Výsledková listina první série dvacátého pátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2511*	*2512*	*2513*	*2514*	*2515*	*2516*	*2517*	*2518*	*série*	*celkem*
0.					12	12	7	12	7	10	7	13	59,0	59,0
1.	Rastislav Rabatin	GJHroncaBA	4	4	12	12		9	7	4		13	54,7	54,7
2.	Ondřej Hlavatý	GJirsíkaČB	4	1	8	6	6	11	4,9	5	4,5	13	52,9	52,9
3.	Martin Španěl	ArcibisGPH	4	4	8	12	2		7	8,5	4,5	11,5	50,9	50,9
4.	Lukáš Ondráček	GVolgogrOS	4	6	11		7	11	7			13	50,0	50,0
5.	Martin Černý	G_Sokolov	3	1	5	9	4	6		3	3	11,5	48,1	48,1
6.	Martin Raszyk	G_Karvina	3	11	8		7	12	7		7	13	47,0	47,0
7.	Jan Pokorný	G_Bučovice	1	1	6	6	6,5	0	4,9	5	0,5	12,5	46,7	46,7
8.	Mikuláš Hrdlička	MensaG	2	1	6	4	2	6	7			11	46,1	46,1
9.	Michal Punčochář	GJírovcČB	3	6	8	8	7	3	7	5	7	12	45,7	45,7
10.	Petra Pelikánová	GJarošeBO	4	2	5	3	6,5	5	7	6,5	3	12	45,2	45,2
11.	Vojtěch Hlávka	GŠlapanice	4	16	8	12	7	7	5	5	7	12,3	44,1	44,1
12.	Dominik Macháček	GLanškroun	4	6	8	6			7	5		12,5	44,0	44,0
13.	Dalimil Hájek	GKepleraPH	2	6	7	5	7	6	0,9	4	3	12,8	43,8	43,8
14.	Jan Mikel	G_RožnovPR	4	1	6		3,5		7	6		11,9	43,5	43,5
15.	Matej Lieskovský	GOmskPha	3	6	8	6	7		7			8,8	42,2	42,2
16.–17.	Štěpán Hojdar	GJírovcČB	3	1	8		3		5	3,5		11,8	41,6	41,6
	Jakub Svoboda	GKomHavíř	3	1	4		6		7	4,5		11,5	41,6	41,6
18.	Ondřej Mička	GJírovcČB	4	14	8		7		7	7,5		13	40,6	40,6
19.	Aneta Šťastná	GOmskPha	3	5	8	2			5	6,5		11	39,7	39,7
20.	Alexander Mansurov	GNVPláníPH	4	10	6			8	6	5		12	38,4	38,4
21.	Kateřina Zákravská	GJar	4	2	5		3		3	4	3	11,8	37,9	37,9
22.	Jakub Maroušek	G_Písek	3	1	4		5	1		3,5	2	12	37,4	37,4
23.	Martin Šerý	GJírovcČB	3	2	1		4,5		5	4	2	12,3	35,9	35,9
24.	Marek Dobranský	GHorMichal	3	1	4	3			0,9	5		8	35,3	35,3
25.	Štěpán Trčka	GSlavičín	2	5	3		1	7	6	2	2	9	35,2	35,2
26.	Richard Hladík	GOAMarLaz	0	1	5		2,5		4	1,5		10	35,0	35,0
27.	Radovan Švarc	G_ČTřebová	2	2			7		5	7,5		9,7	34,4	34,4
28.	Ondřej Cífka	GNAlejíPH	4	9				12	7			13	32,0	32,0
29.	Petr Houška	GJírovcČB	3	2	1		4,5		5		2	11	31,1	31,1
30.–31.	Vojtěch Sejkora	SPSE_Pard	4	10				6	5	3,5		11,7	27,3	27,3
	Tomáš Velecký	GBezručeFM	2	4	4				1	4	0	11,5	27,3	27,3
32.	Vladan Glončák	GLŠtúraTN	4	2			4	1	4			13	27,0	27,0
33.	Tomáš Svítil	AES_NewDelhi	4	1	1		1,5			4	0	9,3	24,9	24,9
34.	Mark Karpilovskij	GJarošeBO	4	6	12			12					24,0	24,0
35.	Jozef Kaščák	G_Svidník	4	1	3				2		0,5	8,7	23,3	23,3
36.	Tereza Hulcová	GKlatovy	4	8	11		6		5				23,2	23,2
37.	Jan Knížek	G_Strakon	2	6				0	4		2,5	12,7	21,6	21,6
38.	Vojtěch Vašek	GHli	4	5	0			7				8,5	19,8	19,8
39.	Sabína Fraňová	GDubNVahom	4	2	0	6	2		1	2	0		19,7	19,7
40.	Marek Dědič	GBNěmcovHK	3	1	4							8,8	19,3	19,3
41.	Anna Zákravská	GJar	4	2						2,5		12,5	17,9	17,9
42.	Jonatan Matějka	SŠP_ČB	3	11	1	5				3,5		7,5	17,0	17,0
43.	Milan Šorf	GNeumannŽR	3	1				0	3			7,8	16,4	16,4
44.	Jan-Sebastian Fabík	GJarošeBO	3	6				12					12,0	12,0
45.	Jitka Fürbacherová	GKlatovy	4	7	0		4		2	3			11,9	11,9
46.	Jakub Šafin	GHorMichal	4	1				9					11,2	11,2
47.	Ondřej Hübsch	GArabskáPH	3	16				10					8,8	8,8
48.	Pavel Salva	VOŠSumperk	3	4				6					8,7	8,7
49.	Jan Horešovský	GMěl	3	1			5						6,4	6,4
50.	Dominik Roháček	SPŠLegioJI	3	1				2			0,5		6,0	6,0
51.	Přemysl Šťastný	GZamberk	-1	1	2						0		4,7	4,7

Milí řešitelé a řešitelky!

Podzim je v plném proudu, listí mocně opadává a vy držíte v ruce druhý leták 25. ročníku KSP. Řešení vašich úloh pilně opravujeme a už se těšíme na další. Připomínáme, že z každé série se do celkového bodového hodnocení započítává 5 nejlépe vyřešených úloh.

V každé sérii jsou letos dvě lehké úlohy pro začátečníky za menší počet bodů.

Za úspěšné řešení KSP je také možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

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

Připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok a tužku. Navíc každému, kdo vyřeší alespoň jednu ze tří nejvíc bodovaných úloh druhé série na plný počet bodů, pošleme čokoládu.

Dále všechny řešitele i jiné lidi se zájmem o studium na MFF UK zveme na **Den otevřených dveří MFF UK**, který proběhne již ve **čtvrtek 29. listopadu**. Více informací na adrese *http://www.mff.cuni.cz/verejnost/dod/*.

Termín odevzdání druhé série je stanoven na **pondělí 10. prosince v 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 5. prosince 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*.

Druhá série dvacátého pátého ročníku KSP

„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.

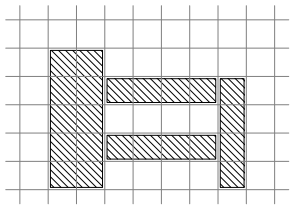
25-2-2 Sekání trávy podruhé — 11 bodů

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 máme 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.

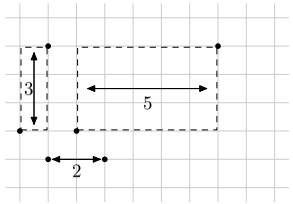
25-2-4 Organizace vykládky — 13 bodů

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 dvou bodů odpovídá větší z rozdílů jejich souřadnic. Tedy

$$d((x_1, y_1), (x_2, y_2)) = \max\{|x_1 - x_2|, |y_1 - y_2|\}$$



Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

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?

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

(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.

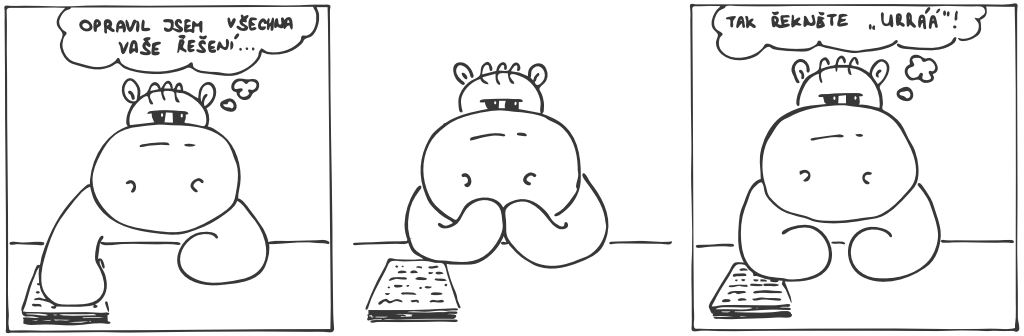
Ještě můžete chtít změnit velikost strany. Šířku a výšku strany určují rozměry `\pdfpagewidth` a `\pdfpageheight`. Šířku a výšku zrcadla (potištěné části papíru) nastavíte v `\hsize` a `\vsize`. Konečně `\hoffset` a `\voffset` mění levý a horní okraj – ten je roven tomuto rozměru **plus 1in**.

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



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.[9]

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.

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

[9] `http://mo.mff.cuni.cz/p/57/reseni-1.html`

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.

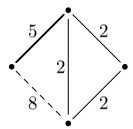
25-2-6 Optimalizace v redakci — 9 bodů

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

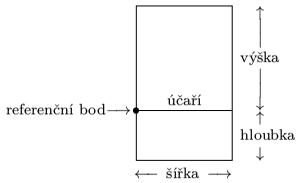
25-2-7 Zaléváme dokument — 13 bodů

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.

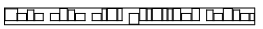
Sazba do boxů

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:[2]

`\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.

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

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.

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 můžete použít 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 pomocí primitiva `\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.[3]

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

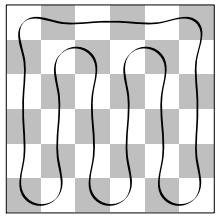
Makra jsou mocnější zbraň, než by se mohlo zdát. Parametry totiž nemusí být jenom uzavřené ve složených závorkách. Mohou být odděleny prakticky čímkoli. Taková definice vypadá například takto:

```
\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.

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

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.



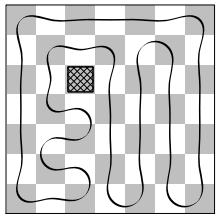
Ale pozor, to stále není všechno. Co když jeden z rozměrů trávníku bude 1? To pak naše řešení tak úplně nefunguje, protože se nemáme jak vrátit. Rozměr 1 tedy musíme vyřešit zvlášť:

- 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 $m_{i-1} < m_i$.

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 algoritmu 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 celého algoritmu je $\mathcal{O}$(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.[8] 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.

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.

[8] `http://ksp.mff.cuni.cz/viz/kucharky/slozitost`

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 neoddděleným parametrem 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.

Když chceme číslo vypsat (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).[4] 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.

Jan „Moskyto“ Matějka

Recepty z programátorské kuchařky: 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.[5]

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

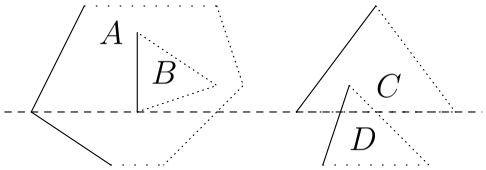
Algoritmus na hledání minimální kostry, který si předvedeme, je typickou ukázkou tzv. hladového algoritmu. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím)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

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

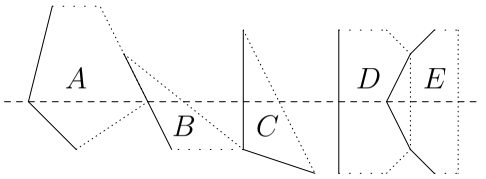
[5] `http://ksp.mff.cuni.cz/viz/kucharky/grafy`

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

25-1-1 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.

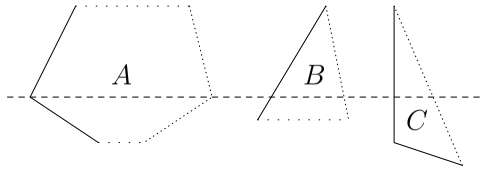
Vyššie popísaná implementácia nám zaručí lineárnu časovú zložitosť. Na každý vrchol sa totižto pozrieme práve raz (keď ho odoberáme) a zároveň nastavíme stupeň každému jeho susedovi. Časová zložitosť je teda $\mathcal{O}(n+m)$, kde m je počet dvojíc. Pamäťová je rovnaká ako časová.

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.

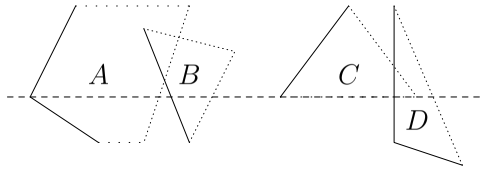
Základem programu tedy bude udržovat si informace o tom, jak vypadá rozdělení na intervaly odpovídající jednotlivým mnohoúhelníkům. V této struktuře budeme potřebovat být schopni rychle provést následující:

- 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[6] (v programu užit AVL strom kvůli vyvažování – viz kuchařku o vyhledávacích stromech).[7] 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 (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).



[6] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`
[7] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`

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 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 (resp. jsme je různými udělali).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 bu-

dou 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 Král', Martin Mareš a Milan Straka