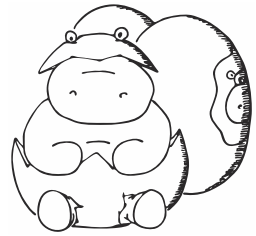
Milí řešitelé a řešitelky!



Orgové vesměs nějakým způsobem přežili zkouškové a ani během něho na vás nezapomněli. Při usilovném přemýšlení u zkoušek se vynořily nápady na zajímavé úlohy, a tak vznikla čtvrtá série 25. ročníku – spousta lehkých i těžkých úloh, další díl seriálu o TEXu a další pokračování už tak dost zamotaného příběhu.

Za úspěšné řešení KSP je možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů, přičemž půjde získat maximálně 300 bodů. Připomínáme, že z každé série se do celkového bodového hodnocení započítává 5 nejlépe vyřešených úloh.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude pravděpodobně příliš pozdě na to, aby pátou sérií dohánělí chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Navíc každému, kdo v této sérii **vyřeší všechny vstupy v CodExu rychleji než vzorové řešení (a správně), pošleme čokoládu**.

Termín odevzdání čtvrté série je stanoven na **pondělí 25. března v 8:00 SEČ**. CodExová úloha má termín o den posunutý, protože nám ji opravuje automat – odevzdejte ji do 26. března, 8:00 SEČ.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou. V tom případě byste jej měli podat do středy 20. března s naší adresou

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

Před tím ale vyplňte přihlášku (a to i tehdy, když jste se KSPčka účastnili loni) na *http://ksp.mff.cuni.cz/*, kde najdete i další informace o tom, jak KSP funguje. Na webu máme také fórum, kde se můžete na cokoli zeptat. Nebo nám můžete napsat na e-mail *ksp@mff.cuni.cz*.

Čtvrtá série dvacátého pátého ročníku KSP

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.

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.

25-4-1 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.

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.

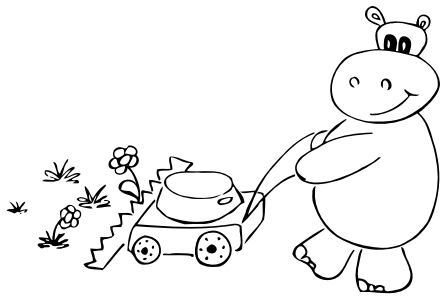
Příklad: (První řádek obsahuje rozměry trávníku, dalších M řádků popisuje hmotnosti trávy na jednotlivých čtvercích, pak následuje číslo K a K řádků popisujících oblasti.)

```
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.



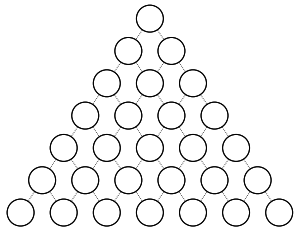
[1] http://ksp.mff.cuni.cz/viz/25-3-3

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.

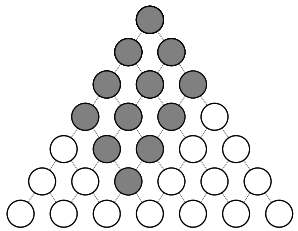
Bohužel, ve zdejších končinách není přípustné nosit samurajský meč a zastrašovat jím drzé zvědavce. Nezbylo mi tedy než sáhnout po telefonu a požádat o laskavost jednoho z mých věrných lidí u policie. Když jsem se tenkrát sázel se svým čínským kolegou, kdo dokáže podplatit i posledního policistu, nikdy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

Výsledková listina třetí série dvacátého pátého ročníku KSP

	řešitel	škola	ročník	sérií	2531	2532	2533	2534	2535	2536	2537	2538	série	celkem
0.					12	10	13	10	9	10	7	13	58,0	176,0
1.	Rastislav Rabatin	GJHroncaBA	4	6	12	9,5		10	9	10	7	6,5	50,8	160,3
2.	Martin Raszyk	G_Karvina	3	13	12	10	13	10	6	10	7	13	58,0	155,0
3.	Michal Punčochář	GJírovcČB	3	8		11		10	9	10		12,5	52,7	151,6
4.	Dominik Macháček	GLanškroun	4	8		4,5	13	6	6	10		12	49,1	143,7
5.	Martin Černý	G_Sokolov	3	3		8		10	3,5	4	7	11	45,2	134,3
6.	Štěpán Hojdar	GJírovcČB	3	3		4,5		6	3,5	6		10,5	41,3	128,3
7.	Martin Španěl	ArcibisGPH	4	6		9	11	2	9				33,7	126,8
8.	Vojtěch Hlávka	GŠlapanice	4	18	12	6	9	10	4,5	10			42,1	125,4
9.	Jakub Maroušek	G_Písek	3	3		5		10	9	5	4,5	12,5	46,9	120,4
10.	Jakub Šafin	GHorMichal	4	3	10	10	11	10	9	10	7		53,6	118,8
11.	Ondřej Mička	GJírovcČB	4	16	12	10		10	7	9	7	12	51,6	115,2
12.	Dalimil Hájek	GKepleraPH	2	8		5		9	3	6		5,5	33,0	114,6
13.	Matej Lieskovský	GOmskPha	3	8		8	7	10	5			4,5	38,6	111,5
14.	Richard Hladík	GOAMarLaz	0	3	12	10	9			5		12,5	53,8	107,4
15.	Ondřej Hlavatý	GJirsíkaČB	4	2									0,0	102,5
16.	Petr Houška	GJírovcČB	3	4		4		6	3,5	7		12,5	41,3	99,8
17.	Mikuláš Hrdlička	MensaG	2	3		4			3,5			8,5	23,5	96,4
18.	Jakub Svoboda	GKomHavíř	3	3		9		4,5	1	0	4		24,5	95,7
19.	Martin Šerý	GJírovcČB	3	4				9,5	3,5	7		12	36,6	94,8
20.	Petra Pelikánová	GJarošeBO	4	4			1	0	3,5	7	4	5,5	30,3	92,9
21.	Marek Dobranský	GHorMichal	3	3		8		4,5	3,5	10	3,5		37,3	89,9
22.	Vladan Glončák	GĽŠtúraTN	4	4					3,5	9	7	12,3	34,9	82,1
23.	Mark Karpilovskij	GJarošeBO	4	8			7	10	9	10	7		44,3	81,3
24.	Lukáš Ondráček	GVolgogrOS	4	8								12	12,3	74,4
25.	Sabína Fraňová	GDubNVahom	4	4		4	5	2	2		3		26,1	74,3
26.	Kateřina Zákravská	GJar	4	4						9		12,5	22,4	72,8
27.	Vojtěch Vašek	GHli	4	7				8	3,5	2	3,5		20,9	72,0
28.	Jan Mikel	G_RožnovPR	4	2					3,5		7	8,5	24,4	67,9
29.	Vojtěch Sejkora	SPSE_Pard	4	12						10		12,5	22,4	64,2
30.	Anna Zákravská	GJar	4	4		4		3		9		12,5	33,8	61,4
31.	Jan-Sebastian Fabík	GJarošeBO	3	7			9		9	10	7	2	39,4	51,4
32.	Jan Knížek	G_Strakon	2	8				5			7	7	21,3	49,6
33.	Jan Pokorný	G_Bučovice	1	1									0,0	46,7
34.	Alexander Mansurov	GNVPláníPH	4	11									0,0	44,4
35.	Aneta Šťastná	GOmskPha	3	5									0,0	39,7
36.	Jan Lejnar	GKlatovy	3	3		5			3,5	0	3		18,1	39,1
37.	Jonatan Matějka	SŠP_ČB	3	12									0,0	37,5
38.	Jitka Fürbacherová	GKlatovy	4	9		3,5		3					7,8	37,1
39.	Štěpán Trčka	GSlavičín	2	5									0,0	35,2
40.	Tomáš Velecký	GBezručeFM	2	5									0,0	34,8
41.	Tomáš Svítil	AES_NewDelhi	4	2									0,0	34,6
42.	Radovan Švarc	G_ČTřebová	2	2									0,0	34,4
43.	Milan Šorf	GNeumannŽR	3	3					1	0			2,2	33,5
44.	Ondřej Cífka	GNAlejíPH	4	9									0,0	32,0
45.	Jozef Kaščák	G_Svidník	4	1									0,0	23,3
46.	Tereza Hulcová	GKlatovy	4	8									0,0	23,2
47.	Michal Staruch	GOA_Vrchla	4	2		5			3,5				13,7	22,7
48.	Ondřej Hübsch	GArabskáPH	3	17									0,0	21,8
49.	Václav Volhejn	GKepleraPH	0	4			9		2,5				15,5	20,7
50.	Marek Dědič	GBNěmcovHK	3	1									0,0	19,3
51.	Pavel Salva	VOŠŠumperk	3	4									0,0	8,7
52.	Tomáš Zahradník	GOPavla PH	3	1		4							7,1	7,1
53.	Jan Horešovský	GMěl	3	1									0,0	6,4
54.	Dominik Roháček	SPŠLegioJI	3	1									0,0	6,0
55.	Přemysl Šťastný	GZamberk	-1	1									0,0	4,7
56.	Michal Kužela	GSlavičín	1	1							2		4,0	4,0
57.	Martin Vzorek	SŠStarTura	2	1		0,5							1,4	1,4

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

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 $10^9 + 7$.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[2] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

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.

[2] `http://ksp.mff.cuni.cz/viz/codex`

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

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 ≈ 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) × 256.07481pt

R2 R4
R3 R5

Všimněte si, že takto nelze natahovat nebo smršťovat samotný obsah boxu. To TEX neumí.[3] 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`.

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

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`.

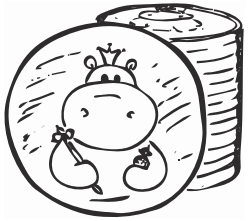
Makro `\m` slouží k polknutí tečky za pořadím účastníka v případě, že chceme prázdný první sloupec.

kolik indexů do zásobníku přidáme. A těch přidáme n, přičemž každý právě jednou. Paměti nám stačí taktéž lineárně. Řešení je zjevně optimální, protože vstup musíme minimálně jednou přečíst, tedy lepší než lineární časové složitosti dosáhnout nelze.

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.[7]

Program (C++):
`http://ksp.mff.cuni.cz/viz/25-3-6.cpp`

Lukáš Folwarczný

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

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 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

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.

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 analogickým způsobem 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 ale 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

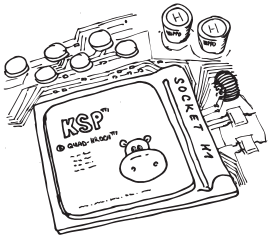
Během sazby je možno pracovat i s jinými soubory než tím vstupním. Je vhodné si je nejprve pojmenovat, to se provádí makrem `\newread` (pro vstup) a `\newwrite` (pro výstup). Přesněji řečeno, tímhle si pojmenujete ukazatel na soubor. Jeden ukazatel nemůže zároveň ukazovat na vstup i výstup a jeden soubor 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)`
`csslt10`	*`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

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[6] 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, 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

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

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,

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 $\mathcal{O}(N)$.

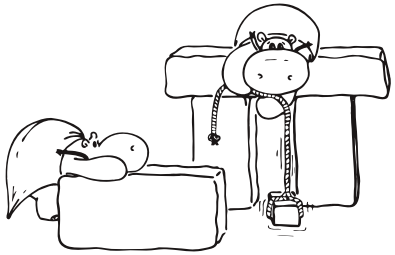
Ukažme teď, že tyto prefixy nám skutečně stačí, abychom námahu svozu trávy z intervalu na vybrané políčko určili v konstantním čase. Mějme $a, b : 1 \leq a \leq b \leq N$ a nějaké $i : a \leq i \leq b$.

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

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.

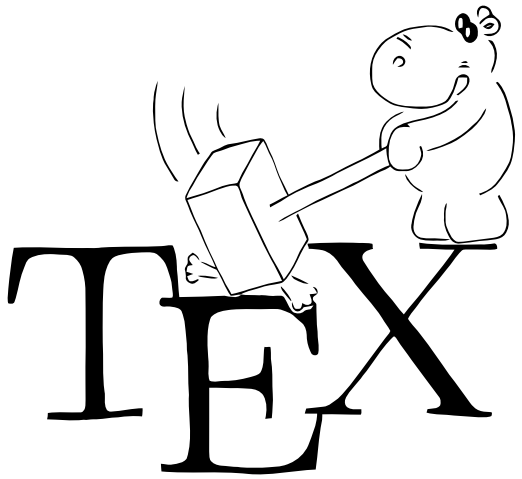
Po nastavení fontu je také potřeba správně nastavit některé další hodnoty, typicky `\baselineskip`.

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.[4] Je to velmi chytře napsaný a mocný systém, který sám běžně používám v sazbě a který používáme i v KSP.

Jan „Moskyto“ Matějka



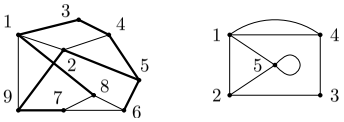
Za obrázek TEXajícího hrocha děkujeme Petře Pelikánové.
Pokud i v sobě máte výtvarného ducha, budeme rádi, pokud ho také probudíte :-)

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

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, …, 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}, …, 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

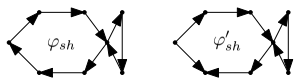
Někdy se hodí jeden z vrcholů stromu prohlásit za *kořen,* čímž jsme si v každém vrcholu určili směr nahoru (ke kořeni – je to zvláštní, ale matematici obvykle kreslí stromy kořenem vzhůru) a dolů (od kořene). Souseda vrcholu směrem nahoru pak nazýváme jeho *otcem,* sousedy směrem dolů jeho *syny*.

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.

Čokolády

Zisk čokolády v této sérii byl podmíněn získáním plného počtu bodů za alespoň tři úlohy. To se podařilo 8 řešitelům. Speciální bychom chtěli vyzdvihnout Martina Raszyka, kterému se povedlo vyřešit dokonce 7 úloh na plný počet bodů, gratulujeme.

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.[5] 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 našem 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 $\mathcal{O}(N \log N)$.

Se škrtáním dvojice

Jak modifikujeme tento algoritmus tak, aby uměl oprošťovat veřejné činitele od pracovních povinností? Jako první se nabízí zkrátka 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 některé 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

[5] `http://ksp.mff.cuni.cz/viz/kucharky/teorie-cisel`

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

Hranová a vrcholová 2-souvislost

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

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

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

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

Proto si pro každý vrchol spočítáme hladinu, ve které se nachází (kořen je na hladině 0, jeho synové na hladině 1, jejich synové 2, ...). Dále si pro každý vrchol v spočítáme, do jaké nejvyšší hladiny (s nejmenším číslem) vedou ryzí zpětné hrany z podstromu s kořenem v. To můžeme udělat přímo při procházení do hloubky, protože než se vrátíme z v, projdeme celý podstrom pod v. Pokud všechny zpětné hrany vedou do hladiny stejné nebo větší než té, na které je v, pak odebráním hrany vedoucí do v z jeho otce vzniknou dvě komponenty souvislosti, čili tato hrana je mostem. V opačném případě jsme nalezli kružnici, na níž tato hrana leží, takže to most být nemůže. Výjimku tvoří kořen, který žádného otce nemá a nemusíme se o něj proto starat.

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

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

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

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

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

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

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

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

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

Dnešní menu Vám servírovali

Martin Mareš, David Matoušek a Petr Škoda

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.

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	1	1
2	1	0	0	1	1	0	0	0	1
3	1	0	0	0	0	0	0	0	0
4	0	1	0	0	1	0	1	0	0
5	0	1	0	1	0	1	0	1	0
6	0	0	0	0	1	0	1	1	0
7	0	0	0	1	0	1	0	1	1
8	1	0	0	0	1	1	1	0	0
9	1	1	0	0	0	0	1	0	0

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

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[i]$	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 $\mathcal{O}(N+M)$.

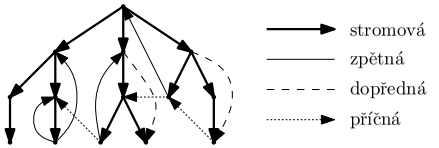
```
var Komponenta: array[1..MaxN] of Integer;
    NovaKomponenta: Integer;

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;
```