Korespondenční Seminář z Programování

31. ročník **KSP** **Leden 2019**

Milí řešitelé, milé řešitelky!

Krátce po Novém roce vám přinášíme autorská řešení druhé série. Račte se podívati, jak jsme úlohy zamýšleli, třeba v našich řešeních najdete jiné přístupy, než které jste zvolili sami.

Připomínáme, že od letoška jsou řešení každé série rozdělena na dvě části: na samotná autorská řešení, která vydáváme brzy po termínu série a jejichž druhou 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 druhé série třicátého prvního ročníku KSP

31-2-1 Objednávka pily

Vyřešme nejprve nejlehčí variantu. V té se nějaký náš řetězec S délky N snažíme zapsat jako $k \times B$, kde B je co nejkratší; jinými slovy chceme pro řetězce nalézt jeho nejkratší periodu, která se opakuje beze zbytku.

Triviální řešení je vyzkoušet všechny možnosti, tedy všechna možná k. Pro každé k pak v lineárním čase ověříme, zda má S periodu délky N/k, třeba tak, že ověříme platnost vztahu $S[i] = S[i + N/k]$ pro všechna i, a za výsledek vezmeme co největší k.

To není tak marný nápad, jak se na první pohled může zdát: aby totiž některé k mělo smysl zkoušet, délka řetězce S musí být tímto k dělitelná. Možných dělitelů je však poměrně málo, odhad, který zde nebudeme dokazovat, říká, že číslo x má nejvýše $\mathcal{O}(\log x / \log\log x)$ dělitelů. Časová složitost tohoto řešení je tedy $\mathcal{O}(N \log N / \log\log N)$, tedy drobet lepší než $\mathcal{O}(N \log N)$.

Inspirujeme se KMP

My se však v našem řešení vydáme jiným směrem a ukážeme algoritmus, který bude pracovat v lineárním čase, a to i pro plnou verzi úlohy. Nepředbíhejme však a pojďme nejprve vyřešit druhou lehčí variantu.

Tentokrát nám na konci řetězce přibyla ještě část C představující poslední částečné zopakování periody. Trik s děliteli kvůli ní už nemůžeme použít, místo toho využijeme přiloženou kuchařku o vyhledávání v textu. Prozkoumejme, co se stane, když uvážíme zpětnou funkci z algoritmu KMP a spočítáme ji pro náš řetězec S. Připomeňme, že zpětná funkce nám pro každou pozici i počítá délku nejdelšího vlastního prefixu řetězce $S[:i]$, který je zároveň jeho suffixem, jinými slovy nejdelší začátek řetězce $S[:i]$, který je zároveň i jeho koncem. Můžeme si představit, že z každého znaku řetězce ukazuje doleva šipka na pozici určenou zpětnou funkcí.

Jako $f(i)$ označme hodnotu zpětné funkce pro pozici i, pak z definice platí $S[:f(i)] = S[i - f(i) : i]$. Bude se nám taky hodit pro každou pozici zjišťovat, o kolik znaků zpětná funkce skáče zpět, pořídíme si tedy ještě pole P, kde $P[i] = i - f(i)$. Co nám říká hodnota $P[N]$? Podle definice platí $S[:f(N)] = S[N - f(N) : N]$, tedy $S[:N - P[N]] = S[P[N] :]$. To vlastně neznamená nic jiného, než že každý znak se shoduje se znakem o $P[N]$ pozic dříve. Přesně to ale znamená, že řetězec je periodický s periodou délky $P[N]$. Zbývá si rozmyslet, že žádnou kratší periodu mít nemůže, protože pak by i $P[N]$ bylo menší, než je.

Celý algoritmus poběží v čase $\mathcal{O}(N)$: nejprve v lineárním čase spočte zpětnou funkci a pole P a pak jen odpoví hodnotou $P[N]$.

Vzorové řešení

Pojďme konečně vyřešit plnou verzi úlohy. V té nám na začátek řetězce přibude ještě neperiodická část A. Začneme tím, že si řetězec obrátíme. Rozmyslete si, že tak se naše úloha změní na nalezení „co nejkratšího“ rozkladu na řetězce $k \times D$, E, F, kde D je perioda, E její poslední částečné zopakování a F libovolný řetězec.

Inspirujeme se předchozím řešením. Stejně, jako jsme si rozmysleli, že $P[N]$ počítá délku nejkratší periody řetězce $S = S[:N]$, si můžeme rozmyslet, že $P[i]$ počítá délku nejkratší periody řetězce $S[:i]$. S touto znalostí je ale vyřešení úlohy už snadné: stačí jednoduše vyzkoušet všechny možnosti, konkrétně všechny předěly mezi periodickou částí a zbytkem. Pro každý z předělů si spočteme, jaký výsledný součet bychom získali, kdybychom řetězec rozdělili na tomto místě: rozdělíme-li řetězec na pozici i, zaplatíme $P[i]$ za délku periody a $N - i$ za délku zbytku. Ze všech možností pak vezmeme tu nejlevnější.

Časová i paměťová složitost řešení zůstává lineární v délce řetězce.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/31-2-1.py`

Říša Hladík

31-2-2 Hledání světýlka

Úlohu budeme řešit tak, že si pro každý strom budeme pamatovat, na jak vysoký strom se z něj lze dostat.

Kdybychom věděli, na jak vysoký strom se lze dostat ze sousedů, snadno zjistíme, na jak vysoký strom se lze dostat z aktuálního stromu:

a) sousední stromy jsou nižší než aktuální. Pak nejvyšší strom, na který se lze dostat, je on sám.
b) nějaký sousední strom je vyšší než aktuální. Pak nejvyšší strom, na který se lze dostat, je maximum z aktuálního stromu a hodnot spočítaných pro vyšší sousedy.

Problém je, že my na začátku algoritmu nevíme, na jak vysoké stromy se lze dostat ze sousedů. Nicméně víme, že můžeme chodit jen na (ostře) vyšší, tedy když se ptáme souseda, na jak vysoký strom se může dostat, tak on se už nás ptát nebude. Můžeme se tedy vždy pro každého vyššího souseda zeptat, na jak vysoký strom se dokáže dostat,

v tomto sousedovi se rekurzivně zeptáme opět jeho vyšších sousedů atd. až rekurze narazí na strom, který nemá žádné vyšší sousedy, a tehdy se rekurze začne vynořovat. Při vynořování z rekurze si musíme zapamatovat nejvyšší strom, na který se bylo možné dostat, protože jinak bychom se opakovaně rekurzili a to by bylo časově náročné.

Když máme pro každý strom napočítán nejvyšší dostupný strom, stačí projít všechny stromy a najít takový, který má maximální rozdíl mezi spočtenou výškou a jeho vlastní výškou.

Časová složitost bude $\mathcal{O}(n)$, kde n je počet stromů, protože se každého stromu maximálně čtyřikrát zeptáme, jaký je nejvyšší dostupný strom. Projití n stromů zabere $\mathcal{O}(n)$.

Paměťová složitost bude také $\mathcal{O}(n)$, protože si pamatujeme dvě pole velikost n. Jedno, kde máme uvedené výšky stromů, a druhé, kde máme uvedené výšky nejvyššího dostupného stromu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/31-2-2.py`

Vojtěch Sejkora

31-2-3 Oprava střechy

Zajímá nás, kolik nejvíce bodů můžeme zakrýt čtvercem o straně délky k. Nejprve si představíme kvadratické řešení a poté se ho pokusíme vylepšit. Všimneme si, že alespoň v jednom z optimálních řešení bude jeden bod ležet na levé straně čtverce. Pokud by tomu tak nebylo, můžeme čtverec posunout o kousek doprava a žádný bod při tom neztratíme.

Použijeme techniku *zametání*, kde projedeme svislou přímkou zleva doprava přes celou rovinu a když tato přímka protne nějaký zajímavý bod, zpracujeme příslušnou událost. Nejprve si tedy seřadíme všechny body podle x-ové souřadnice. Budeme je postupně procházet a pro každý bod zjistíme, kolik nejvíce bodů by zakryl čtverec s levou stranou na stejné x-ové souřadnici jako tento bod. Pro spočtení bodů budeme zametat znovu, ale tentokrát seshora. K tomu si vytvoříme další pole s body, ale seřadíme je tentokrát podle y-ové souřadnice.

Označme souřadnice bodu, který právě zkoumáme, jako $[a, b]$. Budeme postupně procházet pole s body seřazenými podle y a když narazíme na nějaký, jehož x-ová souřadnice spadá do intervalu $(a, a + k)$, zapíšeme si jej do fronty a přičteme jedničku k aktuálnímu počtu bodů ve čtverci. Čtverec má však výšku pouze k, proto nesmíme zapomenout bod včas zase zahodit. Před každým přidáním tedy zkontrolujeme, zda se y-ová souřadnice prvního zapamatovaného bodu ve frontě liší od nového nejvýše o k a pokud ne, budeme body z fronty zase vyhazovat. Každý bod do fronty přidáme a z fronty vyhodíme nejvýše jednou, složitost jednoho průchodu tedy bude lineární.

Pro každý možný levý okraj čtverce jsme tedy projeli rovinu shora dolů, maximum, na které jsme narazili, bude tedy správným řešením. Nejprve jsme body setřídili v čase $\mathcal{O}(n \log n)$ a poté jsme pro každý bod prošli všechny body ještě jednou, celková časová složitost tedy bude $\mathcal{O}(n^2)$. Poznamenejme jenom, že předpokládáme, že se souřadnicemi umíme pracovat v konstantním čase.

Zasadíme strom

U každého bodu se zdržujeme tím, že procházíme všechny ostatní. Můžeme místo toho ale použít jednu šikovnou datovou strukturu: intervalový strom.[1] Dělící hranice mezi intervaly zde budou jednotlivé y-ové souřadnice bodů a čísla $y + k$ (celkem tedy nejvýše $2n$ čísel). Předpokládejme, že všechna čísla jsou různá, jinak bychom museli uvažovat i intervaly nulové délky. Pokud bude v intervalu $[y_1, y_2]$ číslo p, znamená to, že libovolným čtvercem s levým horním rohem v tomto intervalu zakryjeme p bodů.

Můžeme si to představit tak, že intervalový strom nám udržuje stav nějakého pásu, kterým zametáme, a po každém přidání bodu se zeptáme na maximální počet bodů, které lze v tomto pásu zakrýt čtvercem. Algoritmus tedy bude vypadat následovně: podle x-ové souřadnice si seřadíme všechny díry a postupně je budeme procházet. Když narazíme na díru se souřadnicemi $[x_i, y_i]$, přičteme jedničku k intervalu $[y_i, y_i + k]$. Budeme si držet ukazatel na první a poslední díru, která se nám ještě vejde do čtverce. Než se pokusíme přidat novou díru, zkontrolujeme, jestli se x-ová souřadnice nově přidávané díry liší od té první alespoň o k. Pokud ano, první díru vyhodíme a od jejího intervalu opět jedničku odečteme. Pokud ne, můžeme novou díru přidat. Po každém přidání díry pak provedeme v intervalovém stromě dotaz na maximum. Po projetí celého pole s dírami jsme zjistili globální maximum. Pokud jsme na toto maximum narazili po přidání díry se souřadnicemi $[x_i, y_i]$ a interval s nejvyšším číslem byl $[a, b]$, pak můžeme levý horní roh čtverce umístit do bodu $[x_i, a]$ a zakryjeme jím největší možný počet děr.

Zbývá jen vysvětlit, jak budeme k intervalu přičítat nebo odečítat jedničku a jak najdeme interval s nejvyšším číslem. Změnu čísel v intervalu budeme provádět *líně*. To znamená, že pokud budeme chtít zvýšit všechna čísla v intervalu $[i, j]$ o 1, rozložíme tento interval na kanonické intervaly (tedy takové, které celé reprezentují nějaký podstrom) a do kořenů těchto podstromů zapíšeme instrukci „v celém tomto podstromě zvyš všechna čísla o 1“. Když později na tuto instrukci cestou narazí nějaká jiná operace, posune instrukci o úroveň níž, až se někdy dostane konečně do samotných listů. Všimneme si, že tuto informaci o zvýšení čísel v podstromu zapíšeme na každé hladině nejvýše do dvou vrcholů, celkem tedy maximálně do $2 \log n$ vrcholů. Zvýšit i snížit čísla na nějakém intervalu tedy umíme v logaritmickém čase.

Dotaz na maximum umíme také provést v logaritmickém čase: každý vrchol si pamatuje maximum ze všech prvků ležících pod ním. Jednoduše tak zjistíme, ve kterém intervalu se toto maximum nachází.

Pro každý možný levý začátek čtverce tedy zjistíme, jaký nejvyšší počet bodů bychom zvládli zakrýt. Nakonec nám proto algoritmus musí vydat správný výsledek. Pro všechny body provedeme konstantní počet dotazů v logaritmickém čase, celková časová složitost algoritmu tedy bude $\mathcal{O}(n \log n)$.

Program ($\mathcal{O}(n^2)$, C++):
`http://ksp.mff.cuni.cz/viz/31-2-3-n2.cpp`

Program ($\mathcal{O}(n \log n)$, C++):
`http://ksp.mff.cuni.cz/viz/31-2-3-nlogn.cpp`

Zuzka Urbanová & Marek Černý

[1] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`

31-2-4 Továrna na perník

V úloze chceme zjistit, mezi kterými dvojicemi sousedních chaloupek povede potrubí. Poté, co potrubí rozmístíme, stačí postavit jednu továrnu na každém souvislém úseku potrubí. Pro naše potřeby se jako souvislý úsek počítá i osamocená chaloupka, která není nikam napojená.

Mějme dvojici sousedních chaloupek. Rozhodujeme se, zda mezi ně umístit potrubí. Pokud potrubí postavíme, celkový počet potřebných továren se vždy zmenší právě o jednu. Ať už jsou tyto dvě chaloupky napojené na libovolný úsek potrubí (včetně žádného), obě musí být spojeny s nějakou továrnou. (Přesněji, můžeme předpokládat, že obě jsou propojeny s právě jednou továrnou, vyšší počet by byl zbytečný.) Když tyto dva úseky potrubí propojíme, jedna ze dvou továren se stane zbytečnou a můžeme ji odstranit.

Jelikož se propojením libovolné dvojice sousedních chaloupek zbavíme právě jedné továrny ceny A, pro nalezení optimálního řešení nám stačí propojit ty dvojice chaloupek, pro které platí, že $d \cdot B < A$, kde d je vzdálenost mezi chaloupkami.

Toto vede k řešení s lineární časovou složitostí. Pro N chaloupek stačí projít všech $N-1$ sousedních dvojic a rozhodnout, zda je propojíme potrubím. Jelikož pozice chaloupek jsou na vstupu vzestupně seřazené, projít tyto dvojice chaloupek je snadné.

Toto řešení má také konstantní paměťovou složitost – nemusíme si pamatovat, kde jsme potrubí postavili a kde ne, stačí nám udržovat si cenu, kterou jsme zaplatili.

Konkrétní implementace může vypadat třeba takto: Postupně projdeme všechny pozice chaloupek na vstupu. Budeme si pamatovat, jakou cenu jsme zatím zaplatili, a pozici předchozí chaloupky. Do první chaloupky umístíme továrnu, což znamená pouze to, že k celkové ceně přičteme A. Pro každou další chaloupku spočítáme vzdálenost d k předchozí a buď k celkové ceně přičteme cenu potrubí $d \cdot B$, nebo do této chaloupky „postavíme“ továrnu za cenu A.

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

Kuba Pelc

31-2-5 Zhasínání pecí

První, co by nás mohlo napadnout, je prostě jít jedním směrem a všechny pece zhasínat. Jak ale poznáme, že máme skončit? Když se jen zastavíme u první zhasnuté pece, tak ještě nemáme jistotu, že jsme doopravdy zhasli všech N pecí!

Zkusíme tedy takový obousměrný postup: Nejprve Jeníček s Mařenkou zažehnou pec v počáteční místnosti a potom budou všechny ostatní pece postupně zhasínat. Nejprve zhasnou pec v místnosti vpravo od počáteční (vzdálené 1), pak se vrátí zpět do počáteční místnosti. Pak zhasnou v místnosti vpravo od počáteční vzdálené 2 a zase se vrátí; obecně v i-té vlně dojdou až do místnosti vzdálené i a vrátí se. To budou opakovat do té doby, než po nějaké vlně zjistí, že je zhasnutá pec v počáteční místnosti. V té mohli ale zhasnout jedině oni, když zhasínali pece ve vzdálenosti N od počáteční, tudíž víme, že museli zhasnout všechny pece. Do počáteční místnosti se vždy v každé vlně vrací právě proto, že potřebují zjistit, jestli ji už nějakou náhodou v minulé vlně nezhasli.

Co si k tomu musí pamatovat? Určitě vzdálenost od počáteční místnosti (to je nejvýše N), jestli zrovna jdou „od“ počáteční místnosti, nebo „k“ ní (to je vždy 1 bit informace) a v jaké vlně postupu jsou – v jaké vzdálenosti chtějí pec zhasnout (to je také nejvýše N). Tedy v paměti máme zabraný jen konstantní počet buněk.

Už jen takový jednoduchý algoritmus nám dává celkem použitelné řešení. Jen zbývá vyřešit, jak je rychlé – kolik zabere Jeníčkovi a Mařence kroků. Všimneme si, že algoritmus udělá vždy $2i$ kroků pro i-tou vlnu – doprava a zpátky a dělá vždy kroky délky rovné velikosti vlny. Celkem je to tedy $\sum_{i=1}^{N} 2i$ kroků, což je asymptoticky $\mathcal{O}(N^2)$ kroků.

Jde to ale i o něco lépe. Vlny se nemusí zvětšovat po 1, ale jejich velikosti se mohou vždy zdvojnásobovat. Tedy i-tá vlna bude velikosti 2^i a budou v ní zhasnuty všechny místnosti napravo od počáteční s vzdáleností menší rovnou 2^i. Největší k-tá vlna bude velikosti $2^k \geq N$, a tedy $k = \lceil \log(N) \rceil \leq \log(N) + 1$. Celkem je to

$$\sum_{i=0}^{\log(N)+1} 2 \cdot 2^i = 2 \cdot (2^{\log(N)+2} - 1) = 8 \cdot 2^{\log(N)} - 2,$$

tedy asymptoticky $\mathcal{O}(N)$ kroků. (Zde využíváme vzorce pro součet geometrické řady, $1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1$.)

Jirka Beneš

31-2-6 Hroznýš v událostech

Druhý díl už vyžadoval zkoumat a psát složitější kusy kódu, stále by však měl být poměrně dobře řešitelný i bez nahlížení do anglické dokumentace. Přesto silně doporučuji těm, kdo to ještě neudělali, do dokumentace alespoň nahlédnout a vyhledat si ty konstrukce a objekty, které jsme si v seriálu ukázali. Získáte tak poměrně dobrou představu o tom, jak je Qt dokumentované, díky čemuž pak v dokumentaci snáze vyhledáte to, co budete potřebovat do nějakého svého vážného projektu.

Řešení **úkolu 1** bylo jednoduché; stačilo přidat label a do obsluhy událostí jeho úpravy. Například takto:[2]

```
class CrossingS2U1(Crossing):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.stateLabel = QLabel(self,
                text="Disconnected")
        self.layout.addWidget(self.stateLabel)
        self.socket.disconnected.connect(
                self.disconnect) # (!)

    def connected(self):
        super().connected()
        self.stateLabel.setText("Connected")

    def connect(self):
        self.stateLabel.setText("Connecting")
        super().connect()

    def disconnected(self): # (!)
        self.stateLabel.setText(
                "Disconnected") # (!)
```

Řádky označené vykřičníkem ošetřují případ, kdy spojení z nějakého důvodu spadne. Vzhledem k tomu, že jsme signál

[2] Dědíme v prvním z programů v zadání.

`disconnected` neuvedli v zadání, budou body i za řešení, které toto neošetřuje.

Pokud bychom potřebovali podrobnější informace o stavu připojení, můžeme si registrovat signál `stateChanged`, který se posílá při každé změně stavu.

Čudlík na odpojení podle **úkolu 2** nebyl o moc těžší.

```
class CrossingS2U2(CrossingS2U1):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.disconnectButton = QPushButton(
                self, text="Stop")
        self.disconnectButton.clicked.connect(
                self.disconnect)
        self.layout.addWidget(
                self.disconnectButton)

    def disconnect(self):
        self.socket.disconnectFromHost()
        self.stateLabel.setText("Disconnected")
```

Řešení **úkolu 3** už vyžadovalo i nějaké datové struktury, konkrétně seznam. Jak zadání napovídá, je možné použít například mnoho `QLabel`ů a tlačítek. Činíme tak poněkud prasácky; ve třetí sérii si ukážeme, jak to udělat pořádně.

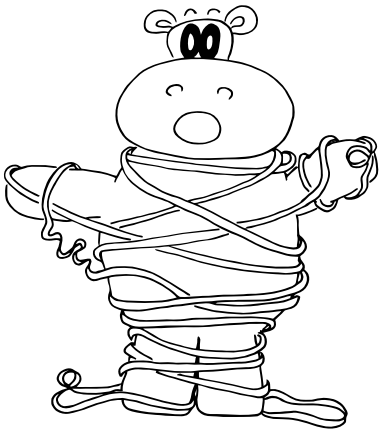
```
from PyQt5.QtWidgets import QHBoxLayout

class TravellerL(QWidget):
    def __init__(self, crossing, text,
                 *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.crossing = crossing
        self.text = text
        self.label = QLabel(self, text=text)

        self.backButton = QPushButton(self,
                text="Return")
        self.backButton.clicked.connect(
                self.sendBack)

        self.layout = QHBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.backButton)
        self.setLayout(self.layout)

    def sendBack(self):
        self.crossing.sendBack(self)

class CrossingS2U3L(CrossingS2U2):
    def read(self): # Nahrazujeme původní metodu
        # Přečteme všechno, co jsme dostali
        while self.socket.bytesAvailable() > 0:
            self.readBuffer += \
                    self.socket.read(128)

        # Rozdělíme na řádky
        lines = self.readBuffer.split(b"\n")

        # Zbytek uložíme na příště
        self.readBuffer = lines.pop()

        # Zpracujeme řádky, které dorazily
        for l in lines:
            self.gotLine(l.decode().rstrip())

    def gotLine(self, line):
        self.layout.addWidget(
                TravellerL(self, line))

    def sendBack(self, traveller):
        text = traveller.text + "\n"
        self.socket.write(text.encode())
        traveller.deleteLater()
```

Zde jsme použili metodu `deleteLater`, která nebyla v zadání; do budoucna se vám však může hodit. Místo toho bychom mohli například udržovat nepoužité objekty `TravellerL` v nějakém seznamu a recyklovat je a po každém smazání objektu přegenerovat celý layout. To je sice neoptimální, ale na těch pár položkách se to ztratí.

Tahle metoda se hodí na jisté smazání jakéhokoliv objektu, který patří Qt. Není smazán hned, ale až při další otočce *smyčky událostí*, viz zadání první série.

Druhá navržená cesta, jak řešit **úkol 3**, byla přes `QComboBox`. Oproti první cestě nevytvářela žádné další objekty, nebylo třeba nic mazat nebo recyklovat a především okýnko samotné nevyrostlo do strašlivé velikosti.

```
from PyQt5.QtWidgets import QComboBox

class CrossingS2U3C(CrossingS2U2):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.combo = QComboBox(self)
        self.backButton = QPushButton(self,
                text="Return")
        self.backButton.clicked.connect(
                self.sendBack)

        self.layout.addWidget(self.combo)
        self.layout.addWidget(self.backButton)

    def read(self): # Nahrazujeme původní metodu
        # Přečteme všechno, co jsme dostali
        while self.socket.bytesAvailable() > 0:
            self.readBuffer += \
                    self.socket.read(128)

        # Rozdělíme na řádky
        lines = self.readBuffer.split(b"\n")

        # Zbytek uložíme na příště
        self.readBuffer = lines.pop()

        # Zpracujeme řádky, které dorazily
        for l in lines:
            self.gotLine(l.decode().rstrip())

    def gotLine(self, line):
        self.combo.addItem(line)

    def sendBack(self, traveller):
        text = self.combo.currentText() + "\n"
        index = self.combo.currentIndex()
        self.socket.write(text.encode())
        self.combo.removeItem(index)
```



Úkol 4 vyžadoval trochu uvažování v událostech. Bylo třeba asynchronně poslat `BYE`, počkat na odpověď, tu vypsat a zavřít spojení. Odpojovací metoda tedy pouze pošle `BYE` a skutečné odpojení se provede až z obsluhy čtení ze socketu.

```
class CrossingS2U4(CrossingS2U3C):
  def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)
    self.statisticsLabel = QLabel(self)
    self.layout.addWidget(self.statisticsLabel)
    self.disconnecting = False

  def disconnect(self):
    if not self.disconnecting:
        self.disconnecting = True
        self.socket.write("BYE\n".encode())
        self.stateLabel.setText("Sent BYE")

  def gotLine(self, line):
    if self.disconnecting and line.startswith("STATS"):
        self.statisticsLabel.setText(line)
        super().disconnect()
    else:
        super().gotLine(line)

  def connected(self):
    super().connected()
    self.statisticsLabel.setText("")
```

V úkolech 5 a 6 budeme dědit poslední z programů v zadání. K řešení **úkolu 5** stačilo přidat dva labely a při změnách jim nastavovat správnou hodnotu.

```
class CrossingS2U5(Crossing):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.carNumLabel = QLabel(self,
                text="Cars: 0")
        self.pedNumLabel = QLabel(self,
                text="Pedestrians: 0")

        self.layout.addWidget(self.carNumLabel)
        self.layout.addWidget(self.pedNumLabel)

        self.carNum = 0
        self.pedNum = 0

    def updateNums(self):
        self.carNumLabel.setText(
                "Cars: %(num)d" %
                { "num": self.carNum })
        self.pedNumLabel.setText(
                "Pedestrians: %(num)d" %
                { "num": self.pedNum })

    def addTraveller(self, traveller):
        super().addTraveller(traveller)
        if type(traveller) == Car:
            self.carNum += 1
        else:
            self.pedNum += 1
        self.updateNums()

    def sendBack(self, traveller):
        super().sendBack(traveller)
        if type(traveller) == Car:
            self.carNum -= 1
        else:
            self.pedNum -= 1
        self.updateNums()
```

Úkol poslední, tedy **šestý**, byl asi nejpracnější. Bylo třeba nejen zařídit výpis každého cestovatele, ať již dvounohého či gumokolného, ale také pravidelně vypisovat, jak daleko je od vstupu do oblasti, což se dalo zařídit asi nejlépe pravidelným tikem časovače.

```
class CrossingS2U6(Crossing):
    longTimerTick = 2000000000
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.longTimer = QTimer()
        self.longTimer.start(self.longTimerTick)

        self.entryTimes = {}

        self.updateTimer = QTimer()
        self.updateTimer.timeout.connect(
                self.updateLabels)
        self.updateTimer.start(250)

        self.statusWidget = QWidget(self)
        self.layout.addWidget(self.statusWidget)

    def now(self):
        return self.longTimer.remainingTime()

    def addTraveller(self, traveller):
        print("+", traveller)
        super().addTraveller(traveller)
        tid = traveller.id
        self.entryTimes[tid] = self.now()

    def sendBack(self, traveller):
        print("-", traveller)
        self.entryTimes[traveller.id] = None
        super().sendBack(traveller)

    def updateLabels(self):
        print("update")
        self.statusWidget.deleteLater()
        self.statusWidget = QWidget(self)
        self.layout.addWidget(self.statusWidget)

        statusLayout = QVBoxLayout(
                self.statusWidget)
        self.statusWidget.setLayout(
                statusLayout)

        now = self.now()

        for _,tr in self.travellers.items():
            if tr is None:
                continue
            entryTime = self.entryTime[tr.id]
            elapsed = entryTime - now
            if elapsed < 0:
                elapsed += longTimerTick

            position = elapsed * tr.speed / 1000
            text = ("%(who)s at %(pos).3fm " +
                   "of %(len)dm") % { "who": tr,
                   "pos": position, "len":
                   tr.roadLength }
            label = QLabel(self, text=text)
            statusLayout.addWidget(label)
```

Řešení je poněkud prasácké tím, že pokaždé celý výpis vygeneruje znovu, a taktéž využívá `deleteLater` jako minule, tentokrát ke smazání celého výpisu. Snad chápu z dokumentace správně, že po zavolání téhle funkce se automaticky

smažou i všichni potomci, konkrétně tedy všechny `QLabely` ve výpise.

Využívá se zde také trik takový, že se počítá s nějakou příčetnou dobou, kterou stráví cestující v oblasti – i kdyby auto jelo 500 metrů rychlostí 1mm/s, tak mu to bude trvat 500 000 sekund; nikdy se tedy nestane, že by za dobu, kdy se vyskytuje v oblasti, tiknul `longTimer` vícekrát než jednou.

Zdá se však, že možných řešení zrovna úkolu 6 bude daleko více, tak uvidíme, co jste vymysleli. Už si na vás brousím myš.

Maria Matějka