Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

30. ročník | **KSP-Z** | **Březen 2018**

Omlouváme se za to, že se vydání finálních řešení zdrželo, ale doufáme, že i tak vám naše vzorová řešení pomohou k tomu, abyste se v programování a hlavně v řešení problémů pořád zlepšovali. Gratulujeme všem, kdo získali nějaké body!

Jako vždy, pokud se vám cokoli nezdá nebo máte nějaký dotaz, neváhejte se ozvat na našem fóru.

Řešení druhé série začátečnické kategorie 30. ročníku KSP

30-Z2-1 K-k-koktavý K-K-Kevin

Naším úkolem bylo všechny souvislé posloupnosti stejných znaků nahradit znakem jediným, a to na každém řádku zvlášť.

Přímočaré řešení je projít vstup znak po znaku. Vždy si budeme pamatovat, který znak jsme přečetli před aktuálním znakem. Pokud se nově přečtený znak rovná tomu předchozímu, prostě jej budeme ignorovat. Jinak jej vypíšeme na výstup. První znak nebudeme porovnávat s ničím.

Problémy v tomto přístupu by mohly nastat v případě více prázdných řádků po sobě. Protože naším úkolem je slučovat skupiny stejných znaků do jednoho pouze v rámci jednoho řádku, je potřeba zajistit, abychom neslučovali také znaky konce řádku a nenahradili více prázdných řádků jedním. To je však velmi jednoduché – většina programovacích jazyků umí načítat vstup po řádcích. Tudíž stačí vždy načíst jeden řádek a na něm spustit algoritmus výše.

Co v případě, že tuto možnost nemáme? Pak pokaždé, když narazíme na znak nového řádku, tak jej prostě vypíšeme bez jakéhokoliv porovnávání. Stále si jej ale budeme pamatovat jako předchozí znak. To nám zajistí, že následující porovnání prvního znaku na řádku dopadne nerovností. Kdybychom toto neprovedli, tak by se mohlo stát, že porovnáme první znak na novém řádku s posledním znakem na předchozím řádku, což by nedopadlo hezky.

Jak je vidět, samotný algoritmus je velmi jednoduchý a běží v čase $\mathcal{O}(N)$, kde N je počet všech znaků, s využitím pouze jedné pomocné proměnné.

Dodatek o kódování znaků

Ještě dodejme, že kdyby text na vstupu obsahoval nějaké složitější znaky (třeba česká písmenka s diakritikou), museli bychom se v programu starat o způsob, jakým jsou znaky kódovány. Dnes se nejčastěji používá znaková sada Unicode[1] a její kódování `UTF-8`, v němž různé znaky zabírají různý počet bajtů.

Například znaky anglické abecedy zabírají jeden bajt, ale takové české Ř zabírá dva bajty. Taktéž jeden reálný znak může být složený z více „podznaků“: Ř může být uloženo jako kombinace obyčejného `R` a háčku, což potom zabere dokonce tři bajty. Je smutné, že ještě dnes existují programy, které s vícebajtovými znaky nefungují, tak si raději prostudujte, jak se znakovými sadami zachází váš oblíbený programovací jazyk. Naše testovací vstupy nicméně byly krotké – jako kódování bylo použito ASCII, které sice umožňuje ukládat pouze znaky anglické abecedy, čísla a základní interpunkci, ale vše kóduje jednobajtově.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z2-1.py`

Václav Končický

30-Z2-2 Hřiště pro tarantule

Úloha se dá poměrně přímočaře vyřešit následováním zadání – stačí pohyb pavouků po krocích odsimulovat a v každém kroku zkontrolovat, zda náhodou nestojí nějací dva pavouci na stejném políčku. Jen je třeba dát pozor na to, abychom se všemi pavouky hýbali najednou, tj. aby náš program nevyhodnotil prohození nějakých dvou pavouků jako srážku, protože se nejdřív pokusí pohnout s prvním pavoukem a zjistí, že na cílovém políčku už nějaký pavouk stojí.

Představíme si několik různě rychlých algoritmů, které naši úlohu řeší. Začneme s tím nejjednodušším, avšak nejpomalejším, a postupně budeme zrychlovat.

Přímočaré řešení

Pro každého pavouka si budeme pamatovat dvojici $[x, y]$ – souřadnice políčka, na kterém právě stojí, přičemž na začátku stojí i-tý pavouk na políčku $[i, i]$. Vstup budeme zpracovávat po řádcích: vždy přečteme jeden řádek, každému pavoukovi přepočítáme jeho pozici podle instrukce, kterou dostal, a po provedení všech instrukcí zkontrolujeme, zda se nějací dva pavouci neocitli na stejném políčku. Pokud ne, pokračujeme čtením dalšího řádku, pokud ano, vypíšeme číslo aktuálního kroku a ukončíme se, aniž bychom museli číst zbytek vstupu.

Zbývá dořešit detaily. Posouvání pavouků můžeme dělat podobně, jako jsme v první úloze minulé série posouvali prázdné místo v krabičce – jen nám navíc přibyla i instrukce „zůstaň na místě“. Na začátku programu bychom tedy mohli mít inicializaci podobnou této:

```
mv_x = {'^': 0, 'v': 0, '<': -1, '>': 1, 'o': 0}
mv_y = {'^': -1, 'v': 1, '<': 0, '>': 0, 'o': 0}
```

a samotný rozdíl souřadnic pro konkrétní instrukci bychom pak počítali následovně:

```
diff_x = mv_x[instruction]
diff_y = mv_y[instruction]
```

Jak kontrolovat, zda se nějací dva pavouci neocitli na stejném políčku, tedy jak rychle zjistit, zda pole s K dvojice-

[1] Unicode umí vyjádřit většinu světových písem a spoustu věcí navíc. Nechte si o tom někdy na nějakém našem soustředění vyprávět :-)

mi čísel (souřadnicemi) neobsahuje nějakou dvojici dvakrát (nebo i vícekrát)? Na plný počet bodů stačil nejpřímočařejší přístup: prostě vyzkoušíme všech $\mathcal{O}(K^2)$ možných kombinací a ověříme, že žádné dva páry souřadnic nejsou stejné.

Zrychlujeme poprvé

S tím se ale nemusíme spokojit, zvlášť když je časová složitost celého algoritmu dána tím, jak rychle dokážeme kontrolovat srážky, protože zbytek jistě zvládneme v čase lineárním k délce vstupu.

Všimneme si, že když souřadnice vhodně seřadíme, dostanou se duplikáty vedle sebe, takže pak stačí seřazené pole projít a dívat se, zda nejsou nějaké dvě sousední dvojice stejné. Řadit můžeme např. podle tzv. lexikografického uspořádání, které můžete znát třeba ze slovníku. V něm (a, b) zařadíme před dvojici (c, d), pokud $a < c$, nebo $a = c$ a $b < d$. Pomocí nějakého standardního řadicího algoritmu (např. použitím funkce/metody `sort` ve vašem oblíbeném programovacím jazyce) tak dostáváme algoritmus, který zjistí, zda je v poli duplikát, v čase $\mathcal{O}(K \log K)$.

Mohlo by se zdát, že řazením algoritmus rozbijeme, protože po seřazení nevíme, které souřadnice patří ke kterému pavoukovi. Na to je ovšem snadná pomoc – prostě si před každým seřazením pole nakopírujeme a po projití kopii zahodíme. Tím určitě algoritmus zpomalíme nejvýše konstantakrát.

Zrychlujeme podruhé

Umíme to však ještě rychleji. S použitím hešování můžeme dosáhnout průměrné časové složitosti $\mathcal{O}(K)$. Do hešovací tabulky budeme postupně přidávat souřadnice a přitom kontrolovat, zda se nějaké souřadnice nesnažíme přidat podruhé.

V Pythonu je v tomto ohledu velmi příjemné použít `set`, datovou strukturu reprezentující množinu nějakých prvků. Jelikož množina má tu vlastnost, že nemůže obsahovat stejný prvek vícekrát, a tedy po vícenásobném přidání stejného prvku tam prvek pořád bude jen jednou, dává nám `len(set(a))` počet navzájem různých prvků v seznamu `a`.

Zrychlujeme podvaapůlté

„K čemu," říkáte si, „když už máme lineární řešení?" To sice máme, ale jen v průměrném případě. Kdybychom měli velkou smůlu, nebo vstupy dostávali od někoho zlomyslného, může $\mathcal{O}(K)$ přidání prvku do hešovací tabulky trvat až $\mathcal{O}(K^2)$ (kdyby vás zajímalo proč, podívejte se do naší kuchařky o hešování).[2]

V řešení, které je optimální i v nejhorším případě, se vrátíme k myšlence hledání duplicit pomocí třídění. Třídění K prvků sice v obecném případě nemůže být rychlejší než $\mathcal{O}(K \log K)$, naše prvky jsou nicméně dost speciální, jelikož je to K dvojic čísel, kde obě čísla ve dvojici můžou nabývat jen hodnot $1, \ldots, K$.

Použijeme třídící algoritmus zvaný RadixSort, který je v plné obecnosti popsán v naší třídící kuchařce.[3] Základní myšlenka je, že pokud třídíme celá čísla předem známého malého rozsahu (řekněme od 1 do L), umíme je setřídit v čase $\mathcal{O}(N + L)$, kde N je počet čísel, namísto standardního $\Theta(N \log N)$. Schválně si zkuste rozmyslet, jak (nápověda: pořiďte si pole velikosti L). RadixSort pak tuto myšlenku zobecňuje: máme-li N uspořádaných p-tic (tj. například p-rozměrných souřadnic nebo slov pevné délky p znaků), kde každá složka p-tice může nabývat jen L možných hodnot, dokážeme je setřídit v čase $\mathcal{O}(p(N + L))$.

My třídíme K uspořádaných dvojic a obě složky jsou v rozsahu 1 až K, tedy časová složitost RadixSortu je $\mathcal{O}(K)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z2-2.py`

Říša Hladík

30-Z2-3 Klonování pavouků

Popis rodokmenu pavouka byl zadán *rekurentně* – tedy mohl vnořeně obsahovat popis rodokmenů jiných pavouků, konkrétně potomků toho předešlého. Pro zpracování vstupu se tedy přímo nabízí použít rekurzi. Pokud zatím nevíte, co rekurze znamená, určitě se nejprve podívejte do příslušného místa v kuchařce pro začátečníky.[4]

Abychom mohli vypsat rod nějakého pavouka, musíme mít někde uložené všechny jeho předchůdce. Vzhledem k našemu způsobu zápisu rodokmenu však tito předchůdci mohou být v řetězci na vstupu jak nalevo, tak napravo od pavouka. Je tedy jasné, že si nejprve budeme chtít řetězec projít a načíst do paměti záznam o každém pavoukovi. V tomto záznamu chceme mít uložené jeho jméno (znak anglické abecedy) a odkaz na záznam levého a pravého potomka, pokud takoví existují.

Když máme tyto informace v paměti, je jednoduché získat správný výstup. Vytvoříme funkci `vypiš`, která bere dva argumenty: odkaz na záznam pavouka, jehož rod má vypsat, a řetězcový *buffer*, v němž musí být za sebou uložena jména všech předchůdců. Funkce vezme znak představující jméno pavouka a přidá ho na konec bufferu. Pokud pavouk nemá další potomky, tak buffer vypíše; pokud má potomky, tak pro každý jeho záznam zavolá `vypiš` s rozšířeným bufferem. Aby začala rekurze probíhat, zavoláme `vypiš` na předka všech pavouků, bufferem bude prázdný řetězec, protože tento pavouk žádné předchůdce nemá. Je snadno vidět, že výpis proběhne v $\mathcal{O}(N)$ vzhledem k délce vstupu N.

Zbývá tedy vyřešit, jak načíst informace o pavoucích do paměti. Nejprve si řekneme, jak budeme tady zařizovat rekurzi: chceme mít funkci `načti_pavouka`, která přijme nějaké argumenty (zatím nemáme určeno, jaké) a vrátí záznam, ve kterém je správně vyplněné jméno pavouka a odkazy na jeho potomky. Pokud pavouk nemá některého z potomků, tak na místě odkazu bude nějaká neplatná hodnota (např. -1, pokud číslujeme pavouky od nuly a tato čísla využíváme jako odkazy). Pokud pavouk má potomka, potřebujeme získat odkaz na jeho záznam. To provedeme tím, že funkce `načti_pavouka` zavolá rekurzivně sama sebe.

A jak bude funkce konkrétně vypadat? Jednoduchým způsobem je vzít řetězec s popisem rodokmenu pavouka a najít pozici jeho jména. Uvědomte si, že ať už máme rodokmen zadaný jakkoliv, tohle vždy zvládneme během jediného průchodu řetězcem. Složitější je to jen, pokud rodokmen levého potomka obsahuje závorky – pak je musíme počítat a skončit následující pozici po té, kdy se počet levých závorek rovná počtu pravých závorek. Abychom získali odkaz na levého pavouka, zavoláme sami sebe, argumentem teď bude

[2] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`
[3] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`
[4] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni-algoritmy`

	řešitel	*škola*	*ročník*	*sérií*	*Z2-1*	*Z2-2*	*Z2-3*	*Z2-4*	*Z2-5*	*Z2-6*	*série*	*celkem*
57.	Martin Kostrubanič	GČáslav	4	2	8	10					18,0	24,0
58.–59.	Ondřej Buček	GJarošeBO	4	2							0,0	23,0
	Tomáš Husák	GLitoměřPH	4	2							0,0	23,0
60.–61.	Štěpán Henrych	GŽat	2	3	8						8,0	20,0
	Filip Krul	SPŠSmíchov	2	1							0,0	20,0
62.	Tomáš Sláma	GTurnov	3	1							0,0	19,0
63.–68.	Matej Hockicko	TAPoprad	4	2							0,0	18,0
	Petr Hýbl	G UherBrod	2	2	8						8,0	18,0
	Patrik Janko	SPŠSmíchov	2	1							0,0	18,0
	František Kmječ	G Brandýs	2	3							0,0	18,0
	Jakub Ucháč	ŠMaVVzt	2	4							0,0	18,0
	Dávid Šutor	GTerVans	3	3							0,0	18,0
69.–71.	Alexandra Géciová	GJHroncaBA	2	5	2,7	0,3					3,0	16,0
	Jakub Hroník	GJiříPoděb	4	3	8						8,0	16,0
	Bronislav Růžička	GRokycany	−1	2	8						8,0	16,0
72.	Evgenia Golubeva	GJosefskPH	3	1							0,0	15,0
73.	Antonín Rousek	GDašickáPA	0	1							0,0	13,0
74.–75.	Ondřej Daniš	GFPValMez	3	2						4	4,0	12,0
	Vojtěch Žák	GŠpitálsPH	2	1	8	4					12,0	12,0
76.–78.	Martin Sobotka	GLitoměřPH	2	3							0,0	11,0
	Erik Řehulka	ŠPMNDaGB	2	3							0,0	11,0
	Jan Štěch	GJirsíkaČB	1	6	0						0,0	11,0
79.–81.	Tomáš Kratschmer	GMikulášPL	2	1	8	1					9,0	9,0
	Antonín Musil	PORGPha	1	2							0,0	9,0
	Kateřina Vokálová	G Kolín	2	1							0,0	9,0
82.–84.	Václav Kelímek	SPŠBruntál	3	4							0,0	8,0
	Vojtěch Krupka	GJungmanLT	4	2							0,0	8,0
	Adéla Návratová	ZŠ MTyrš	0	2							0,0	8,0
85.	Adam Húšťava	EupSchoolLux	0	1	2,7				5		7,7	7,7
86.	Vít Novotný	GVoděraPH	3	1							0,0	4,0
87.	Vojtěch Kuchař	VOŠJičín	1	9	0						0,0	3,0
88.–91.	Lukáš Caha	GZborovPH	4	3							0,0	1,0
	Jan Hřebenář	20. ZŠ	−1	2							0,0	1,0
	Michal Rod	GJirsíkaČB	3	1							0,0	1,0
	Jakub Zemek	GUHradiště	3	1	1						1,0	1,0

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

Výsledková listina druhé série začátečnické kategorie 30. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z2-1*	*Z2-2*	*Z2-3*	*Z2-4*	*Z2-5*	*Z2-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	132,0
1.	Ondřej Jamelský	G Cheb	0	6	8	10	10	12	12	14	66,0	132,0
2.	Petr Aubrecht	GHeyrovPH	3	6	8	10	10	6	11	14	59,0	122,0
3.	Daniel Skýpala	GTomkovaOL	0	10	8	10	10	8	11	1	48,0	112,0
4.	Petr Budai	G JGJ PH	1	6	8	10	10	2	11	4	45,0	107,0
5.–6.	Michal Bravanský	GBílovec	0	2	8	10	10	8	11	4	51,0	103,0
	Dalibor Kramář	G BO-Řeč	3	5	8	10	10	6	11	0	45,0	103,0
7.–8.	Ondřej Bleha	GBNěmcovHK	3	2	8	10	10	6			34,0	98,0
	Jiří Kvapil	GTomkovaOL	0	6	8	10	10	6	11	13	58,0	98,0
9.	Klára Tauchmanová	GOhradníPH	4	2	6	10		6	12		34,0	87,0
10.	Filip Kastl	GKepleraPH	2	3	8	10	6,7	8	5		37,7	86,7
11.–12.	Terézia Strišovská	GJHroncaBA	2	6	8	10	8,7	9,3			36,0	76,0
	Jan Vodstrčil	G VMýto	1	5	8	10		6	5	1	30,0	76,0
13.–14.	Ondřej Hráček	GOlgHavl	1	3	8	10	3	10	5		36,0	74,0
	Michal Kodad	SPŠSmíchov	2	5	8	10	10	6			34,0	74,0
15.	Janek Hlavatý	GJirsíkaČB	−1	14	8	8			6	13	35,0	66,0
16.	Jakub Šuráň	GStrážnice	3	5							0,0	62,0
17.	Vojtěch Káně	G Brandýs	2	10	8						8,0	61,0
18.	Martin Zmitko	G FrýdlNOs	2	6	8	10		6			24,0	58,0
19.	Martin Bencko	GOhradníPH	1	10	4	10		10		2	26,0	57,0
20.	Jakub Komárek	GUHradiště	3	2	8	10	10	4	12		44,0	56,0
21.–22.	Robert Jaworski	GÚstavníPH	0	9	8	10	10	4			32,0	55,0
	Lucie Vomelová	GŠpitálsPH	2	5	8	2	1	2	10	4	27,0	55,0
23.	Michal Mlčoch	G UherBrod	3	7	6	10		8	12		36,0	54,0
24.–26.	Radim Buráň	G UherBrod	3	5	8	10		6	5		29,0	52,0
	Jiří Tlamicha	GŘíč	1	2	8	10	5				23,0	52,0
	Matěj Volf	GCoubTábor	0	2	8	10		6			24,0	52,0
27.–28.	Erik Berta	GAlejKošice	3	6	8	10	1	0			19,0	51,0
	Lukáš Gáborik	GTajBanBys	1	2	8	10		4			22,0	51,0
29.	Vojtěch Poupa	CírkG Plzeň	0	3	8				2	0	10,0	49,0
30.	Jan Kotovský	GPísnickáPH	−1	6	8	9	5	4	3	2	31,0	48,0
31.	Matyáš Lorenc	GJungmanLT	4	2	8	10		6			24,0	47,0
32.	Michal Starý	GNoMěsNMor	4	1							0,0	46,0
33.	Jan Černý	BiGy Žďár	2	2	8			8			16,0	44,0
34.	Jakub Nevařil	G UherBrod	0	2	8	10		6			24,0	42,0
35.	Jiří Vlček	GFXŠaldyLI	2	2	8	5					13,0	41,0
36.	Jaroslav Paška	ŠPMNDaGB	3	1							0,0	39,0
37.	Dávid Oravec	G DubNVáh	3	5	8	10		6			24,0	37,0
38.–40.	Kristina Galikova	ŠPMNDaGB	3	2	8						8,0	36,0
	Albert Kučera	GNadŠtolPH	2	1							0,0	36,0
	Vojtěch Michal	GNVPlániPH	3	2	8						8,0	36,0
41.	Jan Koška	GJírovcČB	−2	5	8	10		6			24,0	35,0
42.	Václav Zvoníček	GJarošeBO	2	2	8	10		3,4			21,4	34,4
43.	Jakub Ferenčík	GDašickáPA	0	2	8	10		4			22,0	33,0
44.–45.	Radoslav Hašek	GČáslav	4	10	8	10		6			24,0	32,0
	Radim Kopunec	G UherBrod	−2	2	8			6			14,0	32,0
46.	Ondřej Wrzecionko	GTěš	3	6	8						8,0	31,0
47.–48.	Michael Kozel	GZborovPH	4	9	4						4,0	29,0
	Dennis Pražák	GJirsíkaČB	3	6							0,0	29,0
49.–51.	Martin Cmiel	GOlgHavl	1	1							0,0	28,0
	Dominik Dinh	GNVPlániPH	3	4							0,0	28,0
	Jindřich Dítě	VOSPŠŽďár	2	6							0,0	28,0
52.–54.	Jiří Bleha	SPSEPard	1	2	8						8,0	26,0
	Ondřej Gonzor	G Brandýs	1	8	8			6	12		26,0	26,0
	Jan Kučera	SŠKlatovskáPL	1	6	8						8,0	26,0
55.–56.	Vladimír Chudý	ZŠRonov	1	5							0,0	25,0
	Anna Hollmannová	GSRandyJN	1	8							0,0	25,0

podřetězec s rodokmenem levého pavouka (nalevo od jména našeho pavouka). Pro získání odkazu na pravého pavouka postupujeme analogicky.

Ačkoliv je tento přístup evidentně funkční, není příliš efektivní. Pro každého pavouka musíme projít celý jeho rodokmen a pro všechny dohromady to může zabrat až kvadraticky mnoho oproti délce vstupu. Můžete si to rozmyslet např. pro vstup `(_a(_b(_c(_de))))`, kde kromě posledního pavouka má každý jen pravého potomka.

Ukážeme si lepší metodu, ve které nám bude stačit jediný průchod řetězcem na vstupu pro načtení informací o všech pavoucích. Dokonce nemusíme mít řetězec uložený v paměti, místo toho budeme postupně načítat jednotlivé znaky. Funkce `nacti_pavouka` tentokrát nepotřebuje žádné vstupní parametry. Předpokládá, že znak, který je právě na vstupu, je prvním znakem rodokmenu pavouka.

Při spuštění se podívá na znak, který je aktuálně na vstupu. Pokud je znakem písmeno anglické abecedy, jde o pavouka bez potomků, takže stačí vrátit záznam s doplněným jménem. Pokud je znakem podtržítko, vrátíme informaci, že na této pozici žádný pavouk není. Složitější situace nastává, pokud je znakem závorka. Pak máme pavouka s (alespoň) levým potomkem, takže nejprve obsloužíme toho potomka – načteme nový znak, čímž se dostaneme na začátek jeho rodokmenu, a rekurzivně zavoláme `nacti_pavouka`. Následně se na vstupu musí vyskytovat písmeno označující jméno našeho pavouka (jinak je vstup chybný), pak znovu načteme nový znak a znovu zavoláme `nacti_pavouka`, čímž si načteme pravého potomka. Pak už jen stačí vrátit vytvořený záznam.

Uvědomte si, že tentokrát mají všechny rekurzivně spuštěné funkce společnou věc, totiž znak na vstupu. Pokud pouštíme `nacti_pavouka`, víme, že si funkce čte vstup, dokud nenačte rodokmen, a že jakmile skončí, bude na vstupu první následující znak po rodokmenu. Tím pádem je načítání, stejně jako výpis, v $\mathcal{O}(N)$ a my jsme tím pádem obdrželi lineární algoritmus.

Při rozboru paměťové složitosti rekurzivních algoritmů si zapamatujte, že každé vnořené volání funkce zabere nějaký prostor v paměti, tedy pokud nějakou funkci voláte vnořeně d-krát, bude spotřebovávaná paměť alespoň v $\mathcal{O}(d)$. V nejhorším případě budeme naše funkce volat vnořeně tolikrát, kolik je celkový počet pavouků, což je ale znovu omezené velikostí vstupu.

Program (C++):
`http://ksp.mff.cuni.cz/viz/30-Z2-3.cpp`

Kuba Maroušek

30-Z2-4 Příliš bílý displej

Hned zkraje přiznáváme, že tato úloha byla o něco zákeřnější než je na začátečnickou kategorii zvykem. V zadání totiž nebylo řečeno, v jakých rozmezích se pohybuje velikost displeje a počet příkazů, natožpak v jakém jsou vztahu. Takovou příležitost museli organizátoři chytit za pačesy a pořádně ji využít. Prvních pět vstupů, které jsme generovali, používalo *rozumné* velikosti plochy a škálovalo v počtu příkazů. K jejich vyřešení v časovém rozmezí tedy bylo potřeba si poradit s vysokým počtem příkazů. Poslední vstup byl ale jiný – příkazů bylo relativně málo, zato plocha byla obrovská tak, že veškeré pokusy o zapamatování stavu displeje selhávaly.

K vyřešení posledního vstupu bylo tedy potřeba použít pravděpodobně jiný algoritmus, než na předchozí vstupy. Na naší obranu, používat jeden univerzální program vás v opendata úlohách nikdo nikdy nenutí. Navíc, takovéto zákeřnosti bychom se nedopustili, kdybyste neměli vstupy k dispozici, a nemohli se na jejich parametry podívat.

Běžným postupem v případě, že byste chtěli mít univerzální program, je za běhu se podívat na parametry vstupu (nejen proto je píšeme hned na začátek) a podle nich se rozhodnout, kterým algoritmem vstup půjde vyřešit. Nebo, pokud si to můžete dovolit, spustit oba najednou – a výsledek vzít od toho, který doběhne rychleji. To je ale zase trochu jiná pohádka. Pojďme se vrhnout na možná řešení.

Řešení pro rozumně velkou plochu displeje

Pro snazší pochopení si nejprve zkusíme úlohu vyřešit na jednořádkovém displeji. Mějme obyčejné jednorozměrné pole, které si na začátku naplníme nulami. S každým požadavkem na rozsvícení úseku pak projdeme všechny prvky odpovídající úseku, a uložíme si do pole jedničky. Takovéto přímočaré řešení je ale samozřejmě příliš pomalé – každý příkaz může rozsvítit řádově celý displej.

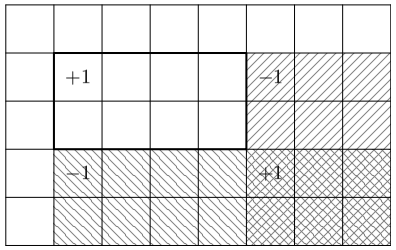
Půjdeme na to tedy trochu jinak: v každém prvku pole si budeme počítat, kolik úseků v něm začíná a kolik končí. Potom je každý příkaz jen otázkou zvýšení dvou počítadel na správných místech. Jakmile provedeme každý příkaz, projedeme displej zleva, a budeme si udržovat proměnnou L, kolikrát byl tento pixel rozsvícen. Pokud je L nenulová, pixel je rozsvícen, a do celkového součtu tedy přičteme jedničku. Je potřeba si rozmyslet, že musíme nejdříve přičíst začátky, potom se na hodnotu podívat, a potom teprve odečíst konce.

Aplikováním tohoto jednorozměrného řešení na každý řádek displeje už jste mohli získat nějaký ten bod navíc. Ještě než se vrhneme do jeho zdvourozměrnění, vyřešíme technickou komplikaci s dvěma počítadly. Všimněte si, že při pohybu mezi pixely (tj. s výjimkou prvního a posledního) děláme to, že odečteme úseky končící na pixelu i, a vzápětí přičteme začátky na pixelu $i+1$. Proč tedy rovnou nezapočítat konec jako -1 začátek o pixel dál? Pro úsek (a, b) tedy přičteme do pole jedničku na indexu a, a odečteme jedničku na indexu $b+1$. V poli se nám tedy objeví i záporná čísla, ale to ničemu nevadí.

To, co nyní v závěru program dělá, nápadně připomíná počítání prefixových součtů. A vskutku, řešení úlohy je vlastně počítání v rozdílech tak, aby po spočítání prefixových součtů vyšel výsledek. Kdo tedy ovládá 2D prefixové součty, pravděpodobně může následující odstavce přeskočit.

V jednorozměrné variantě s plus a mínus jedničkami jsme se mohli na čísla podívat tak, že „odtud doprava je vše rozsvíceno o jedna vícekrát (méněkrát)“. Ve dvourozměrném případě se nám tato intuice bude velice hodit. Pořídíme si tentokrát už dvourozměrné pole reprezentující celý displej. Každý prvek bude označovat, o kolik více/méněkrát je rozsvícena oblast od tohoto pixelu doprava dolů. Pojmenujme si tuto oblast *čtvrtrovina* s počátkem v daném pixelu. Jak si ale správně poznačit počátky čtvrtrovin, aby nám to vyšlo?

Začneme tím nejjednodušším – do levého horního rohu oblasti přičteme jedničku. V pravém horním a levém dolním jedničku odečteme. Protože jsme tím ale čtvrtrovinu s počátkem v pravém dolním rohu odečetli dvakrát, tak ji zase jednou přičteme zpátky. Pro lepší představu přikládáme obrázek.



Jak z tohoto ale na konci vykoukat výsledek? Nejjednodušší možností je spočítat prefixové součty nejprve pro sloupce, potom pro řádky. Prvním sečtením vlastně vyrobíme R instancí jednorozměrného řešení pro každý řádek zvlášť. Kdo by stál o řešení pouze jedním průchodem tak si rozmyslí, že může použít podobný postup jako při konstrukci rozdílového pole. Procházíme-li pixely postupně shora dolů a zleva doprava, tak výsledná hodnota každého pixelu je součtem hodnot pixelu bezprostředně vlevo a nad, snížená o hodnotu pixelu diagonálně vlevo nahoře – protože jsme ji započítali dvakrát.

Toto popisované řešení potřebuje $\mathcal{O}(RS+K)$ času. Inicializace a konečné sčítání seběhne v $\mathcal{O}(RS)$, pro každý příkaz pak děláme konstantní práci, tedy v součtu $\mathcal{O}(K)$. Paměti však potřebujeme $\mathcal{O}(RS)$, což je pro poslední vstup příliš.

Řešení pro nerozumně velkou plochu

Předchozí řešení stavělo na tom, že si zapamatovalo stav celého displeje. Pokud je ale plocha příliš velká, nezbývá než se tomu prostě vyhnout. Základním kamenem našeho druhého řešení bude, že příkazy se vzájemně neovlivňují – můžou proběhnout v libovolném pořadí, a výsledek se nezmění. Toto by například neplatilo, kdybychom mohli nejen rozsvěcet, ale i zhasínat. (Pečlivější si rozmyslí, že toto jsme potřebovali i v předchozím řešení.)

Použijeme postup, který se používá v geometrických algoritmech – zametání. Kdyby toto byla úloha v hlavní kategorii, pravděpodobně bychom nestihli ani zkoumat každý řádek zvlášť – takto si ale situaci komplikovat nebudeme, bude nám stačit jednodušší varianta, ve které zameteme každý řádek zvlášť.

Pro každý řádek potřebujeme spočítat, kolik pixelů v něm svítí. Abychom to dokázali zvládnout rychle, seřadíme si příkazy podle levého kraje (tj. souřadnice X_1). Pořadí mezi příkazy, které mají shodný levý sloupec, už může být libovolné.

V každém řádku Y proletíme pixely zleva doprava. Nebudeme si ale všímat příkazů, které se netýkají řádku Y, tj. těch kde $Y > Y_2$ nebo $Y < Y_1$ – vždy je rovnou přeskočíme a v dalším textu už je nebudeme uvažovat. Zbylé příkazy zůstávají seřazeny dle začátku od nejlevějšího. Označme si proměnnou X bod, od kterého už počítáme pixely jen vpravo. Na začátku je $X = 0$. S každým příkazem se nejprve podíváme, jestli náhodou X_1 není větší než X. Pokud ano, pixely $X \ldots X_1$ můžeme prohlásit za zhasnuté, protože se jich už určitě žádný další příkaz netýká, nastavíme tedy $X = X_1$. Poté nám oblast určuje nějaké pixely, které můžeme prohlásit za rozsvícené: pokud je $X_2 > X$, pak jsme našli $X_2 - X$ nových pixelů, a posuneme $X = X_2$.

Všimněte si, že X pouze rostlo, a navíc nabývalo pouze hodnot, které odpovídají krajům oblastí, kterých je maximálně K. Hlavní část algoritmu tedy potřebuje $\mathcal{O}(RK)$ času. Započítáme-li zároveň čas na seřazení příkazů, vychází nám čas $\mathcal{O}(K \log K + RK)$. Paměti pak potřebujeme jen $\mathcal{O}(K)$.

Na závěr dodáme, že tato úloha má nespočet principiálně různých řešení, které si liší svými složitostmi (časovou, paměťovou i tou na pochopení). Mnohé z nich jsou vzájemně neporovnatelné bez znalosti vztahů mezi parametry R, S a K. Budeme rádi, pokud se nám například na fóru pochlubíte, jakým řešením a v jakém jazyce jste dosáhli svého počtu bodů.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z2-4.py`

Program – zametací (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z2-4-sweep.py`

Ondra Hlavatý

30-Z2-5 Spokojenost s výstavou

Máme najít, u jakých zvířat měli Kevin a Zuzka stejnou „spokojenost“. Nejlepší tedy bude si tuto spokojenost pro oba dva spočítat. Pro připomenutí, spokojenost u i-tého zvířete je součet hodnocení všech předchozích zvířat.

Spočítat spokojenost u každého zvířete zvládneme pro oba dva v lineárním čase – pro každého z nich si projdeme jeho seznam hodnocení a budeme si průběžně držet aktuální spokojenost. Pro každý záznam přičteme hodnocení k aktuální spokojenosti a zapíšeme si výsledek do pole. Tak nám vzniknou dvě pole – seznam spokojeností Zuzky a seznam Kevina.

Teď jenom potřebujeme zjistit, jestli je v polích nějaké stejné číslo. To můžeme zjistit mnoha různými způsoby (viz řešení druhé úlohy, jenom pozor, že to není úplně stejný problém).

Jestliže oba seznamy seřadíme a zapamatujeme si původní pořadí, dostaneme dva rostoucí seznamy. Poté můžeme použít řešení pro totemy z předchozí série, akorát při posunech místo přičítání vezmeme nové číslo. Tady nás ale brzdí řazení, trvající $\mathcal{O}(N \log N)$.

V tomto případě vyjde optimálně použít hešovací tabulku. Projdeme Kevinův seznam spokojeností a zapíšeme si každou položku do tabulky, kde jako klíč použijeme spokojenost a jako hodnotu použijeme index v poli (ten pak potřebujeme vypsat). Pak projdeme Zuzčin seznam spokojeností a pro každou položku zkontrolujeme, jestli je v Kevinově tabulce – pokud ji tam najdeme, tak máme místo se stejnou spokojeností a můžeme vypsat výsledek, jinak pokračujeme dál. Pozor na to, že v druhém průchodu do tabulky nezapisujeme – dvě místa, kde měla Zuzka stejnou spokojenost, nehledáme.

Jakou to bude mít složitost? Na začátku si předpočítáme spokojenosti, bude to trvat $\mathcal{O}(N)$ času (N je počet vystavených zvířat) a budeme na to potřebovat $\mathcal{O}(N)$ paměti. Pak projdeme Kevinův seznam a provedeme $\mathcal{O}(N)$ vložení do hešovací tabulky. Každé trvá v průměrném případě $\mathcal{O}(1)$, takže to celkem potrvá průměrně $\mathcal{O}(N)$ a spotřebuje $\mathcal{O}(N)$ paměti. Projití Zuzčiných spokojeností bude podobné – provede $\mathcal{O}(N)$ vyhledání v hešovací tabulce, která mají taky průměrnou složitost $\mathcal{O}(1)$, a tak to celé bude dohromady trvat $\mathcal{O}(N)$ v průměrném případě.

Ještě poznámka: Pokud jste zrovna přečetli řešení druhé úlohy a myslíte si, že by toto také šlo pomocí RadixSortu zrychlit na $\mathcal{O}(N)$ v každém případě, tak to není tak jednoduché. V této úloze totiž nemáme garantovaný rozsah čísel na vstupu a RadixSort potřebuje řádově stejně paměti, jako je velikost rozsahu čísel, takže jej v této úloze bohužel nelze použít.

Standa Lukeš

30-Z2-6 Skořápky

Byli jsme postaveni před úkol poznat, jaký tah v záznamu jedné partie skořápek byl přidán navíc – po odstranění tohoto tahu by měly ostatní zaznamenané tahy správně vést k přesunu kuličky z počáteční do koncové pozice.

Jako první se pojďme zamyslet, jak vlastně simulovat tahy. Tahy máme zadané jako dvojice čísel X a Y znamenající, že se prohodí X-tý a Y-tý kelímek zleva. Mohli bychom si udělat pole o velikosti N (kde N je počet kelímků) a při každé operaci bychom mohli prohodit obsah dvou indexů v poli – ale to je zbytečně pracné.

Kelímky jsou pro nás v podstatě nerozlišitelné, takže nám stačí si jenom pamatovat, kde se zrovna nachází kulička. Pokud se kulička nachází v kelímku, který se právě účastní tahu, tak se nám kulička přesune do druhého kelímku, jinak se nám s kuličkou nestane nic.

Protože simulace každého tahu nám zabere jenom jednu operaci, tak odsimulovat záznam hry obsahující T tahů nám zabere jenom $\mathcal{O}(T)$.

Teď k samotné úloze: Jak poznat, který tah je v záznamu navíc? První, co by nás mohlo napadnout, je zkusit všechny možnosti. Můžeme si zkusit odsimulovat hru, když vynecháme první tah, když vynecháme druhý tah, … Zkrátka máme T možností k odsimulování. Už ale víme, že jedno odsimulování by nám zabralo čas $\mathcal{O}(T)$, a tak bychom tímto primitivním způsobem vyřešili úloha v čase $\mathcal{O}(T^2)$, což je příliš pomalé.

Jak to zrychlit? Zkusme na to jít z obou stran zároveň – můžeme spočítat, kam se kulička dostane po provedení prvních i tahů ze startovní pozice, a naopak můžeme spočítat, kde by kulička musela být před posledními j tahy, aby se dostala do cílové pozice. A když se někde potkáme kuličkou na stejném místě po i-tém tahu a před $(i+2)$-tým tahem, tam můžeme $(i+1)$-tý tah vyloučit jako chybný a dostaneme fungující zápis hry.

Začneme tím, že si pořídíme pole velikosti N, do kterého si budeme zapisovat, v jakých tazích se kulička dostala pod které kelímky. Protože se kulička může na některé pozice dostat vícekrát během hry, budou v jednotlivých políčkách pole bydlet seznamy. Pak už jen provedeme simulaci, jako jsme popsali výše, a po každém tahu přidáme číslo tohoto tahu do odpovídajícího políčka pole.

Druhý krok je pak vydat se odzadu. Vyjdeme z koncové pozice a budeme postupně simulovat hru odzadu a počítat si, kde by musela být kulička před i-tým tahem. Po každém tahu se podíváme, jestli náhodou v odpovídajícím políčku pole není zapsáno číslo $i-2$. Pokud ano, tak to znamená, že po vyloučení $(i-1)$-tého tahu již hra funguje a našli jsme výsledek.

Ještě potřebujeme dořešit jednu drobnost – políčko může obsahovat být více záznamů v seznamu a procházet je všechny by trvalo dlouho. Ale protože záznamy byly přidávány v rostoucím pořadí a nám při hledání od konce čísla tahů jenom klesají, tak můžeme jakákoliv vyšší čísla, než je $i-2$, vyhodit. Čísel celkově vyhodíme nejvýše tolik, kolik jsme jich přidali, a proto nám to časovou složitost nepokazí.

Na úvodní vyrobení prázdného pole potřebujeme čas $\mathcal{O}(N)$, na jeho úvodní naplnění čas $\mathcal{O}(T)$ a na zpětné prohledávání také čas $\mathcal{O}(T)$. Celkově jsme tak úlohu vyřešili s časem $\mathcal{O}(N+T)$.

Poznámka na závěr: Pokud by náhodou bylo skořápek velké množství a tahů by bylo málo, přesněji pokud by platilo $T < \sqrt{N}$, tak by bylo výhodnější provést T simulací v čase $\mathcal{O}(T^2) < \mathcal{O}(N)$. Ale je to spíše okrajový případ.

Jirka Setnička