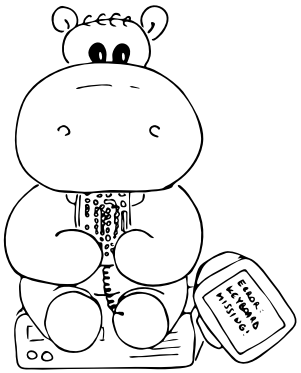
Milí řešitelé a řešitelky!



Vidíte před sebou druhou sérii 23. ročníku KSP. Každá série obsahuje 7 úloh, z toho 4 nejlépe vyřešené se započítávají do celkového bodového hodnocení. Zatímco my budeme opravovat vaše řešení první série, vy už můžete v teple domova, tramvaje, školní lavice v poklidu řešit úlohy série druhé. Vy, kdo řešíte zároveň i PraSátko,[1] si dobře rozvrhněte, kdy budete co řešit, neboť tentokrát máme termín shodný.

Termín odevzdání druhé série je stanoven na pondělí 6. prosince v 8:00 SEČ, což znamená, že papírové řešení byste měli podat na poštu do středy 1. prosince.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš šifrovací certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

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

Na případné dotazy vám rádi odpovíme také na adrese *ksp@mff.cuni.cz* a v diskusním fóru na našem webu.

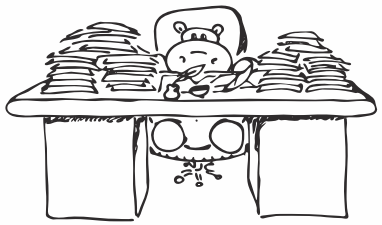
Druhá série třiadvacátého ročníku KSP

Alan Mathison Turing se narodil 23. června 1912 a zemřel o 42 (!) let později. Nevíte-li o něm nic dalšího, než že si nějakým způsobem musel zasloužit, abychom tu o něm psali, můžete začít přemýšlet nad tím, proč tak brzo. Ale radši čtěte dál. Do sebevraždy se možná trefíte, její okolnosti jsou však krajně zajímavé.

Gordon Brown se omluvil 10. září 2009. Vyjádřil se v tom smyslu, že je mu to celé moc líto a že za všechny, kteří díky Turingově práci mohou žít ve svobodě, říká, že… je mu celá věc moc líto. Učinil tak díky petici, kterou zařídil jistý britský programátor.

23-2-1 Balíčky balíčků — 10 bodů

Petice byla samozřejmě elektronická, ale aby ji pan premiér nemohl *zamést pod koberec*, rozhodne se ji její iniciátor John Graham-Cumming vytisknout a zaslat poštou. Poprvé po mnoha letech studuje poštovné a co nevidí?



Výhodné nabídky balíčků! Můžete poslat jeden o váze N kg, dva o váze $N-1$ kg, tři o váze $N-2$ kg, …, N o váze 1 kg, kde N závisí na ročním období, denní hodině a sjízdnosti silnic. To vás nemusí trápit, N dostane váš program na vstupu.

Dále dostanete váhu H petice v celých kilogramech. Vaším úkolem bude vymyslet, které nabídky balíčků je třeba vybrat, aby se do nich dohromady vešlo H kg petice, ale zároveň aby jejich kapacita byla co nejblíže tomuto H.

Je třeba zdůraznit, že „3 balíčky, každý o váze $N-2$ kg“ je jedna nabídka, kterou jako celek buď přijmete, nebo nepřijmete. Chcete-li poslat $3N-6$ kg, je to ideální volba.

Chcete-li poslat $N-2$ kg a N není úplně malé (třeba $N=100$), je lepší zvolit nabídku „1 balíček o váze N kg“, přestože dva kilogramy nevyužijete. Stejně dobré řešení by pak bylo vybrat „N balíčků o váze 1 kg“ a nám je jedno, které z takových dvou stejně dobrých řešení vypíšete.

Chcete-li poslat 100 kg a $N=12$, můžete vybrat třeba kombinaci $3\times(N-2)+3\times(N-2)+5\times(N-4)=3\times10+3\times10+5\times8$.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[2] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Turing byl „zakladatel moderní informatiky“. Co myslíte, dařilo se mu na něco takového balit holky?

Zavedl nejvýznamější teoretický model počítače, kterému se dnes říká Turingův stroj. Můžeme si ho představit jako nekonečnou pásku popsanou symboly z nějaké konečné množiny. Nad páskou se pohybuje hlava stroje a v každém kroku výpočtu podle jednoduché tabulky pravidel přečte symbol, nahradí ho jiným, a přesune se doleva nebo doprava. Existuje teze, že práce každého rozumného (teoretického i skutečného) počítače se dá na práci Turingova stroje převést.

Na základě tohoto modelu dokázal, že neexistuje zaručeně konečný algoritmus, který by dokázal posoudit, zdali se jiný daný algoritmus na daných datech zastaví.

Tím rozhodnul Hilbertův problém z roku 1928, který se ptal po existenci zaručeně konečného algoritmu, který dostane matematické axiomy a domněnku a rozhodne, je-li domněnka z těchto axiomů odvoditelná. Zamítnul existenci takového algoritmu, protože díky formalizaci Turingova stroje uměl vyjádřit „zastaví se algoritmus na daných datech?“ jako matematickou domněnku.

Vymyslel též „Turingův test“. Jde v podstatě o člověkostřednou definici inteligence– objekt je inteligentní právě

[1] `http://mks.mff.cuni.cz/`
[2] `http://ksp.mff.cuni.cz/zaciname/codex.html`

tehdy, nerozezná-li lidský pozorovatel jeho lingvistický výstup od lingvistického výstupu člověka. Takové Turingovské testování provádí každý z nás vždy, když mu píše neznámá entita po IM a nabízí výrobek.

Co na tom, že ho v mnoha (i zmíněných) věcech asi o rok předběhl Alonzo Church – Turingův přístup se ukázal být stravitelnější než Churchův lambda kalkulus.

23-2-2 Zastavení — 10 bodů

Když už jsme u toho zastavování… Máme-li spravedlivou šestistěnnou kostku, umíme na ní generovat (celá) náhodná čísla mezi 1 a 6 (včetně) se stejnou pravděpodobností 1/6. Představme si, že máme po ruce 4-, 6-, 8-, 12- a 20stěnnou kostku. Pro které hodnoty n umíme pomocí těchto kostek generovat (celá náhodná) čísla mezi 1 a n (včetně) tak, aby všechna padala se stejnou pravděpodobností a trvalo nám to zaručeně konečný počet kroků?

Pokud píšeme generovat, myslíme tím prostě, že nějaký váš algoritmus dostane kostku „zapůjčenou“ a může si s její pomocí generovat náhodná čísla, na jejichž základě nakonec spočítá požadované výsledné náhodné číslo. Jakékoli jiné náhodné generátory jsou zakázány. Nepůjde-li vám to ve vší obecnosti, zkuste ověřit, jestli (a jak) jdou vygenerovat čísla od 1 do 120.

Třeba náhodné číslo v intervalu 1 až 32 snadno získáme na dva hody výrazem $(8(d_4-1)+d_8)$, kde d_4 je číslo hozené na čtyřstěnné kostce a d_8 číslo hozené na osmistěnné kostce.

O Turingovi se povídá spousta „geekovských“ drbů. Často se zmiňuje jeho kolo, byl to náruživý cyklista. Začal mu prý jednou padat řetěz a on nedbal opravy, místo toho si při jízdě počítal otočky a pokaždé, když se to mělo stát, seskočil z kola a opatrně posunul řetěz o pár pozic dál.

To zní strašlivě neprakticky, dokud si neosvětlíme, že řetěz padal právě a jen tehdy, sešel-li se jeden ohnutý zoubek na kotouči s jednou nedokonalou pozicí na řetězu. Je pak otázka dělitelnosti, kdy se při jízdě takové dvě chyby setkají: klidně to mohlo nastávat „jen“ každých deset minut.

23-2-3 Projížďka — 12 bodů

Představme si Turinga mířícího vlakem do krajiny, kterou si chce na kole projet. Dostaneme na vstupu seznam rozcestí a cest, které vedou mezi nimi. Jeho kolo je tentokrát bezvadné, leč terén je obtížný a hlavně nevyrovnaný – některé silnice jsou krátké a klidné, dokonce vedou z kopce; jiné jsou dlouhé, klikaté a strmé, takže velmi unavují.

Turing každé z nich při pohledu do mapy přiřadil celé číslo vyjadřující tuhle subjektivní obtížnost – na kladně ohodnocených cestách si bude odpočívat a na záporně ohodnocených tuhle nashromážděnou energii vydá.

Teď by od vás chtěl, abyste napsali program, který mu najde takovou cestu (včetně začátku – a může začínat na libovolném rozcestí), po které jednak projede všechny silnice *právě jednou*, druhak bude na každém rozcestí součet všech Turingem do té chvíle projetých silnic *nezáporný* a navíc se vrátí na rozcestí, na kterém začal. Chcete-li a myslíte-li si, že vám to pomůže, předpokládejte klidně, že z každého rozcestí vychází sudý počet silnic.

Důležité je, že Turingovy vědecké výsledky blednou ve srovnání s jeho srdnatostí za druhé světové války. Účastnil se dešifrování německého šifrovacího stroje Enigma v anglickém Bletchley Parku, postavil při této příležitosti jednoúčelový počítač Bombe, který podstatně urychlil procházení možností. Díky tomu, že byly kódy Německa zlomeny, nabrala válka poněkud jiný rozměr, který hezky zachytil Neal Stephenson ve své knížce Kryptonomikon (ve které mimochodem Turing skutečně vystupuje):

„[Ve filmech] se praví, že Patton a MacArthur jsou odvážní generálové. Svět bez dechu očekává jejich další neohrožené eskapády za nepřátelskou linií. Waterhouse ví, že Patton a MacArthur jsou víc než cokoliv jiného inteligentní konzumenti [prolomených šifer] Ultra/Magic. Používají je k tomu, aby zjistili, kde nepřítel soustředil síly, pak ho obejdou a udeří na místo, kde je nejslabší. To je všechno.“

23-2-4 Plánování — 10 bodů

Pořád by ale bylo poněkud nespravedlivé upřít všem spojeneckým generálům jakoukoliv tvořivost. I když vám cizí zprávy diktují, na která místa potřebujete kdy zaútočit, pořád máte omezené zdroje.

V této úloze dostanete na vstupu seznam časových intervalů zadaných přirozenými čísly, ve kterých je potřeba likvidovat nějaký výhodný cíl ploužící se za frontovou linií. Seznam je uspořádaný podle počátků těchto intervalů.

Chceme od vás, abyste našli minimální počet bombardérů, který stačí k likvidaci všech cílů, a jejich časový rozvrh.

Neuvažujte doby přilétání a odlétání, tankování, údržby a podobně, to už je započítáno v intervalech. Letadlo je vždy plně využito celý požadovaný interval, takže se nesnažte o nějaké triky s předčasným návratem, jedním letadlem na dvou místech apod.

Příklad vstupu:

```
5-8 7-12 11-13 12-15
```

Příklad výstupu:

```
2
1: 5-8 11-13
2: 7-12 12-15
```

23-2-5 Zaměřování — 8 bodů

Historicky se matematika ve válkách používala vedle šifrování také při všemožné balistice. Nabízíme vám touto historií velmi vzdáleně inspirovanou úlohu:

Dostanete na vstupu pozici dvou kanónů v kartézské soustavě souřadnic a po řadě vrcholy i nekonvexního mnohoúhelníka (ale žádné dvě jeho nesousedící hrany se neprotínají), na jehož obvod šílený velitel přikázal střílet. Souřadnice nemusí být celá čísla.

Navíc vyžaduje, aby oba kanóny střílely na takový bod na obvodu, pro který platí, že je obsah trojúhelníka určeného oběma kanóny a tímto bodem co nejbližší zadanému číslu. Pokud je jich více, vypište libovolný z nich.

Příklad vstupu (kanóny, pevnost, obsah):

```
[1, 1] [1, 2]
[2, 1] [2, 2.5] [3.71, 2.5] [3.71, 6] [7, 1]
1
```

Odpovídající výstup může být třeba `[3, 1]`.

Po válce pracoval Turing na stavbě počítačů, tentokrát už ne jednoúčelových, a nějakou dobu se mu dařilo konkurovat podstatně lépe financovanému americkému výzkumu.

23-2-6 Testovací — 10 bodů

Potřebujeme-li u takového jednoduchého počítače otestovat bezchybnost, chceme nějakou dosti jednoduchou úlohu, po jejímž vyřešení a naprogramování si budeme moci být

jisti, že pokud dává počítač špatné výsledky, není to naším naprogramováním.

Co třeba takovou?

Máte zadanou setříděnou posloupnost přirozených čísel a chcete vypsat všechny trojice ve tvaru a, $a+k$, $a+2k$ (pro všechna možná přirozená a, k), které se v ní nacházejí.

Příklady (vstup → výstup):

`1 2 3 5 8 9` → `1-2-3 1-3-5 1-5-9 2-5-8`

`1 2 4 5 10 11` → nic (zde není žádná taková trojice)

Turing byl také mimochodem homosexuál, v Británii to bylo do roku 1968 trestné (u nás „jen“ do roku 1960), a tak mu soud, když se na to přišlo, zakázal pracovat na vládních projektech na výstavbu počítačů a nařídil hormonální „léčbu“. O dva roky později si Turing kousl do jablka, které předtím naplnil kyanidem. Bizarní? Měl moc rád Sněhurku a sedm trpaslíků od Disneyho – inspiraci tedy možná našel v tomto filmu.

Každopádně se všeobecně soudí, že ho k tomu dohnaly dosti nepěkné vedlejší účinky prováděného léčení a to je oním důvodem, proč se mu britský premiér po 55 letech omluvil.

Mezi informatiky je Turing oblíbený, protože jeho životní příběh dokumentuje, jak může být takový teoretik užitečný, když vyvstanou velké praktické problémy. Existují odhady, podle kterých analytici z Bletchley Parku zkrátili válku o rok a zachránili milion lidí, a je samozřejmé nemožné říct, jestli tomu tak je. Je rozhodně dojemné si uvědomit, že po válce samozřejmě jako hrdinové oslavováni nebyli, protože britská vláda nechtěla, aby se o prolomení daných šifer vědělo. Mnoho jich tedy, stejně jako Turing, zemřelo bez jakéhokoliv uznání.

V češtině vyšla v edici Aliter popularizační knížka „Muž, který věděl příliš mnoho“, která se celá věnuje Turingově životu a snaží se jemně vysvětlovat jeho výsledky. Pokud vás zajímá teoretická informatika a chcete být drsní, Charles Petzold nedávno sepsal „Annotated Turing“, což je přetisk Turingovova ústředního článku s poznámkami.

Existuje docela známá beletristická knížka o kryptoanalyticích z Bletchley Parku a špionážních tanečcích kolem, která se jmenuje „Enigma“ – dokonce podle ní natočili film. Tam už ale našeho hrdinu nenajdete.

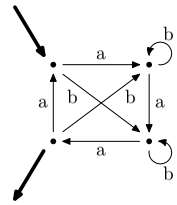
23-2-7 Regulomaty — 12 bodů

Po představení syntaxe regulárních výrazů se podíváme na zoubek tomu, co se s nimi děje uvnitř počítače.

Proč se vlastně oněm výrazům říká regulární? Popisují totiž regulární jazyky, tedy jazyky rozpoznávané konečnými stavovými automaty. Že nevíte, o čem píšu?

Konečný automat je množina (Q, A, δ, q_0, F) … formální definici nechme na seriál 17. série a úlohu 17-2-5.

Konečný automat si představme jako množinu stavů, mezi kterými můžeme různě přecházet tím, že přečteme znak ze vstupu. Ilustrativní obrázek napoví víc než suchá teorie. Tlusté šipky symbolizují vstupní stav a výstupní stavy.



Výstupem našeho automatu pak bude přijetí, nebo odmítnutí, podle toho, jestli řetězec vyhovuje, či nikoli.

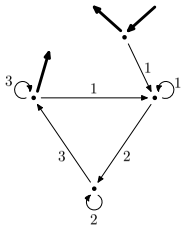
Na začátku jsme v počátečním stavu. Přečteme písmenko ze vstupu a vybereme si příslušnou hranu a po ní přejdeme do dalšího stavu. A zase přečteme písmenko ze vstupu, vybereme si hranu, přejdeme…

Když nemáme co přečíst, stačí nám jednoduše zjistit, jestli jsme zrovna ve výstupním stavu, nebo nikoli. Pokud jsme ve výstupním stavu, tak jsme přijali vstup, jinak ho odmítneme. Vstup taktéž odmítneme, pokud při běhu zjistíme, že z aktuálního stavu žádná vhodná hrana nevede.

Automat na obrázku tedy přijímá taková slova jako `bba`, `bababa`, `aaaaaaa`, ale odmítne například slova `abba`, `baba` (skončí vpravo dole), `ababab` (skončí vpravo nahoře), `aaaa` nebo λ[3] (skončí vlevo nahoře).

Existuje hezká věta, která říká, že každý konečný automat lze převést na regulární výraz. To znamená, že umíme najít regulární výraz, který matchuje právě ty vstupy, které přijme konečný automat (a žádné jiné). Důkaz té věty se dá udělat třeba ukázáním univerzálního postupu, který ale bude až v autorském řešení, jinak byste přišli o tu zábavu vymýšlet, jak na to.

Úkol 1 [4b]: Převeďte automat na obrázku na regulární výraz:



Nebylo to tak hrozné, ne? Co si to vzít obráceně? Existuje docela hezká věta, která dokazuje, že každý regulární výraz lze převést na konečný automat.

Některé jdou i ručně.

Úkol 2 [3b]: Převeďte zadaný výraz na konečný automat:

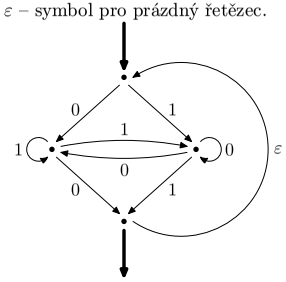
`(1(23|32)|2(31|13)|3(12|21))*`

Čím méně hran a stavů, tím více bodů máte šanci získat (za automat mající více než 10 stavů nebudou ani 2 body).

Drtivou většinu výrazů ale hned tak jednoduše převést nepůjde, nebo to alespoň není na první pohled vidět. Zavedeme proto nedeterministické konečné automaty (NKA).

NKA je automat, u kterého může z jednoho stavu vést více hran pro jedno písmenko. Program si tedy může vybrat, kterou cestou půjde. Vstup je pak přijat, pokud existuje možnost, jak ho přijmout, neboli pokud existuje vhodná cesta (tedy po které program mohl jít), která končí v nějakém z výstupních stavů.

Navíc si dovolíme takzvané ε-přechody. To jsou hrany, po kterých je možno přejít, aniž přečteme znak ze vstupu. Aby bylo poznat, že jsme k hraně nezapomněli připsat její znak, píšeme k ní ε – symbol pro prázdný řetězec.



Tento automat přijímá třeba řetězce `10111`, `1111`, ale třeba ne `000` nebo `111`. Vyzkoušejte si sami, jak.

[3] λ je obvykle používaná zkratka pro prázdné slovo.

Úkol 3 [5b]: Převeďte výraz `(10(1(10)*1)*01)*` na NKA s ε-přechody. Bonus 2b pro ty, kdo vymyslí jednodušší (tedy kratší) ekvivalentní výraz.

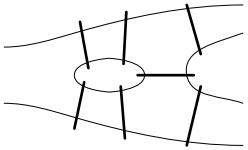
Recepty z programátorské kuchařky

Procházky po grafech

Tento spisek hojně používá jazyka teorie grafů. Pokud ještě termíny jako „hrana“, „cesta“ nebo „stupeň vrcholu“ neznáte, přečtěte si prosím nejprve úvodní kuchařku o grafech na našich webových stránkách.

Historický problém

V roce 1735 se švýcarskému matematikovi Leonhardu Eulerovi na stůl dostal na první pohled jednoduchý problém, který mu předložil starosta města Královec (dnešní Kaliningrad). Královcem teče řeka Pregola, na ní je několik ostrovů a ostrovy byly spojeny se zbytkem města mosty. Dobová ilustrace situaci vystihla takto (schématická kresba):



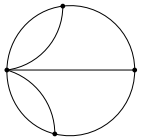
Pan starosta se pana matematika v dopise tázal, jestli je možné začít z některého z břehů (nebo ostrovů) a udělat si vycházku po městě tak, že každým mostem projdeme právě jednou.

Navíc chtěl procházku skončit na kusu suché země, ze kterého jsme vyšli. Euler jej nejprve chtěl poslat k šípku – problém jde snadno vyřešit rozborem případů, což by zvládli i tehdejší studenti střední školy (natož pak ti dnešní).

Profesor Euler se ovšem zachoval jako pravý mamematik – přišel na to, jak problém zobecnit, a mistrně vyřešil hádanku i pro všechna možná města, která kdy budou chtít pořádat podobné procházky.

Eulerovský tah

Pojďme si nyní problém popsat abstraktně a tím si připomenout grafovou terminologii. Vrcholy našeho grafu jsou kusy pevniny, ať už to budou části města nebo ostrovy. Mezi dvěma vrcholy povede hrana, pokud jsou spojeny mostem, a onen most odpovídá hraně. V tomto zadání má smysl uvážit, že mezi dvěma kusy pevniny povede mostů více – například v Praze jich vede tolik, že se na to ptají v leckteré zeměpisné olympiádě. Graf, kde mezi vrcholy vede více hran, nazýváme *multigraf*, a pokud dvě hrany vedou mezi stejnými vrcholy, mluvíme o nich jako o *paralelních* hranách.



Obecná procházka v grafu z vrcholu A do vrcholu B (posloupnost hran taková, že cílový vrchol předchozí hrany je počáteční vrchol hrany následující) se nazývá *sled* z A do B. Ve sledu se mohou opakovat jak hrany, tak vrcholy; sled tedy není řešením našeho problému (ve sledu je možné se vrátit po hraně, ze které jsme právě přišli). Pro naši úlohu se hodí posloupnost hran taková, že vrcholy se opakovat mohou, ale hrany nikoli. Této posloupnosti se říká *tah* z A do B. Kdyby se neopakovaly ani vrcholy, pak posloupnost označujeme jako *cestu*. Tah (respektive sled) je *uzavřený*, pokud začíná v A a končí také v A.

Podíváme-li se tedy na mapu Královce jako na multigraf, ptáme se, zdali existuje uzavřený tah takový, že každou hranu navštíví právě jednou. Takovému tahu pak říkáme *uzavřený eulerovský*.

Mimochodem, tahu se „tah“ neříká jen tak náhodou. Děti se často ve školce překonávají v umění nakreslit obrázek jedním tahem, aby se tužkou nemuselo vracet po už nakreslené čáře. Pokud si obrázek představíme jako graf (čáry jsou hrany, místa jejich setkání vrcholy), pak eulerovský tah nalezneme jen v tom obrázku, který lze nakreslit jedním tahem. V uzavřeném eulerovském tahu se pak vrátíme i do místa, kde jsme začali.

Podmínky tahu

Je na čase poodhalit řešení našeho problému s eulerovským tahem. Půjdeme na to jako matematici – nejprve ukážeme *nutnou* a hned nato *postačující* podmínku. Nutná vlastnost grafu je taková, že bez ní eulerovský tah není možné najít; postačující vlastnost je ta, se kterou vždy eulerovský tah najít umíme. Jsou-li obě podmínky stejné, pak se jedná o ekvivalenci, a tak tomu bude i nyní.

Představme si, že jsme kouzlem nějaký uzavřený eulerovský tah našli, ať už je jakýkoli. Vždy, když se dostaneme do jednoho vrcholu (a není důležité, jestli už jsme v něm byli, nebo ne), tak abychom tah uzavřeli, musíme z něj hned také odejít. A protože tah je eulerovský, každou hranou projdeme jen jednou, takže tyto dvě hrany (tu příchozí a odchozí) už nepoužijeme. U každého vrcholu mimo výchozí tedy platí, že hrany tvoří dvojice – jedna, co vedla dovnitř, a jedna, která z něj vedla ven.

Podobná věc platí i pro startovní vrchol. Sice do něj nevstoupíme poprvé pomocí hrany, takže počet navštívených hran u něj bude stále lichý – ale jen do chvíle, než se do něj naposledy vrátíme a skončíme, protože skončením jsme použili poslední hranu, která bude tvořit dvojici s hranou první.

Jakou vlastnost grafu jsme odhalili? Neplatí, že graf má sudý počet hran (protože trojúhelník jedním tahem nakreslíme a přesto má 3 hrany), ale platí, že do každého vrcholu vede sudý počet hran, tedy že graf má **všechny stupně sudé**. Nezapomeňme také na to, že graf musí být souvislý – dva oddělené obrázky jedním tahem bez zvednutí tužky nenakreslíme. Máme nutné podmínky!

Nalezení tahu

Zbývá tedy ověřit, že podmínky jsou i postačující. Mějme souvislý graf, který má všechny stupně sudé. Umíme v něm vždy najít uzavřený eulerovský tah? Ověřme to, jak se na informatiky patří – algoritmem.

Předložený algoritmus je založený na vylepšeném prohledávání do hloubky, tedy DFS. To patří do základního arzenálu každého programátora, jeho popis naleznete třeba v programátorské bibli[4] nebo na Wikipedii.[5] Také o něm existuje hezká kuchařka na našem webu.[6]

Vyberme si vrchol, v něm začneme. Náš algoritmus musí umět označovat hrany jako „probrané“, jako to dělá DFS. Vyberme si tedy jednu hranu, a pokračujme dále, zatím bez vypisování.

Po nějakém tom procházení se jistě stane, že jsme se zastavili – vrchol už nemá žádné nepoužité hrany. Nutně to znamená, že to je ten vrchol, u kterého jsme začínali. V procházení do hloubky se vracíme zpět, ale my k tomu přidáme

[4] Pavel Töpfer: Algoritmy a programovací techniky

[5] `http://cs.wikipedia.org/wiki/Prohled%C3%A1v%C3%A1n%C3%AD_do_hloubky`

[6] `http://ksp.mff.cuni.cz/tasks/20/cook3.html`

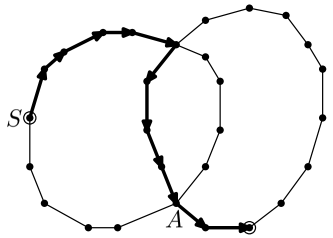
Výsledková listina dvacátého třetího ročníku KSP po první sérii

		Škola	*ročník*	*sérií*	*2311*	*2312*	*2313*	*2314*	*2315*	*2316*	*2317*	*série*	*celkem*
1.	Jakub Zíka	GNAlejíPH	4	1	9	11	11	10	10	10	14	46,0	46,0
2.	Lukáš Folwarczný	GKomHavíř	3	2			10	10	10	10	14	44,7	44,7
3.	Štěpán Šimsa	GJungmanLT	2	10	9		11	10	10		13,5	44,6	44,6
4.	Jan Hadrava	GZborovPH	3	1			9	10	10		13,5	44,4	44,4
5.	Michal Anderle	GTim_Lučen	4	1				10	10	10	12,3	43,6	43,6
6.	Vojtěch Hlávka	GŠlapanice	2	6	9	9	11	10	10	10	10,5	43,1	43,1
7.	Jerguš Greššák	GRaymanaPV	2	2	9			10		10	12,5	42,6	42,6
8.	David Bernhauer	GZborovPH	3	1	6	2	3	10	9,7	3	13,1	41,8	41,8
9.	Matěj Kocián	GLesníZlín	4	3	8			10	9,7	10	8,9	41,7	41,7
10.	Filip Hlásek	GMikulášPL	4	16			11	10	10	10	11,9	41,6	41,6
11.	Juda Kaleta	GKlatovy	2	2	9		7,5	10	10	4	8	41,3	41,3
12.	Ondřej Hübsch	GArabskáPH	1	6	5		6	10		10	12,7	41,2	41,2
13.	Andrej Mariš	PriorPC	3	1	9		8	10		5	8,1	41,0	41,0
14.	David Krška	GJirsíkaČB	4	1	5		9	9	10	10		40,3	40,3
15.	Michal Pokorný	SŠkybernHK	3	2	9		3	9		7	8,4	39,8	39,8
16.	Jan Bok	GJungmanLT	4	2			6	10		5	8,3	38,3	38,3
17.	Peter Zeman	GAnVra	4	1	5	2	3	6	9		8,8	38,2	38,2
18.	Vojtěch Sejkora	SPSE_Pard	2	1	9	2,5	3	10		6	5,7	37,6	37,6
19.	Rastislav Rabatin	GJHroncaBA	2	1	9		6	0	9	8	4,1	37,4	37,4
20.	Ondřej Mička	GJírovcČB	2	5	9		2		10	5	6	34,7	34,7
21.	Filip Matzner		4	2	5			10	2	4	6,5	34,5	34,5
22.	Martin Raszyk	G_Karvina	1	1	6		3	6			6	33,2	33,2
23.	Ondřej Cífka	GNAlejíPH	2	3	5			5	8	5		31,4	31,4
24.	Milan Berka	G_Krumlov	4	1	5		3	10	3,1			29,8	29,8
25.	Jindřich Pilař	GBroumov	3	2	8		3		6	3	0,1	28,7	28,7
26.	Daniel Švec	SPŠERožnov	3	1	5	1	3			5	2,8	27,9	27,9
27.	Michal Punčochář	GJírovcČB	1	1	5	6					4	24,2	24,2
28.	Jiří Eichler	SlovanGOL	3	6				10			11,5	22,7	22,7
29.	Tomáš Varga	GMost	-1	2	4		3			3	1,7	22,0	22,0
30.	Robin Mana	GValašKlob	4	1	3	1,5	3				1,7	19,7	19,7
31.–32.	Anna Dresslerová	GJHroncaBA	4	2	9			10				19,0	19,0
	Mária Mrocková	GJHroncaBA	4	3	9			10				19,0	19,0
33.	Matěj Židek	GBroumov	3	2	5		1		4,5			17,0	17,0
34.	Jan Paštyka	SPSKutHora	2	1	6						3,8	15,9	15,9
35.	Jiří Šebele	GArabskáPH	1	1	5						3,1	14,3	14,3
36.	Jonatan Matějka	GJírovcČB	1	4	2				4,5			10,4	10,4
37.–38.	Alexander Mansurov	GNVPláníPH	2	4				10				10,0	10,0
	Jiří Setnička	G25březnPH	4	11							10	10,0	10,0
39.	Milan Mikuš	GĽŠtúraTN	3	1	9							9,0	9,0
40.	Martin Holec	GSlavičín	4	8	5						2	8,7	8,7
41.	Tomáš Turlík	GRaymanaPV	2	1	6			0			0,1	8,4	8,4
42.	Ondřej Fiedler	GJungmanLT	4	2						5		7,7	7,7
43.–44.	Barbora Hourová	G_Brandýs	4	1	3							5,7	5,7
	Tomáš Velecký	GBezručeFM	0	1	3							5,7	5,7

23-1-4 Program (Ale co trapné numerické chyby?) — C

```
#include <stdio.h>

#define MAX 1000000

int main(void)
{
    int cit, jmen, j, i = 0, zac = 0;

    // Vlastnost jazyka C -- rychlé vynulování pole.
    // Protože pole je plné nul, ukládám si pozici
    // o jedna vyšší než skutečnou.

    int zb[MAX] = {0};
    char vysl[MAX];
    FILE *fin, *fout;
    fin = fopen("zlomky.in", "r");
    fscanf(fin, "%d %d\n", &cit, &jmen);
    fout = fopen("vysledek.out", "w");

    if (cit >= jmen) {
        zac = cit/jmen;
        cit %= jmen;
    }

    fprintf(fout, "%d.", zac);
    if (!cit)
        fprintf(fout, "0");

    while (cit)
    {
        zb[cit] = i+1;
        cit *= 10;
        vysl[i] = cit / jmen + '0';
        i++;
        cit %= jmen;
        if (zb[cit])
            break;
    }

    // Abychom nemuseli vypisovat po znacích,
    // vytvoříme si před začátkem periody "zarážku".
    if (cit)
    {
        j = vysl[zb[cit] - 1];
        vysl[zb[cit] - 1] = 0;
    }

    vysl[i] = 0;
    fprintf(fout, "%s", vysl);

    if (cit)
    {
        fprintf(fout, "(");
        vysl[zb[cit] - 1] = j;
        vysl[i++] = ')';
        vysl[i] = 0;
        fprintf(fout, "%s", vysl + zb[cit] - 1);
    }

    return 0;
}
```


vypisování cesty – postupně pozpátku vypisujeme hrany, kterými se vracíme zpět v prohledávání.



Na obrázku výše je příklad právě probíhajícího algoritmu. Začal ve zvýrazněném vrcholu vlevo, procházel po šipkách až do bodu A, kde volil hrany tak, že hned skončil na začátku. Dále pokračoval vypisováním hran pozpátku, až došel zase do bodu A. Zde si vybral jednu ještě nepoužitou hranu a po ní prošel celou druhou kružnici – zbytek hran – zpět do bodu A. Nyní vypisuje hrany pozpátku od bodu A.

Buď tímto výpisem dojdeme až na začátek, nebo se dostaneme do vrcholu, který má ještě nějaké nepoužité hrany (situace může vypadat třeba jako na obrázku). Potom vypisování zastavíme a pokračujeme v prohledávání DFS přes nepoužitou hranu. I tam se to může zastavit (a zastaví), i tam začneme vypisovat pozpátku. Nakonec dojdeme do původního místa rozbočení, a budeme opět pozpátku vypisovat hrany, které nás nakonec dostanou až na počátek, kde skončíme.

Najde tento algoritmus opravdu korektní uzavřený eulerovský tah? Graf byl souvislý a o algoritmu DFS se ví, že v takovém případě navštíví každou hranu právě jednou. Algoritmus opravdu vypisuje cyklus – jen je u něj trochu zvláštní způsob, jak ho vypisuje. Když dojde na křižovatku s ještě nepoužitými hranami, tak výpis zastaví, tiše po nich kráčí, označuje si je a vypisuje, až když se po nich vrací. Ověřme si, že hrany opravdu navazují.

V duchu argumentů z předcházející části víme, že jediný vrchol grafu s lichým počtem nepoužitých hran je právě ona křižovatka – a algoritmus DFS prochází graf podobně, jako jsme ho procházeli v minulé sekci, takže právě do tohoto vrcholu algoritmus dojde, až se průchod touto částí grafu zastaví. Jakmile sem program dojde (a nezbudou mu volné hrany), začne cestovat zpět a hrany vypisovat – a opravdu, pokračuje se tedy z místa, kde naposledy přestal, a program vskutku vypíše tah přes všechny hrany v grafu – uzavřený eulerovský tah.

Věta o eulerovském tahu v celé své kráse tedy zní:

(Multi)graf obsahuje uzavřený eulerovský tah právě tehdy, když má všechny stupně sudé a je souvislý.

Je třeba podotknout, že složitost našeho algoritmu na bázi DFS je lineární vůči velikosti grafu (počtu vrcholů a hran). Existují i jiné algoritmy pro hledání eulerovského tahu, jedna varianta například prochází grafem a vybírá si na křižovatkách takové hrany, které souvislost grafu pokud možno nepoškodí. Tyto algoritmy už nemusí mít nutně lineární časovou složitost.

Eulerovskými tahy jsme se také zabývali v autorském řešení úlohy 10-3-1 ...

Jiné druhy procházek

Nejen kreslením obrázků ze stejného bodu živ je člověk. Co kdybychom mohli začít a skončit v jiném místě, tedy ptali se po neuzavřených eulerovských tazích, změnilo by se něco? Není tomu tak, pouze nutné a postačující podmínky si vyžádají, aby všechny vrcholy měly sudý stupeň až na právě dva vrcholy, které mají lichý stupeň. Pokud nám to nevěříte, zkuste si to rozmyslet sami, opravdu to není těžké.

Smysl také dává zkusit najít ne uzavřený tah, ale uzavřenou cestu – uzavřenou cestu přes všechny vrcholy, která navštíví každý vrchol právě jednou. Bohužel, ačkoli jsou problémy příbuzné, musíme vás zklamat – není znám žádný efektivní (polynomiální) algoritmus na tento problém, a kdyby jej někdo z vás nalezl, vyřešil by otázku „P vs. NP“ a získal alespoň milion dolarů. Chcete-li si o tomto problému přečíst něco dalšího, napište do vyhledávače „Hamiltonovská cesta“ – tak se ona úloha jmenuje.

V matematice se také někdy zmiňují „náhodné procházky“ po grafech – můžete si je představit tak, že se po mostech města Královce motá opilec, který si hází (opilou nebo spravedlivou) mincí a podle toho se rozhoduje, přes který most jít dál. Použití mají tyto modely hlavně v matematické teorii grafů a teorii pravděpodobnosti. O tom si můžeme povědět zase někdy jindy.

Dnešní menu uvařil a servíruje
Martin Böhm

Vzorová řešení první série

23-1-1 Básníkův deník

Podúloha najít nejvyšší místo, kde spal, splnila svůj účel: vyřešili jste ji všichni. Stačilo si jen počítat nadmořské výšky přičtením toho, co za den ušel, k výsledku včerejšího dne a při tom si v proměnné aktualizovat nejvyšší místo, kam došel. Času to zabere $\mathcal{O}(n)$ a stejně tak paměti (n je počet dnů, po které psal deník).

Nalezení nejčastějšího místa, kde přespal, šlo řešit různými způsoby. Nejoblíbenější byl pomocí třídění.

V setříděných nadmořských výškách už stačí najít tu, která je nejdelší. To jste zvládli při jednom projití. Průběžně si pamatovat, kolik stejných čísel za sebou bylo viděno, a srovnávat to se zatím nejdelším viděným úsekem. Třídění trvá lepšími algoritmy $\mathcal{O}(n \log n)$ a projití $\mathcal{O}(n)$. Celkově tedy $\mathcal{O}(n \log n)$.

Hodně z vás taky využívalo vyhledávacího stromu, někdy převlečeného za mapu či slovník, ale bylo podstatné si při tom uvědomit, že se o vyhledávací strom jedná. Složitosti při tom zůstávají stejné.

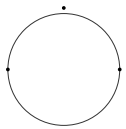
Vzorový program je na konci letáku.

Jitka Novotná

23-1-2 Jedna geometrická

Nejprve pár slov k došlým řešením. Mnozí z vás se u této úlohy pokoušeli hledat dva nejvzdálenější body a sestrojit nad nimi Thaletovu kružnici. To však obecně nefunguje, viz obrázek 1. Hledaný střed kruhu dokonce ani nemusí ležet na ose dvou nejvzdálenějších bodů (není tedy pravda, že oba tyto body leží na jeho obvodu) – protipříklad si zkuste vymyslet sami.

Obrázek 1: protipříklad na hledání 2 nejvzdálenějších bodů. Mezi bodem nahoře a oběma body na obvodu je menší vzdálenost než mezi samotnými body na obvodu.



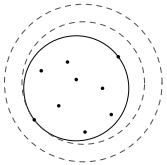
Pouze pár řešení fungovalo a z nich jen jedno pracovalo optimálně. Gratulace putuje k Jakubu Zíkovi, jenž nastudoval lineární algoritmus nezaložený na pravděpodobnosti (těžší na pochopení než zde prezentované řešení pracující lineárně jen v průměru) a pak ho popsal. Jak vidno, ne každá úloha s krátkým zadáním má i krátké a jednoduché řešení.

Jedná se však o problém starý (poprvé se jím zabýval anglický matematik Sylvester v roce 1857) a v praxi využitelný. Vezměte si například firmu, která má po zemi rozmístěné klienty a hledá místo pro své středisko tak, aby k němu žádný klient neměl moc daleko. Říká se mu také „problém bomby“ (chceme zjistit, kde odpálit výbušninu a jak má být velká, abychom zničili všechny cíle).

První pozorování

Nyní k tomu, jak se úloha řeší. Nejmenší kruh obsahující všechny body si označíme K. Při řešení úlohy se budou hodit následující pozorování:

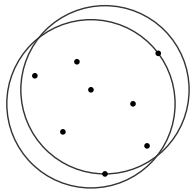
- Dostaneme-li na vstupu jen jeden bod, má kruh nulovou velikost, tento případ tedy nebudeme uvažovat.
- Na obvodu kruhu K leží minimálně dva body, jinak ho můžeme zmenšit (viz obrázek 2).



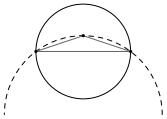
Obrázek 2: čárkované kruhy lze zmenšit, protože se nedotýkají dvou bodů.

- Kruh K je pro danou množinu bodů unikátní, tj. neexistují dva nejmenší kruhy obsahující všechny body (pokud by byly dva různé, jejich průnik obsahuje všechny body a zároveň se musí vejít do kruhu s menším poloměrem, takže tyto dva kruhy nejsou nejmenší, viz obrázek 3)

Obrázek 3: nechť jsou dva kruhy obsahující všechny body nejmenší, ale pak jejich průnik, jenž se dá obklopit ještě menším kruhem, obsahuje všechny body.



- Na určení kruhu nám stačí maximálně 3 body na jeho obvodu. Pokud na jeho obvodu leží pouze dva body, průměr kruhu je vzdálenost mezi nimi. Jestliže je kruh určen 3 body, musí tvořit ostroúhlý nebo pravoúhlý trojúhelník, jinak by mohl mít kruh průměr rovný vzdálenosti strany proti tupému úhlu a bod u tupého úhlu by neležel na obvodu (viz obrázek 4). Jinak řečeno, jsou-li na obvodu kruhu alespoň 3 body, musí se mezi nimi vyskytovat 3 netvořící tupoúhlý trojúhelník.



Obrázek 4: pokud 3 body tvoří tupoúhlý trojúhelník, není řešením opsat jim kružnici.

Pohled do ZOO algoritmů

Algoritmů řešících takovouto úlohu je více, zde si letmo představíme ty jednodušší a letmo ten „nejhustější“, ale kdo chce, ať rovnou přeskočí na sekci randomizovaný algoritmus, kde bude pořádně vysvětleno v průměru lineární řešení.

O kousek výše je v pozorováních zmíněno, že kruh K je určen 2 nebo 3 body. Co prostě vzít všechny dvojice a trojice bodů, opsat jim kružnici, zkontrolovat, jestli v ní leží všechny body, a vybrat nejmenší? To bude určite fungovat, jen časová složitost je nepěkných $\mathcal{O}(N^4)$.

Existuje celkem přímočaré (geometricky myšleno) řešení běžící v čase $\mathcal{O}(N^2)$. Skládá se z následujících kroků:

1. Na začátku vezměte nějaký kruh, který bude určitě obsahovat všechny body (je jedno jaký).
2. Najděte nejvzdálenější bod A od středu kruhu a zmenšete poloměr na vzdálenost mezi A a středem. Kruh se očividně zmenší a stále bude obsahovat všechny body.
3. Pokud na obvodu leží jen bod A, posunujte střed po přímce mezi A a středem směrem k A a zároveň zmenšujte jeho poloměr, aby A stále ležel na obvodu. Pokračujte, dokud se obvod kruhu nedotkne jiného bodu B.
4. Nyní tedy leží na obvodu minimálně 2 body. Dle našich pozorování potřebujeme zjistit, jestli jsou mezi nimi 3 tvořící ostroúhlý trojúhelník. Lze nahlédnout, že takové 3 body neexistují právě tehdy, když na obvodu lze najít část neobsahující body, která je delší než půlka obvodu (podívejte se na poslední obrázek u pozorování). Ta také může být vždy maximálně jedna.
5. Pokud tam taková část není, můžeme skončit. Jinak vezměme dva body na okrajích této části bez bodů (nazveme je D a E), zmenšujme poloměr kruhu a posouvejme střed tak, že D i E jsou stále na jeho obvodu. Mohou nastat dva případy:
   a) Průměr kruhu je vzdálenost mezi D a E: pak jsme nalezli nejmenší kruh obsahující všechny body.
   b) Na obvod kruhu se dostane bod F, máme tedy alespoň 3 body na obvodu a můžeme opět přejít na bod 4 (tj. zkusit najít část bez bodů delší než půlka obvodu a případně opět zmenšovat kruh).

Implementace tohoto geometrického postupu je trochu obtížná. Například zmíněné zmenšování kruhu bude opět hledání jistým způsobem nejvzdálenějšího bodu (přesněji řečeno třeba pro krok 2, pokud jsou dány dva body na obvodu a přímka, po níž se pohybuje střed, je třeba vypočítat, kde bude ležet střed, z toho se získají poloměry a vybere se ten největší).

Kroky 1, 2 a 3 zaberou lineární čas, samotný krok 4 také, ale může se stát až $(N-2)$-krát, že se bude krok 4 opakovat. Proto je časová složitost v nejhorším případě $\mathcal{O}(N^2)$.

Toto řešení bylo objeveno až v roce 1972 pány Elzingou a Hearnem, potom následovaly těsně po sobě nápady na první $\mathcal{O}(N \log N)$ algoritmy (Shamos a Hoey v r. 1975, Preparata v r. 1977 a Shamos v r. 1978).

Zajímavý algoritmus vychází z pozorování, že konvexní obal určuje hledaný nejmenší kruh. V čase $\mathcal{O}(N \log N)$ (resp. $\mathcal{O}(N)$, máme-li body seřazené), najdeme konvexní obal, jeho velikost budiž H, a na něj prostě pustíme kvadratický algoritmus, což dává složitost $\mathcal{O}(N \log N + H^2)$.

23-1-1 Program (Básníkův deník) C

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 42000

// porovnávací funkce pro quicksort (pointer black magic)
int compare(const void *x, const void *y) { return *(int*)x-*(int*)y; }

int main(void){

    int N;                  // počet dnů
    int vrcholy[MAX_N];     // nadmořské výšky tábořišť
    int max = 0;            // nejvyšší tábořiště
    int zac = 0;            // začátek posledního úseku stejných čísel
    int nej_mist;           // místo, kde spal nejčastěji
    int nej_kolik = 0;      // a kolikrát tam spal

    // vstup
    vrcholy[0] = 0;         // začal u moře
    scanf("%d",&N);
    for (int i=1; i<N; i++){
        scanf("%d",&vrcholy[i]);
    // rovnou zjístim přesnou nadmořskou výšku přičtením včerejší výšky
        vrcholy[i] += vrcholy[i-1];
        if (vrcholy[i] > max)
            max = vrcholy[i];
    }

    // třídění quicksortem
    qsort(vrcholy, N, sizeof(int), compare);

    // najdu nejdelší úsek stejných čísel
    for (int i=0; i<N;i++){
        if (vrcholy[i] != vrcholy[zac])   // v posloupnostech se změnilo číslo
            if (i-zac > nej_kolik){       // pokud je tento úsek delší
                nej_mist = vrcholy[zac];
                nej_kolik = i-zac;
                zac = i;
            }
    }

    // výstup
    printf("Nejvýše spal v: %d.\n",max);
    printf("Nejčastěji spal v: %d.\n",nej_mist);

    return 0;
}
```

vrcholů do našeho v. A samozřejmě, abychom mohli posléze zrekonstruovat cestu, si uložíme, který že to vrchol z vrstvy vzdálené $n-1$ byl pro náš v takto výhodný.

Bude to fungovat? Do daného vrcholu prostě musíme přijít z vrcholu v nejvýše vrstvě vzdálené $n-1$, chceme-li dodržovat hranovou vzdálenost coby úhlavní kritérium – a z vrstvy s menším pořadovým číslem nám do vrcholu ve vrstvě n samozřejmě žádná hrana vést nemůže. Pokud tedy věříme, že máme ceny v nižších vrstvách spočítány správně, budeme je mít dobře i ve spočítaném vrcholu. No a protože cena v počátečním vrcholu je dobře (0), roznese nám matematická indukce tuto správnost po všech vrcholech v grafu.

Samozřejmě nepotřebujeme všechny cesty, ale to už je ta potíž s algoritmy pro hledání nejkratší cesty z bodu A do bodu B, že toho většinou musí mimoděk spočítat o hodně víc. Časová složitost našeho řešení je každopádně $\mathcal{O}(n+m)$ a paměťová stejně tak.

Program (C):
`http://ksp.mff.cuni.cz/tasks/23/2316.c`

Lukáš Lánský

23-1-7 Regulární výrazy

Sešlo se nám přes 30 řešení různé kvality a přístupu. Bylo nelehkým úkolem je opravit a alespoň pseudospravedlivě obodovat, takže pokud vám bude připadat, že jsme zrovna k vám byli nespravedliví, tak se ozvěte e-mailem opravujícím, nebo třeba na fóru. Prostoru pro dotazy je dost, ty nejvíce očekávané se zde pokusíme zodpovědět rovnou.

Autorským řešením **úkolu 1** byl výraz `((b?a)*b)?` – ten přijímá opravdu stejné řetězce jako zadaný `b?(a+b)*` – za každým `b` musí nutně následovat alespoň jedno `a`, pokud tedy není na konci řetězce. Musí vyhovovat i prázdný řetězec, což bylo často opomínáno.

Řešení spočívající v náhradě `a+` za `aa*` nebo `b?` za `b{0,1}` jsme hodnotili stylově desetinou bodu. On je to totiž vlastně stejný výraz.

V řešení **úkolu 2** jste se mohli odvážit dál. Mnoho z vás zůstalo u výrazu `(a+|b+)*`, který šlo po krátkém rozmyslu zredukovat na `[ab]*`, což je také autorské řešení.

Úkol 3 byl poněkud šílený. Na něm jste si mohli vyzkoušet tvorbu rozsáhlých regexů, na kterých je poznat každá nesystematičnost, každá výjimka. Zde jsme strhávali body i za používání `(0|2|4|6|8)` místo `[02468]`. Ono to má stejný význam, akorát to první se čte výrazně hůř.

Mnoho řešitelů jednoduše vypsalo všechna koncová trojčíslí dělitelná 8. To je sice hezké, ale pomalé. Každý znak navíc je zpomalení. Porovnejte s autorským řešením (mezery a konce řádků ignorujte):

```
(0|-?(8|[48][08]|[159]6|[26]4|[37]2|
[2468]([048][08]|[159]6|[26]4|[37]2)|
([1-9][0-9]*)?[13579]
        ([048]4|[159]2|[26][08]|[37]6)|
[1-9][0-9]*[02468]
        ([048][08]|[159]6|[26]4|[37]2)))
```

Nelíbila se nám čísla, která začínala řadou nul, stejně tak drobné chyby jako neuvažování nuly nebo záporných čísel, nicméně jsme za ně strhávali výrazně méně než za false positives nebo false negatives.

Následovalo cvičení z exaktního vyjadřování. V **úkolu 4** bylo za úkol popsat, co danému výrazu vyhovuje. Obyčejný popis stylu „sudý počet nul, pak nula, nebo jednička, a potom sudý počet jedniček“ vyfasoval bod.

Nápaditější řešitelé, kteří napsali „sudý počet nul, pak lichý počet jedniček, nebo lichý počet nul a pak sudý počet jedniček“, získali body dva.

Hodí se zmínit, že nula je také sudé číslo. Mnoho z vás si to neuvědomilo a řešili nulu zvlášť.

Nakonec trochu přiblížení reality. **Úkol 5** vyžadoval opravu zadaného výrazu, což je nejčastější problém, se kterým se při práci s regexy setkáte. Zadanému regexu měly vyhovovat právě ty řetězce, ve kterých je sudý počet jedniček, sudý počet nul a nic jiného.

Zadaný výraz byl dost mimo. Jedna z možností, jak ho opravit, spočívala ve zhruba dvojnásobném natažení výrazu, neboť kromě bloku `0(00|11)*1(00|11)*` bylo potřeba ještě zahrnout blok `1(00|11)*0(00|11)*`. Lepší variantou bylo zadaný výraz zahodit a vymyslet úplně nový.

Má-li řetězec sestávat ze sudého počtu nul a sudého počtu jedniček, pak musí mít také celkem sudý počet znaků, tedy nám rozhodně nebude vadit, že jej budeme kontrolovat po dvojicích.

Prázdný řetězec rozhodně vyhovuje. Pokud nyní přečteme dvojici `(00|11)`, bude rozhodně vyhovovat taky. Naopak kdybychom měli řetězec, který nevyhovuje, tak po přečtení dvojice `(00|11)` vyhovovat také nebude. Tedy nás nezajímá, kdy, kde a v kolika exemplářích se nějaká tato dvojice objeví.

Přesně obráceně to platí pro dvojici `(01|10)`. Ta vždy přepne mezi vyhovujícím a nevyhovujícím řetězcem. Té tedy potřebujeme sudý počet. Po poskládání všech požadavků máme výraz `(00|11)*(((01|10)(00|11)*){2})*`

První část spolkne začátek sestávající z `(00|11)*`, další část vždycky přejde do stavu „nevyhovující řetězec“, spolkne `(00|11)*`, přejde do stavu „vyhovující řetězec“ a zase spolkne `(00|11)*`. Jednoduché a účinné.

Jozef Gandžala & Jan „Moskyto“ Matějka

Až v roce 1983 vymyslel Nimrod Megiddo k překvapení všech lineární algoritmus založený na metodě prořezávej a hledej (anglicky *prune and search*). Podstatou algoritmu je na základě několika geometrických triků odstranit v lineárním čase $n/16$ bodů bez změny nejmenšího kruhu obsahujícího všechny body.

Na zbylých $15n/16$ bodů je pušten algoritmus znovu a tak dál, dokud nezbyde jen celkem málo bodů (např. 15), pro než lze úlohu rychle vyřešit i kvadratickým algoritmem. Vtip je v tom, že složitost jednotlivých kroků algoritmu se posčítá díky vlastnostem geometrické řady na lineární složitost, přesněji řečeno: $n + 15n/16 + 225n/256 + \ldots = 16n$. Jelikož úplné vysvětlení by zabralo pěkných pár stránek, raději si přečtěte původní anglický článek. [7]

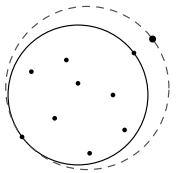
Randomizovaný algoritmus

Jak jsme slíbili, teď předvedeme randomizovaný algoritmus (randomizovaný znamená založený na náhodě, v tomto případě náhoda ovlivňuje časovou složitost), běžící v průměru lineárně. Vymyslel ho Welzl v roce 1991. Ten začne s 2 body, jimž opíše kružnici, a poté postupně přidává bod po bodu a upravuje nejmenší kruh K obsahující všechny dosud přidané body, je-li to nutné. Náhodné pořadí přidávaných bodů zajistí onu lineární složitost, jak později ukážeme.

Na začátku je tedy vhodné náhodně uspořádat body v čase $\mathcal{O}(N)$, aby „zlý“ uživatel nezadal pořadí, na němž program poběží pomalu (třeba i body setříděné dle souřadnice x způsobí pomalý průběh, jak se za chvíli ukáže). Tohle můžeme udělat například tak, že vybereme náhodný prvek z pole (tedy vygenerujeme číslo od 1 do N), ten prohodíme s posledním, pak vezmeme náhodný prvek, ale už jen od 1 do $N-1$, prohodíme s předposledním… A takto postupujeme, dokud nedojdeme na začátek.

Nyní přijde trocha geometrických hrátek s body. Začněme tedy prvními dvěma a opišme jim kruh, jež nazveme K_2. Obecně pak K_i bude nejmenší kruh obsahující body $1 \ldots i$. Co dělat, když přidáme i-tý bod a máme kružnici K_{i-1}? Pokud bod náhodou padne do kruhu K_{i-1} (nebo na jeho obvod), pak $K_i = K_{i-1}$, tedy kruh se nezměnil a můžeme pokračovat vesele dál.

Mnohem zajímavější je případ, kdy přidávaný bod leží mimo kruh K_{i-1}. Označme tento bod B_i. Je zřejmé, že B_i musí ležet na obvodu kruhu K_i, jinak by už ležel uvnitř K_{i-1} (neurčuje kruh, můžeme ho tedy vynechat beze změny kruhu). Takže nyní máme za úkol spočítat nejmenší kruh pro $i-1$ bodů s B_i na obvodu. A jak? Zavoláním stejné funkce pro $i-1$ bodů jen navíc s informací, že jistý bod má být na obvodu.



Obrázek 5: bod B_i leží mimo K_{i-1}, takže je třeba zvětšit kruh.

Naším řešením bude funkce, která v parametrech dostane množinu bodů M (ty, pro něž počítá nejmenší kruh) a seznam bodů, jež musí ležet na obvodu (body na obvodu nemusí být v množině M). Funkce nejprve zkontroluje, jestli už na obvodu nemusí ležet 3 body (pak je kruh jednoznačně určen a dopočítá se) nebo není M prázdná (v tom případě se kruh opíše bodům na obvodu, jsou-li nějaké). Poté se rekurzivně zavolá s množinou M o jeden bod B menší (to je ten přidávaný bod), uloží si vrácený kruh K a následně zjistí, zdali bod B leží v kruhu K nebo ne. Pokud ano, vrátí kruh K, jinak se rekurzivně zavolá s množinou M o bod B menší a s B na obvodu.

Kdo se v tomto odstavci ztratil, může se najít v následujícím pseudokódu (O je množina bodů na obvodu):

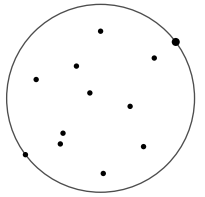
```
function nejmensiKruh(M, O) {
    if (|M| == 0 nebo |O| == 3) {
        Vrať kruh spočtený přímo z množiny O
    }
    Bod B = Vezmi náhodný bod z M
    Kruh K = nejmensiKruh(M - B, O)
    if (B neleží v K) {
        Přidej B do O
        Vrať nejmensiKruh(M - B, O)
    }
}
```

Náhodný výber z množiny, díky němuž už za chvíli získáme průměrnou lineární složitost, zajistíme náhodným seřazením pole. Pak prostě budeme brát poslední prvek.

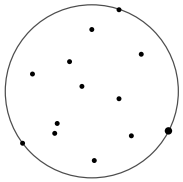
Tak a nyní k **časové složitosti**. Prostým pohledem na pseudokód by člověk řekl, že bude $\mathcal{O}(N^3)$ (pro každý přidaný bod spustíme rekurzivně tu samou funkci s jedním bodem na obvodu navíc), což je také nejhorší možný případ. Jenže nás teď zajímá průměrná časová složitost, k níž nám dopomůže náhodné seřazení bodů.

První rekurzivní volání `minimalniKruh(M - B, O)` teď budeme tiše ignorovat (ono se totiž provede vždy) a budeme předpokládat, že body postupně přidáváme. Zajímá nás tedy, jaká je pravděpodobnost, že se s novým přidaným bodem B zavolá funkce rekurzivně s B na obvodu navíc. Uvažujme nejmenší kruh K obsahující už bod B. Všimněte si, že rekurzivní volání při přidávání bodu B je podobné zmenšení kruhu K po odebrání bodu B, tedy pravděpodobnost „drahého“ rekurzivního volání je stejná jako pravděpodobnost, že se kruh K po odebrání bodu B zmenší.

Někde vysoko nahoře v tomto textu jsem zmínil, že nejmenší kruh je určen 2 nebo 3 body. Hledaná pravděpodobnost při přidávání i-tého bodu tak vyjde $2/i$ nebo $3/i$ (pro jednoduchost budeme dále uvažovat jen $3/i$, není těžké si rozmyslet, že pro $2/i$ vše vyjde stejně).



Obrázek 6: kruh se zmenší právě tehdy, když odebereme jeden ze dvou konkrétních bodů.



Obrázek 7: To samé pro 3 body na obvodu, jež tvoří ostroúhlý trojúhelník.

[7] `http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/Center/centercli.htm`
N. Megiddo, Linear-Time Algorithms for Linear Programming in $\mathbb{R}^3$ and Related Problems, SIAM Journal on Computing, Vol. 12, 759–776, dostupné na `http://www-ma2.upc.es/~geoc/m-lalparp-83.pdf`.

Dále budeme rozebírat časovou složitost podle počtu bodů, jež musí být na obvodu. Pro 3 je to triviálně $\mathcal{O}(1)$, takže začněme se 2 body na obvodu. Rekurzivní volání algoritmu už nás stojí pouze $\mathcal{O}(1)$ (na obvodu musí být 3 body) a počet bodů na obvodu se nikdy nezmenší, takže N bodů se dvěma danými body na obvodu zvládne algoritmus vždy v $\mathcal{O}(N)$.

Zajímavější je situace, je-li dán jen jeden bod na obvodu. Nyní využijeme před chvílí spočtenou pravděpodobnost (zde $2/i$ a ne $3/i$, protože máme jeden bod předem daný na obvodu), a tak můžeme napsat časovou složitost po přidání i-tého bodu takto:

$$\frac{i-2}{i}\,\mathcal{O}(1) + \frac{2}{i}\,\mathcal{O}(i) = \mathcal{O}(1)$$

Pro N bodů s jedním daným na obvodu se časová složitost posčítá na $\mathcal{O}(N)$. A co N daných bodů a žádný, který by byl určitě na obvodu? To je přeci naše původní úloha. Znovu využijeme pravděpodobnost, takže na přidání i-tého bodu spotřebujeme:

$$\frac{i-3}{i}\,\mathcal{O}(1) + \frac{3}{i}\,\mathcal{O}(i) = \mathcal{O}(1)$$

Při součtu ještě přidáme počáteční náhodné zamíchání pole bodů:

$$\mathcal{O}(N) + \sum_{i=1}^{N} \mathcal{O}(1) = \mathcal{O}(N).$$

Sláva! Tak máme dokázánu i časovou složitost. Paměťová je zjevně vždy lineární.

A poučení do příště? U některých úloh, jestliže si s nimi lámete hlavu už docela dlouho, se vyplatí zeptat se vyhledávače, zdali nezná řešení, jež pak případně přečtete, pochopíte a popíšete vlastními slovy.

Program (C++):
`http://ksp.mff.cuni.cz/tasks/23/2312.cpp`

Pavel Veselý

23-1-3 Jedna maticová

K této úloze nám došla spousta řešení, téměř každé fungovalo, ale problém byl ve složitosti. Některá řešení byla příliš pomalá, u jiných byl problém se špatně určenou složitostí. Nezapomínejte, že pro dobré hodnocení je potřeba mít správný a srozumitelný popis vašeho algoritmu a také správnou časovou a prostorovou složitost.

První řešení spočívá v prohledání celé matice řádek po řádku a kontrole každého prvku. Takové řešení samozřejmě funguje, dokonce funguje i pro obecné matice. A to je právě kámen úrazu.

Protože toto řešení nevyužívá vlastností matice, musí se podívat na každý prvek. Jeho složitost je tedy $\mathcal{O}(n \cdot m)$ pro matici velikosti $n \times m$. To ani zdaleka není to, co bychom chtěli a za co bychom byli ochotni dát celých 11 bodů.

Někteří si uvědomili, že když je posloupnost čísel v řádku ostře rostoucí, dalo by se využít binární vyhledávání. A tak jde zlepšit složitost z $\mathcal{O}(n \cdot m)$ na $\mathcal{O}(n \log m)$. Ale věřte tomu nebo ne, ani to nám nestačí.

Když nestačí použít na každém řádku binární vyhledávání, co ještě provést? Správné řešení používá binární vyhledávání na hlavní diagonále matice (tak se říká úhlopříčce vedoucí doprava dolů). Před uvedením algoritmu si musíme uvědomit, že platí dvě důležité věci:

- Pokud je v matici A na indexech i, j (označíme jako $A_{i,j}$) prvek, jehož hodnota je menší než $i+j$ ($A_{i,j} < i+j$), víme z uspořádání prvků v řádcích a sloupcích, že jsou menší i všechny prvky v matici, jejichž souřadnice jsou menší než i a j ($\forall k \le i, l \le j : A_{k,l} < k+l$).

  $A_{i,j}$ je alespoň o jedna menší než $i+j$, tedy i např. $A_{i-1,j}$ musí být alespoň o jedna menší než $A_{i,j}$, což znamená, že je alespoň o jedna menší než $i-1+j$. A takto tranzitivně dále.
- Pokud platí $A_{i,j} > i+j$, pak $\forall k \ge i, l \ge j : A_{k,l} > k+l$. Opět platí obdobně, $A_{i,j}$ je alespoň o jedna větší než $i+j$, takže i všechny následující prvky musí být alespoň o jedna vychýleny.

Z těchto dvou pozorování plyne, že pokud se podíváme na prvek uprostřed matice, tak mohou nastat tři možnosti. Mohli jsme narazit na správný prvek. To znamená, že můžeme skončit. Nebo je nalezený prvek větší než součet jeho souřadnic, pak můžeme zapomenout pravou dolní čtvrtinu matice, případně je prvek menší a zapomeneme levou horní čtvrtinu matice.

Takže budeme provádět binární vyhledávání na hlavní diagonále, buď najdeme správné řešení, nebo nám nakonec zůstane jen pravá horní a levá dolní čtvrtina matice. Na ty zavoláme rekurzivně stejný algoritmus. Právě tento způsob je použit ve vzorovém kódu.

Toto řešení nám přišlo několikrát, ovšem pouze jednou u něj byla uvedena správná časová složitost. Pojďme si ji tedy rozebrat detailně. Čas potřebný pro nalezení řešení je definován rekurzivně: $T(n^2) = 2T(n^2/4) + \log_2 n$ (pro jednoduchost předpokládáme čtvercovou matici).

Každý správný programátor je hlavně hrozný lenoch, využijeme tedy kuchařkovou metodu pro počítání složitosti rekurzivních algoritmů. Ta se jmenuje Master Theorem a řeší rekurzivní vztahy ve tvaru $T(N) = aT(N/b) + f(N)$, kde $a \ge 1, b > 1$. Dále tvrdí, že pokud $f(N) = \mathcal{O}(N^{\log_b(a)-\varepsilon})$ pro nějaké $\varepsilon > 0$, tak $T(N) = \Theta(N^{\log_b a})$.

Pro naši rekurenci tohle všechno platí:

$$a = 2, b = 4, \log_2 n = \mathcal{O}(n^{2\log_4 2-\varepsilon}),$$

takže výsledná složitost je $\Theta(n)$. Prostorová složitost je logaritmická, protože používáme zásobník.

Existuje i jednodušší řešení, které také vede k cíli. Pro něj si stačí uvědomit, že pokud se podíváme na prvek v levém dolním rohu, tak buď jsme našli správné řešení, nebo je větší než součet souřadnic, pak můžeme zahodit celý poslední řádek, nebo je menší než součet souřadnic a můžeme zahodit celý první sloupec. Nakonec se posuneme buď nahoru nebo doprava, podle toho, čeho jsme se zbavili, a pokračujeme stejně.

Takto se v každém kroku zbavíme buď celého sloupce, nebo řádku. V nejhorším případě tedy provedeme $\mathcal{O}(n+m)$ operací. Prostorová složitost je zde konstantní.

Pokud bychom chtěli najít všechny prvky matice, které odpovídají zadání, tak je snadné uvedené dva algoritmy upravit, víme totiž, že pokud najdeme jedno řešení, budou s ním další sousedit, nebo budou v zatím neprozkoumané části matice.

Program (C):
`http://ksp.mff.cuni.cz/tasks/23/2313.c`

David Marek & Karel Tesař

Vida, to je zvláštní druh algoritmu – rychlejší než lineární ve velikosti vstupu (ta je $m \times n$), protože si ani celý vstup nemusí přečíst. Také vám vrtá hlavou, jestli by nestačilo si ze vstupu přečíst ještě méně? Pojďme dokázat, že nestačilo.

Nejdřív si úlohu převedeme na jinou, ekvivalentní, aby se nám o ní snáze přemýšlelo. Místo zadané matice budeme uvažovat stejně velkou matici $B_{i,j} = A_{i,j} - i - j$. Jelikož A byla v řádcích i sloupcích rostoucí, B bude alespoň neklesající (rozmyslete si, proč). A hledané políčko $A_{i,j} = i+j$ odpovídá políčku $B_{i,j} = 0$. Pokud tedy umíme vyřešit původní úlohu, dokážeme vyřešit i tuto, a naopak.

Nyní uvažujme matici B, která bude mít na hlavní diagonále a nad ní hodnoty $+1$ a pod diagonálou samé -1. To je neklesající matice, v níž žádné nulové políčko neexistuje. Kdykoliv ale změníme některou z $+1$ na diagonále na 0, matice bude pořád neklesající, ale řešení v ní už bude existovat:

$$\begin{pmatrix} +1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & +1 & +1 \\ -1 & -1 & +1 & +1 & +1 \\ -1 & -1 & -1 & 0 & +1 \\ -1 & -1 & -1 & -1 & +1 \end{pmatrix}$$

Pokud tedy libovolný algoritmus řešící úlohu spustíme na naši matici B, musí přečíst alespoň všechna políčka na diagonále, aby si ověřil, že v matici žádné nulové políčko není.

Martin M. Mareš

23-1-4 Ale co trapné numerické chyby?

Na této úloze nebylo mnoho těžkého, a tak spousta řešitelů dostala zasloužených 10 bodů. Blahopřejeme, příště už to tak snadné nebude!

Přejděme k úloze samotné. Periodu racionálního čísla nelze poznat jen tak, že se v desetinném zápisu opakuje řetězec (začátek 0.88 neznamená periodu 8, například u 15/17). Když dělíme čitatel a jmenovatel na papíře, poznáme periodu tak, že se „zacyklíme“ – dělíme už jednou to samé číslo, ten samý zbytek. Tak proč to tak neimplementovat? Zbytky po dělení jmenovatele nám budou sloužit jako odkazy do pole, uvnitř pole si zapamatujeme první výskyt odkazovaného zbytku – abychom věděli, kde zapsat závorku. Hotovo!

Poznamenejme, že paměťová složitost je $\mathcal{O}(N)$, kde N je velikost jmenovatele, tedy počet možných zbytků, a časová je lineární vůči velikosti výstupu. (V některých případech je pro dokázání optimality užitečnější měřit časovou složitost nikoli podle vstupu, ale podle velikosti výstupu. Nakonec, i kdybychom uměli dělit rychleji než na papíře, stejně musíme výstup vypsat.)

Vzorový program je na konci letáku.

Martin Böhm & CodEx

23-1-5 Adina knihovna

Očíslujme si N knih po řadě zleva doprava 1 až N. Podívejme se na knihu s číslem 1, která je jistě na kraji. Lze ji přesunout na jediné místo, a to na pozici $1+K$. Dalším pohledem zjistíme, že knihu z pozice $1+K$ musíme přesunout na pozici 1, protože jinou tam dát nesmíme.

Podívejme se na knihu s číslem $\check{c} \le K$. Lze ji přesunout na jediné místo, a to na pozici $\check{c}+K$, odkud přesuneme knihu na pozici $\check{c}$.

Tedy prvních $2K$ knih povyměňujeme mezi sebou a zbyde nám $N-2K$ knih, na které můžu použít stejný argument. Tohle opakujeme i kroků, až nám zbyde $0 \le N-2iK < 2K$. Buďto platí, že $N-2iK = 0$, pak jsme hotovi (a tedy platí, že $2K$ dělí N, protože $N/2K = i \in \mathbb{N}$). Nebo máme nenulový zbytek, ale v tom jistě umíme najít knihu, kterou neumíme přesunout ani vlevo, ani vpravo (třeba tu úplně uprostřed), tedy knihovnu s takovým K nelze přeskládat.

Zbývá tedy první varianta, a tedy bereme pouze taková K, která dělí $N/2$ (pro lichá N úloha nemá řešení). Zjevně je jen jeden způsob, jak knihy přeskládat, což byla druhá věc, na kterou se zadání ptalo.

Někteří řešitelé ještě uvažovali triviální případ, kdy $K = 0$, to funguje pro všechna N (i lichá). Někdo také řešil možnost $K < 0$. To bylo možné, i když jsme to nijak extra nehodnotili.

Jan „Moskyto“ Matějka & Pali Rohár

23-1-6 Babbageova cesta

Píšeme-li v zadání „*Pro jednoduchost* předpokládejme, že použití takového spojení trvá jednotkový čas.“, můžeme tím myslet různé věci. Takové omezení zjednodušuje popisování zadání, zjednodušuje načítání vstupu…

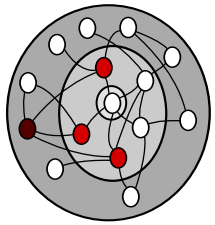
Může se stát, že bychom zjednodušovali práci procesoru, že by asymptotická časová složitost řešení s takto omezeným zadáním byla nižší, než složitost řešení, kde by použití spojení zabralo zadaně vteřin?

Je to tak a většinu z vás jsme na to nachytali. Použití Dijkstrova algoritmu je v daném případě triviálně možné: stačí měřit vzdálenost v uspořádané dvojici (počet použitých hran, součet cen na použitých hranách), kde při porovnávání klademe důraz na druhé složky pouze v případě rovnosti prvních složek.

Do časové složitosti takového řešení se však nevyhnutelně vloudí logaritmy, které tam zaneslo použití haldy coby rozumné implementace prioritní fronty, kterou Dijkstrův algoritmus prostě potřebuje.

Poodstoupíme o krok zpátky: dokud jsme nevěděli, co to Dijkstrův algoritmus je, uměli jsme měřit nejkratší cesty pouze co se počtu hran týče, a to prohledáváním do šířky.

To nám přirozeně rozdělí vrcholy do vrstev podle vzdálenosti od vrcholu, ze kterého jsme prohledávat začali, stačí si k vrcholu tuto vzdálenost připsat (výchozímu vrcholu nastavit nulu) a při vkládání nezpracovaných vrcholů do fronty jim ji přidělovat o jednotku zvýšenou.



Naše úloha je složitější o to, že druhotné kritérium v zadání mluví o ohodnocení hran. S tím se ale vyrovnáme snadno lehkou úpravou prohledávání do šířky: kdykoliv dostaneme z fronty vrchol v s přiřazenou hranovou vzdáleností n, rozhlédneme se po sousedních vrcholech (tj. těch, se kterými v spojuje hrana), vybereme jen ty, které jsou ve vrstvě vzdálené $n-1$ (od výchozího bodu), a vrcholu v nastavíme coby minimální cenu minimum ze součtu cen vrcholů z této vrstvy a příslušných cen přepravy (ohodnocení hran) z těchto