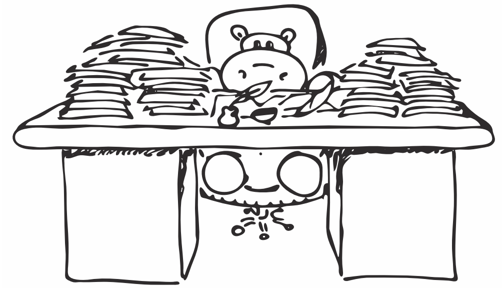
Milí řešitelé!

Jistě máte spoustu práce se získáváním těch správných známek na pololetní vysvědčení. Ale nebojte, je tu i vaše KSP s novou sérií pohodových úložek na příjemné odpočinutí si od školy :-)



Termín odeslání Vašich řešení čtvrté série jest 17. března 2008. Řešení můžete odevzdávat elektronicky na *http://ksp.mff.cuni.cz/submit/*, nebo klasickou poštou na známou adresu:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
Praha 1, 118 00

Aktuální informace naleznete na stránkách *http://ksp.mff.cuni.cz/*, diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz.*

Čtvrtá série dvacátého ročníku KSP

Ráno opět vyšlo slunce. Ne snad proto, že by muselo, ale prostě už bylo tak zvyklé. Opatrně vystrčilo pár paprsků a začalo vymetat zbytky tmy ze Škytánie. Jeden z paprsků polechtal i spícího kocoura Felixe. Kocour dlouze zívl, protáhl se, a přitom nechtě vrazil tlapku přímo do obličeje spícího havrana.

„Krá! Krá?!“ ozval se Kiri popuzeně, čímž vzbudil zbytek družiny.

„To musíš dělat takový rámus?!“ obořil se na něj mág, sotva si protřel oči. „Jestli mě ještě jednou takhle probudíš, proměním tě v žížalu!“

Havran se zatvářil provinile a na svoji omluvu zakrákal: „Never-r mor-re!“

„Vildo, připrav snídani! Chci vyrazit co nejdřív!“ zavelel mág a vstal. Zadíval se do dálky, kde se tyčila Lávová hora. „Máme před sebou ještě kus cesty. . . “

Sestoupili z pahorku, na kterém nocovali a vklouzli do řídkého lesíku. Cesta ubíhala pomalu a les nenápadně houstl. Po několika dnech už nebylo dál možné cestovat s koněm. Stromy rostly příliš blízko u sebe, všude samé houští a mlází. Přeložili všechny věci na Vildova záda a koně poslali domů.

Následujícího dne večer je přepadla ošklivá bouře. Silný vítr kymácel stromy a hustě pršelo. Každou chvíli oblohu prořízl blesk a promptně vyrobil další hromádku palivového dříví.

„Musíme najít nějaký úkryt!“ křičel mág do ohlušujícího větru a ze všech sil se snažil udržet na nohou.

„Myslím, že jsme před chvílí procházeli kolem jeskyně! Můžeme se zkusit schovat tam!“ zahulákal Vilda a rukou chytil Kiriho, který začal prohrávat svůj boj se vzdušnými víry.

Z posledních sil se doplahočili do jeskyně. Sedli si na kamenné podloží, aby popadli dech, a jak už to tak v příbězích bývá, sotva se posadili, oslepil je záblesk doprovázený obrovskou ránou. Vysoká borovice, která stála kousek od vchodu do jeskyně, byla rozpůlená ve dví a z jejího středu se lehce kouřilo.

„Předpokládám, že teď bych měl říct něco jako: 'To bylo o fous!'“ dodal sarkasticky Felix. „Proč jsme sakra nezůstali doma?!“

Vilda začal vybalovat věci a chystal se rozdělat oheň. Vyndal z batohu troud, jehož čvachtavé plesknutí o kámen oznamovalo všem, že rozdělat oheň bude přece jen trochu těžší, než by se snad mohlo zdát. Mág si povzdechl a seslal doprostřed jeskyně kouzelný plamen.

„To by mělo na chvíli stačit,“ prohlásil spokojeně a šel se usušit.

Když se trochu zahřáli, všimli si, že z jeskyně vede několik chodeb. Navíc stěny byly popsány zvláštními značkami. Většina značek byla vodorovná nebo svislá. Mág si je se zájmem prohlížel a pak se ťukl do čela.

„To je přece písmo lesních druidů!“ zvolal vítězoslavně a začal se přehrabovat ve svém vaku. Vytáhl tlustou knihu. Kniha byla opravdu. . . tlustá. Dokonce o poznání tlustší, než obvykle. Mág se na ni podezřívavě podíval. Z knihy ukápla kapka vody. . .

Po hezky dlouhé chvilce strávené rozlepováním listů a sušením nacucaného papíru nalezl mág konečně tu správnou stranu a začal s překladem značek na zdi.

20-4-1 Druidí nápisy — 10 bodů

Mág se s problémem vypořádá jako vždy po svém, ale my obyčejní smrtelníci, kteří neovládáme magii (nebo alespoň ne tak dobře jako Temný mág), si zadání trochu zformalizujeme. Druidí text je zadán jako posloupnost svislých a vodorovných značek bez jakýchkoli mezer, takže si jej lze představit jako posloupnost bitů. K dispozici máme překladovou tabulku, která každému písmenu přiřazuje sekvenci bitů (můžete předpokládat, že všechna písmena jsou reprezentována sekvencemi s maximální délkou řekněme 5 bitů). Dále máme slovník všech možných slov, která se mohou v textu vyskytnout.

Chtěli bychom text přeložit, avšak problém je v tom, že překlad může být mnohoznačný. Naším cílem tedy bude najít celkem K nejkratších možných překladů textu. Délka překladu je udávána počtem slov (tj. chceme překlady s nejméně slovy). Číslo K není konstanta, takže ho nezapomeňte zahrnout do odhadu časové (případně i paměťové) složitosti.

Poznámka autora: Pokud vám tato úloha připomíná úlohu 20-2-3 (Morseovkabezoddělovačů), tak tato podobnost není čistě náhodná. Jen jsme nechtěli, aby se nám v příběhu znovu opakovala morseovka. V této úloze máte téměř stejný úkol, ale nyní musíte najít K nejkratších řešení (ne pouze jedno). Necítíte-li se ve své kůži při pomyšlení na druidy, klidně pracujte s morseovkou.

„Výborně! Už jsem to rozluštil,“ zajásal mág. Všichni ostatní se k němu seběhli a tázavě se zadívali na stěnu. „Zkusím to volně přeložit do naší řeči,“ oznámil a nakrabatil čelo při usilovném soustředění.

„Vstoupili jste – na posvátnou půdu – druidů. Za vaši – troufalost. . . éé. . . zahnete? To asi nebude ono. Že by 'zvadnete'?“ Mág nespokojeně kroutil hlavou a chaoticky listoval knihou. „Néé, už to mám – 'zahynete'! Za vaši troufalost zahynete!“ rozzářil se spokojeně, sklapl knihu a pohladil si plnovous s výrazem triumfu na tváři. Ostatní na něj zírali v němém úžasu. Nastala trapná chvíle ticha. Mágův úsměv se pomalu přeskládal do výrazu panické hrůzy.

„Myslíte, že bychom měli. . . ,“ zašeptal mág a kývl hlavou směrem k východu z jeskyně.

„Krá!“ přitakal Kiri za všechny přítomné.

Rychle naházeli věci do vaků a mág lusknutím prstů uhasil oheň. Venku stále ještě zuřila bouřka, ale nic jiného jim nezbývalo. Vykročili do hustého deště. K jejich nemilému překvapení stál okolo vchodu do jeskyně početný hlouček postav. Všechny byly oblečeny do sněhobílých rouch. Tohle zvláštní oblečení zřejmě nesledovalo předpověď počasí, protože navzdory provazům vody padajícím z nebe bylo dokonale suché. Jedna z postav pozvedla ruku a bouře v jediném okamžiku ustala.

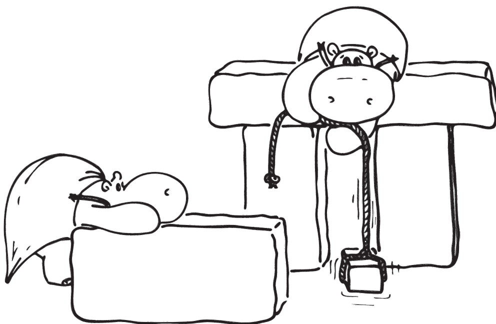
„A do kočičince. . . ,“ ulevil si Felix.

Druidi je vedli lesem dobrou hodinu a každou chvíli měnili směr. Dorazili na rozlehlou louku. Uprostřed stála rozestavěná jakási kamenná konstrukce a kolem ní se hemžili další druidi. Nedaleko bizardní stavby stál statný dub, pod kterým byly rozloženy stoly s nejrůznějším nářadím a nákresy. Došli až k dubu a tam se zastavili. Zanedlouho k nim přispěchal starší druid. Roucho měl úplně stejné jako ostatní, ale i přesto z něho na první pohled sálala autorita. Cestou k nim se několikrát zastavil a rozdal několik rozkazů.

„Zdravím vás,“ promluvil k nim neutrálním tónem. „Jmenuji se Rozmarýn a jsem tady Velkým druidem. Už vás nějakou dobu sledujeme. To víte, když Temný mág opustí Temný hvozd, asi má něco za lubem a je potřeba na něho dávat pozor,“ dodal s lehkým úsměvem. Mág se nesmál. „Každopádně bych chtěl říci, že kdybyste byli kdokoli jiný, nedošli byste živí ani do té jeskyně. Ale než vás propustíme a vyprovodíme z našeho území, budeme po vás chtít malou výpomoc. . . “

20-4-2 Stonehedge — 8 bodů

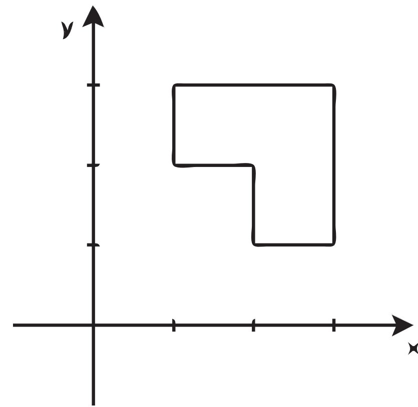
Druidi staví Stonehedge, což by se dalo přeložit jako „Kamenný plot“. Ovšem není to obyčejný plot – je to spíše brána. A přímo do jiného světa. Do technických detailů nebudeme příliš zabíhat, protože je pořádně nepochopil ani mág po Rozmarýnově vyčerpávajícím výkladu. Podstatné je, že druidi mají seznam rohů kamenů, které je potřeba vytesat a správně umístit, a potřebují z nich vytvořit stavební plán.



Každý kámen má pravoúhlou základnu. Tedy jeho půdorys je mnohoúhelník, který má každou stranu rovnoběžnou s některou ze světových stran (v kartézské rovině bychom řekli rovnoběžnou s osami x nebo y). Půdorys kamene je ovšem popsán pouze souřadnicemi rohů (tj. vrcholy polygonu). Záznamy jsou rozházené, takže souřadnice rohů jsou zapsány v náhodném pořadí. Naštěstí ale víme, že kameny v sobě nemají díry (tzn. všechny rohy leží na obvodu kamene) a jejich součadnice jsou celočíselné.

Vašim úkolem je vzít záznamy o každém kameni a zjistit, jaký bude mít půdorys. Výsledkem vašeho snažení by měla být pro každý kámen uspořádaná posloupnost vrcholů (rohů). Jako první můžete uvést libovolný roh a všechny ostatní vypíšete v pořadí, v jakém se vyskytují na obvodu kamene po směru hodinových ručiček.

Příklad:



Máme jeden kámen s vrcholy $[1, 3]$, $[2, 2]$, $[1, 2]$, $[3, 1]$, $[3, 3]$, $[2, 1]$.

Začneme např. v prvním uvedeném bodě $[1, 3]$. Dále pokračujeme po směru hodinových ručiček: $[3, 3]$, $[3, 1]$, $[2, 1]$, $[2, 2]$, $[1, 2]$.

Bylo ráno. Naši hrdinové v klidu podřimovali, až na mága, který stál nad velkým kusem papíru a občas do něho něco zakreslil.

„Tak to by měl být poslední kámen,“ řekl, odložil tužku a promnul si unavené oči. Rozmarýn se zadíval do náčrtků a spokojeně pokýval hlavou.

„Děkuji vám. A teď, když mě omluvíte, musím jít dohlédnout na stavbu. Tady bratr Levandule vás doprovodí až na hranice našeho území.“

Mág nakrabatil čelo: „Ehm – víte, že. . . jaksi. . . slovo Levandule je ženského rodu?“ Levandule na něho vrhl ošklivý pohled. Rozmarýn jen pokrčil rameny: „Nikdo není dokonalý.“ A odspěchal za svými povinnostmi.

Cesta ubíhala. . . potichu. Levandule nebyl zrovna velký řečník a až na příležitostné Felixovo remcání se nikdo neodvážil promluvit. Les začal zvolna řídnout, a když se objevila první cesta, Levandule se s nimi mlčky rozloučil.

„Teď by se nám kůň hodil,“ nahodil Vilda smutně, ale nikdo ho nevnímal.

Pokračovali po cestě až do pozdního odpoledne, když se za nimi ozvalo klapání kopyt a zvuk kodrcajícího se povozu. Za chvíli vůz spatřili na vlastní oči. Byl celkem malý, přikrytý plachtou a táhl ho dobře živený hnědák. Na kozlíku seděl trpaslík a spokojeně bafal z dlouhé fajfky. K všeobecnému překvapení se povedené družinky vůbec nelekl a ještě jim kývnul na pozdrav.

„Brý vodpoledníčko,“ zahlaholil vesele a zastavil vůz. „Vypadáte, jakobystejc tady něco tó – hledali, nemejlim se?“

Mág se podíval na Vildu a Vilda zase na mága. „Vám na nás nepřipadá něco v nepořádku?“ zkusil to opatrně mág.

„Ani né. Tadyc kluk má nějakou nezdravou barvičku, ale pár dní na sluníčku by to spravilo. A ten kocour, vypadá nějakto vopelichaně. . . “

„Já vám dám vopelichaně,“ obořil se na něj Felix.

„No né, vonoto i mluví! No to mě pověs za kšandy a nalakuj,“ pokračoval vesele trpaslík. „Tady u druidích zemí – tady už sem viděl lecojs,“ mávl ledabyle rukou. „Ale prominou mi, ešče sem se nepředstavil. Menuju se Tom. Tom Vytrhnul.“

„Já jsem Temný mág, pán Temného hvozdu,“ oznámil mu vznešeně mág.

„Tak mák, jo?“ prohlásil trpaslík s úsměvem. „Mě ešče

žádnej mák nepřechytračil! Na mě si jen tak někdo nepříde. Ale že voni sou ten mák, tak já je zkusím. Dám vám malej rébous, a když ho uhodnou, svezu je, kam budou chtít."

„A když ne?" Zeptal se Vilda, ale mág ho okamžitě zpražil pohledem.

„Když ne, tak mi nechaj tadytoho mluvícího kocoura. Mít takovou atrakci, to by se zlaťáčky enom sypaly," usmál se na něho Tom a popotáhl ze své fajfky.

20-4-3 Mince — 7 bodů

Tom vyndal ze svého vozu malý soudek a na něj umístil 2×2 mince. Pak nechal mága, aby si zavázal oči. Mág neví, jak jsou mince na začátku otočeny a má za úkol otočit je všechny lícem vzhůru. V každém tahu si může mince libovolně osahat a libovolné otočit, avšak po hmatu nepozná, jak je která mince otočena, ani jestli jsou dvě mince otočeny stejně. Po každém tahu mu trpaslík otočí soudkem, aby ho pořádně zmátl. Pokud mág dostane všechny mince lícem vzhůru, trpaslík sám hru ukončí (místo toho, aby opět točil soudkem).

Pomozte mágovi najít vyhrávající strategii. A jako obvykle pamatujte, že mág má naspěch, takže by vaše strategie měla být co nejkratší.

Strategií rozumíme deterministický návod, který dovede mága po konečném počtukroků k vítězství. Nezapomeňte, že počáteční stav je neznámý a vaše strategie musí fungovat vždy (na všechny možné počáteční konfigurace mincí).

Trpaslík zíral na čtyři mince otočené lícem vzhůru a nevěřícně kroutil hlavou: „To nejni možný! Mě ešče žádnej mák nikdy nepřechytračil!"

„A kolik mágů se o to pokoušelo?" nahodil mág a pozvedl obočí.

„No, voni byli první," připustil po chvíli Tom. „No co. Trpasličí slovo je trpasličí slovo. Tak si naskočej. . . "

Cesta na voze uběhla příjemně. Za dva dny stáli nedaleko hory. Země okolo byla teplá a tvářila se výhrůžně. Trpaslík se s nimi spěšně rozloučil a pobídl koně, aby byl co nejdřív pryč od té prokleté hory. Lávová hora sice zlověstně dýmala, ale lávu už hezkých pár let nechrlila. Budeme se muset dostat dovnitř, pomyslel si mág a vykročil po úpatí hory. Jeho družina ho neochotně následovala.

Vyškrábali se sotva sto metrů po úbočí, když mág objevil úzký tunel vedoucí do nitra hory. Byl příliš úzký, aby se jím protáhl člověk, ale na druhou stranu dostatečně široký. . . řekněme pro kocoura. Mág se zamyšleně podíval na Felixe.

„Na to zapomeň! V žádném případě! Ani mě to nenapadne!" vřískal zděšeně Felix.

O chvíli později se už prodíral mezi vlažnými kameny a hlasitě si na všechno stežoval, aby měl jistotu, že ho uslyší i všichni venku. Náhle se chodba začala rozšiřovat a vyústila do obrovské místnosti. Tady není něco v pořádku, pomyslel si Felix a mžoural do tmy. Ve tmě se něco pohnulo. Ale ne! To je přeci. . .

Pokračování (pravděpodobně už bez Felixe) příště.

20-4-4 Skupinky pro chytré — 10 bodů

V tomto díle mágova dobrodružství se bohužel objevily jen tři zajímavé problémy, které jsme vám mohli zadat jako úložky. Sice bychom si mohli vymýšlet a příběh pozměnit tak, abychom do něho vtěsnali další úložku, ale to už by pak nebyl autentický a pravdivý. Na druhou stranu by bylo škoda, kdybychom vás o úložku ochudili, a proto jsme se rozhodli, že přidáme obyčejný (tzn. čistě programátorský) problém, který nemá s Temným mágem vůbec nic společného. Fanouškům Temného mága se tímto omlouváme a slibujeme, že jim vše vynahradíme v další sérii.

Tak tedy k našemu problému. Budeme pracovat se záznamy lidí. Každý člověk má jméno a IQ (přičemž IQ je celé kladné číslo). Z lidí budeme vytvářet skupinky. Každá skupinka má unikátní ID (identifikační číslo), které je opět celé a kladné. Na počátku máme pouze jedinou prázdnou skupinku s $ID = 1$.

Nad skupinkami chceme provozovat operace:

- `INSERT` – Vloží nového člověka.
- `FIND_BEST` – Nalezne člověka s nejvyšším IQ.
- `DELETE_BEST` – Člověk s nejvyšším IQ odchází za kariérou do zahraničí (odstraníme jej).

Výše uvedené operace dostanou vždy ID skupinky, nad kterou mají být provedeny. Žádná z operací nemodifikuje skupinku, ale místo toho vytvoří skupinku novou, ve které budou uloženy výsledky operace. ID nové skupinky bude nejmenší dosud nepoužité číslo. Pochopitelně operace `FIND_BEST` pouze vrací nalezeného člověka, takže nevytvoří novou skupinku. Skupinky nikdy nezanikají, takže je potřeba si je nějakým způsobem udržovat všechny.

Výsledek každé operace musíte oznámit ještě před tím, než začnete zpracovávat operaci další – tj. nesmíte si operace bufferovat a pak jich provést víc najednou.

Vaším úkolem je navrhnout a popsat vhodnou datovou reprezentaci a jak na ní budou probíhat požadované operace.

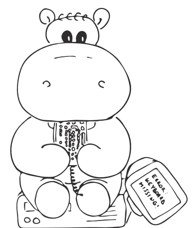
Příklad:

Jak bylo řečeno, na začátku máme jen jednu prázdnou třídu s $ID = 1$. Budeme provádět operace:

- `INSERT("Aleš", IQ=130)` do $ID = 1$ vytvoří skupinku $ID = 2$.
- `INSERT("Petr", IQ=110)` do $ID = 2$ vytvoří skupinku $ID = 3$.
- `INSERT("Jana", IQ=140)` do $ID = 1$ vytvoří skupinku $ID = 4$.
- `FIND_BEST` v $ID = 2$ vrátí `"Aleš"`.
- `FIND_BEST` v $ID = 3$ vrátí také `"Aleš"`.
- `FIND_BEST` v $ID = 4$ vrátí ovšem `"Jana"`.
- `DELETE_BEST` z $ID = 3$ vytvoří skupinku $ID = 5$.
- `FIND_BEST` v $ID = 5$ vrátí `"Petr"`, protože Aleše odstranila předchozí operace.

20-4-5 Roboti na útěku — 15 bodů

Milí řešitelé a řešitelky.



Vánoce skončily a pod stromečkem jsme nalezli krásnou praktickou úložku. A protože nejsme hamouni, rádi se s vámi o ni podělíme. Způsob odevzdávání a všechny ostatní detaily zůstávají stejné jako v minulých sériích. Takže pokud jste zapomněli, jak to v praktické úložce chodí, nebo jste se do řešení KSP zapojili teprve teď, podívejte se na úložku 20-1-5 z první série, kde naleznete potřebné informace.

Zadání:

Gratulujeme, právě jste se stali majiteli dvou zánovních robotů. Bohužel obchodník, který vám je prodal (za extra výhodnou cenu), jaksi zapomněl zmínit, že oba roboti jsou uvězněni v bludišti. A aby toho nebylo málo, tak každý

ve svém. Abyste mohli roboty používat, budete je muset nejprve dostat ven.

Bludiště je reprezentováno čtvercovou mřížkou s očíslovanými políčky, kde [1, 1] je levý-horní (tedy severozápadní) roh. Políčka tohoto bludiště mohou být volně průchozí, nebo na nich může stát zeď. Každý robot má své bludiště i ($i \in \{1, 2\}$). V každém bludišti i se navíc nachází G_i stráží, kde $0 \leq G_i \leq 10$. Stráže se pohybují a roboti se od nich nesmí nechat chytit.

Na začátku každého tahu pošlete svým robotům příkaz (oběma stejný). Příkaz je pouze směr, kterým se má robot pohnout (tzn. sever, jih, východ nebo západ), a oba roboti se posunou o jedno políčko daným směrem. Pokud se robot nemůže pohnout (tj. na cílovém políčku je zeď), zůstane stát na místě. Robot je volný v okamžiku, kdy vyjede ven ze zadaného bludiště, a od té doby jakékoli další příkazy ignoruje.

Stráže se rovněž pohybují na začátku každého tahu – tedy ve stejném okamžiku jako roboti. Každá stráž se pohne o jedno políčko směrem, kterým se právě dívá. Jakmile stráž dorazí na konec své cesty (posune se o jedno políčko méněkrát, než je délka její hlídkovací trasy), udělá čelem vzad a bude se posunovat zpět na políčko, kde začínala. Na počátečním políčku se opět otočí a pokračuje v hlídkování, dokud robot neopustí bludiště.

Hlídkovací trasy stráží jsou navrženy tak, aby žádná stráž nemusela procházet zdí. Trasy několika stráží se mohou protínat, ale jejich pohyb je navržen tak, aby se žádné stráže nikdy nesrazily (tzn. na konci tahu nebudou přebývat na stejném políčku a ani si během tahu vzájemně nevymění políčka). Navíc máte zaručeno, že žádná stráž nebude na začátku obývat stejné políčko jako váš robot.

Stráž chytí robota v případě, že se na konci tahu nachází oba na stejném políčku nebo si během jednoho tahu políčka vzájemně vymění.

Žádné z bludišť nemá rozměry větší než 20×20 a zadaná konfigurace (tzn. bludiště s počáteční pozicí robota a s hlídkovými trasami stráží) je vždy platná (tzn. vyhovuje pravidlům popsaným výše). Vaším úkolem je nalézt nejkratší sekvenci tahů, která dostane oba roboty ven z bludiště, aniž by byli polapeni strážemi. Pokud existuje víc řešení, stačí nalézt jedno libovolné.

Vstup je uložen v souboru `robots.in`. Obě bludiště jsou uložena ve stejném formátu těsně za sebou. Formát jednoho bludiště vypadá následovně:

- Na prvním řádku se nachází dvě čísla R_i a C_i oddělená mezerou, která představují počet řádků (R_i) a sloupců (C_i) bludiště i.
- Následujících R_i řádků obsahuje vždy C_i znaků, kde každý znak popisuje jedno políčko bludiště. Znak `#` představuje zeď, znak `.` volně průchozí pole a znak `X` počáteční pozici robota v bludišti.
- Za mapou bludiště následuje řádek s jediným číslem G_i, které udává počet stráží v bludišti. Připomeňme si, že $0 \leq G_i \leq 10$.
- Následuje G_i řádků. Každý řádek obsahuje tři celá čísla a jeden znak oddělené mezerou a popisuje právě jednu stráž. První dvě čísla určují řádek a sloupec počátečního políčka stráže. Třetí číslo určuje délku hlídkovací trasy (v rozsahu 2…4). A konečně poslední písmeno určuje směr, kterým se stráž na začátku dívá (a kterým se také začne pohybovat). Směr je dán prvním písmenem světové strany v anglickém jazyce, tedy: `N`, `S`, `E` nebo `W` (pro **N**orth, **S**outh, **E**ast a **W**est).

Výstup je očekáván do souboru `robots.out`. Na prvním řádku bude číslo K ($K \leq 10000$), které určuje, kolik tahů bude potřeba, abychom dostali roboty z bludišť. Následujících K řádků obsahuje jednotlivé příkazy v pořadí, v jakém se mají vykonat. Příkazy jsou zapsány písmeny `N`, `S`, `E` a `W`, která určují směr pohybu robotů. Obdobně jako u vstupu jsou tato písmena zkratkami za světové strany v anglickém jazyce. Pokud neexistuje korektní sekvence příkazů, která by dostala roboty z bludiště, pak bude výstupní soubor obsahovat jediný řádek s číslem -1.

Příklad:

Vstup `robots.in`:

```
5 4
####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
####
#...
#X.#
####
0
```

Výstup `robots.out`:

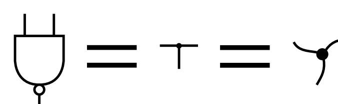
```
8
E
N
E
S
S
S
E
S
```

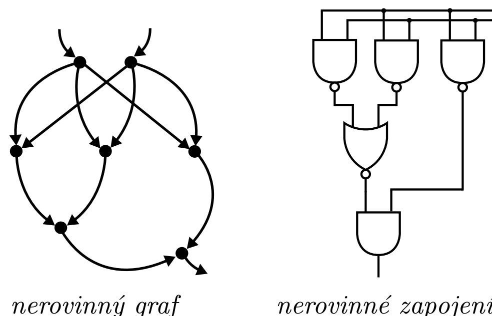
20-4-6 Hrady, hrádky, hradla — 12 bodů

V dnešním díle si budeme povídat o rovinnosti a co to taková rovinnost znamená. Ti z vás, kteří znají pojem rovinnost z teorie grafů mohou definici rovinného zapojení přeskočit.

Nadefinujeme nejdříve pojem rovinný graf, neboť pro rovinný obvod platí definice analogicky. Definice zní: Rovinný graf (obvod) je takový graf (obvod), který má rovinné nakreslení. Rovinné nakreslení je takové nakreslení, při kterém se žádné dvě hrany (vodiče), při průmětu do roviny, nekříží. V jedné větě: Rovinný graf (obvod) lze nakreslit do roviny bez křížení hran (vodičů).

Na obrázcích je zapojení, o kterém vám prozradím, že je nerovinné, a graf, na který lze převést. Graf vyrobíme tak, že hradla zjednodušíme na vrcholy se třemi hranami. Stejně tak místa, kde spojujeme vodiče. Pro názornost ponechávám hranám orientaci, i když pro určení rovinnosti je veskrze zbytečná.





nerovinný graf *nerovinné zapojení*

Dokázat, že je graf rovinný není jednoduché, lze to například provést tak, že se najde jeho rovinné nakreslení. Opačná věta se dokazuje ještě o krapítek složitěji, zájemce o tuto problematiku nechť se poučí například v knize "Kapitoly z diskrétní matematiky" od pánů Matouška a Nešetřila.

V praxi je tato problematika velmi důležitá, neboť pro spojování elektronických součástek se používají plošné spoje. Plošný spoj je destička ze dvou materiálů, jako podklad se používá sklotextit, což je destička ze skleněných, nebo podobných vláken slepená plnivem. Tato vrstva je vrstva nosná. Mnohem zajímavější je ovšem vrstva druhá, ta je z mědi. A právě v této měděné vrstvě se leptají cestičky, které spojují obvody. Díky této technologii výroby je důležité minimalizovat křížení, které se pak musí řešit drátovými propojkami. I když ve skutečnosti se nám do zapojení přimíchají navíc cestičky napájecí a problém bude ještě o něco složitější.

Nyní přichází vaše příležitost, máme pro vás tyto dva úkoly.

1) Vymyslete zapojení, které umí "prohodit" dva signály. Tj. pokud obvod obklopíme kružnicí a půjdeme po průsečících s vstupy a výstupy, dostaneme z původní posloupnosti vstup A, vstup B, výstup B, výstup A posloupnost vstup A, vstup B, výstup A, výstup B. Aby zadání nebylo triviální a v návaznosti na dnešní téma musí takové zapojení být rovinné. [5 bodů]

2) Dokažte, že libovolné zapojení lze převést na rovinné za cenu zpomalení. Ale i tak se snažte toto zpomalení minimalizovat. [7 bodů]

Recepty z programátorské kuchařky

Dnešní povídání o algoritmech a datových strukturách se bude zabývat jedním z nejznámějších algoritmů: Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. K tomu (a nejenom k tomu) se nám bude hodit šikovná datová strukturka zvaná halda, tak si předvedeme nejdříve ji.

Halda

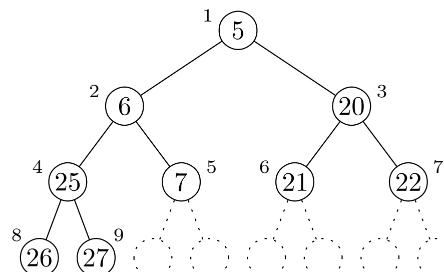
Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných prvků, na kterých máme definováno uspořádání, tj. umíme pro každou dvojici prvků říci, který z nich je menší). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku, nalezení nejmenšího prvku a odebrání nejmenšího prvku. My si ukážeme jednoduchou implementaci haldy, která bude při uložení N prvků potřebovat čas $\mathcal{O}(\log N)$ na přidání či odebrání jednoho prvku a $\mathcal{O}(1)$ (tj. konstantní) na zjištění hodnoty nejmenšího prvku.

Naše implementace bude vypadat následovně: Pokud halda obsahuje N prvků, uložíme její prvky do pole na pozice 1 až N. Prvek na pozici k bude mít dva *následníky,* a to prvky na pozicích $2k$ a $2k+1$; samozřejmě, pokud je k velké, a tedy např. $2k+1 > N$, má takový prvek jen jednoho či dokonce žádného následníka. Naopak prvek na pozici $\lfloor k/2 \rfloor$ nazveme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, v tomto jistě rozpoznali způsob, jak v poli uchovávat úplné binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii, prvek č. 1 je kořen stromu).

Prvky haldy však v poli neuchováváme v úplně libovolném pořadí. Chceme, aby platilo, že každý prvek je menší nebo roven všem svým následníkům. Naše halda tedy může vypadat např. takto:

1	2	3	4	5	6	7	8	9
5	6	20	25	7	21	22	26	27

Tomu odpovídá tento strom:



Z toho, co jsme si právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 1, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. Ještě prozradíme, jak lze prvky do haldy rychle přidávat a odebírat:

Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba x, přidáme na konec pole, tj. na pozici s indexem N. Nyní x porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, je vše v pořádku a jsme hotovi. V opačném případě x s jeho předchůdcem prohodíme. Tím jsme problém napravili, ale nyní může být x menší než jeho nový předchůdce. To lze napravit dalším prohozením a tak budeme pokračovat dále, než se buďto dostaneme do situace, kdy už je x větší nebo rovno svému předchůdci, nebo „vybublá“ až do kořene haldy, kde už žádného předchůdce nemá. Protože se v každém kroku pozice, na níž se prvek x právě nachází, zmenší alespoň na polovinu, provedeme dohromady nejvýše $\mathcal{O}(\log N)$ výměn, a tedy spotřebujeme čas $\mathcal{O}(\log N)$.

Odebírání nejmenšího prvku probíhá podobně: Prvek z poslední pozice (tj. z pozice N) přesuneme na pozici 1, tedy místo minima. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, opět je prohodíme (pokud je větší než oba následníci, prohodíme ho s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $\mathcal{O}(\log N)$.

Jako cvičení si rozmyslete, že v čase $\mathcal{O}(\log N)$ lze z haldy smazat dokonce libovolný prvek, pokud si ovšem pamatujeme, kde se v haldě nachází. Také můžeme prvek ponechat a jen změnit jeho hodnotu.

Ještě si předvedeme program:

```
var halda: array[1..MAX] of integer;
    N: integer;             { počet prvků v haldě }

function nejmensi: integer;
begin
  nejmensi:=halda[1]
end;

procedure vloz(prvek: integer);
var i, x: integer;
begin
```

```
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>1) and (halda[i div 2]>halda[i]) do begin
    x:=halda[i div 2];
    halda[i div 2]:=halda[i];
    halda[i]:=x;
    i:=i div 2
  end
end;

procedure smaz_nejmensi;
var i, j, x: integer;
begin
  halda[1]:=halda[N];
  N:=N-1; i:=1;
  while 2*i<=N do begin
    j:=i;
    if halda[j]>halda[2*i] then j:=2*i;
    if (2*i+1<=N) and (halda[j]>halda[2*i+1]) then j:=2*i+1;
    if i=j then break;
    x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
    i:=j
  end
end;
```

HeapSort

Když už máme k dispozici haldu, můžeme pomocí ní například snadno třídit čísla. Máme-li N čísel, která chceme setřídit, vytvoříme si z nich nejprve haldu o N prvcích (například postupným vkládáním do prázdné haldy), načež z ní budeme postupně N-krát odebírat nejmenší prvek. Tím získáme prvky původního pole v rostoucím pořadí. Celkově provedeme N vložení, N nalezení minima a N smazání. To vše dohromady stihneme v čase $\mathcal{O}(N \log N)$.

Než si ukážeme program, přidáme ještě dva triky, které nám implementaci značně usnadní. Předně si vše uložíme do jednoho pole – to bude při plnění haldy obsahovat na svém začátku haldu a na konci zbytek vstupního pole, přitom zbytek pole se bude postupně zmenšovat a uvolňovat tak místo haldě; naopak v druhé polovině algoritmu budeme zmenšovat haldu a do volného prostoru ukládat setříděné prvky. K tomu se nám bude hodit získávat prvky v opačném pořadí, proto si upravíme haldu tak, aby udržovala nikoliv minimum, nýbrž maximum.

Druhý trik spočívá v tom, že nebudeme haldu vytvářet postupným vkládáním, nýbrž naopak zabubláváním prvků (podobným, jako děláme při mazání minima) od konce. Všimněte si, že takto také získáme správné nerovnosti mezi prvky a jejich následníky, a dokonce tak zvládneme celou haldu vytvořit v lineárním čase [proč to tak je, si zkuste dokázat sami, stačí si uvědomit, kolikrát zabubláváme které prvky]. Zbytek třídění bohužel nadále zůstává $\mathcal{O}(N \log N)$.

Tomuto algoritmu se obvykle říká *HeapSort* (čili třídění haldou) a je jedním z mála známých rychlých třídících algoritmů, které nepotřebují pomocnou paměť.

```
type Pole = array[1..MAXN] of Integer;

procedure HeapSort(var A: Pole);
var i, x: integer;
   procedure bublej(m, i: integer); { "zabublání" prvku }
   { m je velikost haldy, i je index zabublávaného prvku }
   var j, x: integer;
   begin
      while 2*i<=m do begin
         j:=2*i;
         if (j<m) and (A[j+1]>A[j]) then j:=j+1;
         if A[i]>=A[j] then break;
         x:=A[i]; A[i]:=A[j]; A[j]:=x;
         i:=j;
      end;
   end;
begin
   for i:=N div 2 downto 1 do bublej(N,i); { bublej }
   for i:=N downto 2 do begin { vybírej maximum }
      x:=A[1]; A[1]:=A[i]; A[i]:=x;
      bublej(i-1, 1);
   end;
end;
```

Dijkstrův algoritmus

Nyní již konečně k slíbenému *Dijkstrovu algoritmu.* Tento algoritmus dostane orientovaný graf s hranami ohodnocenými nezápornými čísly (viz kuchařka v minulé sérii) a nalezne v něm nejkratší cestu mezi dvěma zadanými vrcholy. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *definitivní.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který ještě není definitivní, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za definitivní. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší, než zatím nalezená cesta z v_0 do v, a je-li tomu tak, upravíme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v. Celý algoritmus skončí, pokud jsou už všechny vrcholy definitivní nebo všechny vrcholy, co nejsou definitivní, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Předtím než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí.

Pro každý z N vrcholů si délku dosud nalezené cesty uchováme v poli. Celý algoritmus má nejvýše N kroků, protože v každém kroku nám přibyde jeden definitivní vrchol. Ten vybíráme z jako minimum z délky aktuální cesty přes všechny dosud nedefinitivní vrcholy, kterých je $\mathcal{O}(N)$. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Počet takových změn pro všechny kroky dohromady je pak nejvýše $\mathcal{O}(M)$, kde M je počet hran vstupního grafu. Z toho vyjde časová složitost $\mathcal{O}(N^2 + M)$, čili $\mathcal{O}(N^2)$, jelikož M je nejvýše N^2. Tuto implementaci Dijkstrova algoritmu najdete na konci naší kuchařky.

K uchovávání délek dosud nalezených nejkratších cest můžeme ovšem použít haldu. Ta bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden: Nalezneme a smažeme nejmenší prvek, to zvládneme v čase $\mathcal{O}(\log N)$, a případně upravíme délky nejkratších cest do sousedů právě zpracovávaného vrcholu. To pro každou hranu trvá rovněž $\mathcal{O}(\log N)$, celkově za všechny hrany tedy $\mathcal{O}(M \log N)$. Z toho vyjde celková časová složitost algoritmu $\mathcal{O}((N + M) \log N)$, a to je pro „řídké“ grafy (tedy grafy s $M \ll N^2$) výrazně lepší.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina definitivních vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0 v_1 \dots v_k v$ takové, že všechny

vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za definitivní. Uvažme nejprve nějaký vrchol v, který je definitivní. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy A bez vrcholu w. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není definitivní. Nechť $v_0 v_1 \ldots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0 v_1, \ldots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus funguje je možné snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Poznámka pro zvídavé: existují i jiné druhy hald, například k-regulární haldy, v nichž má každý prvek k následníků (rozmyslete si, jaká je v takové haldě časová složitost operací a jak nastavit k v závislosti na M a N, aby byl Dijkstrův algoritmus co nejrychlejší), nebo tzv. Fibonacciho halda, která dokáže upravit hodnotu prvku v konstantním čase. S tou pak umíme hledat nejkratší cesty v čase $\mathcal{O}(M + N \log N)$.

Dnešní menu Vám servírovali
Dan Kráľ, Martin Mareš a Petr Škoda

Implementace Dijkstrova algoritmu

```
var N: word;                                { počet vrcholů }
    vahy: array[1..MAX, 1..MAX] of integer; { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;        { délky zatím nalezených cest, -1 = nekonečno }
    def: array[1..MAX] of boolean;          { definitivní? }

procedure Dijkstra(odkud: word);
var i, w, v: word;
begin
  for i:=1 to N do begin
    def[i]:=false;  delky[i]:=-1;
  end;
  def[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not def[i] and ((w=0) or (delky[i]<delky[w])) then w:=i;
    if w<>0 then begin
      def[w]:=true;
      for i:=1 to N do
        if (vahy[w][i]<>-1) and (delky[w]+vahy[w][i]<delky[i]) then delky[i]:=delky[w]+vahy[w][i]
    end
  until w=0;
end;
```

Vzorová řešení druhé série dvacátého ročníku KSP

20-2-1 Volba šamana

Nejjednodušší způsob, jak najít šamana, je nejspíš tento: V prvním mezičase mezi údery bubnu každý elf vytvoří zprávu obsahující jeho jméno a věk a sdělí ji svému sousedovi vlevo. V dalších mezičasech elfové pouze přeposílají poslední přijaté zprávy, a to tak dlouho, než jim přijde jejich původní zpráva (poznají ji podle svého jména). To se stane u všech elfů najednou, protože každá zpráva obejde celé kolo elfů za stejnou dobu. Takže skončí všichni současně. Když si každý elf bude ještě pamatovat zprávu nejstaršího dosud známého elfa, bude na konci rituálu znát jméno a věk šamana, protože každá zpráva (včetně té od budoucího šamana) obešla všechny elfy.

Tímto postupem bude celý rituál s N elfy trvat N úderů bubnu. Jak si mnozí řešitelé všimli, je možné rituál dvakrát zrychlit, když budou elfové svoje zprávy posílat oběma směry. V takovém případě stačí, když zpráva obejde půl kola, a rituál bude trvat jenom $N/2$ úderů. Zbývá vyřešit ukončení rituálu. Pokud je počet elfů v kruhu sudý, pozná se konec jednoduše tak, že ke každému elfovi dojde z obou stran stejná zpráva. Pokud je elfů lichý počet, sejdou se obě kopie zprávy jakoby na půl cesty mezi dvěma elfy. Elf v jednom kroku dostane stejnou zprávu, jakou právě odeslal. Elfové sice neví, jestli jich je sudý nebo lichý počet, ale skončí jednoduše tehdy, když nastane jeden z těchto případů.

Petr Kratochvíl

20-2-2 Setříděné stromy

Temný mág by z Vás měl radost. Většina došlých řešení po-

třebovala $\mathcal{O}(\log_2 K)$ porovnání. Ukážeme si zde jedno, které je sice trochu delší na rozbor, ale porovnává stromy opravdu nejméně a dokážeme si to o něm. Nejdříve tři drobná pozorování.

Pokud je délka první resp. druhé řady stromů N_1, resp. N_2 větší než K, můžeme prvky, které jsou v ní více než K-té, zahodit, jelikož určitě nebudou K-té celkově.

Dále K-tý nejvyšší strom existuje právě tehdy když $N_1 + N_2 \geq K$. Pokud tedy $N_1 + N_2 < K$ vypíšeme, že strom neexistuje a jsme hotovi. Zřejmě jsme tohle byli schopni udělat bez porovnávání výšek stromů a tedy lépe to nešlo.

Porobně triviální případ nastane, když jedna z posloupností bude mít nulovou délku. Pak K-tý strom celkově bude samozřejmě K-tý v řadě stromů s nenulovou délkou. Tím jsme odstranili speciální případy a dále budeme pokračovat s posloupnostmi, pro jejichž délky platí $1 \leq N_1 \leq K$, $1 \leq N_2 \leq K$ a $N1 + N2 \geq K$.

Nyní se dostáváme k vlastnímu algoritmu. Ten je založen na metodě půlení intervalů. Bohužel, pokud chceme mít opravdu optimální řešení, tak se nám toto půlení intervalů rozpadne na 3 případy, které sice budou skoro stejné na naprogramování, nicméně, zvláště u třetího, se budou lišit interpretací toho, co které proměnná znamená. Pokud Vás tedy bude zajímat jen idea programu, stačí si přečíst jen první z nich.

Nejdříve případ kdy $N1 = N2 = K$. Tady si budeme udržovat v paměti o kterých stromech Z_1, resp. Z_2 z první, resp. druhé posloupnosti budeme vědět, že jsou celkově méně než K-té v pořadí. Dále budeme mít v paměti délku intervalu D, kde ještě K-tý nejvyšší strom může být (budou v úvahu připadat stromy $Z_1 + 1, Z_1 + 2, \ldots, Z_1 + D$ z první, resp. $Z_2 + 1, Z_2 + 2, \ldots, Z_2 + D$ z druhé řady). Na D je také možno se dívat jako na počet stromů, které jsou celkově nejvýše K-té, ale které jsme zatím ještě nenašli (rozumějme které jsou více než Z_1., resp. Z_2. v první, resp. druhé řadě). Z tohoto úhlu pohledu je zřejmé interpretace součtu $Z_1 + Z_2 + D$, který bude po celou dobu běhu programu roven K. Na počátku zřejmě $Z_1 = Z_2 = 0$ (a nula tady znamená, že i první strom v příslušné posloupnosti může být K-tý nejvyšší) a $D = K$.

Nyní k vlastnímu půlení intervalů. Vezměme z první, resp. druhé řady strom s pořadím $S_1 = Z_1 + \lfloor D/2 \rfloor$, resp. $S_2 = Z_2 + \lfloor D/2 \rfloor$. Tyhle dva stromy budeme muset porovnat. BÚNO nechť je vyšší první z nich. To znamená, že z druhé řady mohou být vyšší než strom S_1 jen stromy $1, 2, \ldots, S_2 - 1$. Nicméně jak víme, tak $S_1 + S_2 - 1 \leq K - 1$ a tedy strom S_1 může být celkově nejvýše $(K-1)$-ní. Tím jsme o něm dokázali, že musí být před tím, který hledáme a tak můžeme prohlásit, že $Z_1 = S_1$. Tím jsme vyloučili další stromy a tak musíme zkrátit i interval D, ve kterém hledáme, konkrétně na $\lceil D/2 \rceil$. Jak se můžeme snadno přesvědčit, bude opět platit $Z_1 + Z_2 + D = K$.

Předchozí odstavec budeme opakovat dokud $D > 1$. Pak budeme mít o $Z_1 + Z_2 = K - 1$ stromech dokázáno, že jsou nejvýše $(K-1)$-ní a tedy K-tý strom v pořadí bude vyšší z dvojice stromů $Z_1 + 1$ a $Z_2 + 1$.

Nyní k důkazu optimality tohoto případu. X dotazy na porovnání výšek stromů můžeme rozlišit mezi 2^X možnostmi (máme jen 2 možné odpovědí - je vyšší první, resp. druhý strom). Rozhodujeme se mezi $2K$ stromy a tak potřebujeme alespoň $\lceil \log_2(2K) \rceil = \lceil (\log_2 K) \rceil + 1$ porovnání. Náš program bude $\lceil \log_2 K \rceil$-krát porovnávat během cyklu a pak bude jedno porovnání na rozhodnutí, zda je K-tý nejvyšší strom v první, resp. druhé řadě.

Druhý uvažovaný případ nastane, když $N_1 = K$ a $N_2 < K$ (nebo obráceně). Tady můžeme vyloučit na začátku stromy $1, 2, \ldots, K - N_2 - 1$ z první řady aniž bychom potřebovali provnávat - v druhé řadě prostě není dost stromů na to, aby byly celkově K-té. Tím máme v první řadě $N_2 + 1$ stromů, které mají naději stát se K-tým nejvyšším, zatímco v druhé řadě je jich jen N_2. Zavedeme si tedy v druhé řadě virtuální $(N_2 + 1)$-ní strom, který bude nižší než jakýkoliv jiný (v programu je tohle implementováno ve funkci porovnávající stromy), čímž délky obou intervalů srovnáme a můžeme použít výše uvedené půlení intervalů, kde ale počáteční podmínky budou $Z_1 = K - N_2 - 1$, $Z_2 = 0$ a $D = N_2 + 1$.

Důkaz optimality bude analogický. Vybíráme z $2N_2 + 1$ stromů, budeme tedy potřebovat $\lceil \log_2(2N_2 + 1) \rceil$ dotazů. Program bude chtít porovnat $\lceil \log_2(N_2 + 1) \rceil$-krát během cyklu a jednou na rozhodnutí, ve které ze zadaných dvou řad hledaný strom je. Tato dvě čísla jsou shodná, ač to na první pohled není vidět. Pro naše N_2 je $\lceil \log_2(2N_2 + 1) \rceil = \lceil \log_2(2N_2 + 2) \rceil = \lceil \log_2(N_2 + 1) \rceil + 1$. První rovnost neplatí obecně, nicméně my víme, že N_2 je přirozené číslo. Tedy $2N_2 + 1$ je číslo liché větší než 2 proto je horní celá část binárního logaritmu stejná jako horní celá část binárního logaritmu čísla o 1 vyššího. (Kdybychom si nakreslili graf funkce $f(x) = \lceil \log_2(x) \rceil$, tak by vypadal jako konstantní funkce až na body, které jsou mocninou dvojky, kde $f(x)$ zvyšuje skokově svou hodnotu o 1. Nicméně mocniny dvojky větší než dva jsou sudé.)

Zbývá posletní případ, kdy $N_1 < K$ a $N_2 < K$. Tady budeme moci i bez porovnávání tvrdit, že stromy $1, 2, \ldots, K - 1 - N_2$ v první, resp. $1, 2, \ldots, K - 1 - N_1$ v druhé řadě budou v celkovém pořadí maximálně $(K-1)$-ní a tedy je můžeme vyloučit rovnou. Analogicky k předchozímu případu bychom tedy dosadili $Z_1 = K - 1 - N_2$, $Z_1 = K - 1 - N_1$ a $D = N_1 + N_2 - K + 2$. Tohle samořejmě vede k řešení s logaritmickým počtem dotazů, bohužel v některých případech bude chtít měřit jednou navíc. Vybíráme totiž z $2 \cdot (N_1 + N_2 - K + 1)$ stromů (a tedy optimální počet porovnání je $\lceil \log_2(2 \cdot (N_1 + N_2 - K + 1)) \rceil$)a přímočaře zobecněný postup použitý v předchozich dvou případech vede na $\lceil \log_2(N_1 + N_2 - K + 2) \rceil + 1$ porovnání.

Vylepšení na optimální počet porovnání je trochu trik. Místo toho, abychom hledali K-tý nejvyšší prvek, budeme hledat $(K+1)$-ní. Zřejmě budeme tedy mít u výše uvedeného algoritmu počáteční podmínky $Z_1 = K - N_1$, $Z_2 = K - N_2$ a $D = N_1 + N_2 - K + 1$. Po skončení cyklu bude platit $Z_1 + Z_2 = K$ víme, že $(K+1)$-ní nejvyšší storm je vyšší ze stromů $Z_1 + 1$ a $Z_2 + 1$. Co si ale můžeme dovolit tvrdit dále je, že K-tý nejvyšší je nižší ze stromů Z_1 a Z_2. Důkaz toho je jednoduchý - víme, že všechny stromy, které jsou nejvýše K-té nejvyšší jsou $1, 2, \ldots Z_1$ v první, resp. $1, 2, \ldots Z_2$ v druhé řadě. K-tý nejvyšší musí tedy i K-tý nejvyšší být mezi nimi a kvůli setřízenosti vstupních posloupností bude na konci jedné z nich.

Tenhle „trikový" postup potřebuje $\lceil \log_2(N_1 + N_2 - K + 1) \rceil$ dodazů během cyklu a jeden dotaz na rozhodnutí se mezi stromy Z_1 a Z_2. To je ale přesně, jak již bylo uvedeno, minimální počet dotazů potřebných k určení K-tého nejvyššího stromu.

Pavel Čížek

20-2-3 Morseovkabezoddělovačů

Telegraficky: `-..-.---..---..-.-.-.--.-....` :-)

Dobrá, tak trochu podrobněji …

První pokus: Kdybychom chtěli zjistit jen to, zda se zadaný řetězec teček a čárek dá rozložit na slova (tedy aniž bychom uvažovali nad tím, jaká možnost je nejkratší), stačilo by vyzkoušet všechna slova, která pasují na začátek řetězce, a pro každou z těchto možností si zavolat tutéž funkci rekurzivně na zbytek řetězce.

To samozřejmě funguje, jenže někdy až exponenciálně pomalu: pokud budeme mít ve slovníku slova e (`.`) a i (`..`) a vstupní řetězec bude obsahovat spoustu teček a nakonec čárku, náš algoritmus postupně vyzkouší všechny možnosti, jak tečky rozložit na e-čka a i-čka, a při každé z nich se zarazí o koncovou čárku a se svěšeným ocasem se vrátí zpět. Spočítat, kolik přesně těch možností pro n teček bude, není úplně jednoduché (mimochodem, je to $(n+1)$-ní Fibonacciho číslo), ale je jich určitě aspoň $2^{n/2}$. To proto, že když si vezmeme nějakou posloupnost $n/2$ znaků složenou libovolně z e-ček a i-ček, bude její morseovkový zápis tvořen nejvýše n tečkami. Všechny takové posloupnosti (a těch je $2^{n/2}$) náš algoritmus určitě vyzkouší a ještě i mnohé další. Nic moc.

Druhý pokus: Použijeme *dynamické programování* (trochu netradičně odzadu, brzy bude jasné, proč). Nechť vstupní řetězec obsahuje n morseovkových značek $z_1, \ldots, z_n$. My svou úlohu zkusíme vyřešit postupně pro všechny jeho konce: Budeme počítat hodnoty b_i, které budou říkat, na kolik nejméně slov se dá rozložit úsek $z_i, \ldots, z_n$, případně nějaké ∞ (tedy chci říci hrozně velké číslo), pokud se úsek rozložit nedá.

Všimneme si, že tato b_i se dají docela snadno spočítat pozpátku: Začneme trochu šalamounsky u b_{n+1} – to je vždy nula, protože prázdnou posloupnost značek můžeme určitě rozložit na prázdnou posloupnost slov. Dále b_n je buďto 1 (to pokud poslední značka odpovídá nějakému jednopísmenkovému slovu ve slovníku) nebo ∞ (pokud ne). Pokud už máme spočítaná $b_{i+1}, \ldots, b_{n+1}$, snadno dopočítáme b_i: Každý rozklad značek od z_i do konce musí začínat nějakým slovem, řekněme délky ℓ značek, a pokračovat rozkladem značek od $z_{i+\ell}$ do konce. Stačí tedy prozkoumat všechna slova, která do vstupu pasují od i-té pozice, a pro každou z nich se podívat, jestli umíme rozložit zbytek (to nám řekne $b_{i+\ell}$). Pokud je možností víc, vybereme si samozřejmě tu s nejmenším počtem slov (a tedy nejmenším $b_{i+\ell}$). Jinými slovy (ehm, znaky…):

$$b_i = 1 + \min\{b_{i+\ell} : z_i, \ldots, z_{i+\ell-1} \text{ je známé slovo}\}.$$

Takto se postupně dopočítáme až k b_1, což je minimální počet slov, ze kterých se dá poskládat celý vstup, a to je skoro řešení naší úlohy. Chybí jen tato slova najít. K tomu stačí, abychom si u každého (konečného) b_i ještě zapamatovali, které bylo to slovo, jež jsme na tomto místě napasovali do textu. Tomu říkejme třeba s_i a jeho délce ℓ_i.

S těmito informacemi už můžeme rekonstruovat celé řešení (pro změnu odpředu). Našli jsme rozklad s b_1 slovy a jeho první slovo je s_1. Zbytek řetězce tedy musí být rozložen na $b_{1+\ell_1}$ slov a první z nich je $s_{1+\ell_1}$. A tak dále.

Časová složitost se nahlédne snadno. Počítáme celkem n hodnot b_i, pro každou z nich zkoušíme nejvýše tolik pasujících slov, kolik je maximální délka L slova ve slovníku (mohlo by sice existovat více slov, která by pan Morse psal stejně, ale z těch nám jistě stačí vyzkoušet jedno jediné). Pokud budeme odvážně předpokládat, že jedno slovo otestujeme v konstantním čase, stojí nás to celkem čas $\mathcal{O}(nL)$ a paměť $\mathcal{O}(n)$ bez slovníku. Zpětný průchod pak zabere pouze čas $\mathcal{O}(n)$ a konstantní paměť.

Jak ale reprezentovat slovník, abychom s ním dokázali pracovat tak rychle, jak jsme slíbili? Pomůže nám datová struktura zvaná *trie.* To je úplně obyčejný strom, do kterého postupně uložíme všechna známá slova v morseovce tak, že v kořeni se budeme rozhodovat podle první značky, v dalším vrcholu podle druhé značky a tak dále. V každém vrcholu si pak zapamatujeme, která slova tam končí (nezapomeňte, že jich může být víc se stejným zápisem). Každé slovo dokážeme do trie přidat v čase lineárním s jeho délkou, celou strukturu tedy vybudujeme lineárně s velikostí slovníku (čili součtem délek všech slov).

Pro libovolnou posloupnost ℓ značek pak samozřejmě umíme v čase $\mathcal{O}(\ell)$ zjistit, jestli ji máme ve slovníku, ale my jsme přeci slíbili $\mathcal{O}(1)$, tak to také dodržíme. Ona totiž slova, která při počítání jednoho b_i hledáme, nejsou úplně libovolná. První z nich je jen značka z_i, druhé je $z_i z_{i+1}$, třetí $z_i z_{i+1} z_{i+2}$ atd. Úplně tedy stačí, když si zapamatujeme, kde jsme při hledání předchozího slova ve stromu skončili, a odtamtud popolezeme o krok doleva nebo doprava podle toho, jestli následuje tečka nebo čárka. To nás pro každé slovo opravdu stojí jen konstantní čas. (Vidíte, tady se nám hodilo, že řetězec procházíme zprava doleva, jinak bychom totiž museli mít ve slovníku slova uložená pozpátku, což íntnagele ínen étsijaz.

Celkově náš program spotřebuje čas $\mathcal{O}(nL+P)$, kde n je délka vstupního morseovkového řetězce, P celková délka slov ve slovníku a L délka nejdelšího slova.

Poznámka: Někteří z vás chytře využili vyhledávací automat k nalezení všech slov pasujících do textu. Tím se řešení za cenu značného prodloužení potenciálně zrychlí na $\mathcal{O}(n+P+V)$, kde V je počet výskytů slovníkových slov v textu. To může být někdy lepší, ale v nejhorším případě může být $V \approx nL$, takže si obecně nepomůžeme.

Špek na závěr: Jak elegantně *převádět jednotlivá písmena do morseovky.* Jistě bychom si mohli nadefinovat pole řetězců s převody jednotlivých písmenek, ale přeci si pěkné řešení nezkazíme takovou obludností. Pomůže dvojková soustava. Místo teček a čárek si představíme nuly a jedničky a přidáme na začátek jedničku (jinak bychom nepoznali `.` od `....`). To je nějaké dvojkové číslo, tak si ho zapíšeme desítkově a až budeme potřebovat tečky a čárky, budeme toto číslo postupně dělit dvěma a zbytky nám povědí dvojkové číslice, čili tečky a čárky. Jen jaksi pozpátku – to ovšem nevadí, tak si ho uložíme obráceně. Radši ukážeme příklad: písmenko `a` se zapisuje jako `.-`. Obrátíme na `-.`, přepíšeme do dvojkové soustavy jako 110, což je šestka.

Martin Mareš

P.S.: Jak jste jistě uhodli, telegrafická zpráva na začátku našeho řešení zněla *„dynamika a trie“.*

20-2-4 Slovíčkaření

Před samotným řešením se zamysleme. Budete mít tisíc ponožek, které jsou úplně stejné, kromě velikosti. Řekněme, že mají celkem čtyři různé velikosti, a vy je máte podle jejich velikosti roztřídit. Pravděpodobně si nikdo z vás nezvolí nějakou ponožku jako medián a nezačne třídit menší vlevo, větší vpravo, ani si jich všech tisíc nerozložíte po podlaze a

nebudete je spojovat do dvojic, pak do čtveřic atd. Troufnu si odhadnout, že si zvolíte čtyři přihrádky, a do nich budete ponožky rozřazovat. Ano, toto je nejjednodušší a také nejrychlejší řešení. Takovému postupu se říká *přihrádkové třídění*, nebo anglicky *Bucketsort*.

Nyní opusťme ponožky a navraťme se k řetězcům(názvům knih) a významu abecedy. Na moment předpokládejme, že řetězce mají všechny jednotkovou délku. A je jich hodně, s úspěchem můžeme očekávat, že jich je o dost více než prvků abecedy. Zřejmě tedy bude nejvýhodnější zvolit si jednotlivé znaky abecedy jako přihrádky a řetězce do nich „nastrkat“. Označme si N počet řetězců a A počet prvků abecedy. Tento postup od nás vyžadoval přístup k přihrádkám v $\mathcal{O}(A)$ a rozřazení řetězců v $\mathcal{O}(N)$.

Zobecněme problém pro řetězce stejných délek, ale delších než jedna. Mnohé by asi napadlo roztřídit řetězce podle prvního písmene, tam kde by to nestačilo, tak podle druhého atd. My si ukážeme něco mazanějšího. Použijeme Bucketsort na poslední písmena řetězců a podle nich je roztřídíme. Potom na předposlední písmena, přičemž u řetězců, které připadnou do stejné přihrádky, zachováme pořadí z předchozího třídění. Tím získáme roztřídění řetězců podle posledních dvou písmen. My se ale nezastavíme u předposledního a budeme pokračovat dále směrem kupředu. Po i-tém kroku budeme mít řetězce utříděné podle posledních i znaků. V momentě, kdy se dostaneme k prvnímu písmenu a tedy i se bude rovnat společné délce řetězců, máme vyhráno. Řetězce jsou lexikograficky utříděny. Právě jste se seznámili s Radixsortem.

Přitvrdíme a povolíme řetězcům mít různé délky. Předpokládejme, že bychom řetězce doplnili na stejnou délku nějakým znakem, který by byl lexikograficky menší než všechny prvky abecedy. Fungovalo by to, ale mohli bychom si ošklivě ublížit na časové i paměťové složitosti - například bychom měli milion řetězců délky jedna a jeden délky milion... Naštěstí můžeme správný výsledek získat jedním drobným úskokem. Nejprve si však označme L jako délku nejdelšího z řetězců a P jako součet délek všech řetězců. Pokud si na začátku všechny řetězce setřídíme podle délky (opět přihrádkově), můžeme pak postupovat tak, že v i-tém kroku třídění budeme pracovat pouze s řetězci délky alespoň $L-i+1$, tedy v prvním kroku pouze s řetězci délky L atd. Provedeme to tak, že v každém kroku roztřídíme do přihrádek nejprve řetězce z minulého třídění a tedy s větší délkou. Potom před ně do přihrádek naskládáme řetězce s délkou právě $L-i+1$, tedy řetězce, které znakem na zpracovávané pozici končí. Pokud jste uvěřili správnosti Radixsortu, pak je vám nyní jasné, že pomocí tohoto triku opravdu třídíme, a navíc jen tam, kde je to třeba.

Podívejme se, jak rychle takové řešení funguje. Museli jsme provést L kroků (třídíme řetězce od poslední k první pozici). V každém z těchto kroků jsme museli projít přihrádky v setříděném pořadí a inicializovat je v $\mathcal{O}(A)$. Tedy tuto část algoritmu provádíme v $\mathcal{O}(L\cdot A)$. Určitě se právě ptáte, kampak se schovalo oněch N řetězců a jak to, že netrvá každý krok třídění ve skutečnosti $\mathcal{O}(A+N)$, protože až s N řetězci v každém kroku manipulujeme. Pokusme se na to podívat jinak. Díky našemu triku každý řetězec třídíme až tehdy, když je to třeba a má dostatečnou délku. Do té doby s ním nepracujeme. Můžeme tedy tvrdit, že s každým řetězcem pracujeme dohromady tolikrát, kolik má znaků. To v sumě pro všechny řetězce ale znamená $\mathcal{O}(P)$. Podobně počáteční setřídění řetězců podle délky je v $\mathcal{O}(L+N)$. Celý algoritmus tak běží v $\mathcal{O}(L\cdot A+P+N)$, tj. $\mathcal{O}(L\cdot A+P)$. V této knížečce naleznete i vzorovou implementaci. Místo abecedy se v ní uvažují celá čísla v intervalu od 0 do $A-1$, což ale nijak nevadí, protože každá abeceda, kde lze porovnávat, musí být převeditelná na tento případ.

Jako vždy existuje ještě o něco zapeklitější, ale o to efektivnější řešení. Uvědomme si, kde uvedené řešení nejvíce „plýtvá“. V každém kroku prochází všechny přihrádky, bez ohledu na to, jestli jsme je použili, nebo ne. Pokud totiž budeme předem znát, které přihrádky jsou použity, můžeme se dívat pouze do nich a také pouze tyto přihrádky inicializovat a posléze i čistit. Je však důležité, abychom tyto přihrádky znali v setříděném pořadí, protože je tak potřebujeme procházet. Nelze ale pro každý krok tuto informaci počítat zvlášť, protože by to vždy trvalo nejméně $\mathcal{O}(A)$ čemuž se snažíme vyhnout. Proto tento výpočet provedeme na začátku, a to naráz, pro všechny pozice. Uvažujme všechny dvojice (písmeno, pozice), které se v řetězcích vyskytují. Tyto dvojice na počátku setřídíme podle písmen. Tím pro každé písmeno budeme vědět, na kterých pozicích se vyskytuje. Lze si tak vytvořit seznam použitých znaků, pro každou pozici od 1 do L, prostě tak, že projdeme písmena ve vzestupném pořadí a za každou pozici, na které se písmeno vyskytne, přidáme toto písmeno na konec seznamu, který této pozici přísluší. A pak už stačí se při třídění řídit touto informací. Dvojic je P, jako znaků. Tedy toto počáteční předpočítání stíháme v $\mathcal{O}(P+A)$. Každý krok algoritmu tak už netrvá $\mathcal{O}(A)$. Nyní se dohromady ve všech krocích použije $\mathcal{O}(P)$ přihrádek, což je citelně lepší. Tedy výsledná časová i paměťová složitost je $\mathcal{O}(N+P+A)$.

Tuto úlohu je možno řešit také pomocí struktury zvané trie, která v základním stavu dává $\mathcal{O}(P\cdot A)$, ale lze ji také vylepšit výše zmíněným trikem na $\mathcal{O}(N+P+A)$.

Josef Pihera

20-2-5 Hroší ohrádka

Milí chovatelé hrošíků, přestože nikdo z vás nedosáhl maximálního počtu bodů, poradili jste si s řešením Ohrádky velmi dobře. Našim účastníkům, kteří úlohu vyřešili, věnujeme k Vánocům nějaké ty body. Pro všechny ostatní tu máme alespoň návod, jak si Hroší ohrádku postavit.

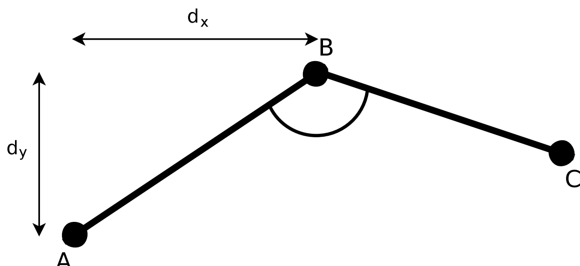
Než začneme se samotným hledáním konvexního obalu, setřídíme si souřadnice kůlů. Primárně budeme třídit podle souřadnice x vzestupně (tedy abychom měli kůly v rovině setříděné zleva doprava). Kůly se stejnou x-ovou souřadnicí setřídíme podle y opět vzestupně (tedy zdola nahoru). Náš algoritmus by s drobnými úpravami fungoval, i kdybychom kůly setřídili jinak. Takto ale dostaneme výsledná data rovnou v pořadí, které vyžaduje zadání.

Nyní budeme procházet kůly a vytvoříme *horní část* konvexního obalu. Analogicky pak projdeme kůly ještě jednou a vytvoříme *spodní část*. Obě části nakonec spojíme a dostaneme výsledný konvexní obal. Zbývá ukázat, jak vytvořit horní část obalu (spodní si ani ukazovat nebudeme – sami si rozmyslete, v čem se bude lišit od horní části).

Na začátku vezmeme první kůl – ten který je nejvíce vlevo (a případně také nejvíce dole). Pak postupně přidáváme další kůly a přitom hlídáme, aby se nám neporušila konvexita. Řekněme, že máme kůly $H_1, H_2, ...H_k$, které tvoří začátek horní části konvexního obalu, a právě chceme přidat kůl I. Podíváme se, zda by přidání tohoto kůlu neporušilo konvexnost již hotové části (matematiku okolo si popíšeme za chvíli). V případě, že by nastal problém – tzn. úsečky

$H_{k-1}H_k$ a H_kI svírají „špatný“ úhel – vyhodíme H_k z horní části a zkusíme I přidat znovu. Pokud žádný problém nenastane, nebo v horní části je zatím jen jeden kůl, vesele přidáme I.

Nyní nám zbývá vypočítat vzoreček, který bude umět rozhodnout, jaký úhel spolu svírají dvě úsečky. Všimněte si, že nás nezajímá hodnota tohoto úhlu, ale pouze zda je větší nebo menší 180. Úsečky si pro lepší přehlednost označíme AB a BC (jsou tedy spojeny v bodě B) a A_x budeme značit x-ovou souřadnici bodu A (analogicky pro y-ové souřadnice).



Při pohledu na obrázek není těžké si rozmyslet, že tyto úsečky jsou konvexní, pokud je směrnice první úsečky menší, než směrnice druhé:

$$\frac{\delta_x(AB)}{\delta_y(AB)} \leq \frac{\delta_x(BC)}{\delta_y(BC)}$$

Za delty si dosadíme souřadnice bodů:

$$\frac{B_x - A_x}{B_y - A_y} \leq \frac{C_x - B_x}{C_y - B_y}$$

Abychom se zbavili dělení a nemuseli ošetřovat zvlášť případ, kdy jmenovatel některého zlomku je nulový, rozšíříme si rovnici na tvar:

$$(B_x - A_x)(C_y - B_y) \leq (B_y - A_y)(C_x - B_x)$$

Z matematického hlediska není výše uvedená úprava úplně vpořádku. Ovšem není těžké si rozmyslet jak se chovají nevhodné případy (tj. $B_y - A_y = 0$ nebo $C_y - B_y = 0$), a že výsledná rovnice je vlastně to, co jsme celou dobu chtěli.

Nalezenou nerovnici stačí použít jako logickou podmínku, která otestuje, zda jsou dvě úsečky konvexní. Pro spodní část konvexního obalu použijeme stejnou nerovnici, ale s opačnou nerovností.

A nyní se podíváme, jak je na tom časová a paměťová složitost. Někteří čtenáři si možná všimli, že výše popsaný algoritmus vypadá jako backtracking a backtracking je často spojován s exponenciální časovou složitostí. Panika ovšem není na místě, protože náš „skoro-backtracking“ je hodný a pokorně seběhne v lineárním čase. Idea důkazu je jednoduchá. Sice se může stát, že při přidávání některého z kůlů budeme muset hodně kůlů z již hotové části vyhodit, ale na druhou stranu víme, že každý kůl do vytvářené části (horní i dolní) vložíme právě jednou a nejvýše jednou každý kůl vyhodíme. Při počítání složitosti ještě nesmíme zapomenout, že jsme na začátku kůly třídili, a třídění nám zabere $\mathcal{O}(N \cdot \log N)$. Takže celková časová složitost bude: $\mathcal{O}(N \cdot \log N + N) = \mathcal{O}(N \cdot \log N)$.

Paměťová složitost je lineární, protože si potřebujeme pamatovat pouze načtené kůly a vytvářený obal (a obal nemůže obsahovat víc, než původní množství kůlů).

V implementaci bylo potřeba se vypořádat ještě s drobnými detaily navíc, ale to už si přečtěte přímo ve zdrojovém kódu.

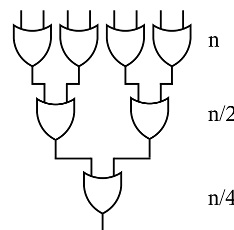
Martin „Bobřík“ Kruliš

20-2-6 Hrady, hrádky, hradla

V druhé serii našeho seriálu o elektronice, hradlech a radostech podobných jste měli za úkol vymyslet "hromadný" OR. Je mnoho způsobů, jak se dá taková funkce OR realizovat. Jak už je obvyklé, měli jste být při návrhu šetrní a ještě navíc, dokázat že vaše řešení je nejšetrnější. Nejdříve dokážeme, že na spočítání jednoho výstupu z n vstupů potřebujeme nejméně $n-1$ hradel. Poté ukážeme, že k takovému výpočtu je třeba minimálně logaritmický počet hladin a to $\lceil log_2(n) \rceil$. Následně ukážeme konstrukci takového obvodu pro obecné n.

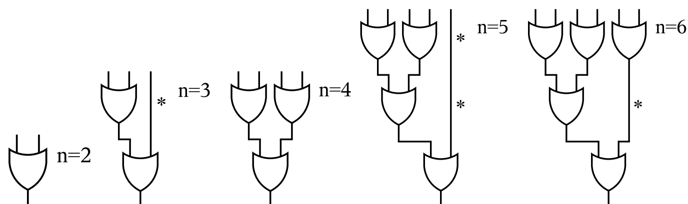
Začneme nejdříve důkazem minimálního počtu hradel. K dispozici máme jenom a pouze hradla dvouvstupová. Výstupy hradel se dle definice z prvního dílu seriálu spojovat nesmí, proto přidáním hradla můžeme ze dvou "drátu" vyrobit jeden. Naopak rozdvojování "vodičů" nám nijak nepomůže. Tedy obecně pro n "vodičů" na vstupu potřebujeme $n-1$ hradel abychom je zredukovali postupným přidáváním hradel na jeden výstup.

Nyní se vrhneme na minimální odhad počtu hladin. Jak už víme z přechozího odstavce, jedno hradlo nám ze dvou "drátu" umí vyrobit jeden a nic lepšího se nám s jedním hradlem vyrobit nepodaří. Tedy za jednotku času, což je čas na průchod informace jedním hradlem, umíme s n "vodičů" vyrobit minimálně $n/2$. Hladin tedy bude tolik, kolik členů posloupnosti $n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, ..., 2$. Z toho nám vychází počet hladin $\lceil log_2(n) \rceil$.



Nyní víme, jaké má mít takový obvod parametry. Zbývá vymyslet, jak vlastně vypadá. Pro n, jež je mocninou dvojky je konstrukce jednoduchá a přímo plyne z důkazu minimální hloubky zapojení. V takovém případě bude mít každá hladina postupně $n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, ..., 2$ hradel, jejichž vstupy jsou zapojené do výstupů předchozí vrstvy. Na obrázku je takový obvod pro $n = 8$.

Pro obecné n budeme obvod konstruovat jakoby indukcí. Začneme pro $n = 2$ jedním hradlem OR. Hradla budeme přidávat dle následujících pravidel. Nejdříve si ale zadefinujeme pojem zaplněná hladina. To je taková, jejíž hranicí neprochází vodič. Následně pokud existuje nějaká nezaplněná hladina, přidáme hradlo OR, tak, že prochází drátem, který procházel hranicí hladiny, tím nám přibyde jedno hradlo a jeden vstup, který protáhneme do vstupní hladiny (tím nám může přibýt prázdných hladin). Pokud prázdné hladiny nejsou, přidáme hradlo tak, že bude dělat OR mezi dosavadním výstupem a přidáváným vstupem (tím nám pro $n \neq 2^n - 1$ jistě vziknou volné hladiny). Na obrázku je prvních pět kroků konstrukce, hvězdičkami jsou označeny volné hladiny.



Cyril Hrubiš

Úloha 20-2-2 – Setříděné stromy – program

```
const MaxN = 10000;

var Posloupnost1:array[1..MaxN] of real;
    Posloupnost2:array[1..MaxN] of real;
    DelkaPosloupnosti1:integer;
    DelkaPosloupnosti2:integer;
    K:integer;
    Zacatek1,Zacatek2:integer;
    DelkaIntervalu:integer;
    Stred1,Stred2:integer;

procedure NactiVstup();
var index:integer;
begin
{V praktické implementaci v lese by tohle již bylo hotovo ...}
   read(DelkaPosloupnosti1);
   for index:=1 to DelkaPosloupnosti1 do
      read(Posloupnost1[index]);
   read(DelkaPosloupnosti2);
   for index:=1 to DelkaPosloupnosti2 do
      read(Posloupnost2[index]);
   read(K);
end;

function PorovnejStromy(Strom1,Strom2:integer):boolean;
{Vrací true, pokud je strom v první posloupnosti vyšší}
begin
   if (Strom1 > DelkaPosloupnosti1) then PorovnejStromy:=false
   else if (Strom2 > DelkaPosloupnosti2) then PorovnejStromy:=true
   else PorovnejStromy:=(Posloupnost1[Strom1] > Posloupnost2[Strom2]);
{Nebo jiná implementace porovnávání stromů ...}
end;

begin
   NactiVstup();
   if (DelkaPosloupnosti1 + DelkaPosloupnosti2 < K) then begin
      writeln('Takový strom neexistuje.');      {Nemáme ani K stromů ...}
   end else if DelkaPosloupnosti1 = 0 then
      writeln(K,'. nejvyšší strom je ',K,'. v druhé posloupnosti.');
   else if DelkaPosloupnosti2 = 0 then
      writeln(K,'. nejvyšší strom je ',K,'. v první posloupnosti.');    {Tím jsme ošetřili speciální případy}
   else begin
      if DelkaPosloupnosti1 > K then DelkaPosloupnosti1:=K;
      if DelkaPosloupnosti2 > K then DelkaPosloupnosti2:=K;
{Dál než K-tý prvek, který nás zajímá, nebude. A teď nás čeká nastavení počátečních podmínek}
      if DelkaPosloupnosti1 = K then begin
         if DelkaPosloupnosti2 = K then begin
            Zacatek1:=0; Zacatek2:=0;
            DelkaIntervalu:=K;
         end else begin
            Zacatek1:=K - DelkaPosloupnosti2 - 1; Zacatek2:=0;
            DelkaIntervalu:=DelkaPosloupnosti2 + 1;
         end;
      end else begin
         if DelkaPosloupnosti2 = K then begin
            Zacatek1:=0; Zacatek2:=K - DelkaPosloupnosti1 - 1;
            DelkaIntervalu:=DelkaPosloupnosti1 + 1;
         end else begin
            Zacatek1:=K - DelkaPosloupnosti2; Zacatek2:=K - DelkaPosloupnosti1;
            DelkaIntervalu:=DelkaPosloupnosti1 + DelkaPosloupnosti2 - K + 1;
         end;
      end;
      while DelkaIntervalu > 1 do begin {Nyní začne binární vyhledávání}
         Stred1:=Zacatek1 + (DelkaIntervalu div 2);
         Stred2:=Zacatek2 + (DelkaIntervalu div 2);
         if PorovnejStromy(Stred1,Stred2) then
            Zacatek1:=Stred1
         else Zacatek2:=Stred2;
         DelkaIntervalu:=(DelkaIntervalu+1) div 2;
      end;
      if (Zacatek1 + Zacatek2 = K - 1) then begin {Máme nalezeno $K-1$ stromů, které jsou nejvýše $(K-1)$-ní}
         if PorovnejStromy(Zacatek1+1,Zacatek2+1) then
            writeln(K,'. nejvyšší strom je ',Zacatek1+1,'. v první posloupnosti.')
         else writeln(K,'. nejvyšší strom je ',Zacatek2+1,'. v druhé posloupnosti.');
      end else begin    {Nebo $K$ stromů, které jsou nejvýše $K$-té}
         if PorovnejStromy(Zacatek1,Zacatek2) then
            writeln(K,'. nejvyšší strom je ',Zacatek2,'. v druhé posloupnosti.')
         else writeln(K,'. nejvyšší strom je ',Zacatek1,'. v první posloupnosti.');
```

```
      end;
   end;
end.
```

Úloha 20-2-3 – Morseovkabezoddělovačů – program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 256
#define INFTY 10000             // Nekonečno

struct vrchol {                 // Vrchol trie
    struct vrchol *syn[2];      // syn[0] pro tečku, syn[1] pro čárku
    struct slovo *slova;        // Seznam slov, která tu končí
    int hloubka;                // Délka morseovkového zápisu (hloubka v trii)
};
struct vrchol koren;

struct slovo {                  // Takhle si ukládáme seznamy slov
    struct slovo *dalsi;        // Vlastně by si stačilo pamatovat jen jedno slovo,
    char s[1];                  // ale třeba se to bude hodit v následující úloze :)
};

// Tabulka kódů:   A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
char morse[26] = { 6,17,21, 9, 2,20,11,16, 4,30,13,18, 7, 5,15,22,27,10, 8, 3,12,24,14,25,29,19 };

void preloz(char *co, char *kam)                                // Přeloží slovo do morseovky
{
    while (*co) {
        int k = morse[*co++ - 'a'];                             // Ďábelský kód písmenka
        while (k > 1) {                                         // Postupně rozkládáme na značky
            *kam++ = ".-"[k%2];
            k /= 2;
        }
    }
    *kam = 0;
}

void nacti_slovnik(void)
{
    char slovo[MAX], mslovo[MAX];
    while (fgets(slovo, sizeof(slovo), stdin) && slovo[0] != '\n') {
        int len = strlen(slovo);
        slovo[len-1] = 0;                                       // Smažeme konec řádku
        preloz(slovo, mslovo);                                  // Přeložíme do morseovky
        struct vrchol *v = &koren;                              // Přidáváme do trie
        for (char *c=mslovo; *c; c++) {
            int z = (*c == '-');                                // Aktuální značka
            if (!v->syn[z]) {                                   // Kam dál? Není-li kam, založíme nový vrchol
                v->syn[z] = malloc(sizeof(struct vrchol));
                memset(v->syn[z], 0, sizeof(struct vrchol));
                v->syn[z]->hloubka = v->hloubka + 1;
            }
            v = v->syn[z];                                      // Vydáme se tím směrem
        }
        struct slovo *s = malloc(sizeof(struct slovo) + len);   // Stojíme ve vrcholu, který odpovídá konci slova,
        s->dalsi = v->slova;                                    // tak tam slovo přidáme
        v->slova = s;
        strcpy(s->s, slovo);
    }
}

int main(void)
{
    char z[MAX];                                                // Vstupní řetězec
    int n;                                                      // Jeho délka
    int b[MAX+1];                                               // Viz popis řešení
    struct vrchol *s[MAX+1];

    nacti_slovnik();                                            // Přečteme slovník a vstup
    fgets(z+1, MAX, stdin);                                     // Pozor, indexujeme od 1
    n = strlen(z+1)-1;

    b[n+1] = 0, s[n+1] = NULL;                                  // Spočítáme všechna b[i] a s[i]
    for (int i=n; i; i--) {
        b[i] = INFTY, s[i] = NULL;
        struct vrchol *v = &koren;                              // Hledáme pasující slova
        int j = i;
        while (v && j <= n) {
```

```
            int zn = (z[j++] == '-');                          // Následující značka
            v = v->syn[zn];                                    // Poskočíme v trii
            if (v && v->slova && b[j] < b[i])                  // Sem nějaké slovo pasuje
               b[i] = b[j]+1, s[i] = v;
         }
      }
      if (b[1] >= INFTY)                                       // Pokud řešení existuje, tak ho vypíšeme
         puts("Řešení neexistuje");
      else {
         printf("Našli jsme řešení na %d slova:\n", b[1]);
         int i = 1;
         while (i <= n) {
            puts(s[i]->slova->s);
            i += s[i]->hloubka;
         }
      }
      return 0;
}
```

Úloha 20-2-4 – Slovíčkaření – program

```
#include <stdio.h>

int main(void)
{
  int  strnum;      //Počet řetězců
  int  alpha;       //Počet znaků v abecedě
  int* str[strnum]; //Pole řetězců (řetězce uvažujeme abstraktně, takže jsou z čísel)
  int  len[strnum]; //Pole délek řetězců
  //Řekněme, že nám obsahy těchto čtyř proměnných někdo naplní (a udělá to správně :) )
  int lensum = 0,maxlen = 0; //součet délek řetězců, délka nejdelšího
  for(int i = 0; i < strnum; i++) {
    lensum += strlen[i];
    if(maxlen < len[i]) maxlen = len[i];
  }
  //Teď si roztřídíme řetězce podle jejich délky
  int lencnt[maxlen + 2], lens[maxlen + 2]; //mapování přihrádek, přihrádky
  for(int i = 0; i <= maxlen; i++) //inicializace
    lencnt[i] = 0;
  lencnt[maxlen + 1] = strnum; // <- trik pro zjednodušení dalšího počítání

  for(int i = 0; i < strnum; i++)
    lencnt[len[i]]++;
  for(int i = 1; i <= maxlen; i++)
    lencnt[i] += lencnt[i-1];
  for(int i=0; i < strnum; i++) {
    lencnt[len[i]]--;
    lens[lencnt[len[i]]]=i;
  }

  int _bucks[alpha + 1];               //Mapovací pole přihrádek - aktuální
  int _prevbucks[alpha + 1];           //Mapovací pole přihrádek - předchozí
  int _bstr[strnum], _prevbstr[strnum]; // Přihrádky - opět aktuální a předchozí
  int *bucks = _bucks;             //tohle nám pomůže,
  int *prevbucks = _prevbucks;     //abychom mohli aktuální a předchozí
  int *bstr = _bstr, *prevbstr = _prevbstr; //snadno zaměňovat

  for(int i = 0; i <= alpha; i++)
    bucks[i] = 0;

  for(int i = maxlen; i>0; i--) {
    int *tmp;
    tmp = bucks; bucks = prevbucks; prevbucks = tmp; //zaměníme mapovací pole
    tmp = bstr;  bstr = prevbstr;   prevbstr = tmp;  //a přihrádky

    for(int j = 0; j <= alpha; j++)
      bucks[j] = 0;

    for(int j = 0; j < prevbucks[alpha]; j++) //napočítáme, kolik bude v
      bucks[str[prevbstr[j]][i - 1]]++; //přihrádkách řetězců z předchozího třídění

    for(int j = lencnt[i]; j < lencnt[i + 1]; j++) //a ještě připočteme
      bucks[str[lens[j]][i - 1]]++;                //řetězce délky i

    for(int j = 1; j < alpha; j++)
      bucks[j] += bucks[j - 1];
    bucks[alpha] = bucks[alpha - 1];  //opět oblíbený trik

    for(int j = 0; j < alpha; j++)
      for(int k=prevbucks[j + 1] - 1; k >= prevbucks[j]; k--)
        //Abychom zachovali uspořádání z předchozího třídění u těch řetězců, které skončí ve stejné přihrádce, musíme je
```

```
        //procházet odzadu, stejně jako se odzadu přidávájí; protože se přihrádky plní odzadu
        int c = str[prevbstr[k]][i - 1];  //roztřídíme do nich nejprve řetězce
        bucks[c]--;                       //z předchozího třídění, protože jsou
        bstr[bucks[c]] = prevbstr[k];     //delší než i
      }
    for(int j = lencnt[i]; j < lencnt[i + 1]; j++) {
      int c = str[lens[j]][i - 1]; //A teď můžeme do předních částí přihrádek
      bucks[c]--;                  // Dát řetězce délky i
      bstr[bucks[c]] = lens[j];
    }
  }
  //Vypíšeme výsledek, aby to vypadalo, že jsme opravdu něco udělali...
  for(int i = lencnt[0]; i < lencnt[1]; i++) //všechny prázdné řetězce přijdou
    printf("\n");                            //na začátek (ani jsme je netřídili)

  for(int k = 0; k < bucks[alpha]; k++) { //a teď vypíšeme ty neprázdné
    for(int i = 0; i < len[bstr[k]]; i++)
      printf("%d ",str[bstr[k]][i]);
    printf("\n");
  }
  return 0;
}
```

Úloha 20-2-5 – Hroší ohrádka – program

```
#include <stdio.h>
#include <stdlib.h>

// Test, zda úsečky AB a BC svírají správný úhel pro horní část obalu.
#define OK_FOR_TOP(A, B, C) ((B.x - A.x)*(C.y - B.y) <= (B.y - A.y)*(C.x - B.x))
// Test, zda úsečky AB a BC svírají správný úhel pro spodní část obalu.
#define OK_FOR_BOTTOM(A, B, C) ((B.x - A.x)*(C.y - B.y) >= (B.y - A.y)*(C.x - B.x))

struct spoint {         // Struktura zapouzdřující souřadnice jednoho kůlu.
        int x, y;
};
struct spoint *points;  // Pole všech kůlů.
int pointsCount = 0;    // Počet všech kůlů.

/* Načte data ze vstupního souboru do globálních proměnných. */
void load(void) {
        FILE *fp = fopen("kuly.in", "r");
        if (!fp) return;

        fscanf(fp, "%d\n", &pointsCount);
        points = malloc(pointsCount * sizeof(struct spoint));

        for(int i = 0; i < pointsCount; i++)
                fscanf(fp, "%d %d\n", &(points[i].x), &(points[i].y));
}
/* Funkce, která porovná souřadnice dvou bodů a vrátí, zda jsou menší, větší, nebo rovny. Použije se jako predikát pro
 * QuickSort. Vracíme záporné číslo, pokud je item1 < item2, kladné, pokud je item1 > item2 a nulu, pokud jsou si rovny. */
int compare(const void *item1, const void *item2) {
        struct spoint *p1 = (struct spoint*)item1;      // Přetypujeme si ukazatele, aby se nám s nimi lépe pracovalo.
        struct spoint *p2 = (struct spoint*)item2;

        if (p1->x == p2->x)     // X-ové souřadnice jsou stejné, takže porovnáme jen y-ové.
                return p1->y - p2->y;
        else
                return p1->x - p2->x;
}
/* Nalezne horní a dolní část konvexního obalu a vypše je do souboru. */
void process(void) {
        if (pointsCount == 0) return;

        struct spoint *resTop = malloc(pointsCount * sizeof(struct spoint));    // Horní část konvexního obalu.
        struct spoint *resBtm = malloc(pointsCount * sizeof(struct spoint));    // Spodní čast konvexního obalu.
        int resTopIdx = 0, resBtmIdx = 0;       // Index posledního kůlu v poli resTop (resp. resBtm).
        int i = 1;                              // Index právě zpracovávaného kůlu.
        // Vložím počáteční bod do horní i dolní části konvexního obalu.
        resTop[0] = points[0];
        resBtm[0] = points[0];
        // Zpracujeme všechny kůly, které leží na stejné x-ové souřadnici, jako první bod.
        while((i < pointsCount) && (points[i].x == points[0].x))
                resTop[++resTopIdx] = points[i++];      // Přidáme je jen do horní hranice.
        // Projdu v cyklu (ve správném pořadí) všechny kůly.
        while(i < pointsCount) {
                // Horní část konvexního obalu: vyhodíme všechny kůly, které porušují konvexitu s novým kůlem i.
                while((resTopIdx > 0) && !OK_FOR_TOP( resTop[resTopIdx-1], resTop[resTopIdx], points[i] ) )
                        resTopIdx--;
                // Přidáme kůl "i" do horní části.
```

```
            resTop[++resTopIdx] = points[i];

            // Spodní část: vyhodíme všechny kůly, které porušují konvexitu s novým kůlem i.
            while((resBtmIdx > 0) && !OK_FOR_BOTTOM( resBtm[resBtmIdx-1], resBtm[resBtmIdx], points[i] ) )
                    resBtmIdx--;
            // Přidáme kůl "i" do spodní.
            resBtm[++resBtmIdx] = points[i];
            i++;    // Vezmem další kůl.
        }
        // Vypíšeme výsledky do souboru.
        FILE *fp = fopen("plot.out", "w");
        fprintf(fp, "%d\n", resTopIdx + resBtmIdx);
        for(i = 0; i < resTopIdx; i++)  // Vrchní část vypíšeme normálně.
                fprintf(fp, "%d %d\n", resTop[i].x, resTop[i].y);
        for(i = resBtmIdx; i > 0; i--)  // Spodní část vypíšeme pozpátku a bez posledního prvku.
                fprintf(fp, "%d %d\n", resBtm[i].x, resBtm[i].y);
}
int main(void) {
        load();         // Načteme souřadnice.
        qsort(points, pointsCount, sizeof(struct spoint), compare);
        process();      // Nalezneme konvexní obal a vypíšeme jej do souboru.
        return 0;
}
```

Výsledková listina dvacátého ročníku KSP po druhé sérii

		škola	*ročník*	*sérií*	*2021*	*2022*	*2023*	*2024*	*2025*	*2026*	*série*	*celkem*
1.	Peter Ondrúška	SPŠDubnica	4	2		10	10	11	7	11	42,0	85,8
2.	Štěpán Weber	GBuďánka	3	2	7	10	9			10	37,4	78,9
3.	Jan Michelfeit	G HBrod	4	7	7	10	10		7	11	39,0	78,4
4.	Alena Skálová	GNaVPláni	4	3	7	10			7	11	36,8	75,5
5.	Filip Hlásek	GMikuláš	1	2	7	10		7,5	3	11	37,2	75,1
6.	Filip Štědronský	GMikuláš	1	2	7	7		6		11	35,4	70,9
7.	Tomáš Toufar	G Bílovec	4	2	7	10			7	11	37,0	68,0
8.	Vlastimil Dort	GŠpitálsPH	2	7	7	10		6	3	11	35,2	67,7
9.	Libor Plucnar	GPBezruče	3	7	7	10	10		1	11	38,0	67,4
10.	Petr Malý	GSladkNám	4	2	7	10			7	11	37,0	64,9
11.	Trung Ha duc	GMasaryk	2	7	7	7		10		11	36,0	64,5
12. – 13.	Jitka Novotná	G Bílovec	3	2	7	10				4,5	24,6	58,6
	Vojtěch Tůma	G Jihlava	4	6	7	10		6		11	35,5	58,6
14.	Lukáš Kripner	G Litvínov	2	6	5	5		7		10,5	31,6	57,7
15.	Pavel Veselý	G Strakon	3	9	7	10		8		10	35,6	55,2
16.	Stanislav Fořt	G Tábor	0	6	4,5	1	2	5	0	10,5	26,1	54,5
17.	Jan Škoda	GMikuláš	1	2	6				0	7	16,2	51,8
18.	Jakub Hrnčíř	GFXŠaldyLI	1	2	7		3		5	4	27,5	49,9
19.	David Marek	SPŠ Zlín	4	4	7		7				15,7	49,2
20.	Jan Matějka	GJírovco	3	4	7	10		8,5			26,4	35,4
21.	Jakub Kaplan	GJKTylaHK	4	16							0,0	34,4
22.	Jiří Zárevúcky	SŠInformFM	3	1							0,0	32,5
23.	Petr Babička	G SvětláNS	3	2	4	2				6	18,9	30,9
24.	David Brázdil	G Zlín	3	6							0,0	29,8
25.	Martin Vlach	G Jihlava	4	1							0,0	29,7
26.	Milan Rybář	GJJunman	3	1							0,0	29,3
27.	Karel Tesař	SPŠEPlzeň	2	2	7					5	15,1	28,4
28. – 29.	Jiří Keresteš	SPŠEPlzeň	2	4	7	1				9	19,2	28,2
	Petr Sokola	SPŠ Zlín	4	2						11	11,0	28,2
30.	Pavel Kratochvíl	ZŠSvětlá	0	2	7					9	17,4	28,1
31.	Radim Cajzl	G NMnMor	1	12	7					8	14,7	27,7
32.	Adam Streck	G Hořice	4	1							0,0	27,6
33.	Vojtěch Kolář	G Neratov	3	1							0,0	27,3
34.	Pavel Taufer	GArcibisk	2	1	7		2		2	7	25,7	25,7
35.	Martin Patera	GArabská	2	1							0,0	25,5
36.	Tomáš Sýkora	G VKlobou	4	10	7						7,0	24,8
37. – 38.	Jakub Suchý	GMikuláš	1	2						4,5	7,6	24,1
	Jan Žák	G HBrod	3	6							0,0	24,1
39.	Alžběta Pechová	SPŠSVsetín	3	3			2	4			10,6	20,6
40.	Jakub Červenka	GŠpitálsPH	2	3	7					11	18,0	18,0
41.	Nikolas Zigmund	ZŠHavířov	1	1							0,0	8,9
42.	Peter Smatana	EkoGLabsBO	3	1							0,0	6,4
43.	Miroslav Klimoš	G Bílovec	3	20							0,0	4,4