Vyčíslitelnost a složitost

HASHIM HABIBALLA - TIBOR KMEŤ

Přírodovědecká fakulta OU, Ostrava

Fakulta prírodných vied UKF, Nitra

Vyčíslitelnost a složitost je pro mnoho studentů informatiky nejtěžší základní partií teoretické informatiky. Zatímco teorie formálních jazyků pracuje s poměrně jednoduchými modely automatů, gramatik a jazyků u vyčíslitelnosti je mnoho těžko pochopitelných vlastností a zejména jejich důkazy a formální pojetí je náročné. Vyčíslitelnost také do značné míry souvisí s vyššími třídami jazyků než jsou regulární nebo bezkontextové. I přesto lze říci, že smysl i základní pojmy této teorie jsou opět velice blízké "selskému rozumu" a informatice v praxi.

Algoritmus je dnes pojmem, který používají nejen informatici. S jistým zjednodušením bychom mohli říci, že algoritmy jsou jádrem informatiky. Čím by byla dnes informatika, kdyby se nesnažila najít postup řešení mnoha problémů od čistě matematických jako je řešení rovnic k ryze praktickým jako jsou algoritmy implementované v informačních systémech, které používáme každodenně (textové editory, tabulkové procesory, databázové prostředky a další).

Abychom však mohli prakticky implementovat, je nutné mít aparát pro jejich zápis, implementaci a používání automatizovanými prostředky (počítači). Samozřejmě, že algoritmem může být chápán i například postup pro přípravu jídla, ale takový vágní popis může někdy stěží zpracovat člověk, natož stroj bez inteligence. Proto je snaha vytvářet umělé jazyky s pevně danou syntaxí a sémantikou, které by popis a implementaci algoritmu umožnily exaktně a jednoznačně. Takových prostředků existuje v informatice velké množství - nepřeberné množství programovacích jazyků vhodných pro různé účely, strojové jazyky či abstraktní a grafické prostředky, používané spíše v teoretickém návrhu nebo výuce.

Půjdeme-li ale ještě dále než k praktickému použití, začnou nás napadat otázky tohoto typu:

- Jaké problémy mohu vlastně pomocí algoritmu vyřešit a jaké již ne?
- Umím například pomocí jazyka Pascal zapsat řešení všech problémů jako strojovým jazykem procesoru počítače a naopak?

- Jak mohu rozpoznat, který algoritmus (program) napsaný pro řešení stejného problému je lepší (např. rychlejší)?
- Lze na takovéto otázky vůbec odpovědět exaktně nebo jen 'hypoteticky'.

Právě na tyto otázky hledá (a již na mnohé našla) odpověď teorie vyčíslitelnosti a složitosti. Aby měly tyto výsledky věrohodnost, je nutné přikročit opět k jisté míře formalizace a používání matematiky, ale v tomto článku se pokusíme ukázat, že základy lze vyložit i názorně.

1 Algoritmy, problémy a jejich řešitelnost

Pokud bychom chtěli najít specifickou činnost, kterou provádí informatik, pak by zřejmě šlo o hledání algoritmického řešení nějakého problému. Toto řešení má tedy splňovat vlastnosti determinismu, obecnosti řešení a rezultativnosti. A právě "problém" je základním teoretickým východiskem vyčíslitelnosti a složitosti. Problémem se zde v užším smyslu myslí otázka a instance (objekt, u kterého tuto otázku zkoumáme). Například můžeme uvažovat problém řešení kvadratické rovnice. Instancí je v tomto případě kvadratická rovnice zadaná kupříkladu ve tvaru $ax^2 + bx + c = 0$ a otázkou - jaké jsou kořeny této rovnice. Pro náš případ bychom našli velice jednoduchý algoritmus, který by kořeny vypočítal. Takovýto problém je tedy **ALGORITMICKY ŘEŠITELNÝ**. Právě algoritmická řešitelnost je klíčovým pojmem zkoumaným exaktně na bázi matematické formalizace. Aby bylo možné se řešitelností efektivně zabývat včetně důkazu vlastností jsou v zásadě třeba dva kroky:

1. Musíme se omezit na speciální případy problémů - tzv. **PROBLÉMY TYPU ANO/NE**. Jde o problémy, které mají otázku formulovanou tak, že na ni lze odpovědět buď ANO nebo NE. Např. výše zmíněný problém řešení kvadratické rovnice by nebyl tohoto typu, neboť odpověď na otázku je složitější objekt (dvě čísla). Pokud bychom chtěli dostat problém typu ANO/NE, museli bychom ho přeformulovat například na otázku, zda existuje reálný kořen. U našeho příkladu bychom pak mohli konstatovat, že jde o problém **ROZHODNUTELNÝ** - tedy takový, pro který máme algoritmus, jenž nám vždy dá správnou odpověď ANO nebo NE. Existují však i problémy, pro které existují pouze algoritmy dávající odpověď ANO avšak nemusejí spolehlivě dávat odpověď v případech NE. Takovéto problémy pak jsou **NEROZHODNUTELNÉ**, a zároveň jsou **ČÁSTEČNĚ ROZHODNUTELNÉ**. O těchto rozhodnutelných, nerozhodnutelných a částečně rozhodnutelných problémech se pak dají dokazovat různé vlastnosti, nicméně je nutné podotknout, že tyto důkazy jsou již často poměrně těžké na pochopení a používají v mnoha případech principu nepřímého důkazu, nikoliv jasné a přímé konstrukce. Pro účely tohoto popularizujícího článku se zmíníme alespoň o jedné lehce pochopitelné a dokazatelné vlastnosti.

Jde o tvrzení známé jako Postova věta. Tvrdí, že pokud máme problém A a jeho doplňkový problém $\overline{A}$ (tedy takový, který má otázku formulovanou opačně - negovaně) a platí, že oba jsou částečně rozhodnutelné, pak problém A musí být rozhodnutelný. Představte si tuto situaci, která by nastala při praktickém programování. Měli byste tedy program A, který dává spolehlivě odpověď ANO pro instance, pro které má být odpověď ano, ale pro instance ostatní nemusí (může se třeba zacyklit v nekonečné smyčce při pokusu odpovědět na otázku). Dále máte program $\overline{A}$, který odpovídá spolehlivě na opačnou otázku opět pouze pro "ANO". Jak můžeme tvrdit, že problém A je rozhodnutelný, tedy že pro něj existuje program, který dává spolehlivě "ANO" i "NE"? Velmi jednoduše, pokud si představíme program, který by po krocích paralelně prováděl instrukce vždy programu A a pak programu $\overline{A}$ (je důležité, aby se vždy provedla jen jedna instrukce). Jelikož možnosti odpovědi jsou jen dvě (ANO, NE), pak máme zaručeno, že buď nám po určitém počtu kroků skončí program A nebo $\overline{A}$, pokud to bude první z nich, pak vrátíme odpověď ANO, protože na otázku A je tato odpověď správná. Pokud však skončí program pro opačnou otázku je odpověď na ni ANO a tedy odpověď na otázku A je NE a tu tedy vrátíme. Takovýto algoritmus vždy skončí, vrátí správnou odpověď a je tedy rozhodnutelný. Kdyby však provádění původních programů nebylo paralelní, pak by tento postup nemusel fungovat - jeden z programů by se mohl zacyklit a již bychom nedostali odpověď.

Vidíte, že otázky, které si klade teorie vyčíslitelnosti a složitosti, nejsou jen odtažitou matematickou látkou, jež studentům informatiky dělá studium náročnější. Tyto otázky totiž informatikovi pomáhají uvědomit si obecné vlastnosti algoritmů a pokud si dokáže promítnout jejich význam pro praktické programování, může to i velmi pozitivně ovlivnit jeho náhled na prakticky řešené problémy.

2. Druhým krokem pro exaktní zkoumání vlastností algoritmů je přesná definice pojmu algoritmus. Již jsme se dotkli problému, že pod pojmem algoritmus si každý čtenář může představit jiný způsob jeho zápisu. Na tomto místě se pokusíme jednoduše a na příkladu ukázat elegantní a přitom velmi jednoduchý způsob - tzv. Turingův stroj (TS). Jeho geniální a přesto prostá myšlenka má svůj původ již ve 30. letech 20. století, kdy jej formuloval Alan Turing (klíčová postava teoretické informatiky). Jde vlastně o automat (viz článek Formální jazyky a automaty uveřejněný v MFI dříve), který však má narozdíl od automatu konečného podstatnou schopnost čtení i zápisu na vstupní pásce, spolu s možností vracet se po potenciálně nekonečné pásce na libovolné místo na ní. Jde tedy opět stroj, který na vstup dostane slovo v určité abecedě, ale narozdíl od konečného automatu může nejen skončit v koncovém stavu z počátečního, ale také může slovo modifikovat a vydat tuto modifikaci jako výsledek. Důležitá je pro jeho funkci správně sestavená přechodová funkce, jenž je formálně zobrazením stavů a symbolů na nový stav, nový zapsaný symbol a příznak posunu čtecí hlavy na pásce (hlava se může posouvat doleva a doprava, případně zůstat na místě). Takovýto Turingův stroj pak můžeme chápat jako prostředek, který realizuje zobrazení, stejně jako jej může realizovat program v Pascalu či strojový jazyk procesoru. Možná se to zdá jako neuvěřitelné, ale i takto jednoduchý formalismus má stejnou výpočetní sílu jako výše zmíněné způsoby zápisu algoritmu. Je sice jasné, že mnoho postupů je v těchto "vyšších" prostředcích již zabudováno a můžeme s nimi pracovat, ale i přesto dokáže

TS realizovat jakoukoliv funkci jako "vyšší" prostředky. TS představuje tedy jakési programování na extrémně nízké úrovni. Výhodou však je, že tato jednoduchost je předurčena pro matematické dokazování vlastností algoritmů. Je totiž velice jednoduchou algebraickou strukturou, kde nejsložitější je přechodová funkce, představující příslušný "program".

Příklad 1:

Uvažujme poměrně jednoduchý problém, kterým je inkrementace čísel v binární soustavě (tedy zvýšení hodnoty čísla x o 1 na x + 1).

Např. 1011 (dekadicky 11) má být inkrementován na 1100 (dekadicky 12).

I v tak jednoduchém prostředků jako je strojový jazyk procesoru existuje prostředek, který nám přímo spočítá součet dvou čísel a tím pádem i dokáže jednou instrukcí inkrementovat hodnotu. V TS však musíme uvažovat na mnohem primitivnější úrovni. Binární číslo je pro nás slovem v abecedě složené ze symbolů 0 a 1. Umíme pouze definovat stavy TS a přechody (instrukce), které říkají jak máme změnit aktuální stav, jaký symbol máme na aktuální místo na pásce zapsat místo původního a kam se máme přesunout. Musíme tedy uvažovat o jednotlivých akcích, které je třeba pro inkrementaci provést. Mělo by jít o tyto činnosti:

1. Přesunout čtecí hlavu na konec slova, neboť tam budeme měnit číslice s nejnižším řádem.
2. Změnit hodnotu nejnižšího řádu z 0 na 1, ale pokud je nejnižší číslice 1, pak musíme provést změnu i ve vyšším řádu a číslici změnit na 0 (přetečení). To se ale může opakovat, takže vlastně musíme najít nejnižší řád, který má číslici 0, kterou pak můžeme změnit na 1 a tím skončit.

Podívejte se na sled těchto akcí na příkladu: **1**011, 101**1**, 10**1**0, 1**1**00

Nyní nám zbývá zkonstruovat TS:

1. Stavy - q_1 - PŘESUN (počáteční stav), q_2 - INKREMENTACE, q_3 - KONEC (koncový stav)
2. Abeceda - 0,1
3. Přechodová funkce (instrukce):
   $1.(q_1, 0) \rightarrow (q_1, 0, P)$, $2.(q_1, 1) \rightarrow (q_1, 1, P)$, $3.(q_1, \varepsilon) \rightarrow (q_2, \varepsilon, L)$ - tyto instrukce představují přesun na konec slova - ε reprezentuje prázdné slovo, tedy symboly za a před slovem, P je přesun hlavy doprava, L přesun doleva a N když hlava zůstává na místě $4.(q_2, 0) \rightarrow (q_3, 1, N)$, $5.(q_2, 1) \rightarrow (q_2, 0, L)$, $6.(q_2, \varepsilon) \rightarrow (q_3, 1, N)$ - postupná inkrementace až nedojde k přetečení

Máme-li sestrojený TS, můžeme jej aplikovat postupným výpočtem na slovo (např. 1011). Výpočet je posloupnost konfigurací TS od počáteční do koncové, kde konfigurace reprezentuje slovo na pásce, v němž je na místě čtecí hlavy symbol aktuálního stavu a znak $\vdash_i$ reprezentuje použití i-té instrukce:

$q_1 1011 \vdash_2 1q_1 011 \vdash_1 10q_1 11 \vdash_2 101q_1 1 \vdash_2 1011q_1 \vdash_3 101q_2 1 \vdash_5 10q_2 10 \vdash_5 1q_2 000 \vdash_4 1q_3 100$

2 Formalizace pojmu algoritmu

Turingův stroj je nejen jednoduchou a přitom zcela exaktní formalizací pojmu algoritmus, ale na druhé straně je i lehce pochopitelný "selským rozumem". Je možné si jej opět jako konečný automat představit i jako fyzický stroj vykonávající instrukce (program). Lze pomocí něj i nahlédnout na zajímavé obecné vlastnosti programů. Jedním z nich je totiž existence tzv. **UNIVERZÁLNÍHO TURINGOVA STROJE (UTS)**. Tento UTS dokáže simulovat libovolný jiný Turingův stroj (pokud se omezíme na jednoduchou abecedu, což ale nesnižuje obecnost). Pokud si opět místo TS představíme například program v Pascalu, pak nám to dává tvrzení, že existuje univerzální pascalovský program, který dokáže simulovat všechny napsané programy v Pascalu. Simulací se zde myslí, že takový univerzální program dostane na vstup kód simulovaného programu, provede ho přesně jako by byl program proveden sám a vrátí výstupy totožné očekávaným výstupům programu. Sestrojit takový univerzální pascalovský program je samozřejmě poměrně složité (i když je to jen otázka času a úsilí), ale právě jednoduchost formalizace TS umožňuje sestrojit takový UTS poměrně rychle a snadno (dokonce to bývá úloha, kterou vysokoškolští studenti řeší jako samostatný domácí úkol!).

Pojem algoritmu je samozřejmě možno formalizovat i jinými prostředky. Jedním z nich je i více matematičtěji orientovaný formalismus, nazvaný jako **PRIMITIVNĚ (OBECNĚ) REKURZIVNÍ FUNKCE (PRF/ORF)**. Tato formalizace bude mít pravděpodobně půvab pro ty čtenáře, kteří jsou více orientovaní na algebraické pojetí vyčíslitelnosti funkcí na množině přirozených čísel. Jejich idea se opírá o dvě základní definice (pro začátek budeme mluvit pouze o primitivně rekurzivních funkcích a později přidáme pojem obecně rekurzivní funkce):

1. Za základní považujeme tyto funkce:

- $o : N \to N, \forall x : o(x) = 0$ *(funkce, která vrací pro jakýkoliv argument 0 - identická nula)*
- $s : N \to N, \forall x : s(x) = x + 1$ *(funkce, která vrací pro jakýkoliv argument jeho následující hodnotu - následník)*

- $I_i^{(n)} : N^n \to N, \forall x_1, x_2, ..., x_n : I_i^{(n)}(x_1, x_2, ..., x_n) = x_i$ *(funkce, která vrací i-tý argument - výběr - je nutná pro tvorbu funkcí více proměnných)*

2. Další funkce můžeme skládat pomocí operátorů:

- Operátory substituce S_n^m: $f = S_n^m(g, h_1, ..., h_m)$, kde platí
  $f(x_1, x_2, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_m(x_1, ..., x_n))$ *(operátor, který umožňuje skládat funkce)*

- Operátory primitivní rekurze R^n: $f = R^n(g, h)$, kde platí
  $f(0, x_2, ..., x_n) = g(x_2, ..., x_n)$ a
  $f(k+1, x_2, ..., x_n) = h(k, f(k, x_2, ..., x_n), x_2, ..., x_n)$ *(operátor, který definuje rekurzivní funkci na základě funkce g - zarážka rekurze pro k = 0, h - následující krok rekurze pro k + 1 definovaný pomocí k)*

Z těchto základních funkcí můžeme pomocí postupné aplikace operátorů vytvářet složitější funkce.

Příklad 2:

Zkusme se podívat na příklad funkce $f(x_1, x_2) = x_1 + x_2$. Podobně jako Turingův stroj je tento formalismus poměrně primitivní, takže i takto jednoduchou funkci zde musíme sestavit. Myšlenka spočívá v tom, že použijeme rekurzi (která nám nahrazuje cyklus) a pomocí rekurze vždy vytvoříme následníka hodnoty až k nule, kdy dosadíme hodnotu druhého argumentu.

Sekvence vypadá následovně:
$f_1 = s, f_2 = I_2^3, f_3 = S_3^1(s, I_2^3), f_4 = I_1^1, f = R(f_4, f_3)$
přičemž jednotlivé funkce vyjadřují:

- f - je funkce, kterou jsme chtěli vytvořit (sčítání dvou argumentů), přičemž jsme museli aplikovat rekurzi následujícím způsobem; $f(0, x_2) = x_2, f(k+1, x_2) = f(k, x_2)+1$; což znamená, že x_1 - krát zvýšíme hodnotu x_2 o jedna a použijeme k tomu upravenou funkci následníka

- f_4 - je funkcí výběru prvního argumentu z jednoho, kterou potřebujeme, abychom správně nadefinovali zarážku rekurze funkce f

- f_3 - je upravená funkce následníka pro druhý argument ze tří (použije se v rekurzi dle formální definice operátoru); $f_3(x_1, x_2, x_3) = x_2 + 1$

- f_2 - je potřebná pro vytvoření následníka druhého argument v f_3; jde o výběr druhého argumentu ze tří $f_2(x_1, x_2, x_3) = x_2$

- f_1 - je základní funkcí následník pro jednu proměnnou ($f_1(x_1) = x_1 + 1$)

Na tomto jednoduchém příkladě jste viděli, že notace primitivně rekurzivních funkcí umožňuje možná poněkud složitě, ale zejména exaktně symbolicky odvodit jakoukoliv funkci. I když je tato konstrukce značně odlišná od formalizace algoritmické vyčíslitelnosti pomocí TS, v konečném důsledku můžeme realizovat přesně Turingovsky vyčíslitelné funkce pouze pokud k definici primitivně rekurzivních funkcí přidáme jeden operátor - tzv. operátor minimalizace, čímž dostaneme obecně rekurzivní funkce. Jelikož je přesná formální definice poněkud složitější, omezíme se na jeho vysvětlení laické. Jde o operátor, který umožňuje vytvořit funkci, která vrací nejmenší hodnotu prvního argumentu, kdy je funkční hodnota 0. Například, kdybychom potřebovali při výstavbě určité funkce znát nejmenší hodnotu funkce $f(x_1) = 5 - x_1$ aplikovali bychom operátor minimalizace, který by vytvořil funkci vracející vždy 5. Samozřejmě by to asi v takto jednoduchém příkladě nedávalo smysl, ale tento operátor lze definovat pro libovolný počet proměnných a libovolnou složitost formule.

Další velice zajímavou formalizací pojmu algoritmu jsou takzvané PL-programy. Zmiňujeme se o nich ihned po rekurzivních funkcích nikoliv náhodou. Jak uvidíte, i když jde opět o formalizaci z naprosto jiného pohledu, lze jednoduše ukázat, že jejich výpočetní síla je totožná. PL-programy (jak již název napovídá) jsou formalizací, kterou zřejmě ocení spíše programátorsky orientovaní čtenáři. PL-program je sekvence příkazů, které mohou obsahovat identifikátory (proměnné). Jazyk je opět velmi jednoduchý, v zásadě máme pouze tyto příkazy a elementy:

- Přiřazovací příkaz $X := 0$ (lze přiřadit nulu libovolnému identifikátoru)
- Příkaz inkrementace INC X ($X := X + 1$)
- Příkaz cyklu LOOP X [seznam příkazů] ENDLOOP (provede seznam příkazů X-krát)
- Návěští L, které určuje bod programu
- Příkaz skoku GOTO L (provede v programu skok do bodu L)
- Specifikace vstupů a výstupu INPUT X_1, X_2, OUTPUT Y

Příklad 3:
Například funkci $f(x_1, x_2) = x_1 + x_2$ lze zapsat následujícím PL-programem:

```
{INPUT X1, X2}
Y := 0;
LOOP X1 INC Y; ENDLOOP
LOOP X2 INC Y; ENDLOOP
{OUTPUT Y}
```

Pravděpodobně se Vám zdá, že tento způsob zápisu má podobné prvky jako rekurzivní funkce. Zamyslíte-li se na jednotlivými elementy, zjistíte například že:

- Přiřazovací příkaz je v podstatě identická nula

- Příkaz inkrementace je vlastně funkcí následníka
- Příkaz cyklu může beze zbytku nahradit rekurzi

Také lze ukázat, že bez příkazu GOTO budou PL-programy mít stejnou výpočetní sílu jako PRF, jinak jsou ekvivalentní ORF. Dále lze jednoduše simulovat libovolný PL-program pomocí Turingova stroje. Vlastně bychom jen na pásce TS vyhradili místo pro hodnoty proměnných a postupně naprogramovali v TS všechny příkazy (viz ukázka TS - následník).

Všechny tyto vlastnosti nejsou jen matematickou teorií, která zastřešuje pojmy a metody, které využíváme v informatice. Studium těchto formalizací vám umožní pochopit, že způsobů zápisu algoritmu je mnoho a i přes jejich různorodost mohou mít stejnou vyjadřovací/výpočetní sílu. Je to podobné jako, když jeden programátor rád píše své programy v jazyce Pascal a jiný v jazyce C. I když techniky v jednotlivých jazycích se mohou lišit, oba programátoři mohou vytvořit plnohodnotné programy, které se budou chovat identicky. Také díky objevování těchto teoretických otázek můžeme hlouběji pochopit proč rekurze a cyklus jsou dva navzájem zaměnitelné nástroje algoritmického řešení problémů. Důležité je, že tyto notace jsou poměrně pochopitelné a lze si k nim vytvářet mnoho jednoduchých i složitějších příkladů, provádět převody mezi jednotlivými způsoby formalizace a tím vytvářet lepší úroveň "informatického myšlení". Pod tímto pojmem si představujeme schopnost navrhovat efektivní algoritmické řešení problémů, což je uvažování, které by mělo být specifickým rysem každého informatika.

3 Složitost řešení problému

Když už mluvíme o efektivním řešení problémů musíme se alespoň krátce zmínit o druhé stránce oboru, kterým se zabývá tento článek. Je jí takzvaná složitost, čímž můžeme chápat především dvě odlišné vlastnosti algoritmů - buď časovou složitost nebo prostorovou složitost. Představte si, že několik studentů dostane za úkol samostatně vyřešit jistý algoritmický úkol. Budeme-li předpokládat, že všichni jej vyřeší správně a sestaví funkční algoritmy (programy) je pravděpodobné, že každý navrhne trochu jiný způsob řešení. U těchto programů pak lze položit otázku, který je vlastně nejlepší. A právě na tuto otázku lze (mimo jiné) nahlížet ze dvou hledisek:

1. který program dá v průměru nejrychleji výsledek?
2. který program spotřebuje ke svému správnému chodu nejméně paměti (prostoru)?

Jistě je odpověď na tyto otázky důležitá. Vždyť aby nám vůbec výpočetní technika v našem životě byla užitečná, musí pracovat rozumně dlouho a na strojích s kapacitou, které

máme v dané době k dispozici. Aby bylo možné zadefinovat exaktně tyto pojmy používá se opět exaktní pojem algoritmu. Pro naše účely zvolme Turingův stroj.

U něj můžeme rozlišovat tyto základní charakteristiky:

1. **Složitost (časová) výpočtu TS** na určitý vstup - jde o počet vykonaných instrukcí TS na tento vstup (je to tedy prosté přirozené číslo);
   např. v příkladu 1 v tomto textu byl uveden výpočet Turingova stroje na slovo 1101, počet vykonaných instrukcí než se stroj zastavil je 8.

2. **Složitost (časová) TS (obecně)** - zde jde již o funkci, kde argument je velikost slova a funkční hodnota je počet instrukcí na slovo o této velikosti v nejhorším možném případě (je jasné, že různých slov o určité velikosti je mnoho - viz příklad 1);
   pro náš příklad 1 bychom opět mohli přemýšlet, jaká funkce to je. Stroj funguje tak, že vždy dojede na konec slova (což je n instrukcí), dále v nejhorším případě projde celé binární číslo (slovo) pozpátku až na začátek před první symbol a skončí v koncovém stavu(n + 2 kroků). Složitost toho TS je tedy $f(n) = n + n + 2 = 2n + 2$

3. **Odhad (časové) složitosti TS** je zjednodušením funkce složitosti TS. Většinou nás totiž nezajímá přesná funkce jako ve výše uvedeném případě, ale stačí nám jen ta nejdůležitější část funkce (kterou označíme $O(f)$), která roste nejrychleji, vzhledem k velikosti vstupního slova. Většinou se za tyto funkce berou $f(n) = n$, $f(n) = n^2$, ... atd.
   Tedy u našeho příkladu by odhad byl $O(n)$.

4. **Třída (časové) složitosti** je množina těch problémů, které jsou vyčíslovány nějakým TS, se složitostí $O(f)$. Většinou se obecně vymezují třídy lineární složitosti $(T(n))$, logaritmicko-lineární $(T(n \cdot log(n)))$, kvadratické $(T(n^2))$, kubické $(T(n^3))$, ..., polynomiální $(T(n^k))$ (PTIME), exponenciální $(T(2^n))$ (EXPTIME). Například do třídy $(T(n))$ patří náš problém inkrementace binárního čísla.

Druhé hledisko tedy prostorová složitost může být definována v podstatě totožně, pokud složitost výpočtu TS deklarujeme jako počet symbolů pásky, které během výpočtu projde TS než se zastaví. Třídy prostorové složitosti se pak označují podobně s příponou SPACE- PSPACE, EXPSPACE. Třídy časové složitosti a prostorové složitosti pak můžeme uspořádat a intuitivně dokázat některé vlastnosti. Například je jasné, že když nějaký problém patří do určité třídy časové složitosti, pak určité patří do příslušné třídy prostorové složitosti (např. PTIME $\Rightarrow$ PSPACE). Proč tomu tak je? Vysvětlení je jednoduché. Turingův stroj nemůže za "polynomiálně mnoho" času projít více než "polynomiálně mnoho" symbolů, neboť se může při jedné instrukci s hlavou posunout o více než jeden symbol.

V informatice je na časové složitosti založen jeden významný problém, který je jen zdánlivě pouze teoretický. Ve skutečnosti je však více "ze života" než si uvědomujeme. Jde o takzvaný **P-NP PROBLÉM**. TS má podobně jako konečný automat také svou nedeterministickou verzi. Nedeterminismus spočívá ve více možnostech při přechodu z určité kombinace - stav, symbol, což je situace běžnému algoritmickému myšlení cizí, nicméně z

hlediska složitosti umožňuje příslušnost do efektivnější třídy nedeterministické složitosti. Vezměme si příklad problému obchodního cestujícího. Spočívá v hledání nejefektivnější (nejkratší) cesty mezi několika městy, přičemž chceme navštívit všechna města právě jednou a vrátit se do výchozího. Jediné "klasické deterministické" řešení, které je možné, spočívá v zásadě vždy ve zkoušení všech možných posloupností (existují i vylepšení, která částečně zlepšují složitost, nikoliv ale v principu). Tedy takové algoritmy patří do třídy (EXPTIME). Nicméně s pomocí nedeterministického TS můžeme tento problém řešit ve třídě NPTIME (nedeterministická polynomiální složitost). Teoretický trik (byť v praxi prozatím nic nepřinášející) je v tom, že takový nedeterministický stroj by prostě mohl z každého města zvolit jakoukoliv možnost a jelikož nás nezajímá, jak dokáže TS tuto nejlepší možnost najít, je jeho složitost lineární (tu správnou možnost TS prostě uhodne).

Proč je to pro praxi důležité? Kdyby se podařilo dokázat, že třída PTIME = NPTIME (jinak řečeno by pro každý problém, který lze řešit nedeterministicky, existoval i deterministický polynomiální algoritmus), pak bychom našli polynomiální deterministický algoritmus pro řešení optimalizačních problémů. Tyto problémy se v každodenním životě řeší často (kupříkladu sestavení rozvrhu hodinu na škole). Jelikož je však umíme řešit pouze v exponenciální čase, jsou při velkém rozsahu nezvládnutelné. Sestavíme jinak řečeno algoritmus, který pro malý počet vstupních prvků (třeba měst u problému obchodního cestujícího) funguje, ale existuje jistá mez, kdy algoritmus bude trvat neúnosně dlouho a nepomůže nám ani zvýšení výkonu stroje, který algoritmus realizuje. Na vině je totiž ona exponenciální funkce složitosti, kdy růst této funkce je vždy od určité hodnoty příliš strmý.

4 Závěr

Pokusili jsme se na poměrně malém prostoru ukázat na základní problémy teorie vyčíslitelnosti a složitosti, a to s ohledem na jejich význam pro vzdělávání informatiků a to nejen na úrovni vysokého školství. Problémy, které zkoumá tato teorie, sice vypadají odtažitě, složitě a značně formalizované, ale jejich význam pro schopnost algoritmicky myslet, uvědomovat si řešitelnost problémů a efektivitu řešení je velký.

Literatura

[1] KUČERA M. Kombinatorické algoritmy, Matematický seminář SNTL 1989, Praha

[2] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and

Computation. Addison-Wesley, Reading (Mass.), 1979

[3] PAVLISKA, V. Vyčíslitelnost a složitost I. Ostravská univerzita, Ostrava, 2003

[4] CHYTIL, M. Automaty a gramatiky. Matematický seminář SNTL Praha, 1984

[5] JANČAR, P. Vyčíslitelnost a složitost. VŠB TU Ostrava, http://www.cs.vsb.cz/jancar/ (2004)

[6] KOUBKOVÁ, A., PAVELKA, J. Úvod do teoretické informatiky. Matfyz press, Praha 1998.

Kontaktní adresa:

Hashim Habiballa
katedra informatiky a počítačů
Přírodovědecká fakulta Ostravské Univerzity
30.dubna 22, 701 03 Ostrava 1,
tel.: +420596160213,
e-mail: habiballa@volny.cz, www: http://www.volny.cz/habiballa/

Tibor Kmeť
katedra informatiky
Fakulta prírodných vied UKF
Tr. A. Hlinku 1
949 74 Nitra