay2B   | 3Az3B
x         x 2Ay2B| 3Az3B
                    xy 3Az3B
1Axxx1B| 2Ayyy2B| 3Azzz3B
1Ax1B| 2Ay2B| 3Az3B
            abcde
```

Podrobnosti o tom, jak se vlastně TEX v tabulkách chová, si najděte například v TEXbooku v kapitole 22 (nebo v TBN[10] v kapitole 4) – překračuje to rámec tohoto seriálu.

Úkol 2 [6b]: Definujte makra pro sazbu výsledkové listiny KSP. Zdroják výsledkovky pak může vypadat například takto:

```
\vysledkovka{
\radek 1. Petr Pilný (GABCD; 4; 7):
       12 7 - 4 9 5 - 7: 49,4 69,3
\radek 2. ...
}
```

Pokud se vám nelíbí současný desing naší výsledkovky, navrhněte lepší a přehlednější.

Poznámka: Pokud byste potřebovali tabulku ne po řádkách, ale po sloupcích, zkuste `\valign`. Funguje to stejně jako `\halign`, jenom otočeně.

Periodická hlavička

Zdvojíte-li v hlavičce na nějakém (nejvýše jednom) místě `&`, říkáte tím TEXu: „Zde začíná periodická část hlavičky.“ Tedy pokud by v nějakém řádku došly vzory pro buňky, tak začne recyklovat vzory od `&&` dál:

[10] Petr Olšák: TEXbook naruby, Konvoj Brno 2001 (2. vydání), ISBN 80-7302-007-6

```
\halign{1# & 2# && 1P# & 2P# \cr
A&B&C&D&E&F&G&H&I&J&K&L\cr
A&B&C&D&E&F&G\cr
}
```

1A 2B 1PC 2PD 1PE 2PF 1PG 2PH 1PI 2PJ 1PK 2PL
1A 2B 1PC 2PD 1PE 2PF 1PG

Tolik k tabulkám.

Matematické závorky

Ve druhé části poněkud rozkouskovaného třetího dílu se budeme věnovat složitějšímu využití matematického módu.

Složité vzorce jsou často různě zběsile vysoké a jedna velikost závorek nestačí. Prohlédněte si příklad:

$$\left(\sqrt{(x+\sqrt{x-1})\left(\sum_{k=1}^{\infty}\frac{1}{k^2}\right)}\right)$$

TEX si umí určit velikost závorek sám, jen je potřeba mu říct, která patří ke které. Používají se k tomu primitiva `\left` a `\right`, kterými se označí příslušné závorky:

```
$$\left(\sqrt{
    \left(x + \sqrt{x - 1}\right)
    \left(\sum_{k=1}^\infty {1 \over k^2}\right)
  }\right)$$
```

TEX sám zvolí velikost závorek takovou, aby byly vhodně vysoké. Pokud byste potřebovali, aby se jedna ze závorek nezobrazila, použijte místo ní tečku:

```
\left({x\over y}\right.
```

$$\left(\frac{x}{y}\right.$$

Typů závorek je hrozná spousta, Plain určitě zná tyto:

```
$$()[]\{\}\vert\Vert\lceil\rceil
  \lfloor\rfloor\langle\rangle
  \backslash/\uparrow\downarrow
  \Uparrow\Downarrow\updownarrow\Updownarrow$$
```

$$()[]\{\}|\|\lceil\rceil\lfloor\rfloor\langle\rangle\backslash/\uparrow\downarrow\Uparrow\Downarrow\updownarrow\Updownarrow$$

Pokud byste čirou náhodou potřebovali použít jiné závorky, konzultujte TEXbook, kapitolu 17 a následující (nebo TBN, kapitolu 5).

Může se vám však stát, že potřebujete jinak velkou závorku, než vám dá TEX. V takovém případě jsou pro vás připraveny varianty `\big`, `\bigg`, `\Big` a `\Bigg`:

```
$\Bigg(\bigg(\Big(\big(()\big)\Big)\bigg)\Bigg)$
```

$$\Bigg(\bigg(\Big(\big(()\big)\Big)\bigg)\Bigg)$$

Zákulisí matematické sazby

Ani celý rozsah seriálu by zdaleka nevystačil na vyložení sazby matematiky v celé její kráse a síle. Donald Knuth na to v TEXbooku vynaložil přes 60 stran. Naznačíme si však některé principy, které by vám při pokročilé sazbě neměly zůstat utajeny.

TEX umí velmi sofistikovaně rozložit mezery do matematického vzorce, pokud ví, jakého typu je která část vzorce. Rozlišuje těchto 13 typů:

Ord	Standardní „atom“	x či 1
Op	Velký operátor	$\sum$ či $\int$
Bin	Binární operátor	$+$ či $-$
Rel	Relace	$=$ nebo $<$
Open/Close	Začátek/konec závorky	$(/)$, $[/]$, …
Punct	Interpunkce	čárka, středník
Inner	Složitější konstrukce	zlomek
Over/Under	Atom s čárou nad/pod	$\overline{x}$, $\underline{x}$
Acc	Atom s „diakritikou“	$\hat{x}$, $\vec{x}$
Rad	Cosi pod odmocninou	$\sqrt{2}$
Vcent	Výstup z `\vcenter`	

Poznámka: PlainTEX má v matematice definována tato diakritická znaménka: `\acute` ($\acute{a}$), `\bar` ($\bar{a}$), `\breve` ($\breve{a}$), `\check` ($\check{a}$), `\ddot` ($\ddot{a}$), `\dot` ($\dot{a}$), `\grave` ($\grave{a}$), `\hat` ($\hat{a}$), `\vec` ($\vec{a}$), `\tilde` ($\tilde{a}$), `\widetilde` ($\widetilde{abc}$), `\widehat` ($\widehat{abc}$). Používají se stejným způsobem jako nematematická.

Poznámka: `\vcenter` je explicitní vbox v matematickém módu, který je stejně vysoký jako hluboký, takže se vysází centrovaný na osu. Vůbec, matematika se sází ne na účaří, ale na pomyslnou vodorovnou osu, neboť vzorce jsou často takto symetrické. Ale toho už jste si jistě všimli.

Mezi jednotlivými kategoriemi je pak definována tabulka, která určuje, kam patří jak velká mezera. Tato tabulka je pro drtivou většinu použití dostačující.

Pokud vám připadá, že TEX nějakou mezeru určil chybně, zkuste označit atomy v jejím okolí jejich typem. K tomu použijte `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, `\mathpunct` a `\mathinner`. Zbytek typů se značí automaticky a nelze je označit ručně.

Pokud selže i označení atomu vhodným typem, můžete použít mezery různých velikostí – `\,`, `\;` nebo `\!`:

```
$$x\!y\quad xy\quad x\,y\quad x\;y$$
```

$$xy \quad xy \quad x\,y \quad x\;y$$

Matematický mód se dělí na několik dílčích módů. Všimněte si, jak se vysází různé výrazy na různých místech – `$$\sum$$`, `$\sum$`, `$x^{\sum}$` či `$x^{x^{\sum}}$`. Nejde jen o velikost, ale také o umístění indexů a rozložení mezer.

Ony dílčí módy jsou D, T, S a SS: „display“, „text“, „script“ a „scriptscript“ uvedené ve stejném pořadí jako v předchozím odstavci.

Chcete-li si vyzkoušet víc magie okolo módů, poradím primitivum `\mathchoice`:

```
\def\te{{\mathchoice{D}{T}{S}{SS}}}
$$\te {\te^{\te^\te} \over \te^{\te^\te}}$$
```

$$D\frac{T^{S^{SS}}}{T^{S^{SS}}}$$

Nebo můžete vynutit konkrétní mód použitím `\displaystyle`, `\textstyle`, `\scriptstyle` a `\scriptscriptstyle`.

Tabulkové konstrukce v matematice

Plain TEX definuje některé maticové konstrukce:

```
$$\pmatrix{a_{11}&a_{12}&\ldots&a_{1n}\cr
           a_{21}&a_{22}&\ldots&a_{2n}\cr
           \vdots&\vdots&\ddots&\vdots\cr
           a_{m1}&a_{m2}&\ldots&a_{mn}\cr}$$
```

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

Použití je obdobné jako u tabulek, jen nemusíte definovat vzorový řádek. Matici bez závorek můžete získat použitím `\matrix`. A na matici s okraji je definováno makro `\bordermatrix` (vyzkoušejte sami).

Úkol 3 [3b]: Vysázejte tento vzorec:

$$Z = \begin{cases} N > 0: & \displaystyle\sum_{i=1}^{N} \left(2 \sum_{i<j} \log |\lambda_i - \lambda_j| - \sum_{i=1}^{N} V(\lambda_i) \right) \\ N < 0: & -\dfrac{1}{N^2} \\ N = 0: & 0 \end{cases}$$

Sázíme odstavec

Další kousek třetího dílu věnujeme podrobnějšímu náhledu na algoritmus lámání odstavce.

Když se na vstupu objeví něco, co by mělo začít odstavec (znak, `\indent`, ...),[11] zkontroluje TEX několik věcí. Na začátek horizontálního boxu, který bude později zalámán na jednotlivé řádky v odstavci, se vloží prázdné místo velikosti `\parindent` (pokud nebyl odstavec zahájen pomocí příkazu `\noindent`).

Pak se expanduje `\everypar`, což je seznam tokenů (pseudomakro), které se má vložit na začátek každého odstavce. Přiřazuje se do něj zavoláním `\everypar={něco}`. Standardně je prázdný. Pak se do horizontálního boxu postupně vkládají znaky, dokud se neobjeví `\par`.

Nyní se na úplný konec boxu vloží prázdné místo velikosti `\parfillskip` (standardně `\hfil`). Pak se TEX pokusí zalámat odstavec s přihlédnutím k nastavenému tvaru odstavce. Každý řádek musí obsahovat nejprve prázdné místo velikosti `\leftskip`, pak příslušný kus odstavce a pak prázdné místo velikosti `\rightskip`. Dohromady je vždy široký přesně `\hsize`.

Poznámka: Pro zjednodušení nyní vynecháváme možnost úpravy tvaru odstavce primitivem `\parshape` a nastavením `\hangindent` a `\hangafter`, které nás čekají v další sérii.

TEX se pokouší zalámat odstavec celkem třikrát. Nejprve v mezerách mezi slovy. Pokud mu to nejde, zkusí rozdělit slova podle pravidel pro dělení slov. Pokud se mu ani toto nepovede, zkusí nepatrně roztáhnout mezery a rozdělit slova. Pokud selže i poslední pokus, vyhlásí chybu, nechá nějaký řádek přetéct a vykreslí slimáka.

[11] Primitivum `\indent` explicitně započne odstavec; `\indent\indent` si vyžádá dvojité odsazení (možno opakovat vícekrát pro vícenásobné odsazení). A konečně explicitní začátek odstavce bez odsazení vynutíte primitivem `\noindent`.

Lámání odstavce probíhá najednou, TEX se snaží, aby nebyl jeden řádek ošklivě stažený a hned ten další ošklivě roztažený – má nějaký základní smysl pro estetiku. Nicméně když při druhém a třetím pokusu dělí slova, musí vědět, kde všude může dělit, jinak může odstavec vyjít zbytečně ošklivě. To je ten zásadní důvod, proč jsem v první sérii všem řešitelům bez `\language\czech` strhával body.

Přesný algoritmus včetně všech podrobností, které jste nikdy nechtěli znát, najdete v TEXbooku v kapitole 14 (nebo v TBN v kapitole 6).

Lepidlo

Prázdné místo je v sazbě důležitá věc. Někdy se s nadsázkou dokonce říká, že celá typografie je věda o prázdném místě. V poslední části tohoto dílu seriálu se naučíme pracovat s jeho roztažností.

TEX nahlíží na prázdné místo jako na kusy „lepidla“, které se může různě roztahovat a smršťovat. Český odborný název je „výplněk“. Je několik druhů roztažnosti:

Pevný výplněk svoji velikost nemění. Je vždy stejně velký:

```
\hbox{A\hskip 2cm\relax B}
```

A B

Omezeně roztažitelný výplněk má základní velikost a meze, kam až se může roztáhnout.

```
\hbox to 15mm{A\hskip 2cm plus 1cm minus 1cmB}
\hbox to 25mm{A\hskip 2cm plus 1cm minus 1cmB}
```

A B
A B

Roztažnost nemusí být na obě strany stejná: `10mm plus 5mm minus 3mm`

Nekonečně roztažitelný výplněk má základní velikost a jinak se může roztáhnout neomezeně podle potřeby.

```
\hbox to 5cm{A\hskip 2cm plus 1fil\relax B}
\hbox to 1cm{A\hskip 2cm minus 1fil\relax B}
```

A B
A B

Nekonečných roztažností jsou tři druhy: `fil`, `fill` a `filll`. Čím víc L, tím agresivněji se roztahuje.

Jeden výplněk je nuda. Co kdyby se někde potkalo více výplňků? Když TEX skládá box, u kterého dopředu neví, jak bude velký, tak se vůbec roztažností neřídí. Ideální je vždycky, když se nic natahovat nemusí, takže použije vždy

pouze základní velikost. V opačném případě má nařízeno, jak musí být výsledný box velký, a musí se do něj chtě nechtě vejít.

Pokud je nutné roztahovat a stahovat bílé místo, TEX určí správné mezery přibližně takovýmto algoritmem:

1. Zjistí, jakou velikost by měl box, kdyby se použily základní velikosti. Určí rozdíl mezi požadovanou a základní velikostí. Je-li rozdíl kladný, bude nás dále zajímat pouze kladná roztažnost; je-li rozdíl záporný, budeme řešit pouze zápornou roztažnost. Tento rozdíl si označme jako w.
2. Sečte povolenou roztažnost přes celý box – zvlášť omezenou a zvlášť každý druh nekonečné roztažnosti.
3. Vybere nejagresivnější roztažnost, která je nenulová, a tou se bude zabývat.
4. Rozpočítá w mezi všechny výplňky, které přispěly do roztažnosti, podle poměru, ve kterém přispěly.

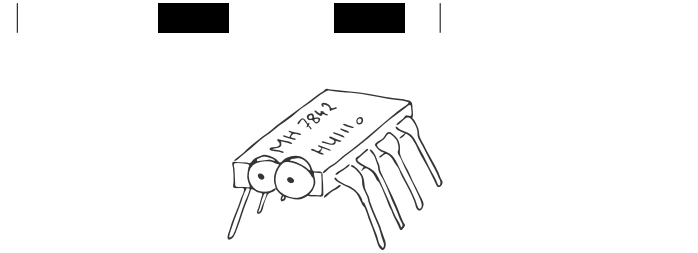
Vizte příklad:

```
\hbox{\vrule
\hbox to 6cm{\strut%
  \hskip 5mm plus 3cm
  \vrule width 1cm
  \hskip 1cm plus 1cm minus 1cm
  \vrule width 1cm
  \hskip 5mm minus 2cm
}\vrule}
```

Vnitřní hbox má být široký 6 cm. Základní šířka boxu vyjde $0{,}5 + 1 + 1 + 1 + 0{,}5 = 4$ cm, $w = 6 - 4 = 2$ cm. Zajímá nás tedy kladná roztažnost. Čáry se neroztahují, řešíme tedy jen skipy: $3 + 1 + 0 = 4$ cm, jiná roztažnost není.

Potřebujeme tedy rozdělit 2 cm mezi první a druhý skip v poměru 3 : 1, tedy prvnímu skipu se přidělí 15 mm a druhému 5 mm. Box se tedy vysází, jako by byl zadán takto:

```
\hbox to 5cm{\hskip 20mm\vrule width 1cm
  \hskip 15mm\vrule width 1cm\hskip 5mm}
```



Druhý příklad bude složitější:

```
\hbox{\vrule
\hbox to 5cm{%
  \hskip 5mm plus 1fil
  \vrule width 1cm
  \hskip 1cm plus 2cm minus 1cm
  \vrule width 1cm
  \hskip 5mm minus 1fil
}\vrule}
```

Základní šířka boxu je 5 cm, obsah má základní šířku 4 cm, $w = 1$ cm. Celková kladná roztažnost je 2 cm + 1 fil, takže nás zajímá jen 1 fil. Prvnímu skipu se tedy přidělí celý 1 cm.

Rozmyslete si, co se stane, když:

- poslední hskip nebude mít `minus 1fil`, ale `plus -1fil`;
- první hskip bude mít `plus 1fill`;
- bude hbox `to 4cm` nebo `to 3cm`.

Nyní je čas na důležitou poznámku. Roztažnost mají pouze skipy (výplňky), mezi které se počítají i běžné mezery mezi slovy. Všechno ostatní (boxy, čáry) má pevnou velikost, jakmile je to vytvořeno. Není možné vytvořit box, jehož rozměry by byly pružné podle jeho okolí. Najděte si třeba ve své sazbě řádek s hodně roztaženými mezerami a část toho řádku uzavřete do hboxu. Všimněte si, co se stane s mezerami, a zkuste si rozmyslet, proč to tak může být.

Připomínám, že je nanejvýš vhodné si vyzkoušet práci s tabulkami i pokročilou matematikou na uvedených příkladech. Zkuste je dál modifikovat a hrát si s nimi, ať si to všechno důkladně zažijete. Příště budeme konečně programovat.

Úkol 4 [2b]: Vymyslete nastavení parametrů odstavce tak, aby se zarovnal na střed, přibližně tak, jako je zarovnáno zadání tohoto úkolu.

Řešení nemusí být univerzální, stačí, aby se zadaný odstavec povedlo vysázet v běžném případě. Nemusíte řešit okrajové případy, kdy je odstavec extrémně krátký apod.

A to bude ze třetí série vše. Těším se na vaše řešení.

25-4-8 TEXgramy 14 bodů

Ve čtvrtém dílu seriálu o TEXu si ukážeme pokročilé programování. Potkáte proměnné, podmínky, čtení souboru i zápis. Naučíme se, jak automaticky číslovat nadpisy, tabulky, obrázky i cokoli jiného.

Čísla, rozměry a skipy

TEX nabízí uživateli 256 celočíselných registrů, ke kterým se přistupuje primitivem `\count` stejně jako ke `\catcode`. S číselným registrem se dá pracovat stejně jako s `\catcode`, jen `\count` má větší rozsah (32bitové celé číslo se znaménkem).

Na ukládání rozměrů poslouží `\dimen`. Je to ve skutečnosti také celočíselný registr, jehož základní jednotkou je ovšem 1 sp (1 pt = 65536 sp). TEX nepracuje s rozměry většími než 2^{30} sp = 16384 pt, což je něco přes 570 cm, takže by vám to do začátku mělo stačit, pokud zrovna nenavrhujete billboard obludných rozměrů.

Registry typu `\skip` pak slouží k ukládání pružných rozměrů (s roztažností). Jejich omezení je stejné jako u pevných rozměrů.

Kromě vypisování primitivem `\the` a přiřazení umí tyto typy registrů také základní aritmetické operace. Primitivum `\advance` například slouží ke sčítání.

```
\count0=1 % přiřaď 1 do \count0
\advance\count0 by 1 % zvyš o 1
\the\count0 % vysázej 2
\advance\count0 by -15 % sniž o 15
\the\count0 % vysázej -13
\dimen0=1in
\advance\dimen0 by 1cm
\the\dimen0 % vysázej 100.72273pt
```

Klíčové slovo `by` je možno vynechat, ale pro přehlednost se hodí.

Celočíselné registry umí také celočíselně násobit a dělit, stejně tak rozměry a skipy.

```
\count0=30
\multiply\count0 by 5
\the\count0 % vysázej 150
\divide\count0 by 7
\the\count0 % vysázej 21

\skip0=1pt plus 2pt minus 3pt
\multiply\skip0 by 3
\the\skip0 % 3pt plus 6pt minus 9pt
```

Rozměry můžeme také násobit číselnou konstantou uvedenou před nimi (skipy ani celočíselné konstanty ne). Pozor, pokud se použije skip tam, kde se očekává rozměr, TEX mlčí jako hrob a jako rozměr vezme základní velikost skipu.

```
\dimen0=1pt
\skip0=\dimen0 plus 0.3\dimen0
\the\skip0 % 1pt plus 0.3pt
\dimen0=0.6\skip0
\the\dimen0 % 0.6pt
```

Pokud si ukázku opravdu spustíte, zjistíte, že některé vypsané rozměry nejsou přesné. TEX je totiž počítá všechny celočíselně a zaokrouhluje. Nicméně 1 sp $\approx$ 5,36 nm, takže případné rozdíly jsou zanedbatelné (vlnová délka viditelného světla je řádově 100 sp). Je však vhodné o zaokrouhlování vědět.

Mnoho interních hodnot TEXu jsou čísla nebo rozměry; kompletní seznam můžete najít v TEXbooku na straně 272 a následujících.

Přiřazení do všech registrů i interních hodnot je lokální v rámci skupiny, není-li uvedeno primitivum `\global`:

```
\dimen0=5pt\dimen1=\dimen0
\the\dimen0 \the\dimen1 % 5pt 5pt
{\dimen0=10pt\global\dimen1=\dimen0
\the\dimen0 \the\dimen1} % 10pt 10pt
\the\dimen0 \the\dimen1 % 5pt 10pt
```

Boxy

TEX poskytuje 256 boxových registrů. Můžete si do nich uložit libovolný hbox nebo vbox a nad nimi pak provádět další operace. Přiřazení do boxu se provádí primitivem `\setbox` a jeho vysázení/použití primitivem `\box`.

```
\setbox0=\vbox{Ahoj Karle\par Jak se máš?}
Něco mezi.\par
\box0 % Box s pozdravem se vloží sem.
```

Po použití primitiva `\box` se registr vyprázdní. Pokud potřebujete, aby tam box zůstal, použijte na to primitivum `\copy`.

```
\def\fivetimes#1{{\setbox0\vbox{#1}%
\copy0\copy0\copy0\copy0\box0}}
```

TEX zná rozměry uloženého boxu. Dostanete se k nim (a dají se i změnit!) použitím primitiv `\ht` (výška), `\dp` (hloubka) a `\wd` (šířka).

```
\def\measure#1{{\setbox0\vbox{#1}%
(\the\ht0\ + \the\dp0) $\times$ \the\wd0}}
\def\nullbox#1{{\setbox0\vbox{#1}%
\ht0=0pt\dp0=0pt\wd0=0pt\box0}}
```

```
\quad R1\par
\setbox1\vbox{\hrule
  \quad\quad R2\par
  \quad\quad\quad R3\par\hrule}
\measure{\copy1}\par
\nullbox{\box1}
\quad\quad\quad\quad R4\par
\quad\quad\quad\quad\quad R5\par
```

R1

(27.55594pt + 0.0pt) × 348.77654pt

R2 R4
R3 R5

Všimněte si, že takto nelze natahovat nebo smršťovat samotný obsah boxu. To TEX neumí.[12] Umí však předstírat, že box má jinou velikost, než je ta skutečná. Toho jste si jistě všimli v příkladu. Možné využití jistě vymyslíte sami.

Rozbalování boxů

Čas od času je potřeba box rozbalit. Například si v boxu poskládáte kousek stránky a chcete, aby se TEX mohl rozhodnout, že uprostřed něj zlomí stránku. V takovém případě se vám můžou hodit primitiva `\unvbox` a `\unhbox`, kterými se vloží obsah odkazovaného boxu. Pokud potřebujete, aby se box akcí nevyprázdnil, použijte primitiva `\unvcopy` nebo `\unhcopy`.

Ještě vyšší liga je pak dělení vboxu primitivem `\vsplit`:

```
% Do vboxu vložíme několik odstavců textu
\setbox0\vbox{...}

% Uřízneme z něj a vložíme prvních 10cm
\vsplit0 to 10cm
\hrule % například čára na oddělení

% Vložíme zbytek
\box0
```

Uříznutí obsahu z boxu se koná na rozhraní boxů uvnitř. Takže obvykle se netrefíte přesně na rozměr. TEX zde používá naprosto stejný algoritmus jako na lámání stránky (ten nás v hrubých rysech čeká příště). Dostanete tedy box vysoký přesně zadaný rozměr se správně roztahanými mezerami.

[12] pdfTEX to ve skutečnosti umí, ale není to úplně přímočaré. Řekněte si kdyžtak na fóru o pohádku o transformačních maticích.

Úmyslně zde píšu box. Následující konstrukce je totiž povolena:

```
% Do vboxu vložíme několik odstavců textu
\setbox0\vbox{...}

% Uřízneme z něj prvních 10cm do boxu 1
\setbox1\vsplit0 to 10cm

% ... a nakopírujeme dvakrát hned pod sebe
\copy1\box1
```

Pojmenované registry

Ve složitějším dokumentu je vhodné pojmenovat si proměnné, neboť mezi očíslovanými boxy se dá jednoduše ztratit. K tomu slouží sada maker `\new...`. Povšimněte si rozdílu mezi prací s boxem a s číselnými veličinami.

```
\newcount\pocitadlo
\newdimen\velikost
\newskip\guma
\newbox\krabicka

\pocitadlo = 5\relax
\the\pocitadlo

\velikost = 5mm
\the\velikost

\guma = 5cm plus 1cm minus 1cm
\the\guma

\setbox\krabicka\vbox{obsah boxu}
\copy\krabicka
\unvcopy\krabicka
\vsplit\krabicka to \velikost
\box\krabicka
```

Vypíše toto:

5

14.22636pt

142.26378pt plus 28.45274pt minus 28.45274pt

obsah boxu
obsah boxu
obsah boxu

Plain TEX rezervuje některé registry pro svá makra a některé registry pro vaše makra (přes `\new...`). Pokud se vám nechce si rezervovat registr, který používáte jenom jako dočasné úložiště někde uvnitř složitých maker, jsou vám k dispozici registry s jednocifernými čísly. I na ně si však dejte pozor – pokud se

v takovém místě začne lámat stránka, nebo pokud je změníte globálně, dočkáte se velmi nepříjemných překvapení.

Pokud si chcete být jisti, používejte vždy a na všechno pojmenované registry.

Podmínka

TEX nabízí sadu podmínek `\if...`, které umožňují větvit kód a psát mocnější makra. Nejprve si ukážeme možnosti, které nám TEX nabízí, a potom detailně prozkoumáme, jak zpracovává zdrojový kód, který obsahuje podmínku.

- `\if` expanduje následující tokeny, dokud to jde. Pokud jsou ve výsledku první dva tokeny stejné, je podmínka splněna. (`\if aa` je pravda, `\if ab` je lež)
- `\ifx` vezme dva následující tokeny bez expanze. Pokud jsou identické (stejný znak, stejná kategorie, případně stejně definované makro nebo stejné primitivum), je podmínka splněna. Tahle podmínka se hodí zvlášť ve chvíli, kdy potřebujete detekovat například prázdný parametr:
  `\def\x#1{\def\p{#1}\ifx\p\empty...}`
- `\ifnum` porovnává dvě čísla. Povolené operace jsou `>`, `<` a `=`, přičemž se také parametr expanduje; `\ifnum\count1>5\xy` nemusí být kompletní podmínka, neboť za pětkou může pokračovat číslo, tedy i `\xy` za pětkou bude expandováno (a případně i další makra).
- `\ifodd` je pravda, pokud je uvedené číslo liché.
- `\ifdim` porovnává dva rozměry podobně jako podmínka `\ifnum`.
- `\ifvoid`, `\ifhbox` nebo `\ifvbox` detekuje, jestli je boxový registr prázdný, zaplněný hboxem nebo vboxem. Jako parametr čte jedno číslo.
- `\ifhmode`, `\ifvmode`, `\ifmmode` a `\ifinner` slouží ke zjištění, v jakém módu zrovna jsme (horizontálním, vertikálním, matematickém, případně vnitřním). První tři se vzájemně vylučují, čtvrtý je nezávislý (podmínka `\ifinner` je splněna, pokud jsme uvnitř explicitního vboxu, hboxu, nebo uvnitř jednodolarové matematiky).

Také si můžete definovat vlastní podmínku makrem `\newif`, kterou si pak můžete přepínat dle libosti.

```
\newif\ifbagr % všechny podmínky mají začínat if
\bagrtrue % nastavím, že je podmínka splněna
\bagrfalse % nastavím, že podmínka není splněna
```

Ještě jsme neukázali kompletní syntaxi podmínky. TEX, když uvidí `\if...`, vyhodnotí podmínku a rozhodne se, jestli je pravdivá, nebo nepravdivá. Pokud je pravdivá, bude pokračovat dále ve zpracovávání, dokud nenajde `\else`. Od této chvíle jen čte tokeny a zahazuje je, dokud nenajde `\fi`. TEX dodržuje uzávorkování podmínek, takže pokud je v zahazovaném seznamu tokenů `\if...`, zahodí i příslušné `\fi`.

Pokud je podmínka nepravdivá, zahodí se všechno do `\else` nebo `\fi`, co nastane dřív. Větev `\else` je totiž nepovinná.

Dejte si pozor na to, že tokeny `\if...`, `\else` a `\fi` ukončují například načítání čísla nebo rozměru. Není tedy možné napsat `\count\if... 5 \else 6\fi` apod. Podmínky ovšem nevytvářejí skupinu, jsou tedy běžné například takovéto konstrukce:

```
\ifnum\count0>10
  \def\next{...}
\else
  \let\next\relax
\fi\next
```

Úkol 1 [4b]: Vymyslete, jak automaticky číslovat nadpisy. Definujte sadu maker pro tři úrovně nadpisů. Makro nesmí brát za parametry nic jiného než text nadpisu. Nadpisy se automaticky číslují (od jedné), čísla nadpisů nižší úrovně začínají vždy od jedné po každém nadpisu vyšší úrovně.

Rozmyslete a vhodně ošetřete situaci, kdy bude text nadpisu příliš dlouhý, takže se nevejde na řádek. V řešení úkolu se zkuste obejít bez primitiva `\global`.

Vzhledem k tomu, že už jste poměrně zkušení, připravte makra včetně vhodného nastavení mezer a velikosti písma (vizte dále). Estetická kvalita výstupu bude zahrnuta do hodnocení.

Soubory: Vstup a výstup

Během sazby je možno pracovat i s jinými soubory než tím vstupním. Je vhodné si je nejprve pojmenovat, to se provádí makrem `\newread` (pro vstup) a `\newwrite` (pro výstup). Přesněji řečeno, tímhle si pojmenujete ukazatel na soubor. Jeden ukazatel nemůže zároveň ukazovat na vstup i výstup a jeden soubor není možno otevřít zároveň pro čtení i pro zápis.

Soubor otevřete primitivem `\openin` nebo `\openout`, pak je z něj možno číst nebo do něj zapisovat primitivem `\read` nebo `\write` a nakonec je vhodné soubor zavřít primitivem `\closein` nebo `\closeout`.

```
\newread\cti
\newwrite\pis
\openin\cti=in % in je jméno vstupního souboru
\openout\pis=out % out je jméno výst. souboru

\read\cti to \neco
\write\pis{\neco}

\closein\cti
\closeout\pis
```

Čtecí operace se odehrávají ihned, zapisovací však až ve chvíli, kdy se definitivně skládá stránka. Pokud však vložíte před takovou operaci primitivum `\immediate`, provede se hned. Důvod je jednoduchý – občas potřebujete při zapisování do souboru vědět, na které stránce nakonec skončí okolní text.

Primitivum `\write` svůj argument před zapsáním kompletně expanduje; potřebujete-li do výstupu propašovat přímo něco s backslashem (například pokud budete ten soubor za chvíli vkládat primitivem `\input`), předřaďte `\noexpand`.

Úkol 2 [4b]: Rozšiřte řešení **úkolu 1** o sazbu obsahu. Tedy přidejte příslušná makra a upravte stávající. Uvažujte sazbu obsahu na konci i na začátku, nezapomeňte rozmyslet sazbu příliš dlouhých nadpisů (které se nevejdou na řádek obsahu, takže bude potřeba je rozdělit) apod.

Obsah vypadá tak, že na každém řádku je na začátku číslo nadpisu, pak text nadpisu a na konci řádku číslo stránky.

Pokud budete sázet obsah na začátku, počítejte s víceprůchodovým zpracováním (na jeden průchod to nejde).

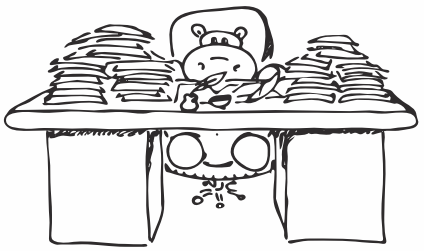
K vyřešení tohoto úkolu se bude hodit vědět, že číslo aktuální strany se nachází v číselném registru jménem `\pageno`.

Úkol 3 [6b]: Vytvořte makro `\multicolumn{X}`, kterému předáte jako `X` jedno celé číslo. Všechno mezi „začátkem“ `\multicolumn{X}` a „koncem“ `\endmulticolumn` bude vysázeno v `X` sloupcích stejné šířky vedle sebe (dohromady včetně mezer dají šířku sazby mimo toto prostředí).

Mezi sloupci nechť je mezera, jejíž celkovou šířku bude určovat registr `\newdimen\multicolumngap`, uprostřed ní nechť je svislá čára oddělující sloupce široká `\multicolumnline`.

Předpokládejte, že výsledná sazba se vejde na jednu stránku, tedy neřešte stránkový zlom. Vyhněte se načítání vnitřku prostředí do parametru makra. `\multicolumn` nechť prostředí inicializuje a `\endmulticolumn` nechť prostředí uzavře a vysází příslušný počet sloupců.

Za řešení, které bude zvládat jen $X = 2$, dostanete maximálně 3 body.



Různé druhy písem

Primitivní metoda, jak pracovat s písmem, je načtení a použití jednoho fontu. Konstrukcí `\font\xyz=csr10` jste řídící sekvenci `\xyz` ztotožnili s použitím běžného počeštěného desetibodového patkového fontu z rodiny Computer Modern.

Uvedená konstrukce může být doplněna ještě upřesněním typu `at 15pt`, což vezme původní vkládaný font a zvětší jej na uvedený rozměr.

Počeštěné fonty z rodiny Computer Modern se jmenují následovně:

`csr10`	Běžné patkové (Roman)
`cssl10`	*Skloněné (Slanted)*
`csti10`	*Kurzíva (Italic)*
`csb10`	**Polotučné (Bold)**
`csbx10`	**Tučné rozšířené (Bold extended)**
`csbxsl10`	***Tučné skloněné (Bold extended slanted)***
`csbxti10`	***Tučná kurzíva (Bold extended italic)***
`cscsc10`	Kapitálky (Small caps)
`cstt10`	`Strojopisné (Typewriter)`
`cssltt10`	*`Skloněné strojopisné (Slanted typewriter)`*
`csitt10`	*`Strojopisná kurzíva (Italic typewriter)`*
`cstcsc10`	`Strojopisné malé kapitálky (Typewriter small caps)`
`csvtt10`	Strojopisné proporcionální (Typewriter variable)
`csss10`	Bezpatkové (Sans-serif)
`csssdc10`	**Bezpatkové úzké (Sans-serif demi condensed)**
`csssi10`	*Bezpatková kurzíva (Sans-serif italic)*
`csssbx10`	**Bezpatkové tučné (Sans-serif bold extended)**
`csu10`	Narovnaná italika (Unslanted)

Číslo u jména fontu udává základní velikost v bodech (pt). U běžnějších fontů (`csr`, `csbx`) se obvykle dodávají i jiné základní velikosti, neboť například v menších velikostech je font širší – fontům se různě mění proporce, zvětšení fontu není prosté geometrické natažení.

Také je nutno poznamenat, že základní velikost fontu není velikost písmenek. Obvykle se jedná o součet maximální výšky a maximální hloubky písmene.

Testovací řetězec `csr3 at 10pt`

Testovací řetězec `csr6 at 10pt`

Testovací řetězec `csr8 at 10pt`

Testovací řetězec `csr10 at 10pt`

Testovací řetězec `csr12 at 10pt`

Testovací řetězec `csr15 at 10pt`

Testovací řetězec `csr20 at 10pt`

Testovací řetězec `csr40 at 10pt`

Rozsah dodávaných fontů záleží na distribuci a na balíkách, které máte nainstalované. V případě CM fontů navíc existuje tzv. Sauterova parametrizace, to je generátor všech základních velikostí v rozsahu cca 2 pt až 50 pt.

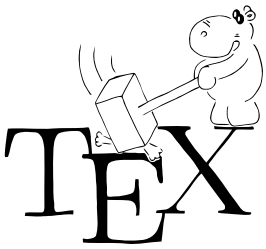
Běžná instalace csplainu obsahuje obvykle font csr velikostí 5, 6, 7, 8, 10, 12 a 17.

Po nastavení fontu je také potřeba správně nastavit některé další hodnoty, typicky `\baselineskip`.

Matematické fonty zůstávají nedotčeny; pokud potřebujete měnit i velikost písma v matematice, zkuste se podívat do TEXbooku (od strany 153) nebo do TBN (sekce 5.3), případně použít nějaký sofistikovaný systém, třeba OFS.

Olšákův systém fontů (OFS)

Pokud potřebujete seriózně pracovat s fonty a nechce se vám zabíhat do detailů, je vhodné použít nějaký balík, který práci s fonty abstrahuje. Osobně doporučím prostudovat dokumentaci k OFS.[13] Je to velmi chytře napsaný a mocný systém, který sám běžně používám v sazbě a který používáme i v KSP.



[13] `http://petr.olsak.net/ftp/olsak/ofs/papers/ofs-slt.pdf`

25-5-8 Boxy, z TEXu ven! — 15 bodů

Poslední díl seriálu věnujeme převážně výstupním rutinám a stránkovému zlomu. Vysvětlíme si, jak fungují penalty a špatnost sazby, a stručně si ukážeme okolí TEXu – formáty a nadstavby. Nakonec vložíme obrázek a vysázíme barevný dokument.

Stránkový zlom

TEX při sazbě stránky skládá boxy pod sebe do speciálního vertikálního boxu. Ve chvíli, kdy zjistí, že se už nevejde s výškou sazby do `\vsize`, najde správné místo, na kterém je nejlepší stránkový zlom, a tam uřízne box. Co se nevešlo, to si schová pro příští stranu.

Nejvýhodnější stránkový zlom se počítá tak, že se mezi každými dvěma položkami v boxu spočítá hodnota

$$c = \begin{cases} \infty, & \text{pokud } b = \infty \text{ nebo } q \geq 10\,000; \\ p, & \text{pokud } p \leq -10\,000; \\ b + p + q, & \text{pokud } b < 10\,000; \\ 100\,000, & \text{pokud } b = 10\,000, \end{cases}$$

přičemž b je „badness“, hodnota určující ošklivost roztažení nebo stažení stránky při zlomu na tomto místě;[14] p je penalta, hodnota určující nevhodnost zlomu na tomto místě (například mezi prvním a druhým řádkem odstavce); q je hodnota `\insertpenalties`, což je součet penalt pro speciální objekty jako poznámky pod čarou odpovídající zlomu.

Jediné, co můžete ovlivnit přímo, je penalta. Uvedete-li `\penalty 15`, vloží se na to místo penalta s hodnotou 15. Čím nižší penalta, tím spíš se na daném místě zlomí. Penalta $-10\,000$ a nižší vyvolá zlom vždy; penalta $10\,000$ a vyšší zlom zakáže. Pokud se někde vyskytnou dvě penalty za sebou, jejich hodnoty se sčítají.

Navíc je povoleno lámat jen na některých místech. TEX rozlišuje „zahoditelné“ a „nezahoditelné“ objekty. První z nich se za zlomem zahazují. Jedná se hlavně o penalty a výplňky.

Lámat se pak smí jen před výplňkem, před kterým je něco nezahoditelného, nebo na penaltě. V TEXbooku nebo TBN si to můžete přečíst precizně.

[14] Badness spočítáme podle vzorce $b = \min(10\,000, 100 \cdot (g/g_0)^3)$, kde g je součet roztažení nebo stažení mezer oproti normálu a g_0 je celkové maximální povolené roztažení nebo stažení.

Zde se můžou hodit vysvětlit některé zkratky, které jsme dříve definovali bez vysvětlení:

```
\def~{\penalty 10000 \ } % nedělitelná mezera
  % Mezera se považuje za výplněk a penalta
  % je zahoditelná ...

\def\break{\penalty-10000 } % zlom vždy
\def\nobreak{\penalty 10000 } % nelámej nikdy
\def\allowbreak{\penalty 0 } % povol zlom
% Na některých místech se nesmí lámat,
% například mezi dvěma čarami.
% Na penaltě se smí lámat vždy.

\def\filbreak{\par\vfil\penalty-200\vfilneg}
% \filbreak využívá skutečnosti, že na začátku
% každé stránky se zahodí všechny skipy.
% Přejde na novou stránku a zbytek vyplní
% prázdným místem, tedy pokud je záporná
% penalta dostatečná.
% Jinak se výplně vyruší:
% \def\vfil{\vskip 0pt plus 1fil}
% \def\vfilneg{\vskip 0pt plus -1fil}
\def\goodbreak{\par\penalty-500 }
\def\eject{\par\break}
\def\supereject{\par\penalty-20000 }
% Penalta -20000 se využívá pro požádání
% výstupní rutiny, aby vysázela všechny
% poznámky pod čarou a podobné elementy.
```

TEX si pak vybere takové místo, pro které je c nejmenší, a tam uřízne box. Co je před řezem, to vloží do vboxu číslo 255 a spustí výstupní rutinu.

Výstupní rutina

Na místo, kde došlo ke stránkovému zlomu, se vloží `{`, obsah seznamu tokenů `\output` a `}`. Cokoli, co vysázíte během výstupní rutiny, se přilepí před to, co zůstalo za stránkovým zlomem, a pokračuje se dál. Takto se tedy může výstupní rutina rozhodnout, že kus materiálu nevysází, a přesunout jej na další stranu. Na konci výstupní rutiny musí zůstat vbox 255 prázdný.

Dejte si pozor na to, že výstupní rutina se může aktivovat pokaždé, kdy vložíte nějaký materiál do hlavního vboxu, mimo jiné tam, kde se objeví `\par`, vložení boxu, čára, ... Pokud tedy v nějakém makru používáte stejné proměnné jako ve výstupní rutině (například `\count0` až `\count9`), pohlídejte si, aby se nespustila výstupní rutina zrovna v tu chvíli, kdy je máte předefinované.

Ve výstupní rutině se provedou všechny takové věci jako zvýšení čísla stránky, připojení hlaviček, patiček a poznámek pod čarou. Ve chvíli, kdy je poskládaná celá stránka, zavolá se `\shipout` a za toto primitivum se vloží box, který tvoří stránku. Tento box se ukotví svým levým horním rohem do bodu vzdáleného 1 in od levého i horního okraje. Tyto hodnoty se dají nastavit jako `\pdfhorigin` a `\pdfvorigin`.

Vzniklá stránka má rozměry `\pdfpagewidth` × `\pdfpageheight`, leda by nějaký z těch rozměrů byl nastaven na nulu. V takovém případě se příslušný rozměr vypočítá jako $x = x_0 + 2(f + r)$, kde x_0 je rozměr boxu předhozeného primitivu `\shipout`, f je `\hoffset` resp. `\voffset` a r je `\pdfhorigin` resp. `\pdfvorigin`.

Veškeré odložené operace (`\write` apod.) se provádějí ve chvíli, kdy příslušné místo projde `\shipout`em. Je tedy potřeba zajistit, aby všechna použitá makra byla definována v místě výstupní rutiny. Dokonce když zadáváte odložený `\write`, tak nemusíte mít použitá makra definována, stačí uvnitř výstupní rutiny.

Když se objeví `\end`, zavolá se výstupní rutina. Pokud po ní něco zbylo, vloží se do výstupu `\line{}\vfill\penalty-'10000000000` a znova se zpracovává token `\end`. Zkuste si předefinovat `\line`, vysázet extrémně dlouhý odstavec, a uvidíte, co se stane. Ve chvíli, kdy už není co zpracovat, TEX skončí.

Známé makro `\bye` je definováno takto:

```
\outer\def\bye{\par\vfill\supereject\end}
```

Výstupní rutina plainTEXu

```
\output{\plainoutput}
\def\plainoutput{%
  \shipout\vbox{%
    \makeheadline\box255\makefootline}%
  \advance\pageno by 1 }
\def\makeheadline{\vbox to 0pt{\vskip-22.5pt
  \line{\vbox to8.5pt{}\the\headline}\vss}
  \nointerlineskip}
\def\makefootline{%
  \baselineskip24pt\lineskiplimit0pt
  \line{\the\footline}}
```

Toto je zjednodušená verze výstupní rutiny plainTEXu. Jejím centrem je makro `\plainoutput`, které pošle stránku do výstupu a zvýší číslo stránky. Stránku poskládá tak, že nahoru vloží `\headline` (vhodně vysázenou), pak přidá samotnou stránku `\box255` a nakonec připojí `\footline`.

Ve skutečnosti se ve výstupní rutině plainu dělá trochu víc věcí, například se vkládají poznámky pod čarou.

Může se vám hodit umět nahradit kus výstupní rutiny plainu nějakým jiným kódem. V reálné výstupní rutině je například použito makro `\pagebody` místo `\box255`, které si můžete předefinovat.

Stejně tak můžete potřebovat například jinak pozicovanou hlavičku nebo patičku stránky. Stačí předefinovat příslušné makro.

Úkol 1 [3b]: Definujte makro `\stopoutput`, které vložením do zdrojáku způsobí, že od toho místa dál se na výstup nic nepošle. Definujte také makro `\startoutput` s opačným efektem, které na výstup data pošle. Vaše makro musí fungovat s libovolnou výstupní rutinou – o jejích vlastnostech nesmíte předpokládat prakticky nic.

Při definici neřešte patologické a okrajové případy, stačí, když bude makro fungovat při obvyklém použití (a dokumentujte, co se v tomto případě myslí obvyklým použitím). Například můžete vyžadovat, aby makro nebylo použito uvnitř explicitního hboxu nebo vboxu, nebo zakázat vnoření.

Může se vám hodit vědět, že TEX inkrementuje čítač `\deadcycles` pokaždé, když vstupuje do výstupní rutiny. Pokud jeho hodnota přeteče 25, skončí s chybou, neboť se domnívá, že máte ve výstupní rutině chybu a jste zacyklení. Čítač se nuluje při použití `\shipout`, nebo ho musíte snižovat ručně.

Úkol 2 [9b]: Upravte (vaši nebo vzorovou) implementaci `\multicolumn` z minulé série tak, že bude možno sázet text a další materiál do více sloupců přes více stran, podobně jako sázíme leták KSP.

Neuvažujte poznámky pod čarou, zkuste však implementovat makro tak, abyste umožnili vnoření. `\multicolumn` uvnitř jiného `\multicolumn` prostě vysází vícesloupcovou sazbu uvnitř vícesloupcové sazby.

Stejně tak se pokuste o to, aby se makro chovalo stejně jako v minulé sérii v případě, že jej použijete uvnitř jiného boxu.

Nezapomeňte na dokumentaci.

Formát

Samotný TEX je poměrně holá a osekaná kostra. Umí jen to nejnutnější, zbytek se definuje ve formátu, což je soubor v běžné syntaxi TEXu, který končí příkazem `\dump`. Tím se vygeneruje komprimovaný vnitřní stav TEXu na konci zpracovávání formátu. Během generování formátu platí omezení, že se nesmí vůbec nic vysázet.

TEX tedy umí pracovat ve dvou módech. První z nich jsme používali celou dobu v seriálu. Vezme uložený formát (v našem případě csplain), načte uložené

hodnoty do paměti a zpracovává a sází vstup. Ve druhém módu vezme vstup pro formát a vygeneruje jej. Tomu se také říká iniTEX.

Chcete-li TEXu nařídit, jaký formát použít, použijte na příkazové řádce parametr `-fmt` a za něj připojte název formátu. Chcete-li TEX spustit jako iniTEX, použijte parametr `-ini`.

Vzpomenete-li si na první díl a instalaci TeXworks, pak stejně jako pdfcsplain si můžete nastavit TEX s libovolným jiným formátem, když do pole Arguments napíšete správné argumenty.

Například známý LATEX, ConTEXt a další jsou jen různé formáty pro TEX, stejně jako plain.

Nadstavby

Původní TEX má mnohá omezení. Generuje výstup ve formátu DVI („device independent“), což bývalo užitečné v dobách, kdy ještě tiskárny neuměly žádný jednotný jazyk a příkazy v DVI se překládaly přímo do jazyka konkrétní tiskárny jejím ovladačem. Navíc se pracovalo na řádkových terminálech, kde nebylo možné si požadovaný výstup zobrazit.

Současné tiskárny umí prakticky všechny PostScript a před tiskem si prohlížíte PDF. Vytvářet DVI je tedy prakticky zbytečné. Proto vzniknul pdfTEX,[15] který generuje přímo výstup v PDF. Nad rámec toho, co umí TEX, implementuje další užitečné vlastnosti a funkce, například přímé vkládání obrázků, základní práci s barvami apod. Některá z těchto rozšíření jste už v seriálu potkali, konkrétně všechno, co začíná `\pdf...`

V dnešním multilingválním a internacionalizovaném světě je TEX se svým 8bitovým chápáním vstupu silně zastaralý. Světem hýbe UTF-8. Situaci se snaží zachránit encTEX,[16] rozšíření, díky kterému je možno mapovat sekvence 8bitových znaků (například znaky z UTF-8) na sekvence tokenů.

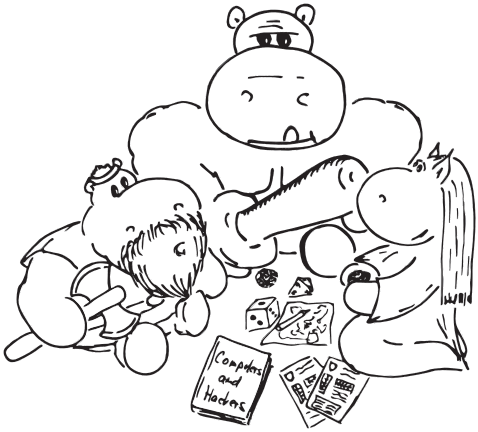
Všechny funkce pdfTEXu a encTEXu by vydaly na samostatnou sérii, tak jen podotkněme, že běžně dodávaný formát `plain-utf8-cs` se zapnutým encTEXem (argument `-enc` pro iniTEX) je csplain v UTF-8:

```
% vygenerování formátu
pdftex -enc -ini plain-utf8-cs
% použití formátu
pdftex -fmt plain-utf8-cs vstup.tex
```

Jako slibný projekt se pak jeví luaTEX,[17] což je implementace TEXu s možností vkládat do vstupního souboru kusy kódu v jazyce Lua. Ten již pracuje

[15] `http://www.tug.org/applications/pdftex/`
[16] `http://petr.olsak.net/enctex.html`
[17] `http://www.luatex.org/`

v Unicode a otevírá velmi zajímavé možnosti při psaní maker – některé konstrukce jsou v klasickém TEXu dosti nepraktické, až nemožné (složitější cyklus, opakovaná tokenizace, zavěšená interpunkce apod.). Některé z těchto nedostatků se snaží napravit rozšíření eTEX. Ještě jste se v těch TEXech neztratili?



Obrázky

Obrázky se vkládají primitivem `\pdfximage` (v pdfTEXu). Je možno nadiktovat si rozměry vkládaného obrázku i další parametry vytvářeného objektu ve výsledném PDF. Kompletní syntaxi a možnosti tohoto primitiva najdete v dokumentaci na webu pdfTEXu.

Primitivum `\pdfximage` pouze vloží obrázek jako objekt do PDF. Pokud jej chcete vložit do stránky, potřebujete primitivum `\pdfrefximage`, za které patří číslo objektu. To získáte primitivem `\pdflastximage` pro poslední obrázek vložený do PDF. (Pokud chcete vkládat jeden obrázek do stránky vícekrát, vložte jej do PDF jen jednou a pak se na něj vícekrát odkažte.)

```
\pdfximage@width@2cm@height@2cm@depth@1cm@{o.jpg}
\pdfrefximage\pdflastximage
```

Podporované formáty jsou JPEG pro fotografie, PNG pro bitmapovou grafiku, JBIG2 pro dvoubarevné bitmapy a PDF pro vektorovou grafiku.

Obrázek vložený ve stránce se chová jako vrule, resp. hrule. Pokud s ním potřebujete dělat nějaké speciality, zavřete jej do boxu.

Barvy

Každý objekt vykreslený TEXem má nějakou barvu, základní je černá. Její nastavení není v původním TEXu podporováno. V pdfTEXu je nutno vložit přímo kus kódu z formátu PDF.

Nejjednodušší způsob, jak změnit barvu, je přímé nastavení:

```
\def\red{\pdfliteral{1 0 0 rg}}
\def\black{\pdfliteral{0 0 0 rg}}
\def\green{\pdfliteral{0 0.5 0 rg}}
Černý text, \red červený text, \green
zelený text, \black černý text.
```

Černý text, červený text, zelený text, černý text.

Příkaz `rg` nastavuje barvu v prostoru RGB. Tři parametry se uvádí před ním, oddělené mezerou. Jsou to reálná čísla v rozsahu 0 až 1. První je červená, druhé je zelená a třetí modrá složka.

Dejte si pozor na to, že přímý zápis do PDF naprosto ignoruje nějaké uzavření do skupin, které vidí TEX, naopak je třeba uvažovat uzávorkování uvnitř PDF. Barva je nastavena obvykle do konce strany.

Chcete-li uložit na zásobník aktuální stav grafiky v PDF, můžete použít příkazy `q` a `Q`:

```
% Ulož stav grafiky
\def\beginpdfgroup{\pdfliteral{q}}
% Vrať stav grafiky
\def\endpdfgroup{\pdfliteral{Q}}
```

Analogicky k příkazu `rg` funguje příkaz `k` se čtyřmi parametry, který pracuje v prostoru CMYK, a příkaz `g` s jedním parametrem, jenž nastavuje barvu ve stupních šedé. Vyrábíte-li tedy PDF pro tisk, použijte CMYK, pokud se má výstup zobrazovat na obrazovce, použijte RGB.

Celé je to ještě trochu ztížené tím, že uvedené PDF příkazy platí jen pro čáry. Některé objekty se vykreslují jako výplň. Pokud se ve výstupu objevují objekty, které nerespektují nastavení barev, přidejte k nastavení barvy ještě jednou totéž, ale velkými písmeny. Všimněte si zlomkových čar:

```
\def\red{\pdfliteral{0 1 1 0 k}}
\def\green{\pdfliteral{1 0 1 0 k}}
\def\black{\pdfliteral{0 g}}
\def\Red{\pdfliteral{0 1 1 0 K}}
\def\Green{\pdfliteral{1 0 1 0 K}}
\def\Black{\pdfliteral{0 G}}
\def\fr{{a+b\over c}\quad}
$\displaystyle\fr\red\fr\green\fr\black
\fr\Red\fr\Green\fr\Black\fr$
```

$$\frac{a+b}{c} \quad \frac{a+b}{c} \quad \frac{a+b}{c} \quad \frac{a+b}{c} \quad \frac{a+b}{c} \quad \frac{a+b}{c} \quad \frac{a+b}{c}$$

Formát PDF je daleko mocnější, co se týče barev, ale to už výrazně přesahuje možnosti našeho seriálu. Máte-li zájem o přímé barvy Pantone, ICC profily a další, zeptejte se na fóru.

Úkol 3 [3b]: Implementujte makra pro pohodlnější práci s barvami. Váš balík musí umět definovat barvu v systémech RGB, CMYK a stupních šedé a pohodlně pak definovanou barvu nastavit. Použití může vypadat například takto:

```
\defrgbcolor\red{1 0 0}
\defcmykcolor\green{1 0 1 0}
\defgrayscalecolor\halfgray{0.5}
\defgrayscalecolor\black{0}

Černý, \red červený, \green zelený,
\halfgray šedý, \black černý text.
```

Při řešení úkolů se vám možná budou hodit nějaké triky, které se objeví v řešení čtvrté série. Nezapomeňte tam nahlédnout.A to je vše, přátelé. Doufám, že TEXu zůstanete věrni i nadále.

Programátorské kuchařky

Kuchařka první série – složitost

Časová a paměťová složitost

V této kuchařce se můžete dočíst o základech časové a paměťové složitosti. Po přečtení byste měli být schopni sami rozebrat složitost jednoduchých algoritmů. To se hodí třeba při návrhu algoritmů a řešení algoritmických úloh, které můžete potkat například v KSP.

Nejdříve si ujasníme, co to ta složitost vlastně je, a ukážeme si pár příkladů. Pak si řekneme, s jakou přesností budeme složitost chtít určovat, a zavedeme si asymptotickou složitost. Na závěr si ukážeme běžné třídy složitosti.

Základní přehled

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítka, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat, že aritmetické operace, přiřazování, porovnávání apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bytů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly

v posloupnosti, a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:

```
posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
   Jestliže posl[i] > max:
      max = posl[i]
Vypiš max
```

Není těžké nahlédnout, že algoritmus provede maximálně $N - 1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N, protože porovnání dvou čísel nám zabere „jednotkový čas“, a paměťová složitost bude také na N záviset lineárně, protože si každé číslo z posloupnosti budeme uchovávat v paměti.

Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K:

```
Pro i = 1 až K:
   Pro j = 1 až K:
      Vypiš i*j a mezeru
   Přejdi na nový řádek
```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2. Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí. To kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách.

Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nej-

jednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme N^k.

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N. Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N. To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N, na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S, podstatných jmen je A a přidavných jmen B.

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka).[18] V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M, kde N je počet vrcholů grafu a M je počet jeho hran. I pro více proměnných vybíráme nejhorší případ.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N.

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B o složitosti N^2. Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A, ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A.

[18] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. **asymptotickou časovou složitost.**

Představme si, že máme algoritmus se složitostí $n^2/4+6n+12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $n^2 + n + 1$.
- Pro velká n je $n + 1$ oproti n^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost n^2. Obecně škrtáme všechny členy, které jsou pro dost velké n menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor $\mathcal{O}$ (velké O), díky kterému budeme umět popsat, co přesně naše „škrtání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \to \mathbb{R}^+$ a $g : \mathbb{N} \to \mathbb{R}^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do kladných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(f)$.

A zde je použití: $n^2/4+6n+12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\ldots)$. Někdy také říkáme, že se konstanty a méně významné členy v $\mathcal{O}$ ztrácí.

Ještě poznamenejme, že operátor $\mathcal{O}(\ldots)$ znamená asymptotický horní odhad funkce. Takže pokud funkce patří do $\mathcal{O}(N)$, tak pak patří i do $\mathcal{O}(N^2)$, $\mathcal{O}(N^3)$, ...

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu BubbleSort (bublinkovém třídění), o kterém se můžete dočíst v kuchařce o třídících algoritmech.[19] Funguje tak, že se dívá na všechny dvojice sousedních prvků, a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

```
BubbleSort(pole, N):
   Opakuj:
      setříděno = 1
      Pro i = 1 až N-1:
         Jestliže pole[i] > pole[i+1]:
            p = pole[i]
            pole[i] = pole[i+1]
            pole[i+1] = p
            setříděno = 0
   Skonči, až bude setříděno = 1
```

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám totiž největší hodnota „probublá“ na konec a ostatní se posunou o jednu pozici doleva. Rozmyslete si, proč. Průchodů je proto nejvýše $N-1$ a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy nastat, pokud necháme setřídit klesající posloupnost. Tam provedeme přesně $N-1$ průchodů.

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídícího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídící algoritmus QuickSort (opět viz třídicí kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

[19] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

$\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)

$\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do $\mathcal{O}$-čka“.

$\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)

$\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)

$\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)

$\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)

$\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsat, dostaneme $\mathcal{O}(N \cdot 2^N)$)

$\mathcal{O}(N!)$ – *faktoriálová*, $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ (nalezení všech permutací N prvků, tedy například všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální.* Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k. Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří např. i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$ (každý algoritmus, který seběhne v čase $\mathcal{O}(\log N)$, seběhne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3,3 ns	4,3 ns	4,9 ns	6,6 ns	10,0 ns	19,9 ns
n	10 ns	20 ns	30 ns	100 ns	1 μs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 μs	20 ms
n^2	100 ns	400 ns	900 ns	100 μs	1 ms	1 000 s
n^3	1 μs	8 μs	27 μs	1 ms	1 s	10^9 s
2^n	1 μs	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Karel Tesař a Martin Mareš

Kuchařka druhé série – minimální kostra

Představme si následující problém: Chceme určit silnice, které se budou v zimě udržovat sjízdné, a to tak, abychom celkově udržovali co nejméně kilometrů silnic, a přesto žádné město od ostatních neodřízli.

Města a silnice si můžeme představit jako graf, o kterém nyní budeme předpokládat, že je souvislý. Kdyby nebyl, náš problém nijak vyřešit nelze. Výsledný podgraf/seznam silnic, který řeší náš problém se sněhem, nazývají matematici *minimální kostra grafu*.

Pokud vůbec netušíte, co je to graf, přečtěte si úvodní grafovou kuchařku na našem webu.[20]

Co se v souvislém grafu přesně myslí pod pojmem *kostra*? Nazveme jí libovolný podgraf, který obsahuje všechny vrcholy a zároveň je stromem. *Strom* jsme si definovali v kapitole o grafech; jsou to přesně ty grafy, které jsou souvislé (z každého vrcholu „dojedeme“ do každého jiného) a bez kružnice (takže nemáme v silniční síti žádné přebytečné cesty).

Pokud každou hranu grafu ohodnotíme nějakou *vahou,* což v našem případě bude vždy kladné číslo, dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* což je taková, pro kterou je součet vah jejích hran nejmenší možný.

Graf může mít více minimálních koster – například jestliže jsou všechny váhy hran jedničky, všechny kostry mají stejnou váhu $n - 1$ (kde n je počet vrcholů grafu), a tedy jsou všechny minimální.

Pro vyřešení problému hledání minimální kostry se nám bude hodit datová struktura *Disjoint-Find-Union* (DFU). Ta umí pro dané disjunktní množiny (disjunktní znamená, že každé 2 množiny mají prázdný průnik neboli žádné společné prvky) rychle rozhodnout, jestli dva prvky patří do stejné množiny, a provádět operaci sjednocení dvou množin.

Algoritmus

Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, což mlčky předpokládáme). Než

[20] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

si ukážeme, že nalezená kostra je opravdu minimální, podívejme se na časovou složitost našeho algoritmu: Pokud vstupní graf má N vrcholů a M hran, tak úvodní setřídění hran vyžaduje čas $\mathcal{O}(M \log M)$ (použijeme některý z rychlých třídicích algoritmů popsaných v jednom z minulých dílů kuchařky) a poté se pokusíme přidat každou z M hran.

V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat nejvýše $\mathcal{O}(M \log N)$. Celková časová složitost našeho algoritmu je tedy $\mathcal{O}(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $\mathcal{O}(M)$.

Důkaz správnosti

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou navzájem různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídicím algoritmem zůstalo zachováno. Tím se kostra nalezená hladovým algoritmem nezmění, a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme si nyní T_{alg} kostru nalezenou hladovým algoritmem a T_{min} nějakou minimální kostru. Co by se stalo, kdyby byly různé? Víme, že všechny kostry mají stejný počet hran, takže musí existovat alespoň jedna hrana e, která je v T_{alg}, ale není v T_{min}. Ze všech takových hran si vyberme tu, která má nejmenší váhu, tedy kterou algoritmus přidal jako první. Když se podíváme na stav algoritmu těsně před přidáním e, vidíme, že sestrojil nějakou částečnou kostru F, která je ještě součástí jak T_{min}, tak T_{alg}.

Přidejme nyní hranu e ke kostře T_{min}. Tím vznikl podgraf vstupního grafu, který zjevně obsahuje nějakou kružnici C – už před přidáním hrany e totiž T_{min} byla souvislá. Protože kostra T_{alg} neobsahuje žádnou kružnici, na kružnici C musí být alespoň jedna hrana e', která není v T_{alg}.

Všimněme si, že hranu e' nemohl algoritmus zpracovat před hranou e: hrana e' neleží v T_{min} na žádném cyklu, takže tím spíš netvoří cyklus v F, a kdyby ji algoritmus zpracoval, musel by ji přidat do F, což, jak víme, neučinil. Z toho plyne, že váha hrany e' je větší než váha hrany e. Když nyní z kostry T_{min} odebereme hranu e' a přidáme místo ní hranu e, musíme opět dostat souvislý podgraf (e a e' přeci ležely na společné kružnici), tudíž kostru vstupního grafu. Jenže tato kostra má celkově menší váhu než minimální kostra T_{min}, což není možné. Tím jsme došli ke sporu, a proto T_{min} a T_{alg} nemohou být různé.

Cvičení

- V důkazu jsme předpokládali, že váhy hran jsou různé.Není potřeba i v samotném algoritmu přičítat velmi malá čísla k hranám se stejnou vahou?

Disjoint-Find-Union

Datová struktura *DFU* slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do stejné podmnožiny rozkladu, pokud mezi nimi v již vytvořené části kostry existuje cesta. Jinými slovy podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find:** Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v případě našeho algoritmu odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti.
- **union:** Sloučení dvou podmnožin do jedné. Tuto operaci v našem algoritmu na hledání kostry provedeme vždy, když do vytvářené kostry přidáme hranu (tehdy spojíme dvě různé komponenty souvislosti dohromady).

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele (trochu nezvykle) od listů ke kořeni. Operaci *find* lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i ve stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci *union* provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme se ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Rodiče jednotlivých vrcholů stromu si pak pamatujeme v poli *parent*, kde 0 znamená, že prvek rodiče nemá, tj. že je kořenem svého stromu. Funkce *root*(v) vrátí kořen stromu, který obsahuje prvek v.

```
var parent: array[1..N] of integer;

procedure init;
var i: integer;
begin
  for i:=1 to N do parent[i]:=0;
end;
```

```
function root(v: integer):integer;
begin
  if parent[v]=0 then root:=v
  else root:=root(parent[v]);
end;

function find(v, w: integer):boolean;
begin
  find:=(root(v)=root(w));
end;

procedure union(v, w: integer);
begin
  v:=root(v); w:=root(w);
  if v<>w then parent[v]:=w;
end;
```

S právě předvedenou implementací operací *find* a *union* by se ale mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“, a pokud budou obsahovat N prvků, na nalezení kořene bude potřeba čas $\mathcal{O}(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank:** Každý prvek má přiřazen *rank*. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace *union* připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression:** Ve funkci *root*(v) přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní implementace funkcí *root* a *union*:

```
var parent: array[1..N] of integer;
    rank: array[1..N] of integer;

procedure init;
var i: integer;
begin
  for i:=1 to N do
    begin
      parent[i]:=0;
      rank[i]:=0;
    end;
end;
```

```
{změna path compression}
function root(v: integer): integer;
begin
  if parent[v]=0 then root:=v
  else begin
    parent[v]:=root(parent[v]);
    root:=parent[v];
  end;
end;

{stejna jako minule}
function find(v, w: integer):boolean;
begin
  find:=(root(v)=root(w));
end;

{změna kvůli union by rank}
procedure union(v, w: integer);
begin
  v:=root(v);
  w:=root(w);
  if v=w then exit;
  if rank[v]=rank[w] then
    begin
      parent[v]:=w;
      rank[w]:=rank[w]+1;
    end
  else if rank[v]<rank[w] then
    parent[v]:=w
  else
    parent[w]:=v;
end;
```

Zaměřme se nyní blíže na metodu *union by rank*. Nejprve učiníme následující pozorování: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků.

Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$. V okamžiku, kdy se rank prvku v mění z $r-1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r-1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků, a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali.

Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se

nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen *union by rank,* je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Potom ale procedura *root* spotřebuje čas nejvýše $\mathcal{O}(\log N)$, a tedy operace *find* a *union* stihneme v čase $\mathcal{O}(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co je *amortizovaná* časová složitost. Řekneme, že nějaká operace pracuje v amortizovaném čase $\mathcal{O}(t)$, pakliže provedení libovolných k takových operací trvá nejvýše $\mathcal{O}(kt)$. Přitom provedení kterékoliv konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak v součtu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Nejdříve si předveďme tento pojem na jednoduchém příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, neboť záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla.

Pokud se nám ale povede ukázat, že N přičtení jedničky k číslu, které je na počátku nula, zabere čas $\mathcal{O}(N)$, pak můžeme říci, že každé takové přičtení trvalo amortizovaně $\mathcal{O}(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $\mathcal{O}(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek, a pokud jich na N operací použijeme jen $\mathcal{O}(N)$, bude tvrzení dokázáno.

Každé jedničce, kterou chceme přičíst, dáme dva penízky. V průběhu celého přičítání bude platit, že každá jednička ve dvojkovém zápisu čísla má jeden penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme).

Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na nejnižší bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu, atd.

Takto splníme podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek. Tedy N přičítání nás stojí $2N$ penízků. Protože počet penízků utracených během jedné operace je úměrný spotřebovanému času, vidíme, že všech N přičtení proběhne v čase $\mathcal{O}(N)$. Není těžké si uvědomit, že přičtení některých jedniček může trvat až $\mathcal{O}(\log N)$, ale amortizovaná časová složitost přičtení jedné jedničky je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze *path compression* a nikoliv *union by rank*, dalo by se dokázat, že každá z operací *find* a *union* vyžaduje amortizovaně čas $\mathcal{O}(\log N)$, kde N je počet prvků. Toto tvrzení nebudeme dokazovat, protože tím bychom si nijak oproti samotnému *union by rank* nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou metod současně dosáhneme mnohem lepšího amortizovaného času $\mathcal{O}(\alpha(N))$ na jednu operaci *find* nebo *union*, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

Dokázat výše zmíněný odhad časové složitosti funkcí $\alpha(N)$ je docela těžké, my si zde předvedeme poněkud horší, ale technicky výrazně jednodušší časový odhad $\mathcal{O}((N+L)\log^* N)$, kde L je počet provedených operací *find* nebo *union* a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2 \uparrow (k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně zlogaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU při současném použití obou metod *union by rank* a *path compression*. Prvky si rozdělíme do skupin podle jejich ranku: k-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (k-1)) + 1$ a $2 \uparrow k$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = \mathcal{O}(\log^* N)$ skupin. Odhadněme shora počet prvků v k-té skupině:

$$\frac{N}{2^{(2\uparrow(k-1))+1}} + \cdots + \frac{N}{2^{2\uparrow k}} = \frac{N}{2^{2\uparrow(k-1)}} \cdot \left(\sum_{i=1}^{2\uparrow k - 2\uparrow(k-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(k-1)}} \cdot 1 = \frac{N}{2 \uparrow k}.$$

Teď můžeme provést časovou analýzu funkce *root*(v). Čas, který spotřebuje funkce *root*(v), je přímo úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto

volání funkce *root*(v), a ty, které zahrneme do faktoru $\mathcal{O}(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce *root*(v) započítáme ty hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $\mathcal{O}(\log^* N)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v k-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow k$ přepojeních je rodič prvku v v $(k+1)$-ní nebo vyšší skupině. Pokud v je prvek v k-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce *root*(v) nejvýše $(2 \uparrow k)$-krát. Protože k-tá skupina obsahuje nejvýše $N/(2 \uparrow k)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $\mathcal{O}(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce *root*(v), nejvýše $\mathcal{O}(N \log^* N)$. Protože funkce *root*(v) je volána $2L$-krát, plyne časový odhad $\mathcal{O}((N+L) \log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce s jedním parametrem $A(k)$ je pak rovna hodnotě $A_k(2)$, takže $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd... Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je tedy nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

Dan Kráľ, Martin Mareš a Milan Straka

Kuchařka třetí série – teorie čísel

Dnes si budeme povídat o různých užitečných vlastnostech celých čísel, především o dělitelnosti a kongruencích. Mohlo by se zdát, že to nemá s informatikou nic společného, ale překvapivě v informatice zakopáváme o teorii čísel takřka na každém kroku. Někdy se jedná o hledání velkých prvočísel, jindy o rychlé násobení čísel s miliony cifer nebo všudypřítomnou asymetrickou šifru RSA.

Začneme vyjasněním základních pojmů, postupně se prokoušeme kongruencemi k hledání největšího společného dělitele a Bézoutových koeficientů, chvilku se zamyslíme nad prvočísly a nakonec si také ukážeme, jak to všechno souvisí s čínskou armádou.

Definice na úvod

Množinu celých čísel si označíme $\mathbb{Z}$ a každé její podmnožině $\{0, 1, \ldots, n-1\}$ budeme říkat $\mathbb{Z}_n$.

Často nás bude zajímat *dělitelnost:* $a \setminus b$ (nebo $a \mid b$) budeme značit, že číslo a je dělitelem čísla b (nebude-li hrozit mýlka, čteme prostě „a dělí b“).

Pro *největšího společného dělitele* dvou čísel zavedeme symbol $\mathrm{nsd}(a, b)$. Pokud $\mathrm{nsd}(a, b) = 1$, říkáme, že čísla a a b jsou *nesoudělná,* zkráceně $a \perp b$. Když budou naopak a a b soudělná, napíšeme $a \parallel b$. Podobně nejmenší společný násobek dvou čísel označíme $\mathrm{nsn}(a, b)$ a všimneme si, že je roven $a \cdot b \,/\, \mathrm{nsd}(a, b)$.

Není-li jedno číslo dělitelné druhým, znamená to, že při celočíselném dělení vznikne zbytek: například pokud vydělíme $23/8$, dostaneme zbytek 7, protože $23 = 8 \cdot 2 + 7$. Obecně *zbytkem po dělení* a/b nazveme hodnotu z v rovnici $a = b \cdot x + z$, kde x je celé číslo a z je nezáporné celé číslo menší než b. Obvyklé programovací jazyky mívají takovouto operaci zabudovanou a říkají jí *modulo.* Programátoři počítají zbytky po dělení často nějakou konstrukcí podobnou `z = a % b`, zatímco matematici spíše píší $z = a \bmod b$.

Dodejme, že pro záporná čísla už není definice zbytku po dělení tak jednoznačná: v některých programovacích jazycích je `(-7)%3 = -1`, jiné se shodnou s naší definicí na tom, že vyjde `2`. Přitom oba výsledky vycházejí z jedné rovnice $a = b \cdot x + z$. Aby měla jednoznačné řešení, požadovali jsme $0 \leq z < b$. Lze na to ovšem jít i jinak: řekneme, že x má být celočíselný podíl a/b. Ten jde ale definovat dvěma způsoby: buď se zaokrouhlením dolů (což se shodne s naší definicí), nebo se zaokrouhlením k nule (to dělá většina procesorů), což dá pro záporné a záporný zbytek. Zkuste zjistit (a vysvětlit), jak je to ve vašem oblíbeném jazyce a co se stane, když je záporné i číslo, kterým modulíme.

Kongruence

Když čísla p a q dávají stejný zbytek po dělení číslem m, píšeme

$$p \equiv q \pmod{m}$$

a čteme „p je *kongruentní* s q modulo m“. To platí právě tehdy, je-li rozdíl $p-q$ dělitelný m.

Zápis kongruence tak trochu připomíná rovnici. To není náhoda – kongruence totiž můžeme upravovat podobně jako rovnice.

Součet dvou kongruencí: Pokud $a \equiv A$ a $b \equiv B$, pak také platí $a+b \equiv A+B$ (to vše modulo totéž m). Že je to pravda, nahlédneme snadno. Napišme si a jako $n_a \cdot m + z_a$ a čísla A, b, B obdobně. Pak dostaneme:

$$\begin{aligned} a+b &= (n_a \cdot m + z_a) + (n_b \cdot m + z_b) = \\ &= (n_a + n_b) \cdot m + (z_a + z_b), \\ A+B &= (n_A \cdot m + z_A) + (n_B \cdot m + z_B) = \\ &= (n_A + n_B) \cdot m + (z_A + z_B). \end{aligned}$$

Protože $a \equiv A$ a $b \equiv B$, musí být $z_a = z_A$ a $z_b = z_B$. Můžeme si tedy všimnout, že rozdíl mezi $a+b$ a $A+B$ je $((n_a+n_b)-(n_A+n_B)) \cdot m$. To je násobek m, takže $a+b \equiv A+B$.

Rozdíl dvou kongruencí: Nahlédneme obdobně.

Přičtení téhož čísla k oběma stranám: Pokud $a \equiv A$, pak platí $a+k \equiv A+k$ pro libovolné k. Stačí totiž přičíst evidentně platnou kongruenci $k \equiv k$.

Přičtení násobku m k jedné straně: Z $a \equiv A$ plyne $a+k \equiv A$ pro libovolné k, které je násobkem modulu m. Přičítáme totiž kongruenci $k \equiv 0$.

Vynásobení dvou kongruencí: Z $a \equiv A$ a $b \equiv B$ plyne $ab \equiv AB$. Stejně jako u součtů a rozdílů, i zde stačí čísla rozepsat na součty násobků m a zbytků po dělení m:

$$\begin{aligned} a \cdot b &= (n_a \cdot m + z_a) \cdot (n_b \cdot m + z_b) = \\ &= (n_a \cdot n_b \cdot m + n_a \cdot z_b + n_b \cdot z_a) \cdot m + (z_a \cdot z_b) \equiv \\ &\equiv z_a \cdot z_b, \\ A \cdot B &= (n_A \cdot m + z_A) \cdot (n_B \cdot m + z_B) = \\ &= (n_A \cdot n_B \cdot m + n_A \cdot z_B + n_B \cdot z_A) \cdot m + (z_A \cdot z_B) \equiv \\ &\equiv z_A \cdot z_B. \end{aligned}$$

Přitom opět víme, že $z_a = z_A$ a $z_b = z_B$.

Vynásobení obou stran kongruence týmtéž číslem: Pokud $a \equiv A$, platí také $ax \equiv Ax$ pro libovolné x. To plyne z násobení kongruencí $x \equiv x$.

Ekvivalentnost úprav: Běžné úpravy rovnic jsou takzvaně ekvivalentní – to znamená, že fungují oběma směry, takže řešení rovnic ani neubírají, ani nepřidávají. Jak je to s kongruencemi? Sčítání kongruencí ekvivalentní musí být, protože opačný směr odpovídá odečtení kongruencí, což víme, že je také korektní úprava.

U násobení kongruencí to už tak jasné není. Zkusme zjistit, jestli je pravda, že z kongruence $ax \equiv Ax$ plyne $a \equiv A$. Pro $x = 0$ to jistě neplatí, ale co když zvolíme jiné x?

Vyzkoušíme to třeba na následujícím příkladě:

$$a \equiv 5 \pmod{14}$$

Takováto kongruence má jednoduché řešení: a je každé celé číslo, které dostanu sečtením 5 a nějakého násobku 14. To se dá zapsat třeba takhle:

$$a \in \{5 + k \cdot 14 \mid k \in \mathbb{Z}\},$$

tudíž a může tedy například 5, 19 nebo 33.

Vyzkoušíme nyní obě strany vynásobit… třeba trojkou:

$$3 \cdot a \equiv 15 \equiv 1 \pmod{14}.$$

Tato kongruence platí pro všechna a, která po vynásobení 3 dávají modulo 14 zbytek 1. Když máme nějaké a z první kongruence, je ve tvaru $5 + k \cdot 14$. Když ho vynásobíme 3, dostaneme $15 + 3 \cdot k \cdot 14$. To je určitě kongruentní s 1 modulo 14, takže žádné řešení jsme neztratili a po chvíli uvažování zjistíme, že jsme ani žádné nepřidali.

Další pokus: původní kongruenci vynásobíme místo trojky dvojkou. Dostaneme:

$$2 \cdot a \equiv 10 \pmod{14}.$$

Řešení původní kongruence pořád sedí, ale nová kongruence platí například i pro $a = 12$. Posuďte sami:

$$2 \cdot 12 = 24 = 10 + 14 \equiv 10 \pmod{14}.$$

Ouha, najednou vynásobení obou stran konstantou není ekvivalentní úprava!

Proč nám násobení trojkou fungovalo, ale násobení dvojkou si vymýšlí kořeny navíc? Postupně se ukáže, že násobení k je ekvivalentní úprava právě tehdy, když $k \perp 14$ (či obecněji $k \perp m$, počítáme-li modulo m).

Než k tomu dojdeme, nejdřív na chvíli odbočíme k největším společným dělitelům.

Euklidův algoritmus

Největšího společného dělitele dvou čísel můžeme vypočítat pomocí prvočíselného rozkladu, ale to je pro velká čísla velmi pomalé. Daleko lepší je použít prastarý Euklidův algoritmus. (Jmenuje se podle starověkého matematika Euklida, v jehož díle Základy se nachází první dochovaná verze. Jedná se zřejmě o nejstarší netriviální algoritmus, jaký se s drobnými úpravami používá dodnes. Dokonce je pravděpodobné, že Euklidés pouze sepsal dávno známý trik.)

Pojďme si odvodit, jak Euklidův algoritmus funguje. Nahlédneme, že pro libovolná čísla a, b $(a > b)$ platí:

$$\mathrm{nsd}(a, b) = \mathrm{nsd}(a - b, b).$$

Proč je to pravda? Dokážeme, že dvojice (a, b) a $(a-b, b)$ sdílejí dokonce všechny společné dělitele, takže i toho největšího:

- Nechť d je společným dělitelem a a b. Platí tedy $a = a' \cdot d$, $b = b' \cdot d$ pro nějaká celá čísla a' a b'. Pak ovšem můžeme zapsat $a - b$ jako $(a' - b') \cdot d$, což je zase dělitelné číslem d.
- Nechť naopak d je společným dělitelem $a-b$ a b. Opět zapíšeme $a-b = c' \cdot d$, $b = b' \cdot d$ a získáme $a = (a - b) + b = (c' + b') \cdot d$.

Euklidův algoritmus dostane na vstupu nějaká dvě čísla a a b a opakovaně odčítá menší z nich od většího. Jak už víme, tato operace zachovává největšího společného dělitele. Pokaždé se přitom součet $a + b$ zmenší, takže po konečně mnoha krocích musíme jedno z čísel vynulovat. Pak víme, že největším společným dělitelem je druhé z nich (platí přeci $\mathrm{nsd}(0, x) = x$).

Pojďme si takhle nějakého největšího společného dělitele spočítat. Abychom netroškařili, zkusme rovnou čísla 1518 a 945.

a	b
1518	945
573	945
573	372
201	372
201	171
30	171

Zastavme se na chvíli. Teď bychom mohli pracně odečítat 30 od b, dokud bychom nenašli v b něco menšího než 30. Buďme trochu líní: když to budeme dělat dost dlouho, zbude nám v b zkrátka zbytek po dělení b číslem 30. Můžeme tedy místo odečítání modulit. Pokračujeme:

a	b
30	171
30	$21 = 171 \bmod 30$
$30 \bmod 21 = 9$	21
9	$3 = 21 \bmod 9$
$9 \bmod 3 = 0$	3

V a nám zbyla 0, takže $\mathrm{nsd}(1518, 945) = 3$.

Pojďme si tento postup převést do návodu pro počítač. Při implementaci Euklidova algoritmu se hodí držet si v jedné proměnné pořád to větší z čísel

a a b. Navíc můžeme využít toho, že každým krokem algoritmu se z většího čísla stane menší, takže je stačí prohodit a není potřeba znovu porovnávat. (Dokonce i porovnání před cyklem bychom si mohli ušetřit, kdyby nám nevadilo, že první průchod cyklem může projít „naprázdno“.)

```
def Euclid(a, b):
    # Prohodíme, je-li třeba
    if b > a:
        a, b = b, a

    # Zde je vždy a >= b
    while b > 0:
        # nsd(a % b, b) = nsd(a, b).
        a = a % b
        a, b = b, a

    # nsd(0, a) = a.
    return a
```

Jak rychle náš algoritmus běží? Podívejme se, co se stane, když pustíme dva kroky algoritmu na čísla a_1 a b_1 ($a_1 \geq b_1$):

$$\begin{aligned} a_2 &= a_1 \bmod b_1 \\ b_2 &= b_1 && \text{(nyní } a_2 < b_2\text{)} \\ a_3 &= a_2 = a_1 \bmod b_1 \\ b_3 &= b_2 \bmod a_2 = b_1 \bmod (a_1 \bmod b_1) && \text{(nyní } a_3 > b_3\text{)} \end{aligned}$$

Dokážeme, že $a_3 < a_1/2$. Rozebereme přitom dva případy podle toho, jestli bylo b_1 menší, nebo větší než $a_1/2$:

- Pokud $b_1 \leq a_1/2$, pak určitě platí $a_3 < b_1$, a tedy i $a_3 < a_1/2$. (Zde využíváme toho, že zbytek po dělení čehokoliv číslem b_1 musí být menší než b_1.)
- V opačném případě leží b_1 mezi $a_1/2$ a a_1, takže $a_3 = a_1 \bmod b_1 = a_1 - b_1 < a_1/2$. (Poslední rovnost platí, protože $\lfloor a_1/b_1 \rfloor = 1$.)

Dokázali jsme tedy, že po dvou krocích algoritmu se větší z obou proměnných zmenší přinejmenším na polovinu a opět bude větší. Po $\mathcal{O}(\log n)$ krocích tedy musí větší proměnná klesnout pod 1, čímž se algoritmus zastaví. Euklidův algoritmus proto provede $\mathcal{O}(\log n)$ elementárních operací.

Jak dlouho ale trvá jedna elementární operace? Pokud počítáme s malými čísly, která se našemu počítači vejdou do celočíselné proměnné, zvládneme ji v konstantním čase. Jsou-li ovšem čísla větší, musíme ještě zohlednit složitost aritmetických operací: porovnání čísel a operace modulo. Když použijeme modulení pomocí školního dělení, které je kvadratické v počtu cifer, strávíme v každém z $\mathcal{O}(\log n)$ kroků Euklidova algoritmu čas $\mathcal{O}(\log^2 n)$. Celková složitost algoritmu tedy vzroste na $\mathcal{O}(\log^3 n)$.

Rozšířený Euklidův algoritmus

Právě jsme našli největšího společného dělitele d nějakých dvou obrovských čísel a a b. Jak ale přesvědčíme svého pochybovačného kolegu, že je náš výsledek správně? Snadno ověříme, že d dělí obě čísla. Ale jak ukážeme, že žádné větší číslo už a ani b nedělí? Překvapivě to jde jednoduše dosvědčit: pokud pro nějaká u a v platí

$$a \cdot u + b \cdot v = d,$$

musí d být dělitelné každým společným dělitelem a a b, takže i číslem $\mathrm{nsd}(a, b)$. Nemůže tedy být menší než $\mathrm{nsd}(a, b)$.

Dobrá – kde taková u a v vzít? Kupodivu snadno: trochu upravíme Euklidův algoritmus. Nejprve ale prozradíme, že rovnici

$$a \cdot u + b \cdot v = \mathrm{nsd}(a, b)$$

se říká *Bézoutova identita* a číslům u a v *Bézoutovy koeficienty*.

Dokážeme, že spustíme-li Euklidův algoritmus na čísla a a b, v každém okamžiku se v proměnných `a` a `b` budou nacházet čísla tvaru $\alpha \cdot a + \beta \cdot b$ (kde α a β jsou nějaká celá čísla). Na začátku to triviálně platí, neboť $\texttt{a} = a$ a $\texttt{b} = b$, a pokaždé, když se proměnné mění, buď se prohazují, nebo se jedna odčítá od druhé. Obě tyto operace z výrazů uvedeného tvaru dělají opět výrazy uvedeného tvaru. Takže i konečný výsledek algoritmu, tedy $\mathrm{nsd}(a, b)$, musí jít zapsat v takovém tvaru.

Algoritmus proto upravíme tak, aby si stále udržoval proměnné α_a, α_b, β_a a β_b a vždy platilo

$$\texttt{a} = \alpha_{\texttt{a}} \cdot a + \beta_{\texttt{a}} \cdot b,$$
$$\texttt{b} = \alpha_{\texttt{b}} \cdot a + \beta_{\texttt{b}} \cdot b.$$

Ke konci algoritmu je, jak víme, $\texttt{b} = \mathrm{nsd}(a, b)$, takže $\alpha_{\texttt{b}}$ a $\beta_{\texttt{b}}$ jsou hledané Bézoutovy koeficienty. Opět si to vyzkoušejme na výpočtu $\mathrm{nsd}(1518, 945)$:

a	$\alpha_{\texttt{a}}$	$\beta_{\texttt{a}}$	b	$\alpha_{\texttt{b}}$	$\beta_{\texttt{b}}$
1518	1	0	945	0	1
573	1	−1	945	0	1
573	1	−1	372	−1	2
201	2	−3	372	−1	2
201	2	−3	171	−3	5
30	5	−8	171	−3	5
30	5	−8	21	−28	45
9	33	−53	21	−28	45
9	33	−53	3	−94	151
0	315	−506	3	−94	151

Algoritmus tedy tvrdí, že hledaný nsd splňuje rovnost

$$1518 \cdot (-94) + 945 \cdot 151 = \mathrm{nsd}(1518, 945) = 3.$$

Snadným výpočtem ověříme, že je to pravda. (Poznamenejme, že Bézoutova identita má nekonečně mnoho řešení. Jak byste našli ta další?)

Převeďme své myšlenky do zdrojového kódu. Do `Aa`, `Ba`, `Ab`, `Bb` budeme ukládat koeficienty $\alpha_{\mathtt{a}}$, $\beta_{\mathtt{a}}$, $\alpha_{\mathtt{b}}$ a $\beta_{\mathtt{b}}$.

```
def ExtEuclid(a, b):
    Aa, Ba = 1, 0 # a = 1 * a + 0 * b
    Ab, Bb = 0, 1 # b = 0 * a + 1 * b
    # Prohodíme, je-li třeba
    if b > a:
        a, b = b, a
        Aa, Ab = Ab, Aa
        Ba, Bb = Bb, Ba

    # Zde je vždy a >= b
    while b > 0:
        # Odečteme od proměnné a proměnnou b
        # tolikrát, kolikrát se tam vejde.
        # ("/" značí celočíselné dělení)
        Aa = Aa - (a / b) * Ab;
        Ba = Ba - (a / b) * Bb;

        # nsd(a % b, b) = nsd(a, b).
        a = a % b

        # Prohodíme
        a, b = b, a
        Aa, Ab = Ab, Aa
        Ba, Bb = Bb, Ba
    # nsd(0, a) = a.
    # Vrátíme také Bézoutovy koeficienty.
    return [a, Aa, Ba]
```

Žádná operace, kterou děláme s koeficienty pro proměnné `a` a `b`, netrvá asymptoticky déle než operace modulo. Přidáním počítání Bézoutových koeficientů si tedy časovou složitost Euklidova algoritmu nezhoršíme.

Řešení lineárních kongruencí

Bézoutovy koeficienty jsou užitečné také k řešení kongruencí. Pojďme si to na jedné kongruenci vyzkoušet.

Máme nakoupena 4 vajíčka. V obchodě se vajíčka prodávají pouze v balíčcích po 6 kusech, zatímco my je skladujeme v platech po 20 kusech. Kolik si musíme koupit balíčků, abychom neměli v žádném platu volno?

Přepišme si tento příklad do formy kongruence:

$$4 + 6 \cdot x \equiv 0 \pmod{20},$$

čili

$$6 \cdot x \equiv 16 \pmod{20}.$$

To je totéž, jako že pro x a nějaké další celé číslo y platí

$$6 \cdot x + 20 \cdot y = 16.$$

Ejhle, to je rovnice podobná Bézoutově identitě. Kdyby na její pravé straně byl $\mathrm{nsd}(6, 20) = 2$, byla by to přesně Bézoutova identita a rozšířený Euklidův algoritmus by nám prozradil, že platí

$$6 \cdot (-3) + 20 \cdot 1 = 2.$$

Tím bychom měli vyřešeno.

Jenže v našem případě je na pravé straně 8krát víc, než bychom potřebovali. Tak obě strany Bézoutovy identity vynásobíme 8:

$$6 \cdot (-3) \cdot 8 + 20 \cdot 1 \cdot 8 = 2 \cdot 8.$$

Řešením naší rovnice tedy je $x = -24$, $y = 8$.

Tak hurá do obchodu nakoupit -24 balíčků vajec. Cože? Že záporné nemají? Nevadí – stačí si vzpomenout, že jsme původně počítali modulo 20, takže k x můžeme přičíst libovolný násobek 20 a dostaneme další řešení. Můžeme tedy jít třeba pro 16 balíčků.[21]

Teď už můžeme zformulovat obecný *návod na řešení kongruence*

$$ax \equiv b \pmod{n}$$

s neznámou x. Kongruenci přepíšeme do tvaru

$$ax - ny = b$$

a označíme $d = \mathrm{nsd}(a, n)$. Rozlišíme 3 případy:

- $d = b$ … tehdy jsou hledaná x a y rovná Bézoutovým koeficientům a najdeme je rozšířeným Euklidovým algoritmem.

[21] Mimochodem, není to nejmenší počet balíčků, který vyhovuje úloze: 6 balíčků by také fungovalo. Rozmyslete si, jak najít *nejmenší* řešení kongruence.

- $d \setminus b$ ... pak najdeme řešení x' a y' rovnice s d na pravé straně a položíme $x = x' \cdot b/d$ a $y = y' \cdot b/d$.
- b není násobkem d ... v tomto případě kongruence nemůže mít žádné řešení, neboť levá strana rovnice je pro každé x a y dělitelná d, zatímco pravá strana dělitelná d nikdy není.

Inverzní prvky modulo m

Vraťme se teď zpátky z dlouhé odbočky a zkusme se znovu zamyslet nad tím, kdy je vynásobení obou stran kongruence ve tvaru

$$x \equiv z \pmod{m}$$

konstantou k ekvivalentní úprava. Tak říkáme úpravě, která neubírá ani nepřidává řešení. Už máme dokázáno, že když $x \equiv z$, tak pro každé k platí i $k \cdot x \equiv k \cdot z$. Takže zbývá zajistit, aby každé řešení kongruence $k \cdot x \equiv k \cdot z$ bylo i řešením $x \equiv z$.

Nejprve ukážeme, že pokud k je soudělné s m, je naše snaha předem ztracená. Označme $d = \mathrm{nsd}(k, m) > 1$. Vezměme libovolnou dvojici x a z splňující kongruenci $x \equiv z$, což je totéž jako $x - z \equiv 0$. Nyní vytvořme novou dvojici $x' = x$ a $z' = z + m/d$. Pro tu dostaneme

$$x' - z' \equiv x - (z + m/d) \equiv x - z - m/d \equiv -m/d \not\equiv 0.$$

Ovšem kongruenci vynásobenou k tato nová dvojice stále splňuje:

$$\begin{aligned} kx' - kz' \equiv kx - k(z + m/d) \equiv kx - kz - km/d \equiv \\ \equiv -km/d \equiv m \cdot (-k/d) \equiv 0. \end{aligned}$$

Dokázali jsme tedy, že pokud číslo k, kterým násobíme obě strany kongruence, je soudělné s modulem m, nejedná se o ekvivalentní úpravu. Teď naopak ukážeme, že jsou-li k a m nesoudělná, ekvivalentní to je.

Nahlédneme, že kdykoliv $k \perp m$, existuje nějaké číslo $k^{-1} \in \mathbb{Z}_m$ takové, že $k \cdot k^{-1} \equiv 1$. Tomuto číslu se říká *inverzní prvek ke k* (nebo také *multiplikativní invers čísla k*) a pokud jím kongruenci $kx \equiv kz$ vynásobíme, získáme

$$k \cdot k^{-1} \cdot x \equiv k \cdot k^{-1} \cdot z \pmod{m},$$

což je kýžená kongruence $x \equiv z$.

Kongruenci $k \cdot k^{-1} \equiv 1$ přitom už umíme vyřešit – předchozí kapitola nám říká, že takové k^{-1} existuje právě tehdy, je-li $k \perp m$, a že se dá najít Euklidovým algoritmem. Dodejme ještě, že prvkům, které mají multiplikativní invers, se říká *invertibilní prvky modulo m*.

Konečná tělesa

Když speciálně zvolíme za m nějaké prvočíslo, budou všechny prvky $\mathbb{Z}_m$ kromě nuly invertibilní. Tím pádem se $\mathbb{Z}_m$ bude chovat dost podobně racionálním

nebo reálným číslům. Má s nimi například tyto společné vlastnosti (sčítáním a násobením v případě $\mathbb{Z}_m$ myslíme operace modulo m):

- *Sčítání* je asociativní a komutativní.
- Pro každé a platí $a + 0 = a$.
- Pro každé a existuje $(-a)$ takové, že $a + (-a) = 0$.
- *Násobení* je asociativní a komutativní.
- Pro každé a je $a \cdot 1 = a$.
- Pro každé nenulové a existuje a^{-1} takové, že $a \cdot a^{-1} = 1$.
- Násobení a sčítání jsou distributivní: $a \cdot (b - c) = a \cdot b - a \cdot c$.

Obecněji, máme-li libovolnou množinu, můžeme v ní označit „jedničku“ a „nulu“ a „přibalit“ operace sčítání, násobení, „dej mi $(-a)$“ a „dej mi a^{-1}“. Operací přitom myslíme libovolnou funkci, která prvkům množiny nebo jejich dvojicím přiřazuje prvky. Pokud navíc pro naši množinu s operacemi platí všechny vyjmenované vlastnosti, říká se jí *komutativní těleso*.

Racionální, reálná i komplexní čísla jsou příklady takových těles a my jsme k nim přidali *konečná tělesa* velikosti prvočísla. (Na okraj poznamenejme, že je známo, že všechna konečná tělesa mají velikost mocniny prvočísla, což ovšem neznamená, že se vždy chovají jako celá čísla modulo nějakým m.)

Malá Fermatova věta

S prvočísly úzce souvisí takzvaná *Malá Fermatova věta.* Říká, že pokud je p prvočíslo a a libovolné číslo od 1 do $p - 1$, tak

$$a^{p-1} \equiv 1 \pmod{p}.$$

Tato věta má mnoho různých použití (třeba ve známém šifrovacím algoritmu RSA nebo níže v algoritmu na testování prvočíselnosti), nám se bude především hodit jako další způsob invertování čísel modulo prvočíslo:

$$a^{p-2} \cdot a \equiv a^{p-1} \equiv 1 \pmod{p},$$

takže a^{p-2} je inverzní prvek k a.

Pojďme nyní Malou Fermatovu větu dokázat. Indukcí podle a budeme dokazovat ekvivalentní tvrzení

$$a^p \equiv a \pmod{p}.$$

(Jelikož a je určitě nesoudělné s modulem p, tak už víme, že násobení obou stran kongruence je ekvivalentní úprava.)

Pro $a = 1$ je snadné vidět, že věta platí:

$$a^p = 1^p = 1 \equiv 1 \pmod{p}.$$

Teď uděláme indukční krok. Řekněme, že máme dokázáno, že naše věta platí pro nějaké a, a chceme ji dokázat i pro $a+1$. K tomu se bude hodit známá Binomická věta, která říká, že pro každé reálné x a y a přirozené n platí:

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i},$$

přičemž $\binom{n}{i}$ je takzvané *kombinační číslo* tvaru

$$\binom{n}{i} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-i+1)}{i \cdot (i-1) \cdot \ldots \cdot 1},$$

mající v čitateli i jmenovali zlomku právě i členů.

Indukce po nás chce, abychom dokázali, že $(a+1)^p \equiv a$. Rozepíšeme tedy levou stranu kongruence pomocí Binomické věty:

$$(a+1)^p = \binom{p}{0} a^0 + \binom{p}{1} a^1 + \ldots + \binom{p}{p} a^p.$$

Jelikož $\binom{p}{0}$ i $\binom{p}{p}$ jsou rovny 1, tvoří první a poslední člen součtu dohromady a^p+1. To je podle indukčního předpokladu kongruentní s a.

Zbývá tedy dokázat, že všechny ostatní členy jsou dělitelné p, takže se v kongruenci modulo p neprojeví. Vskutku: pro $0 < i < p$ se ve zlomku definujícím $\binom{p}{i}$ objeví p v prvočíselném rozkladu čitatele, ale ne v rozkladu jmenovatele, takže se nemá s čím zkrátit.

Tím je indukce hotova.

Fermatův test prvočíselnosti

Jako malou odměnu za dlouhý důkaz předvedeme, jak Malou Fermatovu větu využívat ke zjištění, zda je nějaké obrovské číslo n prvočíslem. Jistě bychom mohli zkoušet všechny kandidáty na dělitele od 2 do $n-1$ (nebo chytřeji do $\lfloor \sqrt{n} \rfloor$), ale to by trvalo příliš dlouho.

Raději zkusíme vybrat nějaké náhodné $a \in \{1, \ldots, n-1\}$ a spočítat, kolik je $a^{n-1} \bmod n$. Pro prvočíselné n musí vyjít jednička, takže pokud vyjde něco jiného, usvědčili jsme n z toho, že není prvočíslem (aniž jsme našli jediného dělitele – zvláštní, že?).

Pokud pro toto konkrétní a jednička vyjde, samozřejmě to neznamená, že n je určitě prvočíslo. Vyzkoušíme proto několik různých a a pokud test pro žádné z nich neselže, drze prohlásíme, že n je pravděpodobně prvočíslo.

Jak moc velká drzost to je? Překvapivě ne moc velká. Pro skoro každé složené číslo n platí, že alespoň polovina a-ček dosvědčí, že se nejedná o prvočíslo. Takže jeden pokus selže s pravděpodobností nejvýše $1/2$ a pokud uděláme t pokusů, pravděpodobnost chybného výsledku je nanejvýš $1/2^t$.

Jedinou výjimku z našeho pravidla tvoří tzv. *Carmichaelova čísla* (nejmenší z nich je číslo 561). To jsou čísla, jejichž složenost prokážeme jen tehdy, když se strefíme do a soudělného s n, a takových a je velmi málo. Naštěstí není Carmichaelových čísel moc (relativně k prvočíslům), takže Fermatův test funguje docela spolehlivě.

Existují i důmyslnější testy, které se Carmichaelovými čísly obalamutit nenechají. Jejich popis, jakož i důkaz našeho tvrzení o spolehlivosti Fermatova testu, najdete v literatuře zmíněné na konci kuchařky.

Rychlé mocnění

Ve Fermatově testu nebo při počítání inverzí pomocí Malé Fermatovy věty potřebujeme spočítat $a^k \bmod m$ pro velké k. Pokud budeme mocninu a^k počítat přímo podle definice, tedy jako $a \cdot a \cdot \ldots \cdot a$, budeme potřebovat $\mathcal{O}(k)$ násobení, což je příliš.

Jednoduchou fintou lze počet operací snížit na $\mathcal{O}(\log k)$. Například a^{16} můžeme spočítat jako:

$$a^{16} = (a^8)^2 = ((a^4)^2)^2 = (((a^2)^2)^2)^2.$$

Pro obecný exponent bude rychlejší umocňování vypadat takto:

```
def FastExp(a, k):
    # Nejdříve ošetříme triviální případy.
    if k == 0: return 1
    if k == 1: return a

    # Když je x sudé, vrátíme a^(k/2) * a^(k/2).
    # Když je x liché, vrátíme a * a^(k - 1).

    if k % 2 == 0:
        i = FastExp(a, k / 2)
        return i * i
    else:
        return a * FastExp(a, k - 1)
```

Každé volání `FastExp` pro sudé k jednou zavolá `FastExp` s polovičním k a jednou vynásobí dvě čísla. Když je k liché, převede se na sudé a provede se jedno vynásobení. `FastExp` tedy provede $\mathcal{O}(\log k)$ násobení.

Je ale důležité uvědomit si, že kdybychom si neuložili výsledek $a^{k/2}$ do pomocné proměnné `i`, ale rovnou vraceli `FastExp(a, k/2) * FastExp(a, k/2)`, byl by náš kód stejně pomalý, jako kdybychom počítali mocninu podle definice!

Pro použití ve Fermatově testu (nebo obecně na spočítání $a^k \bmod m$) stačí po každém násobení výsledek vymodulit m.

Síto na prvočísla

Už jsme zjistili, že se nám hodí umět najít prvočísla. Kde je ale vezmeme? Můžeme určitě zkoušet jedno číslo po druhém a pokaždé otestovat, jestli držíme prvočíslo (třeba Fermatovým testem nebo zkoušením všech dělitelů). Už staří Řekové ale znali algoritmus, který najde všechna prvočísla menší než n efektivněji. Říká se mu *Eratosthenovo síto*.

Síto funguje na docela jednoduchém principu. Budeme si uchovávat v paměti pro každé číslo od 2 do n příznak, jestli je prvočíslo, nebo složené. Začneme u dvojky a označíme všechny násobky 2 ležící mezi 4 a n jako složená čísla. Další prvočíslo je 3. Označíme všechny násobky 3 ležící od 6 do n jako složená čísla. Další číslo na řadě je 4. Když jsme ale vyškrtávali násobky 2, vyškrtli jsme i 4. Nebudeme tedy provádět nic a rovnou přejdeme na 5. Takto najdeme všechna prvočísla od 2 do n a stihneme to rychle.

Ukažme si ještě zdrojový kód v Pythonu:

```
def Eratosthenes(n):
    prvocislo = [ True ] * n

    for i in range(2, n):
        if prvocislo[i]:
            print("%d je prvocislo." % i)

            # Násobky prvočísla jsou složené
            j = i * 2
            while j < n:
                prvocislo[j] = False
            j += i
```

Jak dlouho síto poběží? Dá se dokázat, že jeho asymptotická časová složitost činí $\mathcal{O}(n \log \log n)$, ale není to snadné. My si zde předvedeme jenom slabší odhad $\mathcal{O}(n \log n)$. Zájemce o těžší důkaz menší složitosti odkazujeme na vzorové řešení úlohy 24-3-5.[22]

Síto tráví čas $\mathcal{O}(n)$ hledáním prvočísel a mezitím škrtá jejich násobky. Když škrtáme násobky dvojky, vyškrtneme nejvýše $n/2$ čísel, když škrtáme násobky trojky, vyškrtneme jich nejvýše $n/3$, atd. Složitost Eratosthenova síta tedy bude shora omezena součtem

$$n + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{n} = n \cdot \sum_{i=1}^{n} \frac{1}{i}.$$

[22] `http://ksp.mff.cuni.cz/viz/24-3-5/reseni`

Sumě

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

se říká n-té harmonické číslo a dokážeme o něm, že leží v $\mathcal{O}(\log n)$.

Uvažujme, o co se zvětší H_{2n} oproti H_n:

$$H_{2n} - H_n = \frac{1}{n+1} + \frac{1}{n+2} + \ldots + \frac{1}{2n}.$$

To je součet n členů, z nichž každý je menší než $1/n$. Celý součet je tedy menší než 1.

Zjistili jsme tedy, že $H_{2n} < H_n + 1$. Funkce, které rostou takhle pomalu, jdou shora omezit nějakým logaritmem n, takže $H_n = \mathcal{O}(\log n)$.

Proto složitost celého Eratosthenova síta činí $\mathcal{O}(n \log n)$.

Čínská zbytková věta

Následující věta dostala své jméno po staročínském způsobu počítání vojáků. Čínská armáda je velká, a kdybychom chtěli počítat vojáky jednoho po druhém, trvalo by to dlouho. Pomáhalo prý armádu rozřadit do řad o velikostech m_1, m_2, …, m_n (součin všech m_i si označíme jako M bez indexu). Někdy zbyli nezařazení dva, někdy třicet, někdy se seřadili všichni. Tyto zbytky si označíme z_1, z_2, …, z_n.

A co Čínská zbytková věta říká? Tvrdí, že když jsou všechna m_i navzájem nesoudělná a počet vojáků je menší než M, lze ho ze zbytků z_i jednoznačně určit. Když například rozdělujeme vojáky do řad velikostí 2, 3, 5, 7, 11 a 13, můžeme zbytky z řad jednoznačně vyjádřit každý počet vojáků menší než $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 30\,030$.

Formálněji řečeno: Jsou-li dána navzájem nesoudělná přirozená čísla $m_1, \ldots, m_n$ (jejichž součin označíme M) a zbytky $z_1, \ldots, z_n$, pak existuje právě jedno číslo $x \in \mathbb{Z}_M$ takové, že pro všechna i je

$$x \equiv z_i \pmod{m_i}.$$

A jak dokážeme, že něco takového platí? Mějme 2 čísla $a, b \in \mathbb{Z}_M$ taková, že mají stejné zbytky po dělení všemi m_i. Ukážeme, že musí nutně být stejná.

Víme, že pro všechna i platí

$$a \equiv b \pmod{m_i}.$$

To podle definice kongruence znamená, že rozdíl $a - b$ je dělitelný všemi m_i. Proto je dělitelný i nejmenším společným násobkem všech m_i, což ovšem díky nesoudělnosti musí být jejich součin M.

Máme tedy dvě čísla ze $\mathbb{Z}_M$, jejichž rozdíl je dělitelný M. To nutně znamená, že jsou stejná.

Dokázali jsme tedy, že jedna sada zbytků $z_1, \ldots, z_n$ odpovídá jednoznačně určenému číslu $x \in \mathbb{Z}_M$, ale ještě nevíme, jak bez zkoušení všech možností toto x najít. Půjdeme na to od lesa.

Nejprve se hodí všimnout si toho, že když sečteme dvě čísla, sečtou se i jejich zbytky modulo všemi m_i.

Co kdybychom nyní dokázali sehnat čísla $Q_1, \ldots, Q_n$ taková, že Q_j je dělitelné všemi m_i kromě m_j a že $Q_j \equiv 1 \pmod{m_j}$?

To by potom stačilo položit

$$x = (z_1 \cdot Q_1 + z_2 \cdot Q_2 + \ldots + z_n \cdot Q_n) \bmod M.$$

Vskutku: počítáme-li $x \bmod m_i$, všechny členy $z_j \cdot Q_j$ pro $j \neq i$ vyjdou nulové a člen $z_i \cdot Q_i$ bude roven z_i. To, že celý výsledek nakonec vymodulíme M, na věci nic nemění, protože přičtení či odečtení libovolného násobku M zbytek po dělení žádným m_i neovlivní.

Jak se ale k číslům Q_i dostaneme? Číslo Q_i má být dělitelné všemi m_j kromě m_i. Uvažujme tedy součin

$$S_i = m_1 \cdot \ldots \cdot m_{i-1} \cdot m_{i+1} \cdot \ldots \cdot m_n.$$

Ten modulo každé m_j $(j \neq i)$ dá nulu, zatímco modulo m_i nějaké číslo r_i nesoudělné s m_i (nesoudělné musí být, protože jinak by m_i bylo soudělné s některým m_j). Speciálně to znamená, že r_i není 0.

Potřebujeme tedy z tohoto nenulového zbytku udělat jedničku. To zařídíme snadno: pořídíme si r_i^{-1}, což bude inverzní prvek k r_i modulo m_i, a tímto prvkem celé S_i vynásobíme:

$$Q_i = S_i \cdot r_i^{-1}.$$

Toto Q_i už má požadované vlastnosti: $Q_i \bmod m_j$ pro $j \neq i$ vyjde nulové, protože Q_i je násobkem S_i, které bylo dělitelné m_j. A modulo m_i získáme

$$Q_i \equiv S_i \cdot r_i^{-1} \equiv r_i \cdot r_i^{-1} \equiv 1.$$

„Kouzelná“ čísla Q_i tedy dokážeme sestrojit a jejich zkombinováním i hledané x.

Pojďme si to teď zkusit v praxi. Chceme najít nejmenší x takové, že platí následující kongruence:

$$\begin{aligned} x &\equiv 3 \pmod{5} \\ x &\equiv 1 \pmod{9} \\ x &\equiv 14 \pmod{16} \end{aligned}$$

Spočítáme si nejdříve M a všechna S_i:

$$\begin{aligned} M &= 5 \cdot 9 \cdot 16 = 720, \\ S_1 &= 9 \cdot 16 = 144, \\ S_2 &= 5 \cdot 16 = 80, \\ S_3 &= 5 \cdot 9 = 45. \end{aligned}$$

Teď zjistíme, kolik vychází každé S_i modulo m_i a určíme příslušné multiplikativní inverze (například pomocí rozšířeného Euklidova algoritmu):

$$\begin{aligned} r_1 &= 144 \bmod 5 = 4, \\ r_2 &= 80 \bmod 9 = 8, \\ r_3 &= 45 \bmod 16 = 13, \\ r_1^{-1} &= 4 \quad (4 \cdot 4 \bmod 5 = 1), \\ r_2^{-1} &= 8 \quad (8 \cdot 8 \bmod 9 = 1), \\ r_3^{-1} &= 5 \quad (13 \cdot 5 \bmod 16 = 1). \end{aligned}$$

Z toho vypočteme Q_i jako $S_i \cdot r_i^{-1}$:

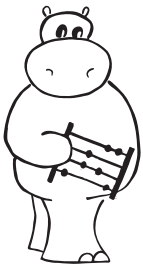
$$\begin{aligned} Q_1 &= S_1 \cdot r_1^{-1} = 144 \cdot 4 = 576, \\ Q_2 &= S_2 \cdot r_2^{-1} = 80 \cdot 8 = 640, \\ Q_3 &= S_3 \cdot r_3^{-1} = 45 \cdot 5 = 225. \end{aligned}$$

Nakonec sečteme příslušné násobky Q_i a zjistíme x:

$$x \equiv 3 \cdot 576 + 1 \cdot 640 + 14 \cdot 225 = 5518 \equiv 478 \pmod{720}.$$

Výsledek opravdu vypadá správně:

$$x = 478 = 3 + (5 \cdot 95) = 1 + (9 \cdot 53) = 14 + (16 \cdot 29).$$



Pár slov na závěr

Doufáme, že se vám naše povídání o teorii čísel líbilo a že jste poznali, že i tak základní objekty, jako jsou celá čísla, mají spousty zajímavých vlastností.

Přejete-li si dozvědět se více o prvočíselných testech nebo o RSA, můžeme navrhnout ke studiu textík *Algoritmy okolo teorie čísel*[23] od jednoho z autorů kuchařky. Důkladný rozbor Eratosthenova síta a jiné zajímavosti o prvočíslech najdete v článku *Tři věty o prvočíslech*[24] od téhož autora.

S teorií čísel také souvisí algebra, která zobecňuje různé poznatky na libovolné množiny opatřené nějakými operacemi (například tělesa). Máte-li o ni zájem, mohla by vám pomoci například skripta *Základy algebry* od Davida Stanovského.

Michal Pokorný a Martin Mareš

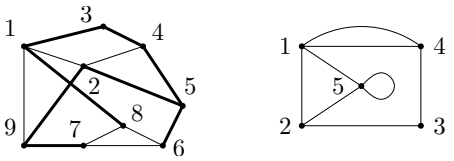
[23] `http://mj.ucw.cz/papers/numth.pdf`
[24] `http://mj.ucw.cz/papers/bert.pdf`

Kuchařka čtvrté 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 takové posloupnosti vrcholů a hran tvaru $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$, 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

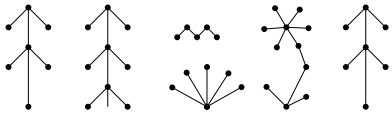
díváme jako na podgrafy, které získáme tak, že z grafu vypustíme všechny ostatní vrcholy a hrany.

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

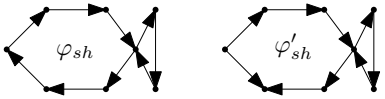
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny.*

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 a 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 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: polem sousedů $S[1 \dots M]$ obsahujícím postupně čísla všech vrcholů, do kterých vede hrana z vrcholu 1, pak z vrcholu 2 atd., a polem začátků $Z[1 \dots N]$, 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 $S[Z[i]], \dots, S[Z[i + 1] - 1]$. 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
$S[i]$	2	3	8	9	1	4	5	9	1	4	2	3	5	2

i	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$S[i]$	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 v 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.
```

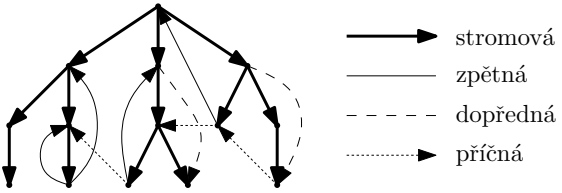
Průběh prohledávání grafu do hloubky můžeme znázornit stromem (říká se mu DFS strom – podle anglického názvu Depth-First Search pro prohledávání do hloubky). Z počátečního vrcholu w učiníme kořen. Pak budeme graf procházet do hloubky a vrcholy zakreslovat jako syny vrcholů, ze kterých jsme přišli. Syny 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

Prohledávání do šířky je založené 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 proměnnou $p = 1$.
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.

Martin Mareš, David Matoušek a Petr Škoda

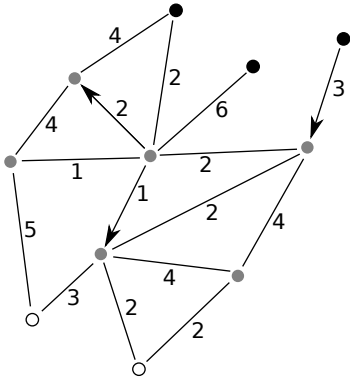
Kuchařka páté série – toky v sítích

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 vězte, 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.



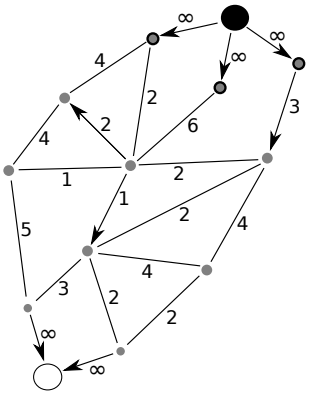
Každá trubka může a nemusí mít definováno, kterým směrem jí má téci ropa. Pro každou trubku zvlášť víme, kolik nejvýše jí za hodinu protlačíme.

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 uniklé ropě utopili vše živé.

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.



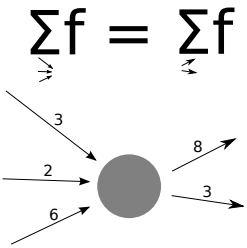
Abychom měli situaci jednodušší, zbavíme se hned na úvod mnohočetnosti zdrojů a stoků. Přikreslíme si dva nové vrcholy – z nadzdroje budeme posílat ropu do všech zdrojů, do nadstoku budeme posílat ropu ze všech stoků. Kapacitu přikreslených hran pak nastavíme na nekonečno.

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:

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

$$\forall v \in V \setminus \{z, s\} : \sum_{\overrightarrow{uv} \in E} f(\overrightarrow{uv}) = \sum_{\overrightarrow{vu} \in E} f(\overrightarrow{vu})$$

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.

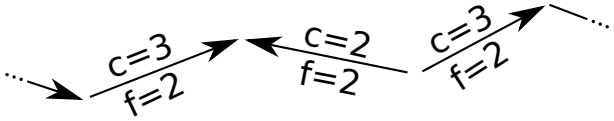
Předvedeme si onen první způsob a algoritmus, který se podle svých autorů jmenuje Fordův-Fulkersonův. Bude se nám odteď hodit tvářit se, jako že mezi každými dvěma vrcholy vede oběma směry hrana. Tam, kde ze vstupu nepřišla, si domyslíme jednu s nulovou kapacitou.

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. Odvolám 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.[25]

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. Zkusme najít nějaký lepší.

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[26] z jedné Medvědovy přednášky předmětu ADS2, ukázka druhého přístupu k řešení hledání maximálního toku je na záznamu[27] jejího pokračování.

[25] `http://kam.mff.cuni.cz/~valla/kg.html`
[26] `http://mj.ucw.cz/vyuka/1112/ads2/3-dinic.pdf`
[27] `http://mj.ucw.cz/vyuka/1112/ads2/4-goldberg.pdf`

K zamyšlení

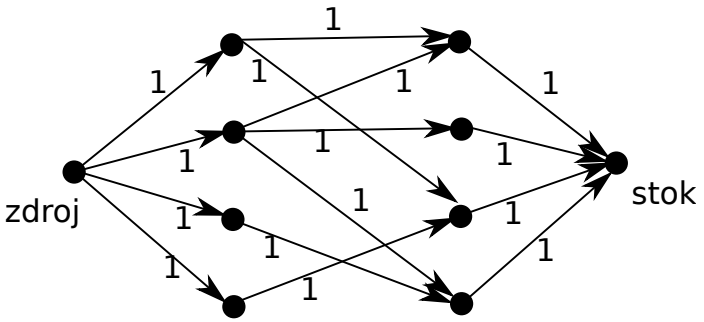
- Důležitou vlastností algoritmu je, že když dostane celočíselné kapacity, vrátí celočíselný tok. Bude se nám to hodit v aplikacích. Dokážete to?
- Rozdíl mezi Fordem-Fulkersonem, který hledá cesty obecným způsobem, a takovým, který to dělá prohledáváním do šířky, je ze složitostního hlediska docela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

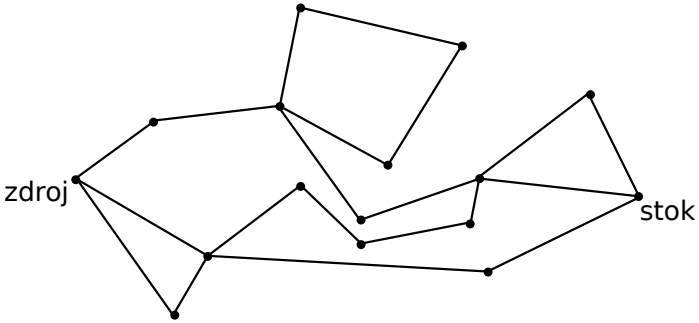
Hledání hranově a vrcholově disjunktních cest

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 existuje 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)[28] 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 navíc počet hranově/vrcholově disjunktních cest roven stupni hranové/vrcholové souvislosti grafu – máme tedy nyní algoritmus, který ji najde.

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

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 náhodou neupravený Edmondsův-Karpův algoritmus rychleji, pokud je graf, jak jsme teď opakovaně viděli, ohodnocený toliko nulami a jedničkami?

Lukáš Lánský

Vzorová řešení

25-1-1 Fotografování

Vzorové riešenie nevyužíva žiadnu prevratnú myšlienku a taktiež nepoužíva žiadnu štruktúru, s ktorou by sa nestretol začiatočník. Vystačíme si so spojákmi a obyčajnými poliami a časová zložitosť vyjde lineárna.

Algoritmus prebehne v n krokoch – v každom kroku odoberieme jeden vrchol v a priradíme mu číslo k_v. Vytvoríme si n-prvkové pole V také, že $V[x] = \deg(x)$. Vždy odoberáme vrchol, ktorý má najmenšiu hodnotu vo V (ak ich je viac, tak vezmeme ľubovoľný) a všetkým jeho susedom s *väčšou* hodnotou vo V znížime hodnotu vo V o 1. Pre vrchol u položíme k_u rovné $V[u]$, pri ktorom sme ho odobrali.

Aby sme nahliadli správnosť algoritmu, stačí ukázať, že pri odobraní bude mať každý vrchol u nastavenú správnu hodnotu vo V. Na začiatku sú určite všetky hodnoty nastavené správne. Ďalej postupujme indukciou. Predstavme si, že odoberáme nejaký vrchol u. Z indukčného predpokladu mu nastavíme správne k_u. Uvážme teraz ľubovoľný vrchol x taký, že $V[x] = V[u]$. Vrcholu x nemôžeme znížiť $V[x]$ o 1, pretože by to znamenalo, že mu priradíme $k_x < V[u]$, čo samozrejme nemôže byť pravda – pre takéto k_x by ešte ostal v hre. Vrcholom s $V[x] > V[u]$ musíme stupeň znížiť, lebo vrchol u vypadne z hry skôr ako akýkoľvek takýto vrchol x.

Časová zložitosť algoritmu zavisí dosť od implementácie. Mnohí použili haldu, v ktorej mali uložené vrcholy podľa stupňa a z toho im vyliezli v zložitosti nejaké logaritmy. To ale vôbec nie je nutné. Stačí si vytvoriť pole veľké n, indexujme ho od 0 do $n-1$. Na i-tej pozícii sa bude nachádzať spoják s vrcholmi stupňa i. Toto pole budeme prechádzať od nultej pozície.

Predstavme si, že sme na nejakej pozícii k. Kým sú v príslušnom spojáku nejaké vrcholy, tak prvý odoberieme a všetkých jeho susedov v konštantnom čase presunieme na správnu pozíciu. K tomu sa nám bude hodit si pre každý vrchol pamätať, kde sa nachádza. Ak sme už pre aktuálne k celý spoják vyčerpali, zvýšime k.

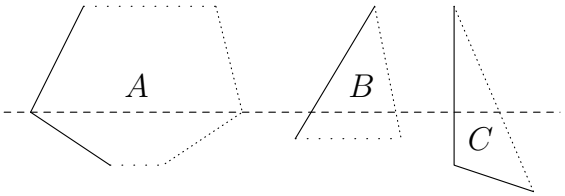
Vyššie popísaná implementácia nám zaručí lineárnu časovú zložitosť. Na každý vrchol sa totižto pozrieme práve raz (keď ho odoberáme) a zároveň nastavíme stupeň každému jeho susedovi. Časová zložitosť je teda $\mathcal{O}(n+m)$, kde m je počet dvojíc. Pamäťová je rovnaká ako časová.

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-1-1.cpp`

Peter Zeman

25-1-2 Stánky na náměstí

Pro zjištění, zdali mají dva stánky kolizi mezi sebou, použijeme tzv. sweep-line („zametací přímku“). Představíme si, že budeme mít pomyslnou přímku rovnoběžnou (např.) s osou X a budeme s ní posouvat z $y = -\infty$ do $y = +\infty$. Tímto nám postupně protne všechny stánky (viz následující obrázek).



Na něm máme znázorněny stánky A, B a C a pomyslnou přímku zobrazenou čárkovaně. Plnou čárou jsou označeny levé okraje mnohoúhelníků (stánků), hustě tečkovanou pravé okraje a řídce tečkovanou okraje rovnoběžné s osou X.

Jak si můžeme všimnout, mnohoúhelníky nám rozdělují přímku na několik intervalů (dle jejich průsečíků). Bez kolize stánků se nám na sweep-line pravidelně střídají jejich levé a pravé okraje. Pokud bychom měli dva levé (či pravé) okraje vedle sebe, došlo by k překrytí vnitřků mnohoúhelníků.

Základem programu tedy bude udržovat si informace o tom, jak vypadá rozdělení na intervaly odpovídající jednotlivým mnohoúhelníkům. V této struktuře budeme potřebovat být schopni rychle provést následující:

- Přidat mnohoúhelník – ve chvíli, kdy se sweep-line dotkne jeho spodního okraje
- Odebrat mnohoúhelník – ve chvíli, kdy sweep-line opustí jeho nejvyšší bod
- Opravit intervaly ve chvíli, kdy narazíme na bod, kde se levý či pravý okraj láme (tj. kdy se dostaneme na některý vrchol mnohoúhelníku)

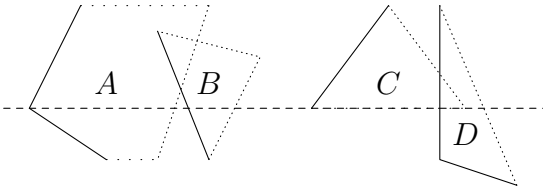
První operace vyžaduje, abychom byli schopni rychle vyhledat, které mnohoúhelníky budou vlevo a vpravo od vkládaného. Proto pro reprezentaci rozdělení sweep-line stánky budeme používat intervalový strom[29] (v programu užit AVL strom kvůli vyvažování – viz kuchařku o vyhledávacích stromech).[30] Tím dosáhneme vyhledání, kam máme nový stánek zatřídit, v čase $\mathcal{O}(\log T)$.

V klasickém intervalovém stromu však ukládáme krajní body – ty se nám ale mění, jak se posouvá sweep-line, a přepočítavat je po každém kroku je pracné. Můžeme si však všimnout, že dokud nedojde ke zkřížení okrajů mnohoúhelníků

[29] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`
[30] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`

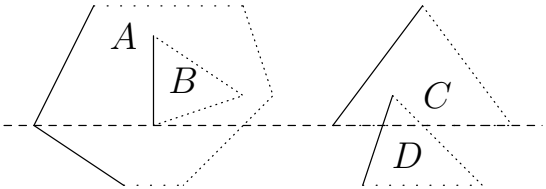
(viz obrázek níže), nebude se měnit pořadí, v jakém intervaly budou na přímce za sebou. Odtud je už jen krůček k myšlence, že není nutné okraje intervalu reprezentovat pomocí dvou bodů, ale je možné užít i dvou úseček (okraje mnohoúhelníku) a vlastní bod bude průsečíkem úsečky a aktuální sweep-line.

Jak můžou kolize stánků (z hlediska přímky) vypadat? Jedna možnost je, že dojde ke zkřížení okrajů (A s B, či C s D).

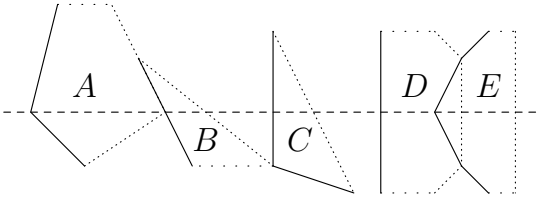


To nemusí být nalezeno jen při vkládání nového mnohoúhelníku, ale i po dosažení vrcholu některého stánku a změně „aktuální“ okrajové úsečky.

Další možnost je, že vkládaný mnohoúhelník je uvnitř jiného (B v A) či jiný stánek bude mezi okraji vkládaného (D v C).



Tyto situace se však v intervalovém stromě snadno detekují – v čase $\mathcal{O}(\log T)$ je možno zjisit, jaký okraj bude levým sousedem vkládaného. Zkřížení okrajů můžeme kontrolovat při každém vkládání okraje do intervalového stromu (lze si rozmyslet, že při vkládání se úsečka porovná s oběma sousedními, pokud existují). A „nekřížící“ situace se snadno detekuje pomocí nalezení levých sousedů vkládaných okrajů (opět čas $\mathcal{O}(\log T)$). První případ (B v A) nastane tehdy, pokud bude sousedem levého okraje levý okraj, druhý případ (D s C) ošetří test, zdali levý soused pravého okraje vkládaného mnohoúhelníku je levý okraj téhož útvaru.



Při dosažení vrcholu, kde se jen láme okraj, stačí na první pohled otestovat zkřížení nové úsečky se sousedy. To je implementováno pomocí odebrání a vložení hrany. V praxi však nastává ještě jeden drobný problém – konkrétně zkřížení ve vrcholu okraje (D s E). Nicméně dotyky okrajů nepovažujeme za překryv (A, B a C) (toho jsme dosáhli tím, že při detekci kolizí neuvažujeme krajní body úseček a při dotyku okrajů je pravý okraj tříděn vlevo od levého).

Jak si snadno čtenář rozmyslí, řešení je analogické testu, zdali nově vkládaný mnohoúhelník je součástí jiného. Konkrétně ozkoušíme, je-li levým sousedem levého okraje nějaký pravý okraj, resp. (v případě, že se „láme“ pravý okraj) zdali je levým sousedem pravého okraje levý okraj téhož mnohoúhelníku.

Vhodnou úpravou lze tuto operaci zrychlit – konkrétně nalézt sousedy v čase $\mathcal{O}(1)$*. Nicméně vzhledem k tomu, že prioritní fronta potřebuje na každou operaci čas* $\mathcal{O}(\log T)$*, tak zpomalení vyřazením a opětovným vložením okraje nám celkovou asymptotickou složitost nezhorší.*

Odebrání mnohoúhelníku ve chvíli, kdy se dostaneme se sweep-line nad něj, je triviální. Tam žádná kolize nevznikne, a je tedy potřeba jen upravit intervalový strom na absenci přislušných dvou okrajů.

Program jen implementuje výše zmíněný postup. Pokud označíme celkový počet vrcholů mnohoúhelníků N a počet stánků T, časovou náročnost můžeme celkově popsat jako $\mathcal{O}(N \log T)$ – údržba stromu stojí $\mathcal{O}(\log T)$ na vložení/odebrání, údržba haldy (prioritní fronty) taktéž (je třeba si uvědomit, že pokud budeme do fronty vkládat vždy jen následující zlom okraje, nebudeme v ní v žádném okamžiku mít více než $\mathcal{O}(T)$ prvků, podobně jako v intervalovém stromě). Paměťová náročnost je lineární vzhledem k velikosti vstupu, tedy $\mathcal{O}(N)$.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/25-1-2.pas`

Pavel Čížek

25-1-3 Řazení hradní stráže

První pozorování: Když si obě řady stejně přečíslujeme (obecně jakkoli přeznačíme), počet nutných přesunů se určitě nezmění. My si tedy přečíslujeme odchozí řadu tak, aby vojáci měli čísla postupně $1, \ldots, N$. Tím jsme úlohu převedli na určení nejmenšího počtu přesunů pro seřazení posloupnosti.

Druhé pozorování: Když musíme přesunout vojáka na i-té pozici, musíme přesunout také všechny, kteří stojí napravo od něj.

Všimneme si, že příchozí řada bude vždy začínat nějakou seřazenou posloupností. Označme délku nejdelší takové posloupnosti jako K. V nejhorším případě je K určitě alespoň 1.

Voják na $(K+1)$-té pozici stojí špatně. Kdyby nestál, měla by maximální seřazená posloupnost délku alespoň o 1 větší. Tohoto vojáka tedy musíme přesunout, a s ním i všechny vojáky napravo od něj. Dohromady tak budeme potřebovat alespoň $N-K$ přesunů.

$N-K$ přesunů nám zároveň stačí. Vojáci na prvních K pozicích jsou vůči sobě správně, nemusíme je tedy přesouvat. Ostatní vojáci se můžou při svém přesunu zařadit na libovolné místo, zařadí se tedy na správné místo. Pro každého z nich tak potřebujeme jen jeden přesun.

Úlohu tedy vyřešíme tak, že si nejprve přečíslujeme obě řady. Na to potřebujeme jednou projít celý vstup, což stihneme v $\mathcal{O}(N)$. Následně budeme procházet příchozí řadu od začátku a vždy zkontrolujeme, jestli má voják vpravo větší číslo. Tím získáme K. V nejhorším případě projdeme řadu celou, tedy opět $\mathcal{O}(N)$. Potřebujeme tři pole o velikosti N, takže paměťová složitost je také $\mathcal{O}(N)$.

Někteří z vás určovali nejen nutný počet přesunů, ale také to, kam se má každý přesouvaný voják zařadit. Těm doporučujeme číst pořádně zadání, ušetří vám to spoustu práce ;) Poznamenáme ale, že kdybychom něco takového chtěli, je nejlepší řešit úlohu pomocí binárního vyhledávacího stromu. Do něj bychom si uložili všechny vojáky ze seřazené části posloupnosti. Pak bychom do něj postupně vkládali vojáky z konce posloupnosti. Podle toho, zda přecházíme do levého, nebo pravého syna, dokážeme říct, na jakou pozici se má daný voják zařadit. Paměťová složitost by v tom případě zůstala $\mathcal{O}(N)$, časová složitost by vzrostla na $\mathcal{O}(N \log N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-1-3.c`

Jiří Setnička a Karolína „Karry“ Burešová

25-1-4 Útěk

Nejdříve se podíváme, jak vypadá řešení pro $N = 0$, tedy hledání nejkratší cesty v bludišti ze startovního políčka $[s_x, s_y]$ do cílového políčka $[s_x, s_y]$.

K řešení budeme využívat datovou strukturu jménem fronta. Fronta funguje jako každá normální fronta. Každý prvek, který do ní přidáme, se zařadí na konec a každý prvek, který odebíráme, odebereme ze začátku. Obojí zvládneme v konstantním čase.

Algoritmus, který zde použijeme, se jmenuje prohledávání do šířky, pomocí něho zjistíme nejkratší vzdálenost každého políčka od startovního. Tyto vzdálenosti si budeme pamatovat v dvourozměrném poli D (na začátku inicializováno na -1).

Na začátku přidáme startovní políčko do fronty a nastavíme $D[s_y][s_x] = 0$. Nyní, dokud fronta není prázdná, odebereme políčko p z fronty a všechna sousední

políčka q, která nejsou zdmi a mají hodnotu D rovnu -1, přidáme do fronty a položíme $D[q_y][q_x] = D[p_y][p_x] + 1$.

Na algoritmu je vidět, že políčka zpracováváme v pořadí dle jejich vzdálenosti od startu, tedy u každého políčka nyní máme spočítanou jeho vzdálenost od startu. Pokud tento fakt hned nevidíte, tak si průběh algoritmu nakreslete do nějakého bludiště.

Časová složitost je $\mathcal{O}$(velikost bludiště), každé políčko právě jednou přidáme do fronty, právě jednou jej odebereme a u každého políčka se díváme jen na 4 sousedy.

Nyní už jen zbývá vypsat, jak jsme se do cíle dostali. To můžeme jednoduše udělat tak, že si v průběhu algoritmu kromě vzdáleností navíc budeme ukládat, z jakého políčka jsme se do něj dostali. Pak jsme schopni pomocí těchto zpětných odkazů získat posloupnost políček z cíle do startu, tuto posloupnost pak stačí jen otočit a máme, co jsme chtěli.

Nyní se podívejme na variantu pro $2 \geq N > 0$. Tentokrát už nám nestačí jen přímočaře procházet bludiště, protože ještě musíme zohledňovat polohy osob.

Opět použijeme procházení do šířky, ale tentokrát nebudeme procházet jen mapu bludiště, ale něco, čemu se říká stavový prostor. Stavový prostor je nějaká množina stavů, kde z některých stavů můžeme přecházet do jiných. Například v bludišti jsou stavy jednotlivá políčka.

Každá osoba i má dané cyklické pořadí k_i políček, která navštěvuje, budeme u ní tedy rozlišovat k_i stavů.

Jednotlivými stavy pro průchod do šířky budou všechny možné pozice nás a čísla pozic osob, při kterých nestojíme na políčku zároveň s osobou. Přechody mezi stavy budou odpovídat jednomu pohybu nás a osob, při kterém se nestřetneme.

Na tomto stavovém prostoru nyní použijeme prohledávání do šířky a jsme hotovi. Ještě poznamenejme, že abychom omezili velikost stavového prostoru, nebudeme brát všechny možné pozice osob, ale že stačí vzít jen nejmenší společný násobek velikostí jejich okruhů. Na tento algoritmus se můžete podívat ve vzorovém zdrojovém kódu.

Časová složitost je tedy $\mathcal{O}$(počet stavů) = $\mathcal{O}$(velikost bludiště $\cdot$ nsn(k_1, k_2)).

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-1-4.cpp`

Karel Tesař

25-1-5 Algoritmus sekretářky

Táto úloha testovala, že či ste správne porozumeli kuchárke o základoch časovej zložitosti.[31] K získaniu plného počtu bodov nebolo nutné vymýšľať, čo robí sekretárka, stačilo popísať, čo robí zdrojový kód.

Program pre každú dvojicu tvaru $(a[i], c[j])$ vypíše nejakú hodnotu ak $a[i] = c[j]$, pričom $i \in \{0, \ldots, N-1\}$ a $j \in \{0, \ldots, M-1\}$. Takýchto dvojíc je presne NM a pre každú vykonáme najviac dve operácie (test, že či sa obe zložky rovnajú a prípadné vypísanie), teda celkový počet vykonaných operácií je určite najviac $2NM$. Z kuchárky vieme, že $2NM \in \mathcal{O}(NM)$, môžeme teda konštatovať, že program má časovú zložitosť $\mathcal{O}(NM)$.

Jednoduchým argumentom dokážeme, že to isté už efektívnejšie nespravíme. Predstavme si, že v každom prvku poľa a a c je uložená tá istá hodnota. Potom je nutné vypísať presne NM hodnôt, teda každý správny algoritmus musí mať časovú zložitosť $\mathcal{O}(NM)$.

Za určenie časovej zložitosti bolo možné získať 1 bod a navyše za korektné zdôvodnenie neexistencie efektívnejšieho algoritmu som udelil plný počet.

Peter Zeman

25-1-6 Sekání trávy

Nejdříve pár slov k došlým řešením. Asi nejčastější chybou bylo, že i když jste správně napsali, kdy trávník posekat lze a kdy ne, tak už jste nenapsali žádné odůvodnění, proč tomu tak je a proč to v jiných případech nelze. Občas se objevovala jen zdůvodnění pro případy, kdy to jde, v jiných řešeních zas pouze zdůvodnění, proč to v některých případech nejde.

Ke kompletnímu řešení se úloha musí rozdělit do nějakých případů a o všech se pak musí něco říct. Pokud u nějakého speciálního případu ukážeme, že řešení existuje, tak to ještě neznamená, že pro ostatní neexistuje, a naopak.

Další častou chybou bylo, že jste zapomínali na okrajové případy, kdy se řešení chová jinak. Například, pokud jeden z rozměrů je 1.

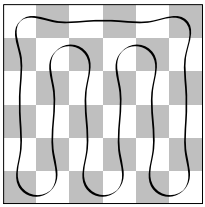
Teď už ale dost připomínek. Pojďme se raději podívat, jak to mělo být správně.

Nejdříve rozebereme případ trávníku bez kytek. Celou plochu trávníku si obarvíme jako šachovnici a všimneme si, že ať po trávníku budeme jezdit jakkoliv, tak se nám na cestě vždy po jednom budou střídat černá a bílá políčka. My chceme postupně projít všechna, každé právě jednou, a vrátit se zpět na začátek. To se nám může povést jen tehdy, pokud budeme mít stejný počet černých a bílých políček. To nastává právě, když alespoň jeden rozměr trávníku je sudý.

[31] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

Nyní jsme tedy ukázali, že pro liché rozměry trávníku řešení nemůže existovat, ale o jeho existenci jsme zatím nic neřekli. Předpokládejme tedy, že alespoň jeden rozměr trávníku je sudý, a zkusme řešení zkonstruovat. Bez újmy na obecnosti budeme předpokládat, že sudá je šířka trávníku.

Pojedeme doprava až ke kraji. Pak ve sloupcích na střídačku budeme jezdit dolů a nahoru, dokud se nevrátíme na začátek. Viz křivku na obrázku. Tento postup funguje vždy, pokud máme sudou šířku. Pokud bychom měli sudou výšku, tak trávník jen otočíme. Ukázali jsme tedy, že řešení existuje, pokud máme alespoň jeden rozměr sudý.



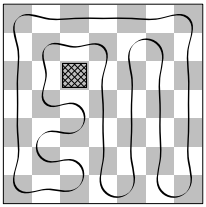
Ale pozor, to stále není všechno. Co když jeden z rozměrů trávníku bude 1? To pak naše řešení tak úplně nefunguje, protože se nemáme jak vrátit. Rozměr 1 tedy musíme vyřešit zvlášť:

- 1×1 posekat lze. To jen stojíme na místě.
- 1×2 posekat také lze. Pojedeme na sousední políčko a hned se vrátíme.
- $1 \times N, N \geq 3$ posekat nelze, protože už se nemáme kudy vrátit.

A to už je opravdu všechno, další případy pro trávník bez kytek nemáme.

Nyní k verzi trávníku s kytkami. Opět si trávník obarvíme jako šachovnici a podíváme se, ve kterých případech máme stejně černých a bílých políček (levé horní políčko vždy obarvíme na černo). Stejně jich máme, pouze pokud má trávník oba rozměry liché a kytky leží na černém políčku. V ostatních případech víme, že trávník určitě posekat nelze.

Pro liché rozměry a kytky na černém políčku zkonstruujeme obdobné řešení jako pro trávník bez kytek. Pojedeme doprava a pak na střídačku nahoru a dolů. Jediný rozdíl je, že v některé dvojici sloupců obsahující kytky budeme kličkovat, abychom obkličkovali kytky, viz obrázek.



A jak poznáme, kdy kličkovat? Bude to tehdy, kdy poprvé vjedeme do sloupce s kytkami. A díky tomu, že obě souřadnice kytek mají stejnou paritu, tak před potkáním kytek budeme mít před sebou vždy lichý počet řádků, tedy před řádkem s kytkami se kličkováním dostaneme do sloupce vlevo od kytek, a po kytkách nám zbyde sudý počet řádků, tedy na konci kličkování budeme otočeni doleva.

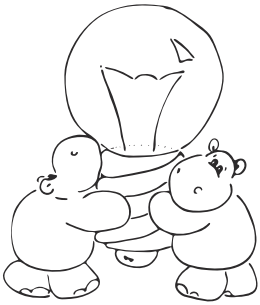
Pokud by kytky byly v prvním sloupci, tak situaci vyřešíme zrcadlově. A pokud by byly v prvním řádku, tak si plánek otočíme.

A opět to funguje až na případy, kdy je jeden z rozměrů roven jedné. Ty zas vyřešíme zvlášť:

- 1×1 nedává smysl, protože se tam s kytkami nevejdeme.
- 1×2 řešení vždy má, jsme tam jen my a kytky.
- 1×3 řešení má, pokud jsou kytky vpravo.
- $1 \times N, N \geq 4$ řešení nemá, protože se nemáme jak vrátit.

A máme vše dokázáno. Jelikož nepotřebujeme znát konkrétní dráhu, tak program není třeba.

Karel Tesař



25-1-7 GPS log

Nejdůležitějším bodem řešení této úlohy byl algoritmus pro hledání nejdelší rostoucí vybrané podposloupnosti. Většina z vašich řešení měla kvadratickou časovou složitost. My si však ukážeme lepší řešení s časovou složitostí $\mathcal{O}(N \log N)$.

K pojmům: Nejdelší rostoucí podposloupností posloupnosti $a_1, a_2, \ldots, a_n$ končící v i-tém prvku budeme rozumět posloupnost prvků $a_{r_1}, a_{r_2}, \ldots, a_{r_l}$ takovou, že $r_l = i$ a zároveň $\forall r_j < i : r_j < r_{j+1} \wedge a_{r_j} < a_{r_{j+1}}$. Její délku značíme d_i.

Nejdelší klesající podposloupnost začínající v i-tém prvku je definována obdobně s tím, že musí začínat i-tým prvkem. Její délku označíme d'_i.

1. pozorování: Jestliže chceme znát délku nejdelší klesající podposloupnosti, lze obrátit pořadí prvků v poli a hledat opět rostoucí podposloupnost.

2. pozorování: Chceme-li vědět, jaký je nejdelší možný GPS log pro daný vrchol, lze ho snadno spočítat jako $d_i + d'_i - 1$. Stačí tedy projít vstupní pole, tuto hodnotu spočítat pro každý prvek a uložit si maximum.

Zbývá nám tedy už jen říct, jak nejdelší rostoucí podposloupnost najít. Ukážeme si nejdřív kvadratické řešení, které později zlepšíme.

Jistě platí $d_1 = 1$. Pro $k > 1$ spočítáme d_k následovně. Nechť máme nejdelší rostoucí podposloupnost končící v a_k. Zakrytím a_k dostáváme opět nějakou rostoucí podposloupnost, tentokrát končící v a_x. Její délka je d_x, tedy délka posloupnosti se zakrytým a_k je $d_x + 1$.

Správné x sice neznáme, lze ho však snadno najít. Víme totiž, že musí platit $x < k$ a navíc $a_x < a_k$. Tedy platí $d_k = \max_{x<k, a_x<a_k} d_x + 1$.

Pro vylepšení algoritmu použijme další pozorování: Jestliže máme dvě stejně dlouhé rostoucí podposloupnosti, z nichž jedna má poslední prvek menší než druhá, vždy se vyplatí použít tu s menším posledním prvkem. Zvládneme-li totiž vylepšit tu „horší“, jistě to dokážeme i pro tu „lepší“.

Stačí si tedy pro každou z možných délek posloupností držet tu nejlepší, tedy končící nejmenším možným prvkem. Označme jako m_i nejmenší hodnotu, kterou může končit i-prvková rostoucí podposloupnost z dosud prošlých prvků, a na začátku inicializujme m takto: $m_i = 0$ pro $i = 1$, jinak $m_i = \infty$.

Nyní postupně projdeme vstupní pole a budeme sledovat, jak se mění hodnoty m_i po zpracování nového členu.

Všimněte si nejprve, že v každém okamžiku platí, že hodnoty m_i (které jsou různé od ∞) jsou rostoucí. Když totiž umíme vytvořit rostoucí podposloupnost délky i, která končí hodnotou m_i, tak jejích prvních $i - 1$ členů tvoří rostoucí podposloupnost délky $i - 1$, která končí členem menším než m_i. Proto nutně $m_{i-1} < m_i$.

Další pozorování: Pro právě zpracovávaný prvek x zjevně existuje právě jedno k takové, že $m_k < x \leq m_{k+1}$.

Co to znamená? V první řadě víme, že dosud „nejlepší“ podposloupnost délky $k + 1$ (a větší) končila číslem větším nebo rovným x. Žádnou takovou posloupnost nemůžeme prodloužit hodnotou x, takže hodnoty od m_{k+2} dále se měnit nebudou.

Podobně se nebudou měnit hodnoty od m_1 po m_k včetně. Všechny už jsou menší než x, tedy je zlepšit nedokážeme.

Změnilo se pouze to, že nyní umíme vybrat rostoucí posloupnost délky $k+1$, která končí hodnotou x. Nastavíme tedy $m_{k+1} = x$. Zároveň víme, že $k + 1$ je délka nejdelší rostoucí podposloupnosti končící právě zpracovaným prvkem.

Nyní si jen stačí uvědomit, že hodnoty m_i jsou seřazeny podle velikosti, takže můžeme nalézt správné číslo k binárním vyhledáváním v čase $\mathcal{O}(\log N)$. Potřebujeme zpracovat všech N prvků pole, časová složitost algoritmu tedy bude $\mathcal{O}(N \log N)$.

Určit ty prvky, které máme z posloupnosti vyškrtnout je už triviální. Stačí si pro každý prvek a_i ukládat index předposledního prvku v nejdelší rostoucí podposloupnosti končící v a_i a pole proskákat až na začátek. Pro klesající část opět analogicky.

Vzorový kód je z větší části přepisem programu Martina Raszyka. Vysvětlení lineárně-logaritmické verze algoritmu je pak z větší části převzato z autorského řešení domácího kola 57. ročníku MO-P.[32]

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-1-7.cpp`

Jan Bok

25-1-8 Sázíme v TEXu

Řešení **úkolu 1** bylo poměrně triviální:

```
\chyph
{\it Poznatky získané cílevědomě
v preadolescentním věku jsou adekvátní
poznatkům pořízeným náhodně ve věku
seniorském. (Co se v mládí naučíš,
ve stáří jako když najdeš.)}
\bye
```

Někteří z vás neřešili `\it`, to jsem taktéž neřešil, neboť ze zadání nebylo úplně jasné, jestli text máte vysázet italikou, nebo romanem (latinkou).

Někteří z vás do některých slov vložili `\-`. To jsem penalizoval ztrátou jednoho bodu, neboť se jedná o nouzové řešení pro případ, kdy se TEXu nepovede zalámat text standardními prostředky. Představte si, že byste takhle ručně měli zalámat stostránkovou knihu.

V souvislosti s tím jsem strhával nějaké body za nepřítomnost `\language\czech`. Na tomto místě se musím omluvit, daleko lepší je místo `\language\czech` zadat `\chyph` (resp. `\shyph` pro slovenčinu) – to jsem v zadání opomenul.

Jaký je mezi tím rozdíl? `\chyph` nastaví kromě českých vzorů pro lámání i několik dalších parametrů, například `\frenchspacing` – v anglických textech se píšou za interpunkčními znaménky dvojité mezery, zato v českých textech ne.

[32] `http://mo.mff.cuni.cz/p/57/reseni-1.html`

Pokud jsem někomu za toto strhnul body, prosím, aby si stěžoval, a omyl bude napraven. Nejsem si však ničeho takového vědom.

Pokud jste se pokusili tento úkol vysázet romanem bez nastavené češtiny, zjistili jste, že na konci prvního řádku máte plže (konkrétně slimáka). Tento po správném nastavení jazyka zmizel.

Úkol 2 byl taktéž snadný. Někteří konali vlastní výzkum, jiní možná chvíli hledali na internetu, každopádně snad všichni, kdo dodali řešení, jej měli správně.

Příkaz `\rm` nastavuje font na roman, latinku, základní řez, jinak funguje úplně stejně jako `\it` nebo `\bf`. Ještě doporučím pozornosti čtenáře příkaz `\tt`, který nastaví neproporcionální strojopisné písmo (typewriter).

Jednotlivé řezy písma se v některých znacích liší. Zdrojáky je buď potřeba sázet typewriterem (`\tt`), nebo si musíte ohlídat znaky jako `{}<>`, na jejichž pozicích jsou v jiných řezech umístěny jiné, potřebnější znaky.

Úkol 3 se dal vysázet čistě na základě látky ze seriálu:

```
$${a^{(b-2d_c)^3} \over \sqrt{2a^{3b_1}}}
+ {}_{{}_{{}_1^2}^{{}_3^4}}
    ^{{}_{{}_5^6}^{{}_7^8}} -
\sqrt{1 + \sqrt{1 -
  \sqrt{1 + \sqrt{1 - \sqrt{
    1 + {1 - {
      1 - {1 - x\over 1 + x}
      \over 1 + {1 - x\over 1 + x}
    } \over 1 + {
      1 - {1 - x\over 1 + x}
      \over 1 + {1 - x\over 1 + x}
    }}
  }}}
}}$$
```

Nejvíc práce zjevně zabral druhý sčítanec – nesmyslná konstrukce z horních a dolních indexů.

Prakticky identickou konstrukci někteří řešitelé vytvořili přes `\atop`, což nebylo zamýšlené řešení. Výsledek však vypadal velmi podobně:

```
$${{1 \atop 2} \atop {3 \atop 4}}
  \atop
{{5 \atop 6} \atop {7 \atop 8}}$$
```

Použité primitivum se chová prakticky stejně jako `\over`, akorát nekreslí zlomkovou čáru.

Řešení **čtvrtého úkolu** jste se zhostili různě. Mnozí dodali hezký zdroják, mám z vás radost. Jiná řešení však byla odfláknutá až hrůza. Mezi nejčastější chyby patřily pomlčky (používání - místo –), uvozovky (”” místo „“) a ruční lámání řádku v případech, kdy stačilo nastavit češtinu. Také někteří z vás trestuhodně ignorovali matematický mód. Minus jedna se sází jako `$-1$`.

Dále jste řešili několik problémů, které jsme v zadání neprobrali, neboť buď nebylo místo, nebo si na ně nikdo nevzpomněl.

Hezké $\mathcal{O}$ ve složitostních vzorcích získáte jako `${\cal O}$`. V matematickém módu se také dají přepínat fonty, přepínač `\cal` vybírá „kaligrafický“ font.

Pro stupně použijte konstrukci `$90^\circ$`: 90°. Vyzkoušejte také `$\cdot$` a `$\times$` pro různé druhy součinů (hvězdička moc hezká není) a `$\ldots$` nebo `$\cdots$` pro různé druhy trojteček.

Někomu se nemusí líbit výchozí nastavení formátu odstavce a stránky. To samozřejmě jde přenastavit:

- `\parindent` je velikost odsazení prvního řádku odstavce;
- `\parskip` je mezera mezi odstavci;
- `\baselineskip` je požadovaná vzdálenost mezi účařími jednotlivých řádků odstavce;
- pokud by po uplatnění pravidla o `\baselineskip` měly být boxy ve vertikálním boxu blíž k sobě (vzdálenost mezi okraji boxů) než `\lineskiplimit`, jsou místo toho umístěny tak, aby mezi jejich okraji byl `\lineskip`;
- předchozí dva body se uplatní na libovolné dva boxy, které se mají umístit do vertikálního boxu hned pod sebe, nejen na řádky odstavce.

Například můj oblíbený styl odstavců je takovýto:

```
\parindent 0pt
%% základní rozměr 3pt, který se smí
%% roztáhnout o 2pt a zmenšit o 1pt,
%% když je potřeba
\parskip 3pt plus 2pt minus 1pt
\baselineskip 11pt
\lineskip 1pt %% default
\lineskiplimit 0pt %% default
```

Na konkrétní odstavec se použijí právě ty rozměry, které jsou platné ve chvíli zpracování primitiva `\par`. Pokud nemá `\par` co vysázet, nevysází nic, ani prázdný řádek.

Na vertikální mezery rozumných velikostí můžete použít předdefinované `\smallskip`, `\medskip` a `\bigskip`, každý z nich je dvojnásobkem předchozího.

Pokud potřebujete, aby se každý řádek zdrojáku choval jako samostatný odstavec, použijte `\obeylines`. To se může hodit třeba na sazbu básní.

Na seznamy se dá použít `\item{odrážka}`, případně pro druhou úroveň `\itemitem`. Pokud byste potřebovali třetí a další úroveň odrážek, nejprve se zamyslete, jestli bude výsledek ještě stále přehledný, nebo jestli to nebude lepší vysázet jinak.

Ještě můžete chtít změnit velikost strany. Šířku a výšku strany určují rozměry `\pdfpagewidth` a `\pdfpageheight`. Šířku a výšku zrcadla (potištěné části papíru) nastavíte v `\hsize` a `\vsize`. Konečně `\hoffset` a `\voffset` mění levý a horní okraj – ten je roven tomuto rozměru **plus 1in**.

Tedy nastavení strany A4 s centimetrovými okraji po stranách vypadá takto:

```
\pdfpagewidth 210mm
\pdfpageheight 297mm
\hsize 190mm
\vsize 277mm
\hoffset -15.4mm
\voffset -15.4mm
```

Tolik první série. Doufám, že vás druhá série neodradí, neboť obtížnost úloh výrazně stoupla; těším se, že budu mít zase přes 30 řešení k opravování.

Jan „Moskyto“ Matějka

25-2-1 Vytíženost dopravy

Na tuto úlohu přišla spousta vašich řešení. Jedním z největších problémů některých z vás se ale ukázalo být to, jak správně rozdělit čas mezi předvýpočet a mezi odpovědi na dotazy. Zaveďme značení N pro počet zastávek (vrcholů našeho stromu) a K pro počet dotazů.

Správné rozdělení času

Zamysleme se nejdříve nad dvěma extrémy: Pokud bychom očekávali malý (konstantní) počet dotazů, bylo by asi nejlepší odpověď pro každý dotaz vyhledat samostatně (třeba jednoduchým procházením do šířky – BFS) a zabralo by nám to čas $\mathcal{O}(N)$ na dotaz a $\mathcal{O}(N)$ celkem. Druhým extrémem by bylo $K \geq N^2$. V takovém případě si můžeme předvypočítat pomocí BFS cesty z každé zastávky na každou v $\mathcal{O}(N^2)$ a odpovídat pak už jen v konstantním čase na dotaz. Tím se dostaneme na celkovou složitost $\mathcal{O}(N^2 + K)$.

My jsme se ale zabývali nejzajímavějším případem, a to když K je řádově stejně velké jako N. Pak je druhý postup příliš pomalý. Ukážeme si, jak udělat předvýpočet v čase $\mathcal{O}(N \log N)$ a odpověď na dotaz v čase $\mathcal{O}(\log N)$.

Lehčí varianta

Nejdříve se zamyslíme nad lehčí variantou s pouhou cestou. To je jako situace přímo stavěná pro maximové intervalové stromy. Intervalový strom je stromová struktura postavená nad nějakou posloupností, která je schopná vracet hodnotu (součet, maximum a podobně) v nějakém intervalu. Dělá to tak, že kořen drží hodnotu celé posloupnosti, jeho synové hodnotu levé a pravé poloviny a tak dále, až na úroveň jednotlivých prvků posloupnosti.

Každý interval jsme pak schopni poskládat z maximálně $\log N$ menších intervalů zastoupených vrcholy a dotaz na intervalový strom tedy trvá $\mathcal{O}(\log N)$, strom se dá vystavět v lineárním čase. Toť řešení jednodušší varianty. Pokud si chcete přečíst něco více, podívejte se do naší kuchařky o intervalových stromech.[33]

Složitější varianta

Nyní se pokusíme některé myšlenky intervalových stromů zobecnit, aby fungovaly nejenom na graf tvaru cesty. Strukturu ale nebudeme potřebovat aktualizovat, stačí nám ji pouze jednou vybudovat a pak nad ní pokládat dotazy. Tím se bude lišit od intervalových stromů, které umožňují rychle provádět i aktualizace.

Zakořeníme si naši grafovou síť zastávek v libovolném vrcholu a postavíme strom. Ve chvíli, kdy dostaneme dotaz na úsek A-B, můžeme ho složit (vzít maximum) z dotazů na úseky A-P a L-P, kde P je společný stromový předchůdce

[33] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`

obou vrcholů (tedy nejvýše umístěný vrchol, přes který cesta z A do B musí jít). K rychlému hledání P se vrátíme později.

Tím jsme si problém zredukovali na nalezení maxima nějaké vertikální cesty ve stromě. To bychom mohli udělat tak, že bychom jí celou prošli, ale to může trvat až lineárně dlouho vzhledem k N (například v grafu tvaru cesty). My bychom ale chtěli dosáhnout času $\mathcal{O}(\log N)$. Zaveďme si tedy v každém vrcholu *zpětné odkazy* různých úrovní. Zpětný odkaz úrovně 0 bude odkaz na otce, zpětný odkaz úrovně 1 bude odkaz na otce otce (tedy o 2 výš) a obecně zpětný odkaz úrovně m povede o 2^m vrcholů výše. Takových zpětných odkazů bude v každém vrcholu maximálně $\log n$ a každý si navíc bude pamatovat maximum na úseku, který pokrývá.

Když budeme chtít vystoupit od A k P, dokážeme tento úsek pokrýt jen pomocí těchto zpětných odkazů a použijeme jich jen $\mathcal{O}(\log N)$ (jednoduchým argumentem: když budeme skákat po největším možném zpětném odkazu, tak každým skokem zmenšíme vzdálenost alespoň o polovinu). Obdobně úsek od B k P. Pokud tedy dostaneme takovouto strukturu, dokážeme odpovědět na libovolný dotaz v $\mathcal{O}(\log N)$.

Teď se vrátíme ke slibovanému hledání P. V každém vrcholu si budeme pamatovat, v jaké je hloubce, a budeme stoupat od A a B zároveň. Nejdříve vystoupáme po zpětných odkazech z toho hlubšího do stejné hloubky (v $\mathcal{O}(\log n)$ krocích) a pak zkoušíme stoupat z obou vrcholů naráz. Vezmeme postupně všechny délky zpětných odkazů (od největších po nejmenší) a když se přes takto dlouhé zpětné odkazy ještě nedostaneme do stejného vrcholu (mohl by to totiž být až nějaký předchůdce P), vystoupáme po nich. Tímto nalezneme snadno P a současně si nerozbijeme logaritmickou délku cest od A a od B k P.

Jak si takovou strukturu rychle pořídit? To už je jednoduché. Zakořenění stromu můžeme udělat pomocí prohledávání do hloubky od libovolného vrcholu, to nám zabere lineárně kroků. V každém vrcholu pak zkonstruujeme maximálně $\log N$ zpětných odkazů a s využitím předchozích odkazů nám každý nový odkaz zabere jen konstantní čas. Vzdálenost od A k B překlenutá zpětným odkazem délky 2^m je totiž překlenutá dvěma zpětnými odkazy: od A k C délky 2^{m-1} a od C k B délky také 2^{m-1}. Když vezmeme z těchto dvou odkazů maximum, máme maximální počet cestujících i na zpětném odkazu délky 2^m.

V každém vrcholu tedy strávíme $\mathcal{O}(\log N)$ a celou strukturu zvládneme vystavět v $\mathcal{O}(N \log N)$. Celkově předvýpočet i odpověď na K dotazů trvá $\mathcal{O}((N + K) \log N)$, což je ideální.

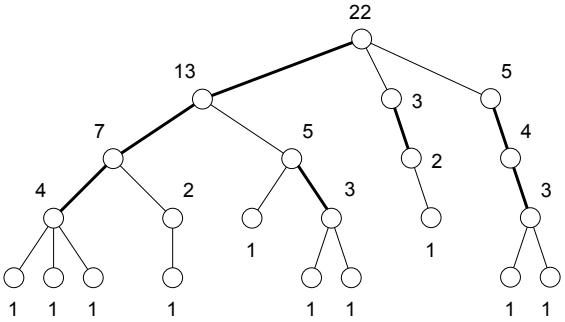
Program (C++):
`http://ksp.mff.cuni.cz/viz/25-2-1.cpp`

Jirka Setnička

Heavy-light dekompozice

Datová struktura z našeho vzorového řešení má jednu nevýhodu: pro strom na N vrcholech zabere řádově $N \log N$ buněk paměti. Načrtneme ještě jeden způsob, jak úlohu vyřešit, který si vystačí s lineární pamětí. Použijeme k tomu takzvanou *heavy-light dekompozici* stromu neboli rozklad stromu na lehké a těžké hrany.

Strom zakořeníme a pro každý vrchol v spočítáme $T(v)$, což bude počet vrcholů v podstromu, jehož kořenem je v. Za *těžké* prohlásíme ty hrany, které vedou z nějakého vrcholu v do jeho syna w, přičemž $T(w) > T(v)/2$. Všechny ostatní hrany budou *lehké*. Dobře je to vidět na následujícím obrázku (čísla udávají velikosti podstromů, těžké hrany jsou nakresleny tučně):



Povšimneme si, že z každého vrcholu může dolů vést nejvýše jedna těžká hrana. Těžké hrany proto tvoří cesty, kterým budeme říkat *těžké cesty* a jejich nejvyšším vrcholům *stopky cest*. Pokud stopka cesty není kořen, vede z ní nahoru lehká hrana, která ji napojí na nadřazenou těžkou cestu (ta ovšem může být triviální – jednovrcholová).

O lehkých hranách platí jiná zajímavá věc: kdykoliv se vydáme z kořene do listu, projdeme po nejvýše $\log_2 N$ lehkých hranách. To proto, že kdykoliv projdeme po lehké hraně, velikost podstromu, v němž se nacházíme, klesne alespoň na polovinu.

Teď popíšeme, jak pomocí naší dekompozice hledat nejbližšího společného předchůdce dvou vrcholů. Stačí si pro každou těžkou cestu zapamatovat pole všech vrcholů, které na ní leží, a naopak si pro každý vrchol zapamatovat, na jaké těžké cestě leží a kolikátý v pořadí je.

Když nám nyní někdo zadá vrcholy x a y, půjdeme z nich do kořene a podíváme se, kde se obě cesty poprvé potkaly. Do kořene přitom vyskáčeme tak, že pokaždé určíme stopku těžké cesty, na níž se nacházíme, a z té vystoupíme po jedné lehké hraně. To může nastat nejvýše $\mathcal{O}(\log N)$-krát a pokaždé nás to stojí konstantní čas.

Podobně zvládneme hledání maxim na cestách. Pro každou těžkou cestu postavíme intervalový strom, pomocí kterého budeme umět rychle nalézt maximum v libovolném úseku cesty.

Kdykoli nám pak někdo zadá cestu z x do y, rozdělíme ji na úsek z x nahoru do nejbližšího společného předchůdce a úsek z něj dolů do y. Stačí tedy umět počítat maxima pro „svislé“ cesty ve stromu. Každá svislá cesta ovšem obsahuje $\mathcal{O}(\log N)$ lehkých hran; těmi jsou spojeny úseky těžkých cest, takže úseků musí být také $\mathcal{O}(\log N)$.

Pro každou těžkou cestu se zeptáme příslušného intervalového stromu, jaké je maximum z příslušného úseku, a vypočteme maximum z těchto maxim a z ohodnocení lehkých hran spojujících těžké úseky. To dává celkem $\mathcal{O}(\log N)$ dotazů na intervalové stromy, z nichž každý trvá $\mathcal{O}(\log N)$. Dohromady $\mathcal{O}(\log^2 N)$, což je příliš.

Učiníme tedy ještě jedno pozorování: skoro všechny dotazy, které intervalovým stromům klademe, se týkají intervalů od nějakého vrcholu těžké cesty k její stopce. Jedinou výjimku tvoří nejvyšší cesta, na kterou se zeptáme. Můžeme si tedy navíc pro každou cestu předpočítat maxima úseků od stopky do ostatních vrcholů. Pak položíme $\mathcal{O}(\log N)$ dotazů trvajících $\mathcal{O}(1)$ a jeden trvající $\mathcal{O}(\log N)$. To je dohromady $\mathcal{O}(\log N)$ na celé nalezení maxima cesty z x do y.

Celá struktura nám přitom zabere $\mathcal{O}(N)$ buněk paměti a jsme ji schopni vystavět v lineárním čase.

Dodejme ještě, že trochu složitější struktury tohoto druhu jde i aktualizovat. To si ale necháme na jindy; pokud jste zvědavi, zkuste si najít něco o Sleatorových-Tarjanových stromech, známých také pod názvem Link-Cut Trees.

Martin „Medvěd“ Mareš

25-2-2 Sekání trávy podruhé

Řešení této úlohy je krátké, jednoduché, ale je třeba uznat, že i nemálo trikové. Pro úplnost zadání budeme předpokládat, že startovní políčko již je posekané. Pak má vyhrávající strategii první hráč. A jaká tedy bude jeho strategie?

Hrací plocha se skládá ze sjednocení obdélníků se sudým obsahem. Tedy alespoň jeden z rozměrů každého obdélníku je sudý. Tedy je možné celý herní plán vyskládat dominovými kostkami. Jak se to přesně udělá, si každý jednoduše rozmyslí.

Nyní k samotné strategii. Hráč jedna začíná na nějaké poloposekané dominové kostce. Tak jediné, co udělá, je, že přejde do druhé části této kostky. Nyní druhý hráč buď už nemůže nikam táhnout, nebo přejde do jiné dominové kostky. Tato kostka zatím nebyla použita, a tedy má volnou druhou půlku. Takže první

hráč zas jen přejde do druhé poloviny. Tuto strategii bude opakovat až do doby, kdy druhý hráč nebude mít kam táhnout.

Na závěr ještě poznamenejme, že obecně se této taktice říká *Párovací strategie* a funguje ve všech grafech, které mají perfektní párování (tj. vrcholy grafu lze rozdělit do dvojic, kde každá dvojice je spojena hranou).

Karel Tesař

25-2-3 Doplňování operátorů

Túto úlohu sme pôvodne zamýšľali ako jednoduchú, teda takú, že jej riešenie je priamočiare a jasné. Nevšimli sme si však značné množstvo slepých uličiek, ktoré vás zmiatli. Úlohu jsme zadávali s tým, že pôjde jednoducho riešiť v lineárnom čase, čo sa nakoniec ukázalo ako zlý predpoklad. Za to sa vám ospravedlňujeme a i kvôli tomu sme bodovali zlé riešenia miernejšie.

Drvivá väčšina riešiteľov sa úlohu pokúšala riešiť hladovo, čo ale nefungovalo. Ďalšia vec je, že skoro všetky riešenia, ktoré prišli, boli zle popísané a často sa stávalo, že si opravujúci musel toho dosť veľa domýšľať. Navyše väčšina z vás zabúdala uvádzať časovú zložitosť.

Najčastejšími protipríkladmi na hladové riešenie boli postupnosti, v ktorých sa vyskytovalo viac jednotiek vedľa seba – napríklad v postupnosti, ktorá je tvorená dvojkou, desiatimi jednotkami a opäť dvojkou, je lepšie sčítať. Na podobných protipríkladoch zlyhali všetky pokusy o lineárne hladové riešenie.

Ukážeme si ako riešiť úlohu v kvadratickom čase pomocou dynamického programovania.[34] Dynamické programovanie je veľmi užitočná programovacia technika, ktorá spočíva v rozdelení problému na nejaké menšie podproblémy, ktoré vieme vyriešiť. Z riešení pre menšie podproblémy potom poskladáme finálne riešenie. Ukážeme si to na našej úlohe.

Predstavme si, že na vstupe dostaneme n čísel, označme ich $a_1, \ldots, a_n$. Uvážme teraz každé $i \in \{2, \ldots, n-1\}$. Predpokladajme, že vieme doplniť operátory medzi $a_1, \ldots, a_i$, tak že po vyhodnotení $a_1\ a_2\ \ldots\ a_i$ dostaneme najlepší výsledok. Ako zistiť, aký najlepší výsledok môžeme dosiahnúť, keď uvažujeme $a_1\ a_2\ \ldots\ a_n$?

Zvoľme nejaké $j \in \{2, \ldots, n-1\}$. Z predchádzajúceho odstavca vieme, že sme schopní optimálne doplniť operátory medzi $a_1\ a_2\ \ldots\ a_j$, označme výsledok v_j. Jeden z možných výsledkov pre $a_1\ a_2\ \ldots\ a_n$ je $v_j + a_{j+1} \cdot \ldots \cdot a_n$, označme ho V_j. Najlepší výsledok dostaneme tak, že vezmeme maximum zo všetkých V_j, pre $j \in \{2, \ldots, n-1\}$.

Čo teda vlastne robíme? Všimnime si, že sme predpokladali, že problém vieme vyriešiť pre všetky i také, že $i < n$ a z toho sme zistili riešenie pre n.

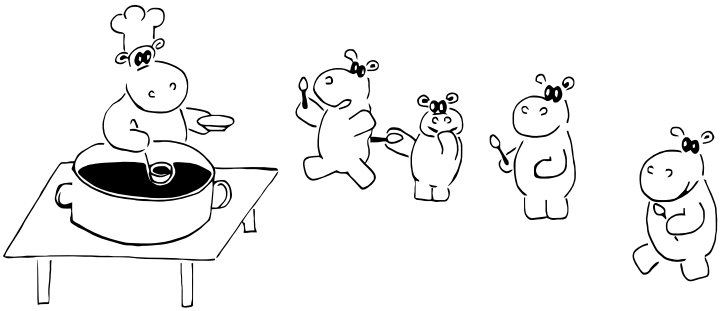
[34] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

Pre úplnosť dodajme ešte, že pre a_1 sme schopní úlohu vyriešiť triviálne, tam žiadne operátory dopĺňať netreba. To znamená, že so znalosťou optimálneho riešenia pre a_1 sme schopní aplikovaním vyššie uvedeného postupu získať optimálne riešenie pre $a_1\ a_2$, ďalej so znalosťou optimálneho riešenia pre a_1 a $a_1\ a_2$ sme schopní aplikovaním rovnakého postupu získať optimálne riešenie pre $a_1\ a_2\ a_3$ atď.

Predchádzajúci odstavec celkom jasne popisuje, ako bude algoritmus fungovať. Jeho časová zložitosť bude $\Theta(n^2)$. Je tomu tak preto, lebo potrebujeme spočítať najlepšie výsledky pre $a_1\ a_2\ \ldots\ a_i$, pre každé $i \in \{1, \ldots, n\}$. Pri počítaní každého takéhoto výsledku spravíme $\Theta(i)$ krokov. Celkový počet krokov je teda $\Theta(1) + \cdots + \Theta(n) = \Theta(n^2)$. Pamäťová zložitosť je $\Theta(n)$.

Program (Python):
`http://ksp.mff.cuni.cz/viz/25-2-3.py`

Peter Zeman



25-2-4 Organizace vykládky

V této praktické úloze byly vstupy rozděleny do čtyř sad podle očekávané efektivity řešení.

Hledaný součet vzdáleností od daného bodu ke všem ostatním v maximové metrice nazveme cenou vykládky pro daný bod. Nejjednodušším řešením úlohy je přímý výpočet ceny vykládky pro každý bod tak, že projdeme všechny ostatní body a sečteme vzdálenosti podle vzorečku. Takové řešení má časovou složitost $\mathcal{O}(N^2)$ a stačilo pro vyřešení prvních dvou sad.

Ve druhé sadě se hodnoty nevlezly do 32-bitových proměnných, museli jste tedy použít 64-bitové celočíselné typy. Co jsem do úlohy nevkládal, ale doporučuji k promyšlení (a nejlépe i naprogramování), je případ, kdy by se čísla nevlezla ani do 64-bitových proměnných a váš jazyk by nepodporoval celočíselné typy s dynamickou délkou. A jak byste postupovali, pokud by čísla nebylo potřeba

pouze sčítat a porovnávat, ale provádět i další operace jako například násobení a dělení?

Ve třetí sadě již byl počet zadaných bodů vysoký ($N \leq 100000$), ale počet navzájem různých bodů byl stále nízký ($K \leq 1000$). Taková situace se dala vyřešit seskupením shodných bodů do jediného bodu, u kterého byla udána jeho násobnost. Lze to provést třeba setříděním bodů (nejpřirozenější způsob je asi lexikografický – nejprve podle souřadnice x, v případě shody podle souřadnice y).

Setřídění vede k tomu, že se shodné body umístí v poli na souvislém úseku. Pak lze pole projít a pro každý bod zkontrolovat, zda je shodný s předchozím prvkem pole. V případě shody je zvýšena násobnost naposledy přidaného bodu, v opačném případě je přidán nový bod. Protože má (rozumné) třídění časovou složitost $\mathcal{O}(N \log N)$, je výsledná časová složitost $\mathcal{O}(N \log N + K^2)$. (Použitím vyhledávacích stromů lze docílit $\mathcal{O}(N \log K + K^2)$, což ale není příliš atraktivní vylepšení.)

Efektivní řešení

Problém dosavadního přístupu tkví v tom, že kvůli nepříjemné formuli pro vzdálenost mezi dvěma body nedokážeme cenu vykládky počítat nějak hromadně. Možností, se kterou jsem původně počítal, je zametat body podél y-ové souřadnice a vhodně si v intervalových stromech udržovat body tak, aby se daly efektivně pro zpracovávaný bod spočítat 4 součty – součty souřadnic všech bodů, pro které se bude: přičítat a odečítat x-ová souřadnice a přičítat a odečítat y-ová.

Asymptoticky stejně efektivní, ale jednodušší je použít trik převzatý z řešení Ondry Hübsche. Trik spočívá ve využití vztahu

$$2\max(|a|, |b|) = |a - b| + |a + b|,$$

ověřte si jej například rozborem případů. Vzdálenost dvou bodů $d = \max(|x_1 - x_2|, |y_1 - y_2|)$ pak můžeme zapsat jako

$$2d = |(x_1 - x_2) - (y_1 - y_2)| + |(x_1 - x_2) + (y_1 - y_2)|.$$

Budeme pracovat s novými souřadnicemi u a v, které vzniknou jako:

$$u = x - y \qquad v = x + y$$

Vzdálenost dvou bodů nám pak bude vycházet jako

$$2d = |u_1 - u_2| + |v_1 - v_2|.$$

Celou dobu budeme počítat dvojnásobné vzdálenosti a pouze na konci výsledek vydělíme dvěma.

Jaká je výhoda tohoto převodu? Zbavili jsme se nepříjemné operace maxima. Součet už dokážeme počítat efektivně, provedeme to ve dvou krocích –

oddělíme $|u_1 - u_2|$ a $|v_1 - v_2|$. V prvním kroku setřídíme body vzestupně podle souřadnice u a spočítáme si prefixové součty pro toto setříděné pole. Pro aktuálně zpracovávaný prvek p pole je pak $|u_p - u_i| = u_p - u_i$ pro všechny prvky i před ním a $|u_p - u_i| = u_i - u_p$ pro prvky za ním.

S prefixovými součty dokážeme započítat všechny body do ceny vykládky pro bod p v konstantním čase. (Pole P prefixových součtů je takové pole, které na i-té pozici obsahuje součet prvních i prvků. Dokážeme jej předpočítat v lineárním čase pomocí vztahu $P[i] = P[i-1] + Q[i]$, kde Q je původní pole. Součet prvků na pozicích k až l je pak $P[l] - P[k-1]$.)

Analogicky postupujeme pro souřadnici v. Nyní máme pro každý bod spočtenou cenu vykládky a stačí vybrat nejmenší z nich. Jedinou složitější operací je třídění, ostatní operace jsou pouhé lineární průchody. Časová složitost řešení je tak $\mathcal{O}(N \log N)$, paměťová $\mathcal{O}(N)$.

Program (C++) – N kvadratický:
`http://ksp.mff.cuni.cz/viz/25-2-4-Nkvadr.cpp`

Program (C++) – K kvadratický:
`http://ksp.mff.cuni.cz/viz/25-2-4-Kkvadr.cpp`

Program (C++) – $N \log N$:
`http://ksp.mff.cuni.cz/viz/25-2-4-NlogN.cpp`

Lukáš Folwarczný

25-2-5 Sbírání papírů

Než se vrhneme na řešení úlohy, učiňme několik stěžejních pozorování. Ta nám už řešení úlohy dají takřka zadarmo.

Novinářka se nesmí vracet dolů. Tím pádem před přechodem na další řádek (nahoru) musí vybrat všechny papíry na řádku.

Zároveň nemá smysl uvažovat jiné papíry než ten nejvíce vlevo a vpravo. Všechny ostatní papíry totiž novinářka sebere automaticky při pohybu mezi nimi.

Dále je důležité uvědomit si, že po sebrání posledního papíru nemá smysl se na tomto řádku dále pohybovat. Stejné kroky totiž lze provést o řádek výše s výsledkem přinejhorším stejným (na aktuálním řádku už žádný další papír nesebereme).

Musíme ještě vyřešit, jak se na řádku pohybovat. Označme si index papíru položeného nejvíce nalevo l a index toho nejvíce napravo r. Index políčka, na které vstupujeme při přechodu na řádek, označme c. Nyní nastávají tři možnosti.

$l \geq c$ Nemá smysl se pohybovat jinak než pořád doleva. $r \leq c$ Nemá smysl se pohybovat jinak než pořád doprava. $l < c < r$ Můžeme jít nejdříve k l a následně k r. Také je možné jít nejdříve k r a poté k l. Obě varianty mohou dát jinou délku cesty.

V tuto chvíli mnoho z řešitelů sáhlo po hladovém algoritmu. Tedy vždy se podívat, zda je výhodnější z c jít nejdříve do l a až poté do r, nebo opačně. Následně lepší z výsledků vzít jako součást řešení a pokračovat stejným způsobem na řádku výše. Tento postup je však chybný. Zkuste si ho například odsimulovat na následujícím vstupu:

```
3 8
Řádek 3: 0 1 0 0 0 0 0 0
Řádek 2: 0 1 0 0 0 0 0 1
Řádek 1: 0 0 1 0 0 0 0 0
```

Z toho plyne, že je třeba počítat s oběma variantami průchodu řádkem a obě si uložit. Všech možných cest je tak sice exponenciálně mnoho, lze ale využít přístupu z dynamického programování a ukládat si mezivýsledky.

Buďme v nějakém řádku i. V řádku $i-1$ jsme dostali dvě optimální řešení. Jedno pro l_{i-1} a jedno pro r_{i-1}. Pro spočítání řešení v l_i tedy vyzkoušíme oba z výsledků pro předchozí řádek a lepší z nich si uložíme. Totéž provedeme pro r_i, čímž dostaneme opět dvě možné varianty. V posledním řádku pak z těchto dvou variant vybereme tu s kratší délkou cesty.

Speciálním případem je první řádek, a to v tom, že musíme vždy dojít do r_1 a optimální řešení je zde tak pouze jedno (o délce r_1). V programu toto snadno ošetříme.

Že je řešení správné si lze snadno rozmyslet přes matematickou indukci. Řešení pro první řádek je indukčním předpokladem. Řešení pro i-tý řádek pak indukčním krokem.

Ještě však nekončíme. Jelikož chceme také zpětně rekonstruovat cestu, musíme si navíc ukládat pole předchůdců a směry pohybu. Pak lze již cestu rekonstruovat. Směr pohybu se navíc bude měnit pro řádek maximálně dvakrát, tedy paměťová složitost uložení cesty je $\mathcal{O}(N)$. Taková je i paměťová složitost celého algoritmu, jelikož není třeba ukládat si celou matici. Pozorní čtenáři si jistě dokážou rozmyslet proč.

Ke zjištění indexů l a r pro každý řádek stačí jednou matici v $\mathcal{O}(NM)$ projít. Následná dynamika nám zabere $\mathcal{O}(N)$ kroků, tedy celková časová složitost je lineární – $\mathcal{O}(NM)$.

Zdrojový kód je z větší části přepisem programu Rastislava Rabatina. Děkujeme.

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-2-5.cpp`

Jan Bok

25-2-6 Optimalizace v redakci

Naším úkolem je pro každou hranu zadaného ohodnoceného grafu určit, zda se vyskytuje ve všech, v pouze některých, nebo v žádných minimálních kostrách. K řešení využijeme upravený Kruskalův algoritmus. Z kuchařky[35] budeme potřebovat především poznatek o jeho správnosti a také poznatek o tom, že v případě více hran stejné váhy může být pořadí zpracování hran libovolné (algoritmus může vydat různé kostry, ale všechny budou minimální).

Popis algoritmu

Algoritmus si bude, stejně jako ten původní, budovat les T – část výsledné kostry. Hrany stejné váhy w_k budeme zpracovávat najednou. Hranu, jejíž oba vrcholy leží v jedné komponentě, rovnou označíme za hranu, která se v žádné minimální kostře nevyskytuje, a dále s ní nepracujeme.

Pro ostatní hrany rozhodneme, zda se vyskytují ve všech, nebo jen některých minimálních kostrách. Ve skutečnosti zjistíme, zda Kruskalův algoritmus hranu (v závislosti na pořadí zpracování) přidá vždy, nebo jen někdy. Později dokážeme, že je obojí ekvivalentní.

Uvažme pro tento účel graf G, který vznikne tak, že do T vložíme všechny hrany váhy w_k. Pokud se odebráním hrany e (váhy w_k) z grafu G sníží počet komponent, přidal by algoritmus hranu ve všech případech, neb nemá jinou hranu téže váhy, jíž by spojil příslušné komponenty. Pokud se naopak počet komponent nezmění, existuje další hrana, kterou lze použít místo e.

Hrana, po jejímž odebrání vzroste počet komponent grafu, se nazývá *most*. Algoritmus na hledání mostů je detailně popsán v kuchařce o grafech,[36] zde si jej popíšeme pouze stručně. Upravíme algoritmus průchodu do hloubky (DFS) tak, aby u vrcholů počítal tzv. hladinu. Graf si zakořeníme, pro lepší představu umístíme kořen „dolů“ a přidělíme mu hladinu 0. Hladina pak ukazuje to, jak vysoko při průchodu vystoupáme, tedy počet hran na cestě od kořene při průchodu do hloubky.

Pro každý vrchol se určí nejnižší hladina, do které se lze z něj dostat při průchodu do hloubky (hrana, po které jsme do něj přišli je přirozeně zapovězena). Každý vrchol si tuto hodnotu spočte rekurzivně jako minimum z hladin vrcholů, do kterých se z něj lze dostat.

Nyní, když se v DFS stromu vracíme z vrcholu w zpátky do jeho otce v, nastávají dvě možnosti: z vrcholu w se lze dostat pouze do hladin vyšších než v, pak je hrana vw most, neb po odebrání se už z w do nižších hladin nijak nedostaneme. Pokud se lze dostat alespoň do hladiny vrcholu v, pak se o most nejedná,

[35] `http://ksp.mff.cuni.cz/viz/kucharky/minimalni-kostra`
[36] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

protože i po odebrání hrany vw bude existovat cesta z v do w a graf tak bude souvislý.

Časová složitost průchodu do hloubky je $\mathcal{O}(N+M)$, kde N je počet vrcholů a M počet hran. Počet skupin hran různé váhy označme jako ℓ a počty hran v těchto skupinách jako m_1 až m_ℓ. V jednom průchodu procházíme pro každou skupinu graf, ve kterém je nejvýše $N-1+m_i$ hran (kostra grafu na N vrcholech má $N-1$ hran). Časová složitost je tak $\mathcal{O}(\ell \cdot N + \sum_i m_i)$, což je v nejhorším případě $\mathcal{O}(MN)$.

Pro vylepšení si uvědomíme, že vrcholy a hrany uvnitř už existujících komponent nás ve skutečnosti vůbec nezajímají. Zajímá nás pouze to, jak propojují aktuálně zpracovávané hrany. Graf proto budeme konstruovat tak, že jeho vrcholy budou tyto komponenty. Navíc komponenty, do kterých nevede žádná z aktuálně zpracovávaných hran, do grafu nezahrneme.

Takový graf už nemusí být klasickým grafem, ale mohou v něm existovat i násobné hrany (více hran mezi dvěma vrcholy, tzv. multihrany). To nevadí, uvědomíme si, že multihrany nemohou být mosty a nic jiného není ovlivněno. Takový graf bude mít m_i hran a nejvýše $2m_i$ vrcholů, kde m_i je počet hran i-té váhy.

Celková složitost hledání mostů bude $\mathcal{O}(m_1 + m_2 + \ldots + m_\ell) = \mathcal{O}(M)$. V Kruskalově algoritmu hrany na začátku setřídíme nějakým efektivním algoritmem, např. QuickSortem, a pak používáme strukturu Disjoint-Find-Union. To nám dává časovou složitost $\mathcal{O}(M \log M)$, což je i složitost celého algoritmu. Paměťová složitost je $\mathcal{O}(M)$.

Důkaz správnosti

Zatím jsme pro hrany nějaké váhy w_i rozhodli pouze to, zda se objeví v nějaké kostře za předpokladu, že už je určena částečná kostra T, která vznikla během Kruskalova algoritmu na všech hranách menší váhy. Dokážeme, že hrana, která spojuje 2 vrcholy stejné komponenty v průběhu našeho algoritmu, se neobjeví v žádné minimální kostře.

Předpokládejme pro spor, že existuje minimální kostra, ve které se vyskytuje tato hrana e. Odeberme tuto hranu – vzniknou dvě komponenty. Uvažme všechny hrany původního grafu, které vedou mezi komponentami (z vrcholu jedné komponenty do vrcholu druhé komponenty). V průběhu našeho algoritmu byly tyto komponenty spojeny lehčí hranou než e, tedy mezi vybranými hranami je tato hrana. Když ji přidáme místo e do kostry, vznikne nová kostra s menší váhou – to je spor s tím, že se jednalo o minimální kostru.

O ostatních hranách víme, že se vyskytují v alespoň jedné minimální kostře. Předpokládejme, že existuje minimální kostra, ve které se nevyskytuje hrana $e = uv$, o které jsme prohlásili, že se vyskytuje v každé kostře. Uvažme cyklus, který vznikne přidáním hrany e do této kostry.

Pokud mají všechny hrany na tomto cyklu menší váhu, pak bychom hranu e do kostry vůbec nepřidávali, místo toho bychom vrcholy u a v spojili těmito levnějšími hranami. Pokud se na cyklu vyskytuje ostře větší hrana, je možno ji za e vyměnit, tím ale vznikne lehčí kostra, což je spor.

Zbývá případ, kdy se na cyklu vyskytuje alespoň jedna hrana $f = wx$ stejné váhy. Pak se ale dvojice (u, w), (v, x) nebo (u, x), (v, w) dají spojit hranou váhy nejvýše stejné jako e a v algoritmu bychom vyhodnotili, že ani e ani f nejsou mosty, to je opět spor.

Pro zbývající hrany jsme přímo ukázali, jak sestrojit minimální kostru, kde se vyskytují, i minimální kostru, kde se nevyskytují. Důkaz je tímto hotov.

Závěrem si dovolím malou poznámku. Jak asi vidíte, tento důkaz je dosti nepěkný a nepřehledný. Kolem koster existuje zajímavá teorie, pomocí které lze tvrzení podobná tomu našemu dokazovat daleko příjemněji. Doporučuji nahlédnout do učebního textu Martina Mareše Krajinou grafových algoritmů.[37] Lze tam nalézt například i to, že pro zavedení minimální kostry vlastně vůbec nemusíme umět váhy hran sčítat – stačí je umět porovnávat. (Což lze možná i odtušit z toho, že Kruskalův algoritmus sčítání nevyužívá.)

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-2-6.cpp`

Lukáš Folwarczný

Teorie minimálních koster

Z té zmíněné teorie minimálních koster si ostatně můžeme malý kousek ukázat. Uvažme nějakou minimální kostru T a hranu $e = uv$, která v této kostře neleží. Víme, že vrcholy u a v jsou v kostře T spojené nějakou cestou $T[u, v]$. Označme f nejtěžší hranu této cesty.

Rozlišíme tři případy:

- f je těžší než e – tento případ nemůže nastat. Tehdy bychom totiž mohli z kostry T odebrat hranu f a vzniklé dvě komponenty spojit přidáním hrany e. Tím bychom získali lehčí kostru, než byla minimální kostra T.
- f je stejně těžká jako e – pak můžeme použít tentýž trik a získat jinou kostru stejně těžkou jako T (tedy též minimální), která obsahuje hranu e. Tím pádem hrana e v některých minimálních kostrách leží a v jiných ne.
- f je lehčí než e – dokážeme, že tehdy se e nemůže vyskytovat v žádné minimální kostře. K tomu se nám bude hodit následující lemma (použijeme ho na cyklus tvořený cestou $T[u, v]$ a hranou e).

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

Cyklové lemma: Nechť C je cyklus v grafu a $e = uv$ jeho hrana, která je těžší než všechny ostatní hrany cyklu C. Potom se e nevyskytuje v žádné minimální kostře.

Důkaz: Pro spor předpokládejme, že existuje nějaká minimální kostra K, která obsahuje hranu e. Odebereme-li z K tuto hranu, kostra se rozpadne na nějaké dva stromy K_u a K_v (označíme je tak, aby $u \in K_u$ a $v \in K_v$). Budeme obcházet cyklus C: začneme ve vrcholu u, půjdeme vzdálenější cestou k vrcholu v (tedy ne po hraně e) a budeme sledovat, ve kterém stromu se zrovna nacházíme. Na začátku to je strom K_u, na konci K_v, takže někde cestou musíme potkat nějakou hranu h, jejíž jeden konec leží v K_u a druhý v K_v. Tato hrana se ovšem nevyskytuje v kostře K a je lehčí než hrana e (protože hrana e byla nejtěžší na cyklu). Proto nahrazením e za h získáme kostru lehčí než K, což je spor s minimalitou K.

Naše tři pravidla nám tedy říkají, jak pro každou hranu, která neleží ve zvolené minimální kostře T, rozhodnout, zda leží v nějaké / všech / žádné minimální kostře. Stačí umět hledat nejtěžší hrany na cestách v T, na což se dá elegantně použít datová struktura z druhé úlohy této série.

Zbývá dořešit, jak je to s hranami, které leží v T, ale to už si zkuste rozmyslet sami. Opět pomůže Cyklové lemma.

Martin „Medvěd“ Mareš

25-2-7 Zaléváme dokument

Nepřišlo sice tolik vašich řešení jako v první sérii, ale stále se jednalo o úlohu s největším počtem odevzdání. Vypadá to, že vás TEX zaujal a to je dobře.

Úkol 1

Řešení tohoto úkolu jste se zhostili velmi úspěšně a vynalézavě. Podívejme se, jakým způsobem se dal řešit. Nejprve bylo potřeba nadefinovat políčka.

```
\def\bile#1{\hskip #1}
\def\cerne#1{\vrule height #1 width #1}
```

Objevily se i jiné, stejně dobré definice bílého pole:

```
\def\bile#1{\hbox to #1{\hfil}}
\def\bile#1{\vrule height 0cm width #1}
```

Pak je potřeba políčka nějak poskládat do šachovnice. Ondra Hlavatý přišel s nápaditým a čistým řešením – definovat oblasti 2×2.

```
\def\ctyrka#1{\vbox{%
  \hbox{\bile{#1}\cerne{#1}}%
  \hbox{\cerne{#1}\bile{#1}}%
}}
```

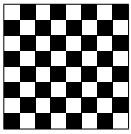
Tyto oblasti pak naskládáme do šachovnice a orámujeme.

```
\def\sachovnice#1{\vbox{\offinterlineskip%
  \hrule\hbox{%
    \vrule\vbox{%
      \hbox{\ctyrka{#1}\ctyrka{#1}%
            \ctyrka{#1}\ctyrka{#1}}%
      \hbox{\ctyrka{#1}\ctyrka{#1}%
            \ctyrka{#1}\ctyrka{#1}}%
      \hbox{\ctyrka{#1}\ctyrka{#1}%
            \ctyrka{#1}\ctyrka{#1}}%
      \hbox{\ctyrka{#1}\ctyrka{#1}%
            \ctyrka{#1}\ctyrka{#1}}%
    }\vrule
  }\hrule
}}
```

Šachovnici vykreslíme zavoláním `\sachovnice{20mm}`.

Vypnutí mezířádkové mezery (`\offinterlineskip`) jste si mohli najít na fóru, případně jste mohli nastavit rozměry `\baselineskip` a `\lineskip` na nulu.

Typickou chybou bylo vynechání vypnutí mezířádkových mezer, což vytvořilo ošklivé bílé pruhy mezi řádky. To jsem penalizoval obvykle jedním bodem. Drobné penalizace jste se také mohli dočkat za nečitelný kód.



Úkol 2

Druhý úkol bylo prosté cvičení na vyloženou látku. Někteří jej hrubě odflákli, objevilo se nezarovnání počtu bodů na pravou stranu nebo ošklivě malá mezera pod textem.

Takto vypadá vzorová implementace používaná v KSP (jen máme okolo přidané ještě nějaké mezery, aby nám nadpisy úloh lépe sedly do sazby):

```
\def\thrule{\hrule height 0.8pt depth 0pt}
\def\dblrule{\thrule\nobreak\vskip 1pt\thrule}
\def\taskheading#1#2#3{\vtop{%
  \dblrule\line{%
    \strut\bf #1 \enspace #2 \hfil #3}
  \dblrule}%
}
```

Makro `\thrule` definuje čáru, `\dblrule` definuje dvojčáru. Makro `\enspace` je definováno v Plainu jako mezera velká přesně `0.5em`.

Asi dva z vás použili i `\strut`, což je zkratka definovaná v Plainu za podlý trik (leč naprosto běžný a standardní): `\vrule width 0pt height 8.5pt depth 3.5pt\relax`

Účel použití této konstrukce naleznete ve třetí sérii. Ti, kdo na to přišli sami, u mě mají malé bezvýznamné plus.

Úkol 3

Toto byl nejtěžší úkol. Definice vlastního prostředí typu verbatim dala mnohým zabrat. Objevilo se několik variant, obvykle jste si zvolili nějaký znak nebo fixní sekvenci, která prostředí verbatim ukončí. Zde uvádíme vzorovou implementaci, ve které si můžete zvolit ukončovací řetězec sami.

Příkaz `\verbatim` je vstupní rozhraní celého prostředí. Otevře se skupina, přestaví se kategorie speciálních znaků, změní se práce s mezerami a předá se řízení do dalšího makra.

```
\def\verbatim{%
  \begingroup
  \verbcats
  \verbspaces
  \verbwork
}
```

Přestavení speciálních tisknutelných znaků se provede s výjimkou složených závorek, které ještě budeme potřebovat, aby nám orámovaly ukončovací řetězec.

Všimněte si speciální sekvence `^^I` a `^^M`. Když TEX potká dva stejné znaky kategorie 7 za sebou, spolkne je a následujícímu znaku přehodí šestý bit. Operace se provádí *před tokenizací* (takže `\^^I` je token (TAB, 0), tedy řídící sekvence se jménem TAB).

Takže `^^I` je znak s kódem 9, tedy tabulátor; `^^@` je znak s kódem 0; `^^.` je znak `n`; `^^?` je znak s kódem 127, zvaný též DEL, jediný znak, který je běžně kategorie 15. Tento zápis se hodí pro přehlednost, aby se v kódu jen tak nepoflakovaly netisknutelné znaky.

Ještě pro úplnost, pokud se za `^^` objeví dvě malé hexadecimální číslice (0 až 9, a až f), nahradí se celá čtveřice za znak s uvedeným hexadecimálním kódem.

```
\def\verbcats{%
  \catcode`\\=12
  \catcode`\$=12
  \catcode`\&=12
  \catcode`\#=12
  \catcode`\^=12
  \catcode`\_=12
  \catcode`\%=12
  \catcode`\~=12
  \catcode`\ =13
  \catcode`\^^I=13
  \catcode`\^^M=13
}
```

Nyní nastavíme chování verbatimu na bílých znacích.

Mezera, tabulátor a konec řádku se stanou aktivními znaky s významem `\verbspc`, `\verbtab` a `\verbcr`. Konec řádku zajistí, že se vstoupí do odstavcového módu a vynutí se vysázení odstavce. Odstavec musí obsahovat nějaký materiál, jinak se nevysází, proto ten `\hskip`.

Mezera se vysází jako explicitní mezera. Tabulátor vysázíme jako čtyři mezery. Pokud odkomentujete šestý řádek, budou mezery vysázeny viditelně. Mezery na koncích řádků se ořezávají ještě před tokenizací, takže je vysázet neumíme.

```
\catcode`\ =13\catcode`\^^I=13\catcode`\^^M=13
\def\verbspaces{\let \verbspc\let
\verbcr\let^^I\verbtab}
\catcode`\ =10\catcode`\^^I=10\catcode`\^^M=5
\def\verbspc{\ }
%\chardef\verbspc 32
\def\verbtab{\verbspc\verbspc\verbspc\verbspc}
\def\verbcr{\noindent\hskip 0pt\par}
```

Nakonec se přečte ukončovací řetězec (který může obsahovat libovolné znaky, jen musí být dobře uzávorkovaný, co se týče složených závorek), nastaví se i kategorie složených závorek na 12, zruší odsazení prvního řádku odstavce a provede finální magie.

```
\def\verbwork#1{%
  \catcode`\{=12%
  \catcode`\}=12%
  \parindent 0pt%
  \def\verbdo^^M##1#1{\tt##1\endgroup}%
  \verbdo
}
```

Makro `\verbatim` se volá takto:

```
\verbatim{EOV}
Nějaký zdrojový kód
   odsazený mezerami
        nebo tabulátorem

s prázdnými řádky
a obsahující různé speciální znaky
jako jsou { a } nebo %
EOV
```

Nyní vysvětlíme magii okolo `\verbdo`. Parametr `{EOV}` se přečte až při volání makra `\verbwork`. To pak definuje `\verbdo` vlastně takto:

```
\def\verbdo^^M#1EOV{\tt#1\endgroup}
```

Pak se makro zavolá, spolkne konec řádku za `{EOV}` a zbytek až do `EOV` vysází. Pak uzavře skupinu, čímž všechny zběsilé změny kategorií zruší a můžeme zase sázet klasicky dál. Místo `EOV` můžeme použít libovolný jiný řetězec, který bude verbatim ukončovat.

Za rozumně funkční řešení jsem dával plný počet bodů. Za vážnější prohřešky jsem pak něco strhával.

Balíky maker

Čím více budete používat TEX, tím více budete mít pocit, že si na začátek souboru kopírujete děsnou spoustu věcí. TEX umí vkládat externí soubory primitivem `\input`, za které uvedete jméno souboru. Můžete si tedy například vytvořit svůj soubor se spoustou maker, který si pak vložíte do každého sázeného textu. Například všechny letáky KSP začínají příkazem `\input kspmac3.2.tex`, tedy vložením souboru s makry KSP, verze 3.2.

Na spoustu různých úkolů pak existují specializované balíky, které si uživatelé můžou vyměňovat přes CTAN.[38] Na adrese `http://www.ctan.org/` tedy najdete několik tisíc různých balíků všeho druhu.

A to je pro dnešek vše. Děkuji vám všem za hezká řešení a těším se na příští sérii.

Jan „Moskyto“ Matějka

[38] Comprehensive TEX Archive Network

25-3-1 Kontrola docházky

S pomocí kuchařky nebyla tato úloha příliš obtížná a je škoda, že jsme nedostali o něco více řešení, neboť všechna byla štědře oceněna. Není těžké poznat, že škrtnutí jedné dvojice je jen drobná úprava klasického příkladu na Čínskou zbytkovou větu.

Naše řešení bude z obvyklého postupu na řešení takového příkladu taky vycházet. Zapomeňme tedy prozatím na škrtání jedné dvojice a stručně si připomeňme, co nám o Čínské zbytkové větě říká kuchařka o teorii čísel.[39] Budeme předpokládat, že s čísly umíme aritmetické operace v konstantním čase a paměti. Protože kvůli stručnosti přeskakujeme některá zdůvodnění a mezikroky, doporučujeme mít při čtení tohoto vzorového řešení po ruce kuchařku.

Bez škrtání dvojice

Když se podle zadání policisté seřadí do řad po M_1 lidech, zbyde jich K_1, když se seřadí po M_2, zůstane jich K_2, a obdobně až do M_N, K_N.

Pro každé i mezi 1 a N vypočítáme „magické“ Q_i, pro které platí $Q_i \equiv 1 \pmod{M_i}$, a pro každé j různé od i naopak $Q_i \equiv 0 \pmod{M_j}$. Tyto „magické koeficienty“ později použijeme ke zjištění počtu policistů na stanici (bez škrtání dvojice):

$$V \equiv \sum_{i=1}^{N} Q_i \cdot K_i \pmod{M_1 \cdot \ldots \cdot M_N}$$

Označíme $\mathrm{nsn}(M_1, \ldots, M_{i-1}, M_{i+1}, \ldots, M_N)$ jako S_i. Protože jsme zvolili M_i v zadání jako navzájem nesoudělná, je S_i rovné $M_1 \cdot \ldots \cdot M_{i-1} \cdot M_{i+1} \cdot \ldots \cdot M_N$.

Snadno si všimneme, že Q_i musí být nějaký násobek S_i. Zbytek po dělení S_i číslem M_i si označíme jako r_i. Pomocí rozšířeného Euklidova algoritmu určíme $r_i^{-1} \pmod{M_i}$. Naše hledané Q_i je pak $S_i \cdot r_i^{-1}$. Tolik se našlo v kuchařce.

Algoritmus psaný podle definice ale není moc rychlý: každé S_i by počítal násobením $N-1$ jednotlivých M_j, na čemž by strávil čas $\mathcal{O}(N^2)$. Rozumnější je předpočítat si pro každé i násobek $\mathrm{A}(i) = M_1 \cdot \ldots \cdot M_i$ a násobek $\mathrm{B}(i) = M_i \cdot \ldots \cdot M_N$. S_i pak dokážeme spočítat snadněji jako $\mathrm{A}(i-1) \cdot \mathrm{B}(i+1)$. S lineárním časem a pamětí na předpočítání tedy najdeme všechna S_i v lineárním čase.

Lineární paměťová složitost zde příliš nevadí, protože samotné zadání je také lineárně velké. Šlo by si taky spočítat předem součin všech M_j, a S_i spočítat jako podíl tohoto součinu a M_i (dokonce v konstantní paměti).

Když dokážeme S_i (například popsanými způsoby) spočítat rychle, má algoritmus na řešení úloh na Čínskou zbytkovou větu časovou složitost $\mathcal{O}(N \log N)$.

[39] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

Se škrtáním dvojice

Jak modifikujeme tento algoritmus tak, aby uměl oprošťovat veřejné činitele od pracovních povinností? Jako první se nabízí zkratka vyzkoušet všechny možnosti seškrtání, a pro každou z nich znova spočítat výsledek podle popsaného postupu. To by trvalo čas $N \cdot \mathcal{O}(N \log N) = \mathcal{O}(N^2 \log N)$.

Pěknější řešení vychází ze znalosti čísel S_i. Nejdříve si pomocí ukázaného postupu spočítáme, kolik policistů by mělo být nastoupeno bez podvádění. Výsledek si označme V.

Platí, že když škrtneme (M_i, K_i), bude muset na stanici zůstat $V \bmod S_i$ policistů. Vskutku: když od V odečteme S_i, snížíme o 1 jeho zbytek po dělení M_i, a ostatní zbytky zůstanou stejné. Nejmenší číslo, které dostaneme opakováním takového kroku, je právě $V \bmod S_i$.

Můžeme tedy předpočítat v $\mathcal{O}(N)$ všechna S_i, pak v čase $\mathcal{O}(N \log N)$ spočítat nepodvádějící řešení V, a nakonec vyzkoušet, pro které i je $V \bmod S_i$ nejmenší. Nalezené i můžeme s čistým svědomím prohlásit za správný výsledek. Protože nejnáročnější krok algoritmu bude řešení kongruencí s časovou složitostí $\mathcal{O}(N \log N)$, poběží celý algoritmus v čase $\mathcal{O}(N \log N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-3-1.c`

Michal Pokorný

25-3-2 Zasedání u kulatého stolu

Nejdříve si všimneme, že existenci mnohoúhelníka u stolu s N místy stačí ověřovat jen pro všechny k-úhelníky, kde $k \geq 3$ a dělí N. Podívejme se tedy, jak ověříme existenci k-úhelníka pro nějaké konkrétní k.

Mnohoúhelník určitě bude obsahovat jedno z prvních N/k míst. Navíc každé z těchto míst nám už jednoznačně určuje celý k-úhelník (ten bude tvořen vybraným vrcholem a každým (N/k)-tým dalším). Pokud ve všech vrcholech některého z těchto k-úhelníků budou jedničky, tak máme vyhráno.

Jeden k-úhelník ověříme v čase $\mathcal{O}(k)$, protože se díváme jen do k vrcholů a pro dané k jich ověřujeme N/k, dostaneme tedy $\mathcal{O}(k \cdot N/k) = \mathcal{O}(N)$. Tento postup opakujeme pro všechny dělitele čísla N, které jsou rovny alespoň 3.

Kolik takových dělitelů může být? Určitě ne víc jak $2\sqrt{N}$, protože ke každému děliteli $\leq \sqrt{N}$ existuje jednoznačně určený dělitel $\geq \sqrt{N}$ a naopak. Tím tedy dostáváme celkovou složitost $\mathcal{O}(N\sqrt{N})$.

To ale není nejlepší řešení, kterého jsme mohli dosáhnout. Pořád jsme zkoušeli zbytečně moc dělitelů. Ono totiž platí, že pokud $k = a \cdot b$ a $a, b > 1$ a existuje k-úhelník, tak určitě existuje i a-úhelník a b-úhelník, protože ty vybírají jen ně-

které vrcholy z celého k-úhelníka. A naopak, pokud neexistuje a-úhelník nebo b-úhelník, tak určitě nemůže existovat ani k-úhelník.

Stačí nám tedy testovat jen prvočíselné dělitele ≥ 3 a pak speciálně otestovat čtverec. Tak si jen N rozložíme na prvočísla, což klidně můžeme udělat jednoduše v $\mathcal{O}(N)$, a pak každé prvočíslo v tomto rozkladu otestujeme.

Nyní už jen odhadneme, kolik různých prvočísel v rozkladu můžeme mít. Každé prvočíslo nám vydělí N alespoň dvěma, takže jich určitě bude $\mathcal{O}(\log N)$, čímž celkem dostaneme $\mathcal{O}(N \log N)$. Tento odhad sice není nejlepší možný, ale na plný počet bodů stačil. Michal Punčochář například dokázal, že prvočíselných dělitelů je maximálně $\mathcal{O}(\frac{\log N}{\log \log N})$ a za tento odhad dostává jeden bonusový bod.

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-3-2.cpp`

Karel Tesař

25-3-3 Do třetice sekání

Přímočarý způsob, jak určit optimální políčko pro zadaný interval, je určit podle definice námahu pro každé políčko a ze spočítaných hodnot vybrat minimum. Zkoušíme tedy až N políček, pro každé z nich potřebujeme projít opět až N políček.

Přímočaré řešení tak má časovou složitost $\mathcal{O}(N)$ na určení námahy pro políčko, tedy $\mathcal{O}(N^2)$ na interval a $\mathcal{O}(D \cdot N^2)$ celkem. V případech, kdy D je řádově stejně velké jako N, můžeme také psát celkovou složitost jako $\mathcal{O}(N^3)$. Paměťovou složitost má $\mathcal{O}(N)$ (stačí nám pamatovat si jednotlivé hmotnosti trávy).

Lepší řešení

Pojďme se podívat, jestli to umíme lépe. Máme optimalizovat právě pro ty případy, kdy D je řádově stejně velké jako N. To nám celkem jasně napovídá, že si budeme chtít něco předpočítat.

To, co si předpočítáme, budou prefixy a „prefixy prefixů“, a to jak zleva, tak zprava. Za chvilku si ukážeme, že ty nám stačí na to, abychom námahu svozu na dané políčko určili v konstantním čase.

Označme t_i hmotnost trávy na i-tém políčku. Pak pro pole prefixů zleva bude platit $Pl_i = \sum_{j=1}^{i-1} t_j$, obdobně zprava bude $Pr_i = \sum_{j=i+1}^{N} t_j$ (speciálně $Pl_0 = Pr_N = 0$). Neboli prefixy nám říkají, kolik trávy se nachází v daném směru od našeho políčka.

Prefixy prefixů označme jako Cl, resp. Cr. Platí $Cl_0 = Cr_N = 0$, $Cl_i = Cl_{i-1} + Pl_i$, obdobně $Cr_i = Cr_{i+1} + Pr_i$. Říkají nám, kolik námahy dá svozit na dané políčko všechnu trávu nalevo, resp. napravo od něj. (Rozmyslete si, že to tak skutečně je – chceme-li svozit trávu na políčko i zleva, stojí nás to stejně

námahy, jako bychom ji sváželi na políčko $i-1$, a navíc musíme všechnu svezenou trávu posunout ještě o jedno políčko navíc.)

Prefixy dokážeme spočítat v lineárním čase. Při prvním průchodu spočítáme prefixy zleva, při druhém prefixy zprava. Časová složitost předzpracování tak bude $\mathcal{O}(N)$.

Ukažme teď, že tyto prefixy nám skutečně stačí, abychom námahu svozu trávy z intervalu na vybrané políčko určili v konstantním čase. Mějme $a, b : 1 \leq a \leq b \leq N$ a nějaké $i : a \leq i \leq b$.

Víme, kolik námahy nás stojí svoz trávy ze všech políček až do $i-1$ na políčko i. My od této námahy ale potřebujeme odečíst námahu na svoz trávy z políček 1 až $a-1$, protože z nich trávu ve skutečnosti svážet nebudeme. Tuto námahu můžeme rozdělit na námahu pro svoz na políčko a a pro následný přesun z a na i.

Už ale víme, kolik námahy stojí svoz na políčko a, je to hodnota Cl_a. Víme ale také to, kolik stojí následný přesun na i. Námaha odpovídá hmotnosti trávy nalevo od a vynásobené vzdáleností a od i, čili $Pl_a \cdot (i-a)$.

Tím tedy umíme spočítat cenu za svoz trávy v intervalu nalevo od i, je to $Cl_i - Cl_a - Pl_a \cdot (i-a)$. Podobně dokážeme spočítat cenu za svoz trávy v intervalu vpravo.

Tím jsme složitost snížili na $\mathcal{O}(N)$ pro každý interval, celkovou složitost pak na $\mathcal{O}(N + DN) = \mathcal{O}(N^2)$. Paměťová složitost zůstala $\mathcal{O}(N)$.

Optimální řešení

To ale pořád není optimální. Jak se změní námaha, když místo na i budeme trávu svážet na $i+1$? Zvýší se nám o Pl_{i+1} (všechnu trávu vlevo od $i+1$ vezeme o políčko dál) a sníží o Pr_i. Jinak řečeno, námaha se mezi dvěma políčky vždy zvyšuje o rozdíl $Pl_{i+1} - Pr_i$.

Protože máme zaručené kladné hmotnosti trávy, určitě platí, že tento rozdíl bude neklesající (ze začátku záporný a na konci kladný). To znamená, že cena se bude nějakou dobu snižovat (dokud budeme přičítat záporný rozdíl), a pak se zase začne zvyšovat.

Pro nás to má velice příjemný důsledek. Na hledání optimálního políčka tak totiž můžeme použít upravené binární vyhledávání. Místo abychom porovnávali hodnoty s nějakou předem určenou, podíváme se vždy na dvě sousední, a vydáme se tím směrem, kterým se hodnoty zmenšují.

Binárním vyhledáváním zvládneme optimální políčko pro daný interval najít v $\mathcal{O}(\log N)$, celková složitost tak bude $\mathcal{O}(N + D \log N) = \mathcal{O}(N \log N)$.

Pro zájemce ještě ukažme, že při tomto zadání není vůbec potřeba počítat prefixy prefixů ani prefixy zprava.

Už jsme ukázali, že cena se zvyšuje o $Pl_{i+1} - Pr_i$. Znamená to, že ideální je cena v případě, kdy $Pl_{i+1} = Pr_i$. Také víme, že $Pl_{i+1} + Pr_i = T$, kde T je součet hmotností veškeré trávy. Jednoduchou úpravou pak pro optimální změnu dostáváme $Pl_{i+1} = \frac{T}{2}$. Ideální cena je tedy tam, kde poprvé platí $Pl_{i+1} \geq \frac{T}{2}$.

Poznamenejme, že v tomto případě je daleko hezčí počítat prefixy jako součet hmotností včetně hmotnosti trávy na daném políčku, pak pro ideální políčko jako první platí $Pl_i \geq \frac{T}{2}$. Přesněji, při omezení na interval je to takové políčko, pro které jako první platí $Pl_i - Pl_{a-1} \geq \frac{Pl_b - Pl_{a-1}}{2}$.

Všimněte si, že tento postup neříká nic o tom, kolik ta námaha je, pouze najde políčko, pro které je optimální. To ale při našem zadání stačilo.

Za pomoc s úlohou děkuju Martinovi Hořeňovskému.

Program (C) – medián:
`http://ksp.mff.cuni.cz/viz/25-3-3-median.c`

Program (C) – prefixy:
`http://ksp.mff.cuni.cz/viz/25-3-3-prefixy.c`

Karolína „Karryanna“ Burešová

25-3-4 Zločinná záležitost

Výrobní fáze, závislosti mezi nimi… to musí být grafová úloha! A taky je. Fáze jsou jednoduše vrcholy a závislosti orientované hrany (vedoucí ve směru od fáze, která by měla proběhnout dříve).

Navíc je graf acyklický, neboť úloha má vždy řešení. Kdyby byl v grafu orientovaný cyklus, při výrobním procesu dojdeme do situace, kdy musíme zpracovat nějakou fázi z cyklu. Každá je však závislá na jiné fázi z cyklu, takže nelze žádnou z nich zpracovat. Neorientované cykly nám nevadí.

Lehčí varianta

Na vyřešení lehčí varianty úlohy stačilo dokonce jen přečíst si grafovou kuchařku[40] a všimnout si, že topologické třídění (neboli uspořádání) přesně řeší náš problém. Nicméně, vymyslet tento algoritmus z hlavy jistě také není těžké ;-)

Než se pustíme do těžší varianty, topologické třídění si krátce popíšeme. Budeme ho dělat po směru hran, čili obráceně než v kuchařce (chceme, aby hrany vedly z vrcholu s menším číslem do vrcholu s větším číslem). Nejprve najdeme *zdroje*, tedy vrcholy, do nichž nevede hrana (mají nulový *vstupní stupeň*). To můžeme udělat třeba při načítání vstupu.

Všechny zdroje si naskládáme do nějaké datové struktury, v níž umíme v konstantním čase odebírat a přidávat prvky, například do fronty. Jak budeme postupně zpracovávat vrcholy, budeme do fronty ukládat všechny vrcholy,

[40] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

které mají po odebrání zpracovaných vrcholů vstupní stupeň 0 (už do nich nevede hrana).

Dále postupně odebíráme z fronty vrcholy, dokud se fronta nevyprázdní. Platí, že k-tý odebraný vrchol bude k-tým v pořadí výrobního procesu. Po odebrání vrcholu z fronty tento vrchol smažeme i z grafu včetně hran z něho vedoucích. Když se nějakému jeho sousedu snížil vstupní stupeň na 0, šoupneme ho také do fronty.

Není těžké nahlédnout, že v grafu bez orientovaných cyklů vždy existuje zdroj a že se fronta vyprázdní až po zpracování všech vrcholů. Na podrobnosti k implementaci algoritmu a zdůvodnění správnosti odkazujeme čtenáře do kuchařky.

Při reprezentaci grafu seznamem sousedů je časová složitost $\mathcal{O}(N + M)$, protože v tomto čase najdeme zdroje spočítáním vstupních stupňů a poté se na každou hranu podíváme jen jednou. Paměťová složitost je na tom asymptoticky stejně, kromě grafu máme jen frontu na vrcholy a pro každý vrchol si pamatujeme aktuální vstupní stupeň. Kdybychom však ukládali graf maticí sousednosti, časová i paměťová složitost naroste na $\mathcal{O}(N^2)$.

Těžší varianta

Pro vyřešení těžší varianty stačilo upravit topologické třídění. Místo jedné fronty budeme mít dvě, každou pro jednu továrnu. Na začátku si vybereme jednu továrnu, a dokud to jde, odebíráme z její fronty. Když je prázdná, provedeme převoz do druhé továrny, odebíráme z její fronty, až se vyprázdní, přesuneme se zpět do první továrny…

V průběhu samozřejmě dáváme vrcholy se vstupním stupněm 0 do fronty té továrny, v níž se má dělat příslušná fáze. Skončíme, když dojdou vrcholy v obou frontách.

Zbývá vyřešit, jakou továrnou začít. Jelikož jsou jen dvě, prostě zkusíme začít nejprve s jednou a pak s druhou a vypíšeme výsledek s méně převozy.

Algoritmus jistě vytvoří topologické uspořádání, nicméně potřebujeme zdůvodnit, že to zvládne na nejmenší možný počet převozů. Prvně jde jednoduše nahlédnout, že když lze nějakou fázi zpracovat bez převozu, můžeme tak učinit hned a neuškodíme si tím. Navíc nezáleží na pořadí výběru fáze ke zpracování, když jich lze zpracovat více bez převozu.

Důležité je, že náš algoritmus vždy zpracuje co nejvíce fází, než proběhne převoz. Nemůže tedy existovat nějaké jiné pořadí s méně převozy – jednak by si nepomohlo tím, že mezi dvěma převozy vynechá fázi, kterou zpracoval náš algoritmus. Druhak by nemohlo zpracovat mezi dvěma převozy ani nic navíc, protože ty fáze by jinak zpracoval ve stejnou dobu i náš algoritmus (musí mít někdy vstupní stupeň 0).

Korektnost algoritmu je zdůvodněna a časová ani paměťová složitost topologického třídění se úpravou pro dvě továrny nepokazí, pořád činí $\mathcal{O}(N + M)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-3-4.c`

Pavel Veselý

25-3-5 Histogram

Pro úlohu se nabízí vcelku triviální řešení, kdy vyzkoušíme všechny možné začátky a konce posloupností sloupečků a vybereme minimum z výšek v této posloupnosti. To nám určí obdélník. Ze všech takovýchto obdélníků nám stačí vzít ten s maximálním obsahem. Potíž tohoto řešení je jeho časová složitost, která je kvadratická. My si ukážeme řešení pracující v čase lineárním.

Mějme i-tý sloupeček s výškou h_i. Pokud chceme zjistit obsah největšího obdélníku, který obsahuje i-tý sloupeček celý, stačí nám zjistit, kolik sloupečků j s výškou $h_j \geq h_i$ se nachází bezprostředně před a po i-tém sloupečku.

Nejdřív spočítáme, kolik je sloupečků s menší nebo větší výškou bezprostředně před i-tým sloupečkem (značíme l_i). Analogický postup pak můžeme použít na sloupečky k i-tému přiléhající zprava (značíme r_i).

Pro nalezení l_i (resp. r_i) nám stačí vhodně použít zásobník. Pro každý sloupeček, počínaje prvním, provedeme následující postup. Pokud je zásobník prázdný, přidáme na něj index i a pokračujeme dále. Pokud zásobník prázdný není, budeme z vrcholu mazat indexy j tak dlouho, dokud bude platit $h_j \geq h_i$ nebo zásobník nebude prázdný. Rozdíl indexu sloupečku na vrcholu zásobníku a indexu i-tého sloupečku je teď evidentně námi hledané l_i. Nakonec na vrchol zásobníku vložíme index i-tého sloupečku.

Zbývá si uvědomit, že zásobník po i-té iteraci je ve stavu, který dá korektní řešení i pro následující sloupečky. To nahlédneme rozborem případů. V případě, že je $(i + 1)$-tý sloupeček menší než byl i-tý, stačí smazat vrchol zásobníku a popřípadě další indexy. Snadno nahlédneme, že to, co bylo smazáno v i-tém kroku, mělo být smazáno.

Naopak, pokud je $(i + 1)$-tý sloupeček ostře vyšší než předchozí, algoritmus ze zásobníku nic neodebere a $l_{i+1} = 0$, což je správně. Pro další sloupečky pak korektnost plyne z matematické indukce podle indexů sloupečků.

Lineární časová složitost plyne z následujícího pozorování. Počet odebrání indexu ze zásobníku bude maximálně tolik, kolik indexů do zásobníku přidáme. A těch přidáme n, přičemž každý právě jednou. Paměti nám stačí taktéž lineárně.

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-3-5.cpp`

Jan Bok

25-3-6 Rytíř a princezny

K řešení využijeme takzvaný hladový přístup. Více o tomto přístupu si můžete vyhledat pod heslem *hladové algoritmy*, resp. *greedy algorithms*.

Budeme postupovat společně s rytířem jeho trasou a pomyslně zabijeme každého draka, který nám vstoupí do cesty. Když přistoupíme k princezně krásy K (jiné než vytoužené poslední), může se přihodit, že jsme zabili více draků, než jsme mohli. V tom případě si chceme zabití některých draků rozmyslet.

Z našeho seznamu zabitých draků odebereme draky s pokladem nejnižší hodnoty tak, abychom kolem princezny mohli bezpečně projít. Zbude tedy $K - 1$ draků s nejhodnotnějším pokladem. Když přistoupíme k poslední princezně, zkontrolujeme počet zabitých draků a zjistíme, zda jsme uspěli.

Nejdříve si rozmysleme, proč tento naivní přístup bude fungovat. V našem postupu dáváme šanci být zabit každému drakovi a mažeme jej až v situaci, kdy musíme počet draků snížit na $K - 1$ draků a známe $K - 1$ draků, kteří jsou alespoň stejně výnosní a lze je zabít. Tedy draka odebereme tehdy, když jistě víme, že jej odebrat musíme, a na konci nám pak zůstanou draci nejlepší.

Jak bude tento přístup efektivní? V našem řešení potřebujeme datovou strukturu s následujícími dvěma operacemi: vložení prvku a odebrání nejmenšího prvku. Pokud bychom použili obyčejné pole, stálo by vložení prvku konstantní čas a odebrání nejmenšího prvku čas lineární. Tak bychom dosáhli časové složitosti řešení $\mathcal{O}(N^2)$, kde N je délka rytířovy cesty.

Vhodnější strukturou je binární halda, která obě operace zvládá v logaritmickém čase. Výsledná časová složitost s haldou je $\mathcal{O}(N \log N)$, paměťová složitost je $\mathcal{O}(N)$. Pro seznámení s haldou nahlédněte do příslušné kuchařky o haldách.[41]

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-3-6.cpp`

Lukáš Folwarczný

25-3-7 Zkratky

Díky omezení počtu a délky zkratek malými konstantami se ukázala tato úloha jako velmi jednoduchá. Nejpřímočařejším postupem bylo asi vydat se vstříc této úloze s pomocí dynamického programování.

Vyjdeme z toho, že prázdné slovo (slovo délky nula) určitě ze zkratek poskládat umíme. Pokud na toto prázdné slovo navazuje řetězec (posloupnost znaků) odpovídající některé ze zkratek, umíme poskládat i toto prodloužené slovo. Této

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

počáteční části slova ze vstupu budeme říkat *prefix*. Takovým způsobem můžeme postupovat dál, dokud se nám nepovede poskládat celé slovo, nebo dokud nezjistíme, že už nemáme žádnou možnost jak pokračovat.

Jak to konkrétně provedeme? Půjdeme na to z druhé strany a budeme postupně pro každý znak slova ze vstupu určovat, jestli v něm končí nějaký poskládatelný prefix. Vezmeme si všechny zkratky a zkusíme je umístit tak, aby končily v právě zpracovávaném znaku.

Pokud se budeme nacházet na K-tém znaku vstupního slova a budeme zkoumat, jestli umíme poskládat tento prefix tak, aby zde končila nějaká zkratka délky S, tak se podíváme, jestli podslovo končící na pozici $K - S$ umíme složit. Pokud ne, nemá smysl tuto variantu dál řešit.

Pokud ale ano, je zde možnost, že za prefix končící na pozici $K - S$ můžeme přidat tuto zkratku a vytvořit tak nový (delší) prefix končící na pozici K. Zkontrolujeme tedy znak po znaku, jestli nám tam tato zkratka sedí. Pokud ano, poznamenáme si to k indexu K a pokračujeme dál. Celé slovo pak lze poskládat právě tehdy, pokud lze poskládat prefix končící posledním písmenem slova.

Správnost jednoduše zdůvodníme následujícím pozorováním. Budeme předpokládat, že pro všechny pozice vlevo od aktuálně zpracovávaného už máme jednoznačně určeno, jestli je umíme poskládat. Každý poskládatelný prefix končící na pozici K můžeme rozložit na dvě části: na nějaký kratší prefix a na některou ze zkratek navazující na tento kratší prefix. Náš postup ale právě takový rozklad najde a tak tuto pozici označí za poskládatelnou.

Naopak, pokud takový rozklad pro tuto pozici neexistuje (a tento prefix tedy poskládat nelze), tak je triviálně vidět, že ani náš algoritmus tuto pozici neoznačí za poskládatelnou. Tím jsme jistě tuto vlastnost dokázali pro další pozici a indukcí dokážeme správnost algoritmu pro celý vstup.

Pro výpočet časové složitosti si označme Z jako součet délek všech zkratek. Pak musíme udělat celkem N kroků algoritmu a v každém projdeme až všechny zkratky, tedy $\mathcal{O}(NZ)$. Pokud si dovolíme považovat Z za konstantu, je pak složitost lineární vzhledem k délce vstupu, tedy $\mathcal{O}(N)$.

Pokud si uvědomíme, že nám stačí si pamatovat jen tu část vstupu, se kterou aktuálně pracujeme (nemusíme sahat více do minulosti, než je délka nejdelší zkratky), tak nám stačí pouze $\mathcal{O}(Z)$, respektive $\mathcal{O}(1)$ paměti, pokud opět Z prohlásíme za konstantu.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-3-7.c`

Jiří Setnička

25-3-8 Tabulatika

Řešení třetího dílu jste se zhostili velmi úspěšně. Nutno ovšem poznamenat, že během sbírání technických zkušeností s TEXem byste měli také sbírat estetické a typografické zkušenosti. Pořád je co dohánět, rád vidím esteticky dotažená řešení některých z vás, vzápětí však skřípu chrupem nad jiným řešením, kde se jiný řešitel ani nesnažil, aby to nějak vypadalo.

Úkol 1

Řešení prvního úkolu bylo jednoduché:

```
\settabs\+Uherské Hradiště &Jablonec nad Nisou
         &30. 12. &Kolik km&\cr
\+\it Odkud&\it Kam&\it Kdy&\it Kolik km\cr
\+Praha&Olomouc&21. 12.&\hfill 250&\cr
\+Olomouc&Uherské Hradiště&30. 12.&
  \hfill 130&\cr
\+Uherské Hradiště&Vyšší Brod&5. 1.&
  \hfill 350&\cr
\+Vyšší Brod&Jablonec nad Nisou&17. 1.&
  \hfill 324&\cr
```

Použili jste znalosti ze seriálu, verze `\settabs` se vzorovým řádkem. Někteří z vás použili `\settabs 4\columns`, což jsem hodnotil jedním záporným bodem, neboť taková verze měla třetí a čtvrtý sloupec šeredně roztahaný.

Všimněte si, že vzorový řádek končí `&\cr`, neboli na konci řádku vzniká fiktivní pátý sloupec, aby bylo k čemu zarovnat obsah čtvrtého sloupce.

Úkol 2

Tento úkol tvořil téměř půlku všech dosažitelných bodů. Uvedu zde jedno z možných řešení, různých přístupů bylo mnoho. Ukážeme si řešení s fixním počtem úloh.

Nejprve si trochu zvětšíme stránku, ať se vejdeme:

```
\hoffset-15mm
\advance\hsize by 3cm
\voffset-15mm
\advance\vsize by 3cm
```

Pak se naučíme zlý trik s `\lowercase`:

```
\lccode‘\~‘\,\relax
\lowercase{%
\def\normalcomma{\def~{,}}
\def\mathcomma{\def~{{,}}}
}
\lccode‘\~‘\-\relax
\lowercase{%
\def\normaldash{\def~{-}}
\def\omitdash{\def~{}}
}
\normalcomma
\normaldash
\def\pointcell{\mathcomma\omitdash}
```

Primitivum `\lowercase` (a jeho bratříček `\uppercase`) překládají tokeny podle tabulek `\lccode` a `\uccode`. Primitivum funguje tak, že ztokenizuje svůj „parametr“ a ve všech tokenech, kromě řídících sekvencí, změní kódy znaků podle příslušné tabulky. V základním nastavení se mění jen velká písmena na malá, resp. obráceně.

Přenastavení nějaké hodnoty se ale zhusta využívá právě uvedeným stylem. Tedy vlnka, která je stadnardně aktivním znakem (token (`~`, 13)), se uvnitř prvního `\lowercase` stane aktivní čárkou (token (`,`, 13)) a uvnitř druhého `\lowercase` aktivní pomlčkou. Jde to i jinak, ale tohle je asi nejčistší.

Uvnitř tabulky se skóre si totiž nastavíme pomlčku i čárku jako aktivní a na různých místech si přejeme, aby se chovaly různě. Konkrétně jde o buňky s počtem bodů, kde chceme, aby se místo pomlčky vysázelo prázdné místo a aby byly na obě strany okolo čárky stejné mezery.

```
\def\scoretable{\begingroup
\catcode‘\, 13 \catcode‘\- 13\relax
\doscoretable}
```

Další trik. Makro `\scoretable` je ve skutečnosti bez parametrů. Nejdřív si přenastavíme kategorie znaků a pak si teprve načteme parametry makra.

Následuje definice hlavičky tabulky:

```
\def\doscoretable#1{%
\line{\hfil\vbox{\halign{\strut%
##\hfil\quad&%
##\hfil\quad&%
##\hfil\enskip&%
\hfil##\hfil\enskip&%
```

```
\hfil##\hfil\enskip\vrule&%
\enskip%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip&%
\hfil\pointcell##\hfil\enskip\vrule&%
\enskip\hfil\pointcell##\quad&%
\hfil\pointcell##\cr
&\it řešitel&\it škola&\it ročník&\it sérií%
&\it 2521&\it 2522&\it 2523&\it 2524%
&\it 2525&\it 2526&\it 2527&%
\it série&\it celkem\cr
#1
}}\hfil}\endgroup}
```

Všimněte si zdvojených `#`. Jsme uvnitř definice makra, tedy je potřeba tento znak zdvojit, aby nebyl interpretován.

Pokud vám chybí `\begingroup` (protože na konci je samotné volání `\endgroup`), podívejte se o kousek výš do definice `\scoretable`.

Ještě by to chtělo definici jednotlivých řádků:

```
\def\scoreline#1. #2 (#3; #4; #5): #6: #7 #8 {%
\def\m.{}%
#1.&#2&#3&$#4$&$#5$&\scorepoints#6!&$#7$&$#8$\cr
}
\def\scorepoints#1 #2 #3 #4 #5 #6 #7!{%
$#1$&$#2$&$#3$&$#4$&$#5$&$#6$&$#7$%
}
```

Zde je důležitý trik s dvoufázovým zpracováním parametrů makra. TEX umí zpracovat jen devět parametrů současně, proto jsme seznam bodů za jednotlivé úlohy nejprve prohlásili za jeden argument, který jsme pak nechali zpracovat makrem `\scorepoints`.

Makro `\m` slouží k polknutí tečky za pořadím účastníka v případě, že chceme prázdný první sloupec.

A teď už vzorové použití:

```
\scoretable{
\scoreline \m. {} (; ; ):
           13 11 6 13 8 9 13: 59,0 118,0
```

```
\scoreline 1. Rastislav Rabatin (GJHBA; 4; 5):
           13 7,5 - 13 8 9 8,5: 54,8 109,5
\scoreline 2. Ondřej Hlavatý (GJirsíkaČB; 4; 2):
           4 5 - 13 8 4 13: 49,6 102,5
\scoreline 3. Michal Punčochář (GJíroČB; 3; 7):
           6,5 11 - 13 8 - 13: 53,2 98,9
\scoreline 11. Jakub Maroušek (G\_Písek; 3; 2):
           5,5 4 - 4 - 0 10,7: 36,1 73,5
\scoreline 12.--13. Mikuláš Hrdlička (MG; 2; 2):
           4 - 3,5 6 2,5 - -: 26,8 72,9
\scoreline \m. Matej Lieskovský (GOmPha; 3; 7):
           2,5 - 4 - 3 7 9: 30,7 72,9
\scoreline 14. Jakub Svoboda (GKomHavíř; 3; 2):
           - 7 4 4 1,5 2 -: 29,6 71,2
\scoreline 54. Přemysl Šťastný (GZamb; -1; 1):
           - - - - - - -: 0,0 4,7
}
```

Pokud se vám nepovedlo správně vysázet čárky nebo vyházet pomlčky, nestrhával jsem za to žádné body. Někteří z vás nedodali makro, ale jenom sazbu, což jsem honoroval zhruba půlkou bodů.

Vytvořit verzi, která zvládne proměnlivý počet úloh, by také šlo, dokonce i bez číselných proměnných a bez podmínek, které vysvětlujeme ve čtvrtém dílu, ale bylo by to příliš ošklivé. Úplně stačila verze pro fixní počet úloh.

Úkol 3

Řešení posledního úkolu bylo přímočaré až na jednu drobnost. Spousta z vás přišla o půlbod kvůli nule ve třetím řádku, kterou jste měli příliš nacpanou na zlomek nad ní. To šlo jednoduše opravit přidáním `\strut`.

```
$$Z = \left\{\matrix{N > 0:&
  \displaystyle\sum_{i=1}^N
    \left(2\sum_{i<j}
      \log\left|\lambda_i-\lambda_j\right|
      - \sum_{i=1}^N V(\lambda_i)\right)\cr
  N < 0:& \displaystyle-{1\over N^2}\cr
  N = 0:& \strut0\cr
}\right.$$
%% Pravá } tu již není, proto \right jen tak.
```

Většina z vás si všimla primitiva `\displaystyle`, díky kterému se daly vysázet hezké velké sumy a zlomky. Někteří použili `\limits`, čímž přehodili indexy u sum nad znaménka, ale stejný trik se nedal použít na zlomek ve druhém řádku, který tak zůstal malinký.

Úkol 4

Centrovaná sazba v posledním úkolu vám dala kupodivu docela zabrat. Nicméně mnoho z vás se dobralo k nějakému správnému řešení, třeba k tomuto:

```
%% Odsadit první řádek by bylo divné.
\parindent 0pt
%% Poslední řádek nebude nijak doplněn.
\parfillskip 0pt
%% Zleva a zprava stejné místo, natahovací.
\leftskip 0pt plus \hsize
\rightskip 0pt plus \hsize
```

Jan „Moskyto“ Matějka

25-4-1 Přesmyčky

Při řešení této úlohy budeme pro jednoduchost předpokládat, že K se nám vejde do nějaké normální proměnné, a tedy že s ním ještě dokážeme provádět aritmetické operace v konstantním čase (v opačném případě bychom pak jen časovou složitost museli vynásobit $\log K$). Druhou věcí, kterou jsme v zadání asi ne úplně přesně uvedli, je to, že operujeme s konstantně velkou abecedou (26 písmen). Pokud by však abeceda byla větší, tak bychom její velikostí museli časovou i paměťovou složitost vynásobit. Při hodnocení vašich řešení však ani jedna z možností neměla na bodový zisk vliv, protože jsme zadání zformulovali volně.

Lehčí varianta

Nejdříve provedeme několik pozorování. Pro jednodušší případ a slovo délky N máme přesně $N!$ možností, jak můžeme toto slovo uspořádat. Když však první písmeno zvolíme pevně, tak máme již jen $(N-1)!$ možností uspořádání zbylých písmen.

Přesmyčka začínající na lexikograficky nejmenší písmeno tak může mít pořadové číslo v rozsahu $1, \ldots, (N-1)!$, přesmyčka začínající na v pořadí druhé písmeno může mít pořadové číslo mezi $(N-1)!+1, \ldots, 2\cdot(N-1)!$ atd. Pokud tedy má hledaná přesmyčka pořadové číslo K, tak jako první znak zvolíme písmeno s pořadovým číslem k (indexujeme od nuly):

$$k = \left\lfloor \frac{K}{(N-1)!} \right\rfloor$$

Tím jsme vyřešili první znak, jak s ostatními? Stačí si uvědomit, že vlastně hledáme nějakou přesmyčku s pořadovým číslem K' na $N-1$ zbylých znacích. Stačí nám od původního K odečíst tolik přesmyček, kolik jsme jich volbou k-tého písmene přeskočili. Tedy zvolíme $K' = K - k\cdot(N-1)!$ a rekurzivně postupujeme pro celé slovo (jen v každém kroku nesmíme zapomenout brát k-té písmeno jen ze zatím nepoužitých písmen).

Implementace je v tomto případě jednoduchá, jen si přepočítáváme průběžně K a N. Pro nalezení a průběžné odmazávání k-tého písmena v pořadí můžeme použít pole nebo nějaký vyhledávací strom.

Těžší varianta

V případě opakování písmen se nám úloha mírně komplikuje. Po zvolení prvního znaku již nemáme právě $(N-1)!$ možností poskládání zbytku slova, ale pokud si jako m označíme počet různých znaků a jako p_i pro i od 1 do m jejich četnosti, tak je to:

$$\frac{(N-1)!}{p_1!\cdot p_2!\cdot \ldots \cdot p_m!}$$

(můžeme si všimnout, že to přesně odpovídá jednoduššímu případu pro všechny četnosti rovny jedné).

Postup je pak už stejný jako v jednodušším případě, jen musíme vymyslet, jak budeme rychle upravovat tento vzorec. Při snížení faktoriálu v čitateli o jedna ho jen vydělíme odpovídajícím N, při snížení četnosti některého z písmen z hodnoty p_i na $p_i - 1$ ho vynásobíme p_i. Obě tyto operace zvládneme stejně rychle jako jiné aritmetické operace.

Paměťová složitost je $\mathcal{O}(N)$, protože si musíme všechny znaky přečíst do paměti a ke každému si pamatovat konstantně mnoho údajů, jako je četnost (můžeme dokonce odhadnout paměťovou složitost jako $\mathcal{O}(m)$, ale m může být až N a tedy se složitost asymptoticky nezmění).

Časová složitost je také lineární k délce vstupu, tedy $\mathcal{O}(N)$. Pokud bychom však pracovali s velkým vstupem a velkou abecedou (viz poznámka v úvodu řešení), tak by se nám změnila až na $\mathcal{O}(N \cdot L \log K)$. Vzorový program implementuje těžší variantu.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-4-1.c`

Jirka Setnička

25-4-2 Plánování trasy

Úloha vypadala na první pohled velmi jednoduše. Proto se do ní pustila téměř polovina z vás, kteří jste poslali řešení alespoň jedné z úloh čtvrté série. Úloha však skrývala několik záludností. Podívejme se na řešení, které se jim vyhýbá.

Nejprve si pro každé políčko předpočítáme vzdálenosti od překážek ve všech čtyřech směrech. Díky tomu pak v programu dokážeme okamžitě určit, na kterém místě budeme příště zatáčet, pokud se vydáme daným směrem.

Vzdálenosti od levé překážky určíme tak, že projdeme postupně celou mapu po řádcích zleva doprava. Pokud je první políčko na řádku volné, přiřadíme mu vzdálenost rovnou nule. Pokud volné není, přiřadíme mu číslo -1. Každému dalšímu políčku, které je volné, přiřadíme vždy hodnotu o jedna větší. Políčkům, která volná nejsou, přiřadíme opět hodnotu -1.

Podobně vypočítáme vzdálenosti od překážek v ostatních směrech: od pravé překážky postupujeme po řádcích zprava doleva, od horní překážky po sloupcích shora dolů a od dolní překážky po sloupcích zdola nahoru.

K čemu nám tato čísla pomohou? Když se z libovolného políčka vydáme některým směrem, budeme vědět, že na překážku narazíme až v políčku, které má příslušnou souřadnici větší nebo menší o takto vypočtenou vzdálenost. Pou-

ze v těchto bodech budeme měnit směr. Libovolnou trasu pak popíšeme jako posloupnost políček, na nichž jsme směr měnili.

Zbývá zajistit, abychom nepřejeli přes cílové políčko. K tomu nám může pomoci malý trik. Pokud se při úvodním výpočtu vzdáleností dostaneme do políčka s cílem, hodnotu vzdálenosti vynulujeme. Tím zabezpečíme, že se zastavíme v cílovém políčku a nepřejedeme je až k následující překážce.

Celý předvýpočet dokážeme provést v čase $\mathcal{O}(MN)$, kde M a N jsou rozměry mapy. Mapu totiž projdeme čtyřikrát, počet průchodů je tedy konstantní.

Teď již můžeme hledat trasu od startu do cíle, která bude obsahovat co nejméně zatáček, druhotně co nejméně políček.

Při hledání optimálních cest se často vyplatí použít nějakou úpravu algoritmu prohledávání do šířky. Prohledávání do šířky je grafový algoritmus. Přečtěte si o něm v grafové kuchařce.[42]

Nyní si místo mapy představme graf, v němž vrcholy odpovídají políčkům změn směrů a hrany odpovídají rovným trasám mezi nimi. Samotný algoritmus prohledávání do šířky nám zajistí minimalizaci počtu obratů.

Potřebujeme ještě mezi trasami se stejným počtem obratů vybrat tu nejkratší. K vrcholům, k nimž při prohledávání do šířky dorazíme, si poznamenáme počet políček, která jsme museli na celé trase od startu k nim překonat. Tuto hodnotu nebudeme nikdy zvyšovat a přepíšeme ji jenom v případě, že tím nezvýšíme počet zatáček na cestě do daného vrcholu.

Na závěr si jenom musíme dát pozor: nemůžeme se zastavit okamžitě, když dojdeme do cíle, ale až tehdy, kdy cílové políčko vyndaváme z fronty.

Složitost celého algoritmu je $\mathcal{O}(MN)$, tedy lineární s počtem políček. Je tomu tak proto, že vrcholů není více než políček mapy a z každého vrcholu vedou maximálně čtyři[43] hrany.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-4-2.c`

Jirka Setnička a Jenda Hadrava

[42] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

[43] Stačí dvě hrany. Snadno nahlédneme, že návrat se nikdy nevyplatí a mimo cílové a startovní pole nelze pokračovat rovně.

25-4-3 Rozpis svozu

Podobně jako u úlohy 25-3-3[44] i tentokrát přímočaré řešení spočívalo ve vyzkoušení všech políček, spočítání příslušné námahy a průběžném přepisování minima.

I tentokrát by takové řešení bylo dost pomalé, přesněji by mělo časovou složitost $\mathcal{O}((NM)^2)$ na každý dotaz, pro K dotazů tedy celkem $\mathcal{O}(KN^2M^2)$. Pokud by M i K řádově odpovídaly N, máme $\mathcal{O}(N^5)$.

Pojďme se tedy zase podívat, jestli to umíme lépe. A začněme bližším prozkoumáním toho, jak se počítá námaha a co z toho plyne.

Nechť t_p je množství trávy na políčku p a p_x, resp. p_y jsou souřadnice políčka p. Námaha na svoz trávy z každého políčka p v nějaké oblasti na políčko se souřadnicemi $[x, y]$ pak odpovídá výrazu:

$$\sum_p (|p_x - x| + |p_y - y|) \cdot t_p$$

Tenhle vzoreček můžeme ale roznásobením a rozepsáním upravit na tvar:

$$\left(\sum_p |p_x - x| \cdot t_p\right) + \left(\sum_p |p_y - y| \cdot t_p\right)$$

Právě jsme ukázali, že souřadnice jsou nezávislé, takže můžeme nezávisle na sobě hledat nejvýhodnější sloupec a nejvýhodnější řádek.

Samo zadání upozorňovalo na podobnost s úlohou minulé série 25-3-3, pojďme tedy prozkoumat, jestli úlohu neumíme převést na jednorozměrnou variantu. Ta pracovala s prefixovými součty trávy a prefixovými součty těchto prefixových součtů na jediném řádku.

Uvažujme bez újmy na obecnosti, že hledáme nejvýhodnější sloupec. Při dotazu na oblast bychom tak potřebovali mít k dispozici nikoli prefixové součty pro řádek, ale pro oblast, resp. součty prefixových součtů přes všechny řádky oblasti.

Představme si, že pro každé políčko víme, kolik námahy stojí svézt do něj trávu z oblasti vymezené levým horním rohem a naším políčkem, to celé za předpokladu, že přesuny po y-ové ose máme zadarmo. Námahu tedy počítáme pouze za přesuny doprava a doleva.

Řekněme, že tuto námahu máme v poli $S\ell$, podobně v Sr budeme mít námahu pro svoz z oblasti vymezené pravým horním rohem a naším políčkem.

[44] `http://ksp.mff.cuni.cz/viz/25-3-3`

Ještě se nám budou hodit pole $P\ell$, resp. Pr udávající, kolik je v těchto oblastech celkem trávy.

Nechť máme oblast vymezenou souřadnicemi $[x, y]$ a $[X, Y]$ a chceme spočítat námahu za svoz trávy na políčko $[a, b]$. Stejně jako v 1D variantě si námahu rozdělíme na námahu za svoz zleva a námahu za svoz zprava.

Námaha zleva bude $S\ell_{a,Y} - S\ell_{a,y-1} - (S\ell_{x-1,Y} - S\ell_{x-1,y}) - (P\ell_{x-1,y} - P\ell_{x-1,y-1} \cdot (a - (x - 1)))$. Základem je $S\ell_{a,Y}$. $S\ell_{a,b}$ totiž bere v úvahu pouze řádky $0 \ldots b$, zatímco $S\ell_{a,Y}$ pokrývá celou zadanou oblast. Připomeňme ještě, že pro hodnoty $S\ell$ počítáme s tím, že přesuny nahoru a dolů máme zadarmo.

Rozdílem $S\ell_{a,Y} - S\ell_{a,y-1}$ jsme tedy získali námahu za přesun veškeré trávy z oblasti $[1, y]$, $[a, Y]$ na políčko $[a, Y]$ (nebo kterékoli jiné v sloupci a).

Dál jsme podobně jako v 1D variantě odečetli námahu za svoz trávy z oblasti $[1, y]$, $[x-1, Y]$ na políčko $[x-1, Y]$ a nakonec námahu na přesun trávy ze stejné oblasti mezi políčky $[x - 1, Y]$ a $[a, Y]$.

Stejným způsobem můžeme spočítat námahu za svoz trávy zleva. Ideální sloupec tedy můžeme najít stejně jako v jednorozměrné variantě úlohy upraveným binárním vyhledáváním tak, že vždy porovnáme námahu pro dvě sousední políčka.

Podobně dokážeme najít ideální řádek. Místo $S\ell$, resp. Sr budeme mít Rh, resp. Rd (shora, zdola).

Zatím jsme předpokládali, že všechna pomocná pole máme k dispozici, ale neukázali jsme, že si je opravdu umíme opatřit. Pojďme to teď napravit.

Pole $P\ell$ vyrobíme iterováním přes řádky. Na začátku máme $P\ell_{x,0} = 0$. Pro každý řádek si pamatujeme dosavadní součet trávy na tomto řádku, řekněme s, pak platí $P\ell_{x,y} = P\ell_{x,y-1} + s$.

Pro pole $S\ell$ platí $S\ell_{0,y} = 0$ a $S\ell_{x,y} = S\ell_{x-1,y} + P\ell_{x-1,y}$ (potřebujeme vynaložit námahu na svoz trávy do vedlejšího sloupce a pak všechnu dosud potkanou trávu převézt ještě o jeden sloupec dál). Podobně $Rh_{x,0} = 0$, $Rh_{x,y} = Rh_{x,y-1} + P\ell_{x,y-1}$.

Pravostranné varianty, resp. varianta zdola, fungují stejným způsobem.

Předpočítat pomocná pole tedy dokážeme v lineárním čase. Výpočet námahy umíme konstantně, vyhledání optimálního sloupce tak umíme v $\mathcal{O}(\log N)$, optimálního řádku v $\mathcal{O}(\log M)$. Celková složitost tedy je $\mathcal{O}(MN + K(\log N + \log M))$. Paměťová složitost je $\mathcal{O}(NM)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-4-3.c`

Karolína „Karryanna“ Burešová

25-4-4 Podplácení

Úloha byla velmi snadná a v drtivé většině jste si s ní hravě poradili. Pojďme si pro ty, co ji neřešili, řešení ukázat.

Dokážeme, že první hráč má vyhrávací strategii, a to pro libovolné N.

První případ nastává pro N lichá. V takovém případě první hráč podplatí policistu ve prostředku poslední řady. Tím vzniknou dvě stejné pyramidy o délce základny $(N-1)/2$. Jakkoli teď zahraje druhý hráč, zahraje v dalším tahu první hráč úplně stejně na druhé pyramidě. Taková strategie se nazývá zrcadlová. Je snadno vidět, že poslední bude táhnout právě první hráč.

Pro sudá N je situace obdobná. První hráč podplatí prostředního policistu v předposlední řadě, čímž vzniknou opět dvě stejné pyramidy. Stačí hrát opět zrcadlově a vítězství je v kapse.

Jan Bok

25-4-5 Účetnictví

Jedno řešení, které můžeme rychle zamítnout, je zkoušet všechny možnosti. Počet způsobů roste plus minus exponenciálně rychle.

Něco nad polovinu bodů dostali ti, které osvítilo dynamické programování. Řekněme, že víme, kolika způsoby je možné se po pěti dnech dostat na všechny částky, které můžeme mít na účtu: 0 Kč tam můžeme dostat třeba pěti způsoby, 1 Kč dvěma, atd.

Kolika způsoby se můžeme do nějaké částky X dostat za šest dní? V šestém dni jsme mohli buď přidat 6 Kč, nebo je odebrat. Stačí tedy sečíst, kolika způsoby jsme se zvládli za pět dní dostat do $(X-6) \bmod N$ a $(X+6) \bmod N$. (Připomeňme si, že N značí číslo, kterým úřad modulí, a K je počet dní naší defraudace.)

Můžeme si takhle postupně stavět počty způsobů, a jakmile projdeme všechny dny, vypíšeme, kolika způsoby se můžeme vrátit na nulu. Jak dlouho tohle bude trvat? $\mathcal{O}(NK)$: pro každý den musíme přepočítat počet způsobů jak se dostat do všech N možných částek. Mohlo by se zdát, že budeme potřebovat i $\mathcal{O}(NK)$ paměti, protože pro každý den počítáme počty způsobů, ale dokážeme to i s $\mathcal{O}(N)$. Stačí si totiž ukládat vždy jenom počty způsobů v předchozím dni a do dočasného pole postupně přičítat způsoby v dalším dni.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-4-5.c`

Na plný počet bodů dosáhli ti, které napadl krok stranou – vyjádření přes matice a jejich rychlé násobení.

Učiníme drobné pozorování: každých N dní algoritmus dělá v podstatě to samé! Když třeba přidáváme $N + 10$ Kč, je to stejná operace, jako kdybychom přidávali jenom 10 Kč.

Použijeme trik a uložíme si do matice (třeba jménem M) popis toho, co se s počty způsobů jak dosáhnout jednotlivé částky stane, když přidáme nebo odebereme nejdřív 1 Kč, pak 2 Kč, pak 3 Kč, a tak dále až do N. A takovouhle matici si můžeme vystavět například tak, že si vytvoříme matice „přesuň 1 Kč“, „přesuň 2 Kč“, ..., a vynásobíme je.

Když M umocníme na $\lfloor K/N \rfloor$, dostaneme tím matici, která spočítá počty způsobů po $K - (K \bmod N)$ dnech. Násobit matice velikosti $N \times N$ umíme za čas $\mathcal{O}(N^3)$.[45] Takových násobení provedeme $\mathcal{O}(K/N) + \mathcal{O}(N)$ – první člen je za „skok“ na den $K - (K \bmod N)$, druhý za dopočítání do K. Celkem by to tedy trvalo $\mathcal{O}(KN^2) + \mathcal{O}(N^3)$, ale protože v naší úloze je K podstatně větší než N, zpomaluje nás nejvíc $\mathcal{O}(KN^2)$.

S tímhle členem ale ještě umíme zamávat. Mocnění matice M na K/N přece umíme rychleji než za $\mathcal{O}(K/N)$ násobení! Můžeme použít trik popsaný v kuchařce o teorii čísel [46] , kterými $\mathcal{O}(K/N)$ umoříme na $\mathcal{O}(\log(K/N))$.

Když použijeme rychlé mocnění matic, najednou vypadá složitost už o něco lépe: $(\mathcal{O}(\log(K/N)) + \mathcal{O}(N)) \cdot \mathcal{O}(N^3) = \mathcal{O}(N^2 \log K) + \mathcal{O}(N^4)$. Teď nás zase ale straší $\mathcal{O}(N^4)$. Toho se ale dokážeme zbavit. Pochází totiž z násobení matic, které posouvají o 1 Kč, 2 Kč, ... Takovými maticemi jde ale násobit rychleji než v $\mathcal{O}(N^3)$, protože každý řádek obsahuje právě 2 nenulové prvky – nemusíme počítat celý skalární součin řádku a sloupce, stačí ze sloupce sečíst ty dva prvky, které chceme.

Tímhle krokem stranou jsme umlátili časovou složitost do $\mathcal{O}(N^2 \log K + N^3)$. Na první pohled vypadá zlověstněji než $\mathcal{O}(NK)$ (už jenom kvůli mocninám, v jakých se v ní vyskytuje N), ale pro $N = 250$, $K = 10^9$ vyjde podstatně lépe.

Pro úplnost ještě uveďme paměťovou složitost, i když na ní příliš nesejde. Sice počítáme $\log K + N$ matic velikosti $N \times N$, ale většinu z nich stejně zahodíme: budeme potřebovat jenom matici posouvající o $1, \ldots, K \bmod N$ a matici posouvající o $1, \ldots, N$. Vejdeme se tedy do $\mathcal{O}(N^2)$.

Program (C) – maticová varianta:
`http://ksp.mff.cuni.cz/viz/25-4-5-matice.c`

Michal Pokorný

[45] Kdybychom chtěli, můžeme rychlost násobení matic vylepšit, ale v téhle úloze to není potřeba.

[46] Viz strana 97 nebo `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

25-4-6 Triády

Kdo se do úlohy pustil, triády by hledal spolehlivě, pokud by však měl dost výpočetního času. Jen nemnozí řešitelé zvládli přijít na relativně rychlé řešení.

Jednoduché řešení za pár bodů se prostě podívá na každou trojici karet a ověří, jestli netvoří triádu. Takto dostaneme časovou složitost $\mathcal{O}(n^3k)$ a paměťovou $\mathcal{O}(nk)$. Faktor k ve složitosti je důležitý, neboť potřebujeme čas $\mathcal{O}(k)$ na ověření, jestli trojice tvoří triádu.

Základní myšlenka asymptoticky rychlejšího řešení nebyla těžká: podíváme se na každou dvojici karet, dopočítáme k nim, jak by měla vypadat třetí karta, a zkusíme ji vyhledat.

Základním pozorováním je, že pro danou dvojici karet máme jednoznačně určenu kartu, která s nimi může tvořit triádu. Pokud se totiž na jedné vlastnosti dané dvě karty shodují, musí mít stejnou hodnotu na této vlastnosti i třetí karta. Jestliže jsou na nějaké vlastnosti dvě karty různé, třetí karta musí mít tu jedinou hodnotu, kterou nemají dané dvě karty.

Nyní už zbývá jenom umět najít třetí kartu. Jedním z řešení je na začátku setřídit karty (stačí i kvadraticky). Pak pro každou dvojici binárně vyhledáme, kde by se třetí karta měla nacházet, a ověříme, jestli tam skutečně je. Ještě je potřeba doplnit ověření, že jsme našli skutečně novou kartu, pokud jsme dostali dvojici identických karet. Takto dosáhneme složitosti $\mathcal{O}(n^2 \log n \cdot k)$.

Ještě rychlejšího řešení dosáhneme pomocí písmenkového stromu neboli trie. (Všimněte si skryté a neplánované nápovědy, totiž podobnosti slov triáda a trie.) Nyní si trii stručně popíšeme, jejich podrobnější vysvětlení najdete v kuchařce o hledání v textu.[47]

Trie je zakořeněný strom, který se staví pro nějakou množinu slov v dané abecedě. Kořen odpovídá prázdnému slovu, synové kořene znakům, kterým začíná nějaké slovo, čili jednoznakovým prefixům. Pokud více slov začíná jedním znakem, syn s tímto znakem je jen jeden. V další úrovni stromu budou dvouznakové prefixy slov (prefix je souvislá část slova, která obsahuje začátek), ve třetí úrovni stromu budou tříznakové prefixy a tak dále.

Stavba trie probíhá tak, že se začne s kořenem a postupně se přidávají slova. Slovo přidáme jednoduše tak, že jdeme do vrcholů odpovídajícím aktuálnímu znaku slova. Pokud vrchol chybí, doplníme ho a přejdeme na další znak.

My použijeme trii na karty, které si můžeme představit jako slova o délce k v abecedě 1, 2, 3. Na začátku algoritmu tedy všechny karty naskládáme do trie. U každého listu v trii si navíc budeme pamatovat, kolik karet k němu náleží,

[47] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

abychom poznali, že tři karty jsou stejné. Pokud se v nějakém listu počet dostane na 3, hned ohlásíme triádu a můžeme skončit.

Pak pro každou dvojici karet dopočteme třetí a zkusíme ji vyhledat v trii. Uspějeme-li, máme triádu. Pokud se třetí karta neliší od karet z dané dvojice, nemusíme ji hledat, neboť identické karty jsme ošetřovali při stavbě trie. Díky tomu také nemusíme ověřovat, jestli jsme v trii našli skutečně novou kartu, tedy že jsme nenalezli jednu z karet z dané dvojice.

Hledání v trii zabere čas $\mathcal{O}(k)$, takže celková časová složitost je $\mathcal{O}(n^2k)$. V paměti se trie vejde do prostoru velikosti $\mathcal{O}(nk)$, neboť každá vlastnost každé karty vytvoří maximálně jeden nový vrchol. Paměťová složitost tedy je $\mathcal{O}(nk)$.

Umíte řešit úlohu asymptoticky rychleji, když k může být velké? Pak budeme rádi, když se s námi o řešení podělíte.

Mimochodem, pokud by k bylo zhruba logaritmicky velké oproti n (což dle zadání nebylo povoleno), vyplatilo by se karty skládat do k-dimenzionální krychle o hraně 3 a procházet všechny úsečky krychle, jež tvoří triádu. To už však přesahuje rámec tohoto řešení.

Pavel „Paulie“ Veselý

25-4-7 Šifrovací knoflíky

Úloha, v té verzi jak jsme ji zadali, se nakonec ukázala být o něco lehčí než jsme původně zamýšleli. Nejdříve ukážeme postup, jakým budeme knoflíky otáčet a pak ukážeme, že tento postup opravdu projde všechny možnosti a skončí opět v počáteční pozici.

Knoflíky si očíslujeme čísly $0, 1 \ldots, n-1$ a kroky otáčení si očíslujeme $1, 2, \ldots, n^k$.

Nejprve tedy postup otáčení. Celkový počet možností, které musíme navštívit, je n^k, a takový je i celkový počet otočení. Stačí tedy jen určit, kdy otáčíme kterým knoflíkem. V kroku i otočíme knoflíkem j takovým, že j je největší číslo, které splňuje $n^j \mid i$ (n^j beze zbytku dělí i).

Nyní nahlédneme, že platí následující dvě tvrzení:

1. Mezi dvěma otočeními knoflíku s číslem větším nebo rovným j se na knoflících $\{1, \ldots, j-1\}$ vystřídají všechny možné kombinace.
2. Po provedení n^k kroků budou všechny knoflíky v počátečních pozicích.

Tvrzení 1 dokážeme matematickou indukcí podle j. Pro $j = 0$ je to jasné, pro $j = 1$ si všimneme, že knoflík s číslem alespoň 1 se otočí každý n-tý krok a zbylých n kroků se otočí knoflík číslo 0. Tedy se na něm opravdu vystřídají všechny možnosti.

Nyní budeme předpokládat, že tvrzení platí pro $j-1$ a dokážeme, že platí pro j. Z podmínek pro otáčení vidíme, že mezi tím, co dvakrát otočíme knoflík s číslem alespoň j, otočíme $(n-1)$-krát knoflíkem $j-1$ a jelikož mezi každými těmito dvěma otočeními se nám na knoflících $0,\ldots,j-2$ vystřídají všechny možnosti, tak po $n-1$ opakování se nám vystřídají všechny možnosti na knoflících $0,\ldots,j-1$. A to jsme přesně chtěli.

Teď nám jen zbývá dokázat Tvrzení 2. Chceme ukázat, že počet otočení každého knoflíku je dělitelný číslem n. To dokážeme také indukcí, ale tentokrát budeme postupovat z druhé strany, od knoflíku s největším číslem. Ten se otočí pokaždé, když $n^{k-1} \mid i$, což se stane právě n-krát.

Nyní provedeme indukční krok. Předpokládáme, že knoflíky s čísly $k-1, k-2,\ldots,j+1$ skončí v počáteční pozici a ukážeme, že pak i knoflík s číslem j skončí v počáteční pozici. Knoflík j se otočí právě $(n^{k-j}-l)$-krát, kde l je počet otočení větších knoflíků. A jelikož víme, že počet otočení všech větších knoflíků je dělitelný číslem n, tak i počet otočení knoflíku j je dělitelný n. A máme vyhráno.

Na závěr se ještě podívejme na časovou složitost algoritmu. Otočení knoflíku provádíme celkem n^k-krát. Podmínky na dělitelnost budeme zkoušet postupně od nejnižšího j. Spočítáme, kolikrát kterou podmínku testujeme. První podmínku testujeme pokaždé, druhou podmínku jen pokud je splněna první, tedy n^{k-1}-krát. Všechny podmínky dohromady testujeme v čase $\sum_{i=0}^{k-1} n^{k-i} = \mathcal{O}(n^k)$.

V každém kroce vypíšeme jen číslo knoflíku, s kterým otáčíme. Časová složitost je tedy $\mathcal{O}(n^k)$. Lepší ani být nemůže, protože algoritmus vydává takto velký výstup.

Karel Tesař

Alternativní řešení

Pro každé n a k chceme najít $R_{n,k}$, posloupnost otáčení k knoflíků s n pozicemi takovou, že každou možnou konfiguraci projde právě jednou a z koncové konfigurace se lze jedním otočením dostat zpět do počáteční. To je jen drobná přeformulace zadání, kde poslední „návratový“ krok za součást řešení nepočítáme (ale víme, že jej lze udělat), což se nám bude za chvíli hodit, abychom mohli tato řešení skládat za sebe. Bez většího rozmýšlení je jasné, že pokud má $R_{n,k}$ projít všech n^k konfigurací, musí ji tvořit n^k-1 otočení.

Zvolíme si pevné n a budeme postupně (induktivně) konstruovat řešení $R_{n,k}$ pro jednotlivá k. Tedy nejdříve vytvoříme $R_{n,1}$ a potom ukážeme, jak z libovolného $R_{n,k}$ vyrobit $R_{n,k+1}$.

Pro situaci s jedním knoflíkem je řešení $(R_{n,1})$ zřejmé: prostě jím $(n-1)$-krát otočíme doprava. Takto určitě projdeme postupně všechny pozice a jedním (n-tým) otočením se můžeme vrátit zpět na začátek. Například pro $n=3$ dostaneme postupně pozice $0,1,2(,0)$.

Nyní chceme z $R_{n,k}$ vyrobit $R_{n,k+1}$. Rozdělíme si knoflíky na dvě skupiny: první (*hlavu*) a všechny ostatní (2 až $k+1$, *ocas*). Je asi jasné proč – ocas je dlouhý k, tedy na něm můžeme nějakým způsobem použít $R_{n,k}$ zděděné z indukce. Zkusíme začít tak, že budeme na ocas postupně aplikovat jednotlivé kroky $R_{n,k}$. Ukážeme si to na příkladu $k=1$. Pro něj dostáváme postupně konfigurace (BÚNO začínáme v $(0,0)$): $(0,0),(0,1),\ldots,(0,n-1)$. V tuto chvíli jsme prošli všechny konfigurace začínající nulou. Dále už nemůžeme pokračovat s ocasem, neb bychom se vrátili do již navštíveného stavu.

Budeme tedy postupovat podobně, jako bychom přičítali jedničku: provedeme jakýsi „přenos do vyššího řádu“ – tedy otočíme hlavovým knoflíkem. Při normálním sčítání bychom zároveň i vynulovali všechny řády ocasu (a dostali bychom v tomto případě konfiguraci $(1,0)$) a pokračovali přičítáním opět od nejnižšího řádu, dostávajíce tentokrát všechny konfigurace začínající jedničkou. Kdybychom tohle zopakovali celkem n-krát, dostaneme všechny konfigurace začínající postupně 0 až $n-1$, tedy úplně všechny.

Ale to nemůžeme, neb smíme otočit jen jedním knoflíkem, dostáváme tedy konfiguraci $(1,n-1)$. To ovšem vůbec nevadí! Díky tomu, že vše je cyklické, je úplně jedno, kde opětovné přičítání na nejnižším řádu začneme: pokud ho provedeme $(n-1)$-krát, vystřídá se na daném knoflíku $n-1$ různých hodnot, tedy opět projdeme každou konfiguraci, začínající tentokrát jedničkou, právě jednou. Následuje další přenos, dalších $n-1$ otočení, etc. Od sčítání se to liší jen tím, že prvky naší posloupnosti nebudou seřazeny vzestupně. Nejlépe to bude vidět na příkladu: $P_{3,2}$ vypadá takto (čteno po sloupcích):

```
00  12  21
01  10  22
02  11  20
```

Zkusme to nyní zapsat obecně. Označíme-li si jako H operaci „otoč hlavovým knoflíkem o jedna doprava“ a jako O operaci „proveď postupně všechny kroky $R_{n,k}$ na ocas“, pak bude $R_{n,k+1}$ vypadat takto:

$$\underbrace{O,H,O,H,\ldots,H,O}_{n\text{-krát } O,\ (n-1)\text{-krát } H}$$

Pro příklad $n=3$ a $k=2$ bude výsledná posloupnost otáčení

$$\underbrace{B,B,}_{O}\ \underbrace{A,}_{H}\ B,B,A,B,B.$$

Snadno ověříte, že opravdu vygeneruje posloupnost konfigurací v příkladu výše.

Takováto posloupnost splňuje všechny požadavky na $R_{n,k}$. Ukážeme, že to platí obecně. Operace O díky vlastnostem $R_{n,k}$, které máme zaručené z indukčního předpokladu, projde v n^k-1 krocích všech n^k možných konfigurací ocasu

(počítáme i počáteční a koncovou), bez ohledu na to, kterou začala. A to zopakujeme postupně pro všechny možné hodnoty hlavy, dostáváme tedy nejdřív všechny konfigurace začínající nulou, pak všechny začínající jedničkou, atd., dohromady tedy úplně všechny.

Co už je méně jasné je, že se z koncového stavu půjde dostat jedním otočením do počátečního. To nahlédneme takto: nejdříve ukážeme, že konfigurace ocasu bude na konci stejná jako na začátku. S ním hýbou jen operace O, kterých provedeme celkem n, přičemž všechny jsou stejné. Tedy pokud O otočí nějakým i-tým knoflíkem p_i-krát, celkem jím bude otočeno $n \cdot p_i$-krát, vrátí se tedy do původní pozice. A hlavovým knoflíkem otočíme celkem $(n-1)$-krát. Tedy pokud s ním otočíme ještě jednou, dostaneme se opravdu zpět do výchozí konfigurace, což jsme přesně chtěli.

Filip Štědronský

25-4-8 TEXgramy

Řešitelů utěšeně ubývá, ale stále je vás dost. Je radost číst řešení, která jdou k věci a dávají smysl. Nikdo není mimo, občas se objeví ukrutně komplikované řešení, ale nic moc hrozného. Až je to občas líto mému zlomyslnému já.

Tentokrát bylo správných přístupů habakuk a vzorové řešení je dlouhé, ukážeme si tedy pouze základní princip. Implementační detaily si prohlédnete ve vzorovém kódu.

Řešení **úkolu 1** bylo poměrně jednoduché. Bylo potřeba zavést si tři číselné registry, ve kterých jste si udržovali aktuální číslo nadpisu. Při vytváření nadpisu jste inkrementovali příslušný registr a případně vynulovali čítače nadpisů nižších úrovní.

Z estetického pohledu bylo potřeba vhodně nastavit mezery pod a nad nadpisem, včetně problémů typu: „Pokud se hned pod sebou sejdou dva nadpisy různých úrovní, tak mezi nimi nesmí být moc velká mezera.“

Taktéž se ve vzorovém řešení ošetřuje případ, kdy se pod sebou sejdou dva nadpisy stejné úrovně s jinak širokými čísly. Na začátku se změří šířka čísla 00, 00.00, resp. 00.00.00 a pak se číslo sází do hboxu fixní šířky, který je zprava doplněn pružným výplňkem.

Sazba obsahu v **úkolu 2** byl o něco větší oříšek. Použití `\immediate\write` nepřicházelo v úvahu, neboť TEX se může pokusit vložit příslušný nadpis ještě do předchozí strany, než přijde na to, že by bylo lepší dopustit se stránkového zlomu někde jinde. Pak by neseděla čísla stran v obsahu.

Naopak vůbec nebylo třeba sypat si do pomocného souboru čísla jednotlivých nadpisů – ta se přece dala vypočítat znovu při načítání obsahu stejným algoritmem.

Při vypisování obsahu se objevil jiný problém – před vložením obsahu bylo třeba přejít na novou stránku, jinak se do něj nezapsaly nadpisy z poslední strany. Bylo třeba také zavřít soubor s obsahem (`\closeout`), jinak se mohlo stát, že jste jej nevložili celý, ale jenom část, nebo dokonce prázdný (zbytek zůstal v zápisovém bufferu).

Sázení do více sloupců v **úkolu 3** nakonec nebylo tak zlé, jak se na první pohled zdálo. V makru `\multicolumn` se spočítá šířka sloupce, nastaví se podle toho `\hsize` a otevře vbox (`\setbox0\vbox\bgroup`). Primitivum `\bgroup` je definované jako `\let\bgroup{`.

Makro `\endmulticolumn` zavře box (`\let\egroup}`), rozseká box 0 na správně vysoké části (správná výška se určí vydělením celkové výšky počtem sloupců) a naskládá je vedle sebe do hboxu oddělené správně širokou mezerou.

A to je protentokrát vše. Těším se na vaše řešení páté série a přeju vám všem hezké jaro … konečně přišlo.

Program (TEX):
`http://ksp.mff.cuni.cz/viz/25-4-8.tex`

Jan „Moskyto“ Matějka

25-5-1 Cesta autobusem

Ukážeme si mírně inženýrské řešení. Vezmeme velmi zjednodušenou úlohu a budeme ji postupně opravovat, abychom se dostali k té složité.

Takže tedy od lesa. Co kdybychom nejezdili autobusem, ale chodili pěšky? To bychom nemuseli řešit, jakým směrem jsme otočení, takže by úloha byla jen obyčejný průchod do šířky, jak je popsaný například v grafové kuchařce.[48] Jak známo, to stihneme v $\mathcal{O}(n)$, kde n je počet políček plánu města.

Tak si úložku malinko ztížíme. Budeme jezdit mikrobusem – to bude autobus délky 1. Už musíme řešit otáčení, ale můžeme se otočit kdekoliv. A na vyřešení otáčení uděláme malý trik. Uděláme si dvě kopie plánu města a umístíme je na sebe – takže budeme mít jakýsi trojrozměrný prostor. V dolním patře budou sousedit políčka jen ve vodorovném směru, v horním patře jen ve svislém. A políčka nad sebou budou sousední vždy.

Všimněme si, že patra plánu odpovídají otočení autobusu. V dolním patře je autobus otočený vodorovně, v horním svisle. Přesun mezi patry odpovídá jeho otočení o $90\,^\circ$ (nebo $\pi/2$, pokud máte tyto jednotky radši).

V tomto dvoupatrovém bludišti tedy opět nalezneme cestu ze startu do cíle (nebo cílů – je jedno, jak budeme natočení v cíli, proto jsou obě políčka nad sebou cílová). Protože jsme zvětšili plán jen dvakrát, časová i paměťová složitost je stále $\mathcal{O}(n)$.

A už se dostáváme k lehčí variantě úlohy ze zadání. Budeme jezdit klasickým autobusem délky k. Pozici autobusu si budeme reprezentovat pozicí jeho levého horního konce. Z toho, ve kterém se nacházíme patře, poznáme, jestli autobus vede dolů nebo doprava. Tak to by byla téměř stejná úloha jako minule. Až na to, že ne všechna políčka nad sebou sousedí (a občas sousedí některá políčka, která nejsou nad sebou – viz např. obrázek v zadání, kde se otočením změní levý horní roh autobusu). Kdybychom ale věděli, jestli stojíme v rohu volného čtverce velikosti alespoň $k \times k$, neměli bychom nejmenší problém určit, jestli se zde otočit umíme, či nikoliv. Hledání čtverců ale odložme až na konec řešení.

Tak tedy, zlatý hřeb úlohy. Potřebujeme si v průběhu prohledávání pamatovat ještě poslední navštívenou zastávku (protože do ní nesmíme hned vjet znovu) a aktuální délku autobusu. No, pro pamatování tohoto si vytvoříme další „patra“ bludiště. Naše dvě patra z lehčí varianty zkopírujeme tolikrát, kolik je zastávek, a v každé zastávce se teleportujeme do stejného políčka v kopii pro danou zastávku. V této kopii se do ní již nesmí vjet (ale vyjet ano – již máme orientovaný graf). Obdobný trik uděláme pro délky autobusů – celé skupinky pater z minulého kopírování nakopírujeme pro každou délku autobusu a budeme teleportovat mezi nimi při každé změně délky.

[48] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

To je celé hrozně hezké, ale udělat všechny ty kopie by trvalo hrozně dlouho a zabralo zbytečně mnoho paměti. Proto si budeme jen „představovat“, že máme všechny tyhle kopie. Pozice autobusu v celé té naší struktuře bude reprezentovaná čtveřicí informací – pozicí na plánu, otočením, poslední navštívenou zastávkou a délkou. Ale plán budeme mít jen jeden. Jen to, která políčka v našem mnoharozměrném prostoru jsou sousední, budeme počítat podle celé čtveřice pokaždé, když budeme potřebovat sousední políčka některé pozice.

Hodilo by se umět spočítat, jaký největší čtverec vede z každého políčka doleva nahoru (pro zbylé 3 směry to bude obdobné, prostě stejný algoritmus pustíme vícekrát v různých směrech). To je ale pouze drobné cvičení na dynamické programování.[49]

Chceme spočítat velikost čtverce pro pozici (x, y). Pokud máme spočítané velikosti maximálních čtverců pro pozice $(x-1, y)$, $(x, y-1)$ a $(x-1, y-1)$, pak z těchto hodnot jednoduše spočítáme velikost našeho čtverce (vezmeme minimum z těch tří a přičteme jedničku – kdo nevěří, ten si to nakreslí). A abychom tyto pozice měli již spočítané, tak půjdeme po řádcích shora zleva.

A jak je to se složitostmi? Určitě potřebujeme $\Omega(n)$ času i paměti na načtení a uložení mapy a spočítání velikostí čtverců. Horní odhad je ale trochu horší. Určitě ale nepotřebujeme navštívit žádný stav dvakrát (tedy, každá čtveřice se ve frontě vyskytne maximálně jednou). Políček je n, možných délek autobusů $\mathcal{O}(n)$ a zastávek nechť je z (kde z je $\mathcal{O}(n)$). Orientace autobusu jsou jen dvě. Takže máme $\mathcal{O}(n^2 \cdot z)$ možných stavů.

Navíc, protože jsme si nevytvořili všechny kopie mapy, musíme si nějak pamatovat, které stavy jsme už navštívili. Pokud to uděláme např. ve stromu, zaplatíme za to ještě logaritmickým zpomalením, takže časová složitost by byla $\mathcal{O}(n^2 \cdot z \cdot \log n)$. Také bychom mohli místo stromu použít hešování a dostat složitost $\mathcal{O}(n^2 \cdot z)$ v průměru.

Lze očekávat, že na „slušně vychovaných“ mapách do takových extrémů nebudeme muset jít, ale jdou vytvořit i „neslušné“ mapy – například začneme s dlouhým autobusem a na jeho zkrácení o 1 je potřeba projet řádově n políček. A do cíle povede klikatá cesta, kterou projede jen kratičký autobus.

Taktéž, pokud bychom si vytvořili všechny ty kopie na začátku, zbavili bychom se onoho logaritmu. Ale za cenu toho, že by nám to vždy trvalo $\Omega(n^2 \cdot z)$. Tedy, musíme si rozmyslet, jestli čekáme spíše slušně vychované mapy, nebo ty neslušně vychované.

Program (Python):
`http://ksp.mff.cuni.cz/viz/25-5-1.py`

Michal „Vorner“ Vaner

[49] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

25-5-2 Telefonní ústředna

Úloha byla poměrně jednoduchou aplikací principu dynamického programování.[50] Pokud tento pojem slyšíte poprvé, doporučuji přečíst si naši kuchařku.

Kromě polí pro záznam a hovor si pořídíme ještě pole `stavy` o délce s (tedy stejně dlouhé jako hovor). Do něj si na i-tou pozici budeme ukládat zatím objevený počet prefixů hovoru délky i.

Postupně tedy procházíme pole záznamu odzadu. V každém kroku tohoto průchodu začneme procházet od začátku celé pole hovoru a vždy porovnáme bity. Pokud se j-tý bit hovoru shoduje s příslušným bitem záznamu, znamená to, že můžeme prodloužit všechny dosud nalezené prefixy hovoru délky $j-1$. To odpovídá přičtení proměnné `stavy[j-1]` k `stavy[j]` v případě, že $j > 1$, a přičtení jedničky v případě, že $j = 1$.

Kýžený výsledek, tedy počet způsobů, jak z bitů záznamu vybrat podposloupnost odpovídající hovoru, se nachází na konci algoritmu ve `stavy[s]`.

Rozmysleme si ještě, že pole záznamu potřebujeme procházet odzadu. Pakliže tak neučiníme a j-tý a $(j-1)$-tý bit hovoru bude stejný, zvýšíme nejdříve `stavy[j-1]` o $k > 0$ a pak `stavy[j]` o k víc, než bychom měli.

Pokud bychom za každou cenu chtěli hovor procházet odpředu, museli bychom použít ještě jedno pomocné pole, kam bychom si změny ukládali a vždy až na konci zpracování indexu záznamu tyto změny k poli `stavy` přičetli.

Nakonec ke složitostem. Paměťová je očividně lineární k délce záznamu a hovoru, tedy $\mathcal{O}(n+s)$. Ohledně časové pak vidíme, že pro každý bit záznamu procházíme celý hovor, tedy výsledná časová složitost je $\mathcal{O}(ns)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-5-2.cpp`

Jan Bok

25-5-3 Špagety

Úloha se špagetami vás očividně zaujala, přišlo na ní skoro nejvíce řešení páté série (jen o jedno řešení méně, než kolik měla „nejoblíbenější“ úloha série 25-5-2).

Ve vašich řešeních se objevovaly dva přístupy, které nakonec vedly skoro k tomu samému. Jedním z nich bylo vytvořit si ze špaget graf. Z podlouhlých špaget se nám stanou vrcholy (z každé špagety jeden) a orientované hrany v tomto grafu natáhneme tam, kde jedna špageta leží přímo na druhé (tedy pokud ve

[50] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

směru gravitační osy jsou dílky špaget přímo nad sebou, nebo je mezi nimi jen prázdné místo).

Pokud do nějaké špagety ještě povedou vstupní hrany, znamená to, že nad ní ještě něco leží a nemůžeme ji tedy odebrat. Navíc si uvědomme, že nám stačí tyto hrany natáhnout vždy jen mezi přímo sousedícími špagetami. Pokud nad sebou totiž leží více špaget, tak nejspodnější špagetu zajímá jen špageta těsně nad ní a je jí jedno, jestli je to jediná špageta, která ji blokuje, nebo je jich nad ní více.

Pak již jen můžeme najít špagety, které nejsou pokryté žádnou hranou (mají vstupní stupeň nula) a postupně všechny odebereme. Během odebírání každé ze špaget zrušíme i příslušné hrany a současně se díváme, jestli jsme nezískali novou volnou špagetu. Pokud ano, vložíme ji do pomocného zásobníku. To je jeden krok.

V dalším kroku vezmeme pomocný zásobník a odebereme všechny špagety, které jsou v něm. Tím nám opět vzniknou nové volné špagety a takto budeme pokračovat, dokud nevyprázdníme talíř, nebo dokud to dál nepůjde.

Druhým přístupem, který v důsledku vede na stejnou strukturu, jen je v něm graf více schovaný, je procházet talíř špaget postupně po sloupcích a k oindexovaným špagetám si ukládat počet špaget, kterými je tato špageta přímo zakryta. Při odebírání se pak tento čítač snižuje a řešení tak funguje stejně, jako výše popsané.

K vypočítání složitosti si označme jako N počet špaget, jako M počet hran mezi nimi a jako V objem talíře. Je jasné, že $N, M \leq V$, protože maximálně můžeme mít jednotkovou špagetu na každém políčku talíře. Může nám také nastat situace, kdy bude hran řádově N^2 (představte si v jedné vrstvě $N/2$ rovnoběžných špaget a pod nimi stejnou vrstvu, jen zrotovanou). Paměťová složitost je tedy $\mathcal{O}(N + M)$, ale jelikož N a M nevíme na vstupu, je asi korektnější odhad udělat jako $\mathcal{O}(V)$.

Časová složitost je pak stejná, protože na každou špagetu se podíváme při výpočtu maximálně jednou a po každé hraně se vydáme také jen jednou, což je zase $\mathcal{O}(N + M)$ na výpočet. Na vytvoření grafu ale určitě potřebujeme projít celý talíř (navíc ho musíme i načíst), tedy časová složitost je také $\mathcal{O}(V)$.

Hodně štěstí i při dalších pokusech nepokecat se špagetami!

Program (C):
`http://ksp.mff.cuni.cz/viz/25-5-3.c`

Jirka Setnička

25-5-4 Výslechy

Niektorí z vás si správne všimli ako úlohu previesť na grafovú. Predstavme si, že na vstupe dostaneme výrok typu „A tvrdí, že B je mafián“. Čo všetko vieme povedať o A a B? Ak A je zamestnanec (budem značiť A_1), tak B bude určite mafián. Ak je zas B mafián, tak A musí byť zamestnanec (mafián by klamal a netvrdil, že B je mafián). Je jednoduché podobne odvodiť, že ak A je mafián (značím A_0), tak B je zamestnanec a ak B je zamestnanec, tak A je mafián. Inými slovami, pri tomto výroku platí

$$A_1 \Leftrightarrow B_0, \quad A_0 \Leftrightarrow B_1.$$

V prípade výroku typu „A tvrdí, že B nie je mafián“ rovnakým postupom odvodíme, že platí

$$A_1 \Leftrightarrow B_1, \quad A_0 \Leftrightarrow B_0.$$

Zo vstupu teda môžeme vyrobiť graf. Za každú novú osobu X pridáme do grafu dva vrcholy, a to X_1 a X_0. Je asi zrejmé, ako budú vyzerať hrany v tomto grafe. V prípade výroku „A tvrdí, že B je mafián“ spojíme hranou vrcholy A_1, B_0 a vrcholy A_0 a B_1. Podobne pri výroku „A tvrdí, že B nie je mafián“ spojíme hranou vrcholy A_1, B_1 a vrcholy A_0, B_0.

K čomu nám pomôže takýto graf? Je vidieť, že ak nastane situácia, že existuje vrchol X_0, ktorý je spojený nejakou cestou s vrcholom X_1, tak práve vtedy si musel niekto protirečiť a počet možností je 0. Ak totiž vyjdeme z nejakého vrcholu a dostaneme sa do iného, tak to znamená, že z výpovedi človeka A dokážeme vyvodiť, že či X je mafián alebo nie podľa toho, či sme vo vrchole X_1 alebo X_0.

Ak situácia z predchádzajúceho odstavca nenastane, tak počet možností je $2^{k/2}$, kde k je počet komponent súvislosti grafu. Všimnime si, že pre každú komponentu je navyše v grafe ešte aj jej negácia. Komponenta súvislosti v našom grafe nám vlastne hovorí nasledovné. Predstavme si, že v nejakej komponente sa nachádza vrchol X_1. Ak poviem, že X je zamestnanec, tak som rozhodol o všetkých vrcholoch v komponente, že či sú zamestnanci alebo mafiáni. Negácia komponenty nám dáva druhú možnosť.

Ak by sme riešenie chceli naprogramovať, tak po skonštruovaní grafu nájdeme prehľadávaním do hĺbky (alebo do šírky) komponenty grafu a počas prehľadávania môžme kontrolovať, že či sa X_0 a X_1 nenachádza v tej istej komponente, pre každé X. Časová zložitosť je $\mathcal{O}(n + m)$, kde $2n$ je počet vrcholov grafu a m je počet hrán grafu. Pri tomto odhade predpokladám, že násobenie má zložitosť $\mathcal{O}(1)$. Všimnite si, že počet možností je v najhoršom prípade exponenciálny, napríklad keď nedostaneme na vstupe žiadne výroky.

Peter Zeman

25-5-5 Úklid trávníku

Označme si celkový počet balíků N (pro účely teoretického rozboru, toto číslo pochopitelně nesmíme v algoritmu použít), jednotlivé balíky $1, \ldots, N$ a k velikost vybíraného vzorku. Budeme předpokládat, že umíme generovat náhodná celá čísla z rozsahu $1, \ldots, m$ v konstantním čase pro libovolné $m \leq N$. Dále předpokládáme, že $N \geq k$ (jinak by úloha vůbec neměla smysl).

Řešení pro $k = 1$

Zařídíme si *odkladiště* (celočíselnou proměnnou), na kterém si budeme uchovávat jeden jakýsi „prozatímně vybraný“ balík z dosud zpracované části vstupu. Algoritmus potom bude pracovat takto: na začátku umístí do odkladiště první vstupní balík a poté postupně prochází všechny další v pořadí, v jakém přicházejí na vstupu. U každého se bude muset rozhodnout, zdali jej umístit do odkladiště (a tedy jím původní vybraný balík nenávratně nahradit), nebo zahodit. Balík, který v odkladišti zůstane po zpracování celého vstupu, označíme za hledaný vzorek.

Budeme postupovat induktivně: chceme, aby na konci každého kroku měly všechny zatím zpracované balíky stejnou pravděpodobnost nacházet se v odkladišti. Pokud toto zajistíme, pak už určitě na konci budou mít všechny balíky stejnou pravděpodobnost výběru.

Na začátku to určitě platí – máme jediný balík, který se v odkladišti nachází s pravděpodobností 1. Nyní předpokládejme, že už máme prvních $n-1$ balíků zpracovaných ($n \geq 2$) a dle indukčního předpokladu se každý z nich nachází v odkladišti s pravděpodobností $1/(n-1)$. Na konci kroku by se měl každý, tedy i nově přidaný, balík objevit na odkladišti s pravděpodobností $1/n$. Nezbývá nám tudíž nic jiného, než tam nový balík s pravděpodobností $1/n$ umístit. Zbývá ověřit, že dostaneme správnou pravděpodobnost i u ostatních balíků (1 až $n-1$). Každý z nich má pravděpodobnost $\frac{1}{n-1} \cdot \frac{n-1}{n} = 1/n$ obsazovat na konci n-tého kroku odkladiště (musel se tam nacházet v předchozím kroku a museli jsme se rozhodnout nenahradit jej n-tým).

Obecné k

Pro obecné k budeme postupovat obdobně, jen na odkladišti (nyní poli délky k) budeme skladovat prozatímně vybranou k-*tici* balíků. A pochopitelně budeme chtít, aby na konci n-tého kroku měly všechny možné k-tice ze zatím načtených balíků stejnou pravděpodobnost obsazovat odkladiště.

Indukci začneme až od k-tého balíku, kdy umístíme do odkladiště rovnou všechny balíky 1 až k – mezi nimi existuje jediná k-tice s pravděpodobností 1, takže naše podmínka je určitě splněna. Nyní předpokládejme, že už máme prvních $n-1$ balíků ($n \geq k+1$) zpracovaných a dle indukčního předpokladu se v odkladišti nachází náhodná k-tice z $\{1, \ldots, n-1\}$.

Nyní zpracováváme n-tý balík a chtěli bychom získat náhodnou k-tici z množiny $\{1,\ldots,n\}$. Libovolná k-tice z $\{1,\ldots,n\}$:

a) Buď obsahuje n. Pak je tvořena $(k-1)$-ticí z $\{1,\ldots,n-1\}$ rozšířenou o n. Každé takové $(k-1)$-tici odpovídá právě jedna k-tice obsahující n a naopak, což má dva příjemné důsledky: (1) celkem jich je stejně, $\binom{n-1}{k-1}$, a (2) náhodnou k-tici obsahující n si můžeme pořídit tak, že vezmeme náhodnou $(k-1)$-tici z $\{1,\ldots,n-1\}$ a přidáme k ní balík n. Předkládáme k intuitivnímu uvěření, že náhodnou $(k-1)$-tici získáme z náhodné k-tice z $\{1,\ldots,n-1\}$ (kterou máme z indukčního předpokladu) zahozením náhodného prvku.

b) Anebo neobsahuje n. Pak ale není ničím jiným než k-ticí z $\{1,\ldots,n-1\}$. Tedy i náhodná k-tice neobsahující n je prostě jen náhodná k-tice z $\{1,\ldots,n-1\}$. A hle, jednu takovou máme z předchozího kroku!

Už umíme vygenerovat náhodnou k-tici obsahující a neobsahující n, teď bychom chtěli tyto výsledky dát dohromady. Pravděpodobnost, že náhodná (libovolná) k-tice obsahuje n, je rovna

$$\frac{\text{počet } k\text{-tic obsahujících } n}{\text{počet všech } k\text{-tic}} = \frac{\binom{n-1}{k-1}}{\binom{n}{k}} = \frac{k}{n}.$$

Tedy náš algoritmus by měl s pravděpodobností k/n vygenerovat náhodnou k-tici obsahující n a s pravděpodobností $(n-k)/n$ náhodnou k-tici neobsahující n.

Tím jsme tedy vlastně (přinejmenším neformálně) dokázali správnost následujícího postupu v n-tém kroku:

1. Vygenerujeme náhodné číslo $r \in \{1,\ldots,n\}$.
2. Pokud $r \leq k$ (nastane s pravděpodobností k/n):
3. Vygenerujeme náhodné číslo $a \in \{1,\ldots,k\}$
4. Nahradíme a-tý (tedy náhodný) balík na odkladišti balíkem n (vygeneruje náhodnou k-tici obsahující n).
5. Jinak (s pravděpodobností $(n-k)/n$):
6. Ponecháme odkladiště beze změny a n-tý balík zahodíme (a máme náhodnou k-tici neobsahující n).

Nyní si všimněme, že 3. bod vůbec není potřeba. V nahrazovací větvi máme zaručeno, že $r \in \{1,\ldots,k\}$ a všechny tyto hodnoty mají stejnou pravděpodobnost (r jsme generovali rovnoměrně náhodně). Tedy můžeme prostě vyhodit z odkladiště r-tý balík.

Tato na první pohled technická drobnost nám umožní se na řešení podívat ještě úplně jinak. Ale o tom až za chvíli, nejprve se podíváme na program a předvedeme formální důkaz správnosti našeho algoritmu.

```
from random import randint

def vyber(baliky, k):
    buf = []
    n = 0
    for balik in baliky:
        n += 1
        if n <= k:
            buf.append(balik)
        else:
            nahrad = randint(1, n)
            if nahrad <= k:
                buf[nahrad - 1] = balik
    return buf
```

Formální důkaz

Ukážeme si jen indukční krok, zbytek je stejný jako u neformálního důkazu. Uvažujme libovolnou k-tici P z $\{1, \ldots, n\}$. Chceme ukázat, že se na odkladišti bude po n-tém kroku nacházet s pravděpodobností $1/\binom{n}{k}$. Rozebereme dva případy:

a) $n \notin P$. Pak P pochází z předchozího kroku, kde se vyskytla s pravděpodobností $1/\binom{n-1}{k}$ (z indukčního předpokladu), a aktuální krok přežila s pravděpodobností $(n-k)/n$ (rozhodli jsme se n-tý balík zahodit). Tedy pravděpodobnost výskytu P po n-tém kroku je

$$\frac{1}{\binom{n-1}{k}} \cdot \frac{n-k}{n} = \frac{1}{\frac{n}{n-k} \cdot \frac{(n-1)\cdot\ldots\cdot(n-k)}{k!}} = \frac{1}{\binom{n}{k}},$$

což jsme přesně chtěli.

b) $n \in P$. Pak P vznikla z nějaké k-tice Q nahrazením nějakého balíku x za n, tedy $P = Q \backslash \{x\} \cup \{n\}$ pro libovolnou z $n-1-(k-1) = n-k$ možných voleb x. Pravděpodobnost, že z daného Q vznikne P, je $\frac{k}{n} \cdot \frac{1}{k}$ (musíme se rozhodnout nahrazovat a vybrat k nahrazení právě x). A pravděpodobnost, že jsme v minulém kroku skončili s jedním z $n-k$ možných Q, je $(n-k)/\binom{n-1}{k}$. Celková pravděpodobnost, že po tomto kroku dostaneme P, je tedy

$$\frac{n-k}{\binom{n-1}{k}} \cdot \frac{k}{n} \cdot \frac{1}{k} = \frac{1}{\frac{n}{n-k} \cdot \binom{n-1}{k}} = \frac{1}{\binom{n}{k}}.$$

A tím je důkaz správnosti hotov. Na konci algoritmu pak budou mít všechny k-tice stejnou pravděpodobnost $1/\binom{N}{k}$ a máme požadovaný výstup. Časová složitost algoritmu je $\mathcal{O}(N)$ a paměťová $\mathcal{O}(k)$.

Alternativní řešení

Náhodnou k-tici můžeme vygenerovat také tak, že vygenerujeme náhodnou permutaci balíků a z ní vezmeme prvních k prvků. To zní trochu zblátadoloužnicky – když nemůžeme použít ani samotné N, kde bychom přišli k takové permutaci? Inu, opět inkrementálně. Budeme chtít na konci n-tého kroku držet v ruce náhodnou permutaci prvků $\{1, \ldots, n\}$. Začneme s triviální permutací obsahující pouze první balík a v každém kroku ji budeme chtít rozšířit o jeden prvek. Nyní předpokládejme, že už máme náhodnou permutaci prvků $\{1, \ldots, n-1\}$.

Chceme vygenerovat náhodnou permutaci na $\{1, \ldots, n\}$. Všimneme si, že prvek n se může vyskytovat stejně pravděpodobně na všech pozicích a že pokud ho prohodíme s prvkem na n-té pozici, dostaneme na konci n a před ním náhodnou permutaci na $\{1, \ldots, n-1\}$. Lze to ale udělat i opačně: vzít náhodnou permutaci na $\{1, \ldots, n-1\}$, na konec přidat n a pak ho prohodit s náhodně vybraným prvkem.

Tedy náhodné permutace vyrábět umíme. Teď nám ještě život komplikuje fakt, že bychom na uložení takovéto permutace potřebovali $\mathcal{O}(N)$ paměti, což si určitě nemůžeme dovolit (kdybychom mohli, prostě si do ní načteme celý vstup, spočítáme délku a vzorky vybereme přímo). My si ovšem celou permutaci pamatovat nepotřebujeme – zajímá nás jen prvních k prvků a všimneme si, že při žádné z úprav nepřenášíme informaci z jednoho místa permutace na jiné. K tomu, abychom určili, jak se změní počáteční úsek permutace, nám stačí znát ten a nově přidávaný prvek. Všechny zásahy do prvků od $(k+1)$ dál můžeme prostě z programu vyházet. Zbude algoritmus nápadně podobný předchozímu:

```
from random import randint

def vyber(baliky, k):
    buf = [-1] * k
    n = 0
    for balik in baliky:
        n += 1
        # Zvolíme místo pro přidávaný prvek
        r = randint(1, n)
        if r <= k:
            # Prohození popisované v řešení:
            # provedeme jen ty části, které
            # zasáhnou prvních k prvků.
            if n <= k:
                buf[n] = buf[r]
            buf[r] = balik
```

Uložený začátek permutace odpovídá našemu odkladišti, a náhodná místa, na která prohazujeme nově přidané prvky, jsou přesně náhodná $r \in \{1, \ldots, n\}$,

která generujeme v k-ticovém algoritmu. Tato verze se liší vlastně jen tím, že v k-tém kroku začíná s náhodnou permutací balíků $\{1, \ldots, k\}$ namísto uspořádaného pořadí. Ukážeme, že je to úplně jedno.

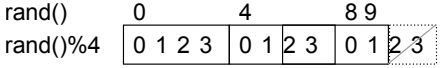
Představme si, že permutačnímu algoritmu v k-tém kroku prostě „pod rukama“ nahradíme náhodnou k-prvkovou permutaci za $(1, 2, \ldots, k)$ a zajímá nás, jaký to bude mít vliv na jeho další průběh. Určitě to nijak nezmění pozice balíků s číslem větším než k – ty jsou vybírány později a nezávisle. Pozice prvních pár balíků se určitě změní, leč nás nezajímají přesně – zajímá nás jen to, které z nich zůstanou na prvních k místech (bez ohledu na to, kde konkrétně). I to se pochopitelně změní: např. pro $k = 2, N = 3$, pokud jsme původně po k-tém kroku měli permutaci $(2, 1)$ a poslední balík nahradíme třetím, dostaneme na konci k-tici $\{2, 3\}$, kdežto pokud ji nahradíme setříděným pořadím $(1, 2)$, skončíme na konci s $\{1, 3\}$. Podívejme se ale na to, jak postupně nahrazujeme balíky $\{1, \ldots, k\}$ v počátečním úseku jinými. Při každé úpravě permutace mají všechny „přeživší“ z prvních k balíků stejnou pravděpodobnost být odsunuty na konec – jinými slovy, permutační algoritmus balík k vyřazení vybírá náhodně. A pokud na začátku prvních k balíků nějak jednorázově přeuspořádáme, nebudou tyto výběry o nic méně náhodné (spíše k intuitivnímu nahlédnutí, poctivý důkaz by dal trochu práce, klíčovým je fakt, že naše přeuspořádání a tyto náhodné výběry jsou *nezávislé*).

Pokud z permutačního algoritmu odstraníme prvních k kroků a začneme $(k + 1)$-ním s permutací $(1, 2, \ldots, k)$, ze všech prohození už se stanou přiřazení (vždy bude alespoň jeden prvek ležet mimo prvních k) a dostáváme algoritmus naprosto identický s původním k-ticovým. Jen jsme k němu přišli tak trochu z jiné strany.

Poznámky

1) Všichni řešitelé až na jednoho považovali za dostatečné dokázat, že každý *balík* má stejnou pravděpodobnost objevit se na výstupu. To ale ještě vůbec *neznamená,* že každá *k-tice* bude mít stejnou pravděpodobnost. Např. pro $k = 2$ a $N = 4$ a algoritmus, který vrací se stejnou pravděpodobností dvojice $\{1, 2\}$ a $\{3, 4\}$ se každý balík vyskytuje právě v jedné ze dvou možných dvojic, objeví se tedy na výstupu se stejnou pravděpodobností $1/2$ (dokonce „tou správnou“, neb $k/N = 1/2$). Ovšem určitě není pravda, že by všechny dvojice ze čtyřech balíků měly stejnou pravděpodobnost být výstupem. Ba co víc, většinu (4) jich algoritmus vůbec vygenerovat neumí! Místo šesti dvojic se stejnou pravděpodobností $1/6$ generuje dvě s pravděpodobností $1/2$ a čtyři s nulovou! A jedno takové řešení nám opravdu přišlo. Samozřejmě chápeme, že to s teorií pravděpodobnosti na středních školách nebude nikterak slavné, pročež jsme řešením, která generovala k-tice rovnoměrně, ale pořádně to o sobě nedokázala (většina došlých), strhávali jen jeden bod.

2) Někteří řešitelé používali ke generování náhodných čísel z rozsahu $0 \ldots m-1$ ve svých zdrojácích konstrukci `rand() % m`, kde `rand()` je funkce vracející náhodná čísla z nějakého fixního velkého rozsahu $0 \ldots M-1$, typicky daného maximálním rozsahem celočíselného datového typu (M je 2^{31} či 2^{63} pro Céčkový `int`), a `%` operátor modula. Pro m nesoudělné s M nebude výsledek takovéhoto výrazu úplně rovnoměrně náhodný, jak naznačuje pro příklad $M = 10$, $m = 4$ obrázek níže:



Vidíme, že čísla 0 a 1 mají větší pravděpodobnost (3/10) než ostatní (2/10), neboť když rozdělíme interval $0 \ldots 9$ na úseky délky 4, objeví se i v posledním neúplném. Obecně mají čísla menší než $M \bmod n$ pravděpodobnost $\lceil M/m \rceil$ a ostatní $\lfloor M/m \rfloor$. Tyto pravděpodobnosti budou stejné právě tehdy, když M/m je celočíselné. Obecně to, jak moc velká nerovnoměrnost bude, záleží na poměru $(M \bmod m)/M$. Za tuto technickou drobnost jsme pochopitelně nestrhávali žádné body, ale pokud někdy budete programovat něco hodně závislého na rovnoměrně náhodných číslech, je dobré mít to na paměti.

Dalo by se to obejít tak, že pokud nám „padne“ 8 nebo 9, tento výsledek zahodíme a zkusíme to znovu (i opakovaně). Ale to není předmětem této úlohy. Bylo naprosto oprávněné předpokládat, že umíme generovat náhodná čísla z daného rozsahu v konstantním čase. Pokud na to váš oblíbený jazyk nemá žádnou funkci, můžete si nějakou vymyslet a v kódu tento fakt okomentovat (programy jen čteme, obvykle se je nesnažíme spouštět). V případě této úlohy by bylo asi nejjednodušší prostě místo zdrojáku poslat pseudokód.

Martin Mareš & Filip Štědronský

25-5-6 Dělení dortu

Přímočarým řešením je vyzkoušet všechny možné průběhy hry a z nich vybrat ten nejlepší. V prvním tahu je na výběr N způsobů jak táhnout, v následujících $N-2$ tazích jsou způsoby právě dva. Celkem tedy máme $N2^{N-2}$ možných scénářů hry. Získali bychom tedy řešení s exponenciální časovou složitostí.

Chceme-li se zbavit exponenciály, stojí za zamyšlení otázka, zda něco nepočítáme zbytečně. Opravdu nás pro získání výsledku zajímají všechny možné průběhy hry?

Povšimněme si následujícího – jedinou informací o pozici hry, která má vliv na volbu dalšího tahu je aktuální nesnědený úsek dortu. Pro každý úsek délky 1 až $N-1$ existuje N pozic, kde tento úsek začíná. Úsek reprezentující celý dort je pouze jeden, ale bude se nám pro naše řešení hodit uvažovat jej jako N různých úseků podle toho, kde pomyslně začíná. Uvažujeme tedy N^2 úseků.

Pro každý úsek zjistíme, kolik nejvýše může hráč získat, pokud na tomto dílku táhne jako první, a kolik, pokud táhne jako druhý.

Pro úseky délky 1 ukořistí první hráč dílek odpovídající tomuto úseku a druhý nic. Pro úsek délky ℓ začínající na dílku i a končící na dílku j má hráč na tahu 2 možnosti – sníst dílek na pozici i, nebo j. V prvním případě bude zisk s_i (velikost i-tého dílku) plus zisk druhého hráče na úseku délky $\ell - 1$ končícím na pozici j, v druhém případě bude zisk s_j plus zisk prvního hráče na úseku délky $\ell - 1$ začínajícím na pozici i.

Řešením úlohy pak bude největší hodnota z možných zisků na dílcích délky N. Hodnoty zisků pro jeden dílek naznačeným způsobem spočteme v konstantním čase, časová složitost postupu tak bude $\mathcal{O}(N^2)$. Všimněme si, že si stačí v průběhu řešení pamatovat pouze zisky pro úseky délky $\ell - 1$ a ℓ, pak získáme paměťovou složitost $\mathcal{O}(N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/25-5-6.c`

Lukáš Folwarczný

25-5-7 Policejní koridor

K zadaniu tejto úlohy bola priložená kuchárka o tokoch v sieťach. Malo vám to napomôcť, že úlohu treba riešiť pomocou tokov. Toho ste sa skoro všetci chytili a použili kuchárkový Ford-Fulkersonov (F-F) algoritmus na vyriešenie ľahšej varianty. Viacerí z Vás ste ale vo svojich riešeniach zabudli napísať alebo ste nedôsledne čítali, že F-F algoritmus dostane kapacity na hranách a nie vo vrcholoch. F-F algoritmus si normálne s kapacitami vo vrcholoch neporadí, a preto je najprv potreba graf upraviť a previesť kapacity z vrcholov na hrany.

Ale poporiadku. Najprv sa pozrieme, ako sa dala vyriešiť ľahšia varianta. Úlohou bolo zistiť, či sa dá v neohodnotenom neorientovanom grafe zo štartu (vrchol A) dostať do cieľa (vrchol C) cez sklad (vrchol B) s tým, že niektoré vrcholy môžeme navštíviť iba raz (tie, na ktorých je polícia).

Hneď na začiatok si môžeme všimnúť, že nemá význam ľubovoľným vrcholom prechádzať viac ako dva krát (ak je to možné). Ak sme navštívili nejaký vrchol tretí krát, znamená to, že sled, ktorým prechádza ten vrchol, má dve slučky. Vrchol B potrebujeme navštíviť iba raz, a teda jedna z tých slučiek bude určite zbytočná. Môžeme ju zo sledu vyhodiť a vrchol navštívime už iba dva krát. Z toho vyplýva, že každý vrchol nám stačí navštíviť maximálne dva krát a neprídeme tým o žiadne riešenie.

Keďže naším cieľom je na túto úlohu použiť F-F algoritmus, musíme sa najprv vysporiadať s policajtmi vo vrcholoch. Vrcholy podrozdelíme. Každý takýto vrchol nahradíme dvojicou nových. Do prvého nového vrcholu budú viesť všet-

ky hrany, čo viedli do pôvodného vrcholu, z druhého nového vrcholu budú viesť hrany, ktoré viedli z pôvodného vrcholu. Tieto dva nové vrcholy spojíme hranou.

Tým sa naša úloha nezmenila, iba sme obmedzenie na navštívenie vrcholov presunuli na hrany. Počet hrán sa zvýšil maximálne o počet vrcholov, čo nám nič nezhorší. Keďže graf v zadaní bol neorientovaný a F-F algoritmus pracuje s orientovanými hranami, každú (neorientovanú) hranu nahradíme dvojicou orientovaných.

Ďalej si vytvoríme nový vrchol (označme D) a spojme ho s A a C. Na obe hrany, ktoré z vrcholu D vedú, postavíme policajtov. Potom môžeme našu úlohu preformulovať na nájdenie dvoch ciest z vrcholu B do D s tým, že môžeme prejsť každou hranou, na ktorej je policajt, maximálne raz. Keďže na hranách C-D a A-D sú policajti, nie je možné, aby obe cesty viedli po spoločnej jednej hrane do D.

Teraz už môžeme použiť F-F algoritmus. Kapacita hrany nám bude udávať, koľko krát cez ňu môžeme prejsť. Hrany, na ktorých sú policajti, budú mať kapacitu 1. Keďže hľadáme práve dve cesty medzi A a D, kapacita väčšia ako 2 nemá význam (aj z toho dôvodu, že každý vrchol nám stačí navštíviť max. dva krát). Za zdroj zoberieme vrchol B, za stok vrchol D.

Spustíme F-F algoritmus a ten nám vráti maximálny tok v tom grafe. Ak bude tok nulový, znamená to, že neexistuje žiadna (zlepšujúca) cesta z B do D, a teda z B sa nevieme dostať ani do A a ani do C. V tom prípade úloha nemá riešenie. Ak tok bude veľkosti jedna, znamená to, že existuje nejaká cesta z B do D, ale neexistujú tie cesty dve, ktoré by spĺňali podmienku policajtov. Teda z B sme sa nevedeli dostať buď do A, alebo do C, keďže na oboch hranách pred vrcholom D sú policajti. Ani v tomto prípade riešenie zjavne neexistuje.

A posledný prípad, ktorý môže nastať, je tok veľkosti 2. Väčší už byť nemôže, lebo súčet kapacít hrán, ktoré vedú do stoku, je 2. Ak nám F-F našiel tok veľkosti 2, znamená to, že každou hranou, na ktorej je policajt, prechádzame maximálne raz a máme dva sledy z B do D, ktoré sú disjunktné na hranách s policajtmi. Teda nutne existuje sled z A do B a aj sled z B do C a pritom platí, že každá hrana, na ktorej sú policajti, sa vyskytne v sledoch A-B a B-C maximálne raz. Teda v tomto prípade, keď existuje tok o veľkosti 2, úloha má riešenie.

Ak tieto dva sledy chceme vypísať, môžeme použiť rovnaký algoritmus ako na vypísanie hranovo disjunktných ciest, ktorý je popísaný v kuchárke.

Úloha sa dá aj priamo previesť na hľadanie hranovo disjunktných ciest. Zdroj a stok zoberieme rovnaký ako vyššie a podrozdelíme tentokrát každý jeden vrchol, čím opäť presunieme políciu na hrany. Hrany, na ktorých polícia nestojí, zdvojíme a potom postavíme políciu na každú jednú hranu. Zdvojením sme zabezpečili, že sa medzi vrcholmi dá prejsť aj dva krát a teda pridaním polície

na každú hranu nič nepokazíme. A teraz, keď už budú mať všetky hrany svojho policajta, môžeme im nastaviť kapacitu 1 a použiť kuchárkový algoritmus na hľadanie hranovo disjunktných ciest. V tomto prípade potrebujeme dve hranovo disjunktné cesty zo zdroju do stoku, čo bude ekvivalentné s existenciou riešenia ľahšej varianty úlohy.

Pozrime sa ešte na časovú zložitosť. Treba nám nájsť tok o veľkosti 2. To znamená, že potrebujeme spraviť max. dve iterácie F-F algoritmu. Teda nájsť max. dve zlepšujúce cesty. Nájsť zlepšujúcu cestu vieme prehľadaním do šírky. Teda časová zložitosť celého algoritmu bude rovnaká ako jedna iterácia F-F a bude to lineárne od počtu hrán a vrcholov. Pamäťová zložitosť taktiež.

Poďme sa teraz pozrieť na ťažšiu variantu tohto príkladu. V nej bolo potrebné zo všetkých ciest z A do C cez B nájsť tú najkratšiu, ktorá by spĺňovala podmienku policajtov. Ako prvé by nás mohlo napadnúť použiť rovnaký algoritmus ako pri ľahšej variante a upraviť hľadanie zlepšujúcich ciest tak, aby sme vyberali vždy tú najkratšiu.

Vyzerá, že by to mohlo fungovať, ale je v tom háčik. Niekedy pri hľadaní zlepšujúcej cesty sa nám oplatí tlačiť tok po nejakej hrane spať. A teda nemôžeme hľadať zlepšujúcu cestu, ktorá bude najkratšia v počte hrán. Môže existovať nejaká zlepšujúca cesta s väčším počtom hrán, ktorá bude na veľkom počte hranách tlačiť tok spať. Po pridaní takejto zlepšujúcej cesty už nebude po tých hranách nič tiecť, a teda tie hrany, po ktorých sme tlačili tok späť, nechceme započítať do dĺžky výslednej cesty medzi A-C (pretože po nich nakoniec nepôjdeme). Ba naopak potrebujeme ich odpočítať, lebo v nejakej predchádzajúcej iterácii hľadania zlepšujúcej ceste sme ich do dĺžky cesty pripočítali a teraz už po nich nič netečie. Teda v jednej F-F iterácii z pomedzi všetkých zlepšujúcich ciest potrebujeme vybrať takú, ktorá bude mať súčet ohodnotení hrán najnižší.

V prvej iterácii začíname s nulovým tokom, a teda tu nám stačí nájsť najkratšiu zlepšujúcu cestu (po žiadnej hrane nebudeme tlačiť tok späť, keďže je nulový). Pred začiatkom druhej iterácie, bude po všetkých hranách tiecť tok buď 0, alebo 1. V prípade, že budeme chcieť ísť v druhej iterácii po hrane, po ktorej zatiaľ nič netečie, ohodnotenie hrany bude naďalej 1. V prípade, že budeme chcieť ísť po hrane po smere toku (veľkosti 1), potom ohodnotenie hrany bude tiež 1. V prípade, že budeme chcieť ísť po hrane proti smeru toku (veľkosti 1), tak ohodnotenie takejto hrany bude -1. Tým docielime, že ak po hrane budeme tok tlačiť späť, tak sa nám naozaj zníži aj dĺžka cesty A-C presne o tok, čo pred tým po tej hrane tiekol. A teda ohodnotenie hrán nám bude udávať dĺžku cesty medzi A-C.

Nájdenie zlepšujúcej cesty bude spočívať v nájdení najkratšej cesty s naším novým ohodnotením hrán. Na to ale nemôžeme použiť obyčajné prehľadávanie do šírky, keďže to nefunguje na grafe so zápornými hranami. Budeme musieť použiť napr. Bellman-Fordov algoritmus, ktorý si poradí aj so zápornými hrana-

mi. Opäť budeme potrebovať spraviť dve iterácie F-F algoritmu, ale s úpravou hľadania zlepšujúcej cesty. Časová zložitosť Bellman-Forda je lineárna od súčinu počtu hrán a vrcholov, čo je aj výsledná zložitosť celého algoritmu.

Nie je to ale optimálne riešenie. Brzdí nás práve hľadanie najkratšej cesty, kde ohodnotenie hrán môže byť záporne. Existuje ale spôsob ako sa tomu vyhnúť. Trik spočíva vo vhodnom ohodnotení hrán pri hľadaní zlepšujúcej cesty. Je to už ale nad rámec nášho riešenia.

Pavol „Pali“ Rohár

25-5-8 Boxy, z TEXu ven!

První úkol nebylo těžké vymyslet. Stačilo v upravené výstupní rutině vyprázdnit box 255 tak, aby se nikam nevypsal. Například jste mohli použít příkaz `\setbox0{\box255}`. Tedy, to bylo jádro úkolu, pak bylo potřeba zajistit, abyste tím nic nerozbili.

1. Vysypání dříve vypsaného materiálu. Bylo potřeba vložit na příslušné místo `\vfil\eject`. Dá se i napsat cyklus, kdyby po vysypání poslední strany ještě něco zbylo, ale nebylo to nutné. Makro `\stopoutput` se tedy smí volat jen na takovém místě, kde se smí objevit konec strany.
2. Uložení původní výstupní rutiny. Zde bylo potřeba například zakázat vnoření, nebo nějak chytře vynutit, aby se makra `\stopoutput` a `\startoutput` chovala jako třetí typ závorek.
3. Bylo třeba nulovat proměnnou `\deadcycles`, aby si TEX nemyslel, že se mu výstupní rutina zacyklila.
4. Vysypání vysázeného materiálu před `\startoutput`, tedy vložit i tam bylo potřeba `\eject`.

Nebo jste mohli předefinovat `\shipout`, aby místo vypisování boxů svůj materiál zahazoval. Zde je jenom potřeba vědět, že některé zběsilejší pluginy také předefinovávají `\shipout` a dát si pozor na kolize. Taktéž je potřeba na správných místech vložit `\eject`.

```
\let\primitiveshipout\shipout
\newbox\stopoutputbox
\def\stopoutput{%
  \vfil\eject
  \def\shipout{%
    \deadcycles=0\setbox\stopoutputbox
  }%
}
```

```
\def\startoutput{%
  \vfil\eject
  \let\shipout\primitiveshipout
}
```

Ještě jednodušší verze, leč také funkční (v běžných případech), vypadala takto:

```
\newbox\stopoutputbox
\def\stopoutput{%
  \setbox\stopoutputbox\vbox\bgroup
}
\let\startoutput\egroup
```

Každý přístup má své ply a míny, záleží na tom, co si od maker slibujete. Přesná sémantika úmyslně nebyla zadána, abyste si mohli vybrat. Když jsem před časem psal podobné makro pro účely extrakce vzorců pro web, použil jsem první variantu s pár dalšími specifiky pro naše letáky, a každý extrahovaný vzorec pak je samostatnou stránkou, se kterou se pracuje dál.

Zlatý hřebíček posledního dílu seriálu v **úkolu 2** jste úspěšně zatloukli. Jak sázet do více sloupců?

0. Otevřeme skupinu (`\begingroup`), aby všechny změny, které napácháme, zůstaly lokální.
1. Předefinujeme výstupní rutinu tak, aby svůj výstup odložila do speciálního boxu. Pak ukončíme stránku, čímž si dosud nevysázený materiál uložíme stranou.
2. Předefinujeme rozměry stránky, aby odpovídaly situaci, kdy se pod sebe naskládají všechny sloupce, které se mají na straně objevit. Upravíme výstupní rutinu, viz následující body.
3. Když je ve výstupním seznamu dostatek materiálu, spustí se výstupní rutina, která sloupce nakrájí na správnou výšku a nasází vedle sebe. Zbytek se vrací do zpracování v dalším cyklu. Je potřeba nezapomenout na vrácení odloženého materiálu na správné místo.
4. Na konci vícesloupcové sazby je nakonec potřeba ošetřit případy, kdy celá sekce vyjde na jednu stranu, nebo by se vešla, ale musí se rozdělit mezi dvě strany.
5. Drobná výhybka na začátku `\multicolumn` za pomoci podmínky `\ifinner` vybírá mezi vnitřní verzí (jako v minulé sérii) a vnější, která může být přes více stran.
6. Zavřeme skupinu, aby všechny napáchané změny zůstaly lokální. Tím jsme si zajistili, aby se nám to při vnořování nerozbilo úplně.

Co se týče vnořování, úplně stačilo, když jste vnitřní vícesloupcovou část vysázeli bez dělení, jako podle minulé série. Hlavně šlo o to, aby se to tímto vnořením celé nerozbilo.

V řešení se nakonec proti původnímu plánu objevilo primitivum `\pagetotal`. To říká, kolik materiálu se už nashromáždilo k vysypání do tisku. Občas se to může hodit, například ve chvíli, kdy je potřeba na konci vícesloupcové sekce zjistit, jestli se celý výsledek vejde na stránku, nebo je potřeba lámat.

Zdrojový text maker:
`http://ksp.mff.cuni.cz/viz/25-5-8.tex`

Řešení **úkolu 3** pak bylo přímočaré. Taková oddychovka.

```
\def\defrgbcolor#1#2{%
  \def#1{\pdfliteral{#2 rg #2 RG}}%
}
\def\defcmykcolor#1#2{%
  \def#1{\pdfliteral{#2 k #2 K}}%
}
\def\defgrayscalecolor#1#2{%
  \def#1{\pdfliteral{#2 g #2 G}}%
}
```

A to je, milí přátelé, pro letošek všechno. Doufám, že jste se TEXu nezalekli a příští rok budou vaše řešení (nejen) KSP čistě a úhledně vysázena vaším vlastním makrobalíkem.

Přišel čas se s vámi, milí řešitelé, rozloučit. Napřesrok už organizovat nebudu, místo mě bude jiný, mladší, ale to bude sekáč ... Děkuju vám za krásná řešení, rád jsem s vámi ušel kus vaší cesty ku vědění a k umění programování. Mnoho štěstí a hezké prázdniny vám přeje autor seriálu

Jan „Moskyto“ Matějka

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
0.				39	298.0
1.	Rastislav Rabatin	GJHroncaBA	4	31	268.1
2.	Dominik Macháček	GLanškroun	4	27	232.6
3.	Martin Raszyk	G_Karvina	3	26	229.3
4.	Michal Punčochář	GJírovcČB	3	26	214.7
5.	Richard Hladík	GOAMarLaz	0	22	195.5
6.	Jakub Maroušek	G_Písek	3	26	194.0
7.	Štěpán Hojdar	GJírovcČB	3	25	188.5
8.	Dalimil Hájek	GKepleraPH	2	31	186.4
9.	Jakub Šafin	GHorMichal	4	20	180.4
10.	Petr Houška	GJírovcČB	3	23	175.0
11.	Jakub Svoboda	GKomHavíř	3	26	173.2
12.	Ondřej Mička	GJírovcČB	4	24	170.8
13.	Martin Španěl	ArcibisGPH	4	20	168.2
14.	Matej Lieskovský	GOmskPha	3	23	167.7
15.	Martin Černý	G_Sokolov	3	22	165.3
16.	Martin Šerý	GJírovcČB	3	24	162.2
17.	Marek Dobranský	GHorMichal	3	24	160.4
18.	Vojtěch Hlávka	GŠlapanice	4	25	153.1
19.	Jan Mikel	G_RožnovPR	4	15	135.0
20.	Mikuláš Hrdlička	MensaG	2	18	126.2
21.	Lukáš Ondráček	GVolgogrOS	4	11	109.5
22.	Jan-Sebastian Fabík	GJarošeBO	3	11	106.4
23.	Mark Karpilovskij	GJarošeBO	4	10	104.3
24.	Ondřej Hlavatý	GJirsíkaČB	4	14	102.5
25.	Petra Pelikánová	GJarošeBO	4	18	99.1
26.	Vojtěch Sejkora	SPSE_Pard	4	11	94.6
27.	Kateřina Zákravská	GJar	4	10	83.4
28.	Vladan Glončák	GĽŠtúraTN	4	11	82.1
29.	Vojtěch Vašek	GHli	4	12	75.9
30.	Sabína Fraňová	GDubNVahom	4	17	74.3
31.	Štěpán Trčka	GSlavičín	2	15	73.5
32.	Anna Zákravská	GJar	4	8	72.0
33.	Jan Knížek	G_Strakon	2	11	67.4
34.	Aneta Šťastná	GOmskPha	3	7	56.0
35.	Jonatan Matějka	SŠP_ČB	3	11	52.7
36.	Štěpán Šimsa	GJungmanLT	4	5	49.5
37.	Václav Rozhoň	GJirsíkaČB	2	6	47.6
38.	Jan Pokorný	G_Bučovice	1	8	46.7

39.	Alexander Mansurov	GNVPláníPH	4	6	44.4
40.	Jitka Fürbacherová	GKlatovy	4	10	40.9
41.	Jan Lejnar	GKlatovy	3	7	39.1
42.	Dominik Smrž	GOhradníPH	3	4	38.8
43.	Tomáš Velecký	GBezručeFM	2	6	34.8
44.	Tomáš Svítil	AES_NewDelhi	4	7	34.6
45.	Radovan Švarc	G_ČTřebová	2	4	34.4
46.	Milan Šorf	GNeumannŽR	3	10	33.5
47.	Ondřej Cífka	GNAlejíPH	4	3	32.0
48.	Václav Volhejn	GKepleraPH	0	4	29.5
49.	Ráchel Sgallová	GZborovPH	3	3	28.2
50.	Ondřej Theiner	GJírovcČB	3	4	26.5
51.	Ondřej Hübsch	GArabskáPH	3	3	26.1
52.	Jozef Kaščák	G_Svidník	4	4	23.3
53.	Tereza Hulcová	GKlatovy	4	3	23.2
54.	Michal Staruch	GOA_Vrchla	4	4	22.7
55.	Marek Dědič	GBNěmcovHK	3	2	19.3
56.	Veronika Klapová	GMHorPH	4	2	14.6
57.	Michal Kužela	GSlavičín	1	3	14.0
58.	Pavel Salva	VOŠŠumperk	3	1	8.7
59.	Tadeas Friedrich	GOhradníPH	3	1	8.0
60.	Tomáš Zahradník	GOPavla PH	3	1	7.1
61.	Jan Horešovský	GMěl	3	1	6.4
62.	Dominik Roháček	SPŠLegioJI	3	2	6.0
63.	Vojta Staněk	PORGPha	-1	1	5.7
64.	Přemysl Šťastný	GZamberk	-1	2	4.7
65.	Dominika Macháčová	GSereď	3	1	4.4
66.	Martin Vzorek	SŠStarTura	2	1	1.4

Obsah

Jiří Setnička a kolektiv

Korespondenční seminář z programování XXV. ročník

Autoři a opravující úloh:
Jan Bok, Karolína Burešová, Pavel Čížek, Lukáš Folwarczný, Jan Hadrava, Martin Mareš, Jan Matějka, Michal Pokorný, Pavol Rohár, Jiří Setnička, Filip Štědronský, Karel Tesař, Pavel Veselý, Michal Vaner, Peter Zeman

Autoři příběhů v zadání:
Radim Cajzl, Karolína Burešová, Lukáš Folwarczný, Michal Vaner, Jiří Setnička

Vydal MATFYZPRESS
vydavatelství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze
Sokolovská 83, 186 75 Praha 8
jako svou 441. publikaci.

TEX-ová makra pro sazbu ročenky vytvořili Martin Mareš, Jan Matějka, Radim Cajzl a Jiří Setnička.

S jejich pomocí ročenku vysázel Jan Bok.

Obrázek na obálce nakreslila Petra Pelikánová.

Sazba byla provedena písmem Computer Modern v programu TEX.

Vytisklo Reprostředisko UK MFF.

Vydání první, 210 stran
Náklad 200 výtisků
Praha 2013

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

ISBN 978-80-7378-247-4

ISBN 978-80-7378-247-4