Pracovní text ke kursu VYČÍSLITELNOST A SLOŽITOST I v ZS 1996/97
P. Jančar

1 Vyčíslitelnost

Nejdříve se budeme věnovat oblasti vyčíslitelnosti. Základní otázkou, na kterou tato disciplína hledá odpověď, zní:

Co všechno je a co není algoritmicky vyčíslitelné (řešitelné) ?

Jinak řečeno

Ke kterým problémům existují algoritmy, jež je řeší, a ke kterým problémům takové algoritmy neexistují ?

Abychom takovou otázku mohli rozumně zodpovědět, je mj. třeba podrobněji specifikovat, co rozumíme pod slovem problém. Význam tohoto slova v přirozeném jazyce je velmi široký (může se jednat jak o problém řešení kvadratické rovnice, tak např. o milostný problém studenta Ostravské univerzity apod.). Zde se pochopitelně soustředíme na problémy, jejichž podstata umožňuje vůbec uvažovat o potenciálním nasazení algoritmických metod, potažmo počítačových programů. (Proto je nám zde bližší spíše problém kvadratické rovnice než druhý zmíněný problém.)

Je rozumné se shodnout na tom, že každý uvažovaný problém lze popsat následujícím schématem

Problém XY (označení, či výstižný název problému)

Instance: zde je popsáno, co může být zadáním (vstupem) u našeho problému (např. libovolná kvadratická rovnice)

Výsledek: zde je popsáno, jak vypadají výsledky (výstupy), a jak se žádaný výsledek má vztahovat k zadané instanci (např. to mají být kořeny zadané rovnice)

Budeme předpokládat, že výsledek je jednoznačně určen zadanou instancí (nebudeme tedy připouštět výsledek typu 'nějaký z kořenů dané rovnice' apod.).

Dále budeme (celkem přirozeně) předpokládat, že jakoukoli instanci problému, stejně jako jakýkoli výsledek, lze zadat řetězcem (posloupností symbolů) v nějaké konečné abecedě (obsahující např. písmena, číslice, interpunkční znaménka a případné další symboly).

Poznámka. Všimněme si, že v tomto smyslu není uvedený problém kvadratické rovnice ještě jednoznačně specifikován (jak zadáme konečným řetězcem libovolné reálné číslo ?). Pokud se např. omezíme na racionální koeficienty a kořeny budeme uvažovat zaokrouhlené na určitý počet desetinných míst, nabízí se už běžný popis koeficientů a kořenů v desítkové soustavě.

Pozastavme se ještě nad problémem abecedy – množiny symbolů, pomocí nichž jsou popsány instance a výsledky. Je možné např. ke každému problému uvažovat jeho specifickou abecedu. Je ale také možné vzít jednu (univerzální) abecedu a v ní popisovat instance a výsledky kteréhokoli problému. Např. je možné se omezit na znaky ASCII. Od této představy je už jen krůček k uvědomění si, že jako ona univerzální může vlastně stačit dvouprvková abeceda $\{0,1\}$! Zápisy v ní sice pro nás asi nebudou příjemně čitelné, nicméně všichni víme, jak zkonstruovat jednoduché algoritmy (prográmky) převádějící řetězce (texty) z naší oblíbené abecedy do abecedy $\{0,1\}$ a zpět – pro studované otázky algoritmické rozhodnutelnosti tedy nic neztratíme, omezíme-li se na onu dvouprvkovou abecedu.

Na problém ve výše uvedeném smyslu se lze dívat jako na zobrazení typu $\Sigma^* \longrightarrow \Sigma^*$ pro danou abecedu Σ (ve světle předešlého odstavce si zde vždy lze za Σ dosadit abecedu $\{0,1\}$), které každé (povolené) instanci problému (resp. jejímu popisu v dané abecedě) přiřazuje žádaný výsledek (resp. jeho popis v dané abecedě); k řetězci ze Σ^*, který nepředstavuje (povolenou) instanci problému, je možno přiřadit nějaký speciální výsledek (znamenající 'Nekorektní zadání'). Je možné ale tohle neudělat a pro nepovolené instance prostě výsledek nedefinovat. Příslušné zobrazení $\Sigma^* \longrightarrow \Sigma^*$ se pak chápe jako částečné (tj. nedefinované pro všechny řetězce ze Σ^* – na rozdíl od obvyklého totálního zobrazení definovaného všude).

Speciálními problémy budou *problémy typu ANO/NE*, kde žádaný výsledek (příslušný k povolené instanci) je prvek dvouprvkové množiny (např. $\{ANO, NE\}$, či $\{1,0\}$). Takový problém se nejčastěji zadává následujícím schématem.

> *Problém XY* (označení, či výstižný název problému)
>
> *Instance:* zde je popsáno, co může být zadáním (vstupem) u našeho problému (např. libovolná kvadratická rovnice)
>
> *Otázka:* zde je otázka (vztahující se k instanci), na niž je vždy odpověď ANO nebo NE (např. existuje kořen zadané rovnice ?)

Problém tohoto typu se často ztotožňuje s množinou právě těch řetězců v dané abecedě, které popisují ty instance problému, jimž je přiřazena odpověď ANO. Jedná se tedy vlastně o určitý jazyk v abecedě Σ (tj. o určitou podmnožinu množiny Σ^*).

Nyní, po upřesnění toho, co rozumíme pod pojmem 'problémy', vymezíme jejich vlastnosti, která nás zde zajímají.

Definice 1.1 *O daném problému řekneme, že je* algoritmicky řešitelný *(neboli odpovídající (částečné) zobrazení* $\Sigma^* \longrightarrow \Sigma^*$ *je* algoritmicky vyčíslitelné*), jestliže existuje algoritmus, který je schopen jako vstup přijmout libovolnou instanci daného problému a jeho výpočet pro libovolný takový vstup vždy skončí, přičemž výstupem bude požadovaný výsledek.*

Je-li problém chápán jako výše zmíněné *částečné* zobrazení $\Sigma^* \longrightarrow \Sigma^*$, pak se z technických důvodů obvykle požaduje, že příslušný algoritmus se pro nekorektní zadání (tj. pro

řetězec, jenž nepopisuje instanci problému) nezastaví (a jeho výstup není v tom případě definován).

My se budeme hlavně zajímat o problémy typu ANO/NE, s kterými se snadněji pracuje, a na něž se úvahy o obecných problémech dají převést. Pojem algoritmické řešitelnosti pro ně vyjadřujeme následovně:

Definice 1.2 *O daném problému typu ANO/NE řekneme, že je* algoritmicky rozhodnutelný, *pro stručnost* rozhodnutelný, *jestliže existuje algoritmus, který je schopen jako vstup přijmout libovolnou instanci daného problému a jeho výpočet pro libovolný takový vstup vždy skončí, přičemž výstupem bude požadovaná odpověď ANO či NE.*

Celkem přirozeně lze tato definice přenést na jazyky, tedy množiny řetězců v dané abecedě:

Definice 1.3 *Jazyk L v abecedě Σ je* rozhodnutelný, *tj. množina $L \subseteq \Sigma^*$ je* rozhodnutelná, *jestliže problém příslušnosti k L je rozhodnutelný (Instance: $w \in \Sigma^*$; Otázka: Platí $w \in L$?).*

Budeme předpokládat, že pro libovolný uvažovaný problém P s 'popisnou' abecedou Σ je následující ANO/NE problém daný schématem

Instance: $w \in \Sigma^*$; *Otázka:* Je w (korektním) popisem instance problému P ?

rozhodnutelný. Pak je zřejmé, že ANO/NE problém je rozhodnutelný právě tehdy, když jemu příslušný jazyk (sestávající ze všech instancí na které je odpověď ANO) je rozhodnutelný (tzn. že v tomto smyslu nedefinovanost problému pro případné nekorektní instance nehraje roli).

Bude se nám hodit i související pojem částečné rozhodnutelnosti:

Definice 1.4 *O daném problému typu ANO/NE řekneme, že je* částečně rozhodnutelný *(neboli příslušný jazyk v abecedě Σ je* částečně rozhodnutelný, *tj. daná podmnožina množiny Σ^* je* částečně rozhodnutelná*), jestliže existuje algoritmus, jehož výpočet skončí právě pro takové vstupy, které odpovídají instancím daného problému s odpovědí ANO.*

Poznámka. Každé tvrzení týkající se (částečné) rozhodnutelnosti problémů lze přeformulovat pro jazyky, tedy množiny, a naopak. Čtenář je vyzýván, ať to má neustále na paměti (důkladně uvědomit si to může např. na následujících tvrzeních, která jsou explicitně formulována jen pro množiny).

Všimněme si následujících vztahů mezi uvedenými pojmy.

Tvrzení 1.5 *Je-li množina $A \subseteq \Sigma^*$ rozhodnutelná, pak je i částečně rozhodnutelná.*

Tvrzení 1.6 *Množina $A \subseteq \Sigma^*$ je rozhodnutelná právě když $\overline{A}$ (doplněk A, tj. $\Sigma^* \setminus A$) je rozhodnutelná.*

Věta 1.7 *(Post) Množina $A \subseteq \Sigma^*$ je rozhodnutelná právě když A i $\overline{A}$ jsou částečně rozhodnutelné.*

Důkaz obou tvrzení, a tím i důkaz jednoho směru Postovy věty, by měl být nabíledni. Pro opačný směr věty si stačí uvědomit, že potřebný algoritmus (rozhodující A) prostě 'paralelně spustí' (tj. např. střídavě krok po kroku simuluje) dva příslušné algoritmy (částečně rozhodující A a $\overline{A}$), 'počká' až jeden z nich skončí a poté vydá příslušnou odpověď.

Nyní už má pro nás základní otázka formulovaná

Které problémy jsou algoritmicky řešitelné (či rozhodnutelné) ?

podstatně konkrétnější smysl. Opírá se ovšem stále o intuitivní pojem 'algoritmus', který je pro naše úvahy nutné formalizovat, zpřesnit.

Poznámka. Všimněte si, že Postovu větu jsme prokázali bez dalšího zpřesnění pojmu 'algoritmus'. Opírali jsme se zde konkrétně např. o to, že ke každým dvěma algoritmům lze navrhnout třetí algoritmus, který ty dva 'paralelně' (či 'střídavě sekvenčně') provádí – to jsme vzali jako intuitivně zřejmý fakt. Ve skutečnosti se už dostáváme na trochu nebezpečnou půdu a měli bychom si začít uvědomovat, proč asi je zpřesnění pojmu 'algoritmus' potřebné.
Markantnější to bude na následujícím příkladu: později naznačíme důkaz toho, že částečně rozhodnutelné jazyky jsou přesně jazyky typu 0 z Chomského hierarchie (generované obecnými generativními gramatikami). Když bychom měli předvést, že každý částečně rozhodnutelný jazyk je generován zmíněnou gramatikou, budeme zpřesnění pojmu algoritmus potřebovat jako sůl.
Nejdůležitějším motivem je ovšem snaha prokázat algoritmickou nerozhodnutelnost konkrétních problémů – zde se již bez zmíněné formalizace nemůžeme obejít.

O avízovanou formalizaci se pokusil v 30. letech mj. anglický matematik Alan Turing, pro jehož formalizaci pojmu algoritmus se brzy vžil název 'Turingův stroj'. Zde toto 'zařízení' nedefinujeme, předpokládáme, že s ním je čtenář obeznámen (a že bere jako zřejmé, že každý Tur. stroj je příkladem algoritmu; o opačném směru bude řeč níže).

Úmluva. Řekneme-li, že Turingův stroj má abecedu Σ (např. $\{0,1\}$), zařazujeme do Σ jen 'normální' (tj. neprázdné) znaky; automaticky budeme předpokládat, že instrukce počítají i se speciálním prázdným znakem (někdy se označuje #), který představuje prázdné políčko pásky. Navíc budeme z technických důvodů předpokládat, že v koncové konfiguraci tvoří neprázdné znaky na pásce vždy *souvislý* úsek.

Uveďme následující přirozené definice (využívající předchozí úmluvu).

Definice 1.8 *Turingův stroj M s abecedou Σ* realizuje (částečné) zobrazení $\mathcal{M} : \Sigma^* \longrightarrow \Sigma^*$ *definované následovně pro libovolné $w \in \Sigma^*$: zastaví-li se M pro vstup w (tedy skončí-li jeho výpočet, začne-li se slovem w na pásce), pak $\mathcal{M}(w)$ je definováno a rovná se řetězci (souvislému úseku neprázdných znaků), který je obsahem pásky v příslušné koncové konfiguraci; nezastaví-li se M na w, pak $\mathcal{M}(w)$ definováno není.*

Tur. stroj M přijímá (akceptuje, částečně rozhoduje) jazyk (množinu) $L \subseteq \Sigma^*$, *jestliže pro lib. $w \in L$ se zastaví a pro lib. $w \notin L$ se nezastaví.*

Tur. stroj M rozhoduje jazyk (množinu) $L \subseteq \Sigma^*$, *jestliže pro lib. $w \in \Sigma^*$ se zastaví a rozhodne (např. konc. stavy q_{ANO}, q_{NE}), zda $w \in L$ či nikoliv.*

Všimněte si, že v definicích 1.1, 1.2, 1.3 můžeme pojem 'algoritmus' nahradit pojmem 'Turingův stroj'; dostaneme pak definice 'turingovsky vyčíslitelných (částečných) zobrazení', 'turingovsky rozhodnutelných jazyků (množin)' apod.

Poznámka. Je vhodné říci, že jako synonymum k pojmu 'turingovsky vyčíslitelné (zobrazení)' se užívá pojem *částečně rekurzivní (funkce)*, synonymem k 'turingovsky rozhodnutelný jazyk (množina)' je *rekurzivní jazyk (množina)* a místo 'turingovsky částečně rozhodnutelný jazyk (množina)' se užívá *rekurzivně spočetný jazyk (množina)*.

Různých jiných formalizací pojmu algoritmus se v 30. letech a později objevilo více (např. Postovy systémy, Markovovy algoritmy, kalkul částečně rekurzivních funkcí atd.; nám je dnes asi nejbližší ztotožnění pojmu algoritmus s pojmem počítačový program – např. program v jazyce Pascal). Všechny tyto formalizace se ukázaly být ekvivalentní (umějí realizovat tatáž zobrazení $\Sigma^* \longrightarrow \Sigma^*$, resp. přijímat (rozhodovat) tytéž jazyky – podmnožiny Σ^*). Tato ekvivalence se většinou dá ukázat předvedením schopnosti simulovat jeden prostředek jiným. V našem kursu to ilustrujeme prokázáním ekvivalence různých modifikací základního modelu Turingova stroje (např. stroje s oboustranně nekonečnou páskou, více pásek, vícestopé pásky, nedeterministické stroje (akceptory) atd.) a také např. ekvivalencí se zásobníkovým automatem s dvěma zásobníky.

Speciální význam má pro nás fakt, že libovolný Turingův stroj M lze simulovat Turingovým strojem M' s abecedou $\{0, 1\}$. (Předpokládáme, že čtenáři je toto naprosto jasné.)

Prokázání zmíněných ekvivalencí a další velmi dobré důvody nás vedou k přijímání tzv. *Churchovy (či Church-Turingovy) teze*

> *Ke každému algoritmu je možné zkonstruovat s ním ekvivalentní Turingův stroj (při vhodném vyjádření vstupů a výstupů jako řetězců v určité abecedě); ekvivalencí zde rozumíme podmínku, že algoritmus i Turingův stroj se zastaví (tj. jejich běh, výpočet, se zastaví) právě pro tytéž vstupy, přičemž pro tyto vstupy budou příslušné výstupy totožné.*

Jinými slovy: každý rozhodnutelný problém (jazyk, množina) je turingovsky rozhodnutelný (tj. rekurzivní); podobně každý částečně rozhodnutelný problém (jazyk, množina) je turingovsky částečně rozhodnutelný (tj. rekurzivně spočetný).

(Všimněte si, že jelikož každý Turingův stroj je příkladem algoritmu, je obrácené tvrzení zřejmé.)

Churchovu tezi není možné dokázat jako matematickou větu (právě kvůli 'intuitivnímu charakteru' pojmu algoritmus); jelikož však máme velmi dobré důvody k jejímu přijetí, často pojmy 'rekurzivní' a 'rozhodnutelný' ztotožňujeme. Níže zmíníme důkazy algoritmické nerozhodnutelnosti konkrétních problémů; ve skutečnosti exaktně dokážeme jen to, že nejsou rekurzivní (tj. turingovsky rozhodnutelné) – v tvrzení se tedy implicitně odvoláváme na Churchovu tezi, čehož bychom si měli být vědomi.

Upozornění. V textu se budeme držet 'všeobecnějších' pojmů jako je 'rozhodnutelný', 'částečně rozhodnutelný' i tam, kde by z hlediska matematické exaktnosti mělo být raději 'rekurzivní', 'rekurzivně spočetný'. Bylo by dobré, aby si toto čtenář aspoň na několika případech důkladně promyslil. Dobrým cvičením může být příklad Postovy věty: promyslete si její znění 1.7 ve vztahu ke znění 'A je rekurzivní právě když A i $\overline{A}$ jsou rekurzivně spočetné'. (Čím se liší důkazy věty v jednotlivých zněních ? Proč jde vlastně o jednu a tutéž větu ?)

Poznámka. Už jsme si dříve zmínili, že jazyky typu 0 v Chomského hierarchii (jazyky generovatelné obecnými generativními gramatikami) jsou právě částečně rozhodnutelné jazyky. K dané obecné gramatice je snadné navrhnout např. nedeterministický Turingův stroj, který akceptuje (tj. má možnost se zastavit pro) právě ta slova, jež lze nagenerovat gramatikou. Pro opačný směr nyní (v souladu s Churchovou tezí) stačí ukázat, jak k danému Turingovu stroji (s poč. stavem q_0) zkonstruovat gramatiku, jež např. nejprve vygeneruje slovo tvaru $w\$q_0w\$$ ($q_0, \$$ chápány jako neterminály) a pak její odvození v podstatě kopíruje výpočet Turingova stroje na w – s tím, že terminální slovo lze odvodit jen při docílení koncové konfigurace a tím slovem, které pak 'zbude', bude ono první w.

Ve světle předchozích úvah bychom nyní měli chápat, že budeme-li dále otázky teorie vyčíslitelnosti studovat na Turingových strojích, dostaneme výsledky, které jsou *robustní* – nezávislé na volbě tohoto konkrétního modelu (ke stejným výsledkům bychom např. dospěli, pokud bychom používali např. pascalské programy – technicky by ovšem vše bylo náročnější [důvod je zřejmý již z porovnání definic Turingových strojů a Pascalu]).

O jaké (robustní) výsledky jde ? Především samozřejmě půjde o prokázání nerozhodnutelnosti konkrétních problémů. Ještě předtím si ale ukážeme výsledek týkající se existence určitého, tzv. univerzálního, algoritmu.

Představte si, že vám někdo předloží zadání Turingova stroje M (v dohodnutém formátu; de facto se určitě jedná o řetězec v určité dohodnuté abecedě, byť napsaný např. ve více řádcích na papíře) a nějaké jeho vstupní slovo w; dotyčný vás vyzve, ať předvedete (simulujete) výpočet stroje M na slově w. To, co pak budete provádět, bude zajisté naprosto mechanický (algoritmický) postup, který jste schopni aplikovat na libovolný Tur. stroj a libovolné jeho vstupní slovo. Realizujete tedy určitý konkrétní algoritmus !

Všimněte si, že zastaví-li se M na w, což budeme značit $!M(w)$, vaše činnost po nějakém čase skončí – odhlížíme teď od času a prostoru, který byste k dané simulaci potřebovali – a jste schopni vydat 'výstup' totožný s (neprázdným) obsahem pásky v koncové konfiguraci stroje M dosažené při výpočtu nad w. Tento obsah pásky (konečný řetězec) značíme $\mathcal{M}(w)$ (v souladu s definicí 1.8) nebo prostě $M(w)$. Pokud se M na w nezastaví, tj. $\neg!M(w)$, vaše simulace je potenciálně nekonečná (a o nějakém 'výstupu' pak nebudeme hovořit).

Tento algoritmus, který vykonáváte při zmíněné simulaci Turingových strojů, musí ovšem podle Churchovy teze být rovněž realizován nějakým konkrétním Turingovým strojem ! Nepřekvapí nás proto následující věta. Zde $Kod(M)$ bude představovat dohodnutou formu zápisu Turingova stroje M; mělo by být zřejmé, že také zde můžeme předpokládat, že užijeme jen abecedu $\{0,1\}$, tj. $Kod(M) \in \{0,1\}^*$.

Věta 1.9 *(O* univerzálním Turingovu stroji.*) Lze sestrojit Turingův stroj U takový, že pro libovolný Turingův stroj M s abecedou $\{0,1\}$ a libovolné $w \in \{0,1\}^*$ platí:*
1/ $!M(w) \Leftrightarrow !U(Kod(M) \cdot w)$ a
2/ jestliže $!M(w)$, pak $M(w) = U(Kod(M) \cdot w)$.

Úkol. Sestrojte takový konkrétní stroj U. Vyžaduje to samozřejmě určitý čas, ale de facto se jedná o naprogramování jednoduchého algoritmu, který dobře znáte – musíte ovšem programovat v jazyce hodně 'nízké úrovně'. Váš univ. stroj může samozřejmě používat pro jeho naprogramování co nejvýhodnější formu Kod neomezující se na abecedu $\{0,1\}$. Uvědomte si ale, že byste měli být schopni rutinně upravit váš U pro případ $Kod(M) \in \{0,1\}^*$, a navíc by šlo požadovat, aby U měl také jen abecedu $\{0,1\}$. Mimo jiné to znamená, že jej lze aplikovat i na svůj vlastní kód – zamyslete se nad tím !

Poznámka. Předchozí věta mluví o univerzálním Turingovu stroji (schopným 'provádět práci' libovolného Tur. stroje). Z hlediska dříve zmíněné *robustnosti* našich výsledků existuje takový univerzální 'program' v libovolné formalizaci algoritmů. Obecně tedy můžeme hovořit o *univerzálním algoritmu*; promyslete si, proč se na počítač lze dívat jako na zařízení realizující onen univerzální algoritmus.

Je načase uvést příklad problému, který algoritmicky rozhodnutelný není.

Problém HP (Halting Problem)

Instance: Tur. stroj M – resp. jeho kód $Kod(M)$ v abecedě $\{0,1\}$ a slovo $w \in \{0,1\}^*$.

Otázka: Zastaví se M na w (platí $!M(w)$) ?

Věta 1.10 *Problém HP není rozhodnutelný.*

Důkaz: Důkaz je vedený sporem. Předpokládejme, že ex. Tur. stroj H, který se pro lib. vstup $u \in \{0,1\}^*$ tvaru $u = Kod(M) \cdot w$ pro něj. M a slovo w zastaví a rozhodne, zda $!M(w)$ či nikoliv (skončí např. buď ve spec. stavu q_{ANO} nebo v q_{NE}). U stroje H je možné předpokládat abecedu $\{0,1\}$. Sestrojme nyní stroj H' s abecedou $\{0,1\}$, který se chová následovně: vstupní slovo $v \in \{0,1\}^*$ nejprve zdvojí (vytvoří slovo vv) a na to 'spustí' (jako podprogram) stroj H. Jestliže (podprogram) H skončí ve stavu q_{ANO}, stroj H' přejde do nekonečného cyklu (a tedy se nezastaví); jestliže H skončí ve stavu q_{NE}, stroj H' se zastaví (stav q_{NE} bude také jeho koncovým stavem). Když ovšem nyní prozkoumáme, zda se H' při spuštění na svůj kód $Kod(H')$ zastaví či nezastaví, dospějeme při obou možnostech k logickému sporu (zastaví se a nezastaví se zároveň). □

Jelikož HP je zřejmě částečně rozhodnutelný (částečně ho rozhoduje např. univerzální Turingův stroj), dostáváme:

Důsledek 1.11 *Problém HP je příkladem problému (či jazyka), který je částečně rozhodnutelný, ale není rozhodnutelný.*

Je užitečné si všimnout, že při důkazu nerozhodnutelnosti problému HP jsme vlastně ukázali nerozhodnutelnost jeho podproblému, který označíme DHP *(Diagonal Halting Problem) – Instance:* Stroj M daný svým kódem $Kod(M)$; *Otázka:* Zastaví se M na svůj kód (tj. na slovo $Kod(M)$) ?

Věta 1.12 *Problém DHP je částečně rozhodnutelný, ale není rozhodnutelný.*

Máme-li dokázánu nerozhodnutelnost jednoho problému, je možno ji využít k prokázání nerozhodnutelnosti dalších problémů. Např. z nerozhodnutelnosti problému P ihned plyne nerozhodnutelnost jeho doplňkového problému (ANO, NE přehozeny); srovnej Tvrzení 1.6. Více užitečný je ovšem následující pojem:

Definice 1.13 *Problém $P1$ je* (algoritmicky) převeditelný *na problém $P2$, označme $P1 \rightsquigarrow P2$, jestliže existuje algoritmus A, který pro libovolnou instanci I problému $P1$ (chápanou jako jeho vstup) sestrojí (tzn. skončí svůj výpočet a jako výstup vydá) instanci problému $P2$, označme ji $A(I)$, přičemž platí, že odpověď na otázku problému $P1$ pro instanci I je ANO právě tehdy, když odpověď na otázku problému $P2$ pro instanci $A(I)$ je ANO.*

Poznámka. Čtenáře jistě nepřekvapí, že pojem rekurzivní převeditelnosti se definuje obdobně s tím, že pojem algoritmus se nahradí pojmem Turingův stroj. Připomeňme, že při přijetí Churchovy teze jsou pojmy rekurzivní a algoritmické převeditelnosti totožné.

Jelikož problému typu ANO/NE koresponduje dříve uvedeným přirozeným způsobem určitý jazyk (sestávající ze všech řetězců popisujících instance s odpovědí ANO), dostáváme takto rovněž pojem *(algoritmické) převeditelnosti jazyků.*

Užitečnost uvedeného pojmu pro naše účely vyslovuje následující tvrzení, jehož důkaz by měl být zřejmý.

Tvrzení 1.14 *Je-li $P1 \rightsquigarrow P2$ a problém $P1$ je nerozhodnutelný, je i problém $P2$ nerozhodnutelný.*

Příklad. Takto se např. prokáže nerozhodnutelnost problému UHP (Uniform Halting Problem; Instance: Tur. stroj M; Otázka: Zastaví se M na každý vstup (tj. platí $\forall w :!M(w))$?.) Při prokázání $HP \rightsquigarrow UHP$ stačí navrhnout algoritmus, který k zadanému M, w sestrojí stroj M', jenž nejdříve otestuje vstup a v případě, že jde o w, 'spustí' (podprogram) M a v opačném případě se ihned zastaví.

Uvedeme příklad jiného důležitého nerozhodnutelného problému:

Problém PKP (Postův korespondenční problém)

Instance: dvojice seznamů $u_1, u_2, \ldots, u_n$ a $v_1, v_2, \ldots, v_n$ (pro něj. $n \geq 1$) neprázdných řetězců (slov) v nějaké abecedě.

Otázka: má PKP pro danou instanci řešení , tj. existují indexy $i_1, i_2, \ldots, i_r$, $r > 0$, tak, že $u_{i_1} u_{i_2} \ldots u_{i_r} = v_{i_1} v_{i_2} \ldots v_{i_r}$? (Jestliže $i_1 = 1$, hovoříme o iniciálním řešení.)

Důkaz nerozhodnutelnosti problému PKP lze provést např. prokázáním převeditelností $HP \rightsquigarrow IPKP \rightsquigarrow PKP$, kde problém $IPKP$ je zadán obdobně jako PKP, jen otázka se ptá, zda existuje iniciální řešení pro danou instanci. Hlavní myšlenka převeditelnosti $HP \rightsquigarrow IPKP$ spočívá v následujícím: k danému M, w se sestrojí první členy seznamů $u_1 = \$$, $v_1 = \$q_0w\$$ (kde q_0 je poč. stav M). Další dvojice se volí tak, aby jediná možná cesta k získání iniciálního řešení spočívala v určité simulaci výpočtu M na w s tím, že řešení existuje právě když M se zastaví na w.

Poznámka. PKP se dá užít k důkazu nerozhodnutelnosti některých problémů v teorii formálních jazyků. Např. lze PKP snadno převést na následující problém:

> *Instance:* dvě bezkontextové gramatiky G_1, G_2
>
> *Otázka:* Platí $L(G_1) \cap L(G_2) \neq \emptyset$? (Tzn. 'Lze nějaké slovo vygenerovat oběma gramatikami ?')

Podobně se dá dokázat, že pro bezkontextové gramatiky je nerozhodnutelným problémem otázka '$L(G) = \Sigma^*$?', tedy i otázka '$L(G_1) = L(G_2)$?' apod.

Všimneme si nyní, že všechny Turingovy stroje (s abecedou $\{0,1\}$) lze přirozeně seřadit (očíslovat, enumerovat). Už jsme si uvědomili, že Turingovy stroje (s abecedou $\{0,1\}$) lze kódovat řetězci nul a jedniček. Když si uvědomíme, že pro libovolnou konečnou abecedu Σ (tedy i pro $\Sigma = \{0,1\}$) je množina Σ^* nekonečná spočetná (řetězce lze např. uspořádat podle délky a v rámci stejné délky lexikograficky (abecedně)), je ihned jasné, že i Turingových strojů je nejvýš spočetně – samozřejmě ovšem nekonečně spočetně. Navíc nám zmíněné uspořádání řetězců z $\{0,1\}^*$ automaticky dává přirozené uspořádání Turingových strojů (podle jejich kódů v abecedě $\{0,1\}$): je tedy možné hovořit o *enumeraci* $M_0, M_1, M_2, \ldots$ Turingových strojů (či jejich kódů). Navíc příslušné zobrazení $\mathcal{N} \longrightarrow \{w \in \{0,1\}^* \mid w = Kod(M)$ pro něj. Tur. stroj $M\}$ je bijekce, která je (obousměrně) algoritmicky vyčíslitelná ($\mathcal{N}$ označuje množinu všech celých nezáporných čísel).

Zamysleme se na chvíli nad otázkou, zda existují jazyky (v abecedě $\{0,1\}$), které nejsou částečně rozhodnutelné. Dříve uvedená Postova věta spolu s faktem, že (jazyk) HP (nebo DHP) je částečně rozhodnutelný a není rozhodnutelný, už poskytuje odpověď: doplněk jazyka HP (či DHP) není částečně rozhodnutelný.

Zmíněný fakt je možné ukázat i následovně. Jelikož Turingových strojů je spočetně mnoho, je také částečně rozhodnutelných jazyků (nejvýš, ovšem samozřejmě právě) spočetně mnoho. Díky obecné Cantorově větě (mohutnost množiny všech podmnožin množiny M je ostře větší než mohutnost M), je zřejmé, že množina všech jazyků $\{L \mid L \subseteq \{0,1\}^*\}$ je nespočetná.

Je ilustrativní neodvolávat se na obecnou větu, ale provést přímý důkaz tzv. *Cantorovou diagonalizační metodou*; tato metoda je totiž v oblasti vyčíslitelnosti a složitosti velmi užitečná.

Věta 1.15 *Pro libovolnou neprázdnou abecedu Σ existují jazyky $L \subseteq \Sigma^*$, které nejsou částečně rozhodnutelné.*

Důkaz: (Ilustrace Cantorovy diagonalizační metody).

Pro libovolnou (např. výše uvedenou) enumeraci (všech) Turingových strojů $M_0, M_1, M_2, \ldots$ a pro libovolné uspořádání (všech) slov $w_0, w_1, w_2, \ldots$ v abecedě Σ (např. zmíněné uspořádání 'délka+lexikogr.') definujme jazyk L následovně: pro každé i slovo w_i patří do jazyka L právě tehdy, když stroj M_i slovo w_i nepřijímá (nezastaví se na něj). Je zřejmé, že L není přijímán žádným ze strojů M_i. □

Nyní si uvedeme důležitou, tzv. Riceovu, větu, která zahrnuje celou třídu nerozhodnutelných problémů. Vyslovíme ji nejdříve v obšírnějším znění:

Jakákoli netriviální vlastnost Tur. strojů týkající se výhradně jejich vstupně/výstupního chování (tzn. každé dva Tur. stroje, které realizují totéž zobrazení, buď oba vlastnost mají nebo oba vlastnost nemají; netrivialita spočívá v tom, že ex. Tur. stroj, jenž vlastnost má, a ex. Tur. stroj, jenž ji nemá) je nerozhodnutelná (tj. množina všech kódů (indexů) Turingových strojů s danou vlastností) je nerozhodnutelná.

Níže vyslovíme totéž v elegantnější podobě. Připomeňme si nejprve naši enumeraci $M_0, M_1, M_2, \ldots$; o čísle i budeme hovořit jako o *indexu* Turingova stroje M_i. Dále připomeňme, že každý M_i realizuje (částečné) zobrazení $\{0,1\}^* \longrightarrow \{0,1\}^*$. Dále si ještě uvědomme, že pojem rozhodnutelnosti (jako i částečné rozhodnutelnosti apod.) máme vlastně definován i pro množiny přirozených čísel – množinu $\mathcal{N}$ totiž můžeme např. velmi přirozeně ztotožnit s množinou řetězců nad jednoprvkovou abecedou.

Věta 1.16 *(Rice) Nechť $\mathcal{A}$ je nějaká množina algoritmicky vyčíslitelných (částečných) zobrazení typu $\{0,1\}^* \longrightarrow \{0,1\}^*$. Potom množina $B = \{i \in \mathcal{N} \mid \mathcal{M}_i \in \mathcal{A}$ (tzn. M_i realizuje zobrazení patřící do $\mathcal{A}$)$\}$ je rozhodnutelná právě když $B = \emptyset$ nebo $B = \mathcal{N}$.*

Důkaz: Vezměme nějakou takovou $\mathcal{A}$, která není prázdná ani nezahrnuje všechny alg. vyčíslitelné funkce. Nechť nikde nedefinované zobrazení $- : \{0,1\}^* \longrightarrow \{0,1\}^*$ nepatří do $\mathcal{A}$ (opačný případ se řeší podobně). Nechť M_{i_0} realizuje $-$, tedy $i_0 \notin B$, a nechť M_{j_0} realizuje zobrazení z $\mathcal{A}$ (nutně takový existuje); tedy $j_0 \in B$.

Ukážeme, že problém DHP je převeditelný na B (tj. na problém příslušnosti k B), čímž prokážeme nerozhodnutelnost B. Algoritmus $PREV$ převodu $DHP \rightsquigarrow B$ pracuje následovně:

K danému stroji (kódu stroje) M (tj. k instanci problému DHP) algoritmus $PREV$ nejdříve sestaví stroj M', který je 'naprogramován' tak, že jeho činnost je následovná:

M' nejprve vpravo vedle svého vstupu (na kterém v této chvíli nezáleží) zapíše slovo $Kod(M)$ a na něj spustí (podprogram) M; pokud tento (pod)výpočet skončí, smaže M' případný zbytek po tomto výpočtu, najede na původně daný vstup a spustí na něj M_{j_0}.

Po sestavení tohoto M' spočte $PREV$ jeho index a ten vydá.

Je zřejmé: když M se zastaví na $Kod(M)$ (tj. odpověď na onu instanci DHP je ANO), realizuje M' totéž zobrazení jako M_{j_0} a jeho index tedy patří do B; když M se nezastaví na $Kod(M)$ (odpověď v DHP je NE), realizuje M' zobrazení $-$, tedy totéž jako M_{i_0} a jeho index tedy do B nepatří. □

2 Složitost

Poté, co jsme uspokojivým způsobem zodpověděli otázku 'Co všechno je algoritmicky řešitelné (vyčíslitelné) ?', zauvažujeme o základní otázce teorie složitosti:

Jak je řešení (vyčíslování) algoritmicky řešitelných problémů složité ?

Zkušenost nám říká, že tentýž úkol (problém) lze řešit různými metodami (algoritmy), které mají různou složitost. Navíc je intuitivně také zřejmé, že každý problém má určitou 'vnitřní složitost' (odpovídající, zhruba řečeno, složitosti toho nejoptimálnějšího algoritmu řešícího daný problém), a rovněž problémy lze tedy určitým způsobem porovnávat podle jejich (vnitřní) složitosti. Rovněž už asi máme zkušenost s tím, že některé problémy jsou sice algoritmicky řešitelné, ale v praxi nezvládnutelné.

Z těchto intuitivních úvah lze již odvodit základní úkoly teorie složitosti: precizovat pojmy *složitost algoritmu*, *složitost problému* a pokud možno vymezit třídu *(prakticky) zvládnutelných problémů*. To vše lze udělat různými způsoby, jde ovšem o to, aby zvolený způsob dával rozumné výsledky pro praxi a aby přitom zvolené pojmy byly dostatečně jednoduché a 'průhledné'.

Začneme u pojmu složitosti algoritmu; roli algoritmů budou pro nás, ve světle předcházející kapitoly, hrát Turingovy stroje (čtenář si samozřejmě místo Turingových strojů může představovat programy v jeho oblíbeném programovacím jazyce a vše níže uvedené si patřičně 'překládat'); v této souvislosti je ovšem důležitá poznámka, která je uvedena na závěr textu.

Co si představovat pod pojmem složitost Tur. stroje (programu) není jednoznačné. V jistém kontextu to může být např. počet instrukcí, hloubka 'vnořených cyklů' apod. Nám zde ovšem hlavně půjde o časovou (případně paměťovou) náročnost výpočtů daného stroje. Poznamenejme hned, že v případě nekonečných výpočtů složitost nedefinujeme – to v dalším textu nebudeme uvádět (implicitně budeme předpokládat, že relevantní výpočty jsou konečné, tj. že dojde k zastavení Tur. stroje):

Definice 2.1 Časová složitost výpočtu *Turingova stroje M nad slovem w se definuje jako počet elementárních kroků (instrukcí), které M nad w vykoná, než se zastaví.*

Časová složitost stroje M by se teď dala chápat jako funkce typu $\Sigma^* \longrightarrow \mathcal{N}$ (kde Σ je abeceda stroje a $\mathcal{N}$ je množina přirozených čísel). Tento pojem je ale příliš detailní a navíc se explicitně odkazuje k abecedě daného stroje. Lépe se osvědčuje následující definice:

Definice 2.2 Časovou složitostí *Turingova stroje M rozumíme funkci $T_M : \mathcal{N} \longrightarrow \mathcal{N}$, kde $T_M(n)$ znamená čas. složitost výpočtu M nad vstupem délky n v nejhorším případě (tj. $T_M(n) = max\,\{\, k \mid k$ je čas. slož. výpočtu M nad w, kde délka$(w) = n\,\}$).*

Při analýze časové složitosti konkrétního Turingova stroje (či programu, chcete-li) M nám v praxi většinou nejde o přesný popis funkce T_M, ale jen o její odhad. Navíc se většinou zanedbávají konstatní faktory, což vede k následujícímu značení:

- Značením $f \in O(g)$, nebo $f(n) \in O(g(n))$, rozumíme, že ex. k a n_0 tž. $\forall n \geq n_0$: $f(n) \leq k \cdot g(n)$.
- $f \in o(g)$, nebo $f(n) \in o(g(n))$, znamená, že pro každé (reálné) $k > 0$ ex. n_0 tž. $\forall n \geq n_0$: $f(n) < k \cdot g(n)$.
- $f \in \Theta(g)$, nebo $f(n) \in \Theta(g(n))$, znamená, že $f \in O(g)$ a $g \in O(f)$.
- $f \in \Omega_\infty(g)$, nebo $f(n) \in \Omega_\infty(g(n))$, znamená, že ex. $k > 0$ a nekonečně mnoho n tž. $f(n) \geq k \cdot g(n)$.

Nejběžnější funkce vyskytující se v odhadech jsou funkce $log\, n$, n, $n \cdot log\, n$, n^2, n^3, 2^n apod. (log se většinou chápe se základem 2; uvědomte si, že díky zanedbávání konst. faktoru na tom nezáleží).

Teď už je např. jasné, co to znamená, že časová složitost nějakého Tur. stroje je v $O(n^2)$, či v $O(n \cdot log\, n)$ apod. Všimněme si, že O hraje roli (neostrého) horního odhadu, o roli ostrého horního odhadu, Ω_∞ představuje určitý dolní odhad a Θ je vlastně horní i dolní odhad zároveň (složitost je v tom případě 'přesně' určena – samořejmě až na zanedbávané faktory).

Na rozdíl od složitosti algoritmu (Tur. stroje), je pojem složitosti problému hůře definovatelný (zamyslete se nad tím !). Užitečné řešení spočívá v následujícím utřídění problémů:

Definice 2.3 Třídou (časové) složitosti $\mathcal{T}(f)$ *pro funkci* $f : \mathcal{N} \longrightarrow \mathcal{N}$ *rozumíme třídu těch problémů, které jsou rozhodovány (vyčíslovány) Tur. stroji s čas. složitostí v* $O(f)$.

Všimněme si, že určitě platí např.

$$\mathcal{T}(n) \subseteq \mathcal{T}(n \cdot log\, n) \subseteq \mathcal{T}(n^2) \subseteq \mathcal{T}(n^3) \subseteq \mathcal{T}(2^n)$$

(Z hlubších výsledků teorie složitosti, které zde nebudeme zmiňovat, plyne, že každá z uvedených inkluzí je vlastní.)

Část teorie, která se někdy nazývá *konkrétní složitost*, studuje složitost konkrétních problémů (a algoritmů), resp. příslušné horní a dolní odhady. My se zde dotkneme spíše tzv. *strukturální složitosti*, jež má za úkol zkoumat strukturu tříd složitosti problémů. Podotkněme ovšem, že obě zmíněné partie se samozřejmě prolínají a ovlivňují. Jedním z nejdůležitějších cílů teorie (strukturální) složitosti je co možná nejlépe charakterizovat *třídu zvládnutelných problémů* (tj. třídu problémů, pro které existují 'dostatečně rychlé' algoritmy).

Jako nejrozumnější aproximace třídy zvládnutelných problémů se (zatím) ukázala třída označovaná $PTIME$, nebo jen P (ze slova 'Polynomial'), definovaná následovně

$$PTIME = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

To znamená, že pojem 'rychlý algoritmus' je ztotožňován s pojem 'polynomiální algoritmus' (tj. algoritmus s polynomiální časovou složitostí). To není samozřejmě ideální (např. algoritmus s časovou složitostí zhruba $n^{1000000}$ těžko lze považovat za rychlý), zatím je však tato charakterizace shledávána jako vyhovující (poznamenejme, že se ukazuje, že existuje-li pro problém 'z praxe' polynomiální algoritmus, pak exponent v polynomu je velmi malý – řekněme menší než 5).

Každý čtenář jistě zná spoustu zvládnutelných problémů (prvků $PTIME$), později ukážeme (rozhodnutelné) problémy, o kterých je dokázáno, že zvládnutelné nejsou (jsou mimo $PTIME$).

Podobnou roli jako algoritmická převeditelnost pro (ne)rozhodnutelnost problémů přináší tzv. polynomiální převeditelnost pro (ne)zvládnutelnost problémů (definice je v podstatě stejná, jen u algoritmu převodu je vyžadována polynomiální časová složitost – tzn. složitost v $O(n^k)$ pro nějaké $k \in \mathcal{N}$):

Definice 2.4 *Problém $P1$ je* polynomiálně převeditelný *na problém $P2$, označme $P1 \lhd P2$, jestliže existuje Tur. stroj M* s polynomiální časovou složitostí, *který pro libovolnou instanci I problému $P1$ sestrojí instanci problému $P2$, označme ji $M(I)$, přičemž platí, že odpověď na otázku problému $P1$ pro instanci I je ANO právě tehdy, když odpověď na otázku problému $P2$ pro instanci $M(I)$ je ANO.*

Je zřejmé, že jestliže $P1 \lhd P2$ a $P2$ je v $PTIME$, pak i $P1$ je v $PTIME$; naopak, když $P1$ není v $PTIME$, ani $P2$ není v $PTIME$.

Jednou ze silných motivací pro rozvoj strukturální složitosti je fakt, že u mnoha konkrétních praktických problémů nejsme (zatím) schopni prokázat, zda jsou či nejsou v $PTIME$. O těchto problémech většinou víme, že jsou v třídě $EXPTIME$, kde

$$EXPTIME = \bigcup_{k=0}^{\infty} \mathcal{T}(2^{n^k}).$$

Jsou známy konkrétní problémy, které jsou v $EXPTIME$ ale ne v $PTIME$, o spoustě z nich ale nepříslušnost k $PTIME$ prokázána není. Když si např. celkem přímočaře zavedeme třídy $PSPACE$, $EXPSPACE$ založené na prostorové (paměťové) složitosti, lze snadno ukázat, že

$$PTIME \subseteq PSPACE \subseteq EXPTIME \subseteq EXPSPACE;$$

neví se ovšem, které inkluze jsou vlastní. Např. je jasné, že jedna z inkluzí $PTIME \subseteq PSPACE$, $PSPACE \subseteq EXPTIME$ musí být vlastní. Zdá se sice, že vlastní jsou obě, nicméně stále není vyloučena možnost $PTIME = PSPACE$! Přitom o spoustě praktických problémů se ví, že jsou v $PSPACE$, ale neumí se prokázat nepříslušnost k $PTIME$. Mnoho těchto problémů je speciálního charakteru: jsou rozhodnutelné v polynomiálním čase *nedeterministickým* Turingovým strojem.

Definice 2.5 *Daný problém (typu ANO/NE) je rozhodován nedeterministickým Turingovým strojem M, jestliže všechny výpočty M jsou konečné a vydávají ANO nebo*

NE, přičemž pro lib. instanci I daného problému existuje (alespoň jeden) výpočet M nad I vydávající ANO právě když (správná) odpověď na I je ANO.
Složitost takového nedet. Tur. stroje M pro slovo w je pak definována jako délka nejkratšího možného výpočtu nad w vydávajícího ANO – pokud takový existuje; v opačném případě lze vzít délku nejkratšího výpočtu (vydávajícího NE).

Další definice lze již standardně doplnit, takže by mělo být jasné, co se myslí třídou (problémů typu ANO/NE) označovanou $NPTIME$, nebo někdy jen NP (N ze slova 'nondeterministic').

Dá se celkem snadno ukázat, že

$$PTIME \subseteq NPTIME \subseteq PSPACE.$$

Takto jsme se dostali k velmi známé dosud otevřené otázce, zda $PTIME = NPTIME$ (dané otázce se často říká P-NP problém).

Poznámka. Podobně lze dodefinovat třídu $NPSPACE$ apod. Savitch ukázal elegantní důkaz rovnosti $PSPACE = NPSPACE$.

O spoustě praktických problémů (jedním z nich je tzv. 'problém obchodního cestujícího', dále se ještě zmíníme o problému splnitelnosti booleovských formulí) se snadno ukáže, že jsou v NP, ale nikdo pro ně nezná (deterministický) polynomiální algoritmus. Tyto problémy jsou jistým způsobem nejtěžší ve třídě NP (jsou tzv. NP – *úplné*):

Definice 2.6 *Mějme třídu složitosti $\mathcal{C}$. O problému P řekneme, že je $\mathcal{C}$-*těžký*, jestliže pro lib. $P' \in \mathcal{C}$ platí $P' \lhd P$. Je-li navíc $P \in \mathcal{C}$, říkáme, že P je $\mathcal{C}$-*úplný*.*

Podle definice lze (vcelku přímočaře, i když poměrně pracně) dokázat tzv. Cookovu větu:

Věta 2.7 *Problém splnitelnosti booleovských formulí (Instance: booleovská formule [obvykle předp. v konj. norm. formě]; Otázka: existuje ohodnocení proměnných, při němž je formule pravdivá ?) je NP-úplný.*

Když už máme k dispozici jeden NP-úplný problém, dá se NP-úplnost dalších problémů prokázat využitím následujícího tvrzení.

Tvrzení 2.8 *Jestliže $P_1 \lhd P_2$ a P_1 je NP-těžký, pak P_2 je rovněž NP-těžký. Speciálně, když P_2 je v NP (P_1 je pak rovněž v NP), pak P_2 je NP-úplný.*

Jestliže prokážeme NP-úplnost nějakého problému, znamená to pro nás, že je prakticky nezvládnutelný (tzn. napíšeme-li program, který daný problém přesně rozhoduje, budeme ho moci skutečně použít jen na velmi malá vstupní data) – teoreticky ovšem pořád ještě možnost rychlého algoritmu existuje. Podobně to platí i pro $PSPACE$-úplné problémy (jako je třeba problém, zda dané dva regulární výrazy jsou ekvivalentní [tj. představují tentýž jazyk]). Je-li ovšem nějaký problém např. $EXPSPACE$-úplný, je už určitě

(dokazatelně) nezvládnutelný; příkladem je problém ekvivalence reg.výrazů s mocněním [je možno psát $(RV)^2$ místo $RV \cdot RV$].

Existují samozřejmě i dokazatelně těžší (superexponenciální) problémy; mezi ně patří např. problém rozhodování Presburgerovy aritmetiky (rozhodování pravdivosti formulí teorie sčítání).

Poznámka. Pozorný čtenář se již možná zamyslel nad *robustností* uvedených pojmů a výsledků – nejsou náhodou závislé na námi zvoleném modelu algoritmů, tj. na Turingových strojích ? Úvahy tohoto typu jsou určitě oprávněné: ačkoli se nám např. nepodaří navrhnout stroj pro rozpoznávání slov typu ww^R se složitostí menší než řádově n^2, v případě modelu Tur. stroje s dvěma hlavami je složitost daného problému zřejmě lineární. Oba modely se ovšem vzájemně simulují s 'polynomiální ztrátou', takže definice tříd P, NP atd. jsou pro ně stejné. Zakončeme konstatováním, že všechny 'rozumné' (realistické) modely algoritmů jsou polynomiálně ekvivalentní Turingovým strojům (vzájemně se simulují s 'polynomiální ztrátou'); pro ně jsou tedy výše uvedené úvahy (týkající se např. NP-úplnosti apod.) totožné.