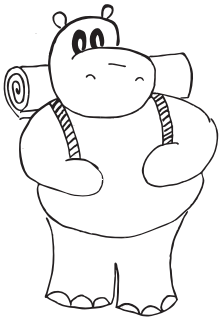
Milí řešitelé a řešitelky!



Prázdniny jsou tu a s nimi i konec 24. ročníku KSP. Než se rozjedete do všech možných i nemožných koutů světa, přinášíme vám vzorová řešení poslední série.

Doufáme, že se vám uplynulý ročník líbil a do budoucna na nás nezanevřete. Abychom se mohli nadále zlepšovat, prosíme o vyplnění ankety.[1] Již odmaturovavší řešitele zveme ke spolupráci při organizaci ročníků příštích.

Přejeme příjemně strávené prázdniny!

Organizátoři KSP

Vzorová řešení páté série dvacátého čtvrtého ročníku KSP

24-5-1 Holubí centrála

Většina z vás volila jednoduchý algoritmus, který spočíval v procházení grafu do hloubky nebo do šířky od každého z vrcholů grafu. Takové řešení je sice správné, ale jeho kvadratická složitost je příliš veliká. Ukážeme si řešení s lineární složitostí, a to hned dvě. Jedno přímočaré a druhé o kapku složitější.

Řešení prohledáváním do hloubky

Základem je klasické prohledávání do hloubky (DFS). Z libovolného vrcholu (označme ho v) pustíme DFS. Jestliže tímto průchodem objevíme všechny vrcholy, máme jednoho z hledaných členů. Znovu spustíme DFS od v, tentokrát ale po opačně orientovaných hranách. Dostaneme tak všechna další řešení. Proč všechna?

Předpokládejme pro spor, že jsme takto nenalezli nějakého z členů, který patří k řešení. I pro něj musí platit, že se dokáže dostat ke všem ostatním. Speciálně i k členovi v, ze kterého se poprvé DFS spouštělo. V tom případě ale musel být nalezen průchodem do hloubky po opačně orientovaných hranách z v, čímž dostáváme spor.

Zbývá vyřešit situaci, kdy první DFS nenajde člena vyhlašovatele. Pak zjevně ani žádný další člen objevený tímto DFS nemůže být řešením. Označme si každého z nich jako již navštíveného a spusťme DFS na nějakém dosud nenavštíveném. Takto postupujeme až do chvíle, kdy označíme všechny vrcholy.

Jediným kandidátem je nyní člen, ze kterého jsme DFS spustili naposled. Dalším průchodem tedy zjistíme, zda patří k řešení, a v případě, že ano, najdeme zase průchodem po opačně orientovaných hranách celou množinu řešení.

Řešení pomocí komponent silné souvislosti

Základem bude hledání komponent silné souvislosti grafu. Komponenta silné souvislosti je maximální podgraf orientovaného grafu G takový, že pro každé dva různé vrcholy u a v z tohoto podgrafu existuje cesta jak z u do v, tak z v do u.

Dejme tomu, že jsme získali rozklad grafu na KSS. Následuje několik jednoduchých pozorování, která nám už dají řešení úlohy.

Kondenzace grafu je graf, kde vrcholy odpovídají jednotlivým KSS a orientované hrany mezi nimi vedou právě tehdy, pokud mezi nějakým vrcholem jedné komponenty a nějakým vrcholem druhé vede hrana. Taková kondenzace je nutně acyklickým grafem (zkuste si rozmyslet, co by znamenalo, kdyby v kondenzaci cyklus byl).

Dále pokud existuje nějaký hledaný vrchol v komponentě K, od kterého se lze dostat ke všem ostatním, pak i všechny další vrcholy v K jsou řešením.

Nakonec si uvědomme, že taková komponenta K může být právě jedna a musí mít vstupní stupeň roven 0. Kdyby tomu tak nebylo, nedalo by se dostat do komponent, ze kterých do K vede hrana. To vyplývá z acykličnosti kondenzace. Pokud by takových komponent bylo více, nedalo by se kvůli nulovému vstupnímu stupni dostat z jedné do druhé.

Aby komponenta K byla řešením, musí z ní vést cesta do všech ostatních komponent. To zjistíme triviálně zavoláním DFS na původní graf od libovolného z vrcholů K. Pokud se počet objevených vrcholů rovná počtu vrcholů grafu, je K řešením úlohy.

Zbývá vyřešit, jak rozklad grafu najít. Na to lze použít například Tarjanův algoritmus. Správnost a průběh Tarjanova algoritmu na tomto omezeném místě nemá smysl rozvádět. Počkejte si třeba na jednu z dalších kuchařek. Pro nás je v tuto chvíli důležité, že nám v čase lineárním v počtu hran a vrcholů najde kýžený rozklad.

Časová i paměťová složitost algoritmu je v obou případech lineární ku počtu hran a vrcholů.

Program (C++) – DFS:
`http://ksp.mff.cuni.cz/viz/24-5-1-dfs.cpp`

Program (C++) – komponenty:
`http://ksp.mff.cuni.cz/viz/24-5-1-komponenty.cpp`

Jan Bok

24-5-2 Labutí broadcasting

Na vstupu máme *zprávu* α v podobě řetězce K bitů. Chceme jí přiřadit nějaký *kód,* což bude opět řetězec bitů (jeho délku označíme N) tak, abychom po přijetí libovolné rotace kódu uměli zjistit původní zprávu. Také si to můžeme představovat tak, že kódy jsou *cyklické* posloupnosti a nevíme, kde mají začátek.

Nejjednodušší řešení

Vytvoříme kód tvaru $\mathbf{01}^{K+1}\mathbf{0}\alpha$ – jinými slovy předřadíme zprávě nulový bit, pak $K+1$ jedničkových bitů a opět jeden nulový. Pokud kód čteme cyklicky, narazíme na jednu jedinou $(K+1)$-tici jedniček a ta nám řekne, odkud číst zprávu. Kód měří $N = 2K+2$ bitů (první nulový bit by dokonce šel vynechat, ale tím si moc nepomůžeme). Kódování i dekódování jistě zvládneme v lineárním čase.

Blokový kód

Úspornější způsob kódování spočívá v rozdělení zprávy na bloky velikosti b (konkrétní hodnotu zvolíme vzápětí). Za

[1] `http://ksp.mff.cuni.cz/about/anketa.cgi`

každý blok připíšeme nulu, čímž zařídíme, že se ve zprávě nevyskytuje víc než b jedniček za sebou. Stačí tedy na začátek přidat synchronizační značku $\mathbf{1}^{b+1}\mathbf{0}$ a jsme hotovi.

Kolik jsme potřebovali bitů? Značka je dlouhá $b+2$, za ní následuje $\lceil K/b \rceil$ bloků délky $b+1$. Celkově tedy $N = b+2+\lceil K/b \rceil \cdot (b+1) \leq b+2+((K/b)+1)(b+1) \leq K+K/b+2b+3$.

Teď využijeme, že jsme mohli b zvolit libovolně, a vybereme si takovou hodnotu, aby N vyšlo co nejmenší. Jelikož s rostoucím b výraz $2b$ roste, zatímco K/b klesá, jejich součet bude (asymptoticky) nejmenší, když se vyrovnají. To odpovídá volbě $b = \lceil \sqrt{K} \rceil$, což vede na $N \leq K + K/\lceil \sqrt{K} \rceil + 2\lceil \sqrt{K} \rceil + 3 \leq K + K/\sqrt{K} + 2\sqrt{K} + 5 = K + 3\sqrt{K} + 5$. Celkem tedy přidáváme $\mathcal{O}(\sqrt{K})$ bitů; kódování i dekódování opět stihneme v lineárním čase.

Abstraktní pohled

Je naše blokové řešení optimální, nebo si lze vystačit s řádově menším počtem bitů? Abychom na tuto otázku odpověděli, zkusme se na úlohu podívat trochu abstraktněji.

Existuje 2^K možných zpráv a každé z nich chceme přiřadit jeden z 2^N možných kódů. Pokud se nějaké dva kódy liší pouze rotací, můžeme použít nejvýše jeden z nich (takovým kódům budeme říkat *ekvivalentní*). Rozdělíme tedy množinu kódů na skupiny tak, že uvnitř každé skupiny budou všechny kódy ekvivalentní a kódy z různých skupin nikdy ekvivalentní nebudou. Každé zprávě pak přiřadíme jednu ze skupin.

(Maličko podvádíme: předpokládáme, že všem 2^K zprávám přiřazujeme kódy téže délky. Nemohlo by pomoci, kdybychom uvažovali i kratší kódy? Asymptoticky nikoliv, protože jak za chvíli uvidíme, počty skupin rostou s N exponenciálně, takže všech kódů délky menší než N je asymptoticky stejně jako kódů délky přesně N.)

Hrubá síla

Potřebujeme zvolit co nejmenší N, pro které už bude skupin dostatečný počet. Označíme-li počet skupin $s(N)$, musí platit $s(N) \geq 2^K$. Jak ale $s(N)$ spočítat?

Pro $N = 4$ je snadné skupiny sestrojit ručně:

0000	**0001**	**0011**	**0101**	**0111**	**1111**
	0010	**0110**	**1010**	**1110**	
	0100	**1100**		**1101**	
	1000	**1001**		**1011**	

Pro trochu větší N si můžeme napsat jednoduchý program, který bude generovat všechny kódy a zařazovat je do skupin. Tak vznikla následující tabulka:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$s(N)$	2	3	4	6	8	14	20	36	60	108	188	352	632	1182

Podle této tabulky můžeme snadno zjistit, že pro $K = 8$ (což jsme zadávali jako jednodušší podúlohu) je potřeba $N = 12$ labutí. $2^8 = 256$ totiž leží mezi $s(11)$ a $s(12)$.

Pro výrazně větší K ale tento postup není použitelný, neboť vyžaduje probrat a zařadit do skupin exponenciálně mnoho kódů.

Věštíme z tabulky

Jednoduchý vzorec pro $s(N)$ našim snahám zatím uniká, tak zkusme podle hodnot v tabulce odhadnout, jak rychle $s(N)$ přibližně roste. Trocha experimentování odhalí, že $s(N) \approx 2^N/N$. Kdyby to byla pravda, plynulo by z toho, že dokážeme zakódovat až $\lg s(N) \approx N - \lg N$ zpráv (kde lg značí dvojkový logaritmus). Měli bychom si tedy vystačit s přidáním řádově $\lg K$ bitů, což je mnohem méně než našich $\sqrt{K}$.

Zatím ovšem jenom hádáme. Časem dokážeme, že náš odhad počtu skupin je řádově správný; pokud jste netrpěliví, můžete mezitím zkusit najít čísla z tabulky v Online Encyclopedia of Integer Sequences.[2] Zde nejprve předvedeme kódování, kterému řádově logaritmický počet bitů stačí.

Skoro optimální kódy

Myšlenku kódování pomocí bloků lze vylepšit. Pokud by se nám podařilo zařídit, že žádný blok nebude tvořen samými jedničkami, nepotřebujeme bloky prostrkávat nulami. Z neexistence jedničkového bloku totiž plyne, že se ve zprávě nikdy nevyskytne více než $2b-2$ po sobě jdoucích jedniček. Postačí tedy synchronizační značka s $2b-1$ jedničkami.

Jenže jak se vyhnout jedničkovým blokům? Snadno: zprávu budeme považovat za zápis čísla ve dvojkové soustavě a toto číslo převedeme do soustavy o základu 2^b-1. Každou číslici pak zapíšeme dvojkově do jednoho bloku. Převodem mezi soustavami si sice pokazíme časovou složitost, ale stále zůstane polynomiální (zkuste vymyslet, jak převod zvládnout v čase $\mathcal{O}(K^2)$).

Kolik takto vytvoříme bloků? Pokud nějaké číslo x zapisujeme v soustavě o základu z, potřebujeme nejvýše $1+\log_z x = 1+\lg x/\lg z$ číslic. Naše číslo není větší než 2^K, takže počet bloků nepřekročí $1+\lg(2^K)/\lg(2^b-1) = 1+K/\lg(2^b-1)$.

Každý blok přitom zabere b bitů a navíc přidáme synchronizační značku délky $2b$. Vytvoříme tedy kód délky

$$N \leq 3b + \frac{bK}{\lg(2^b-1)}.$$

Pěkně to vyjde, pokud K je mocnina dvojky. Tehdy nastavíme $b = \lg K$ a za chvíli ukážeme, že kód si opravdu vystačí s logaritmickým počtem přidaných bitů. Dosazením do předchozí nerovnosti získáme:

$$N \leq 3\lg K + \frac{\lg K \cdot K}{\lg(2^{\lg K}-1)}.$$

Jmenovatele zjednodušíme a všechny malé členy posbíráme do $\mathcal{O}$, čímž dostaneme:

$$N \leq \underbrace{\frac{K \lg K}{\lg(K-1)}}_{Z} + \mathcal{O}(\lg K).$$

Zlomek Z je očividně „něco málo přes K“. O kolik přesně? Počítejme:

$$Z - K = \frac{K\lg K - K\lg(K-1)}{\lg(K-1)} = \frac{K\cdot(\lg K - \lg(K-1))}{\lg(K-1)}.$$

Nyní se nám bude hodit jedna ne úplně standardní nerovnost pro logaritmy:

$$\lg n - \lg(n-1) < 2/n. \qquad (*)$$

Když ji dosadíme do předchozího výpočtu $Z-K$, dostaneme:

$$Z - K \leq \frac{K \cdot 2/K}{\lg(K-1)} = \frac{2}{\lg(K-1)} \leq 2. \qquad (\text{pro } K \geq 3)$$

Potvrdilo se tedy podezření, že Z není o mnoho větší než K, což nyní dosadíme do nerovnosti pro N a získáme kýžené:

$$N \leq K + 2 + \mathcal{O}(\lg K) = K + \mathcal{O}(\lg K),$$

takže náš kód je až na konstantu skrytou v $\mathcal{O}$-čku optimální. (Tedy aspoň pro K, které je mocninou dvojky. Zkuste

[2] `http://oeis.org/`

vymyslet, jak kód upravit, aby tento předpoklad nepotřeboval. Nápověda: každé přirozené číslo lze rozložit na součet navzájem různých mocnin dvojky.)

Nerovnost s logaritmy

Ještě si dlužíme důkaz nerovnosti $(*)$. S dovolením budeme předpokládat, že n je sudé číslo; pro liché bychom postupovali obdobně.

Označíme $\ell_i = \lg(n-i+1) - \lg(n-i)$ a uvážíme součet $S = \ell_1 + \ell_2 + \ell_3 + \ldots + \ell_{n/2}$. Všimneme si, že:

- V součtu S se sousední členy vyruší: $S = \lg n - \lg(n-1) + \lg(n-1) - \lg(n-2) + \lg(n-2) - \lg(n-3) + \ldots - \ldots - \lg(n/2) = \lg n - \lg(n/2) = 1$. (Takovým sumám se říká teleskopické podle starodávných dalekohledů, jejichž části se do sebe podobným způsobem zasouvaly.)
- Posloupnost $\ell_1, \ell_2, \ell_3, \ldots, \ell_{n/2}$ je neklesající. Vskutku: pro každé t platí $\lg t - \lg(t-1) \leq \lg(t-1) - \lg(t-2)$. Rozdíl logaritmů totiž můžeme napsat jako logaritmus podílu:
$$\lg \frac{t}{t-1} \leq \lg \frac{t-1}{t-2},$$
což odlogaritmujeme:
$$\frac{t}{t-1} \leq \frac{t-1}{t-2}.$$
Vynásobením součinem jmenovatelů (ten je pro zajímavá t nezáporný) dostaneme:
$$t \cdot (t-2) \leq (t-1) \cdot (t-1),$$
čili
$$t^2 - 2t \leq t^2 - 2t + 1,$$
a to je pravda pro všechna t.
- V každé posloupnosti reálných čísel je minimum menší nebo rovno aritmetickému průměru. Zde je minimem ℓ_1 a průměrem $S/(n/2) = 2/n$, jinak řečeno
$$\lg n - \lg(n-1) \leq 2/n,$$
což je přesně nerovnost, kterou jsme chtěli dokázat.

(Podobným trikem by šla dokázat i nerovnost v opačném směru, totiž $\lg n - \lg(n-1) \geq 1/n$. Zkuste vymyslet, jak.)

Počítáme skupiny

Abychom uspokojili hloubavou mysl, vraťme se ještě k počtu skupin $s(N)$, který jsme zatím pouze „vyvěštili“.

Uvažme nějaký kód délky N a počítejme, jak velká je skupina, do které patří. Kdyby byly všechny skupiny stejně velké, stačilo by vydělit počet kódů velikostí skupiny. Jenže už v našem příkladu pro $N = 4$ narazíme: existují skupiny velikostí 1, 2 i 4.

Zkusme přijít na to, jak pro daný kód α zjistit, jak velká je jeho skupina. Ta je tvořena všemi rotacemi řetězce α. Pokud jsou všechny rotace různé, skupina obsahuje N kódů. V opačném případě je kód α roven nějaké své rotaci. Takové řetězce musí být nutně periodické (tzn. jsou tvořeny opakováním nějakého kratšího řetězce; rozmyslete si, proč).

Toto pozorování nám pomůže spočítat $s(N)$ pro prvočíselné N. Délka periody periodického řetězce totiž musí být dělitelem jeho délky, takže pokud je N prvočíslo, jediné periodické řetězce jsou $\mathbf{0}\ldots\mathbf{0}$ a $\mathbf{1}\ldots\mathbf{1}$. Dvě ze skupin proto mají velikost 1 a ostatní velikost N. Celkem se v nich nachází 2^N kódů, tudíž musí platit:
$$s(N) = (2^N - 2)/N + 2.$$
Naše hypotéza se tedy aspoň pro prvočíselná N potvrdila. (Vrtá vám hlavou, proč je $2^N - 2$ dělitelné číslem N? To plyne z Malé Fermatovy věty a drobným rozšířením našich úvah o řetězcích bychom dokonce získali její kombinatorický důkaz.)

Pokud je N složené číslo, je situace daleko komplikovanější a neumíme ji vyřešit bez použití trochu pokročilejší kombinatoriky (ale moc pěkné, zkuste si najít tak řečené Burnsidovo lemma). Prozradíme alespoň, co vyjde:
$$s(N) = \frac{1}{N} \cdot \sum_{d \backslash N} \varphi(d) \cdot 2^{N/d}.$$
Suma běží přes všechny dělitele čísla N, $\varphi(d)$ je Eulerova funkce, která udává počet čísel od 1 do $d-1$ nesoudělných s d. Výsledné $s(N)$ opět řádově nepřekročí $2^N/N$.

Závěrem

Po troše počítání jsme našli řešení, které přidává pouze řádově logaritmický počet bitů, a to je až na konstantu optimální.

Kdyby nám na konstantách záleželo, problém by byl mnohem obtížnější. V zásadě bychom potřebovali vybrat si nějaké reprezentanty skupin. Vhodnými kandidáty jsou jejich lexikograficky nejmenší prvky – těm se říká *náhrdelníky* a v naší tabulce pro $N = 4$ leží na prvním řádku. Množinu všech náhrdelníků bychom pak taktéž uspořádali (třeba zase lexikograficky) a chtěli bychom v ní umět najít i-tý nejmenší náhrdelník, aniž bychom všechny náhrdelníky vyjmenovali. Ačkoliv podobné algoritmy jsou známé třeba pro permutace, nalezení i-tého nejmenšího náhrdelníku v polynomiálním čase je stále otevřený problém.

Pokud vás teorie náhrdelníků zaujala stejně jako mne, doporučuji začíst se do knížky Combinatorial Generation od Franka Ruskeyho (dostupná i online).

Program (C) – generátor skupin:
`http://ksp.mff.cuni.cz/viz/24-5-2-generator.c`

Program (C) – kodér:
`http://ksp.mff.cuni.cz/viz/24-5-2-koder.c`

Program (C) – dekodér:
`http://ksp.mff.cuni.cz/viz/24-5-2-dekoder.c`

Martin „Medvěd“ Mareš

24-5-3 Struktura organizace

Obecné řešení

Aby skupina mohla komunikovat, musí existovat nějaký šéf, který má (nepřímo) podřízené všechny zaměstnance. To platí i pro podskupinku zaměstnanců, kterou pošleme do akce.

Pro jednoduchost tedy zvolme šéfa skupinky (S), která jde do akce. Každý z jeho přímých podřízených (P) buď do akce jít nemusí, nebo může. V prvním případě to odpovídá jedné variantě (pokud tam nejde P, nemůže jít ani libovolný z jeho (nepřímých) podřízených, jelikož by nebyl schopen předat zprávu např. S). V druhém případě lze vzít do akce i nějaké podřízené P. Počet možností, kolika způsoby je lze vybrat, je přesně počet možností, kolika můžeme vybrat akční skupinu, kde šéf bude P.

Celkový počet možností, kolika vybrat podřízené S, spočteme uvážíme-li, že výběr je nezávislý pro každého podřízeného P, tedy
$$\text{Možnosti}(S) = \prod_{P \in \text{podřízení } S} (1 + \text{Možnosti}(P)).$$

Pokud chceme zjistit počet skupin s libovolným šéfem, stačí sečíst počty možností přes všechny vedoucí, tedy

$$\text{AkčníchSkupin} = \sum_{S \in \text{všichni}} \text{Možnosti}(S).$$

Tahle myšlenka se v programu implementuje přímočaře, časová složitost bude $\mathcal{O}(NM)$, kde N je počet zaměstnanců a M čas potřebný na jednu aritmetickou operaci. Aritmetických operací vskutku provedeme $\mathcal{O}(N)$, jelikož operace uvnitř $\sum$ nebo $\prod$ můžeme „naúčtovat“ podřízeným, kteří se jich účastní, a každému podřízenému takto naúčtujeme nejvýše konstantní počet operací.

Obrovská čísla a Karacubův algoritmus

Proč je ale ve složitosti zmiňováno M? Bohužel počet možností, kolika lze poskládat skupinku, roste rychle. Horní odhad je počet možností, jak vybrat libovolnou skupinku zaměstnanců (2^N), a není příliš nadhodnocený. Uvážíme-li např., že šéf S má K přímých podřízených a další zaměstnanci neexistují, bude počet možných skupin vyslaných do akce $2^K + K$, což se od triviálního odhadu (2^{K+1}) příliš neliší.

Potřebujeme tedy počítat s obrovskými čísly, která mají $C = \mathcal{O}(N)$ cifer. Budeme je v paměti reprezentovat jako pole číslic. Sčítání pak lze stihnout v čase $\mathcal{O}(C)$ vcelku triviálně. S násobením je to horší. Ukážeme si zde, jak dvě čísla vynásobit v čase $\mathcal{O}(C^{\log_2 3})$ pomocí Karacubova algoritmu.

To, že čísla ukládáme po cifrách, je podstatné. Existují reprezentace (např. pomocí zbytků), ve kterých jde násobit i sčítat v čase $\mathcal{O}(C)$. Problém pak mají s výpisem výsledku.

Základní myšlenka Karacubova algoritmu je rozděl a panuj.[3] Nechť násobíme čísla A a B o C cifrách. Rozdělme si každé na 2 čísla A_h a A_d o $C/2$ cifrách tak, že bude platit $A = A_h 10^{C/2} + A_d$ (efektivně poloviny čísla dle cifer v desítkovém zápisu), pro B analogicky. Pak platí $A \cdot B = A_h B_h 10^C + (A_h B_d + A_d B_h) 10^{C/2} + A_d B_d$, efektivně tedy potřebujeme kromě sčítání 4 násobení (vynecháme-li násobení mocninami desítky, což je však v zápisu po cifrách jednoduchý posun).

Podívejme se na násobení pořádně. S $A_h B_h$ a $A_d B_d$ moc neprovedeme, s $A_h B_d + A_d B_h$ se však dá ještě pracovat. Konkrétně lze snadno nahlédnout, že

$$A_h B_d + A_d B_h = (A_h + A_d)(B_h + B_d) - A_h B_h - A_d B_d.$$

Hle – poslední dva součiny už známe. Spočteme tedy součiny Malý $= A_d B_d$, Střední $= (A_h + A_d)(B_h + B_d)$ a Velký $= A_h B_h$ a pak platí $A \cdot B = \text{Malý} + (\text{Střední} - \text{Malý} - \text{Velký}) \cdot 10^{C/2} + \text{Velký} \cdot 10^C$. Tím jsme zredukovali počet potřebných násobení na tři.

Čas potřebný k spočítání součinu čísel o C cifrách, $T(C)$, lze tedy vyjádřit jako $T(C) = 3T(C/2) + fC$, kde f je vhodná konstanta. Čas fC odpovídá času spotřebovanému na sčítání, posouvání apod., vše jde stihnout lineárně či lépe vzhledem k počtu cifer. Vyřešením takovéhle rekurence zjistíme, že násobení spotřebuje čas $\mathcal{O}(C^{\log_2 3}) \approx \mathcal{O}(N^{1,585})$.

S tímhle násobením (a uvážením, že $C = \mathcal{O}(N)$) tedy bude program mít časovou složitost $\mathcal{O}(N \cdot N^{\log_2 3}) \approx \mathcal{O}(N^{2,585})$ a paměťovou $\mathcal{O}(NH)$, kde H je počet hladin stromu, tj. kolik (nepřímých) šéfů má libovolný zaměstnanec maximálně nad sebou (+1).

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-5-3-karacuba.pas`

[3] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

Binární stromy

V případě, že strom zaměstnanců je úplný binární (což jsme zadávali jako podúlohu), lze algoritmus zjednodušit. Konkrétně víme, že oba podstromy synů budou pro každý vrchol stejné. Tedy i počty možností, kolik skupin můžeme stvořit, se bude shodovat.

Pro šéfa v hloubce h (velký šéf má hloubku 0, jeho přímí podřízení 1, atd.) bude tedy platit

$$\text{Možnosti(hloubka } h) = (1 + \text{Možnosti(hloubka } h+1))^2,$$

samozřejmě že pro listy (zaměstnance bez podřízených) platí Možnosti $= 1$.

Dále v hladině s hloubkou h bude 2^h zaměstnanců. Počet akčních skupin pak bude

$$\text{AkčníchSkupin} = \sum_{h=0}^{H} 2^h \cdot \text{Možnosti(hloubka } h).$$

Tohle lze stihnout spočítat v čase $\mathcal{O}(HM)$, kde H je hloubka stromu ($H = \lceil \log_2(N+1) \rceil$) a M je opět náročnost násobení. Celkově (s Karacubovým algoritmem) bude časová složitost $\mathcal{O}(N^{\log_3 2} \cdot \log_2 N)$.

Rychlejší násobení

Násobení nás evidentně brzdí. Není možné jej stihnout rychleji? Lze ukázat, že Karacubův algoritmus je součástí třídy algoritmů Tooma a Cooka, které spočítají násobení v čase $\mathcal{O}(C^{\log(2k-1)/\log(k)})$, kde k je přirozené číslo. Pro každé $\varepsilon > 0$ tedy můžeme zvolit dost velké k tak, abychom násobili v čase $\mathcal{O}(C^{1+\varepsilon})$; konstanta v $\mathcal{O}$ přitom s klesajícím ε obludně roste.

Naznačme si však, jak pracuje jeden z (asymptoticky) nejrychlejších algoritmů na násobení, Schönhageův-Strassenův algoritmus – počítá součin čísel v čase $\mathcal{O}(C \log C \log\log C)$. Vzhledem k tomu, že však spoléhá na některé složitější výsledky z algebry, nebudu zde jeho správnost dokazovat, čtenář si v případě zájmu jistě vyhledá tento algoritmus detailně sám. (Doporučuji se podívat i na Carmichaelovu funkci, která souvisí s volbou základu p okruhu $\mathbb{Z}$.)

Základní myšlenka je, že vynásobení frekvenčních koeficientů po provedení Fourierovy transformace odpovídá konvoluci v „přímém“ obrazu.

Uvažujme Fourierův obraz daného čísla A. To je nějaký vektor $\mathcal{F}[A]$, pro jehož k-tou složku platí:

$$\mathcal{F}[A]_k = \sum_{j=0}^{C-1} A_j \alpha^{jk}.$$

kde A_j je j-tá cifra A, α C-tá odmocnina z jedničky (primitivní, tj. taková, že $\alpha^k \neq 1$ pro $0 < k < C$), analogicky pro B. Definujeme-li Fourierův obraz D jako součin Fourierových obrazů A a B, tedy

$$\mathcal{F}[D]_k = \mathcal{F}[A]_k \cdot \mathcal{F}[A]_k,$$

pak bude platit

$$D_k = \sum_j A_j B_{k-j},$$

přičemž suma jde přes všechny hodnoty, kde sčítanci mají smysl. Vzhledem k této definici však platí

$$A \cdot B = \sum_k D_k 10^k.$$

Tedy bude stačit jen znormalizovat zpět D_k na cifry (jelikož konvoluce nezaručuje, že vyjdou jednotlivé cifry, ale jen že předchozí suma je rovna součinu).

K implementaci budeme potřebovat znát ještě inverzi Fourierovy transformace, která lze zapsat jako

$$D_k = \frac{1}{C} \sum_{j=0}^{C-1} \mathcal{F}[D]_j \alpha^{-jk},$$

a způsob, jak rychle spočíst Fourierovu transformaci (inverze evidentně, až na normalizaci, je Fourierova transformace s použitím primitivní odmocniny $1/\alpha$) a dále jak najít ono číslo α, aniž bychom ztráceli přesnost.

První problém lze vyřešit použitím algoritmu pro rychlý výpočet Fourierovy transformace, v implementaci je použit Cooleyův-Tukeyův algoritmus, který pracuje v čase $\mathcal{O}(C \log C)$.

Problémům s přesností se nevyhneme, pokud budeme počítat Fourierovu transformaci klasicky, tj. v komplexních číslech. Pomůže ale přesunout se do nějakého konečného okruhu, v našem případě do celých čísel modulo 469 762 049 (kde 33 je primitivní 2^{26}-tá odmocnina z jedničky).

Po použití tohoto algoritmu bude celý program pracovat v čase $\mathcal{O}(C^2 \log C \log \log C)$.

Program (Pascal):
`http://ksp.mff.cuni.cz/viz/24-5-3-s-s.pas`

Pavel Čížek

Medvědí poznámka: Ani Schönhageův-Strassenův algoritmus není posledním slovem v oblasti rychlé aritmetiky. S použitím takřka ďábelských triků se dá násobit i v lineárním čase. O těchto algoritmech a fascinující historii jejich objevování znamenitě vypráví pan Donald Knuth ve svých Seminumerical Algorithms (2. díl jeho veledíla The Art of Computer Programming). Pokud by vás zajímalo, jak se takové věci dělají, rád se nechám přemluvit k půlnoční přednášce na soustředění. A pokud toužíte „jen" po pochopení Fourierovy transformace, zkuste nahlédnout na stránky přednášky z ADS2.[4]

Martin „Medvěd" Mareš

24-5-4 Sraz na náměstí

Napřed chvíli předpokládejme, že architekt náměstí byl při smyslech a udělal ho konvexní. Řešení pro nekonvexní náměstí není o moc složitější, ale problém se řeší lépe postupně. Taktéž předpokládejme, že žádné dva vrcholy nemají stejnou y-ovou souřadnici. Programu to nijak nevadí (pokud se vrcholy se stejnou y-ovou souřadnicí prochází např. zleva doprava), ale ve vysvětlování by rušily.

Všimneme si, že alespoň jedna z nejdelších vodorovných úseček bude končit ve vrcholu. Pokud se totiž podíváme na nějakou úsečku, která nekončí ve vrcholu, tak buď není od kraje ke kraji, nebo oběma konci končí na hranách náměstí. Tyto dvě hrany jsou buď rovnoběžné, potom úsečku můžeme posouvat jak nahoru, tak dolů, dokud nenarazíme na vrchol, aniž by se změnila délka. A nebo tyto hrany rovnoběžné nejsou, ale v tom případě se jedním směrem od sebe vzdalují, úsečku tedy můžeme posunout tímto směrem a tím ji prodloužit.

Tedy pro vyřešení problému s konvexním náměstím nám stačí zamést jej přímkou odshora dolů (což bylo ukázáno v geometrické kuchařce).[5] Budeme si udržovat, která hrana je aktivní na levé a na pravé straně. V každém vrcholu spočítáme délku úsečky od tohoto vrcholu k protější aktivní hraně. Poté vyměníme aktivní hranu na straně, kde se nacházel vrchol.

Nyní, co za problémy nám přinese nekonvexnost náměstí? Prvním je to, že z vrcholu už nemusí vést jedna úsečka nahoru a druhá dolů. Může se nám stát, že obě vedou nahoru nebo obě dolů. A z toho plyne další drobný problém. Úsečka už teď nemusí být v dané výšce jen jedna, ale může jich být několik vedle sebe, oddělených od sebe zuby.

Jak to vyřešíme? Jednotlivé úsečky, co aktuálně existují, si uložíme do vyhledávacího stromu, v pořadí odleva doprava. Jejich konce se sice stále mění, proto nemohou být uložené, ale to nevadí, můžeme uložit hrany, na kterých ty konce leží, a počítat je průběžně dle potřeby. Jejich pořadí zůstane po celý život úsečky stejné.

Když potkáme vrchol, který má jednu svou úsečku nahoru a jednu dolů, najdeme úsečku, která v té horní končí, a hranu v ní nahradíme. Pokud má vrchol obě hrany dolů, pak buď dělí existující úsečku na dvě (pokud se náměstí nachází na vnější straně úhlu), nebo vytváří novou úsečku začínající v tomto vrcholu.

Pokusíme se tedy najít úsečku, která tento bod obsahuje. Pokud existuje, úsečku rozdělíme na dvě. Pokud ne, vytvoříme novou úsečku. Vrchol, kde vedou obě hrany nahoru, funguje obdobně, jen dvě úsečky spojujeme nebo aktuální jednu úsečku uvnitř zubu mažeme.

V každém případě zjistíme délku té úsečky, na kterou jsme sáhli. V případě, že rozdělujeme nebo spojujeme, uvažujeme tu vcelku, neboť je to ta nejdelší, kterou máme k dispozici.

Nyní, k odhadům složitostí. Pokud má náměstí n vrcholů, tak má také tolik hran. Ve stromu budeme mít maximálně tolik úseček, neboť ke vzniku nové úsečky potřebujeme vždy vrchol. Takže paměťová složitost bude lineární. Na začátku potřebujeme vrcholy setřídit a děláme $\mathcal{O}(n)$ operací na stromě, kde každá operace trvá $\mathcal{O}(\log n)$. Dohromady máme tedy časovou složitost $\mathcal{O}(n \cdot \log n)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-5-4.cpp`

Michal „Vorner" Vaner & Lucka Mohelníková

24-5-5 Řezání kabelů

S úlohou o řezání kabelů se můžete setkat na Medvědových cvičeních z Programování II na Matfyzu jako s řezáním trámů. Možná budu nosit dříví do lesa, než s ním dojdu na pilu, ale rád bych úvodem řekl něco o řešeních, která nikam nevedou. Nebojte, optimální řešení také zmíním a nakonec se dozvíte i pár slov o tom, jak úloha souvisí s kompresí dat.

Zopakujme ve stručnosti zadání: Kabel o délce K máme nařezat na n kusů zadaných délek k_1 až k_n. Můžeme přitom vždy řezat jen jeden kus na dva; takový řez nám zabere tolik času, jako je délka řezaného kusu. Hledáme postup, kterým nařežeme celý kabel za nejkratší možnou dobu T^*.

Hrubá síla

Hrubá síla dá správný výsledek vždycky. Bohužel, tady je příliš hrubá i pro velmi malé vstupy.

[4] `http://mj.ucw.cz/vyuka/ads2/`
[5] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

Kusům, jejichž délky jsou na vstupu, říkejme *základní kusy*. Do podoby původního kabelu je za sebe můžeme slepit $n!$ způsoby.[6]

Na kabelu si můžeme udělat rysky, podle kterých budeme řezat a kterých bude pro každý způsob $n-1$. Počet řezů je to jediné, čím si můžeme být docela jisti – někteří z vás viděli spojitost se známou úlohou o lámání čokolády, ale rysky podle mě dávají ještě jednodušší představu; podle každé očividně musíme kabel přeříznout právě jednou.

Různých pořadí výběru rysek je $(n-1)!$. Udržujeme si v paměti dosavadní nejlepší postup řezání a v čase lineárním vzhledem k n s ním porovnáváme každý nově vygenerovaný postup. Celkem tedy hrubá síla trvá $\mathcal{O}(n! \cdot (n-1)! \cdot n) = \mathcal{O}((n!)^2)$. To je mnohem, mnohem, mnohem horší než exponenciální. Paměť $\mathcal{O}(n)$ nás pak ani nebude zajímat a honem poběžíme hledat něco, co má šanci doběhnout před koncem světa. (Takže do zimy.) ;-)

Heuristiky s půlením délek

Nemálo z vás přistupovalo k úloze s dobrou intuicí, že se vyplatí kabel nejprve rozříznout někde „uprostřed“, abyste čas na řezání velkého kusu investovali jednou a řezali pak už jen menší kusy.

Takto vágní popis algoritmu se rozrostl v celou řadu heuristik, bez výjimky chybných. Samotné rozdělení kusů do dvou v součtu stejně dlouhých skupin je problém dvou loupežníků,[7] o kterém jsme loni měli úlohu a který patří mezi těžké problémy.[8] Příklad ze zadání je přímo protipříkladem na heuristiku ze zmiňované loňské úlohy (rozdělování od nejdelších kusů). Nemohu vyloučit, že některý z algoritmů jdoucích tímto směrem bude dost blízko korektnímu řešení, ale vážně o tom pochybuji.

Hladové lepení – optimální algoritmus

Jak mělo vypadat optimální řešení? Mělo se na to jít z opačného konce. Místo abychom kabel řezali, budeme ho z už nařezaných kousků lepit. Potom jenom pustíme záznam postupu pozpátku.

Lepit k sobě budeme vždycky dva nejkratší kusy kabelu, které zrovna máme. Opakujeme, dokud nemáme celý kabel. Je to tak prosté, až se nechce věřit, že je to správně. Korektnost postupu si ale hned dokážeme.

Nejprve si všimneme, co se stane, když budeme líní. Líný programátor nevyhodnocuje aritmetické výrazy, jenom k nim připisuje další operace. Pokud si v průběhu lepení poznamenáváme délky *slepených kusů* líně, dojdeme k tomu, že každý *základní kus* (jeden z n kusů na vstupu) přispěje do celkového času tolikrát, kolikrát se účastnil lepení sám nebo jako součást už slepeného kusu.

Už z tohoto pozorování je možné uhodnout, že bude moudré lepit nejdřív dva nejmenší kusy, protože u těch nejméně vadí, že se budou lepení účastnit víckrát.

Exaktní důkaz povedeme sporem. Označme si l_i počet lepení, kterých se účastnil základní kus i, čas řezání $T_l = \sum_{i=1}^{n} k_i \cdot l_i$. Optimální čas řezání $T^* = \min_l T_l$. *Optimálním postupem* myslíme postup, který trval dobu T^*.

Dokazovat budeme tvrzení, že dva základní kusy x a y, které se v nějakém optimálním postupu účastnily nejvíce lepení a jako první se slepily spolu ($l_x = l_y = \max_{1 \le i \le n} l_i$), jsou ty dva nejkratší ($k_x = \min_i k_i$; $k_y = \min_{i \neq x} k_i$; tedy $k_x \le k_y \le k_i \forall i$). Kdyby v optimálním postupu x a y nebyly dva nejkratší kusy kabelu, musel by existovat kus $z \neq x$ o délce $k_z < k_y$. Tento kus by se díky volbě l_y účastnil $l_z \le l_y$ lepení. Prohodíme l_y a l_z, jinak postup zachováme.

$$l'_y = l_z,$$
$$l'_z = l_y,$$
$$l'_i = l_i \forall i \notin \{y, z\}$$

- Pokud $l_z = l_y$, čas řezání T_l se prohozením nezměnil a postup už vyhovuje tvrzení. Chtěli jsme, aby nějaký takový postup existoval.
- Pokud $l_z < l_y$, prohozením jsme čas T_l snížili, protože $l_z \cdot k_z + l_y \cdot k_y > l'_y \cdot k_y + l'_z \cdot k_z$ a levou dvojici sčítanců jsme v T_l při přechodu k l' vyměnili za pravou. Čas po prohození $T_{l'} < T_l$, ale předpokládali jsme $T_l = T^*$, což je spor s předpokladem. Lepšího než optimálního času řezání dosáhnout nemůžeme, takže původní postup nebyl optimální.

Tvrzení dokázáno a s ním i korektnost algoritmu. Zapomeneme, že jsme x a y slepili, a po nalezení zbytku optimálního postupu si na to zase vzpomeneme; v tom tkví celé kouzlo.

Složitost optimálního řešení

Jakou má náš algoritmus složitost? To záleží, jak chytře budeme průběžně hledat nejkratší kusy. Dobré je vložit všechny délky kusů do haldy a vždycky dva nejkratší slepit a výsledný kus zase vrátit, dokud nebudeme mít celý kabel. Při jednom lepení potřebujeme dva výběry minima z haldy a jedno vložení do haldy. Tyto operace s haldou trvají $\mathcal{O}(\log n)$, protože v haldě máme $\le n$ kusů. Jak už jsme zjistili dřív, lepení je $n-1$, celkem tedy potřebuje náš algoritmus čas $\mathcal{O}(n \log n)$. Paměti potřebuje $\mathcal{O}(n)$ kvůli haldě.

Pokud bychom chtěli vypisovat na výstup řezy a ne lepení, mohli bychom si lepení ukládat na zásobník a na konci programu je vypsat. Časové ani paměťové nároky algoritmu to nezhorší.

Při implementaci haldy si musíme dát pozor, aby zvládala pracovat s duplicitními klíči. Pokud v haldě máme dvě stejná čísla, je zcela očividně jedno, které z nich vybereme. Kdybychom v jiné úloze měli v haldě složitější objekty, už na volbě záležet může.

Drobné vylepšení

Ještě si můžeme rozmyslet, že haldu programovat nemusíme. Stačí si všimnout, že každý další slepený kus je nejméně tak velký, jako je ten předchozí. Když je budeme dávat do fronty, budeme je z ní vybírat v setříděném pořadí. Algoritmus tedy můžete najít ve vzorové implementaci zhruba takto:

1. Načti počet kusů kabelu `n`, pokud `n` < 2, skonči.
2. Načti délky kusů `k`.
3. Setřiď `k` vzestupně.
4. Vytvoř frontu délek slepených kusů `s`, výstupní zásobník `o`.
5. Vyber do `a`, `b` první dva prvky `k`.
6. Do `o` ulož (`a`, `b`), do `s` přidej `a + b`.
7. Proveď $(n-2)$-krát...
   - Vyber minimum `a` z čel front `k` a `s` a z původního umístění ho odeber.

[6] http://cs.wikipedia.org/wiki/Permutace

[7] http://ksp.mff.cuni.cz/viz/23-4-3/reseni

[8] http://ksp.mff.cuni.cz/viz/kucharky/tezke-problemy

- Vyber minimum `b` z čel front `k` a `s` a z původního umístění ho odeber.
- Do `o` ulož (`a`, `b`), do `s` přidej `a + b`.

8. Každou dvojici z `o` vypiš ve formátu `"(a+b) -> a + b"`.

Můžeme si všimnout, že nejpomalejší na celém postupu je třídění; pokud použijeme některý z rychlých algoritmů,[9] zůstaneme na časové složitosti $\mathcal{O}(n \log n)$. Pokud ale dostaneme vstup už setříděný, můžeme si polepšit – zbytek algoritmu totiž běží v čase $\mathcal{O}(n)$, jelikož během každého lepení děláme jen konstantní počet operací konstantní složitosti.

Dynamika? Pomalá...

Některé z vás mohlo napadnout, že by mohlo existovat řešení založené na dynamickém programování; jedno takové jsem dokonce dostal. Původně jsem nevěřil tomu, že funguje, ale opravdu je to tak. Ovšem je pomalé a těžkopádné proti tomu, které jsem už ukázal. V podstatě se zakládá na přístupu hrubou silou, ale navíc potřebuje důkaz celkem netriviálního tvrzení.

To tvrzení říká, že v řešení hrubou silou není potřeba zkoušet všechna různá pořadí nařezaných kusů, protože když najdeme nejkratší postup řezání pro setříděnou permutaci, jde upravit na nejkratší postup pro každou jinou. Setříděnou permutací myslíme takovou, že délky kusů na kabelu zleva doprava rostou. Díky tomuto omezení rozsahu úlohy srazíme složitost hrubé síly na $\mathcal{O}(n!)$, což je jenom mnohem pomalejší než exponenciální – další dvě „mnohem“ už si mohu odpustit. Když navíc přidáme dynamické programování, získáme už polynomiální algoritmus.

Ten zkouší, podle které rysky v pevně daném, setříděném pořadí základních kusů je nejvýhodnější začít řezat. Pro každou z $\mathcal{O}(n^2)$ posloupností po sobě jdoucích základních kusů postupně spočítá minimální čas, za který jde nařezat. Postupuje přitom od těch, které obsahují nejméně kusů, po ty, které jich obsahují nejvíce. Jednokusové posloupnosti jdou nařezat triviálně za nulový čas. Pro delší budeme počítat časy řezání začínajících vybranou ryskou, z nich minimum přes všechny rysky uvnitř této posloupnosti.

Řez podle vybrané rysky rozdělí posloupnost základních kusů na dvě kratší, pro které výsledek už známe. Spočítáme tedy minimální čas řezání posloupnosti, které začíná tímto řezem. Časy řezání kratších posloupností sečteme a přičteme k nim čas potřebný pro jejich oddělení, tedy celkovou délku právě řezané posloupnosti. (Reálnou délku, už ne v počtu kusů!)

Časy si můžeme v průběhu výpočtu uchovávat třeba v tabulce (matici, vícerozměrném poli), kterou budeme indexovat délkou posloupnosti (opět v počtu kusů) a pořadovým číslem rysky, na které začíná. Posloupnost délky n kusů je jen jedna, celý kabel. V jemu odpovídající buňce najdeme na konci výpočtu minimální celkový čas řezání.

Teď ještě najít i konkrétní postup. Je to klasická dynamika... Pro každou posloupnost si budeme pamatovat (v extra tabulce), který řez je pro ni nejvýhodnější provést jako první. To už v průběhu výpočtu zjišťujeme, tak si to teď budeme i pamatovat. Z takové informace už je na konci výpočtu postup řezání celého kabelu triviální sestavit.

Algoritmus získaný metodou dynamického programování potřebuje $\mathcal{O}(n^2)$ paměti kvůli tabulce pro rekonstrukci postupu a $\mathcal{O}(n^3)$ času, protože při výpočtu hodnoty každé z $\mathcal{O}(n^2)$ buněk tabulky spotřebuje čas $\mathcal{O}(n)$ na hledání nejlepší rysky.

Čas třídění délek kusů je asymptoticky menší než $\mathcal{O}(n^3)$, takže složitost nezvýší, buněk tabulky je $\mathcal{O}(n^2)$ proto, že tolik je různých dvojic začátek-konec posloupnosti.

Připomínám, že jsme nedokázali onen netriviální předpoklad, že stačí úlohu vyřešit pro setříděné pořadí základních kusů. Důkaz nechávám jako cvičení pro pokročilé. Kdybyste ho nemohli vymyslet a moc vás zajímal, zeptejte se mě mailem nebo raději na fóru.

Bodování

Za přehledně popsaný optimální algoritmus včetně výpočtu časové a paměťové složitosti, se zdůvodněním korektnosti (důkaz jsem nechtěl, ten by byl za bonusový bod) bylo možné získat plný počet 9 bodů.

Za špatně popsaný optimální algoritmus, u kterého jste se složitostí ani korektností vůbec nezabývali, jste dostávali 6 bodů. Stejný počet bodů jsem měl v plánu dávat i dobře popsaným algoritmům se zdůvodněním korektnosti a výpočtem složitosti, které nejsou optimální, ale pořád běží v polynomiálním čase. Přišel mi ale jen jeden 4bodový s divokým popisem a bez zdůvodnění korektnosti. Ten mi zabral na opravení nejvíc času. Pamatujte, že počet přidělených bodů je nepřímo úměrný času, který opravující org nad řešením stráví. ;-)

Méně než čtyři body dostávala řešení, která nefungovala nebo nebyla polynomiální. Použitelné jsou totiž obě kategorie zhruba stejně. Nulu nedostal nikdo; konkrétní počet bodů jsem uděloval podle přítomnosti užitečných pozorování o úloze, přehlednosti vyjadřování (ani nad nefunkčním řešením nechci strávit odpoledne) a za snahu.

Souvislost s kompresí dat

Na konec slibovaná perlička ohledně komprese dat. Jak spousta z vás postřehla, na postup řezání je možné se dívat jako na binární strom. Základní kusy tvoří listy, slepené kusy tvoří vnitřní vrcholy, celý kabel je kořen. Zároveň každý vnitřní vrchol má právě dva syny a představuje jedno řezání.

Zapomeňme na to, že šlo o kabely, k základním kusům připišme písmena abecedy a na délky kabelů se koukejme jako na četnosti písmen v nějakém textu. Na hrany směřující doleva napišme nuly, na hrany směřující doprava jedničky a už po cestě z kořene do listu můžeme číst kód, kterým budeme znak zapisovat.

Díky tomu, že písmena jsou jenom v listech, není žádný kód prefixem (předponou) jiného, takže text zapsaný pomocí takto zakódovaných písmen je jednoznačně dekódovatelný.

Když se znovu podíváme, co je vlastně čas řezání, zjistíme, že je to vážený součet délek kódů, kde váhy jsou četnosti znaků. To je ale přeci celková délka zakódovaného textu! Jak jsme o kousek výš dokázali, menší už být nemůže...

Tomuto optimálnímu prefixovému kódu se říká Huffmanovo kódování.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-5-5.c`

Tomáš „Palec“ Maleček

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

24-5-6 Minové pole

Je zřejmé, že minové pole samotné má velikost $\mathcal{O}(MN)$ a celé je musíme vypsat, budeme se tedy snažit o právě takovou složitost.

Nejprve jednorozměrná varianta: tam obdélníky popisující dosah min (dále jen obdélníky) jsou vlastně úsečky, a tak nás jen zajímá, kde na řádce začínají a kde končí.

Uvědomíme si, že nás často budou zajímat hranice, a tak si vytvoříme pole o délce $n + 1$, které místo políček v matici obsahuje informace o hranách čtverečků. To nám pomůže vyřešit případy 1×1.

Toto pole budeme chtít projít právě jednou, a tak si do něj uložíme levé a pravé hranice obdélníků na vstupu. Na levou hranici uložíme `+1` a na pravou `-1`. Pak budeme procházet naše pole hranic zleva doprava, v pomocné proměnné budeme udržovat součet plus a mínus jedniček, a kdykoli budeme uvnitř čtverečku (tedy mezi hranicemi), tak vypíšeme pomocnou proměnnou.

Teď ještě tu těžší část algoritmu – dvourozměrné obdélníky. Nedala by se stejná myšlenka s plus a mínus jedničkami použít i pro obdélníky? Dala, vyřešíme nejprve sloupečky a pak zopakujeme náš původní, řádkový postup.

Chtěli bychom, aby na konci druhé fáze v pomocné matici měl každý obdélník na příslušných řádkách jednu `+1` a jednu `-1` přesně tam, kde je jeho levá a pravá svislá hranice.

Tohle by ale hravě vytvořil náš původní algoritmus, pokud bychom jej spustili na sloupečky, do levého horního rohu vložili `+1`, a do levého spodního rohu `-1`. To vyřeší levou hranici, pro pravou hranici uděláme to samé, jen s opačnými znaménky.

Pro pořádek sesumírujeme: levý horní roh dostane `+1`, pravý horní `-1`, levý spodní `-1` a pravý spodní `+1`. Spuštění algoritmu na sloupečky vytvoří levou hranici obdélníků z `+1` a pravou hranici obdélníků z `-1`. V matici samozřejmě budou vyšší čísla, protože tento postup provádíme pro všechny obdélníky současně, stejně jako v jednorozměrné variantě. Spuštění algoritmu znovu, ale nyní na řádky, už dodá správné součty do políček.

Časová složitost byla opravdu $\mathcal{O}(MN)$, protože jsme nejprve za každý obdélník uložili čtyři hodnoty do matice, a pak ji dvakrát prošli – jednou po sloupcích a jednou po řádcích. Matici jsme měli pouze jednu, o velikosti $(M+1) \times (N+1)$, a tak je paměťová složitost stejná jako časová.

Program (C):
`http://ksp.mff.cuni.cz/viz/24-5-6.c`

Martin Böhm

24-5-7 Cesta přes hranice

Mapa měst je vlastně ohodnocený neorientovaný graf, ve kterém jsou některé vrcholy celnicemi. Pro jednoduchost bude Linz vrchol A, Pasov vrchol B a Praha vrchol C.

Jednodušší varianta úlohy je vlastně jen hledání nejkratší cesty mezi vrcholy A a B a pak mezi vrcholy B a C, kde při cestování musíme zohledňovat, i kolika celnicemi jsme projeli. Na vzdálenost mezi dvěma vrcholy se budeme dívat jako na dvojici čísel (i, j), kde i je počet celnic, kterými jsme projeli, a j je vzdálenost, kterou jsme při tom urazili.

Tyto dvojice pak budeme porovnávat lexikograficky, tedy $(i, j) < (k, l)$ právě tehdy, když $i < k$ nebo $i = k$ & $j < l$. Nyní jen stačí použít Dijkstrův algoritmus a při porovnávání vzdáleností zohledňovat počet projetých celnic a máme výsledek. O detailech Dijkstrova algoritmu se můžete dočíst v naší kuchařce o haldě a Dijkstrově algoritmu.[10]

Nyní k těžší variantě. Opět hledáme nejkratší cestu z A do C přes B, akorát s tím rozdílem, že každou celnici započítáváme jen jednou. Je důležité si uvědomit, že nemusíme nutně použít nejkratší cesty A do B a z B do C, protože se nám může stát, že méně výhodnou cestu z A do B pak efektivně využijeme při cestování z B do C.

Po chvilce přemýšlení si všimneme, že v nejkratší cestě určitě bude existovat vrchol X takový, že nejdříve jdeme z A do X, pak z X do B, poté se z B vracíme zpět do X a nakonec cestujeme z X do C.

Jinými slovy při cestě z B do C nejdříve jdeme po stejné cestě, po které jsme přišli, pak se od ní odpojíme ve vrcholu X a už cestu z A do B nikdy křižovat nebudeme.

Proč? Kdybychom cestu z A do B křížili vícekrát, tak by to znamenalo, že jsme mezi těmito dvěma kříženími našli kratší cestu bez celnic, než je na příslušné části cesty z A do B, tedy by i původně bylo výhodnější jít po tomto nově nalezeném úseku.

Nyní, když víme, že nejkratší cesta takový vrchol X obsahuje, tak můžeme zkusit všechny možnosti toho, který to bude (včetně vrcholů A, B, C). Pokud zvolíme vrchol X pevně a vzdálenost X a A bude (i, j), vzdálenost X a B bude (k, l) a vzdálenost X a C bude (m, n), tak celková délka trasy při využití X bude $(i + k + m, j + 2l + n)$.

Jako X tedy vyzkoušíme všechny možné vrcholy a nejmenší vypočítaná hodnota bude naším řešením. Ke kompletnímu řešení nám už jen zbývá spočítat vzdálenosti z vrcholů A, B, C do všech ostatních vrcholů a to uděláme tak, že pro každý z nich zvlášť spustíme Dijkstrův algoritmus a tím například zjistíme vzdálenost vrcholu A od všech ostatních vrcholů.

Časová složitost obou variant je stejná jako časová složitost Dijkstrova algoritmu, tedy $\mathcal{O}(n^2)$, nebo $\mathcal{O}((n + m) \log n)$, pokud v Dijkstrově algoritmu používáme haldu.

Ve vzorovém zdrojovém kódu můžete vidět řešení těžší varianty v jazyce C++. Pro přehlednost hlavních částí algoritmu není počítána výsledná cesta, ale pouze optimální vzdálenost.

Program (C++):
`http://ksp.mff.cuni.cz/viz/24-5-7.cpp`

Karel Tesař

24-5-8 Jak hraje deskovky počítač?

Úkolem bylo prozkoumat Dvonn a zamyslet se nad součástmi algoritmu Alfa-beta, které jsou specifické pro tuto hru. To je právě zajímavá část vývoje umělé inteligence robota hrajícího hru, pokud je kostra algoritmu již daná. Do Dvonnu se sice pustili jen dva odvážlivci, nicméně obě řešení se mi líbila.

Nezaručuji, že zde předvedené myšlenky povedou k nejlepšímu možnému počítačovému soupeři, určitě lze toto řešení v mnohém ještě vylepšit. Předpokládána je alespoň

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

povšechná znalost pravidel.[11] Tahem se myslí tah jednoho hráče (někdy také půltah).

Zaměříme se především na část hry po rozestavení kamenů. Pro první část, tedy rozestavování kamenů, je doporučenou strategií dávat své kameny na okraj a co nejblíže červeným kamenům, ale nepokládat moc svých kamenů vedle sebe. Podle tohoto popisu je možné udělat ohodnocovací funkci pro Alfa-beta prohledávání.

Reprezentace pozice a generování tahů

Základem programu jistě musí být nějaká *reprezentace pozice* a na ní postavené generování možných tahů. Políčka na desce jsou v šestiúhelníkové mřížce, tu však lze reprezentovat přímo jen těžko. Proto políčka na desce trochu posuneme, přesněji řečeno i-tou řádku posuneme o $(i-1)/2$ políček a dostaneme dvourozměrné pole.

Lépe půjde transformace pochopit z obrázku, který je převzat z řešení Vojty Hlávky. `X` znamená herní políčko, políčka `S` jsou sousedé `P`, `.` je políčko mimo desku (nicméně ve výsledném poli být musí):

```
  X X X X X X X X X . .
 X X X X S S X X X X .
X X X X S P S X X X X
 . X X X X S S X X X X
. . X X X X X X X X X

            ↓

X X X X X X X X X . .
X X X X S S X X X X .
X X X X S P S X X X X
 . X X X X S S X X X X
 . . X X X X X X X X X
```

V každém prvku tohoto dvourozměrného pole musí být uložena barva a výška sloupku (0 pro prázdné políčko), a jestli se někde ve sloupku nachází červený kámen. Jednou možnou implementací je mít 3 pole, každé pro jednu vlastnost. Spolu s proměnnou značící, kdo je na tahu, máme takto kompletní reprezentaci pozice.

Generování tahů lze udělat jednoduše procházením všech políček a pro sloupky (věžičky) hráče na tahu vyhledat, kam mohou skočit. Nicméně v koncovce, kdy je spousta políček prázdných, už se vyplatí generovat tahy chytřeji.

Jednou z možností je udržovat si pro oba hráče seznam jejich sloupků na desce. Ten pak stačí projít a podívat se, kam mohou sloupky skočit. Seznamy se vygenerují před prohledáváním a budou se udržovat při provádění a vracení tahů.

Možná ještě efektivnější, ale složitější na implementaci je před prohledáváním vygenerovat všechny tahy a pak jen udržovat jejich seznam. Je však třeba si dát pozor při odstraňování kamenů na to, že bude třeba smazat mimo jiné i tahy vedoucí mezi odstraněné kameny.

Provádění tahů je většinou přímočará záležitost, v této hře však má háček: je třeba zjišťovat, jestli se nějaká skupina kamenů nestala po tahu nedosažitelnou od červených kamenů.

Podíváme se proto na okolní kameny právě odebraného kamene, konkrétně na to, jestli se vyskytují v nějakých shlucích (např. dva sousedi vedle sebe, prázdné políčko, dva sousedi a prázdné políčko dávají dva shluky). Shluky mohou být maximálně tři. Pokud je shluk jen jeden a není skákáno s věžičkou obsahující červený kámen, určitě se nic odstraňovat nebude.

Jinak je nejspíš nutné spustit prohledávání, v kterých komponentách příslušejících shlukům jsou červené kameny. Asi nejlépe to půjde prohledáváním do šířky, dokud v každé komponentě nenajdeme červený kámen nebo ji neprojdeme celou (v tom případě ji odstraníme).

Pokud neskáčeme se sloupkem obsahujícím červený kámen, můžeme zastavit prohledávání už po odstranění všech komponent až na jednu – určitě v ní červený kámen někde bude.

Při prohledávání stromu hry je třeba tahy i vracet, je tedy nutné udržovat si historii provedených tahů. Vracení odstraněných kamenů lze udělat pomocí spojového seznamu. Ten se vytvoří při provedení tahu a při jeho vracení se projde.

Pro účely určení, kdo vyhrál, se hodí udržovat si součet výšek sloupků hráče, což lze jednoduše doplnit do provádění a vracení tahů.

Ohodnocování pozice a tahů

Ohodnocování pozice bývá pro algoritmy založené na Minimaxu nejspíše tou nejtěžší částí. Na jednu stranu by mělo být velmi rychlé, vyplatí se totiž prohledávat o jedna hlouběji, než mít pomalou ohodnocovací funkci. Na druhou stranu chceme umět rozlišit slibné pozice od těch špatných. Tahle část řešení úlohy je bez praktického vyzkoušení nejvíce diskutabilní.

Z hlediska efektivity není dobré, aby ohodnocovací funkce prošla v každém listu prohledávacího stromu celé herní pole, hodí se tedy udržovat si některé vlastnosti inkrementálně – měnit je jen při provádění a vracení tahů na základě políček ovlivněných tím tahem. Z takových vlastností už se pak může spočíst výsledná hodnota v každém listu v čase nezávislém na velikosti herní desky.

Přestože vyhraje ten, kdo má vyšší součet výšek věžiček, není podle toho možná dobré ohodnocovat, protože některé věžičky může jednoduše vzít soupeř. Spíše je zajímavější určit pro vyšší věžičky poblíž červeného kamene, kdo by vyhrál, kdyby se o tu věžičku začalo bojovat (hráči by na ni střídavě pokládali kameny).

Je tedy třeba vědět, které kameny mohou na tu věžičku doskočit, a udržovat si součet ovládaných věžiček pro oba hráče (ale jen těch poblíž červených kamenů). Na druhou stranu je nutné dát si pozor, jelikož některé kameny mohou napadat či chránit více věžiček najednou.

Zajímavou heuristikou může být počet možných tahů, tedy kdo má více tahů, může lépe ovlivňovat hru a je ve výhodě. V koncovce často vyhraje ten, kdo zahraje posledních pár tahů, přičemž soupeř musí kola vynechávat, protože mu došly tahy.

Výhodné jsou často jen tahy vedoucí na soupeřův nebo červený kámen anebo chránící vlastní vysokou věžičku, ty by měly mít větší vliv na hodnocení. Je však třeba jejich počet udržovat při provádění tahů, což nemusí být lehké.

Někdy je výhodné mít ve svém sloupku červený kámen, protože pak s ním lze uskočit v případě možnosti připravit soupeře o jeho věžičky. Sloupek s červeným kamenem se asi hodí započítávat, jen když má možnost někam se pohnout.

[11] `http://deskovehry.blogspot.com/2009/10/pravidla-dvonn.html`

Pokud bychom toto dokázali udržovat inkrementálně, ohodnocení pozice hráče by vypadalo takto, přičemž konstanty chtějí ještě doladit:

$$\texttt{pocetTahu} + 2 \cdot \texttt{pocetTahuNaVezSoupere} + 10 \cdot \texttt{soucetOvladanychVezicek} + 30 \cdot \texttt{pocetVezicekSCervenymKamenem}.$$

Ohodnocení pozice z pohledu bílého hráče je pak jednoduše ohodnocení bílého mínus ohodnocení černého. Z pohledu černého to samé vynásobené -1.

Alfa-betě se kvůli efektivnímu ořezávání hodí mít tahy seřazené od těch nejlepších. Jako první se asi vyplatí vyzkoušet tahy, které odstraní více soupeřových kamenů než našich nebo které tomuto odpojení napomáhají (po tahu půjde skupina odpojit jedním tahem).

Potom bývají dobré tahy, díky kterým můžeme nějakou vysokou věžičku získat, a dále ty, při nichž skáčeme na soupeřův kámen. Až naposledy se vyplatí zkoušet tahy, při nichž skočíme na svůj vlastní kámen, což je většinou nevýhodné.

Herní strom

U hry se často zkoumá *velikost herního stromu*, aby bylo možné odhadnout, jak moc těžké je hru vyřešit, tedy najít vyhrávající strategii. Definuje se jako počet listů stromu, neboli počet různých her, které je možné sehrát.

Obvykle se nedá spočíst přesně a odhaduje se umocněním typického větvení (počtu možných tahů hráče v nějaké pozici) na maximální délku hry (někdy maximální až na výjimečně dlouhé hry). Pro jednoduchost se omezíme na jedno konkrétní rozmístění kamenů, tj. vynecháme první část hry.

Pro Dvonn je možné odhadnout větvení počtem kamenů hráče na začátku (23) krát počet možných směrů (6), tedy 138. V každém tahu hráče se musí uvolnit alespoň jedno políčko a na konci musí být alespoň jedno políčko obsazené, což dává 48 půltahů a tedy maximálně $138^{48} = 5{,}18 \cdot 10^{102}$ možných her. Proti tomu je odhadovaný počet atomů ve vesmíru jako nic.

Skutečná velikost herního stromu je samozřejmě o dost nižší a odhad na větvení lze podstatně zlepšit. Můžeme například využít faktu, že počet sloupků při hře neustále klesá a tedy klesá i počet možných tahů.

Další věc, která se pro hry odhaduje, je počet dosažitelných stavů. Jelikož jeden stav se ve stromě může vyskytovat mnohokrát, bývá toto číslo mnohem menší, přesto však pořád astronomické. Odhadnout tento počet shora je pěkné kombinatorické cvičení.

Složitost různých her a více informací lze najít na Wikipedii.[12] Tolik v „krátkosti“ pro Dvonn. Pokud vás toto téma zajímá hlouběji, můžete se mi ozvat. :-) Užijte si léto!

Pavel „Paulie“ Veselý

Tento ročník pro vás připravovali

Opravující a autoři úloh
Martin Böhm
Jan Bok
CodEx
Pavel Čížek
Karel Král
Tomáš „Palec“ Maleček
Martin „Medvěd“ Mareš
Jan „Moskyto“ Matějka
Lucie Mohelníková
Jitka Novotná
Jiří Setnička
Karel Tesař
Michal „Vorner“ Vaner
Pavel „Paulie“ Veselý
Peter „pizet“ Zeman a.k.a. Andrej Kupecký

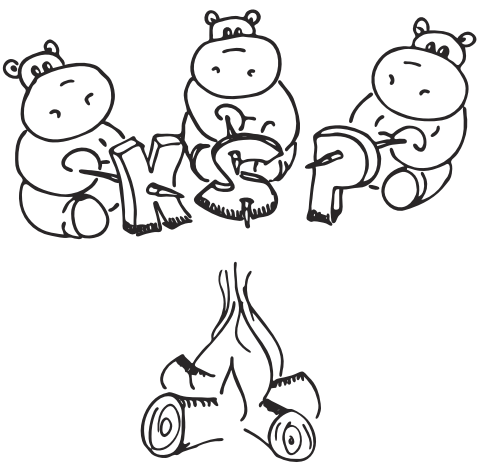
Autoři příběhů
Pavel Veselý (1. série)
Jan Matějka (2. série)
Martin Böhm (3. série)
Jiří Setnička (4. série)
Radim Cajzl (5. série)

Autoři kuchařek
Karel Tesař (Složitost a Intervalové stromy)
Jiří Setnička (Geometrie)

Seriál připravovali Pavel „Paulie“ Veselý a Lukáš Lánský.

Obrázky s hrochy kreslila Lucie Mohelníková, pár jich stvořil také Jan „Moskyto“ Matějka a některé archivní kreslil Martin „Bobřík“ Kruliš.

O **technické zázemí** se starali převážně Martin Böhm, Radim „Rumcajz“ Cajzl, Tomáš „Palec“ Maleček, Martin „Medvěd“ Mareš, Jan „Moskyto“ Matějka a Jiří Setnička.



[12] http://en.wikipedia.org/wiki/Game_complexity

Výsledková listina dvacátého čtvrtého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*2451*	*2452*	*2453*	*2454*	*2455*	*2456*	*2457*	*2458*	*série*	*celkem*
0.					9	11	11	11	9	13	13	15	63,0	300,0
1.	Vojtěch Hlávka	GŠlapanice	3	15	9	7	10	8	4	13	13	15	59,5	268,3
2.	Martin Raszyk	G_Karvina	2	10	9		10			13	7		39,6	235,3
3.	Lukáš Ondráček	GVolgogrOS	3	5		7	10	8		13			41,9	218,1
4.	Dominik Macháček	GLanškroun	3	5	5			6	1	3	7		31,3	167,2
5.	Mark Karpilovsky	GJarošeBO	3	5	9		9			13			32,1	165,0
6.	Michal Pokorný	SŠkybernHK	4	9									0,0	161,0
7.	Vojtěch Vašek	GHli	3	4	9		10	8	8,5	13			51,1	155,7
8.	Alexander Mansurov	GNVPlániPH	3	9						13			13,0	150,4
9.	Jiří Eichler	SlovanGOL	4	12						0			0,0	138,0
10.	Martin Španěl	ArcibisGPH	3	3	7,5	8	10	6		0		11	51,2	136,5
11.	Jan Knížek	G_Strakon	1	5	3	6	5	4			3,5		31,9	116,1
12.	Jerguš Greššák	ŠPMNDaGB	3	8									0,0	110,9
13.	Ondřej Mička	GJírovcČB	3	13	7,5		11				13		31,2	108,2
14.	Matej Lieskovský	GOmskPha	2	5	5	3	10	8					31,7	106,9
15.	Vojtěch Sejkora	SPSE_Pard	3	9									0,0	100,9
16.	Michal Punčochář	GJírovcČB	2	5									0,0	99,1
17.	Štěpán Trčka	GSlavičín	1	4	4					3			11,4	96,5
18.	Dalimil Hájek	GKepleraPH	1	5	5			5	1	3	4		27,3	94,6
19.	Rastislav Rabatin	GJHroncaBA	3	3									0,0	90,8
20.	Lukáš Folwarczný	GKomHavíř	4	10			10			13			23,1	89,7
21.	Jan-Sebastian Fabík	GJarošeBO	2	5						13			13,0	87,0
22.	Aneta Šťastná	GOmskPha	2	4				8					9,7	80,2
23.	Martin Mirbauer	PORGPha	4	3									0,0	77,1
24.	Martin Šerý	GJírovcČB	2	1	4,5	8			7	13	6,5		49,1	49,1
25.	Jonatan Matějka	GJírovcČB	2	10	5			6					11,7	46,0
26.	Jitka Fürbacherová	GKlatovy	3	6	4,5	8	1	4	6				30,0	45,7
27.	Kateřina Zákravská	GJar	3	1	5	8	6	6	8				44,6	44,6
28.	Ondřej Cífka	GNAlejíPH	3	8									0,0	43,4
29.	Ondřej Hübsch	GArabskáPH	2	15						0			0,0	43,0
30.	Petra Pelikánová	GJarošeBO	3	1	5	8	5	7	3				41,3	41,3
31.	Ráchel Sgallová	GZborovPH	2	1	5	7	10	3		3			40,7	40,7
32.	Petr Houška	GJírovcČB	2	1	4				6,5	13	8		39,3	39,3
33.	Anna Zákravská	GJar	3	1	5	8		6	7		1,5		38,9	38,9
34.	Joel Jančařík	MensaG	4	1									0,0	38,8
35.	David Bernhauer	GZborovPH	4	4									0,0	38,5
36.	Jan Hadrava	GZborovPH	4	5									0,0	32,6
37.	Sabína Fraňová	GDubNVahom	3	1	5	8		1	2	3			31,1	31,1
38.	Jindřich Pilař	GBroumov	4	8									0,0	30,2
39.	Tereza Hulcová	GKlatovy	3	7									0,0	28,5
40.	Bohumil Mravenec	GArabskáPH	3	2						2			4,5	25,0
41.	Václav Volhejn	GKepleraPH	-1	2		4			6				15,0	24,8
42.	Pavel Kratochvíl	VOŠGSvětlá	4	15									0,0	24,5
43.	Radovan Švarc	G_ČTřebová	1	1	5		8	0			2,5		23,3	23,3
44.	Josefina Mádrová	GDobruška	4	3									0,0	21,4
45.	Pavel Salva	VOŠŠumperk	2	3									0,0	19,4
46.	Dominik Smrž	GOhradníPH	2	8			10	8					19,1	19,1
47.	Pavel Bárta	SPŠTrutnov	2	1	3	1	1	1	1		1,5		17,2	17,2
48.	Štěpán Šimsa	GJungmanLT	3	14									0,0	16,3
49.	Tomáš Velecký	GBezručeFM	1	3	3,5					7			15,9	15,9
50.	Jan Žárský	VSSKopř	1	1									0,0	14,1
51.	Zuzana Vozárová	GJHroncaBA	4	1									0,0	13,8
52.	Vojtěch Bednárik	MG_Vsetín	4	1						7			10,7	10,7
53.	Vojtěch Polívka	GMikulášPL	4	1									0,0	9,5
54.	František Zajíc	G_Nymburk	-1	1									0,0	9,2
55.	Tomáš Hromada	MG_Vsetín	4	1						5			9,0	9,0
56.	Michal Hruška	GJirsíkaČB	4	1									0,0	7,6
57.	Matěj Židek	GBroumov	4	7									0,0	6,2
58.	Vladan Glončák	GĽŠtúraTN	3	1									0,0	6,0
59.	Břetislav Hájek	GČesBrod	-2	3									0,0	5,9
60.	Juda Kaleta	GKlatovy	3	8									0,0	5,2
61.	Jan Pavlík	VOŠŠumperk	4	1									0,0	1,3