Korespondenční Seminář z Programování

31. ročník **KSP** **Květen 2019**

Milí řešitelé, řešitelky a řešitelčata!

Právě držíte v rukou leták s řešeními úloh čtvrté série. Pojďte se podívat, jak se daly řešit úlohy, které jsme si na vás vymysleli.

Připomínáme, že od letoška jsou řešení každé série rozdělena na dvě části: na samotná autorská řešení, která vydáváme brzy po termínu série a jejichž třetí várku najdete v tomto letáku, a na komentáře k došlým řešením, která vydáváme až po opravení vašich řešení.

Pokud se vám cokoliv nezdá nebo máte nějaký dotaz, neváhejte se ozvat na našem fóru nebo emailem na známou adresu.

Vzorová řešení čtvrté série třicátého prvního ročníku KSP

31-4-1 Kouzelné zrcadlo

Posloupnosti osob vlastně reprezentují pole seřazené podle nějakého kritéria.

Protože jsou posloupnosti setříděné, dá se aplikovat slévací princip z Mergesortu (pro osvěžení paměti se můžete podívat do naší kuchařky o třídění).[1] Je ale zbytečné slévat celé seznamy, protože se stačí zastavit po určení k-té osoby. A navíc není třeba nikam ukládat celý slitý seznam.

Co se tedy dá udělat, je začít porovnávat osoby ze začátku obou posloupností. Pak se v seznamu s krásnější osobou v porovnávané dvojici budeme opakovaně posunovat na následující osobu. Tímto způsobem se po $k-1$ krocích dosáhne dvojice takové, že o obou osobách v ní umíme říci, že je krásnější než $k-1$ jiných osob a ve své posloupnosti je první taková. Z této dvojice vybereme krásnější osobu.

Toto není zrovna příliš efektivní, protože se provede okolo k porovnání.

Se setříděnými seznamy se dá dělat ale ještě něco: Binárně v nich vyhledávat. Co kdyby se pomocí tohoto zefektivnil předchozí přístup, když nás nezajímá vzájemné pořadí prvních $k-1$ nejkrásnějších?

Srovnáme-li i-té osoby obou posloupností (označme je X a Y), ukážeme o té krásnější z nich, že patří mezi prvních $2i-1$, stejně jako všechny osoby v posloupnosti před ní. Bude-li například krásnější X než Y, víme, že X je krásnější než všechny osoby za X ve stejném seznamu, než Y a než všechny osoby za Y. Osob před X je jen $i-1$ a osob před Y také jen $i-1$. X je tedy v kráse na pozici nejvýše $2i-1$. Osoby v seznamu před X jsou krásnější než X, proto je jejich pozice ještě lepší. Tedy je-li $2i-1 < k$, prvními i osobami jedné posloupnosti už se nemusíme dále zaobírat. Můžeme si pak představit, že taková posloupnost vlastně začíná až indexem $i+1$.

Takto získáme rekurzivní algoritmus: Chceme-li k-tou osobu, porovnáním i-tých osob vyřadíme prvních i v jedné posloupnosti a pak budeme rekurzivně řešit problém nalezení $(k-i)$-té osoby. Při volbě i jako přibližně jedné poloviny k se v každé fázi rekurze podaří snížit k na půl. Získáme potom složitost $\mathcal{O}(\log k)$.

Rekurzivní volání tedy dostává tři parametry: kolik osob bylo vyřazeno ze začátku obou posloupností a pořadí hledané osoby. Vyhledávání i-té osoby ve zkráceném seznamu se provede jako vyhledání pozice i plus počet vyřazených osob ze seznamu, a to v původním seznamu bez vyřazení.

Ještě zbývá vyřešit, co dělat, pokud některý seznam není dost dlouhý, čili zbývá v něm méně než $k/2$ prvků. (Zřejmě pokud v obou zbývá méně než $k/2$ prvků, řešení neexistuje.) V takovém případě porovnáme prvek v polovině krátkého seznamu s $(k/2)$-tým prvkem v druhém seznamu. Tak zkrátíme buď jeden seznam na polovinu, nebo snížíme k na polovinu. Pokud se jeden seznam podaří zkrátit na 0 prvků, v druhém už výsledek najdeme pouze podíváním se na jeden index. Každopádně po $3 \log k$ operacích by musely být oba seznamy prázdné a $k = 0$, tudíž více porovnání určitě provedeno nebude.

Zajímavé může být, že na celkovém počtu osob složitost algoritmu vůbec nezávisí.

Jiří Škrobánek

31-4-2 Stromy na mýtince

Jak poznáme, že jsou dva stromy izomorfní? Podle zadání musí mít oba stejný tvar včetně zachování pořadí jednotlivých synů vrcholu. Mohli bychom tedy projít oba stromy současně do hloubky a celou dobu kontrolovat, že se – až na čísla vrcholů – v ničem neliší.

Jenže jedno takové porovnání dvou stromů T_i a T_j nám zabere čas $\mathcal{O}(\min(N_i, N_j))$, kde N_k je počet vrcholů stromu T_k. My však máme na vstupu S stromů a v krajním případě, kdy žádné dva nejsou stejné, bychom museli porovnávat každý s každým. Pokud by všechny stromy byly přibližně stejně velké a obsahovaly by řádově N vrcholů, dostaneme tím řešení se složitostí $\mathcal{O}(S^2N)$.

Může nás napadnout, že to asi nebude nejrychlejší způsob. Každý strom projdeme až $S-1$ krát, ale přitom všechny průchody jednoho stromu budou probíhat vždy naprosto stejně. Algoritmus by tak trávil čas opakovaně tou samou činností.

Zkusme udělat krok stranou. Místo toho, abychom strom procházeli vždy znovu, jej projdeme jen jednou a vše důležité o průchodu si zaznamenáme do jednoho textového řetězce. Tento řetězec pak bude jednoznačně kódovat tvar stromu. Pokud nás pak bude zajímat, zda jsou dva stromy izomorfní, bude nám stačit porovnat jen tyto řetězce.

Možná se vám teď zdá, že jsme si přece nijak nepomohli. Takový řetězec bude mít délku řádově stejnou jako počet

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

vrcholů původního stromu. Takže porovnání dvou řetězců bude trvat stejně dlouho, jako porovnání dvou stromů. To je sice pravda, ale řetězce mají jednodušší strukturu oproti původním stromům, takže je budeme schopni porovnávat všechny najednou, a tím i efektivněji.

Kódování stromu řetězcem

Jaké jsou naše požadavky na kódování stromů? Potřebujeme, aby všechny izomorfní stromy byly zákódovány pomocí stejného řetězce. A naopak, aby jeden řetězec nemohl kódovat dva neizomorfní stromy. Druhou podmínku splníme snadno tím, že bude existovat jednoznačný způsob, jak z řetězce sestavit původní tvar stromu.

Funkčních kódování je celá řada. Například můžeme projít strom do hloubky (s tím, že všechny syny projdeme vždy ve správném pořadí zleva doprava) a cestou si značit kompletně vše, co potkáme. Jen toho nesmíme zaznamenávat příliš – potřebujeme, aby délka řetězce byla lineární. Pokud bychom do řetězce ukládali čísla, může se nám snadno stát, že bude mít délku řádově $N \log N$.

My si v této úloze vystačíme s velmi jednoduchou reprezentací: Budeme si značit jen každý přesun po hraně – třeba písmenem `D` vždy, když půjdeme dolů směrem k listům, a `N`, když se budeme vracet.

Například strom T_1 ze zadání by byl kódovaný řetězcem `DDNDNDNNDN`, T_2 `DNDDNDNDNN` a stromy T_3 a T_4 `DDNNDN`.

Řetězce kódují stromy jednoznačně; schválně si zkuste nějaký zrekonstruovat. Pokud je v řetězci znak `D`, vytvoříme novou hranu (v době vzniku nejpravější) z aktuálního vrcholu a přesuneme se po ní do nového vrcholu. Naopak znak `N` nás přesune o jednu hranu směrem k otci, který je v zakořeněném stromě určen jednoznačně.

Ještě poznamenejme, že toto kódování zaznamenává pouze hrany, a neumí proto odlišit graf prázdný (bez vrcholu) a jednovrcholový (samostatný kořen). Pokud má naše řešení správně fungovat i na prázdné grafy, musíme toto nějak ošetřit.

Například si ke každému stromu můžeme přimyslet ještě „nadkořen“ – nový vrchol, ze kterého teprve povede hrana do opravdového kořene původního stromu. Původní jednovrcholový strom by potom byl reprezentován pomocí řetězce `DN` a prázdný graf by měl prázdný řetězec. A všem ostatním stromům by se na začátek řetězce přidalo `D` a na konec `N`.

Každý strom T_i zvládneme projít a zakódovat v čase $\mathcal{O}(N_i)$ a vygenerujeme pro něj řetězec dlouhý $2N_i$ znaků.

Dělení řetězců na hromádky

Zbývá nám vyřešit poslední část: jak rozdělit všechny řetězce na hromádky? Pokud jste v nedávné době četli naši kuchařku o hledání v textu,[2] určitě si na funkční řešení vzpomenete. Použijeme šikovnou datovou strukturu zvanou trie, neboli „písmenkový strom“.

Jednotlivé řetězce jsou v trii reprezentovány pomocí cest z kořene dolů, přičemž v každém patře se strom větví podle toho, jaký znak v řetězcích následuje. V našem případě máme jen dva znaky `N` a `D`, takže trie bude binární zakořeněný strom s odlišeným pravým a levým synem.

Každý řetězec do trie vložíme snadno v čase lineárním s jeho délkou. Začneme slovo v trii hledat procházením od kořene a pokud některý vrchol ještě nebude existovat, jednoduše si jej vytvoříme. Na koncích řetězců si chceme poznačit, které řetězce v daném vrcholu trie končí. Protože jich může končit více, můžeme použít například spojové seznamy.

Po přidání všech řetězců do trie budou jednotlivé spojové seznamy odpovídat právě hledaným hromádkám.

Celková časová složitost bude lineární se součtem velikostí všech stromů na vstupu, tedy pro podobně velké stromy s N vrcholy dostaneme $\mathcal{O}(SN)$, takže jsme původní řešení zlepšili. Obecně bychom složitost mohli zapsat jako $\mathcal{O}(\sum_{i=1}^{S} N_i)$.

Na závěr poznamenejme, že místo trie bychom mohli použít také hešování. Pak bychom ale měli stejnou časovou složitost slíbenou jen v průměrném případě a museli bychom si dát práci se správným výběrem hešovací funkce.

Program (C):
`http://ksp.mff.cuni.cz/viz/31-4-2.c`

Jenda Hadrava

31-4-3 Nejvíc spánku

V úloze chceme vybrat z několika intervalů takovou podmnožinu, ve které je součet délek intervalů co největší, ale zároveň se žádné nepřekrývají. Nabízí se nám grafový pohled na úlohu: každý začátek či konec intervalu je vrchol, který si můžeme představit jako bod na časové ose, každý interval je orientovaná hrana mezi jeho krajními vrcholy ohodnocená dobou, kterou Sněhurka v tomto intervalu naspí, tedy jeho délkou. Protože vybraná množina může být taková, že na sebe intervaly nemusí nutně navazovat, vytvoříme také hrany mezi sousedními vrcholy na časové ose, které ohodnotíme hodnotou 0. Tyto hrany reprezentují dobu, kdy je Sněhurka vzhůru a čeká na uvolnění nějaké postele.

V takovém grafu bude řešení úlohy odpovídat nejdelší možné cestě mezi počátečním a koncovým vrcholem, což budou vrcholy reprezentující nějaký nejlevější levý a nejpravější pravý konec intervalu.

Hledání nejdelší cesty v obecném grafu je těžké, nicméně můžeme si všimnout, že náš graf nikdy nebude obsahovat cykly (Sněhurka neumí cestovat zpět časem), a jedná se tedy o orientovaný acyklický graf – DAG (directed acyclic graph). V takovém grafu už nejdelší cestu najít umíme.

Hledání nejdelší cesty

Jako triviální algoritmus se nám nabízí obyčejné rekurzivní prohledávání do hloubky, které spustíme z počátečního vrcholu. Toto řešení sice najde nejdelší cestu, ale provede to vyzkoušením všech možných cest, kterých může být až exponenciálně mnoho.

Všimněme si, že tato rekurze spoustu práce dělá zbytečně pořád dokola: pokaždé, když přijde k nějakému vrcholu, rekurzivně spočítá nejdelší cestu z tohoto vrcholu do koncového, i když tento výpočet zůstává stejný, ať už se rekurze k tomuto vrcholu dostala jakkoliv. Tuto hodnotu si tedy můžeme u každého vrcholu pamatovat. Při prvním dotazu na nejdelší cestu z daného vrcholu do cíle ji spočítáme, při každém příštím dotazu už rovnou vrátíme tuto hodnotu.

A aby se nám algoritmus lépe analyzoval, provedeme ještě jeden trik – úplně se zbavíme rekurze. Jelikož se jedná o DAG, všechny hrany, po kterých se můžeme z nějakého

[2] `http://ksp.mff.cuni.cz/viz/kucharky/hledani-v-textu`

vrcholu při rekurzi vydat, vedou jen dopředu. Pokud budeme vrcholy vyhodnocovat od posledního k prvnímu, tak všechny vrcholy, na které se musíme podívat pro spočítání nejdelší cesty z nějakého konkrétního vrcholu, už budou mít svou hodnotu připravenou a nikdy nebude potřeba na nich rekurzivně spouštět výpočet.

Jak to bude rychlé? Pro každý vrchol grafu něco provádíme a každou hranu zpracujeme právě jednou, pokud si tedy počet vrcholů grafu označíme v a počet hran e, dostáváme časovou složitost $\mathcal{O}(v+e)$. Pozor, tato složitost předpokládá, že už nám graf někdo dopředu připravil, což u této úlohy neplatí.

Stavba DAGu

Ještě ale musíme graf sestrojit, a to do podoby seznamů sousedů, kde si pro každý vrchol grafu pamatujeme, do kterých vrcholů z něj vede hrana a jak je ohodnocená.

Vrcholy grafu jsou pravé a levé konce intervalů, ty ale mohou mít všelijaké arbitrární hodnoty, takže abychom uměli rychle seznamy sousedů vytvořit, potřebujeme si krajní body intervalů nějak šikovně přečíslovat. Stačí, když je uspořádáme od nejlevějšího po nejpravější a jako číslo vrcholu použijeme index v tomto uspořádání. Toto uspořádání je potřeba také k tomu, abychom uměli vrcholy při hledání nejdelší cesty v DAGu projít ve správném pořadí, tedy od konce k začátku.

Pro každý interval na vstupu přidáme hranu z vrcholu odpovídajícího jeho levému konci, která povede do jeho pravého konce a bude ohodnocena jeho délkou. Čísla těchto vrcholů budou indexy v uspořádaném poli krajních bodů všech intervalů a můžeme je efektivně najít pomocí binárního vyhledávání či hešování.

Pro vytvoření grafu potřebujeme v čase $\mathcal{O}(n \log n)$ provést třídění a následně zpracovat všech n intervalů, kde pro každý provedeme dvě binární vyhledávání v $\mathcal{O}(\log n)$, a následně spustíme hledání nejdelší cesty, které seběhne v $\mathcal{O}(n)$, protože vrcholů i hran je řádově tolik, kolik je intervalů. Celková časová složitost je tedy $\mathcal{O}(n \log n)$.

Bez grafu

I když se nad grafem hezky přemýšlí, k řešení ho vůbec nemusíme konstruovat. Úloha lze řešit tak, že pro každý časový okamžik spočítáme, kolik nejvíce jednotek času může Sněhurka spát od počátku po daný okamžik. Počet těchto „časových okamžiků“ může být arbitrární a mnohem větší, než je velikost vstupu, všimneme si ale, že s touto hodnotou se bude něco dít pouze na hranicích intervalů ze vstupu.

Pro každý z těchto zajímavých okamžiků (kterých bude nejvýše $2n$) si budeme pamatovat, kolik nejvíce času lze do tohoto okamžiku spát. Počáteční hodnota této veličiny bude 0. Postupně okamžiky projdeme od nejdřívějšího. Pro každý se jednak podíváme, jestli jeho hodnota není menší než hodnota předchozího okamžiku, a pokud ano, tak jeho hodnotu hodnotou předchozího okamžiku nahradíme. Dále pro každý interval, který v tomto okamžiku začíná, zapíšeme do jeho koncového okamžiku součet délky intervalu a naspaného času pro aktuální okamžik, ovšem pouze tehdy, pokud je tento součet větší než hodnota, která v koncovém okamžiku už je.

Všimněme si, že toto alternativní řešení dělá něco velmi podobného jako grafové řešení. Stále potřebujeme setřídit hranice intervalů, tentokrát kvůli nalezení zajímavých časových okamžiků, které odpovídají vrcholům. Pro každý z okamžiků projdeme všechny intervaly, které v něm začínají, což odpovídá odchozím hranám DAGu, a počítání maxima hodnoty současného a předchozího okamžiku odpovídá hranám v DAGu, které spojují sousední vrcholy. Celková časová složitost je stále $\mathcal{O}(n \log n)$, indexy časových okamžiků v setřízeném poli opět hledáme binárním vyhledáváním.

Vlastně je to opět algoritmus, který v DAGu hledá nejdelší cestu, ovšem trochu jiný než ten, který jsme použili u grafového řešení.

Program (C):
`http://ksp.mff.cuni.cz/viz/31-4-3.c`

Kuba Pelc

31-4-4 Otrávené ovoce

Předně bychom se vám chtěli omluvit za nechtěný podraz v zadání. Problém s převážením ovoce v té podobě, jak jsme ho formulovali, není rozhodovací, neboť odpověď není Ano nebo Ne, nýbrž číslo označující minimální potřebný počet služebníků. Naštěstí ho ale na rozhodovací problém můžeme přirozeně převést: budeme odpovídat na otázku, zda pro dané k už k služebníků dokáže ovoce přepravit. Rozmyslete si, že původní a nový problém jsou v nějakém smyslu stejně těžké, protože pomocí polynomiálně mnoha volání řešení jednoho problému dokážeme vyřešit ten druhý a naopak (nápověda: k můžeme binárně vyhledat). Ve zbytku řešení se tedy budeme snažit dokázat, že i tato rozhodovací verze je $\mathcal{NP}$-úplná.

Zadání bylo nepřesné ještě v jedné věci: není jasné, zda máme od každého druhu ovoce jen jediný kus, nebo zda můžeme jeden druh dát do košíku několika služebníkům. Ukazuje se však, že na tom vůbec nezáleží: Pokud máme nějaký rozpis cest a rozdělení ovoce ve verzi s více kusy na druh, dokážeme z něj vyrobit řešení ve verzi s jedním kusem: spočítáme si pro každého služebníka, které kusy doopravdy donese do chaloupky a které mu cestou vezmou, a každý druh ovoce dáme libovolnému služebníkovi, který ho dokáže donést do chaloupky (a všem ostatním ho „vyndáme“). Obě verze jsou tedy stejně těžké. Ve zbytku řešení budeme pracovat s verzí s jedním kusem ovoce na druh.

Teď už k samotnému řešení: K tomu, aby byl náš problém $\mathcal{NP}$-úplný, je potřeba, aby ležel v $\mathcal{NP}$ a zároveň na něj byl převoditelný nějaký jiný $\mathcal{NP}$-úplný problém. První část dokážeme snadno: stačí jako certifikát zvolit přiřazení ovoce jednotlivých sluhů a popis jejich cest. Takový certifikát má jistě polynomiální velikost a také ho v polynomiálním čase zvládneme ověřit: stačí si pro každého sluhu ověřit, že mu cestou žádné ovoce neseberou.

Zbývá převést nějaký jiný $\mathcal{NP}$-úplný problém na problém přepravy ovoce. Popíšeme převod 3-obarvitelnosti grafu na náš problém, respektive dokonce k-obarvitelnosti, kde k je to samé k jako to, co jsme dostali na vstupu (můžete si rozmyslet, že to je také $\mathcal{NP}$-úplný problém). Existují však přímočaré převody i z dalších problémů, například ze 3-SATu.

Začneme jednoduchým pozorováním: máme-li nějaký graf G obarvený k barvami a vezmeme-li libovolnou množinu vrcholů stejné barvy, je tato množina nezávislá, tedy mezi žádnými dvěma vrcholy nevede hrana. A naopak, dokážeme-li G rozdělit na k nezávislých množin, umíme ho už snadno obarvit k barvami – jednoduše dáme všem vrcholům ze stejné množiny stejnou barvu.

Náš záměr bude následující: vytvoříme si pomocný graf P, který pak předhodíme naší černé skříňce schopné řešit problém přepravy ovoce. Vytvoříme jeden druh ovoce za každý vrchol grafu G. Naším cílem bude vyrobit takové P, aby ze startu do cíle šly přepravit pouze množiny ovoce odpovídající nezávislým množinám v původním grafu, tedy všechny množiny vrcholů, které můžeme obarvit stejnou barvou. Krabička nám pak spočítá nějaké rozdělení ovoce mezi co nejméně služebníků takové, že každý služebník do cíle dopraví množinu ovoce obarvitelného stejnou barvou. Jinými slovy nám spočítá právě obarvení původního grafu co nejméně barvami.

Jak ale sestavit P? Vytvoříme dlouhý řetěz, který bude postupně kontrolovat, zda služebník nenese dva kusy ovoce spojené hranou. Konkrétně bude sestávat z $M+1$ vrcholů, kde M je počet hran grafu G, a mezi každými sousedními vrcholy povede zleva doprava orientovaná dvojice hran (pokud se vám nelíbí, že máme mezi jednou dvojicí vrcholů více hran, můžete si představit, že hrany veprostřed rozdělíme pomocnými vrcholy). Jedna z hran bude povolovat průchod se všemi druhy ovoce kromě ovoce u, druhá bude obdobně povolovat všechny druhy ovoce kromě v, kde u a v jsou nějaké dva vrcholy spojené v G hranou. Tak donutíme služebníka, aby si jednu z hran vybral, a tudíž jeden ze dvou kusů ovoce zahodil. Když takovéto rozcestí přidáme pro všechny hrany grafu G, zaručíme, že do cíle se lze dostat právě s nějakou nezávislou množinou.

Graf P má jistě polynomiální velikost a v polynomiálním čase ho dokážeme vytvořit. Poté ho předáme krabičce na řešení problému přenosu ovoce spolu s k, které jsme obdrželi na vstupu, a dostaneme výsledek. Tím jsme ukázali, že problém přenosu ovoce je $\mathcal{NP}$-úplný.

Říša Hladík

31-4-5 Dělení království

Zlomek h/k, který máme spočítat, si s dovolením přejmenujeme na a/b. Nejprve vypíšeme celou část: to je $\lfloor a/b \rfloor$. Čitatele pak nahradíme zbytkem $a' = a \bmod b$ a pustíme se do vypisování desetinné části.

Zalovíme v hlubinách naší mysli a vytáhneme algoritmus na dělení čísel na papíře, který nás kdysi učili ve škole. Jak funguje? Udržuje si aktuální zbytek z, což je na začátku a'. V každém kroku připíše ke zbytku následující číslici – to je v našem případě vždy 0, takže zbytek násobíme 10. Poté zbytek vydělíme b: dostaneme podíl, což je další číslice výsledku, a nový zbytek.

Pokud dostaneme zbytek 0, skončíme. Jinak se zbytky nutně začnou opakovat a tím pádem i číslice výsledku. (Věnujeme chvíli přemýšlení tomu, proč nemohou být číslice periodické dřív, než se začnou opakovat zbytky…)

Tato myšlenka stačí na řešení, které je rychlé, ale spotřebuje spoustu paměti (tedy řešení úlohy 31-Z2-4). Použijeme dva průchody: první průchod sleduje, které zbytky se už objevily. Jakmile se nějaký zbytek z zopakuje, pustíme se do druhého průchodu. Ten dělí znovu od začátku, ale už vypisuje číslice. Když poprvé narazí na zbytek z, ohlásí začátek periody. Když na něj narazí podruhé, oznámí konec periody a zastaví se. Druhý průchod jsme stihli v čase $\mathcal{O}(N)$, kde N je délka výstupu, a stačila nám konstantní paměť. První potřeboval $\mathcal{O}(b)$ paměti na tabulku spatřených zbytků a stejné množství času na její inicializaci a nalezení zopakovaného zbytku.

Optimální řešení ponechá druhý průchod tohoto algoritmu a první předělá, aby nepotřeboval tolik času a paměti.

Představme si graf, jehož vrcholy jsou možné zbytky a hrana vede vždy ze zbytku x do $(10x) \bmod b$. Náš algoritmus vyrazí z vrcholu a' po hranách. Zastaví se, když se poprvé zopakuje vrchol. Část grafu, kterou jsme prošli, vypadá jako cesta, na kterou navazuje kružnice. Čili takové „kolečko s ocáskem“. Kolečko odpovídá periodě, ocásek předperiodě, dohromady obsahují přesně N vrcholů. Stačí tedy najít vrchol, v němž se ocásek napojuje na kolečko, a můžeme spustit druhý průchod.

Zkusme do grafu vypustit želvu a zajíce. Želva začne ve vrcholu a' a každým krokem se posune po jedné hraně. Zajíc začne tamtéž, ale běží rychleji: za jeden krok se posune po dvou hranách. Ukážeme, že po nejvýše $2N$ krocích se potkají a že to bude někde na kolečku (to je jednoduché: na ocásku je zajíc vždy před želvou). Nejprve je necháme běžet N kroků. Pokud se ještě nepotkali, jsou už v tomto okamžiku určitě někde na kolečku. Označme d, o kolik hran je želva před zajícem. Toto d je nejvýše N a každým dalším krokem se zmenšuje o 1. Po nejvýše N dalších krocích tedy musí zajíc želvu dohonit.

Našli jsme tedy nějaký vrchol v, který leží na kolečku, ale nejspíš to není ten správný, kde se napojuje ocásek. Pokračujeme dál. Zajíce necháme odpočinout (však toho naběhal dvakrát tolik), želvu necháme jít dál. Současně ale vypustíme ze startovního vrcholu korytnačku, která se pohybuje stejně rychle jako želva. Počkáme stejný počet kroků, jak dlouho nám prve trvalo setkat se ve v, a všimneme si, že želva s korytnačkou se právě musely potkat. Vskutku: želva šla dvakrát déle než předtím, takže urazila stejnou vzdálenost, jako předtím zajíc. A korytnačka nachodila stejně, jako předtím želva. Jenže želva i korytnačka jdou stejnou rychlostí, takže se mohly potkat jedině ve vrcholu, do kterého vedou dvě hrany, a takový je jenom jeden – napojení ocásku na kolečko.

Simulace želvy, zajíce a korytnačky trvá $\mathcal{O}(N)$ času a stačí si pamatovat dvě počítadla kroků a aktuální polohu všech tří zvířátek. Na to nám vystačí konstantní paměť.

Pro úplnost ještě dodejme, že konstantní paměti lze také dosáhnout různými způsoby založenými na binárním vyhledávání. Ty jednodušší z nich mívají časovou složitost $\mathcal{O}(N \log N)$, chytřejší ji zlepšují na $\mathcal{O}(N)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/31-4-5.c`

Martin „Medvěd“ Mareš

31-4-6 Kde je hroznýš? Kuk!

V **úkolu 1** bylo třeba přepsat `mousePressEvent`. Kromě samotné úpravy počtu paprsků se nastavil časovač, který každých 250 milisekund udělal požadovanou operaci. Navíc bylo třeba implementovat i `mouseReleaseEvent`, kde se časovače zrušily.

Například takto:

```
    def up(self):
        print("up")
        self.uptimer = QTimer()
        self.uptimer.timeout.connect(self.up)
        self.uptimer.start(250)
        self.beams += 1
        self.update()

    def down(self):
        print("down")
        self.downtimer = QTimer()
        self.downtimer.timeout.connect(self.down)
        self.downtimer.start(250)
        if self.beams > 0:
            self.beams -= 1
            self.update()

    def mousePressEvent(self, event):
        # Obsluha myšítkových událostí
        print(event.button())
        if event.button() == Qt.MouseButton.LeftButton:
            self.beams += 1
            self.uptimer = QTimer(singleShot=True)
            self.uptimer.timeout.connect(self.up)
            self.uptimer.start(1000)

        if event.button() == Qt.MouseButton.RightButton:
            self.beams -= 1
            self.downtimer = QTimer(singleShot=True)
            self.downtimer.timeout.connect(self.down)
            self.downtimer.start(1000)

        # Vyžádáme si překreslení
        self.update()

    def mouseReleaseEvent(self, event):
        if event.button() == Qt.MouseButton.LeftButton:
            if self.uptimer is not None:
                self.uptimer.stop()
                self.uptimer = None

        if event.button() == Qt.MouseButton.RightButton:
            if self.downtimer is not None:
                self.downtimer.stop()
                self.downtimer = None
```

Základním řešením **úkolu 2** bylo napsat `mouseMoveEvent`, který spočítal z polohy myše, jestli se sluníčko má nebo nemá ukázat, a podle toho také vyvolal nebo nevyvolal překreslení.

To nebylo úplně čisté; pokud jste chtěli, aby widget reagoval na jiné změny než pohyb myše, bylo třeba připsat ještě přinejmenším obsluhu `resizeEvent` a `moveEvent`, kdy se widget pohnul, ale myš ne, a ani to není úplně všechno. Pravděpodobně nejčistší by bylo spočítat, jestli se má sluníčko kreslit nebo ne, až v `paintEvent`.

Nicméně nám úplně stačilo implementovat `mouseMoveEvent` dle zadání.

```
    def hideSunUpdate(self, point):
        w = self.width()
        h = self.height()

        radius = min(w, h) / 4

        hideSun = (point.x() - w/2)**2 + (point.y() - h/2)**2 < radius**2

        if hideSun == self.hideSun:
            return

        self.hideSun = hideSun
        self.update()
```

```
def mouseMoveEvent(self, event):
    self.hideSunUpdate(event.pos())
```

Další část seriálu se ukázala jako velice komplikovaná. To bylo jedním z důvodů (kromě bezprecedentní neschopnosti autorky naplánovat si práci), proč se řešení seriálu tak opozdilo. Za svoji neschopnost (a taktéž za neschopnost o této neschopnosti včas informovat, pokračujte dále do rekurze ad absurdum) se autorka sice může omlouvat do aleluja, ale nic to nespraví na tom, že to je prostě pozdě.

Vzhledem k časovému odstupu vydání seriálu ale můžeme konstatovat, že se nikdo z účastníků nezaleknul toho, že při neuváženém kliknutí na CrossingView celá aplikace vytuhla. Nepřekvapivě to však ani téměř nikdo příliš neřešil; ono to ostatně není úplně jednoduché. Do bodování se tento problém samozřejmě nepromítnul.

Než se podíváme na chyby ve View, ukážeme, kterak vyřešit úkoly ze zadání.

Řešením **úkolu 3** bylo jednoduše prohodit pořadí, ve kterém se vykreslují chodci, chodník a auta. K tomu bylo asi nejjednodušší nejprve vykreslit silnici, pak projít model a vykreslit auta, pak přes to nakreslit chodník a nakonec znovu projít model a vykreslit chodce.

Řešení **úkolu 4** bylo taktéž velmi přímočaré. Bylo třeba v konstruktoru třídy `Pedestrian` přidat náhodnou volbu „horizontální“ polohy chodce a následně ji předávat do View například jako další roli v modelu.

To se však nedá říct o **úkolu 5**. Za ten sice dostala řada účastníků plný počet bodů, jednalo se však vždy o nějakým způsobem rozbitá řešení. Jak to udělat pořádně?

Začneme tím, že rozšíříme silnici na dva pruhy (úpravou vykreslování). Každé auto si na vstupu náhodně zvolí pruh, kterým vjede do oblasti, ale nadále bude ostatní auta ignorovat.

Pak musíme detekovat kolize. Pořídíme si tedy seznam aut v každém pruhu, setříděný podle polohy, a spočítáme si pro každou dvojici aut v jednom pruhu hned za sebou, za jak dlouho se srazí. Z toho si vybereme minimum a nastavíme si časovač; když se spustí, tak auto zpomalíme na rychlost toho před ním.

Konečně přijde na řadu implementace přejíždění z pruhu do pruhu, kdy místo zpomalení prvně zkusíme předjíždět. Pokud zjistíme, že se tedy auta mohou srazit, tak se podíváme do vedlejšího pruhu, jestli se tam náhodou nevejdeme – a pokud ano, tak se tam přemístíme.

V tom celém se dá samozřejmě udělat spousta hloupých chyb, zejména tzv. ±1-chyb. Na ladění vzorového programu se hodily hojné ladicí výpisy; to je pro tento typ programu asi nejefektivnější metoda, jak se dobrat chyby – vypíšu si, co program dělá, a porovnám to se svojí představou, jak by měl fungovat správně.

Konečně bychom si měli ukázat, jaké byly **chyby v předloženém View** ze zadání.

View i v read-only režimu umožňuje **výběr**. V zadání jsme zapomněli implementovat metody, které se starají o obsluhu událostí myše a klávesnice, které by v jiných Views učinily výběr. Pokud chceme umět View bez (axiomu) výběru, stačí je implementovat úplně obyčejně:

```
def moveCursor(self, action, modifiers):
    return QModelIndex()

def setSelection(self, rect, flags):
    self.selectionModel().select(QModelIndex(), flags)
```

Bez implementace těchto dvou metod nám celý program po kliknutí myšem na View zatuhne a nezbývá nám, než jej zabít.

A jak se na tuhle chybu dalo přijít? Z dokumentace těžko, protože tam nikde není napsaná minimální sada metod, kterou je třeba implementovat. Python však má magickou metodu `__getattr__(self, name)`, která se zavolá pokaždé, když se nepodaří najít požadovaný atribut (a metoda je atributem) objektu.

Pokud tedy doimplementujeme

```
def __getattr__(self, name):
    print("Sháním neexistující atribut", name)
    sys.exit(1)
```

Pak nám při kliknutí na View bez definované metody `moveCursor` nebo `setSelection` vypadne hláška *Sháním neexistující atribut moveCursor* a po pohledu do dokumentace už jen vymyslíme, jak bude vypadat triviální implementace takové metody.

A to je ze seriálu všechno. Děkujeme za trpělivost a přejeme mnoho štěstí v dalším ročníku.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/31-3-6.py`

Maria Matějka