Korespondenční Seminář z Programování

29. ročník | **KSP** | **Duben 2017**

Vzorová řešení třetí série dvacátého devátého ročníku KSP

29-3-1 Verbování

Představme si, že jsme u města Leyfast a procházíme očíslované domy. V každém z domů máme několik možností, jak se rozhodnout. Abychom nalezli nejlepší řešení, můžeme zkusit každou možnou kombinaci rozhodnutí a vybrat tu nejvýhodnější, ale to by trvalo příliš dlouho.

Můžeme také zkusit vybírat další krok *hladově*, neboli vybírat možnost neporušující pravidla, která nám lokálně (pro tento dům) dá nejlepší výsledek. To bude sice rychlé, ale nedostaneme takhle správnou odpověď. Zkuste si to na nějakých vstupech, k rozbití tohoto postupu stačí již ukázkový vstup ze zadání.

Problémem hladového řešení je, že nijak nerespektuje to, že volba v i-tém domě ovlivňuje možné volby v okolních domech. Připomeňme si, co můžeme v domě udělat. Pokud skrz domy půjdeme odzadu, tak můžeme:

- Naverbovat vojáka, pokud jsme tak neučinili v předchozím a neučiníme-li tak v ani následujícím domě.
- Vzít zbraně, pak musíme nutně naverbovat vojáka v následujícím domě.
- Nevzít zbraně ani nenaverbovat vojáka.

Volba, která ovlivňuje výběr v okolních domech, je verbování. Pokud se zkusíme podívat na problém omezený jen na prvních i domů, tak by nás pro další rozhodování mohlo zajímat, jaké nejvyšší bojeschopnosti umíme dosáhnout, pokud si v i-tém domě dovolíme naverbovat vojáka a pokud si zde vojáka nepovolíme naverbovat.

Postavíme si pro to rekurzivní funkci $B(i, (true|false))$, která bude počítat přesně toto. Pokud se nám povede ji spočítat, tak celkovou maximální bojeschopnost získáme zavoláním $B(N, true)$.

Teď si budeme muset sestavit funkci (pro připomenutí, v Z_i je zisk bojeschopnosti při braní zbraní z i-tého domu, v V_i to samé, ale pro verbování z i-tého domu).

- Pro $i \leq 0$ bude mít funkce vždy hodnotu 0.
- $B(i, false)$ bude maximum z:
  - $Z_i + V_{i-1} + B(i-2, false)$ (budeme brát zbraně, což vynutí verbování v $i-1$; lze použít jen pro $i > 1$)
  - $B(i-1, true)$ (nebudeme dělat nic)
- $B(i, true)$ bude maximum z $B(i, false)$ a navíc:
  - $V_i + B(i-1, false)$ (budeme verbovat)

Takovouto funkci lze jednoduše naprogramovat. Horší je exponenciální časová složitost způsobená větvením výpočtu v každém domu. Naštěstí funkce, kterou jsme právě definovali, závisí pouze na dvou parametrech: i a *verbovat*.

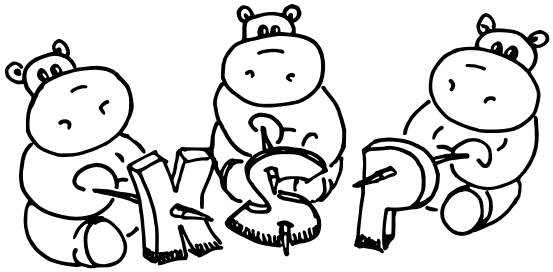
Výsledky volání si tak můžeme ukládat do tabulky velikosti $2N$. Při opakovaném zavolání pak stačí vrátit už dříve spočítaný výsledek. Spočítáme tedy nejvýše $2N$ hodnot funkce B, proto dostáváme lineární složitost vzhledem k počtu hodnot na vstupu.

Zbývá domyslet, jak navíc zjistit jeden z plánů verbování nabývající hodnoty $B(N, true)$. Například můžeme spustit znovu trochu modifikovanou funkci počítající B, která bude nyní vracet odkaz na i-tý prvek spojového seznamu, který reprezentuje jeden ze znaků (z,v,-). Všimněme si, že takto použijeme jen N položek seznamu, protože už máme spočítány hodnoty B a v každém kroku tak voláme pouze jednou naši modifikovanou funkci.

Na trochu (ale jen konstantně) elegantnější řešení s nahrazením spojového seznamu řetězcem se můžete podívat do vzorového programu v C++.

Program (C++):
`http://ksp.mff.cuni.cz/viz/29-3-1.cpp`

Marek Černý



29-3-2 Trpasličí závaží

K porovnávání váhy sad závaží a protizávaží se dalo použít několik různých postupů. Jedním z nich bylo převést zápis na takový, který bude obsahovat jen jedničky a nuly, a tento pak porovnat jako se porovnávají binární čísla. Druhou možností je porovnat zápisy v této „rozšířené dvojkové soustavě“ přímo.

Za chvíli si ukážeme obě možnosti, nejdříve ale proveďme pozorování. Podobně jako v klasické dvojkové soustavě má každá pozice dvojnásobnou hodnotu než pozice předchozí. Když si budeme postupně sčítat hodnoty na dalších pozicích (za nějakou pozicí s hodnotou x), tak nejdříve dostaneme polovinu x, z další pozice čtvrtinu (neboli polovinu té zbývající poloviny do x) a tak dále. Každá další pozice nám součet více přiblíží k hodnotě x, ale nikdy ho nepřesáhne. Protože pozic v binárním čísle je konečně mnoho, přiblíží se na jedničku a víc už ne – matematici by řekli:

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Takže pokud bude nejlevější nenulová pozice čísla zapsaného v této soustavě záporná, tak i kdyby všechny ostatní pozici již byly kladné, dostaneme nejvýše -1 (a tedy celé číslo bude záporné). A naopak, pokud bude kladná, tak číslo bude kladné.

Podle tohoto můžeme udělat první porovnání a pokud mají nejvyšší pozice obou porovnávaných čísel rozdílná znaménka, můžeme rovnou oznámit výsledek a končíme. Dál tedy budeme zabývat jen případy, kdy jsou znaménka na nejvyšších pozicích stejná.

Další triviální pozorování je, že překlopením všech znamének na opačná vlastně jen změníme znaménko celého čísla.

Převod do dvojkové soustavy

Pro jednoduchost budeme popisovat převod pro čísla, jejichž nejvyšší nenulová pozice je kladná (kdyžtak si je podle předchozího pozorování překlopíme a zapamatujeme si, že je číslo vlastně záporné).

Budeme se chtít zbavit všech výskytů -1 v zápisu čísla. Všimněme si, že následující zápisy můžeme bez změny hodnoty převádět:

- $(1, -1) \to (0, 1)$
- $(0, -1) \to (-1, 1)$

V obou situacích děláme to, že přičteme dvojici $(-1, +2)$, což je v součtu nula, a tím vlastně posíláme mínus jedničku dál doleva.

Můžeme tedy zahájit převod od nejmenšího řádu zprava a takto si mínus jedničky průběžně eliminovat, nebo si je posílat dál doleva. Máme ale jednu situaci, kterou jsme si nepopsali – co když se nám vedle sebe objeví dvě mínus jedničky?

Na chvíli si povolíme použít i hodnotu -2 a podívejme se, co se nám při přičtení dvojice $(-1, +2)$ může stát:

- $(-1, -1) \to (-2, 1)$
- $(-1, -2) \to (-2, 0)$
- $(0, -2) \to (-1, 0)$
- $(1, -2) \to (0, 0)$

Zkusme si to na čísle 5 zapsaném jako $1, 0, -1, -1$. Při převodu zprava dostaneme postupně $1, 0, -2, 1$, pak $1, -1, 0, 1$ a nakonec $0, 1, 0, 1$, což odpovídá číslu 5.

Převod tedy umíme udělat lineárním průchodem číslem od nejmenšího řádu k největšímu a eliminováním mínus jedniček pomocí přičítání vzoru $(-1, +2)$. Pokud takto převedeme obě čísla, už je snadno porovnáme binárně (průchodem od největšího řádu a hledáním první pozice, kde se liší).

Porovnání odečtením

Pokud vám převod do normální dvojkové soustavy přijde jako nehezký trik, dá se porovnání udělat i odečtením jednoho čísla od druhého. Podle toho, jestli nám výsledek vyjde kladný, nebo záporný (což poznáme podle znaménka nejvyšší jedničky), určíme snadno, které číslo je větší.

Odečítání můžeme dělat klasickým školním postupem od nejmenšího řádu. Pokud nám výsledek odečtení vyjde 1, 0 nebo -1, je vše v pořádku. Pokud nám vyjde menší, než -1, tak musíme udělat převod -1 do vyššího řádu (a k tomu současnému přičteme $+2$, vlastně opět aplikujeme vzor $(-1, +2)$). Pokud nám vyjde naopak větší než 1, přičteme -2 a pošleme převod 1 (tedy použijeme vzor $(+1, -2)$).

Pojďme se podívat na průběh výpočtu třeba čísla $1, -1, -1$, od kterého odečteme číslo $1, 1$ (neboli $1 - 3 = -2$). V prvním kroku nám na poslední pozici výsledku vznikne -2, což převedeme na 0 a do vyššího řádu pošleme -1. Na druhé pozici dostaneme -3 (i s převodem), což převedeme na -1 a do vyššího řádu pošleme -1. A nakonec na nejvyšší pozici dostaneme $1 - 1 = 0$, čímž získáme správný výsledek $0, -1, 0$.

Tento postup zabere také lineární čas vzhledem k velikosti vstupních čísel. Na oba postupy se můžete podívat v přiloženém programu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-3-2.py`

Jirka Setnička

29-3-3 Skřetí věže

Mnoho z vás přišlo s nápadem zkoušet různé přímky a ověřit, jestli se náhodou nejedná o hledanou osu. Všechny možné přímky však určitě vyzkoušet nemůžeme, těch je nekonečně mnoho. Které přímky tedy připadají v úvahu?

Využijeme toho, že každý bod musí mít při osové souměrnosti svůj obraz. Očíslujeme si tedy body postupně P_0, P_1, …, P_{N-1}. Budeme nejprve předpokládat, že bod P_0 se zobrazí na nějaký jiný bod a ne sám na sebe.

Všimněte si, že pokud bychom věděli, na který bod se P_0 zobrazí, je osa souměrnosti jednoznačně určená: musí to být osa úsečky spojující P_0 s jeho obrazem. My samozřejmě nevíme, na který bod se P_0 zobrazí, ale můžeme vyzkoušet všechny možnosti. Tím získáme $N - 1$ přímek, mezi nimiž se určitě hledaná osa nachází (za předpokladu, že nějaká osa existuje a bod P_0 není obrazem sebe sama).

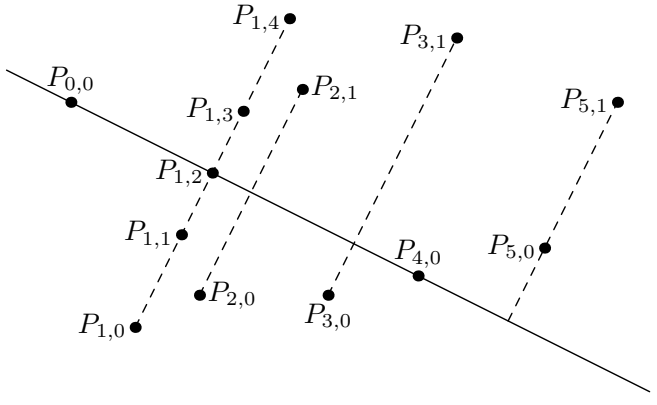
Rozmyslíme si ještě případ, kdy P_0 je sám sobě obrazem a nachází se tedy přímo na ose. Nejjednodušší je vzít místo P_0 bod P_1 a k němu stejným postupem zkoušet body P_2 až P_{N-1}. Takto vyřešíme případy, kdy alespoň jeden z bodů P_0 a P_1 neleží na ose. Pokud by oba ležely na ose, je osou přímka P_0P_1 – tu také přidáme do seznamu kandidátů.

Tímto postupem jsme tedy získali $2N - 2$ přímek, mezi nimiž se určitě osa nachází (existuje-li). Stačí pro každou z přímek ověřit, jestli osou skutečně je, tj. jestli má každý bod svůj obraz.

Nejprve si pro každý bod spočítáme, kam by se při dané ose zobrazil. Pokud bod leží přímo na ose, je sám sobě obrazem. Pokud na ose není, chce to trochu počítání, ale nic náročného. Vezmeme přímku procházející daným bodem, která je kolmá na osu, a spočítáme její průsečík s osou. Tento průsečík se musí nacházet ve středu úsečky spojující daný bod a obraz, takže souřadnice obrazu se už jednoduše dopočítají.

Pro každý bod tedy víme, kam se zobrazí. Teď už stačí zkontrolovat, že v místě obrazu leží nějaký jiný bod – v opačném případě zkoumaná přímka osou není. Pokud bychom při kontrolování obrazu pokaždé procházeli všechny body, bude nám kontrola jednoho obrazu trvat $\mathcal{O}(N)$, kontrola všech obrazů $\mathcal{O}(N^2)$ a prozkoumání všech $2N - 2$ přímek $\mathcal{O}(N^3)$.

Tento postup lze zrychlit vhodným setříděním bodů. Pro každou potenciální osu body setřídíme podle polohy jejich průmětu na osu (to je pata kolmice k ose, na níž leží daný bod). Všimněte si, že tento průmět jsme už spočítali při hledání souřadnic obrazu. Můžeme využít toho, že pokud má být bod P_k obrazem P_ℓ, budou souřadnice jejich průmětů na osu stejné.



Pokud máme tedy takto setříděné body, můžeme je brát postupně. Vždy vezmeme všechny body se stejnými souřadnicemi průmětu a uložíme je do dalšího pole. Toto další pole opět setřídíme, tentokrát podle pozice na přímce, na které se všechny nacházejí (to je nějaká přímka kolmá k ose). Je zřejmé, že první bod v tomto menším seznamu musí být obrazem posledního, druhý předposledního atd. Pokud si nějaká tato dvojice není navzájem obrazem, některý bod z dvojice nemá obraz a zkoumaná přímka není osou.

Původní setřídění bodů zvládneme v čase $\mathcal{O}(N \log N)$, třídění menších seznamů stihneme ještě rychleji. Kontrolu po setřídění stihneme v lineárním čase. Jelikož zkoumaných přímek je stále lineárně, činí celková složitost $\mathcal{O}(N^2 \log N)$.

Celý tento algoritmus můžeme ještě zrychlit použitím hešovací tabulky.[1] Jednoduše si souřadnice všech bodů do jedné takové tabulky uložíme. Pak pro každý bod spočítáme souřadnice jeho obrazu podle dané osy a podíváme se do tabulky, jestli se tam bod s takovými souřadnicemi nachází. Jelikož zjištění existence v hešovací tabulce proběhne v průměrně konstantním čase, získáme ověření osy v čase průměrně lineárním. Celkově tedy v průměrném čase $\mathcal{O}(N^2)$. Toto řešení už stačilo na získání plného počtu bodů.

Těžiště na pomoc

Pojďme se ale ještě podívat na řešení jiného typu, třeba povede k ještě lepším výsledkům. Někteří z vás chytře využili toho, že těžiště bodů se jistě nachází na ose souměrnosti. Proč tomu tak vlastně je? Těžiště můžeme počítat postupně, tedy tak, že si body rozdělíme do skupinek, spočítáme těžiště skupinek a pak spočítáme těžiště těchto těžišť (každé z těchto těžišť ještě vážíme počtem bodů z příslušné skupinky).

Můžeme tedy vzít body po dvojicích – vždy si vezmeme bod a jeho obraz (body, co leží přímo na ose, necháme samostatně). Těžiště každé této dvojice (tj. střed příslušné úsečky) se nachází přesně na ose. Těžiště samostatného bodu je přímo tento bod. Každé takto spočítané těžiště se nachází na ose, tedy i těžiště těchto těžišť – jakkoli vážené – se bude opět nacházet na ose. Tím pádem tam bude ležet i těžiště původních bodů.

Poznamenáme ještě, že těžiště dokážeme spočítat v lineárním čase. Stačí spočítat průměr x-ových a průměr y-ových souřadnic jednotlivých bodů. Výsledkem jsou souřadnice těžiště.

Takto získáme jeden bod osy, musíme ještě přijít na druhý. Jeden způsob je opět zkoušet středy úseček P_0P_k pro ostatní k. Tento způsob je zdánlivě lepší oproti předchozímu v tom, že můžeme poměrně rychle odmítat přímky, které nejsou osami. Protože aby se P_0 zobrazilo na P_k, musí být přímka P_0P_k se středem S kolmá na osu TS (T je těžiště) a navíc musí TS protínat P_0P_k ve středu úsečky P_0P_k. Všechno toto dokážeme zkontrolovat v konstantním čase a rychle tak odmítnout spoustu potenciálních os.

Když však všechny tyto rychlé kontroly uspějí, musíme opět ověřit, jestli je daná přímka skutečně osou. To můžeme provést jedním ze způsobů popsaným v předchozí části.

Nicméně v obecném případě jsme si moc nepomohli. Jako účinný protipříklad se ukáže množina vrcholů pravidelného n-úhelníku sjednocená s množinou bodů rovnostranného trojúhelníku se stejným těžištěm, která způsobí, že celá množina osově symetrická není.

Sami si můžete vyzkoušet, že pokud budou vrcholy z trojúhelníku brány až jako poslední v pořadí, skutečně jsme si, co se rychlosti týká, oproti předchozímu postupu vůbec nepomohli – stále budeme muset zkoušet $\mathcal{O}(N)$ os a žádnou se nám nepodaří odmítnout rychlým způsobem. Každou osu budeme muset ověřit pomalým způsobem, celkově jsme tedy stále na složitosti $\mathcal{O}(N^2)$. Pro jiné přístupy je ještě horší případ, kdy všechny body z trojúhelníku i z n-úhelníku mají od těžiště stejnou vzdálenost.

Zdá se, že najít těžiště nám vlastně vůbec nepomohlo. Kdepak, opak je pravdou. Na následujících řádcích si ukážeme ještě rychlejší řešení, které se právě o těžiště opírá.

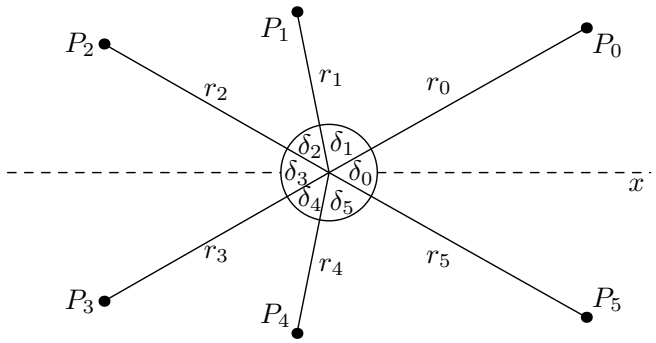
Jde to ještě rychleji

Odrazíme se od souřadnic těžiště. Všechny body posuneme tak, aby těžiště bylo na počátku souřadnicového systému a nadále budeme počítat s těmito upravenými souřadnicemi. Pak víme, že osa souměrnosti, pokud existuje, bude procházet počátkem (tj. těžištěm).

Dále budeme pracovat s takzvanými polárními souřadnicemi, tj. místo x-ové a y-ové souřadnice budeme mít u každého bodu úhel, který svírá x-ová osa s přímkou spojující daný bod s počátkem (těžištěm), a vzdálenost od počátku.

Potom si seřadíme body podle úhlu (prozatím budeme předpokládat, že žádné dva body nemají stejný úhel). Máme tedy u každého bodu P_i úhel φ_i. V tomto setříděném pořadí budeme nadále vrcholy zpracovávat, ale ukáže se, že důležité pro nás bude pamatovat si rozdíl úhlu oproti předchozímu vrcholu. Tedy pro každý vrchol spočítáme $\delta_i = \varphi_i - \varphi_{i-1}$ a pro nultý vrchol $\delta_0 = \varphi_0 + 360° - \varphi_{N-1}$. Všimněte si, že součet přes všechna δ_i nám dá 360°.

Pokud vzdálenost jednotlivých bodů budeme značit pomocí r_i, můžeme teď úhly a vzdálenosti zapsat do řetězce $s = \delta_0 r_0 \delta_1 r_1 \ldots \delta_{N-1} r_{N-1}$. Tedy jednotlivé r_i a δ_i budeme chápat jako jednotlivé znaky řetězce. Všimněte si, že z tohoto řetězce lze zpětně zrekonstruovat původní rozložení bodů (až na rotaci okolo středu, která neovlivňuje osovou souměrnost). Prostě se vždy otočíme o daný úhel δ_i a nakreslíme bod ve vzdálenosti r_i od počátku.



Představme si na chvíli, že osa x je hledanou osou souměrnosti. Uvažujme případ, kdy žádný bod neleží na pravé straně této osy (tedy neexistuje bod s $\varphi_i = 0$). Pak není těžké si představit, že některé úhly a vzdálenosti si musí odpovídat. Konkrétně $r_0 = r_{N-1}$, $\delta_1 = \delta_{N-1}$, $r_1 = r_{N-2}$, $\delta_2 = \delta_{N-2}$ atd. Pro případ, kdy bod leží na pravé straně osy x dostaneme podobné rovnosti, jen trochu posunuté, $\delta_0 = \delta_{N-1}$, $r_1 = r_{N-1}$ atd.

Každopádně si všimněte, že oba případy znamenají, že řetězec s je *skoropalindrom* (palindrom je řetězec, který se

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

stejně čte zepředu i zezadu, tedy že první znak je stejný jako poslední, druhý je stejný jako předposlední atd.). Přesněji řečeno v prvním případě dostaneme palindrom, když za s připíšeme první znak s (tj. δ_0), ve druhém, když před s připíšeme jeho poslední znak (tedy r_{N-1}).

Bohužel nemáme zaručeno, že osou souměrnosti bude osa x. Takže musíme řetězec s vhodně *zrotovat*, tj. opakovaně brát poslední znak řetězce a dát jej na první místo. Všimněte si, že pokud zrotujeme do nějakého stavu

$$r_k\delta_{k+1}\ldots\delta_{N-1}r_{N-1}\delta_0 r_0\ldots\delta_{k-1}$$

a na konec zkopírujeme první znak (r_k), výsledek je palindromem právě tehdy, když osa prochází bodem r_k. Analogicky pokud rotací dostaneme řetězec

$$\delta_k r_k\ldots\delta_{N-1}r_{N-1}\delta_0 r_0\ldots\delta_{k-1}r_{k-1}$$

a opět zkopírujeme δ_k, výsledek je palindromem právě tehdy, když osa prochází středem úsečky mezi body P_{k-1} a P_k.

Uvědomme si, že to jsou jediné možnosti, kde osa souměrnosti může ležet. Máme-li totiž těžiště T (nebo jiný libovolný bod na ose souměrnosti), bod A a jeho obraz A', tak úhel mezi přímkou TA a osou souměrnosti musí být stejný jako mezi osou a přímkou TA'. Představme si, že by tedy osa dělila nějaký úhel δ_k na úhly δ'_k a δ''_k. Nechť je δ'_k ten menší z nich a bod při tomto úhlu (P_k nebo P_{k-1}) označíme K. Bod K se musí zobrazit na jiný bod, který dává s osou úhel δ'_k (resp. $360^\circ - \delta'_k$, podle toho, jak se na to díváte), ale všechny ostatní body dávají úhel větší (resp. menší), K tedy nemá obraz a zkoumaná přímka není osa.

Stačí vyzkoušet všechny tyto rotace a podívat se zda nejsou *skoropalindromem*. Pokud si budeme uchovávat řetězec ve spojovém seznamu s ukazatelem na začátek i konec, další rotaci vytvoříme v konstantním čase, stačí odebrat prvek z konce seznamu a dát jej na začátek. Samotná kontrola, jestli je řetězec *skoropalindromem*, bude v lineárním čase. Stačí zkontrolovat jestli je první prvek shodný s předposledním, druhý s předpředposledním atd. To zvládneme pomocí dvou ukazatelů, které vždy posuneme o jednu pozici.

Nicméně to opět vypadá, že jsme si vůbec nepomohli. Jednu rotaci zkontrolujeme v čase $\mathcal{O}(N)$, ale rotací je také $\mathcal{O}(N)$, dohromady dostaneme opět $\mathcal{O}(N^2)$. Nicméně častým trikem, když hledáme vhodnou rotaci, je negenerovat nové a nové rotace, nýbrž řetězec zkopírovat dvakrát za sebe. Zkusme to také.

Původně jsme hledali rotaci s palindromem délky $2N-1$ (nezapomínejme, že N je počet bodů, délka s je tedy $2N$), stejně tak můžeme hledat palindrom délky $2N-1$ ve zdvojeném řetězci. To můžeme udělat tak, že najdeme nejdelší palindrom liché délky, pokud je delší než $2N-1$, můžeme jej jednoduše zkrátit (odebíráním vždy dvojicí znaků z kraje) na tuto délku. A jak najít nejdelší palindrom? Na to se podíváme spolu v páté sérii. Zatím jen prozradíme, že to zvládneme v lineárním čase.

Už jsme téměř na konci, ale nesmíme zapomenout ještě na jednu věc. Na začátku tohoto řešení jsme předpokládali, že žádné dva body nebudou mít přiřazený stejný úhel φ_i.

S tím se už dá celkem snadno vypořádat. Trochu upravíme konstrukci řetězce s. Když budeme mít několik bodů stejný úhel, napíšeme jejich vzdálenosti do tohoto řetězce hned za sebou v setříděném pořadí. Protože ale potřebujeme, aby se sekvence bodů se stejným úhlem četla stejně popředu i pozpátku (abychom mohli problém převést na hledání palindromu), tak ji hned za ni zapíšeme znova, v opačném pořadí. Náš řetězec může vypadat například takto: $\delta_0 r_0 r_1 r_2 r_2 r_1 r_0 \delta_3 r_3 r_3 \delta_4\ldots$

Odpovídajícím způsobem upravíme i délku hledaného palindromu. Ta je $2N+A$, kde N je počet bodů a A je počet nenulových δ_i.

Pojďme si to shrnout a podívat se na výslednou složitost. Posunout body podle těžiště, stejně tak spočítat polární souřadnice zvládneme v lineárním čase. Setříděním bodů podle úhlů strávíme $\mathcal{O}(N\log N)$. Zkonstruovat a zdvojit řetězec opět zvládneme lineárně a konečně jsme slíbili, že nalezení samotného palindromu jde také rychle. Celkově jsme se tedy konečně dostali na časovou složitost $\mathcal{O}(N\log N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-3-3.py`

Dominik Smrž & Martin „Medvěd“ Mareš

29-3-4 Mezi hlídkami

Ze všeho nejdříve úlohu převedeme na variantu, kde je zakázáno vstupovat pouze na políčka s hlídkou, ale na sousední šlápnout můžete. Uděláme to tak, že přidáme „virtuální“ hlídku na všechna pole sousedící s hlídkami ze zadání. Odpověď pro takto upravenou úlohu a vstup bude stejná jako pro původní zadání.

Jednou z možností, jak se úloha dala řešit, bylo si na začátku najít pomocí prohledávání do šířky nejkratší cesty mezi všemi dvojicemi políček. Při odpovídání na dotaz už budeme mít délku nejkratší cesty předpočítanou a můžeme ji jen vrátit.

Protože počítáme cesty na poli velikosti $N\times M$ políček, zabere nám vyhledání nejkratších cest z jednoho políčka do všech ostatních čas $\mathcal{O}(NM)$ (jedno prohledávání do šířky). Jelikož potřebujeme počítat vzdálenost mezi všemi dvojicemi políček, musíme prohledávání spustit pro každé políčko samostatně, což zabere $\mathcal{O}((NM)^2)$ času a $\mathcal{O}((NM)^2)$ paměti. To není příliš rychlé, zkusíme to vylepšit.

Rychlejší postup

Můžeme si všimnout, že vzhledem k malému počtu hlídek nejkratší cesta půjde většinu času po části pláně, kde žádné hlídky nejsou. Toho jde využít a dosáhnout tak lepší časové i paměťové složitosti. Dál v řešení budeme pracovat se souřadnicemi startu x_s, y_s a souřadnicemi cíle x_c, y_c.

Řešení si rozdělíme na dva případy – buď neexistuje hlídka, která je v obdélníku definovaném startem a cílem, a délka nejkratší cesty je tedy $|x_s - x_c| + |y_s - y_c|$, nebo nám po cestě nějaká hlídka bude překážet a budeme to muset vyřešit. Tyto dva případy také musíme být schopni odlišit, což vyřešíme v poslední části řešení.

Chtěli bychom si předpočítat nejkratší cestu mezi každými dvěma vrcholy, jenže to by trvalo příliš dlouho. Všimneme si ale, že ve sloupcích a řádcích, kde není hlídka, se „nic neděje“. Pokud je takových řádků nebo sloupců více vedle sebe, tak vždy všechny sloučíme (zkontrahujeme) do jednoho a poznamenáme si ke každému políčku, kolika políčkům odpovídá ve vertikálním a horizontálním směru. Jednotlivá zkontrahovaná políčka tak mohou odpovídat i docela velkých obdélníkům v původní pláni. Nejkratší cesty si pak předpočítáme až na této upravené pláni.

Při slučování si u každého políčka z původní pláně navíc zaznamenáme, kde je jeho „sloučená verze“ v kontrahova-

né pláni. To se nám bude později hodit a takto se k této informaci dostaneme v konstantním čase.

Na této kontrahované pláni budeme chtít hledat nejkratší cesty. Může se zdát, že po úpravě, kdy některá políčka reprezentují větší vzdálenosti, nebude běžné prohledávání do šířky stačit, ale můžeme si rozmyslet, že díky kontrahování vždy celých řádků a sloupců bude i obyčejné prohledávání do šířky stále dostávat políčka ve správném pořadí – takové prohledávání do šířky totiž přiřadí políčkům stejná ohodnocení, jako kdybychom ho spustili na původní pláni. Nad tímto prohledáváním do šířky také můžeme přemýšlet jako nad Dijkstrovým algoritmem, který namísto haldy používá frontu.

Protože hlídek bylo k, tak takto upravená pláň má rozměry $O(k^2)$ (mezi sousedními hlídkami je maximálně jeden zkontrahovaný sloupec nebo řádek) a předpočítat si všechny nejkratší cesty tedy bude trvat $O(k^4)$.

Potřebujeme ještě umět zjistit, zda je v obdélníku definovaném startem a cílem hlídka. To můžeme udělat jednoduše pomocí dvoudimenzionálních prefixových součtů (o nich si můžete přečíst třeba v naší kuchařce základních algoritmů).[2] Hlídky budeme považovat za jedničky a prázdná políčka za nuly. V daném obdélníku pak bude hlídka právě tehdy, když je v něm nenulový součet.

Dotaz pak bude vypadat následovně: Pokud mezi políčky není žádná hlídka, délka nejkratší cesty je $|x_s - x_c| + |y_s - y_c|$. Pokud mezi nimi hlídka je, využijeme naší kontrahované pláně, kde máme pro každou dvojici políček předpočítanou nejkratší cestu

Drobnou nesnází je, že start (nebo cíl) mohou ležet uvnitř nějakého kontrahovaného obdélníku. Můžeme si rozmyslet, že část cesty, která je v tomto obdélníku, může jít libovolnou nejkratší cestou do rohu nejbližšího k cíli (respektive startu), tuto část cesty spočítáme jako v případě výše.

A dál už pak vyhledáváme jen ve zkontrahované pláni, respektive v předpočítané datové struktuře délek cest mezi dvojicí zkontrahovaných políček.

Předpočítání kontrahované pláně bude trvat $O(MN + k^4)$ a stejně prostoru bude zabírat výsledná datová struktura. Na dotazy pak budeme schopni odpovídat v konstantním čase.

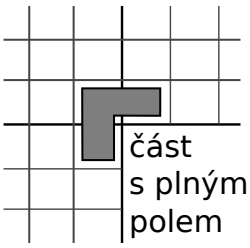
Kuba Tětek

29-3-5 Dračí zámek

K zadání této úlohy jste všichni dostali nápovědu, totiž kuchařku o metodě Rozděl a panuj. Toho jste všichni správně využili, a třebaže došlá řešení využívala různé přístupy, vždy byly založeny na této metodě.

Takže jak se k úloze správně postavit? Připomeňme, že čtvercová mřížka má rozměry $N \times N$, kde N je nějaká mocnina dvojky. Je snadné si všimnout, že pro $N = 1$ má úloha triviální řešení: máme jediné plné pole.

Pro větší N chceme celé zadání rozdělit na menší úlohy. Mřížku uprostřed rozsekneme na čtyři podmřížky, každou s rozměry $\left(\frac{N}{2}\right) \times \left(\frac{N}{2}\right)$. Na podmřížku, kde se vyskytuje plné pole, můžeme rekurzivně zavolat stejný algoritmus. Pro ostatní podmřížky si pomůžeme tak, že do rohu, který vzájemně tvoří, vložíme navíc dílek:



V každé z nich se teď nachází plné pole, tudíž i na ně se můžeme zavolat rekurzivně.

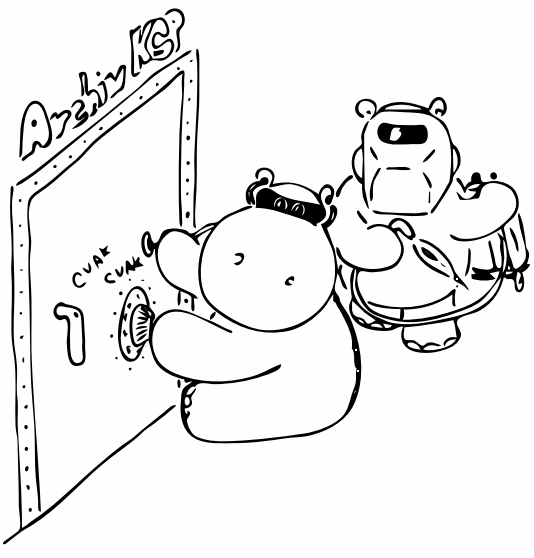
Celý algoritmus slouží zároveň i jako důkaz, že kteroukoliv mřížku velikosti $N = 2^k$ lze pokrýt dílky. Počáteční pozorování pro $k = 0$ je indukčním předpokladem, rozdělení na podmřížky indukčním krokem.

Pokud bychom chtěli algoritmus implementovat (což jsme od vás nepožadovali), je zajímavé se podívat na časovou složitost. Budeme následovat výše uvedený postup a vytvoříme funkci, která vždy položí nový dílek a navíc pro $N > 1$ zavolá rekurzivně čtyřikrát sama sebe. Samotný průběh funkce má konstantní časovou složitost (pouze vypočítáme polohu plného pole), takže zbývá vyřešit, kolikrát se zavolá.

Zkusíme pro změnu přemýšlet odspodu: voláme funkci na každou podmřížku velikosti 1×1, a těch se v mřížce nachází N^2; dále na každou podmřížku velikosti 2×2, těch je celkem $\frac{N^2}{4}$. Dostáváme se tak k součtu řady: $N^2 + \frac{N^2}{2} + \frac{N^2}{4} + \ldots + 1$. K jejímu řešení můžeme využít například kuchařkovou větu (Master Theorem), která nám odpoví, že celková časová složitost je $O(N^2)$.

Program (Python):
`http://ksp.mff.cuni.cz/viz/29-3-5.py`

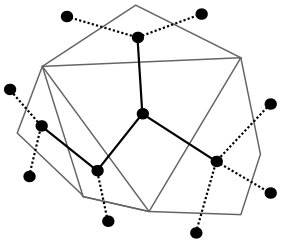
Kuba Maroušek



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

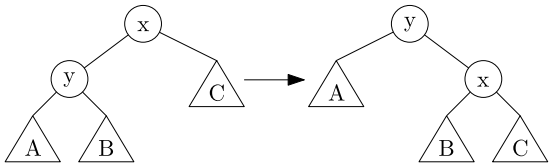
29-3-6 Obrazec pro draka

Pro jednodušší řešení úlohy je dobré převést si mnohoúhelník na něco, s čím se pracuje lépe. Vytvoříme si graf, jehož vrcholy představují trojúhelníky a hrany reprezentují tyče. Vrcholy sousedních trojúhelníků jsou tedy spojené hranou. Ještě se nám bude hodit, když i strany mnohoúhelníku budou hrany a za každou stranou budeme mít také vrchol. Můžeme si všimnout, že tento graf je stromem.

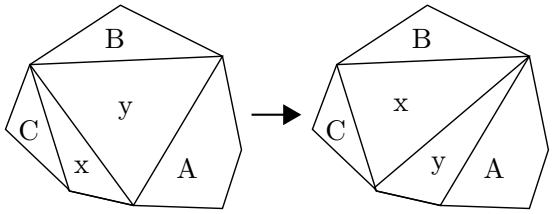


Nyní si můžeme vybrat jeden z vrcholů mimo mnohoúhelník a prohlásit ho za kořen našeho, nyní binárního, stromu. Teď by nás zajímalo, co udělá s naším stromem jedno překlopení tyče. Ukážeme si, že odpovídá operaci stromové rotace.

Rotace je „otočení“ hrany mezi dvěma vrcholy, kde zachováváme pořadí vrcholů a podstromy převěsíme, viz obrázek.



Přesně tohle udělá zvednutí tyče a její umístění napříč:



Počáteční i cílový obrazec převedeme na binární strom, kde za kořen zvolíme ten stejný vrchol (neboli vrchol za stejnou stranou mnohoúhelníku). Teď hledáme, jak převést pomocí rotací jeden na druhý. Ještě je dobré si očíslovat listy – tedy vrcholy za hranami mnohoúhelníku. Aby dva stromy reprezentovaly stejnou triangulaci, tak musí sedět i očíslování listů.

Stromové rotace mají jednu důležitou vlastnost, totiž zachovávají pořadí listů. Nestane se nám tak, že by se pořadí listů nějakým způsobem pomíchalo (což by znamenalo, že by se nám i mnohoúhelník musel nějak překlápět). Zachování této vlastnosti je důležité, protože bychom jinak mohli sice vymyslet způsob, jak přejít od jednoho stromu k druhému, ale neseděly by nám listy, tedy by vlastně vůbec nemuselo jít o tu samou triangulaci.

Nyní už jsme velmi blízko cíle. Připomínáme, že hledáme libovolnou posloupnost rotací, nemusí být nutně nejkratší. Pokud budeme opakovat dostatečně dlouho rotace do jednoho směru, třeba doleva, získáme lineární strom.

Stačí začít u kořene a opakovat rotace, dokud není vpravo jenom list. Poté půjdeme k jeho pravému synovi a budeme to opakovat. V každé rotaci jeden vrchol zařadíme do lineárního stromu a tedy nám to bude trvat $\mathcal{O}(n)$ kroků.

To samé můžeme provést s cílovým stromem. Využijeme toho, že k rotaci vpravo je rotace vlevo inverzní operací. Abychom získali hledanou posloupnost překlopení, můžeme provést rotace stromu počátečního obrazce na lineární strom a pak pozpátku ty, co jsme provedli s cílovým stromem.

Strom sestrojíme v lineárním čase, obě převedení na lineární strom zvládneme $\mathcal{O}(n)$ rotacemi, a protože každá trvá jen konstantní čas, tak celkový čas bude $\mathcal{O}(n)$. Počet rotací bude také $\mathcal{O}(n)$.

Najít nejkratší posloupnost rotací, která převádí jeden binární strom na druhý, v polynomiálním čase zatím bohužel neumíme. Jestli to vůbec jde, je stále otevřený problém. Umožnilo by nám to efektivně spočítat rotační vzdálenost dvou stromů, totiž kolik nejméně rotací je potřeba pro převedení jednoho stromu na jiný. To je hezká metrika – způsob, jak měřit „vzdálenost“ (rozdílnost) dvou stromů.

Jirka Sejkora

29-3-7 Stromoví předci

Úkol 1: Chytřejší značkování

Ke kořeni chceme stoupat z obou vrcholů současně. Jelikož nám asi nedali paralelní počítač, budeme to co nejvěrněji simulovat. Vždycky jeden krok na cestě z prvního vrcholu, pak jeden z druhého, a tak dále. Vrcholy na obou cestách značkujeme a jakmile první vrchol dostane obě značky, je to hledaný společný předchůdce (LCA).

Jak dlouho to trvá? Označme d_1 a d_2 vzdálenosti k LCA. Tento LCA dostane první značku po d_1 krocích, druhou po d_2. Algoritmus se tedy zastaví po $\mathcal{O}(\max(d_1, d_2))$ krocích, což je totéž jako požadovaných $\mathcal{O}(d_1 + d_2)$.

Úkol 2: Minimum svislé cesty

Chceme počítat minimum ohodnocení hran na „svislé“ cestě mezi vrcholem x a jeho předkem p. Předpočítáme si hloubky vrcholů $d(v)$, takže dotaz umíme přeložit na minimum na cestě mezi x a $Pra(x, d(x) - d(p))$.

V zadání jsme ukázali, jak si předpočítat skočky, tedy hrany z v do $Pra(v, 2^i)$, a pak počítat $Pra(w, k)$ složením $\mathcal{O}(\log n)$ skoček. Nyní si pro každou skočku předpočítáme ještě minimum z ohodnocení přeskakovaných hran. To pro každou zvládneme v konstantním čase složením dvou už spočítaných minim. A při skládání $Pra(v, 2^k)$ ze skoček rovnou složíme i příslušná minima.

Předvýpočet trvá $\mathcal{O}(n \log n)$, pak odpovídáme v $\mathcal{O}(\log n)$.

Úkol 3: Součet svislé cesty

Součet je mnohem jednodušší. Spočítáme si analogii prefixových součtů, tedy součty $S(v)$ všech hran z kořene do v. Součet na cestě mezi x a jeho předkem p pak je prostě $S(x) - S(p)$. Předvýpočet trvá $\mathcal{O}(n)$, na dotazy odpovídáme v $\mathcal{O}(1)$.

Úkol 4: Minimum obecné cesty

Minimum nebo součet na obecné cestě mezi x a y spočteme tak, že nejdříve nalezneme $\ell = lca(x, y)$. Pokud je $x = \ell$ nebo $y = \ell$, cesta je svislá a jsme hotovi. V opačném případě cestu rozložíme na dvě svislé cesty: z x do ℓ a z ℓ do y, pro které už umíme odpovědět.

Úkol 5: Součet pomocí ET-posloupnosti

Dostaneme ET-posloupnost, do níž jsme při průchodu hranou směrem dolů napsali její ohodnocení, a při návratu nahoru minus ohodnocení. Chceme počítat součty na svislých cestách, opět označíme x nižší vrchol cesty a p ten vyšší.

Nejprve dokážeme, že součet všech čísel mezi dvěma výskyty téhož vrcholu v v ET-posloupnosti je nulový. Určitě to stačí dokázat pro „sousední“ výskyty, tedy takové, mezi nimiž jsme v nenavštívili. Nechme DFS běžet tak dlouho, než dospěje do prvního z našich dvou výskytů vrcholu v. Pak bude pokračovat do některého ze synů vrcholu v, načež proleze celý podstrom pod tímto synem, a nakonec se vrátí zpět do v, což bude druhý z výskytů. Kdykoliv při tomto průchodu prošel po nějaké hraně dolů, vrátil se po ní pak nahoru, takže k celkovému součtu tato hrana přispěje nulou.

Nyní dokážeme, že součet všech čísel mezi jakýmkoli výskytem vrcholu p a jakýmkoli výskytem vrcholu x je roven součtu cesty mezi x a p. Vzhledem k předchozímu odstavci si můžeme vybrat konkrétní výskyty: pro p si vybereme ten, z nějž odejdeme hranou vedoucí směrem k x; pro x zvolíme první výskyt.

Uvažujme, co DFS provede mezi těmito dvěma výskyty. Určitě prošlo po cestě z p do x. Všechny podstromy odpojující se od této cesty doleva, kompletně prošlo, takže celkem přispěly nulou. Podstromy odpojující se vpravo vůbec nenavštívilo. Nenulou tedy přispěly pouze hrany na cestě.

K odpovídání na daný typ dotazů tedy stačí předpočítat prefixové součty pro ohodnocenou ET-posloupnost, a pamatovat si pro každý vrchol jeho libovolný výskyt. To zvládneme v lineárním čase, na dotazy pak odpovídáme odečtením dvou prefixových součtů, tedy v konstantním čase.

Úkol 6: Syn v zadaném směru

Dostaneme vrchol x a jeho předchůdce p. Chceme najít toho ze synů vrcholu p, který leží na cestě z p do x. Použijeme podobný trik jako pro výpočet LCA. ET-posloupnost ohodnotíme hloubkami a budeme hledat vrchol s minimální hloubkou ležící mezi libovolným výskytem vrcholu x a posledním výskytem vrcholu p.

Z toho přímo nic nezjistíme: minimum se evidentně nabývá pro vrchol p. Ale pokud v zadaném intervalu nalezneme *nejlevější* minimum, je to ten z výskytů vrcholu p, do nějž jsme se z x vrátili. Těsně před ním v posloupnosti leží hledaný syn.

Stačí nám tedy vylepšit strukturu pro intervalová minima, aby vždy našla nejlevější výskyt minima v intervalu. To se dá zařídit třeba tak, že do ET-posloupnosti pro i-tý výskyt vrcholu v místo hloubky $d(v)$ zapíšeme uspořádanou dvojici $(d(v), i)$ a dvojice budeme porovnávat lexikograficky. Nebo můžeme dvojici zakódovat do přirozeného čísla $d(v) \cdot n + i$.

Takto upravená struktura bude stejně rychlá jako ta původní, takže s předvýpočtem v $\mathcal{O}(n \log n)$ dokážeme odpovídat v konstantním čase.

Martin „Medvěd“ Mareš

Výsledková listina třetí série dvacátého devátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*H3-1*	*H3-2*	*H3-3*	*H3-4*	*H3-5*	*H3-6*	*H3-7*	*série*	*celkem*
0.					8	10	11	11	9	13	15	60,0	180,0
1.	Lukáš Rozsypal	GÚstavníPH	4	6	8	10	5,5		9		11	47,0	140,2
2.	Richard Hladík	GOAMarLaz	4	23	8							8,0	121,6
3.	Tomáš Domes	MendelG_OP	4	4		10		7	8		11	39,1	112,3
4.	Jakub Pelc	G UherBrod	3	8		10	11		9		15	45,0	100,6
5.	Pavel Turek	GTomkovaOL	4	7	8	10	6		9		12	47,6	99,8
6.	Roman Bujdák	G JM Galanta	3	3	8	10	4		7			31,2	99,5
7.	Peter Grajcar	GMetodovaBA	3	3	2	10	3		6			27,1	92,7
8.	Rajmund Hruška	GPošKošice	4	2	8	10			9			27,0	70,0
9.	Pavel Turinský	G Brandýs	4	12	8	10			9		2	28,4	67,4
10.	Filip Geib	G MMH LM	3	5		6			5			14,5	66,6
11.	Jonáš Fiala	GJungmanLT	4	7								0,0	56,0
12.	Martin Picek	GJirsíkaČB	2	2	8							8,0	55,0
13.	Martin Kurečka	GJarošeBO	3	1	8	10			9	8	15	53,3	53,3
14.	Jakub Pintera	SPŠ Prosek	4	2	8							8,0	51,4
15.	Miroslav Hrabal	GTomkovaOL	3	4	8	10						18,0	43,6
16.	Matouš Bílek	GJŠkodyPŘ	2	1		10	6	6			10	41,0	41,0
17.	Matouš Mařík	G_Krumlov	4	1								0,0	40,7
18.	Lukáš Caha	GZborovPH	3	2								0,0	38,7
19.	Kateřina Čížková	G_Rokycany	3	2	8	8			8			25,8	34,6
20.	Jan Kaifer	GKepleraPH	1	5	8	10	6					24,0	34,5
21.	Tomáš Raunig	GHlu	2	2								0,0	34,3
22.	František Kmječ	G Brandýs	1	4								0,0	33,3
23.	Kryštof Mitka	ZŠUniverzum	0	3	2	10						13,8	31,2
24.	František Deckert	GOpatovPHA	4	1	8	10	11					29,0	29,0

	řešitel	*škola*	*ročník*	*sérií*	*H3-1*	*H3-2*	*H3-3*	*H3-4*	*H3-5*	*H3-6*	*H3-7*	*série*	*celkem*
25.	Filip Masár	PiarGNitra	3	2								0,0	27,4
26.	Petr Gebauer	GMělník	3	2								0,0	26,8
27.	Michal Kodad	SPŠ_Smíchov	1	6								0,0	26,5
28.	Jiří Löffelmann	GLitoměřPH	3	6	8	10						18,0	25,9
29.	Václav Pavlíček	SPSE_Pard	1	6								0,0	25,5
30.	Ondřej Gonzor	G Brandýs	0	2	2	1						4,8	23,6
31.	Anna Řechtáčková	GJarošeBO	4	2								0,0	22,7
32.	Kristián Jacik	GSRandyJN	4	1								0,0	22,6
33.	Ondřej Krsička	GJarošeBO	1	2								0,0	22,0
34.	Stanislav Lukeš	GPísnickáPH	4	13		10						10,0	21,9
35.	Anna Hollmannová	GSRandyJN	0	3			1	1				2,8	21,5
36.	Daniel Skýpala	GTomkovaOL	−1	2			6					7,1	19,6
37.	Radek Olšák	MensaG	2	1								0,0	18,4
38.	Jindřich Dítě	VOSPŠŽďár	1	2						1		1,6	17,2
39.	Přemysl Šťastný	GŽamberk	4	15	8							8,0	15,6
40.	Ondřej Cach	SPSE_Pard	1	1								0,0	15,4
41.–42.	Vojtěch Hudec	G_ČTřebová	3	3								0,0	12,1
	Josef Polášek	GKepleraPH	1	1	8	1	1					12,1	12,1
43.	Vojtěch Lengál	GZborovPH	3	1								0,0	11,0
44.	Dalibor Kramář	G BO-Řeč	2	1	0				8			8,7	8,7
45.–46.	Adam Dřínek	GNAlejíPH	3	1								0,0	8,0
	Jakub Suchánek	GOpatovPHA	3	3	8							8,0	8,0
47.	Jan Neumann	GNAlejíPH	3	2								0,0	7,7
48.–49.	Jakub Dobrý	GMikulášPL	3	4								0,0	7,6
	Anna Šebestíková	GČeskáČB	2	2								0,0	7,6
50.	Michael Kozel	GZborovPH	3	1								0,0	7,5
51.	Jan Jeníček	GNAlejíPH	1	1								0,0	7,4
52.	Jakub Jirkal	GJungmanLT	2	1								0,0	7,2
53.	Jakub Spišák	G VBN Prie	4	1								0,0	7,0
54.–55.	Erik Kučák	GHorMichal	4	1								0,0	6,7
	Martin Miller	GVoděraPH	3	1								0,0	6,7
56.	Michal Töpfer	G DrJPekMB	4	11								0,0	6,6
57.	Eliška Vlčinská	GHladnov	2	2								0,0	6,3
58.	Václav Šraier	GČeskoliPH	4	10			6					5,7	5,7
59.	Jan Bíl	GDašickáPA	4	1	2							4,0	4,0
60.	Jonáš Havelka	GJírovcČB	1	2								0,0	2,2