Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

30. ročník | **KSP-Z** | **Prosinec 2017**

Milí řešitelé, milé řešitelky a milá řešitelčata!

Doufáme, že jste si užili prázdniny, a jsme rádi, že jste na jejich program zařadili i řešení úloh, které jsme na vás přichystali. V tomto letáku najdete jejich autorská řešení.

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

Řešení první série začátečnické kategorie 30. ročníku KSP

30-Z1-1 Kevinova nepatnáctka

Pro sledování pohybu prázdného místa v krabičce $M \times N$ pochopitelně nebylo nutné si pamatovat stav každého pole v krabičce – stačilo si vytvořit dvě číselné proměnné m a n a v nich si držet pozici prázdného místa. Při provádění posunů pak bylo třeba proměnné správně aktualizovat, a protože krabička má omezené rozměry, kontrolovat, že jsme z ní „nevypadli“.

Výchozí hodnoty m a n dostaneme na vstupu. Pak postupně procházíme znaky posunu, ale před úpravou m a n si vždy zkontrolujeme, zda lze posun uskutečnit. To můžeme provést například tak, že si vytvoříme proměnné d_m a d_n, do kterých uložíme rozdíl nové a staré pozice (například pro znak `<` bude $d_m = 0, d_n = -1$). Pak si zkontrolujeme, že $m + d_m$ je ve správném rozsahu mezi jedničkou a M a podobně $n + d_n$ v rozsahu mezi jedničkou a N. Pokud tomu tak je, aktualizujeme m a n.

Z programátorského hlediska mohou být zajímavé různé možnosti, jak v závislosti na znaku posunu správně přiřadit hodnoty do d_m a d_n. Autorské řešení, psané v Pythonu, používá konstrukce `if` a `elif` (else-if). Předpokládáme, že v proměnné `move` je uložen znak posunu:

```
diff_x = 0
diff_y = 0
if move == "^":
        diff_y = -1
elif move == "v":
        diff_y = 1
elif move == "<":
        diff_x = -1
elif move == ">":
        diff_x = 1
else:
        print("Chyba vstupu")
```

Ale například v mnoha jiných jazycích můžete použít konstrukci `switch`, která rovnou počítá s tím, že rozhodujete na základě jediné proměnné. Třeba v jazyce C:

```
int diff_x = 0;
int diff_y = 0;
switch (move) {
        case '^':
                diff_y = -1;
                break;
        case 'v':
                diff_y = 1;
                break;
        case '<':
                diff_x = -1;
                break;
        case '>':
                diff_x = 1;
                break;
        default:
                printf("Chyba vstupu\n");
}
```

Pokud by možných posunů bylo více, mohlo by se vyplatit si pořídit *slovník*, datovou strukturu, kde budou pohyby uloženy pod příslušným znakem. Při procházení pohybu se pak jenom podíváte na příslušné místo do slovníku. Takhle by mohla vypadat inicializace na začátku programu (tentokrát znovu v Pythonu):

```
moves_x = {'^':  0, 'v': 0, '<': -1, '>': 1}
moves_y = {'^': -1, 'v': 1, '<':  0, '>': 0}
```

a následné použití:

```
diff_x = moves_x[move]
diff_y = moves_y[move]
```

Pro více informací se můžete podívat do naší hešovací kuchařky.[1] Protože je však pokročilejší, ujistěte se nejprve, že jste dobře pochopili obsah kuchařky pro začátečníky.[2]

Ať už použijeme jakýkoliv výše uvedený postup, časová složitost bude $\mathcal{O}(T)$, kde T je počet posunů. Paměťová složitost bude konstantní – udržujeme si velikost krabičky a aktuální pozici prázdného místa, nic navíc.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z1-1.py`

Kuba Maroušek

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

30-Z1-2 Sářiny loutky

Tato úloha se dala řešit tzv. hladovým algoritmem: tak říkáme algoritmům, které jdou během výpočtu „rovnou za nosem“ a příliš si nelámou hlavu s tím, že to, co se nyní jeví jako nejlepší nápad, se nám později může vymstít.

Řekněme, že jako první se budeme zaobírat nejtěžší loutkou. Tu určitě nakonec někdy musíme pověsit, a pro výsledek je jedno, jestli to uděláme teď, nebo později.

Označme nejtěžší loutku jako T a nejlehčí loutku jako L. *Přípustné* budeme říkat všem loutkám, které můžeme s T pověsit, aniž by byla překročena kapacita věšáku. Podíváme se tedy, s jakou další loutkou nejtěžší loutku pověsíme. Může se stát, že kvůli nosnostem věšáků není k T žádná loutka přípustná, pak nemáme na výběr a musíme ji pověsit samotnou. V opačném případě bychom intuitivně chtěli k T pověsit z přípustných loutek co nejtěžší, abychom si co nejvíce ulehčili práci s rozvěšováním zbylých loutek. Ukáže se však, že ve skutečnosti na váze druhé loutky nezáleží a můžeme si z přípustných loutek vybrat třeba tu nejlehčí.

Jak se taková věc dokazuje? Například můžeme ukázat, že když existuje nějaké řešení, ve kterém jsme „donuceni“ dát k T nějakou jinou loutku, pak musí existovat stejně dobré řešení, ve kterém k T můžeme dát nejlehčí loutku L.

Řekněme, že máme řešení, ve kterém k naší loutce T pověsíme nějakou loutku A. Chceme toto řešení umět upravit tak, abychom k T pověsili L. To však umíme: stačí se podívat, na jaký věšák jsme L v tomto řešení umístili, a pak A a L prohodit. Rozmyslíme si, že tím nosnosti věšáků nepokazíme: Ať jsme A prohozením umístili kamkoliv, kapacitu nepřekročíme, protože když se A vešlo vedle nejtěžší loutky, vejde se vedle libovolné jiné. Na věšáku, na kterém je T, kapacitu také nepřekročíme, protože L je lehčí než A, takže T s L je lehčí než T s A.

Dostáváme hrubý obrys funkčního algoritmu: vezmeme nejlehčí loutku L a nejtěžší loutku T a zkusíme, zda $L+T$ překročí nosnost věšáku. Pokud ano, musíme T pověsit samotnou (nevejde se k nejlehčí loutce, takže ani k žádné jiné), pokud ne, pověsíme T s L. V obou případech pověšené loutky odstraníme a postup opakujeme na zbylé loutky, dokud máme ještě nějaké nepověšené.

Všechny loutky si tedy na začátku uložíme do pole, které seřadíme vzestupně podle hmotnosti. Dokud není pole prázdné, opakovaně z něj vybíráme nejtěžší (tedy poslední) loutku a věšíme ji buď s první (nejlehčí) loutkou, nebo samotnou. Po každém pověšení odstraníme poslední prvek pole a případně i první, pokud se na věšák vešly obě loutky.

Takový algoritmus je však příliš pomalý, protože odstranění prvku ze začátku pole nás pro N loutek stojí $\mathcal{O}(N)$ času a těchto odstranění provedeme až $\mathcal{O}(N)$.

Můžeme však časovou složitost snadno zlepšit: to, co nás stojí hodně času, je opakované odstraňování prvků ze začátku seznamu. To se ale chová celkem předvídatelně: po k odstraněních ze začátku bude na prvním místě v seznamu prvek, který byl na začátku programu na $(k+1)$-tém místě. Podobně se chová odstraňování z konce pole.

Nemusíme tedy z okrajů seznamu vůbec odstraňovat, stačí pracovat s původním polem a pamatovat si navíc ukazatele na nejlevější a nejpravější ještě nepověšenou loutku, které zvýšíme resp. snížíme o jedna vždy, když jsme v původním algoritmu odstraňovali prvek ze začátku resp. konce seznamu. Také se nám změní podmínka ve `while` cyklu, protože už netestujeme, zda je seznam prázdný, nýbrž zda je výřez pole určený našimi dvěma ukazateli prázdný (tedy zda se první ukazatel dostal za druhý).

Takto upravený algoritmus už pro každý věšák vykoná jen konstantně mnoho práce: podívá se na dva indexy do pole, spočte součet dvou hmotností a posune nejvýše oba ukazatele o jedna. Jelikož je však věšáků řádově tolik, co loutek (určitě jich není méně než $N/2$ a více než N), stráví algoritmus věšením loutek $\mathcal{O}(N)$ času. Nesmíme však zapomenout na seřazení, které na začátku musíme provést. K tomu můžeme použít libovolný rychlý třídící algoritmus, například některý z naší třídící kuchařky,[3] ve které se mimo jiné dočtete, že za rychlé považujeme třídící algoritmy, které pro obecná data běží v čase $\mathcal{O}(N \log N)$. To je tedy zároveň celková časová složitost našeho programu. Paměťová složitost je $\mathcal{O}(N)$, protože nám stačí pamatovat si všechny loutky v poli, případně s nějakým čítačem počtu použitých věšáků.

Program (C++):
`http://ksp.mff.cuni.cz/viz/30-Z1-2.cpp`

Říša Hladík

30-Z1-3 Petrovo luštění zprávy

Naším cílem je najít pro každou pozici v textu nejčastější znak z několika zpráv. Ukážeme si několik způsobů, jak na to, a porovnáme jejich efektivitu. Zpráv je N a jejich délku nazvěme například T.

Nabízí se vytvořit dvourozměrné pole velikosti $26 \times T$ s počty výskytů znaků, kde každý řádek odpovídá jednomu z písmen abecedy a sloupeček pozici v textu. Políčko v poli tedy bude odpovídat počtu výskytů daného znaku na dané pozici v textu.

Pokud nyní budeme procházet přes políčka našeho pole a snažit se vypočítat jejich hodnoty, tak skončíme s nepříliš efektivním algoritmem. Abychom vyplnili hodnotu v políčku pro daný znak a pozici v textu, tak musíme z každé zprávy přečíst jeden znak, tedy celkem projdeme N znaků. Toto budeme muset udělat pro každé políčko, kterých je $26 \times T$, a tedy celkem při vyplňování tabulky provedeme $26 \times T \times N$ operací.

Půjdeme na to jinak, vyhneme se opakovanému čtení zpráv. Pokud máme na začátku toto velké pole vynulované, tak můžeme postupně procházet text zpráv ze vstupu. Pro každý přečtený znak známe jeho pozici a o jaký znak jde, tedy můžeme v tabulce do patřičného políčka přičíst 1. Tento postup provede pouze $T \times N$ operací a také nám vyplní tabulku. Zároveň víme, že přesně tolik má znaků vstup, tedy tato část rychleji ani nepůjde.

Poté, co máme takovéhle pole, nám stačí projít všechny sloupce a v každém najít pozici maxima. Pak na vstup můžeme vypsat znaky, kterým tato maxima přísluší.

Zbavme se ještě potřeby velkého pole. Jeho velikost je sice $\mathcal{O}(N)$, ale vystačíme si s polem velikosti jenom 26. Pro zjištění jednoho znaku pro výstup totiž potřebujeme jen informace z jednoho sloupečku. Můžeme tedy jen projít znaky všech zpráv na té pozici a spočítat si četnost výskytů. Pak vybereme maximum, vypíšeme znak, vyčistíme pole a přesuneme se na další pozici.

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

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
58.	Jan Kotovský	GPísnickáPH	−1	5	5	2	10				17,0	17,0
59.	Evgenia Golubeva	GJosefskPH	3	1	8		7				15,0	15,0
60.–63.	Alexandra Géciová	GJHroncaBA	2	4	8	5					13,0	13,0
	Dávid Oravec	G DubNVáh	3	4	8	5					13,0	13,0
	Antonín Rousek	GDašickáPA	0	1	8	5					13,0	13,0
	Václav Zvoníček	GJarošeBO	2	1	8	5					13,0	13,0
64.–65.	Štěpán Henrych	GŽat	2	2	8	4					12,0	12,0
	Jakub Komárek	GUHradiště	3	1	8	4					12,0	12,0
66.–70.	Jakub Ferenčík	GDašickáPA	0	1	8	3					11,0	11,0
	Jan Koška	GJírovcČB	−2	4	1		10				11,0	11,0
	Martin Sobotka	GLitoměřPH	2	3	8	3					11,0	11,0
	Erik Řehulka	ŠPMNDaGB	2	3	8	3					11,0	11,0
	Jan Štěch	GJirsíkaČB	1	5	8		3				11,0	11,0
71.	Petr Hýbl	G UherBrod	2	1			10				10,0	10,0
72.–73.	Antonín Musil	PORGPha	1	2	8	1					9,0	9,0
	Kateřina Vokálová	G Kolín	2	1	8	1					9,0	9,0
74.–80.	Ondřej Daniš	GFPValMez	3	1	0				8		8,0	8,0
	Radoslav Hašek	G Čáslav	4	9	8						8,0	8,0
	Jakub Hroník	GJiřiPoděb	4	2	8	0					8,0	8,0
	Václav Kelímek	SPŠBruntál	3	4	8						8,0	8,0
	Vojtěch Krupka	GJungmanLT	4	2	8						8,0	8,0
	Adéla Návratová	ZŠ MTyrš	0	2	8						8,0	8,0
	Bronislav Růžička	G Rokycany	−1	1	8						8,0	8,0
81.	Martin Kostrubanič	G Čáslav	4	1	1	5					6,0	6,0
82.	Vít Novotný	GVoděraPH	3	1	3	1					4,0	4,0
83.	Vojtěch Kuchař	VOŠ Jičín	1	8	3	0					3,0	3,0
84.–86.	Lukáš Caha	GZborovPH	4	3	1						1,0	1,0
	Jan Hřebenář	20. ZŠ	−1	2	1						1,0	1,0
	Michal Rod	GJirsíkaČB	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 první série začátečnické kategorie 30. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z1-1*	*Z1-2*	*Z1-3*	*Z1-4*	*Z1-5*	*Z1-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	66,0
1.	Ondřej Jamelský	G Cheb	0	5	8	10	10	12	12	14	66,0	66,0
2.–3.	Ondřej Bleha	GBNěmcovHK	3	1	8	10	10	12	10	14	64,0	64,0
	Daniel Skýpala	GTomkovaOL	0	9	8	10	10	12	10	14	64,0	64,0
4.	Petr Aubrecht	GHeyrovPH	3	5	8	10	10	12	9	14	63,0	63,0
5.–6.	Petr Budai	G JGJ PH	1	5	8	10	10	12	10	12	62,0	62,0
	Jakub Šuráň	GStrážnice	3	5	8	10	10	12	8	14	62,0	62,0
7.	Dalibor Kramář	G BO-Řeč	3	4	8	10	10	6	12	12	58,0	58,0
8.–9.	Vojtěch Káně	G Brandýs	2	9	8	10	10		12	13	53,0	53,0
	Klára Tauchmanová	GOhradníPH	4	1	8	10	10	1	10	14	53,0	53,0
10.	Michal Bravanský	G Bílovec	0	1	8	10	10	12	9	3	52,0	52,0
11.	Filip Kastl	GKepleraPH	2	2	8	10	10	12	9		49,0	49,0
12.–13.	Michal Starý	GNoMěsNMor	4	1	8	5	10	1	10	12	46,0	46,0
	Jan Vodstrčil	G VMýto	1	4	8	10	10		11	7	46,0	46,0
14.–16.	Michal Kodad	SPŠ Smíchov	2	4	8	10	10	12			40,0	40,0
	Jiří Kvapil	GTomkovaOL	0	5	8	10	10	12			40,0	40,0
	Terézia Strišovská	GJHroncaBA	2	5	8	10	10	12			40,0	40,0
17.–18.	Jaroslav Paška	ŠPMNDaGB	3	1	8	10	10	11			39,0	39,0
	Vojtěch Poupa	CírkG Plzeň	0	2	8	5	10		9	7	39,0	39,0
19.	Ondřej Hráček	GOlgHavl	1	2	8	10	10	1		9	38,0	38,0
20.	Albert Kučera	GNadŠtolPH	2	1	8	10	10	0	8		36,0	36,0
21.	Martin Zmitko	G FrýdlNOs	2	5	8	8	10	8			34,0	34,0
22.	Erik Berta	GAlejKošice	3	5	8	5	10	0	6	3	32,0	32,0
23.–24.	Martin Bencko	GOhradníPH	1	9	8	3	10	9		1	31,0	31,0
	Janek Hlavatý	GJirsíkaČB	−1	13	8	10	3		10		31,0	31,0
25.–27.	Lukáš Gáborik	GTajBanBys	1	1	8	10	10	1			29,0	29,0
	Dennis Pražák	GJirsíkaČB	3	6	8	10	10	1			29,0	29,0
	Jiří Tlamicha	GŘíč	1	1	8	10	10	1			29,0	29,0
28.–36.	Martin Cmiel	GOlgHavl	1	1	8	10	10				28,0	28,0
	Dominik Dinh	GNVPláníPH	3	4	8	10	10				28,0	28,0
	Jindřich Dítě	VOSPŠŽďár	2	6	8	10	10				28,0	28,0
	Kristina Galikova	ŠPMNDaGB	3	1	8	10	10				28,0	28,0
	Vojtěch Michal	GNVPláníPH	3	1	8	10	10				28,0	28,0
	Jiří Vlček	GFXŠaldyLI	2	1	8	10	10				28,0	28,0
	Matěj Volf	GCoubTábor	0	1	8	10	10				28,0	28,0
	Lucie Vomelová	GŠpitálsPH	2	4	8	5			9	6	28,0	28,0
	Jan Černý	BiGy Žďár	2	1	8	10	10				28,0	28,0
37.–39.	Vladimír Chudý	ZŠRonov	1	5	8	7	10	0			25,0	25,0
	Anna Hollmannová	GSRandyJN	1	8	5				10	10	25,0	25,0
	Michael Kozel	GZborovPH	4	8	8	7	10				25,0	25,0
40.–45.	Radim Buráň	G UherBrod	3	4	8	5	10				23,0	23,0
	Ondřej Buček	GJarošeBO	4	2	8	5	10				23,0	23,0
	Tomáš Husák	GLitoměřPH	4	2	8	5	10				23,0	23,0
	Robert Jaworski	GÚstavníPH	0	8	8	5	10				23,0	23,0
	Matyáš Lorenc	GJungmanLT	4	1	8	4	10	1			23,0	23,0
	Ondřej Wrzecionko	GTěš	3	5	8	5	10				23,0	23,0
46.	Filip Krul	SPŠ Smíchov	2	1	8	2	10				20,0	20,0
47.	Tomáš Sláma	GTurnov	3	1	8	1	10				19,0	19,0
48.–57.	Jiří Bleha	SPSE Pard	1	1	8		10				18,0	18,0
	Matej Hockicko	TAPoprad	4	2	8	0	10				18,0	18,0
	Patrik Janko	SPŠ Smíchov	2	1	8		10				18,0	18,0
	František Kmječ	G Brandýs	2	3	8		10				18,0	18,0
	Radim Kopunec	G UherBrod	−2	1	8		10				18,0	18,0
	Jan Kučera	SŠKlatovskáPL	1	5	8	0	10	0			18,0	18,0
	Michal Mlčoch	G UherBrod	3	6	8	5	5	0			18,0	18,0
	Jakub Nevařil	G UherBrod	0	1	8		10				18,0	18,0
	Jakub Ucháč	ŠMaVVzt	2	4	8	10					18,0	18,0
	Dávid Šutor	GTerVans	3	3	8	10					18,0	18,0

Všechny naše algoritmy měly sice asymptotickou časovou složitost $\mathcal{O}(TN)$, ale první algoritmus byl 26-krát pomalejší, což se u opendatové úlohy projeví.

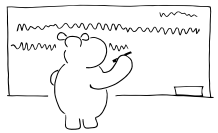
Paměťová složitost prvního přístupu s velkou tabulkou byla zřejmě $\mathcal{O}(TN)$. Pro řešení s polem konstantní velikosti je ale bohužel také $\mathcal{O}(TN)$, jelikož si musíme do paměti uložit text načtený ze vstupu, abychom ho mohli procházet po sloupcích. Pokud se ale podíváme na konstanty, tak jsme si polepšili. Nyní totiž máme pole, do kterého ukládáme maximálně 26 různých hodnot. Naopak pro velkou tabulku četností musíme být schopni uložit N různých hodnot.

Ještě se bude hodit jeden implementační detail. Pro naše řešení je potřeba nějak přepočítat znaky A až Z na čísla 0 až 25 pro indexování v poli. Naštěstí v ASCII kódování jsou všechna tato písmena za sebou a v běžných programovacích jazycích lze typicky znak převést na číslo v ASCII. Pak můžeme prostě odečíst 65, což je hodnota `A` v ASCII. Pro převod z indexu na znak naopak 65 přičteme. Například v Pythonu k tomuto slouží funkce `ord()` a `chr()`.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-Z1-3.c`

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z1-3.py`

Jirka Sejkora



30-Z1-4 Zuzčin projekt

Úloha po nás chce pro každý dotaz sestavit organizační hierarchii a pak se podívat na jednoho pracovníka a jeho pomocníky. Můžeme postupovat poměrně přímočaře a výběr spolupracovníků odsimulovat. Nejdříve najdeme hlavního ředitele, pak všechny jeho přímé podřízené, pak všechny jejich podřízené atd. Takto budeme postupně budovat jednotlivá „patra“ hiearchie, dokud nedojdeme k hledanému člověku, nebo neprojdeme všechny.

Jen je potřeba dát pozor na to, abychom nevolili jako podřízené lidi, které už jsme zvolili dříve – je potřeba si pro každého člověka pamatovat, jestli už je zařazen, například v seznamu indexovaným číslem člověka. Pak je třeba dát pozor na to, že každý dává přednost výše postaveným nadřízeným a pak těm s nižším číslem. Nikdo výše postavený ho při našem budování pater už oslovit nemůže, protože to by se stalo už při procházení předchozího patra; nižší číslo však někdo další mít může. Stačí si ale před průchodem patro setřídit podle čísla člověka a oslovovat postupně od člověka s nejmenším číslem, aby každého oslovil první ten, kdo má „největší přednost“. Třídění pravděpodobně najdete ve standardní knihovně svého oblíbeného jazyka a běžně používaným algoritmům to trvá $O(N \log N)$, kde N je velikost pole. Pokud nevíte, jak to funguje, můžete se podívat do kuchařky o třídění[4]

Jak dlouho to bude trvat, když provedeme až $\mathcal{O}(N)$ setřídění až $\mathcal{O}(N)$ velkého pole? Můžeme ale nahlédnout, že každý člověk bude jen v jednom patře, protože každého vybereme jen jednou. Takže se nám sice může stát, že v jednom patře bude třeba polovina všech lidí, ale určitě se nám nestane, že ve všech patrech dohromady bude víc lidí než jich je celkem. Pak je docela zřejmé, že třídění nám zabere dohromady maximálně $\mathcal{O}(N \log N)$ času, protože dohromady nebudeme třídit víc než N prvků. Pak ještě potřebujeme u každého nadřízeného projít všechny kamarády a zkusit je přiřadit – což je vlastně projití všech „kamarádských vazeb“. Celková časová složitost tak bude $\mathcal{O}(N \log N + M)$ pro každý dotaz, kde N jsme si označili počet lidí a M počet vazeb mezi nimi.

Toto řešení stačilo na plný počet bodů. Poznamenejme, že to jde i v $\mathcal{O}(N+M)$; toto řešení zde však detailně popisovat nebudeme. Zájemci mohou nahlédnout do zdrojového kódu.

Standa Lukeš & Říša Hladík

Program (Python 3, $\mathcal{O}(N \log N + M)$):
`http://ksp.mff.cuni.cz/viz/30-Z1-4.py`

Program (Python 3, $\mathcal{O}(N + M)$):
`http://ksp.mff.cuni.cz/viz/30-Z1-4-bfs.py`

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

30-Z1-5 Třicet totemů

Naším úkolem bylo nalézt správné počty klád K a P tak, aby totemy, které Kevin a Petr postaví, byly stejně vysoké. Informaticky řečeno máme na vstupu dvě posloupnosti délek klád a hledáme takové jejich počáteční úseky, které mají stejný součet.

Pro potřeby určení složitosti algoritmů si řekněme, že počty klád na hromadách jsou M a N, tedy $K < M$ a $P < N$.

Nejprve si rozmyslíme, jak je vlastně rychlé to nejjednodušší řešení: vyzkoušení všech kombinací. Pro každé K si projdeme všechna možná P a zjistíme, jestli výsledné totemy nejsou stejně vysoké.

Pokud bychom počítali výšku totemů pro každé P a K znovu, zabralo by nám toto zjištění $\mathcal{O}(M+N)$ času. Každý ale vidí, že pokud je budeme brát postupně, můžeme si spoustu práce ušetřit. Pokud známe výšku totemu pro nějaké K, výšku totemu pro $K+1$ získáme přičtením délky $(K+1)$-ní klády. Pro P to platí analogicky.

S tímto pozorováním už sestavíme následující algoritmus, který vyzkouší všechny kombinace počtů použitých klád:

```
Delky_K = [...]
Delky_P = [...]

Totem_K = 0
for K in 1 .. M + 1:
   Totem_K = Totem_K + Delky_K[K - 1]
   Totem_P = 0
   for P in 1 .. N + 1:
      Totem_P = Totem_P + Delky_P[P - 1]
      if Totem_K == Totem_P:
         print(K, P)
```

Pro zachování programátorských konvencí předpokládáme, že první prvek pole má index 0, a for cyklus `0 .. N` projde možnosti od 0 do $N-1$ včetně.

Poměrně snadno vidíme, že vnitřní cyklus udělá nejvýše tři kroky, a spustí se (MN)-krát. Proto je časová složitost algoritmu $\mathcal{O}(MN)$.

S tím se ale nespokojíme! Je přeci jasné, že vyzkoušet všechny možnosti nemůže být to nejrychlejší řešení. To rychlejší

může být vidět už z představy, jak by úlohu řešili Petr s Kevinem s opravdovými kládami: pochybuji, že by zkoušeli zvyšovat vyšší z totemů s nadějí, že se tím výška vyrovná.

Z této myšlenky vymyslíme algoritmus rychlejší. Oba aktéři začnou s prázdnými totemy, a v každém kroku jednu kládu přidá ten, jehož totem je menší:

```
Totem_K = Totem_P = 0
K = P = 0

while True:
    if Totem_K < Totem_P and K < M:
        K = K + 1
        Totem_K = Totem_K + Delky_K[K - 1]
    elif P < N:
        P = P + 1
        Totem_P = Totem_P + Delky_P[P - 1]
    else:
        break

    if Totem_K == Totem_P:
        print(K, P)
```

Všimněte si, že v případě stejné výšky obou totemů (i nulové) kládu přidává Petr.

Je potřeba si důkladně rozmyslet, že tento algoritmus vydá správný výsledek. Vcelku zřejmě nevypíše žádnou možnost, která není správná, ale není zřejmé, že nějakou správnou možnost nepřeskočí.

Na chvíli podezřejme algoritmus z toho, že přeskočil správnou možnost K_x, P_x s výškou totemů T. Všimněte si, že druhá větev podmínky neobsahuje ověření, že Petrův totem je nižší – algoritmus skončí, až když $P = N - 1$. Víme tedy jistě, že algoritmus v jednu chvíli měl $P = P_x$. Jak vypadalo K ve chvíli, kdy se tak stalo poprvé?

Jednou možností je, že Kevinův totem byl sice ostře menší, ale už mu žádné další klády nezbývají. To se ale stát určitě nemohlo, vždyť by pak Kevin nikdy nemohl postavit totem výšky T.

Druhou možností je, že Kevinův totem byl vyšší než Petrův, a proto algoritmus zvýšil P. Protože předpokládáme, že algoritmus možnost K_x, P_x přeskočil, musel v tu chvíli už mít $K > K_x$, jinak by možnost jistě našel.

To ovšem znamená, že už dříve nastala situace, ve které měl algoritmus $K = K_x$, a $P < P_x$. Snadno ale vidíme, že v takové situaci by algoritmus jen zvyšoval P, dokud by ztracenou možnost nenašel. Algoritmus tedy žádnou správnou možnost přeskočit nemohl.

Jak je teď algoritmus rychlý? V každé iteraci se zvýší jedna z proměnných K nebo P. Ani jedna z nich však nemůže být vyšší než M, respektive N. Dostali jsme tedy algoritmus s lineární časovou složitostí $\mathcal{O}(M+N)$. To už je hodně dobré, protože stejný čas nám zabere i čtení vstupu. Zrychlovat už nejspíše není třeba.

Poslední, nad čím bychom se mohli zamyslet, je paměťová složitost. Nám stačí pouze čtyři pomocné proměnné. Vstup potřebujeme přečíst pouze jednou, a s drobnou modifikací bychom ho mohli číst průběžně a nemuseli bychom si ho pamatovat celý. Podobně výstup můžeme vypisovat průběžně. Náš algoritmus by pak mohl mít až konstantní paměťovou složitost.

Ondra Hlavatý

[5] Takové seřazení se nazývá *lexikografické*

30-Z1-6 Zuzčina první šifra

Naším úkolem je zjistit, na které z pozic je nejvíce položených písmen Zuzčiny šifry a kolik jich je. Avšak na vstupu jsme souřadnice nedostali přímo, ty musíme nejprve spočítat.

Zaveďme si ze začátku pár pojmů. Počet centimetrů na východ označíme x, na sever y, samotná poloha poté bude (x, y). Dále N bude počet všech písmen a K bude počet všech navzájem různých poloh. Pokud bychom měli deset písmen na jednom místě, pak $N = 10, K = 1$.

Každý záznam v Zuzčině seznamu je relativní posun aktuální pozice vůči té předchozí. Dále víme, že Zuzka začala na pozici $(0, 0)$ před prvním položením písmena. Tudíž například pro záznamy $(1, 3), (-2, 5), (0, -4)$ jsou skutečné souřadnice písmen $(1, 3), (-1, 8), (-1, 4)$.

Algoritmus, jak tyto souřadnice spočítat, je tedy jednoduchý. Na začátku si nastavíme souřadnice aktuální pozice Zuzky $P = (0, 0)$. Nyní budeme postupně číst záznamy $Z = (x, y)$ a každý z nich postupně přičteme k P. Tak získáme jak polohu jednoho z písmen, tak novou aktuální polohu P' pro následující záznam.

Nuže, nyní umíme spočítat skutečné souřadnice, pojďme najít tu s nejvyšším počtem výskytů.

Přímočaré řešení

Uložme si všech N souřadnic písmen do pole. Nyní se jako velmi jednoduché řešení nabízí pole pro každý prvek projít a spočítat, kolik souřadnic se tomuto prvku rovná, a vrátit tu s nejvyšší četností. Toto je potřeba provést, protože dvě stejné souřadnice můžou být na úplně různých místech v poli.

Tento způsob prohledávání je ale velmi pomalý. Pro každou souřadnici v poli procházíme celé pole znovu, provedeme tedy celkem $\mathcal{O}(N^2)$ operací.

Pojďme tento postup vylepšit. Pole souřadnic tentokrát seřadíme, a to tak, že kdykoliv rozhodujeme o vzájemném pořadí dvou prvků, nejprve porovnáme x a v případě rovnosti ještě porovnáme y.[5]

Toto seřazení nám způsobí, že se stejné souřadnice dostanou k sobě, tvoříce souvislý úsek. Můžeme proto projít pole zleva doprava a vždy počítat počet totožných sousedních souřadnic. Bude nás zajímat souřadnice se zatím nejvyšším počtem výskytů; tu si budeme spolu s tímto počtem udržovat a na konci obojí vrátíme.

Jak dlouho nám celý tento algoritmus potrvá? Samotné zjištění souřadnic písmen a jejich vložení do pole využije $\mathcal{O}(N)$ času. Následovné seřazení pole nás bude stát $\mathcal{O}(N \log N)$ času a konečné procházení polem na zjištění nejčastějšího výskytu znovu potrvá $\mathcal{O}(N)$ času. Celkový strávený čas nám tedy vyjde $\mathcal{O}(N \log N)$. Paměti spotřebujeme $\mathcal{O}(N)$ kvůli uloženému poli.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z1-6-sort.py`

Takto efektivní řešení stačí pro plný počet bodů. Ukážeme si však, že tuto úlohu jde vyřešit i rychleji.

Využití jiných datových struktur

Náš problém nyní vyřešíme bez použití (setříděného) pole. Místo toho použijeme datovou strukturu, která nám na danou souřadnici odpoví, kolik je na ní písmen.

Zkusme nejprve použít tabulku, kde řádky odpovídají všem možným souřadnicím na východ, sloupce všem na sever a odpovídající buňka má uložený počet písmen s touto souřadnicí.

To bohužel vůbec nefunguje. Zahrada je nekonečně rozlehlá, souřadnice jednoho písmene může nabývat nesmírně vysokých hodnot. Taková tabulka se zcela jistě nevejde do paměti.

Tento nápad přesto nevede do slepé uličky. Místo tabulky uvažujme slovník.[6] Takový slovník si pamatuje dvojice (klíč, hodnota), přičemž lze do něj klíče přidávat či odstraňovat v průběhu algoritmu. Dále slovník umí zjistit, zda klíč existuje a jakou má hodnotu, nebo tuto hodnotu změnit. Rovněž umí slovník projít všechny klíče v něm uložené.

V našem slovníku budou klíčem souřadnice písmen a hodnotou počet písmen s touto souřadnicí. Souřadnice, na kterých žádné písmeno není, nebudeme do slovníku vkládat. Celkový počet klíčů v tomto slovníku bude potom K.

V průběhu algoritmu nyní postupujme takto: Při počítání souřadnic se pro každou z nich vždy podíváme do slovníku, jestli se v něm jako klíč již vyskytuje. Pokud ne, přidáme ji a nastavíme počet výskytů na 1. Pokud ano, přičteme za ní výskyt. Nakonec projdeme všechny klíče a vrátíme ten s nejvyšší hodnotou.

Jak jsme na tom s časem a pamětí tentokrát? Nevyhneme se počítání souřadnic, to nám vždy zabere $\mathcal{O}(N)$ času. Existují různé vnitřní podoby slovníků. My budeme pro jednoduchost uvažovat vyhledávací stromy, o kterých se lze více dozvědět v naší kuchařce.[7]

V takovém případě nám každé vyhledání i vložení souřadnice do vyhledávacího stromu potrvá $\mathcal{O}(\log K)$ a celkem jich proběhne N. Přičtení počtu výskytu v již nalezeném klíči potrvá $\mathcal{O}(1)$. Nakonec projití všech klíčů nás bude stát $\mathcal{O}(K)$. Celková časová složitost nám v tomto případě vyjde jako $\mathcal{O}(N + K + N \log K)$ a paměťová složitost bude $\mathcal{O}(K)$. Pro K výrazně nižší než N jsme si tedy pomohli.

Na závěr podotkněme, že se na slovníky častěji používají hešovací tabulky,[8] které umí vkládat a vyhledávat až v čase $\mathcal{O}(1)$ v průměru. Ty jsou ale podstatně složitější než vyhledávací stromy.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-Z1-6-dict.py`

Václav Končický

[6] V běžných programovacích jazycích se se slovníkem můžete setkat pod názvem *dictionary*, případně *map*.

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

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