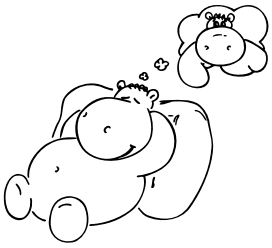
Korespondenční Seminář z Programování

31. ročník **KSP** **Listopad 2018**

Milí řešitelé a řešitelky!



Zdá se, že babí léto se nám protahuje do poloviny listopadu. Organizátoři KSP ale žádné průtahy nemají rádi, a proto už teď přicházejí s další várkou rozmanitých informatických úložek k řešení.

Všichni zájemci o studium informatiky jsou srdečně zváni na Den otevřených dveří na Matfyzu, který se koná 22. listopadu. Večer předtím si nenechte ujít Kalíšek, setkání s organizátory a řešiteli KSPčka. Více informací najdete na našem webu.

Připomínáme, že do celkového hodnocení se Vám z každé série započte 5 nejlépe vyřešených úloh. Každému řešiteli, který získá v tomto ročníku z každé série alespoň 5 bodů, pošleme KSP propisku, blok, placku a třeba i něco navíc.

Díky řešení KSP se také můžete vyhnout přijímacím zkouškám na MFF UK! Stačí, když získáte alespoň polovinu bodů z ročníku (tedy 150 bodů) a my vám vystavíme osvědčení, díky kterému vás přijmou na MFF bez zkoušek. Pozor ale: pokud studujete poslední ročník střední školy a chcete letošní osvědčení využít, musíte mít potřebné body již po čtvrté sérii.

Termín série: **7. ledna v 8:00**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** si vysloužíš každý, kdo nám k řešení přiloží ručně kreslený obrázek hrošíka. Hrošík by měl vykonávat nějakou netriviální činnost, představivosti se meze nekladou :). Pokud řešení odevzdáváte elektronicky, můžete nám oskenovaný obrázek poslat emailem na známou adresu.

Druhá série třicátého prvního ročníku KSP

V jedné malé chaloupce na kraji lesa žila neúplná rodinka – tatínek a dvě děti. Říkejme jim třeba Jeníček a Mařenka. Doba byla zlá, peněz málo, do chaloupky rodince teklo, ale i přesto se měli rádi. Tatínek se živil jako dřevorubec. Každý den odcházel brzy ráno a vracel se až po setmění. Děti tak vídal pouze zřídka.

Jednoho rána se mu přihodilo tuze veliké neštěstí! Pila se mu rozbila – rozlomila se vejpůl. Co si teď má počít? Pracovat nemůže, ale peníze nutně potřebuje!

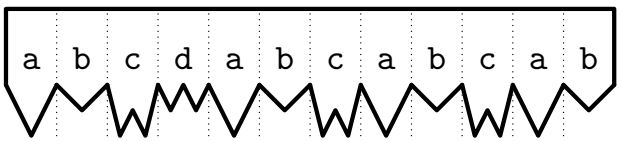
31-2-1 Objednávka pily — 14 bodů

Tatínek si potřebuje objednat novou pilu. To ale není tak jednoduché, protože je dělaná na míru tatínkovým potřebám. Pila má na začátku zuby nepravidelné, aby se jimi dobře zařezávala do dřeva. Dále se pak pravidelně opakuje, což umožňuje dělat dlouhé tahy. Protože se za každý znak vyskytující se v objednávce platí, potřebuje tatínek schéma zubů co nejvíce zkrátit.

Schéma zubů je popsáno řetězcem tvořeným písmeny anglické abecedy, přičemž každé písmenko je nějaký typ zubu podle vzorníku. Řetězec chceme rozdělit na tři části A, $k \times B$ a C. Část B je perioda, která se v řetězci k-krát opakuje. Část C je poslední zopakování periody, které může být pouze částečné. A konečně první část A je nějaká posloupnost písmen, která se nachází před periodou. Vaším cílem je najít takový zápis řetězce, kde součet délek A a B bude co nejmenší.

⇧ **Lehčí varianta (za 10 bodů):** Můžete předpokládat, že část A má nulovou délku, tedy řetězec má pouze části $k \times B$ a C.

⇧ **Lehčí varianta (za 5 bodů):** Jak část A, tak část C mají nulovou délku, jinými slovy, řetězec je tvaru $k \times B$.



Pro příklad na předchozím obrázku je jediným správným řešením $A =$ `abcd`, $B =$ `abc`, $C =$ `ab`, součet délek A a B je 7. Například rozdělení $A =$ `abcdab`, $B =$ `cab`, $C = \emptyset$ je sice také platné, ale nemá nejkratší možný součet délek.

Toho rána se tatínek vrátil domů brzy. Zbytek dne chtěl strávit se svými dětmi. „Jeníčku! Mařenko! Půjdeme na jahody! Znám jednu mýtinku, kde rostou ty nejsladší, jaké jsem kdy jedl!“ řekl tatínek. „Je to ale hluboko v lese, tak se mi hlavně nikde neztraťte.“ Děti radostí nadskočily, popadly džbánky a už už se hrnuly ven.

Jen co děti dorazily na mýtinku, vrhly se na jahody. Jak tatínek sliboval, byly úžasné. Sbíraly a sbíraly, dokud všechny jahody nevysbíraly. S plnými bříšky pak zalehly pod strom a usnuly.

Probudily se až za tmy. „Tatínkůůůůů!“ volaly vystrašeně. Bez odpovědi... „Jak se teď dostaneme domů?“ plakala Mařenka, „Les v noci vypadá úplně jinak!“ Jeníček naštěstí dostal nápad: „Počkej tady. Já zatím vylezu na strom a podívám se, jestli neuvidím nějaké světýlko.“

31-2-2 Hledání světýlka — 9 bodů

Jeníček chce vylézt na strom, ze kterého bude mít co možná nejlepší výhled, ale ze kterého se také dokáže dostat dostatečně nízko, aby si při seskoku na zem nezlámal nožičky. Šplhat po stromech umí už od narození, takže přeskakovat ze stromu na strom pro něj není žádný problém.

Les si lze představit jako čtvercovou mřížku. V každém vrcholu se nachází jeden strom, jehož výšku máme zadanou.

Při šplhání může Jeníček přeskočit ze stromu, na kterém se právě nachází, do libovolného ze čtyř směrů, pokud se tam nějaký strom nachází (tj. nedostane se mimo mřížku) a výška stromu v daném směru je ostře větší než výška aktuálního stromu. Chceme pro Jeníčka najít trasu splňující předchozí kritéria, kde rozdíl mezi výškami prvního a posledního stromu bude co největší.

Jak Jeníček řekl, tak také udělal. Vyšplhal na strom, rozhlédl se po okolí a opravdu, v dálce něco svítilo. „Musíme se vydat tam!“ volal dolů na Mařenku. Ladně jako veverka seskákal zpátky na zem a společně s Mařenkou se vydali na cestu za světýlkem.

Když dorazili k chaloupce, oněměli úžasem. Nebyla to totiž obyčejná chaloupka, byla to chaloupka celá z perníku. Po dlouhé cestě už jim docela vyhládlo, a protože stavení vypadalo opuštěně, s chutí se pustili do rámu okolo dveří. Když byly dveře na spadnutí, přesunuli se na střechu. Nakonec ani komín nezůstal nedotčen!

„Kdopak mi to tu loupe perníček?“ ozvalo se z chaloupky, zrovna když děti olizovaly okenní tabulky. Okousané dveře se sesypaly na zem a z chaloupky vykoukla hlava rozespalé ježibaby. Děti se lekly, ale hned pohotově odpověděly: „To nic, to jenom větříček.“ A s chutí jedly dál. Ježibaba ovšem nebyla úplně naivní. Takových loupežníků, co se přiživovali na jejích perníčcích, už několik nachytala. Vylezla tedy ven a objevila Jeníčka s Mařenkou. Na nic nečekala, popadla děti za ruku, vtáhla je do chaloupky a zavřela do klece. Pak se vrátila zpět do postele.

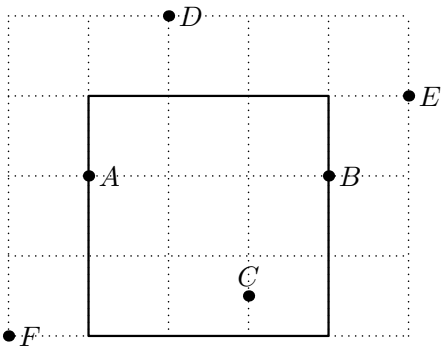
Netrvalo dlouho a ježibaba byla opět na nohou. Ráno totiž začalo pršet a vykousanými otvory ve střeše jí kapalo do postele. Aby mohla dál spát, nezbylo jí nic jiného než díry zakrýt.

31-2-3 Oprava střechy — 12 bodů

V noci děti do střechy vykousaly spoustu malých děr, které teď chce ježibaba zakrýt. Protože ale byla líná, upekla jenom jeden veliký perník. Chce ho na střechu umístit tak, aby pod něj schovala co nejvíce děr. Při umisťování však musí dbát na to, aby perník nepokazil celkový vzhled chaloupky, a proto je potřeba ho položit rovnoběžně s dolním okrajem střechy.

Vykousané díry představují body v rovině. Perník má tvar čtverce s pevně danými rozměry a chceme ho do roviny umístit rovnoběžně s osami tak, aby se pod ním nacházelo největší možné množství bodů. Kolik nejvíce bodů můžeme čtvercem přikrýt?

V příkladu na následujícím obrázku jdou čtvercem o straně délky 3 pokrýt nejvýše tři body, jedno z možných řešení je zakresleno. Kdybychom měli čtvercem dovoleno otáčet, zvládli bychom pokrýt body A, B, C, D, my však otáčet nemůžeme.



Oprava střechy ježibabě zabrala téměř celé odpoledne. A to byl jenom jeden perník! Kdyby jich měla péct a pokládat víc, střechu by nestihla spravit dřív, než by se zase objevili nějací lumpové, kteří by jí kus střechy uloupli.

Naštěstí nemusí perníky pokaždé péct sama. Jedna zdejší ježibaba si také postavila továrnu na perník, a tak se na zásobování podílejí společně. Jediný problém je s dopravou, protože na koštěti se perníky nepřepravují zrovna nejpohodlněji. Cesta trvá dlouho a poryv větru občas perníky z koštěte shodí.

31-2-4 Továrna na perník — 8 bodů

Když Jeníček s Mařenkou slyšeli, jak tady probíhá zásobování perníkem, hned je napadlo, že by to šlo zařídit mnohem lépe. Co postavit z každé továrny potrubí, kterým by šel perník přepravovat mnohem rychleji? Určitě by to bylo i mnohem levnější a třeba by se šlo s ježibabami domluvit, aby zásobovaly perníkem i jiné vesnice, zajisté by o to byl velký zájem!

Na přímce leží N ježibabích chaloupek. Všechny chceme napojit na nový systém zásobování. K tomu můžeme provést operace dvou druhů: postavit na nějakém místě továrnu za cenu A a spojit nějaká dvě místa potrubím za cenu $d \times B$, kde d je vzdálenost mezi nimi. (Ceny A i B jsou pevné a nezávislé na tom, na jakém místě stavíme továrnu, popř. jaká dvě místa spojujeme.) Prochází-li místem, na kterém stojí továrna nebo chaloupka, nějaké potrubí, je na něj daná budova připojena. Každá továrna může zásobovat libovolné množství chaloupek. Továrnu můžeme postavit i na místě, kde už stojí nějaká vesnice. Kde postavit továrny a potrubí, aby byly všechny chaloupky (i nepřímo) připojeny potrubím k nějaké továrně a celková cena byla co nejmenší?

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku se nachází tři celá kladná čísla oddělená mezerou – N, A a B. Na druhé řádku je N mezerou oddělených čísel – celočíselné pozice jednotlivých chaloupek. Máte zaručeno, že chaloupky jsou na vstupu vzestupně seřazeny podle souřadnice, a že všechny souřadnice jsou v rozsahu od -10^9 do 10^9.

Formát výstupu: Vypište jediné číslo: nejmenší možnou cenu, za kterou dokážeme každou chaloupku spojit s nějakou továrnou. Pozor, na reprezentaci výsledku můžete potřebovat 64bitová čísla (`long long` v C, `long` v Javě a C#; v Pythonu to není potřeba řešit).

Ukázkový vstup:

```
5 7 2
-3 0 4 6 8
```

Ukázkový výstup:

```
28
```

Jedno z možných řešení je postavit továrny na pozicích -1 a 5 a postavit potrubí délky 3 spojující -3 a 0 a potrubí délky 4 spojující 4 a 8. Celková cena je $2 \cdot A + (4+3) \cdot B = 28$.

Ježibaba se dopálila, když se Jeníčkovi s Mařenkou podařilo po krátké úvaze vymyslet efektivnější způsob zásobování. Po celém dnu měla dětí až po krk, a tak se rozhodla, že je upeče a sní. Aspoň pak bude mít zase svůj klid.

Ježibaba vytáhla lopatu a poručila dětem, aby se na ni posadily. První byla na řadě Mařenka. „Nikdy jsem na takové lopatě neseděla. Vždyť já ani nevím, jak na to.“ hrála Mařenka hloupou. Ježibaba vůbec netušila, že jde o lest a bez sebemenšího podezření si začala na lopatu sedat. „Ach ty dnešní děti. Ani na lopatu si sednout neumí! Děcka, teď se dobře dívejte, jak se to správně dělá.“ Jen, co to dořekla, Mařenka vyskočila, popadla lopatu a s vypětím všech svých sil strčila ježibabu do pece. Pak za ní ještě pořádně zabouchla dvířka. Osvobodila Jeníčka a chystali se společně utéct.

Děti si však uvědomily, že se v perníkové chaloupce nachází spousta pecí a že by nebylo dobré jen tak odejít a nechat je hořet bez dozoru. Chaloupka by mohla chytit a podpálit tak celý les...

31-2-5 Zhasínání pecí — 10 bodů

V chaloupce se nachází N navzájem nerozeznatelných do kruhu umístěných místností, kdy z místnosti i (pro $i = 0, \ldots, N-1$) vedou dveře do místnosti $(i \pm 1) \bmod N$ (což znamená, že pro $0 < i < N-1$ vedou dveře do místnosti $i-1$ a $i+1$, pro $i = 0$ do první a $(N-1)$-té místnosti a pro $i = N-1$ do $(N-2)$-té a do nulté místnosti). V každé místnosti se nachází pec, která buď hoří, nebo ne. Pec se zapíná/vypíná přepnutím páčky na její zadní části.

Jeníček s Mařenkou samozřejmě netuší, kolik je v chaloupce celkem místností. Neví ani, ve které místnosti se právě nachází. Nechtějí se od sebe raději moc vzdalovat, proto budou všechny místnosti procházet společně. Jejich úkolem je v konečném čase povypínat všechny pece a pak s jistotou prohlásit, že jsou všechny vypnuté.

Obvykle se soustředíme hlavně na to, aby algoritmus doběhl co nejrychleji. V této úloze tomu bude ale jinak! Primárním kritériem vašeho řešení je paměťová složitost a až sekundárním složitost časová. Samotnou paměť budeme přitom počítat v *buňkách,* kde do jedné buňky se vejde číslo velikosti řádově N.

Když Jeníček s Mařenkou uhasili oheň v poslední peci, vyběhli z chaloupky a utíkali domů, co jim nohy stačily. Chtěli být co nejdříve pryč.

Ode dne, co tatínek své milované děti v lese opustil, je chodil do lesa hledat. Jakou pak měl radost, když se konečně zase shledali! Celí šťastní se společně vrátili do své malé chaloupky na kraji lesa. A vůbec jim nevadilo, že nebyla z perníku.

Klárka Tauchmanová & Zuzka Urbanová & Vašek Šraier

31-2-6 Hroznýš v událostech — 15 bodů

Po stručném úvodu v první sérii budeme pokračovat důkladným procvičováním událostí a jejich obsluhy. Dosud jsme se naučili zpracovávat události časovače (když vypršel) a uživatelské akce (kliknutí). V dnešním dílu si předvedeme, jak se připojit k serveru na internetu a vyřizovat takovou komunikaci.

Qt nám na komunikaci se serverem poskytuje poměrně pohodlnou třídu QTcpSocket. Pojďme se podívat, jak se používá.

V celém druhém díle seriálu budeme vyrábět simulátor dopravy na křižovatce. Server použijeme jako generátor provozu a klient bude rozhodovat, kdy bude na jakém semaforu zelená. Začneme ale úplně obyčejným jednoduchým klientem.

```
from PyQt5.QtWidgets import \
        QApplication, QWidget, QLabel,\
        QPushButton, QVBoxLayout
from PyQt5.QtNetwork import QTcpSocket

import sys

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

        # Výroba ovládacích prvků
        self.connectButton = QPushButton(
                self, text = "Start")
        self.connectButton.clicked.connect(
                self.connect)
        self.messageLabel = QLabel(self)

        # Rozložení ovládacích prvků
        self.layout = QVBoxLayout(self)
        self.layout.addWidget(
                self.connectButton)
        self.layout.addWidget(self.messageLabel)

        # Příprava TCP socketu
        self.socket = QTcpSocket(self)
        self.socket.readyRead.connect(self.read)
        self.socket.connected.connect(
                self.connected)
        self.readBuffer = bytearray()

        # Zobrazení
        self.setLayout(self.layout)
        self.show()

    def connect(self):
        # Nejdřív se odpoj, pokud už
        # spojení běží
        self.socket.abort()

        # A znovu se připoj
        self.socket.connectToHost(
                "ksp.mff.cuni.cz", 48888)

    def connected(self):
        # Pozdravíme server
```

```
        self.socket.write("HELLO\n".encode())

    def read(self):
        # 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.messageLabel.setText(
                    l.decode().rstrip())

# Spuštění celého programu
app = QApplication(sys.argv)
crossing = Crossing()
app.exec()
```

Program vyrobí triviální GUI a `QTcpSocket`. Po kliknutí na tlačítko se socket pokusí připojit k zadanému serveru. Když se povede připojení, pošleme zprávu Hello. A když přijdou data, přečteme je, rozdělíme po řádkách a každý řádek vypíšeme do labelu.

Všimněte si, jakým způsobem jsou vstupní data uložena. Jedná se o `bytearray`, nikoli o řetězec. Dekódujeme je do řetězce až jako celé řádky – co kdyby náhodou přišel znak v UTF-8 rozdělený v půlce?

Celý mechanismus, kterým se pracuje se socketem, je událostní. O přítomnosti dat na vstupu se dozvíme pomocí události `readyRead`, o připojení taktéž (a můžeme tedy server pozdravit). Další zajímavou událostí, kterou nyní neobsluhujeme, je například `disconnected`.

Do socketu se nezapisuje přímo. Když zapíšete metodou `write`, jsou data uložena do bufferu, který se postupně vyprazdňuje. Pokud chcete vědět, kdy jsou data skutečně odeslána, objednejte si událost `bytesWritten`, která se vyvolá pokaždé, když se skutečně zapíšou data do socketu. Pozor, tato událost má jeden argument, který říká, kolik bajtů bylo zapsáno; všechny dosud používané události žádný argument neměly. Pokud bychom ji chtěli přidat do našeho programu, objednáme ji pořád jako `.bytesWritten.connect(self.handler)`, ale definice musí obsahovat onen argument: `def handler(self, bytes)`

Mnoho dalších zajímavých vlastností `QTcpSocket`u naleznete v jeho dokumentaci;[1] znovu připomínáme, že se jedná o dokumentaci pro C++, takže je třeba provést si v hlavě příslušnou konverzi.

Úkol 1 [1b]: Stav připojování a odpojování se občas hodí vidět. Přidejte do programu další `QLabel`, který bude zobrazovat aktuální stav připojení (například Disconnected, Connecting a Connected).

Úkol 2 [2b]: Přidejte do programu tlačítko na odpojení, které socket odpojí. Na to se hodí použít metodu `disconnectFromHost`, která se pokusí o slušné uzavření spojení, narozdíl od `abort`.

Pokud necháte program puštěný déle, zjistíte, že server postupně přestane posílat auta i chodce. Očekává totiž, že mu budete posílat auta a chodce zpátky. Naučíme se to nejprve ručně: přidáme si k aktuálně přidanému řádku tlačítko „Return“, které přijatý dopravní prostředek vrátí zpět.

```
class Crossing(QWidget):
    def __init__(self, *args, **kwargs):
    # ... viz výše
        self.backButton = QPushButton(
                self, text = "Return",
                enabled = False)
        self.backButton.clicked.connect(
                self.sendBack)
        self.layout.addWidget(self.backButton)

    def read(self):
    # ...
        # Zpracujeme řádky, které dorazily
        for l in lines:
            self.messageLabel.setText(
                    l.decode().rstrip())
            self.backButton.setEnabled(True)

    def sendBack(self):
        text = (self.messageLabel.text() + "\n")
        self.socket.write(text.encode())
        self.backButton.setEnabled(False)
```

Úkol 3 [3b]: Výše navržená úprava ovšem vrátí vždy jen poslední přijatý dopravní prostředek. Napište program, který uživateli nabídne k vrácení všechny dopravní prostředky, které dosud nevrátil zpět.

K řešení předchozího úkolu můžete využít například další a další přidávané `QLabel`y a tlačítka, ale také třeba `QComboBox`. To je okýnko s jedním řádkem textu a šipkou, která vybalí další řádky na výběr. Jak se používá? Ukažme si to na úplně hloupě napsaném ilustrativním příkladu objednávkového systému v restauraci.

```
from PyQt5.QtWidgets import QApplication, \
        QWidget, QLabel, QPushButton,\
        QHBoxLayout, QVBoxLayout, QComboBox
import sys

class Hospoda(QWidget):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # Ovládací prvky
        self.availableMeals = QComboBox(self)
        self.tables = QComboBox(self)
        self.orderedMeals = QComboBox(self)
        self.orderButton = QPushButton(
                self, text="Objednat")
        self.orderButton.clicked.connect(
                self.order)
        self.doneButton = QPushButton(
                self, text="Vydat")
        self.doneButton.clicked.connect(
                self.done)

        # Rozložení
        self.layout = QVBoxLayout(self)
        self.menuLayout = QHBoxLayout()
        self.menuLayout.addWidget(
                self.availableMeals)
        self.menuLayout.addWidget(self.tables)
        self.menuLayout.addWidget(
```

[1] `http://doc.qt.io/qt-5/qtcpsocket.html`

```
                self.orderButton)
        self.orderLayout = QHBoxLayout()
        self.orderLayout.addWidget(
                self.orderedMeals)
        self.orderLayout.addWidget(
                self.doneButton)
        self.layout.addLayout(self.menuLayout)
        self.layout.addLayout(self.orderLayout)
        self.setLayout(self.layout)

        # Data
        self.tables.addItems(["u okna",
            "u dveří", "uprostřed", "salonek"])
        self.availableMeals.addItems(
                ["knedlo-zelo-vepřo",
                 "svíčková se šesti",
                 "řízek se salátem"])

        self.show()

    def order(self):
        # Sestavení objednávky
        # z aktuálně vybraných položek
        meal = self.availableMeals.currentText()
        table = self.tables.currentText()
        objednavka = meal + " " + table
        self.orderedMeals.addItem(objednavka)

    def done(self):
        # Smazání aktuálně vybrané položky
        index = self.orderedMeals.currentIndex()
        self.orderedMeals.removeItem(index)

app = QApplication(sys.argv)
hospoda = Hospoda()
app.exec()
```

Všimněte si, že zde poměrně nevybíravě mícháme data s logikou a zobrazováním. To se u jednorázového bastlu dá přežít, pro větší programy si ale příště představíme princip Model–View–Controller (MVC), ve kterém jsou právě tyto tři části odděleny, resp. jeho modifikaci vhodnou pro Qt.

Tentokrát se ale ještě bez teorie obejdeme a budeme pokračovat ve stavbě programu tak, jak nám to přijde pod ruku; v příštím díle se pak dopustíme úklidu. Dovysvětlíme si také, jak se QComboBox používá pořádně, neboť se nám bude pro vysvětlení principu MVC velmi hodit.

Ještě než budeme pokračovat – zatajila jsem vám, že pokud před ukončením spojení pošlete serveru řádek `BYE`, dostanete zpátky řádek `STATS` s jednoduchými statistikami.

Úkol 4 [2b]: Upravte odpojovací metodu z úkolu 2 tak, aby před odpojením ještě poslala `BYE` (nezapomeňte na znak konce řádku!) a zobrazila statistiky v okně.

Zatím si jen tak posíláme se serverem autíčka a chodce sem a tam. Pojďme si napsat program, který už bude skutečně něco simulovat. Začneme mimoúrovňovou křižovatkou, čili nadchodem. Využijeme k tomu informaci o rychlosti aut a chodců, kterou získáme od serveru spolu s autem/chodcem samotným (`speed=`).

```
from PyQt5.QtCore import QTimer
from PyQt5.QtWidgets import \
        QApplication, QWidget, QLabel, \
        QPushButton, QVBoxLayout
from PyQt5.QtNetwork import QTcpSocket

import sys

# Obecný cestovatel
class Traveller:
    def __init__(self, id, speed):
        self.speed = float(speed)
        self.id = int(id)

        self.timer = QTimer()
        self.timer.timeout.connect(self.done)

    # Cestovatel vstupuje do sledovaného úseku
    def start(self, crossing):
        self.crossing = crossing

        # Časovač je v milisekundách
        self.timer.start(1000 * self.roadLength
                         / self.speed)

    # Obsluha časovače: cestovatel opouští úsek
    def done(self):
        self.crossing.sendBack(self)

    # Převod argumentů zpět na řetězec
    def strArgs(self):
        return ("id=" + str(self.id)
                + " speed=" + str(self.speed))

class Car(Traveller):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Sledujeme 500 metrů silnice
        self.roadLength = 500

    def __str__(self):
        return "CAR " + super().strArgs()

class Pedestrian(Traveller):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Sledujeme 100 metrů chodníku
        self.roadLength = 100

    def __str__(self):
        return "PEDESTRIAN " + super().strArgs()

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

        # Výroba ovládacích prvků
        self.connectButton = QPushButton(
                self, text = "Start")
        self.connectButton.clicked.connect(
                self.connect)

        # Rozložení ovládacích prvků
        self.layout = QVBoxLayout(self)
        self.layout.addWidget(
                self.connectButton)

        # Příprava TCP socketu
        self.socket = QTcpSocket(self)
        self.socket.readyRead.connect(self.read)
        self.socket.connected.connect(
                self.connected)
        self.readBuffer = bytearray()

        self.travellers = {}

        # Zobrazení
        self.setLayout(self.layout)
        self.show()

    def connect(self):
        # Nejdřív se odpoj,
```

```
    # pokud už spojení běží
    self.socket.abort()

    # A znovu se připoj
    self.socket.connectToHost(
            "ksp.mff.cuni.cz", 48888)

def connected(self):
    # Pozdravíme server
    self.socket.write("HELLO\n".encode())

def read(self):
    # 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:
        stripped = l.decode().rstrip()
        args = stripped.split(" ")
        travellerType = args.pop(0)
        argmap = dict(map(
            lambda x: x.split("="), args))

        if travellerType == "CAR":
            self.addTraveller(Car(**argmap))
        elif travellerType == "PEDESTRIAN":
            self.addTraveller(
                    Pedestrian(**argmap))

def addTraveller(self, traveller):
    # Uložíme si cestovatele
    self.travellers[traveller.id] = \
            traveller
        # Nechť cestovatel vstoupí do oblasti
        traveller.start(self)

    def sendBack(self, traveller):
        # Cestovatel opouští sledovanou oblast
        self.travellers[traveller.id] = None

        # Vrátíme cestovatele serveru
        text = str(traveller) + "\n"
        self.socket.write(text.encode())

# Spuštění celého programu
app = QApplication(sys.argv)
crossing = Crossing()
app.exec()
```

Výše uvedený program jenom simuluje cestování samotné a řeší komunikaci se serverem. Není tedy vůbec vidět, co se vlastně děje. Váš úkol bude nyní k tomuto napsat úplně jednoduché GUI. Nebojte se během programování používat ladicí výpisy na terminál, odevzdané programy by však neměly obsahovat žádný `print`.

Úkol 5 [2b]: Dopište do programu zobrazování počtu aut a chodců, kteří jsou aktuálně v oblasti.

Úkol 6 [5b]: Zařiďte, aby program v otevřeném okně zobrazoval polohu všech aut i chodců (například v milimetrech od začátku sledovaného úseku) aktualizovanou každých 250 milisekund.

Stejně jako minule, pokud učiníte všechny požadované úkoly v jednom programu, je možné výsledek odevzdat jako řešení všech úkolů dohromady.

V příštím díle se zaměříme na princip Model–View–Controller a budeme pokračovat ve vývoji jednoduchého simulátoru křižovatky.

Maria Matějka

Recepty z programátorské kuchařky: Hledání v textu

Řetězec je v podstatě jakákoliv posloupnost symbolů zapsaná za sebou a s nimi budeme v této kuchařce pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu;“ ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a algoritmů s nimi pracujících) najdeme v biologii. Například DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu do práce s řetězci popíšeme dva *stavební kameny* textových algoritmů, což bude datová struktura pro slovníky – *trie* a vyhledání v textu s předzpracováním hledaného slova a jeho rozšíření pro více slov. S jejich znalostí pak bude mnohem snazší vymýšlet řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen $\{0, 1\}$ pro čísla v binárním zápisu, klasické $\{\texttt{A-Z}, \texttt{a-z}\}$ pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kuchařkách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

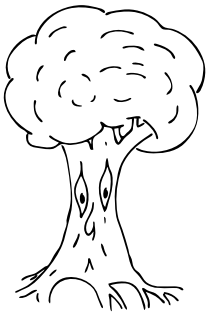
Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme umět rozhodnout, který je menší a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné, a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního $\texttt{0} < \texttt{1}$ se často používá „telefonní“ $\texttt{A} = \texttt{a} < \texttt{B} = \texttt{b} < \ldots < \texttt{Z} = \texttt{z}$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon < \texttt{A} < \texttt{AUTO} < \texttt{AUTOBUS} < \texttt{AUTOGRAM} < \texttt{AUTOR} < \texttt{BAMBITKA} < \texttt{BARNABAS} < \texttt{Z}$.



Adresář pomocí trie

Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

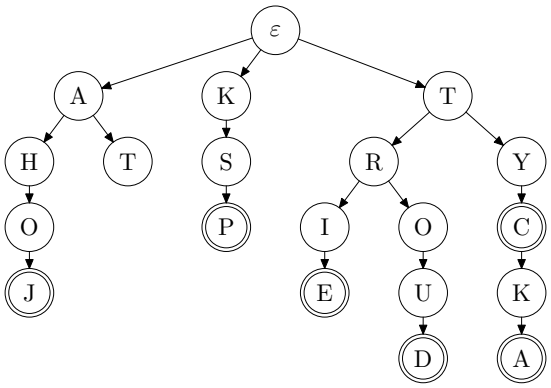
Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[2] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

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

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku, a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova, nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro {`A-Z`, `a-z`} je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba $\{0, 1\}$. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2 |\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log |\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log |\Sigma|)$.

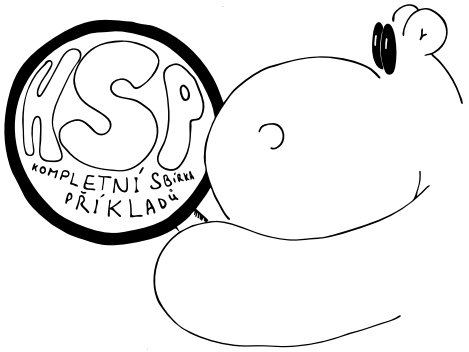
A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti; jednak pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů*.[3]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

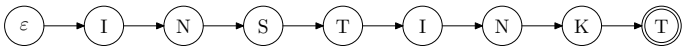


[3] `http://mj.ucw.cz/vyuka/ga/`

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

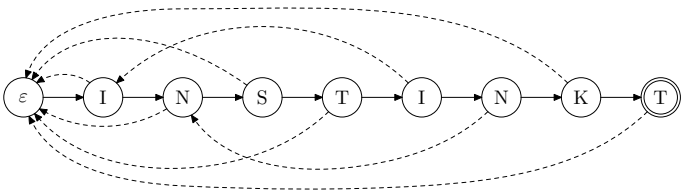
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:

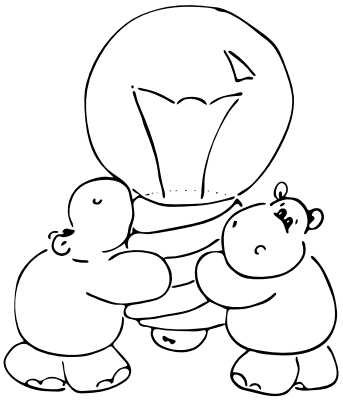


Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, tj. $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.



Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již

máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
jehla = "INSTINKT"
seno = "INSTINSTINKTINSTINKT"
J = len(jehla)
S = len(seno)
F = [None] * J  # Zpětná funkce

def krok(i, znak):
    if i < J and jehla[i] == znak:
        return(i + 1)
    elif i > 0:
        return krok(F[i - 1], znak)
    else:
        return 0

# Konstrukce zpětné funkce
F[0] = 0
for i in range(1, J):
    F[i] = krok(F[i - 1], jehla[i])

# Procházení textu
stav = 0
for i in range(S):
    stav = krok(stav, seno[i])
    if stav == J:
        print(i - J + 1, "až", i)
```

Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* (neboli „okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

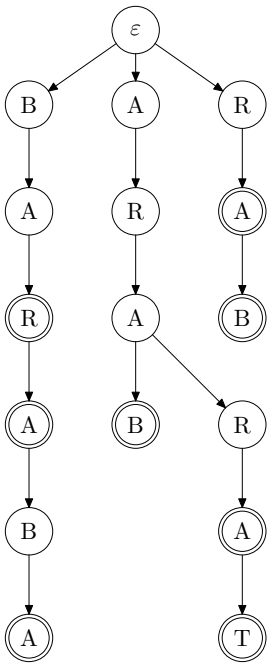
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. V trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

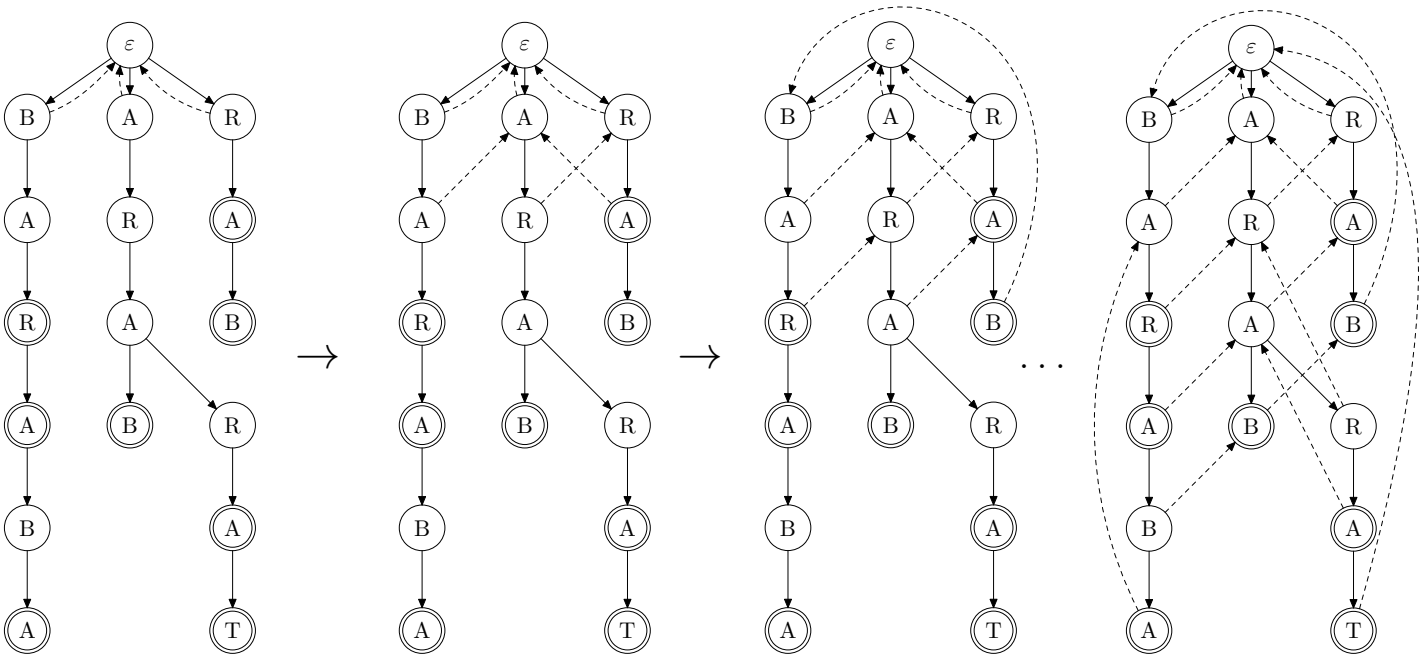
Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé ze slov. Ouha, to ale také nefunguje. Když začneme slovem `BARABA`, a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tak povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

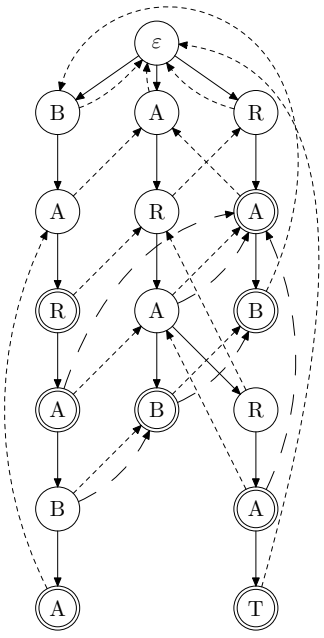


Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

1. $c =$ poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$.

Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.



Zkusme tedy automatem projít text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl algoritmus na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme. Na rozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít všechny suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

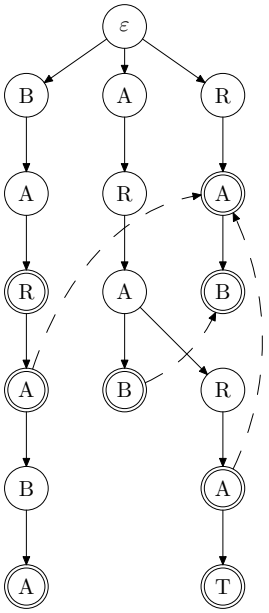
Představme si například slovník obsahující `A` a `AAAA...A` (délky $J-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(S + O)$, resp. $\mathcal{O}(S \cdot \log |\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O + S + J \cdot |\Sigma|)$, resp. $\mathcal{O}(O + (S + J) \cdot \log |\Sigma|)$.



Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

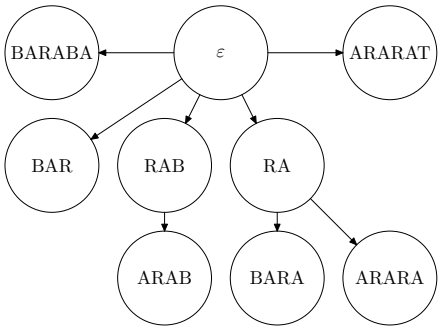
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale

maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si implementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda