Studijní opora k předmětům teoretické informatiky (speciálně v oblasti teorie vyčíslitelnosti a složitosti)

Petr Jančar

22. ledna 2004

Obsah

Přednáška 1

Úvod, cíle kursu, literatura

Kurs je určitým úvodem do partií teorie algoritmů, především partií, které se nazývají vyčíslitelnost (computability) a složitost (complexity). Základním úkolem teorie vyčíslitelnosti je charakterizovat ty problémy, jež jsou algoritmicky řešitelné (vyčíslitelné, vypočitatelné). Teorie složitosti se pak zabývá především otázkou, jak jsou příslušné výpočty, potažmo algoritmy, složité (z hlediska nároku na čas, paměť apod.).

Konkrétnější představu o náplni kursu si lze udělat z obsahu. Např. si tak lze všimnout, že kurs se dotkne mj. též obecných metod návrhu algoritmů, což je důležité téma, které do rámce vyčíslitelnosti a složitosti spadá jen nepřímo. (V obsahu je pro lepší orientaci rovněž uvedeno předpokládané rozdělení probírané látky do jednotlivých přednášek kursu.)

Cíle kursu

Absolvent má získat hlubší (teoretický) základ pro pojmy složitosti algoritmů a problémů, i pro jejich analýzu (odhady) u konkrétních případů – mj. u algoritmů navrhovaných obecnými metodami (prohledávání, "rozděl a panuj", 'greedy' přístup, dynamické programování). Dále má zvládnout klasifikaci typických (praktických) problémů vzhledem k základním třídám složitosti (PTIME, NPTIME, PSPACE, ...), včetně prokázání případné nerozhodnutelnosti problému. Rovněž má získat představu, podloženou pochopením konkrétních příkladů, o problematice aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů.

Poznámka. U každé kapitoly textu jsou vedle stručného obsahu uvedeny studijní cíle. Ty jsou uvedeny jen orientačně, nelze je chápat tak, že by přesně vyjadřovaly, co má (a co nemusí) studující zvládnout.

Literatura

Základní studijní pomůckou ke kursu je předkládaný pracovní text a je jej možné využít i jako základ samostudia. Text je ovšem na mnoha místech velmi stručný; jeho studium by mělo být významně ulehčeno *aktivní* účastí na přednáškách a cvičeních (kde jsou studované pojmy a metody názorněji ilustrovány na příkladech, myšlenky textu jsou dále rozvedeny a vysvětleny, apod.).

Poznámka. Program cvičení bude průběžně zvřejňován na Webu

http://www.cs.vsb.cz/jancar/VYCSLOZ/vycsloz.htm;

příklady a referáty na nich probrané jsou integrální součástí kursu !

Velmi vhodné by samozřejmě bylo, aby si posluchači znalost a pochopení probíraných témat upevnili studiem další odborné literaury. Uveďme alespoň některé knihy v češtině či slovenštině pokrývající části přednášené látky. Partie z teorie složitosti (včetně konkrétních algoritmů a problémů) najdeme např. v knihách [Kuč83], [Mor84] (zejména první z nich vřele doporučuji !). Další partie (jako Turingovy stroje, problém zastavení, zavedení výpočtové složitosti aj.) jsou pokryty např. v knihách [Chy84], [HU78]. Dále např. 1.kapitola knihy [Man81] je stručně věnována teorii vyčíslitelnosti.

Podstatně bohatší je literatura v angličtině. Z ní je možné doporučit např. [Sip97], [Har93], [CLR90], [HU79]. Samozřejmě spoustu relevantního materiálu (různé kvality) lze také získat "surfováním" na Webu.

Poznámka. Jako vždy, člověk se nejvíce naučí vlastními *aktivními* pokusy o řešení příslušných problémů – souběžně s "pasivním" studiem. Teprve (a jen) tehdy může pochopit, o co vlastně jde, a ocenit nastudované řešení.

1 Složitost algoritmu. Model RAM.

Stručný obsah: Pojem složitosti (časové či paměťové náročnosti) algoritmu – vztažen k referenčnímu abstraktnímu modelu počítače a definován jako funkce velikosti vstupu. Model (referenčního) stroje RAM (tj. počítače s okamžitým přístupem k libovolné buňce paměti); jednotková vs. logaritmická míra v definici složitosti. Jako příklad algoritmy *bubblesort* a *heapsort*; pro ilustraci návrh a analýza časové složitosti RAMu pro *bubblesort*.

Studijní cíle: (Pochopit a) umět vysvětlit pojem složitosti algoritmu včetně role referenčního modelu počítače. Schopnost porozumění a analýzy složitosti (jednoduchých) programů pro RAM. Umět vysvětlit roli jednotkové a logaritmické míry v definici složitosti. Pochopení programování na úrovni RAMu, které dostačuje pro analýzu algoritmů zapsaných jazyky 'vyšší úrovně'.

Čtenář už jistě má určitou představu o tom, že např. pro jeden a tentýž problém existují různé algoritmy, které ho řeší; takové algoritmy (a koneckonců nejen ty řešící stejný problém) je možné vzájemně srovnávat z různých hledisek. Pro konkrétnost si připomeňme problém třídění (sorting; v češtině by zde byl vhodnější termín 'seřazování'):

> *Název (označení) problému*: *Třídění čísel*
>
> *(Očekávaný) vstup*: konečná posloupnost přirozených čísel
> *(Požadovaný) výstup*: posloupnost týchž čísel uspořádaná podle velikosti ve vzestupném pořadí.

V učebnicích se často mezi prvními algoritmy řešícími daný problém uvádí tzv. *bubblesort*, jehož základní myšlenka se dá vyjádřit takto:

> Projdi posloupnost zleva doprava, přičemž prohazuješ sousední dvojice čísel, pokud v nich větší číslo předchází menšímu.
>
> Tento postup procházení posloupnosti opakuj, dokud nedostaneš kompletně uspořádanou posloupnost.

Poznámka. Poznamenejme, že *bubblesort* se v učebnicích vyskytuje spíše jako odstrašující příklad (jelikož lze snadno navrhnout podstatně lepší algoritmy, jak o tom také budeme hovořit dále). My zde tento algoritmus také uvádíme jen pro jeho jednoduchost a ilustraci dále zkoumaných pojmů, nikoliv snad pro jeho 'hodnotu'.

Zpřesněné vyjádření algoritmu programátorským pseudokódem by mohlo vypadat takto:

```
{členy vstupní posloupnosti jsou označeny A[1], A[2], ... A[n]}
while Nesetříděno do
  for i := 1 to n − 1 do
    if A[i] > A[i + 1] then prohoď A[i] a A[i + 1]
```

(V této chvíli se náš návrh nezabývá tím, jak se přiřazuje do booleovské proměnné 'Nesetříděno'.)

Po přesvědčení se, že algoritmus je korektní – tj. vždy skončí a výsledná posloupnost je uspořádaná (jak byste to dokázali ?), je možné využít většího porozumění předepsaného procesu třídění a upravit (a zpřesnit) algoritmus následovně:

Bubblesort – progr. verze

```
{členy vstupní posloupnosti nejprve načteme do pole A}
{předp., že členy jsou nenulové a hodnota 0 označuje konec vstupu}
n := 0;
repeat
  n := n + 1; read(A[n])
until A[n] = 0;
n := n − 1;
{v n je uložen počet členů vstupní posloupnosti}
for j := 1 to n − 1 do
  for i := 1 to n − j do
    if A[i] > A[i + 1] then (pom := A[i]; A[i] := A[i + 1]; A[i + 1] := pom);
{výsledná seřazená posloupnost se vypíše}
for i := 1 to n do
  write(A[i])
```

Že tento algoritmus (to je výpočetní proces jím předepsaný) pro každou (konečnou) vstupní posloupnost skončí, je zde zřejmé (proč ?); přesvědčte se, proč je výsledná posloupnost určitě uspořádaná.

Jak jsme už zmínili, *bubblesort* zdaleka není nejlepším algoritmem pro daný problém třídění. Připomeňme si teď metodu (čili algoritmus) *heapsort*. K tomu je potřebné si připomenout datovou strukturu *halda* (heap), tj. (speciální) binární strom: každý vrchol v je ohodnocen číslem $n(v)$ (prvkem tříděné posloupnosti), přičemž je-li v' následníkem v, pak $n(v) \leq n(v')$. Zařazení dalšího prvku do haldy i výběr nejmenšího prvku z haldy se dají snadno realizovat x kroky, kde x je hloubkou haldy (stromu); při počtu vrcholů n je tedy přibližně $x = \log n$.

Poznámka. V informatice při neuvedení základu $\log n$ většinou myslíme dvojkový logaritmus $\log_2 n$. Později vysvětlíme, proč je základ logaritmu pro účely analýzy algoritmů v zásadě nepodstatný.

Důležitou myšlenkou algoritmu *heapsort* je rovněž efektivní způsob reprezentace haldy jednorozměrným polem.

Vše se dá vyčíst z dále uvedeného pseudokódu; je ovšem velmi žádoucí, ať si čtenář běh algoritmu ilustruje (připomene) na rozumně zvoleném malém příkladu.

Heapsort – progr. verze

```
{proměnná H představuje haldu (var H: array[1.. ] of integer)}
{kon udává aktuální koncový index haldy}
kon:=0; {halda je prázdná}
read(clen);
while clen ≠ 0 do
  Zarad-do-haldy(clen); read(clen);
while kon > 0 {halda není prázdná} do
  Vydej-min-z-haldy(clen); write(clen)

procedure Zarad-do-haldy(k);
kon := kon + 1; H[kon] := k;
p := kon;
while (p > 1 and H[p div 2] > H[p]) do
  prohod H[p div 2] a H[p]; p := p div 2;

procedure Vydej-min-z-haldy(var min);
min := H[1];
if kon > 1 then H[1] := H[kon];
kon := kon − 1;
p := 1;
while (2 ∗ p + 1 ≤ kon and (H[p] > H[2 ∗ p] or H[p] > H[2 ∗ p + 1])) do
  if H[2 ∗ p] ≤ H[2 ∗ p + 1]
  then (prohod H[p] a H[2 ∗ p]; p := 2 ∗ p)
  else (prohod H[p] a H[2 ∗ p + 1]; p := 2 ∗ p + 1);
if (2 ∗ p = kon and H[p] > H[2 ∗ p])
then prohod H[p] a H[2 ∗ p]
```

Oba algoritmy (*bubblesort* a *heapsort*) řeší náš problém třídění, přičemž *heapsort* je očividně složitější z hlediska návrhu, zápisu i porozumění (ověření správnosti). V čem je tedy *heapsort* lepší ?

Zkušený čtenář asi odpoví, že *heapsort* má menší časovou složitost (náročnost) než *bubblesort*. Označujeme takto fakt, že (hodně neformálně řečeno) *heapsort* 'běhá rychleji' než *bubblesort*. Určitým způsobem se o tom můžeme přesvědčit, naprogramujeme-li obě metody v námi oblíbeném programovacím jazyku a srovnáme běh obou programů na počítači na sadě instancí (tj. povolených vstupů) problému třídění – pro každou instanci měříme čas, který na její zpracování jednotlivé programy spotřebují.

Doufejme, že čtenář není natolik 'prakticky orientovaný', že mu výše zmíněný test stačí, ale že by rád více porozuměl, proč tomu tak je, a své poznání opřel o solidnější základ. (Neměl by stačit argument 'Protože jsem *bubblesort* a *heapsort* naprogramoval v Céčku a na mnou zvolených deseti příkladech běžel *heapsort* na PC-čku vždycky rychleji, je *heapsort* lepší').

Chtělo by to definovat pro každý algoritmus nějakou kvantitativní charakteristiku, nazvěme ji časová složitost (či jen složitost, když se 'časová' rozumí samo sebou), podle které pak bude možné různé algoritmy srovnávat. Složitost ovšem musí zachycovat 'dobu běhu' globálně – tj. pro všechny přípustné vstupy, nejen pro vybranou sadu testovacích případů.

Nabízí se zmíněnou časovou složitost algoritmu prostě definovat jako funkci (zobrazení), která každému (přípustnému) vstupu přiřazuje 'dobu běhu' algoritmu na onen vstup. To má ovšem několik "vad na krásе" (např. pak není jasné, jak srovnávat rychlost algoritmů pracujících s různými vstupy). Jako vhodnější (jednodušší a přitom postačující) se ukazuje definovat

složitost jako *funkci velikosti vstupu*.

Poznámka. Složitost (jakožto funkce) má většinou nekonečný definiční obor (např. i v našem problému třídění délku vstupní posloupnosti nijak neomezujeme); nelze ji tedy zadat výčtem hodnot, ale je nutno hledat nějaký konečný popis (např. algebraické vyjaádření jako $3n^2 - 4n + 3$ apod.).

Vstupů se stejnou velikostí n ovšem může být hodně a výpočty pro tyto jednotlivé vstupy mohou trvat různou 'dobu'. Co je pak hodnotou složitosti (tj. zmíněné funkce) pro n ? Pro praktické účely často stačí přístup

podle nejhoršího možného případu (worst-case),

kdy dané velikosti n přiřazuje ona funkce maximum z 'dob běhu' algoritmu na všech vstupech velikosti n.

Musí se samozřejmě vyjasnit několik věcí – např. co je to *velikost vstupu*. Později se k tomu ještě vrátíme, teď poznamenejme, že u našeho problému třídění je většinou postačující velikost vstupu definovat jako počet členů zadané posloupnosti (později dodáme: pokud je velikost čísel=členů omezená).

Poznámka. Obecně použitelné řešení je chápat velikost daného vstupu jako počet bitů, které vstup zabírá (při "přirozeném" zakódování). Často lze však dostatečné výsledky analýzy složitosti dosáhnout bez nutnosti uvažovat tuto "nízkou" úroveň (která může přidávat zbytečné technické komplikace).

Jistě jste si povšimli jiného velmi slabého místa v uvedených definicích: užívání pojmu 'doba běhu'. Vždyť např. při různých implementacích (v různých programovacích jazycích, na různých počítačích apod.) budou 'doby běhu' jednoho a téhož algoritmu pro jeden a tentýž vstup různé ! Jako nejvhodnější exaktní definování pojmu 'doba běhu' se ukazuje volba nějakého (abstraktního) modelu počítače, ke kterému se pak budeme odkazovat jako k jakémusi *referenčnímu modelu*; dobu běhu, tj. 'dobu trvání výpočtu' (neboli délku výpočtu), pak budeme měřit počtem provedených (elementárních) instrukcí.

Modelů sloužících těmto účelům byla navržena celá řada (později se k tomu ještě dostaneme). My se teď seznámíme se strojem RAM (Random Access Machine), který je česky někdy zván *'počítač s libovolným přístupem'*; název není zcela výstižný, znamená prostě to, že přístup k libovolné buňce paměti je okamžitý (či asi vhodněji: doba přístupu k libovolné buňce paměti je konstantní).

Abstraktní stroj RAM (Random Access Machine)

Stroj RAM se skládá z programové jednotky, (operační) paměti, vstupní jednotky a výstupní jednotky. (Pro ilustraci je vhodný obrázek, čtenář je vyzýván, ať si jej nakreslí.)

V programové jednotce je zapsán program, tvořený konečnou posloupností instrukcí (příkazů), které budou popsány dále. Dále je v ní programový registr ukazující, která instrukce má být v daném okamžiku prováděna (programový registr prostě obsahuje pořadové číslo příslušné instrukce).

Paměť je tvořena potenciálně nekonečným polem paměťových buněk očíslovaných (adresovaných) přirozenými čísly $0, 1, 2, \ldots$; každá buňka může obsahovat *libovolně velké celé číslo*. Buňky s adresou 0 a 1 mají zvláštní postavení a nazývají se **pracovní registr** (buňka 0) a **indexový registr** (buňka 1).

Vstupní jednotka se skládá z (potenciálně nekonečné) pásky rozdělené na políčka, z nichž každé může obsahovat *libovolně velké celé číslo*, a z hlavy, která v každém okamžiku snímá jedno z políček pásky. Základní krok v činnosti hlavy spočívá v přečtení obsahu snímaného políčka a posunutí doprava o jedno políčko.

Podobným způsobem je konstruována výstupní jednotka, jejíž hlava je však schopna pouze zapisovat do políček výstupní pásky a posouvat se zleva doprava o jedno políčko.

V počáteční konfiguraci (tj. na začátku výpočtu) je na určitém počátečním úseku vstupní pásky uložen vstup (prvních n políček, pro určité n, obsahuje (vstupní) čísla $c_1, c_2, \ldots, c_n$; vstupní hlava snímá první buňku s číslem c_1). Zbylá políčka vstupní pásky, všechna políčka výstupní pásky a všechny paměťové buňky obsahují číslo 0; programový registr ukazuje na první instrukci programu (tj. obsahuje číslo 1).

Konfigurace (tj. stav výpočtu) se mění krok za krokem prováděním předepsaných instrukcí. (Je možné si představit, že RAM má ještě jakési výkonné jednotky, jako např. aritmetickou jednotku, umožňující provádění příslušných operací).

Nyní uvedeme instrukce stroje RAM, z nichž lze sestavovat program (pro názornost se čtenář může podívat na konkrétní RAM-program – v prvním sloupci na 'obrázku' o několik stran dále). Tvary 'operandů' instrukcí a jejich příslušné hodnoty jsou patrny z následující tabulky (i je zápis přirozeného čísla). Za touto tabulkou pak již následuje přehled instrukcí, logicky rozdělených do několika skupin. (Označení 'návěští' zde představuje přirozené číslo, udávající pořadové číslo instrukce, která bude prováděna jako následující, dojde-li ke skoku.)

Tvary operandů

tvar	hodnota operandu
$=i$	přímo číslo udané zápisem i
i	číslo obsažené v buňce s adresou i
$*i$	číslo v buňce s adresou $i+j$, kde j je aktuální obsah index. registru

Instrukce vstupu a výstupu (jsou bez operandu):

zápis	význam
READ	do prac. registru se uloží číslo, které je v políčku snímaném vstupní hlavou, a vstupní hlava se posune o jedno políčko doprava
WRITE	výstupní hlava zapíše do snímaného políčka výstupní pásky obsah prac. registru a posune se o jedno políčko doprava

Instrukce přesunu v paměti:

zápis	význam
LOAD operand	hodnotou operandu se přepíše obsah prac. registru
STORE operand	hodnota operandu se přepíše obsahem prac. registru (zde se nepřipouští operand tvaru $=i$)

Instrukce aritmetických operací:

zápis	význam
ADD operand	číslo v prac. registru se zvýší o hodnotu operandu (tedy přičte se k němu hodnota operandu)
SUB operand	od čísla v prac. registru se odečte hodnota operandu
MUL operand	číslo v prac. registru se vynásobí hodnotou operandu
DIV operand	číslo v prac. registru se 'celočíselně' vydělí hodnotou operandu (do prac. registru se uloží výsledek příslušného celočíselného dělení)

Instrukce skoku:

zápis	význam
JUMP návěští	výpočet bude pokračovat instrukcí určenou návěštím
JZERO návěští	je-li obsahem prac. registru číslo 0, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí
JGTZ návěští	je-li číslo v prac. registru kladné, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí

Instrukce zastavení

zápis	význam
HALT	výpočet je ukončen ('regulérně' zastaven)

Jak lze očekávat, provedení instrukce zpravidla také znamená zvýšení programového čítače o jedničku (výpočet pokračuje prováděním bezprostředně následující instrukce); výjimkou jsou případy, kdy dojde ke skoku (a také případ instrukce HALT).

Předpokládáme, že kdykoli by mělo při běhu dojít k nedefinované akci (dělení nulou, progr. čítač ukazuje 'mimo program', adresa při použití operandu $*i$ vyjde záporná), výpočet se ('neregulérně') zastaví.

Značení. $\mathcal{N}$ v celém textu označuje množinu všech přirozených čísel $\{0, 1, 2, \ldots\}$.

Nyní můžeme pro RAMy (RAM-programy, chcete-li) exaktně definovat časovou a paměťovou složitost (jakožto funkce velikosti vstupu). Přitom se omezujeme jen na RAMy, které se pro každý vstup zastaví (provedou HALT, případně skončí 'neregulérně'); nekonečnou časovou ani paměťovou složitost neuvažujeme.

Definice 1.1 Velikostí vstupu *stroje RAM rozumíme počet buněk (vstupní pásky), které daný vstup zabírá.*
Délka výpočtu *RAM-stroje M pro konkrétní vstup se definuje jako počet provedení instrukcí, které M pro daný vstup vykoná, než se zastaví.*
Časovou složitostí *RAM-stroje M rozumíme funkci $T_M : \mathcal{N} \longrightarrow \mathcal{N}$, kde $T_M(n)$ znamená délku výpočtu M nad vstupem velikosti n v nejhorším případě; tedy $T_M(n) = max\{\, k \mid k$ je délka výpočtu M nad (nějakým) vstupem velikosti $n\,\}$.*

Definice 1.2 Velikostí paměti *RAM-stroje M (potřebné při výpočtu) pro konkrétní vstup rozumíme číslo $p+1$, kde p je maximum z adres buněk, jež jsou během výpočtu (nad daným vstupem) navštíveny.*

Paměťovou složitostí *(nebo též* prostorovou složitostí*) RAM-stroje M rozumíme funkci $S_M : \mathcal{N} \longrightarrow \mathcal{N}$, kde $S_M(n)$ znamená velikost potřebné paměti při výpočtu M nad vstupem velikosti n v nejhorším případě; tedy $S_M(n) = max\{\, k \mid k$ je velikost paměti potřebné při výpočtu M nad (nějakým) vstupem velikosti $n\,\}$.*

Pozorný čtenář si už možná všiml podezřelého místa v uvedených definicích: nijak neomezujeme velikost čísla, které je možno uložit do jednotlivé buňky paměti ! Přitom velikost paměti zabrané danou buňkou (a čas elementární operace pracující s danou buňkou) chápeme jako jednotku, jinými slovy uvažujeme tzv.

jednotkovou (či uniformní) míru.

Takto však lze 'šikovně šetřit paměť' kódováním celé série čísel (např. matice čísel) číslem jediným. Podobnými 'triky' se dá šetřit i čas výpočtu (počet provedených instrukcí) a získané výsledky pak neodpovídají realitě.

Pokud podobný efekt hrozí, uvažuje se místo jednotkové míry

míra logaritmická:

je-li v buňce uloženo číslo z, počítá se, že je takto zabrána paměť velikosti $\lceil \log_2(|z|+1)+1 \rceil$ (počet bitů potřebných k zapsání z; znaky $\lceil\ \rceil$ znamenají zaokrouhlení nahoru). Podobně i cena (čas) provedení jedné instrukce není 1, ale je úměrná velikosti čísel, se kterými se při provádění instrukce operuje.

V následující analýze algoritmů pro problém třídění budeme užívat jednotkovou míru. V tom případě ovšem naše analýza může dávat realistické výsledky jen tehdy, když je velikost tříděných čísel (předem) omezená (čísla mají omezený počet cifer); přesněji řečeno, když je rozumné předpokládat, že operace jako načtení čísla, porovnání dvou čísel apod. trvají konstantní čas.

Zkusme nyní naprogramovat algoritmus *bubblesort* pro RAM a analyzovat jeho časovou složitost. Výsledkem vcelku přímočarého 'přeložení' dříve uvedeného pseudokódu (*Bubblesort – progr. verze*) do 'jazyka' RAM může být níže uvedený program (předpokládáme v něm, že vstupní posloupnost není prázdná).

Vlastní RAM-program, tvořený posloupností 69 instrukcí je uveden v prvním 'sloupci'. V druhém sloupci je tentýž program v poněkud srozumitelnější podobě – užívá symbolických návěští a symbolického adresování (N=2, J=3, HM=4, I=5, IPJ=6, POM=7, X=8, A=8; člen $A[1]$ bude uložen v buňce 9, $A[2]$ v buňce 10, $A[3]$ v buňce 11 atd.). Třetí sloupec obsahuje komentáře, ze kterých by mělo být patrno, že se skutečně jedná o 'překlad' uvedené verze *bubblesortu*. (Připomeňme, že všechny paměťové buňky mají na začátku hodnotu 0.)

Bubblesort, RAM-verze

```
01 LOAD   =1     |            LOAD   =1     {do indexreg. se vloží 1, tj.}
02 STORE  1      |            STORE  1      {první volný index pole A }
```



```
03 READ          |  Cykl-vst: READ          {načtení dalšího vstupu    }
04 JZERO  10     |            JZERO  Kon-vst {0 znamená konec vstupu    }
05 STORE  *8     |            STORE  *A      {A[indexreg]:=vstup        }
06 LOAD   1      |            LOAD   1       {                          }
07 ADD    =1     |            ADD    =1      {indexreg. se zvýší o 1    }
08 STORE  1      |            STORE  1       {                          }
09 JUMP   3      |            JUMP   Cykl-vst{                          }
10 LOAD   1      |  Kon-vst:  LOAD   1       {                          }
11 SUB    =1     |            SUB    =1      {                          }
12 STORE  2      |            STORE  N       {N obsah. počet vst. čísel }
13 LOAD   3      |  Cykl-1:   LOAD   J       {                          }
14 ADD    =1     |            ADD    =1      {J:=J+1                    }
15 STORE  3      |            STORE  J       {                          }
16 LOAD   2      |            LOAD   N       {                          }
17 SUB    3      |            SUB    J       {                          }
18 JZERO  58     |            JZERO  Vystup  {skok při J=N              }
19 ADD    =1     |            ADD    =1      {                          }
20 STORE  4      |            STORE  HM      {HM (hor. mez) = N–J+1     }
21 LOAD   =0     |            LOAD   =0      {                          }
22 STORE  5      |            STORE  I       {I:=0                      }
23 LOAD   5      |  Cykl-2:   LOAD   I       {                          }
24 ADD    =1     |            ADD    =1      {I:=I+1                    }
25 STORE  5      |            STORE  I       {                          }
26 ADD    =1     |            ADD    =1      {                          }
27 STORE  6      |            STORE  IPJ     {IPJ=I+1                   }
28 LOAD   4      |            LOAD   HM      {                          }
29 SUB    5      |            SUB    I       {                          }
30 JZERO  13     |            JZERO  Cykl-1  {skok při I=HM             }
31 LOAD   5      |            LOAD   I       {                          }
32 STORE  1      |            STORE  1       {                          }
33 LOAD   *8     |            LOAD   *A      {                          }
34 STORE  8      |            STORE  X       {X:=A[I]                   }
35 LOAD   6      |            LOAD   IPJ     {                          }
36 STORE  1      |            STORE  1       {                          }
37 LOAD   8      |            LOAD   X       {                          }
38 SUB    *8     |            SUB    *A      {                          }
39 JGTZ   41     |            JGTZ   Prohod  {skok při X>A[I+1]         }
40 JUMP   23     |            JUMP   Cykl-2  {                          }
41 LOAD   5      |  Prohod:   LOAD   I       {                          }
42 STORE  1      |            STORE  1       {                          }
43 LOAD   *8     |            LOAD   *A      {                          }
44 STORE  7      |            STORE  POM     {POM:=A[I]                 }
45 LOAD   6      |            LOAD   IPJ     {                          }
```

```
46 STORE 1    |              STORE 1    {                          }
47 LOAD  *8   |              LOAD  *A   {                          }
48 STORE 8    |              STORE X    {X:=A[I+1]                 }
49 LOAD  5    |              LOAD  I    {                          }
50 STORE 1    |              STORE 1    {                          }
51 LOAD  8    |              LOAD  X    {                          }
52 STORE *8   |              STORE *A   {A[I]:=X                   }
53 LOAD  6    |              LOAD  IPJ  {                          }
54 STORE 1    |              STORE 1    {                          }
55 LOAD  7    |              LOAD  POM  {                          }
56 STORE *8   |              STORE *A   {A[I+1]:=POM               }
57 JUMP  23   |              JUMP  Cykl-2 {                        }
58 LOAD  =2   |  Vystup:     LOAD  =1   {                          }
59 STORE 1    |              STORE 1    {indexreg.:=1              }
60 LOAD  *8   |  Cykl-vys:   LOAD  *A   {                          }
61 WRITE      |              WRITE      {write(A[indexreg.])       }
62 LOAD  2    |              LOAD  N    {                          }
63 SUB   1    |              SUB   1    {                          }
64 JZERO 69   |              JZERO Konec {skok při indexreg.=N     }
65 LOAD  1    |              LOAD  1    {                          }
66 ADD   =1   |              ADD   =1   {                          }
67 STORE 1    |              STORE 1    {indexreg. se zvýší o 1    }
68 JUMP  60   |              JUMP  Cykl-vys {                      }
69 HALT       |  Konec:      HALT       {                          }
```

Spočtěme nyní, kolik instrukcí bude provedeno při zpracování vstupu velikosti n (v nejhorším možném případě).

První (vstupní) fáze výpočtu, od začátku po první příchod na instrukci s návěštím Cykl-1, zřejmě zabere 'čas' (tj. počet provedení instrukcí)

$$T1 = 2 + 7n + 2 + 3 = 7n + 7$$

Podobně třetí (výstupní) fáze, od skoku na Vystup po Konec, zabere zřejmě čas

$$T3 = 2 + 9(n-1) + 5 + 1 = 9n - 1$$

Více složitější je analyzovat zbylou (prostřední) fázi, od prvního skoku na Cykl-1 po skok na Vystup. Také s přihlédnutím k naší progr. verzi *bubblesortu* není ovšem zase tak obtížné odvodit, že ona prostřední fáze trvá

$$T2 = \Big(\sum_{j=1}^{n-1}\Big(10 + \Big(\sum_{i=1}^{n-j} 34\Big) + 8\Big)\Big) + 6$$

Poznamenejme, že vždy předpokládáme ten horší (tj. delší k zpracování) případ, kdy skutečně dojde k prohazování prvků (tj. provádí se 'podprogram' Prohod).

Výraz pro $T2$ můžeme upravovat standardní manipulací se sumami např. takto:

$$T2 = 6 + \sum_{j=1}^{n-1} 18 + \sum_{j=1}^{n-1}\sum_{i=1}^{n-j} 34 = 6 + 18(n-1) + \sum_{j=1}^{n-1} 34(n-j) =$$

$$= 18n - 12 + 34\sum_{j=1}^{n-1} n - 34\sum_{j=1}^{n-1} j$$

Dosadíme-li za první sumu $n(n-1)$ a za druhou sumu $(1+2+\ldots+n-1) = (n/2)(n-1)$, odvodíme pak již přímočarými úpravami vztah

$$T2 = 17n^2 + n - 12$$

Celkový čas potřebný pro zpracování vstupu velikosti n je tedy $T1 + T2 + T3 = 17n^2 + 17n - 6$. Označíme-li náš RAM-stroj M a jeho časovou složitost T_M, ukázali jsme tak, že pro každé $n \geq 1$ je

$$T_M(n) = 17n^2 + 17n - 6$$

Přednáška 2

---

2 Odhady složitosti; značení O aj.

Stručný obsah: Značení používané v odhadech složitosti (O, Θ, o, Ω, ω) a jeho význam. Ilustrace analýzy průměrného případu na příkladu quicksortu.

Studijní cíle: Umět vysvětlit a používat značení O, Θ, o, Ω, ω užívané v odhadech složitosti. Umět vysvětlit význam a ilustrovat klady a zápory analýzy složitosti v nejhorším možném a průměrném případě.

Při analýze časové složitosti *bubblesortu* jsme viděli, že přesné (algebraické) vyjádření funkce T_M není technicky úplně triviální úkol již u velmi jednoduchého programu. U větších a komplikovanějších programů by už takový postup byl nesmírně náročný.

Pro naše účely (srovnávání algoritmů) naštěstí přesné vyjádření časové složitosti není nutné; většinou postačí 'rozumný' odhad příslušné funkce T_M. Např. v našem případě jsme T_M vyjádřili ve tvaru $T_M(n) = an^2 + bn + c$, kde a, b, c jsou konstanty ($a = 17$, $b = 17$, $c = -6$). Když nám nezáleží na přesné hodnotě oněch konstant, mohli jsme analýzu urychlit (místo přesného počítání jsme konstanty mohli odhadovat shora – s určitou rozumnou rezervou) a dospět tak k (hornímu) odhadu např. $T_M(n) \leq 20n^2 + 50n + 100$.

Všimněme si, že pro získání podobného odhadu jsme *bubblesort* ani nemuseli fyzicky programovat pro RAM – pokud již máme určitou programátorskou zkušenost, dokážeme časovou složitost odhadnout např. již z pseudokódu (promyslete si to !).

Uvědomme si dále, že v našem vyjádření $T_M(n) = an^2 + bn + c$ má 'největší váhu' člen an^2. Byť by byla konstanta a 'hodně menší než' b (ale samozřejmě kladná), vždy od jistého n výše je hodnota an^2 (čím dál výrazněji) větší než hodnota 'zbytku' $bn + c$; exaktněji řečeno

$$\lim_{n\to\infty} \frac{bn+c}{an^2} = 0$$

Značení O, Θ, o, Ω, ω

Ono soustředění se na 'rozhodující člen' a zanedbávání přesných hodnot konstant nás vede k tzv. značení *velké-O*. O námi zjištěné funkci $T_M(n) = 17n^2 + 17n - 6$ pak prostě řekneme $T_M(n) \in O(n^2)$.

Přesněji uvedeme značení velké-O takto:

Definice 2.1 *Pro libovolné funkce $f, g : \mathcal{N} \to \mathcal{N}$ řekneme, že $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě tehdy, když platí* $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) \leq k \cdot g(n)$.

Je-li $f(n) \in O(g(n))$, říkáme také, že $f(n)$ roste řádově nejvýše jako $g(n)$. $O(g(n))$ tedy slouží jako určitý horní odhad funkce $f(n)$.

Když tedy řekneme, že "bubblesort je v $O(n^2)$" (což rozumíme jako zkratku pro "časová složitost algoritmu *bubblesort* je v $O(n^2)$"), pak není vyloučeno, že jej lze (shora) odhadnout lépe. Ve skutečnosti je ovšem funkce n^2 pro *bubblesort* nejen horním, ale i spodním (řádovým) odhadem (proč ?). K vyjádření podobných faktů se vedle O hodí i další značení (f, g jsou libovolné funkce $f, g : \mathcal{N} \to \mathcal{N}$; pro přehlednost zde opakujeme i definici O):

- $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě když platí
  $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) \leq k \cdot g(n)$
- $f \in \Theta(g)$, nebo $f(n) \in \Theta(g(n))$, znamená, že $f \in O(g)$ a $g \in O(f)$.
- $f \in o(g)$, označujeme též $f(n) \in o(g(n))$, právě když platí
  $(\forall k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : k \cdot f(n) < g(n)$
- $f \in \Omega(g)$, označujeme též $f(n) \in \Omega(g(n))$, právě když platí $g \in O(f)$, tj.
  $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : k \cdot f(n) \geq g(n)$
- $f \in \omega(g)$, označujeme též $f(n) \in \omega(g(n))$, právě když platí $g \in o(f)$, tj.
  $(\forall k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) > k \cdot g(n)$

Značení O, o hrají roli neostrého resp. ostrého horního odhadu, značení Ω, ω roli neostrého resp. ostrého dolního odhadu. (Jakou roli hraje značení Θ ?)

Jak už jsme se zmínili, uvedená značení se využívají např. pro možnost stručného vyjádření při analýze časové složitosti (podobně samozřejmě i u prostorové složitosti) konkrétního algoritmu, resp. příslušného (třeba ani fyzicky nesestrojeného) RAM-stroje M, kdy nám většinou postačí jen určitý odhad (růstu) funkce T_M, odhad, v němž se zanedbávají konstatní faktory.

Poznámka. Místo $f(n) \in O(g(n))$ (podobně pro další značení) se někdy (s vědomím záměrné nepřesnosti) píše $f(n) = O(g(n))$; tato notace je pak výhodnější např. v rovnicích, v nichž se značení O a další vyskytují.
Říkáme pak také např. 'časová složitost algoritmu XY je $O(n^2)$' místo přesnějšího '... je v $O(n^2)$'.

K *bubblesortu* můžeme dodat, že je v $\Theta(n^2)$. Potom fakt (který se dá přímočaře ukázat), že *heapsort* je v $O(n \log n)$ (je i v $\Theta(n \log n)$), skutečně prokazuje, že *heapsort* je lepší než *bubblesort* (pro dostatečně velké vstupy).

Poznámka. Čtenář je vyzýván, ať si provede srovnání růstu funkcí n, $n \log n$, n^2, n^3, 2^n, $n!$, n^n apod.
Je potřeba si také ujasnit, že např. (jenom) fakt, že algoritmus A má složitost $\Theta(n^2)$ a algoritmus B má složitost $O(n \log n)$, znamená, že A je zaručeně rychlejší než B jen *asymptoticky*, tj. pro 'dostatečně velké' hodnoty. (Záleží totiž na konstantách skrytých v značení Θ, O atd.)

Nejhorší vs. průměrný případ

Zamysleme se na chvíli nad zvoleným přístupem tzv. *nejhoršího možného případu*. Měli bychom si být alespoň vědomi, že tento přístup nemusí vždy poskytovat směrodatné výsledky pro praxi.

Běžně se tento fakt ilustruje na příkladu dalšího algoritmu třídění, tzv. *quicksortu*. Jeho časová složitost z hlediska nejhoršího možného případu je $\Theta(n^2)$, přitom v praxi je ale rychlejší než např. *heapsort* (který má složitost $\Theta(n \log n)$). Dá se totiž ukázat, že složitost *quicksortu* z hlediska *průměrného případu* je také $\Theta(n \log n)$, přičemž příslušná konstanta je menší než u *heapsortu*.

Poznámka. Jak čtenář očekává, u průměrného případu vyjadřuje $T(n)$ průměr z délek výpočtů pro všechny vstupy velikosti n. Předpokládá se tedy rovnoměrné rozložení pravděpodobnosti na množině vstupů. (Ve specifických případech může mít samozřejmě smysl uvažovat i jiné rozložení.)

Obvykle je ovšem analýza průměrného případu těžší než analýza nejhoršího možného případu. U námi dříve uvedené verze *bubblesortu* je sice vcelku zřejmé, že algoritmus má složitost $\Theta(n^2)$ i v průměrném případě (proč ?), u dále připomenutého *quicksortu* už analýza tak zřejmá není. Algoritmus *quicksort* (který pro parametry A, p, q seřadí v poli A prvky $A[p], A[p+1], \ldots, A[r]$ vzestupně) zde připomeneme jen pseudokódem.

```
procedure Quicksort(A, p, r) ;
if  p < r
then
   q := Partition(A, p, r)
   Quicksort(A, p, q)
   Quicksort(A, q+1, r)

procedure Partition(A, p, r) ;
x := A[p]; i := p − 1; j := r + 1;
while TRUE
do
   repeat j := j − 1 until A[j] ≤ x;
   repeat i := i + 1 until A[i] ≥ x;
   if  i < j then  exchange A[i], A[j] else  return j
```

Poznámka. Analýza složitosti průměrného případu bude předmětem referátu na cvičení.

3 Návrh a analýza konkrétních (rychlých) algoritmů

Stručný obsah: Obecné metody návrhu algoritmů. Konkrétní algoritmy a jejich časová složitost: algoritmy pro prohledávání (binární hledání, max. vzdálenost v polygonu), metoda 'rozděl a panuj' (merge sort, násobení matic), 'greedy' (tj. hltavé) algoritmy (výběr aktivit, minimální kostra), dynamické programování (nejdelší společná podsekvence, násobení řetězce matic).

Studijní cíle: Umět vysvětlit podstatu probíraných metod. Umět ilustrovat vhodnost použití jednotlivých metod a porozumět souvisejícím otázkám analýzy složitosti.

V této části si připomeneme základní obecné metody návrhu algoritmů. Budeme je ilustrovat na příkladech, které budeme analyzovat z hlediska časové složitosti.

3.1 Prohledávání

Součástí řešení mnohých problémů je nutnost prohledání jakéhosi prostoru (který obsahuje např. všechna 'přípustná řešení', z nichž je potřeba vybrat optimální).

Příkladem je hledání zadaného prvku v *utříděném* seznamu, např. jména v telefonním seznamu

'Naivní' algoritmus založený na myšlence

> projdi sekvenčně všechny prvky seznamu, přičemž každý z nich porovnáš se zadaným

má očividně složitost $\Theta(n)$ (jako velikost vstupu bereme počet prvků v seznamu). Čtenář ale zajisté řeší tento problém v praxi jinak a je schopen navrhnout algoritmus se složitostí $\Theta(\log n)$. (Zde je ovšem předpokládán přímý přístup k prvkům seznamu – u stroje RAM je tedy $\Theta(\log n)$ relevantní, pokud je již seznam uložen v paměti).

Uveďme další problém:

> *Název (označení) problému*: *Max. vzdálenost v polygonu*
>
> *(Očekávaný) vstup*: konvexní polygon v dvourozměrném prostoru – zadaný posloupností vrcholů $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ ('dokola', např. ve směru hodinových ručiček)
> *(Požadovaný) výstup*: dvojice vrcholů s max. vzdáleností.

Je přirozené považovat n, tj. počet vrcholů, za velikost vstupu.

Velmi jednoduše lze navrhnout algoritmus, který při hledání maxima probírá všechny dvojice vrcholů. Ten pak má celkem očividně složitost $\Theta(n^2)$. Ovšem při určitém větším vhledu a zamyšlení se nad problémem není těžké navrhnout algoritmus se složitostí $\Theta(n)$. Klíčem je idea prohledávání jen "perspektivních" (konkrétněji: "protilehlých") dvojic. (Zkuste domyslet pořebné detaily !)

3.2 Metoda 'rozděl a panuj'

Často použitelnou metodou při řešení 'velkého' úkolu (tj. nalezení výstupu pro 'velký' vstup určitého problému) je rozdělení na podúkoly, jejich vyřešení a nalezení hledaného výstupu zkombinováním mezivýsledků získaných z podúkolů.

Jestliže zmíněné podúkoly spočívají v řešení téhož problému pro menší vstupy, nabízí se rekurzivní algoritmus.

Pěkným příkladem je utřídění (velké) posloupnosti čísel. Tu je možné rozdělit na dvě části, utřídit každou zvlášť a výsledky pak 'slít' dohromady. Rekurzivní algoritmus založený na této myšlence se nazývá *Mergesort*.

Označíme-li $T(n)$ čas, který Mergesort spotřebuje při setřídění posloupnosti délky n, je zřejmé, že platí

- $T(1) = \Theta(1)$ ($\Theta(1)$ znamená prostě nějakou konstantu)
- pro $n \geq 2$: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + f(n)$

Zde $f(n)$ znamená čas potřebný k slévání a zřejmě je $f(n) = \Theta(n)$.

Pomineme-li zde nepodstatné komplikace se zaokrouhlováním (např. všechny zlomky si můžeme představovat zaokrouhlené nahoru), můžeme psát

$$T(n) = 2T(n/2) + \Theta(n).$$

Dá se ukázat (např. dosazením) že řešení uvedené rekurentní rovnice splňuje podmínku $T(n) = \Theta(n \log n)$ (stejně jako tomu bylo u *heapsortu*).

Dále uvedeme obecné tvrzení, které je užitečným nástrojem pro řešení podobných rekurentních rovnic.

Přednáška 3

Tvrzení 3.1 *Nechť $a \geq 1$, $b > 1$ jsou konstanty, f je funkce (typu $\mathcal{N} \to \mathcal{N}$) a pro funkci $T : \mathcal{N} \to \mathcal{N}$ platí rekurentní vztah*

$$T(n) = aT(n/b) + f(n).$$

Pak platí:

1. *Je-li $f(n) = O(n^c)$ a $c < \log_b a$, pak $T(n) = \Theta(n^{\log_b a})$.*
2. *Je-li $f(n) = \Theta(n^{\log_b a})$, pak $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *Je-li $f(n) = \Theta(n^c)$ a $c > \log_b a$, pak $T(n) = \Theta(n^c)$.*

Důkaz alespoň nastíníme, a sice pro případ $T(n) = aT(n/b) + n^c$. (Problémy se zaokrouhlováním ignorujeme.)

Představa rekurzivního výpočtu hodnoty $T(n)$ vede přirozeně k a-árnímu stromu, jehož hloubka je $\log_b n$; počet listů je tedy $a^{\log_b n}$ neboli $n^{\log_b a}$ (je totiž $\log_b a^{\log_b n} = (\log_b n) \cdot (\log_b a) = \log_b n^{\log_b a}$).

Předpokládáme-li rovnou, že $T(1) = 1$, pak se $T(n)$ rovná následujícímu součtu:

$$n^c + a\left(\frac{n}{b}\right)^c + a^2\left(\frac{n}{b^2}\right)^c + \ldots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^c =$$

$$= n^c + n^c\left(\frac{a}{b^c}\right) + n^c\left(\frac{a}{b^c}\right)^2 + \ldots + n^c\left(\frac{a}{b^c}\right)^{\log_b n} =$$

$$= n^c\left(1 + \left(\frac{a}{b^c}\right)^1 + \left(\frac{a}{b^c}\right)^2 + \ldots + \left(\frac{a}{b^c}\right)^{\log_b n}\right)$$

Připomeňme si součet (začátku) geometrické řady

$$1 + q + q^2 + \ldots + q^k = \frac{q^{k+1} - 1}{q - 1}$$

V případě $0 < q < 1$ máme $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$,

v případě $q = 1$ máme $\sum_{i=0}^{k} q^i = k + 1$ a

v případě $q > 1$ máme $\sum_{i=0}^{k} q^i = \frac{q^{k+1}}{q-1} - \frac{1}{q-1} = O(q^k)$.

Tedy v případě

- $\frac{a}{b^c} < 1$ (tj. $\log_b a < c$) máme
$$T(n) \leq n^c \frac{1}{1 - \frac{a}{b^c}} = \Theta(n^c)$$
- $\frac{a}{b^c} = 1$ (tj. $\log_b a = c$) máme
$$T(n) = n^c(1 + \log_b n) = \Theta(n^{\log_b a} \log_b n)$$
- $\frac{a}{b^c} > 1$ (tj. $\log_b a > c$) máme
$$T(n) = n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n + 1} \cdot \frac{1}{\frac{a}{b^c} - 1} - n^c \cdot \frac{1}{\frac{a}{b^c} - 1}$$
Tedy $T(n) = \Theta(n^c \cdot n^{\log_b \frac{a}{b^c}})$. Jelikož $\log_b \frac{a}{b^c} = \log_b a - \log_b b^c = \log_b a - c$, dostáváme $T(n) = \Theta(n^{\log_b a})$.

Všimněme si, že u našeho příkladu Mergesort nastával 2. případ ($a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta(n^{log_2 2})$), což dává onen výsledek $T(n) = \Theta(n \log n)$.

Podívejme se teď na problém násobení matic

> *Název (označení) problému*: *Násobení dvou matic*
>
> *(Očekávaný) vstup*: dvě čtvercové matice A, B rozměrů $n \times n$
> *(Požadovaný) výstup*: matice C tž. $C = AB$.

Předpokládáme, že prvky matice jsou celá čísla (omezené velikosti). Z hlediska našeho vnímání je zde vhodné jako velikost vstupu považovat číslo n, ačkoliv počet zadávaných čísel je $\Theta(n^2)$ (je pro nás totiž přirozenější tvrzení 'program za 1 minutu zvládne násobení matic 800×800' než '... matic s 640000 prvky').

Přesvědčte se, že běžný algoritmus (podle definice násobení) má složitost $\Theta(n^3)$, dá-li se omezit konstantou čas pro provedení jedné aritmetické operace. (Tento odhad se samozřejmě vztahuje k uvedenému chápání velikosti vstupu. Kdybychom jako velikost vstupu brali počet vstupních čísel, dostali bychom odhad $\Theta(n^{1.5})$!)

Podívejme se, jestli přístup 'rozděl a panuj' přinese zlepšení. Omezíme se na zkoumání případů, kdy n je mocninou dvojky (jinak bychom toho mohli docílit příslušným doplněním matic nulami, čímž by se velikost vstupu zvětšila méně než dvakrát).

Matice A vznikne 'přirozeným poskládáním' ze čtyř čtvercových matic rozměrů $n/2 \times n/2$, označme tyto matice A_{11}, A_{12}, A_{21}, A_{22}. Podobně označme příslušné podmatice pro B a C. Snadno ověříme, že

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Pro časovou složitost $T(n)$ odvodíme z tohoto postupu rekurentní vztah

$$T(n) = 8T(n/2) + \Theta(n^2),$$

což dá rovněž řešení $T(n) = \Theta(n^3)$.

Strassen ovšem vymyslel postup, při kterém se jedno násobení dá ušetřit (za cenu zvýšení počtu sčítání), a kde pak příslušná rovnice je ve tvaru

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Výsledná složitost $T(n) = \Theta(n^{\log_2 7})$ (pozn.: $\log_2 7$ je zhruba 2.81) je asymptoticky lepší než u standardního algoritmu.

(Postup ve Strassenově algoritmu a další jeho aspekty budou diskutovány na cvičení.)

3.3 'Greedy' algoritmy

Slovo 'greedy' zde budeme překládat jako 'hltavý' (autor si není vědom zaužívaného českého termínu).

Hltavý přístup se osvědčuje u některých optimalizačních problémů. Jsou to problémy, u nichž je třeba udělat řadu rozhodnutí – lokálních kroků. Hltavý přístup vždy volí z možných kroků ten (lokálně) nejnadějnější.

Příslušný algoritmus je pak většinou velmi jednoduchý, ovšem použitelný pro daný problém je jen tehdy, jestliže série oněch lokálně nejnadějnějších rozhodnutí skutečně vede ke globálně optimálnímu řešení. Tuto pěknou vlastnost samozřejmě nemají všechny optimalizační problémy a důkaz toho, že daný problém (resp. daný hltavý přístup) tuto vlastnost má, je často obtížný (obvykle mnohem obtížnější než návrh příslušného algoritmu).

Uvedeme příklad úspěšného použití hltavého algoritmu:

> *Název problému*: *Výběr aktivit*
>
> *Vstup*: množina konečně mnoha aktivit $\{1, 2, \ldots, n\}$ s pevně určenými časovými intervaly (s_1, f_1), (s_2, f_2), …, (s_n, f_n), kde $(\forall i, 1 \leq i \leq n) : s_i < f_i$
> *Výstup*: množina obsahující největší možný počet vzájemně kompatibilních aktivit (tj. aktivit s vzájemně se nepřekrývajícími intervaly)

Je možné si např. představit, že máme dán seznam, kde každý prvek seznamu je nějaká přednáška, cvičení, školení, či jiná aktivita s pevně přidělenou dobou konání (tj. s přiděleným počátkem s a koncem f). Jde o to, umístit maximum z těchto aktivit do jedné

posluchárny. Poznamenejme nejdříve, že přístup *hrubou silou*, spočívající v prověření všech možných výběrů aktivit, je exponenciální složitosti (časová složitost příslušného algoritmu je $2^{\Theta(n)}$) a již při 'malém' n nepoužitelný.

Hltavým způsobem můžeme řešit problém např. takto:

> Dokud to jde, opakuj následující krok:
> vyber aktivitu, která je kompatibilní s dosud vybranými (na začátku nejsou vybrány žádné) a skončí co nejdříve (když je takových víc, tak kteroukoli z nich).

Hltavost, či maximální 'lokální nadějnost' spočívá v tom, že po každém takovém výběru zbyde maximum času pro další aktivity.

Poznámka. Všimněme si jisté 'nedeterminističnosti' našeho problému – k danému vstupu může uvedené podmínce odpovídat více výstupů. I uvedený postup není plně deterministický (proč ?). Řešíme-li takový problém (deterministickým) algoritmem, upřednostňujeme vlastně vždy jeden výstup mezi všemi možnými. Tak je tomu i v dále uvedeném pseudokódu.

Je zřejmé, že je užitečné aktivity nejdříve setřídit podle koncových časů; to je možné udělat nějakým známým algoritmem v čase $O(n \log n)$ (počet aktivit budeme považovat za velikost vstupu).

```
{předp., že poč. časy jsou již v poli s}
{a konc. časy v poli f}
{dále již předp. f[1] ≤ f[2] ≤ ... f[n],}
{kde n představuje počet aktivit}
A := {1}; j := 1;
for i := 2 to n do
   if s[i] ≥ f[j] then
      (A := A ∪ {i}; j := i)
{vybr. aktivity jsou v množ. A}
```

V uvedeném případě je důkaz faktu, že hltavý přístup skutečně vede k optimálnímu řešení, celkem snadný. (Indukcí podle počtu aktivit n.)

Jak jsme již uvedli, počet aktivit n je zde přirozenou mírou velikosti vstupu a snadno odvodíme, že uvedený algoritmus má složitost $\Theta(n)$, když předpokládáme, že aktivity jsou již utříděny podle koncových časů.

Typickým příkladem úspěšného použití hltavého přístupu je problém minimální kostry v grafu ($\mathcal{N}_+$ označuje množinu všech kladných celých čísel):

Název problému: *Minimální kostra*

Vstup: neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E_G \to \mathcal{N}_+$
Výstup: graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, který je souvislý a přitom $\sum_{e \in E_H} f(e)$ je minimální.

Jednou možnou interpretací je problém stavitele železnic. Má dánu množinu měst (vrcholů grafu) a dále všechna možná spojení mezi dvojicemi měst, kudy je možné vést železnici. Každému takovému možnému spojení (hraně grafu) odpovídají určité náklady na postavení železnice (ohodnocení příslušné hrany). Úkolem stavitele je postavit železnici tak, aby každé město bylo spojeno s každým (případně přes další města) a aby náklady stavby byly co nejmenší.

Opět poznamenejme, že přístup hrubou silou (probrání všech možností postavení železnice) vede k exponenciálnímu algoritmu a nepřipadá pro větší vstupy v úvahu.

Všimněme si nyní, že řešení (tj. graf H) je určitě stromem; kdyby ne, obsahoval by cyklus a mohli bychom pak při zachování souvislosti jednu hranu odstranit a dostat tak lepší řešení. Jednou z 'hltavých' možností je následující přístup 'líného' stavitele:

Postav nádraží v libovolném městě.
Dokud to jde, opakuj:
k dosud postavené železnici připoj co nejlevnější úsek (hranu), ovšem tak, aby nevznikl cyklus.

Tento postup je zachycen následujícím pseudokódem. (Zde necháváme určitý nedeterminismus i v pseudokódu; samozřejmě jakýkoli deterministický výběr z příslušných možností vede k cíli – tj. k jedné z minimálních koster grafu).

{předp., že jsou dány V_G, E_G, f}
zvol lib. $u \in V_G$; $V_H := \{u\}$; $E_H := \emptyset$; $M := V_G - \{u\}$;
for each $v \in M$ **do**
  if $(u, v) \in E_G$
  then $(otec(v) := u$; $d(v) := f((u, v)))$
  else $d(v) := \infty$;
while $M \neq \emptyset$ **do**
  najdi $v \in M$ tž. $d(v) = min\{d(w) \mid w \in M\}$;
  $V_H := V_H \cup \{v\}$; $E_H := E_H \cup \{(v, otec(v))\}$; $M := M - \{v\}$;
  for each $w \in M$ **do**
    if $(v, w) \in E_G$ and $f((v, w)) < d(w)$
    then $(otec(w) := v$; $d(w) := f((v, w)))$
return $H = (V_H, E_H)$

Není samozřejmé, že tento přístup vede k optimálnímu řešení, a je to potřeba dokázat.

Ukažme sporem. Uvažujme první situaci, kdy náš algoritmus k dosud vytvořenému stromu T, který je (dosud) podgrafem nějaké minimální kostry K, přidá hranu (u, v) způsobící, že takto vzniklý strom (již) není podgrafem žádné minimální kostry. V K ovšem vede cesta z u do v přes nějakou hranu (x, y), kde $x \in T$ a $y \notin T$. Odejmu-li ovšem z K hranu (x, y) a přidám (u, v), dostanu opět minimální kostru (promyslete si to !), což je spor.

Pokud za velikost vstupu považujeme počet vrcholů n (kolik hran pak graf může mít ?), je složitost uvedeného algoritmu $\Theta(n^2)$. Poznamenejme, že existují i jiné algoritmy; pro jejich srovnání je pak vhodné uvažovat složitost jako funkci dvou parametrů: počtu vrcholů n a počtu hran m (jeden z nich má např. složitost $O(m \log n)$).

Přednáška 4

3.4 Dynamické programování

Pojem 'dynamické programování' pochází z teorie řízení a nemá nic společného s běžným významem tohoto sousloví v oblasti programování počítačů. Zde se jedná o název metody, při které se problém řeší pomocí postupného (vyplňování 'tabulky') řešení podproblémů (od nejmenších po největší). (Někteří autoři hledají jiné názvy, např. dynamické plánování).

Metoda dynamického programování je jakýmsi limitním případem metody 'rozděl a panuj'. Řešení (velké) instance (tj. vstupu) problému také spočívá v kombinaci výsledků podúkolů; jelikož ovšem při řešení různých podúkolů by mnohokrát docházelo k řešení společných podpodúkolů (atd.), je lepší rekurzivní řešení (přístup shora-dolů) nahradit přístupem zdola-nahoru: nejdříve se vyřeší všechny potřebné elementární (nejmenší) úkoly, z jejich řešení pak odvodíme řešení větších úkolů, z nich pak řešení ještě větších atd. až získáme řešení našeho (velkého) úkolu.

Tento přístup se někdy uplatní u optimalizačních úloh, u kterých selhávají hltavé algoritmy.

Jedním z typických příkladů je problém *nalezení nejdelší společné podsekvence*. Řekneme, že slovo (řetězec, posloupnost) u je podsekvencí slova v, jestliže u dostaneme z v vymazáním některých (třeba žádných) výskytů symbolů (členů posloupnosti). Instancí zmíněného problému jsou dvě slova v, w a úkolem je najít nejdelší slovo u, které je podsekvencí slova v i slova w.

Podrobněji budeme metodu dynamického programování ilustrovat na jiném příkladu:

Název problému: *Násobení řetězce matic*

Vstup: řetězec matic $A_1 A_2 \dots A_n$
Výstup: plně uzávorkovaný součin $A_1 A_2 \dots A_n$ tž. při použití standardního algoritmu násobení matic se vykoná minimální počet skalárních násobení.

Připomeňme, že součin matic AB, kde A má rozměry $k \times \ell$ a B má rozměry $m \times n$, je definován jen v případě $\ell = m$ (výsledná matice má rozměry $k \times n$); počet potřebných skalárních násobení je zde $k\ell n$.

Budeme tedy předpokládat, že pro každé $i, 1 \leq i \leq n$ má matice A_i rozměry $p_{i-1} \times p_i$, kde $p_0, p_1, \ldots, p_n$ jsou příslušná celá kladná čísla. Všimněme si, že vlastně tato čísla jsou podstatná v našem problému, nikoli hodnoty prvků jednotlivých matic.

Navíc připomeňme, že násobení matic je asociativní, takže nezáleží na pořadí násobení dvojic matic, které zvolíme při výpočtu součinu $A_1 A_2 \ldots A_n$ (pořadí násobení dvojic matic jednoznačně určíme příslušným vložením závorek).

Čtenář se snadno přesvědčí (konstrukcí jednoduchého příkladu), že počty potřebných skalárních násobení se při různých uzávorkováních mohou výrazně lišit.

Dá se také ukázat, že počet možných uzávorkování roste s rostoucím n exponenciálně, takže probrání všech možností nepřipadá v úvahu (s výjimkou malých hodnot n).

Všimněme si, že optimální vynásobení řetězce $A_1 A_2 \ldots A_n$ lze popsat tak, že nejdříve se optimálně vynásobí řetězec $A_1 A_2 \ldots A_k$, potom řetězec $A_{k+1} A_{k+2} \ldots A_n$ a na závěr se vynásobí získané dva mezivýsledky; k je ovšem (neznámý) 'optimální index' v rozmezí $1 \leq k \leq n-1$ (k určuje umístění 'nejvnějšnějších závorek').

Zmíněný optimální index označíme $s[1, n]$; obecně $s[i, j]$ $(1 \leq i < j \leq n)$ označuje optimální index při násobení řetězce $A_i A_{i+1} \ldots A_j$. Označme dále jako $m[i, j]$ počet skalárních násobení při optimálním vynásobení řetězce $A_i A_{i+1} \ldots A_j$ (zde připouštíme i rovnost $i = j$; pochopitelně $m[i, i] = 0$). Všimněme si, že je-li $s[i, j] = k$, pak $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Následující procedura, parametrizovaná vektorem čísel (rozměrů matic) $p = (p_0, p_1, \ldots, p_n)$ postupně vyplní 'tabulky' m a s:

```
Matrix-chain-order(p)
n := length(p) − 1;
for i := 1 to n do m[i, i] := 0;
for ℓ := 2 to n do
  for i := 1 to n − ℓ + 1 do
    j := i + ℓ − 1; m[i, j] := ∞;
    for k := i to j − 1 do
      q := m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j;
      if q < m[i, j] then (m[i, j] := q; s[i, j] := k);
return m, s
```

Je snadné vyvodit, že složitost Matrix-chain-order je $O(n^3)$ (a také $\Omega(n^3)$, tedy $\Theta(n^3)$). Vlastní program násobení řetězce $A_1 A_2 \ldots A_n$, označme tento řetězec A, pak spočívá ve vytvoření tabulky s pomocí procedury Matrix-chain-order (volané pro příslušný vektor rozměrů) a v následném vyvolání Matrix-chain-multiply$(A, s, 1, n)$, kde procedura Matrix-chain-multiply je rekurzivně definována takto:


```
Matrix-chain-multiply(A, s, i, j)
if j > i
then
  X :=Matrix-chain-multiply(A, s, i, s[i, j])
  Y :=Matrix-chain-multiply(A, s, s[i, j] + 1, j)
  return X · Y
else
  return A_i
```

4 Problémy, třídy složitosti problémů, horní a dolní odhady

Stručný obsah: Další poznámky k analýze složitosti algoritmů. Vymezení pojmu 'problém' (nebo též 'výpočetní problém'); speciální případ ANO/NE problému (tj. tzv. rozhodovacího problému). Složitost problému; třídy časové a prostorové složitosti. Algoritmy jakožto horní odhady. Dolní odhady, obtížnost jejich získávání. Mezery mezi známými horními a dolními odhady složitosti problémů.

Studijní cíle: Umět definovat a ilustrovat pojem 'problém', umět vysvětlit způsob definice složitosti problémů, horních a dolních odhadů. Schopnost vše ilustrovat na příkladech.

Poznámka. Kromě výrazů 'stroj RAM' či 'RAM-stroj' budu užívat jen zkratku 'RAM'; budu např. mluvit o sestrojení RAMu apod.

Další poznámky k složitosti algoritmů

Vrátíme se k některým věcem týkajícím se (časové či paměťové) složitosti algoritmů, o kterých jsme zatím explicitně moc nemluvili, ale kterých bychom si měli být velmi dobře vědomi.

Zatím jsme přesně definovali pojem časové (a prostorové) složitosti RAMu. Většinou jsme ale uvedli algoritmus zapsaný pseudokódem a hovořili jsme o složitosti tohoto algoritmu. Striktně vzato bychom ale vždy místo o složitosti algoritmu A měli hovořit o složitosti RAMu M_A, který je možné k algoritmu A v podstatě mechanicky sestrojit (tj. který by byl z A vytvořen určitým 'překladačem').

Nedefinovali jsme ovšem, jaké příkazy se mohou objevovat v pseudokódu, ani jsme samozřejmě nedefinovali příslušný překladač. Mlčky jsme vlastně udělali určitou úmluvu: provedli jsme konstrukci příslušného RAMu jen pro několik málo algoritmů zapsaných pseudokódem a spoléháme se na to, že algoritmy popisujeme vždy dostatečně podrobně k tomu, abychom v případě potřeby mohli příslušný RAM přímočaře zkonstruovat. Přitom samozřejmě předpokládáme, že zmíněnou přímočarou konstrukcí bychom všichni dospěli v podstatě k jednomu a témuž RAMu. To 'jednomu a témuž' nelze brát úplně doslova, stačí samozřejmě, že příslušné RAMy by měly v podstatě stejnou složitost – to jest, mohly

by se lišit v konkrétních počtech instrukcí realizujících ten či onen příkaz pseudokódu, což ovšem nevadí, nebazírujeme-li na přesných hodnotách příslušných konstant.

Právě to, že přesné hodnoty konstant pro nás nejsou podstatné a složitost vždy jen určitým způsobem odhadujeme (aproximujeme), nám umožňuje používat způsob analýzy složitosti algoritmů, při němž se přímo nezmiňujeme o RAMu v pozadí, natož abychom ho konstruovali.

Poznamenejme ještě další věc. Vstupem pro RAM je vlastně posloupnost čísel. Takže chceme-li mluvit o složitosti algoritmu, měl by vlastně jeho vstup (i výstup) být posloupností čísel – nebo by mělo být jasné, jaké kódování vstupu pomocí posloupnosti čísel máme na mysli. U námi dosud zkoumaných problémů to bylo v podstatě zřejmé: např. graf lze přirozeně zadat incidenční maticí, matici je možné přímočaře 'linearizovat' (zapsat např. po řádcích – s dohodnutými oddělovači) apod.

Připomeňme si ještě, že velikost vstupu u RAMu jsme definovali jako počet členů vstupní posloupnosti čísel (to je v případě uvažování jednotkové míry; v případě použití logaritmické míry je velikost vstupu v podstatě rovna počtu bitů, do kterých lze vstup zapsat). Ovšem když jsme např. analyzovali problém násobení matic, brali jsme za míru velikosti vstupu číslo udávající rozměr matic; přitom (ono přirozené linearizované) zadání čtvercové matice s rozměrem n (tedy typu $n \times n$) zabere n^2 vstupních buněk RAMu (případně další buňky jako oddělovače řádků).

Když tedy hovoříme o složitosti algoritmu v takovém případě, vztahujeme ji pořád k příslušnému RAMu, ale měníme standardní definici velikosti vstupu – to musíme vždy jasně říci. Jak jsme viděli, děláme to tehdy, když pro daný problém je pozměněná definice velikosti vstupu vhodnější při vyjadřování a umožňuje 'průhlednější' podání výsledků analýzy složitosti. Jak jsme také zmínili v případě minimální kostry grafu, je z takových důvodů někdy vhodné popisovat velikost vstupu více čísly (např. počet vrcholů n, počet hran m); složitost je pak funkcí více parametrů.

Všechny uvedené poznámky je vhodné si důkladně promyslet a u každého konkrétního případu je nutné si uvědomovat, co je (třeba mlčky) předpokládáno.

Některé aspekty si ještě osvětlíme na příkladu problému, který hraje důležitou roli např. v kryptografii.

Název: *Prvočíselnost*

Vstup: přirozené číslo k
Výstup: ANO, když k je prvočíslo, NE, když k je číslo složené.

Pravděpodobně nás rychle napadne následující algoritmus:

{předp. vstup $k \geq 2$}
$hm := \lfloor \sqrt{k} \rfloor$;
for $i := 2$ **to** hm **do**
  if $k \bmod i = 0$ **then return** NE
  return ANO

Když chceme odhadnout složitost algoritmu, musí být samořejmě jasné, co se rozumí velikostí vstupu. Jelikož vstupem je jedno číslo, má být velikost vždy 1 ? U předchozích problémů (jako třeba násobení matic) jsme předpokládali omezenou velikost zadávaných čísel (a tedy konstantní čas při aritmetických operacích s nimi) – neomezovali jsme ovšem délku zadávané posloupnosti. Nyní ovšem neomezujeme velikost (jednoho) zadávaného čísla. Takový vstup si 'přímo říká' o použití logaritmické míry, kde tedy velikost vstupu pro vstupní číslo k je rovna $\log_2 k$ (přesněji $\lceil \log_2(k+1) + 1 \rceil$, což ale není podstatné).

Uvedený algoritmus má takto exponenciální časovou složitost; všimněte si, že v případě, že k je prvočíslo, provede se tělo cyklu zhruba $k^{0.5}$ krát, tj. $2^{0.5 \log_2 k}$, tedy $2^{\Theta(n)}$ krát, kde n je velikost vstupu. Z toho je vidět, že algoritmus je použitelný jen na čísla s malým počtem míst v dekadickém zápisu (a rozhodně ho nelze použít např. na 100-místná čísla vyskytující se v kryptografických problémech).

Všimněme si ještě, že kdybychom za velikost vstupu považovali přímo hodnotu čísla k (číslo bychom vlastně zadávali v unární soustavě – jako posloupnost k jedniček), byla by složitost algoritmu určitě v $O(n^2)$; 'najednou' by to bylo v praxi zvládnutelné pro vstupy délky 100 (problém je samozřejmě v tom, že např. vstup délky 100 v tomto případě odpovídá vstupu délky 3 v případě předchozím).

Definice pojmu 'problém'

Algoritmy jsme prezentovali jako návody k řešení určitých problémů. Slovo 'problém' má v přirozeném jazyce hodně významů; co ale znamená tento pojem v našem kontextu ?

Připomeňme, že problémy jsme zadávali většinou následujícím schématem:

Název (označení problému): XY

(Očekávaný) vstup: zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému
(Požadovaný) výstup: zde je popsáno, jaký výstup (výsledek) je očekáván pro zadaný vstup (je přiřazen zadanému vstupu)

Pokud chceme napsat formální definici pojmu problém, mohla by vypadat takto:

Definice 4.1 Problém *je určen trojicí* (IN, OUT, p)*, kde* IN *je množina (přípustných)* vstupů*,* OUT *je množina* výstupů *a* p *je zobrazení (tedy funkce)* $p : IN \to OUT$.

Takto definovaný problém se někdy též nazývá 'výpočetní problém' (computational problem).

Jak jsme už řekli, aby mělo smysl hovořit o tom, že určitý RAM řeší daný problém, musí být množiny IN i OUT jistými množinami posloupností celých čísel, tedy $IN, OUT \subseteq \mathcal{Z}^*$ (kde $\mathcal{Z}$ označuje množinu všech celých čísel):

Definice 4.2 RAM M řeší problém $P = (IN, OUT, p)$, *kde* $IN, OUT \subseteq \mathcal{Z}^*$, *jestliže má tuto vlastnost:*
začne-li výpočet v počáteční konfiguraci se vstupem $c_1c_2 \dots c_n \in IN$, *pak svůj výpočet (regulérně) skončí, přičemž na výstupu je* $p(c_1c_2 \dots c_n)$.

Jak jsme již poznamenali, u námi zkoumaných problémů byly přípustné vstupy de facto posloupnosti čísel. Ovšem např. v problému vyhledávání vzorku v textu

> *Název*: *Vzorek v textu*
>
> *Vstup*: text t ('dlouhá' posloupnost symbolů), vzorek p ('krátká' posloupnost symbolů)
> *Výstup*: místo prvního výskytu p v t, či odpověď 'nevyskytuje se'.

čísla přímo nevystupují. Stačí ale kódovat užité symboly čísly (vzpomeňte např. na ASCII-kód) a máme vstupy a výstupy opět v žádaném tvaru.

Poznámka. Čísla zapisujeme většinou (v dekadické soustavě) jako řetězce symbolů z určité konečné abecedy. Když se na chvíli zamyslíme, uvědomíme si, že vlastně lze rozumně předpokládat, že jakýkoli vstup jakéhokoli problému lze vždy (více či méně elegantně) ztotožnit s řetězcem symbolů z pevně dané abecedy (např. '1-bytové' abecedy s 256 prvky).
Proto si lze pod množinou přípustných vstupů vždy představit množinu řetězců ze Σ^* (kde Σ je ona pevná abeceda), které splňují nějakou algoritmicky snadno verifikovatelnou podmínku (např. jsou dohodnutým zápisem řetězce matic apod.)
Podobně i množinu výstupů je možné ztotožnit s (pod)množinou (množiny) Σ^*.

Speciálním případem problémů jsou tzv.

rozhodovací problémy, neboli *ANO/NE problémy.*

U takového problému je množina OUT dvouprvková; standardně pak předpokládáme, že OUT= {ANO,NE} (či OUT= $\{1, 0\}$).

Příkladem ANO/NE problému je výše uvedený problém prvočíselnosti. Poznamenejme, že k některým (obecným) problémům (např. optimalizačním) lze přirozeně přiřadit tzv. rozhodovací (ANO/NE) verzi problému: např. u problému minimální kostry se vstup rozšíří o číslo c a požadovaný výstup pak bude ANO, jestliže ex. kostra s ohodnocením nejvýše rovným c, a NE v opačném případě. V této formě je pak přirozenější zadat problém schématem

> *Název*: *Minimální kostra (ANO/NE verze)*
>
> *Vstup*: neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E \to \mathcal{N}_+$, číslo c
> *Otázka*: existuje graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, který je souvislý a přitom $\sum_{e \in E_H} f(e) \leq c$?

Obecně běžné schéma pro zadávání ANO/NE problémů je tedy

> *Název (označení problému)*: *XY*
>
> *Vstup*: zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému.
> *Otázka*: zde je otázka týkající se (zadaného) vstupu, na niž je odpověď ANO nebo NE.

U problému prvočíselnosti by otázka samozřejmě zněla: 'je k prvočíslo ?'.

Uveďme si ještě jeden ANO/NE problém, který bude hrát důležitou roli v dalším.

> *Název*: SAT (*problém splnitelnosti booleovských formulí*)
>
> *Vstup*: booleovská formule v konjunktivní normální formě
> *Otázka*: je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá) ?

Složitost problémů

Intuitivně cítíme, že různé problémy mohou být různě 'složité'; co to ale je ona složitost problémů ? Zatím jsme hovořili jen o složitosti algoritmů, přesněji řečeno RAMů. Víme-li např. o algoritmu (RAMu), který daný problém řeší a má složitost $\Theta(n^3)$, je toto $\Theta(n^3)$ jen určitým horním odhadem 'skutečné' složitosti problému (můžeme říci 'složitost problému je nejvýše kubická'). Je např. možné, že nalezneme jiný algoritmus, který řeší náš problém a který má složitost $O(n^2)$; tím jsme náš dosud známý odhad zlepšili (víme už, že 'složitost problému je nejvýše kvadratická'). Tyto úvahy nás přivedou k závěru, že složitost problému lze ztotožnit se složitostí 'optimálního' algoritmu, který daný problém řeší. Při exaktní definici onoho pojmu 'optimální' ovšem vzniknou jisté komplikace. Ty obejdeme použitím následujícího přístupu:

Definice 4.3 *Pro funkci* $f : \mathcal{N} \longrightarrow \mathcal{N}$ *rozumíme* třídou časové složitosti $\mathcal{T}(f)$, *též značenou* $\mathcal{T}(f(n))$, *množinu těch problémů, které jsou řešeny RAMy s čas. složitostí v* $O(f)$.
Třídou prostorové složitosti $\mathcal{S}(f)$, *též* $\mathcal{S}(f(n))$, *rozumíme třídu těch problémů, které jsou řešeny RAMy s prostorovou složitostí v* $O(f)$.

Důležitá úmluva. V předchozí definici a všude, kde hovoříme o třídách složitosti vztahujících se k RAMu jako k referenčnímu modelu, máme vždy na mysli **logaritmickou míru**

při odkazu na složitost RAMu. Jde o to, aby takto definované třídy složitosti rozumně odpovídaly realitě.

Řekneme-li tedy např., že problém P patří do $\mathcal{T}(n^2)$ (taky se v této souvislosti říká 'časová složitost P je v $O(n^2)$' či ještě stručněji 'P je v $O(n^2)$'), říkáme tím, že existuje algoritmus s nejvýš kvadratickou časovou složitostí, který řeší problém P (přesněji řečeno: existuje RAM s nejvýš kvadratickou časovou složitostí v logaritmické míře, který řeší problém P).

Poznámka. Připomeňme, že pro příslušnost problému k dané třídě složitosti je důležitý i způsob kódování vstupu (a ten je tedy nutno chápat jako součást definice problému).

Všimněme si, že platí

$$(P \in \mathcal{T}(f)) \wedge (f \in O(g))) \Longrightarrow (P \in \mathcal{T}(g)).$$

Takže např.

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

Jak jsme již řekli, algoritmy řešící daný problém poskytují *horní odhady složitosti problému*. Horší je to s *dolními odhady*. Chceme-li např. ukázat, že daný problém je v $\mathcal{T}(n^2)$, stačí ukázat algoritmus se složitostí $O(n^2)$, který jej řeší. Chceme-li ale ukázat, že problém není v $\mathcal{T}(n^2)$, musíme ukázat, že žádný algoritmus (RAM), který jej řeší, nemá složitost v $O(n^2)$. Získat takový dolní odhad je obvykle velmi těžké.

Na cvičení si ukážeme, že každý algoritmus řešící problém třídění (a založený na porovnávání dvojic) má nutně složitost $\Omega(n \log n)$. Složitost problému třídění je takto určena přesně (samozřejmě až na zanedbávané konstantní faktory) – horní odhad se rovná dolnímu.

Poznamenejme, že u mnohých problémů, se kterými jsme se setkali a setkáme, je velká 'propast' mezi (dosud známými) horními a dolními odhady. Např. pro výše uvedený problém SAT je známý horní odhad exponenciální, ovšem známý dolní odhad je pouze lineární (tj. $\Omega(n)$) ! (S dalšími problémy tohoto typu se setkáme v části o NP-úplných problémech).

Přednáška 5

5 Třída PTIME a její robustnost. Turingovy stroje.

Stručný obsah: P (=PTIME) jako aproximace třídy prakticky zvládnutelných problémů. Vysvětlení Její robustnosti vůči (rozumným) výpočetním modelům. Turingovy stroje; vzájemná 'polynomiální' simulace s RAMy. Zmínka o dalších výpočetních modelech a jejich vzájemné polynomiální simulaci.

Studijní cíle: Umět vysvětlit význam a robustnost třídy PTIME. Umět definovat a ilustrovat Turingovy stroje, vysvětlit ideje polynomiální simulace s RAMy.

Část teorie, která se někdy nazývá *konkrétní složitost*, studuje složitost konkrétních problémů (a algoritmů), resp. příslušné horní a dolní odhady. Tzv. *strukturální složitost* 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', tj. v praxi použitelné, algoritmy).

Třída P (neboli PTIME)

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

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

To znamená, že pojem 'rychlý algoritmus' je ztotožňován s pojmem '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 však nebyla nalezena lepší charakterizace. V praxi se ukazuje, že když je pro nějaký problém nalezen polynomiální algoritmus, obvykle se podaří nalézt i algoritmus s nízkým stupněm polynomu (řekněme menším než 6).

Poznámka. Brzy se dostaneme k problémům, pro něž polynomiální algoritmy neexistují či nejsou známy. Klademe-li si (jen) otázku, zda daný problém patří do PTIME, pohybujeme se na na úrovni (podstatně) 'hrubší' analýzy než při pouhém zanedbávání konstant u značení O, Θ apod. Takto např. i metoda *bubblesort* prokazuje, že pro problém třídění existuje rychlý (rozuměj polynomiální) algoritmus (byť v detailnějším pohledu, tj. na jemnější úrovni analýzy, vnímáme *bubblesort* jako 'pomalý').

Uvědomme si, že třídy složitosti problémů, jak jsme je definovali v definici 4.3, i právě definovaná třída PTIME jsou závislé na našem zvoleném referenčním modelu – vztahují se tedy k RAMu (s logaritmickou mírou). Navrhneme-li jiný výpočetní model (do nějž budeme 'překládat' naše algoritmy), můžeme dostat jiné třídy složitosti !

Ovšem třída PTIME je (díky své 'hrubosti') *robustní*: PTIME je nezávislá na tom, zda zvolíme jako referenční model RAM nebo jiný 'rozumný' výpočetní model. Ukázalo se totiž, že všechny navržené rozumné modely počítače jsou 'polynomiálně ekvivalentní', tj. jsou schopny se vzájemně simulovat s 'pouze' polynomiální ztrátou; to znamená, že pro každé dva takové modely $\mathcal{M}_1$, $\mathcal{M}_2$ existuje konstanta c tak, že když problém P patří do $\mathcal{T}(f_1)$ pro referenční model $\mathcal{M}_1$, tak P patří do $\mathcal{T}(f_2)$, kde $f_2(n) = (f_1(n))^c$, pro referenční model $\mathcal{M}_2$. Všechny zmíněné rozumné modely tedy definují jednu a tutéž třídu PTIME.

Samozřejmě se naskýtá otázka, co to jsou *rozumné* výpočetní modely. Obecně řečeno se tím myslí ty, u nichž analýza algoritmů (v nich 'naprogramovaných') dává realistické výsledky pro praxi (alespoň na úrovni oné 'hrubé' analýzy diskutované výše). Technicky lze definovat jako rozumné ty modely, jež jsou polynomiálně ekvivalentní modelu RAM (myslí se samozřejmě s logaritmickou mírou, jak bylo dohodnuto v úmluvě za definicí 4.3). V literatuře se ovšem často v uvedené definici 'rozumnosti' odkazuje k historicky prvnímu modelu

počítače (resp. 'výpočtáře'), jenž známe pod názvem Turingův stroj a jenž si připomeneme níže.

Poznamenejme ještě, že uvažujeme *sekvenční modely*, k otázce *paralelních modelů* se letmo dostaneme později.

Turingovy stroje

Předpokládaným vstupem pro Turingův stroj je řetězec (vstupních) symbolů. Ten je rovnou uložen na (pracovní) pásce stroje (tj. oboustranně potenciálně nekonečné lineární pásce, rozdělené na buňky; každá buňka může obsahovat jeden symbol). Tímto vstupem je určena počáteční konfigurace stroje; tato konfigurace se mění krok za krokem podle předepsaných pravidel (daných přechodovou funkcí). Stroj má (na rozdíl od RAMu) sekvenční přístup k 'paměti' (v jednotlivém kroku má přístup jen k jedné buňce, 'posunout' se v jednom kroku může vždy jen na buňku sousední). Řetězec (neprázdných) symbolů na pásce po ukončení výpočtu (dosažení koncové konfigurace) je považován za výstup (příslušný původnímu vstupu).

Definice 5.1 Turingův stroj M *je určen následujícími parametry (formálně řečeno, je to uspořádaná šestice):* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$*, kde* Q *je konečná neprázdná množina* stavů, Γ *je konečná neprázdná množina* (páskových) symbolů, $\Sigma \subseteq \Gamma$, $\Sigma \neq \emptyset$ *je množina* vstupních symbolů, $q_0 \in Q$ *je* počáteční stav, $F \subseteq Q$ *je množina* koncových stavů, $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ *je přechodová funkce. Předpokládáme, že v* $\Gamma - \Sigma$ *je vždy obsažen speciální prvek* $\Box$ *označující* prázdný znak.

Konfigurací Turingova stroje M *rozumíme libovolné slovo tvaru* uqv*, kde* $u, v \in \Gamma^*$ *a* $q \in Q$*.* Konfigurace uqv *je* počáteční*, jestliže* u *je prázdné slovo,* $q = q_0$ *a* $v \in \Sigma^*$ *;* uqv *je* koncová konfigurace*, jestliže* $q \in F$*.*

Konfiguraci qv *ztotožňujeme s konfigurací* $\Box qv$*, podobně* uq *s* $uq\Box$*. (Obecně přidání lib. řetězce z* $\{\Box\}^*$ *vlevo či vpravo nehraje roli – vede k ekvivalentní konfiguraci.)*

Konfigurace $K = uaqbv$*, kde* $u, v \in \Gamma^*$*,* $a, b \in \Gamma$ vede v jednom kroku ke konfiguraci K'*, příslušnou relaci označujeme* $\vdash_M$ *nebo jen* $\vdash$ *(píšeme tedy* $K \vdash K'$*) právě když platí jedna z těchto možností:*

- $\delta(q, b) = (q', b', 0)$ *a* $K' = uaq'b'v$*,*
- $\delta(q, b) = (q', b', +1)$ *a* $K' = uab'q'v$*,*
- $\delta(q, b) = (q', b', -1)$ *a* $K' = uq'ab'v$*.*

Výpočet Turingova stroje nad zadaným vstupem se tedy zastaví v momentě dosažení koncového stavu (dojde-li k tomu vůbec). Výstupem (výpočtu) pak obvykle rozumíme řetězec z $(\Gamma - \{\Box\})^*$ zapsaný na pásce v příslušné koncové konfiguraci.

Turingovy stroje představují výpočetní model, který je možné brát jako určitou alternativu k strojům RAM. Již jsme se zmínili o tom, že ačkoliv RAM pracuje s čísly a Turingův stroj se znaky (symboly abecedy), jedná se v zásadě o totéž, jelikož čísla běžně zapisujeme řetězcem symbolů a naopak symboly abecedy jsou běžně kódovány čísly.

Poznámka. Alan Turing navrhl 'svůj' model již v třicátých letech 20. století, dříve než byly vyvinuty samočinné počítače. RAM byl navržen o několik desetiletí později jako realističtější model počítače.

Časovou a prostorovou složitost konkrétního Turingova stroje definujeme samozřejmě obdobně jako u RAMu:

Definice 5.2 Velikostí vstupu *Turingova stroje* M *rozumíme počet buněk pásky, které daný vstup zabírá (tedy délku vstupního řetězce).*
Délka výpočtu *Turingova stroje* M *pro konkrétní vstup se definuje jako počet provedení (elementárních) kroků, které* M *pro daný vstup vykoná, než se zastaví.*
Časovou složitostí *Turingova stroje* M *rozumíme funkci* $T_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $T_M(n)$ *znamená délku výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $T_M(n) = max\{\, k \mid k$ *je délka výpočtu* M *nad (nějakým) vstupem velikosti* $n \,\}$*.*

Definice 5.3 Velikostí paměti *Turingova stroje* M *(potřebné při výpočtu) pro konkrétní vstup rozumíme počet buněk pásky, které jsou během výpočtu nad daným vstupem navštíveny.*
Paměťovou složitostí *(nebo též* prostorovou složitostí*) Turingova stroje* M *rozumíme funkci* $S_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $S_M(n)$ *znamená velikost potřebné paměti při výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $S_M(n) = max\{\, k \mid k$ *je velikost paměti potřebné při výpočtu* M *nad (nějakým) vstupem velikosti* $n \,\}$*.*

Připomeňme, že definice dříve zavedených tříd složitosti $\mathcal{T}(f)$, $\mathcal{S}(f)$ závisí na zvoleném referenčním modelu, kterým jsou v našem případě stroje RAM. Vzali-li bychom jako referenční model Turingovy stroje, dostaneme jiné třídy !

Poznámka. Intuitivně snadno vidíme, že Turingův stroj je 'pomalejší' než RAM už z důvodů jeho sekvenčního přístupu k jednotlivým buňkám pásky (na rozdíl od 'libovolného', tj. přímého přístupu v případě RAMu). Existují tedy např. problémy, které lze řešit RAMem s čas. složitostí $O(n)$ (a které tedy patří do $\mathcal{T}(n)$ definované vzhledem k RAMu), ale nelze je řešit Turingovým strojem s čas. složitostí $O(n)$ (a nepatřily by tedy do $\mathcal{T}(n)$, vzali-li bychom Turingovy stroje jako referenční model).

Cílem následujících úvah bude ovšem ilustrace faktu, že

Turingovy stroje a RAMy jsou polynomiálně ekvivalentní.

Rozumíme tím, že

> Ke každému RAMu M existuje (dá se zkonstruovat) Turingův stroj M', který realizuje tutéž vstupně-výstupní funkci jako M (tj. řeší tentýž problém) a navíc pro příslušné časové a prostorové složitosti platí $T_{M'}(n) \leq (T_M(n))^{c_1}$, $S_{M'}(n) \leq (S_M(n))^{c_2}$ pro nějaké (malé) konstanty c_1, c_2.
> Totéž přitom platí i v opačném směru: ke každému Turingovu stroji M existuje RAM M' s příslušnými vlastnostmi.

Abychom to nahlédli, stačí si promyslet, že ke každému RAMu je možné (algoritmicky) zkonstruovat Turingův stroj, který jej *simuluje*; přitom dochází jen k 'malé ztrátě' z hlediska složitosti (odpovídající polynomu malého stupně) – zde znovu připomínáme úmluvu o uvažování logaritmické míry u RAMů. Naopak ke každému Turingovu stroji je možné (algoritmicky) zkonstruovat RAM, který jej *simuluje* s malou ztrátou z hlediska složitosti.

Uvedeným tvrzením ještě věnujeme několik odstavců; nicméně nepůjdeme do detailů – o nich předpokládáme, že by si je čtenář při svých programátorských zkušenostech snadno doplnil.

Především se zastavme u pojmu 'simulace'. Ten je jistě čtenáři intuitivně zřejmý. Podrobněji vysvětlit by se dal např. následovně.

Předpokládáme, že výpočet stroje nad zadaným vstupem lze chápat jako posloupnost konfigurací, kterými stroj (krok po kroku) prochází; začíná v počáteční konfiguraci určené zadaným vstupem a eventuálně skončí v jisté koncové konfiguraci s určitým výstupem – pokud jeho výpočet není nekonečný. $Con(M)$ nechť označuje množinu všech konfigurací stroje M.

Nyní lze vyjádření *'stroj M_1 je simulován strojem M_2'* vysvětlit takto:

> Existuje prostá funkce $cod : Con(M_1) \to Con(M_2)$ taková, že cod i cod^{-1} jsou (jednoduše) algoritmicky vyčíslitelné. Přitom pro libovolný výpočet $K_0, K_1, \ldots, K_m$ stroje M_1 prochází výpočet stroje M_2 začínající v $cod(K_0)$ postupně konfiguracemi $cod(K_0), cod(K_1), \ldots, cod(K_m)$ – s případnými 'mezikonfiguracemi'.

Např. lze snadno ukázat, jak lze (standardní) Turingův stroj simulovat Turingovým strojem s jen jednostranně nekonečnou páskou, jak lze vícepáskový Turingův stroj simulovat standardním Turingovým strojem, jak se lze omezit na případ, kde páskové symboly jsou pouze $0, 1, \Box$ apod.

Vzájemnou (polynomiální) simulací RAMů a Turingových strojů se budeme podrobněji věnovat na cvičení.

Přednáška 6

Jen stručně zmíníme ještě jeden používaný výpočetní model – *stroje s čítači* ('counter machines'; podobné jsou 'register machines').

Stroj s čítači C má fixní počet (celočíselných nezáporných) čítačů $c_1, c_2, \ldots, c_m$ a jeho 'program' je posloupnost příkazů

$$1 : COMM_1;\ 2 : COMM_2;\ \ldots\ldots;\ n : COMM_n$$

kde $COMM_n$ je instrukce $HALT$ a $COMM_i$ $(i = 1, 2, ..., n-1)$ jsou příkazy následujících dvou typů (předp. $1 \leq k, k_1, k_2 \leq n$, $1 \leq j \leq m$)

1/ $c_j := c_j + 1;\ goto\ k$,

2/ $if\ c_j = 0\ then\ goto\ k_1\ else\ (c_j := c_j - 1;\ goto\ k_2)$.

Lze si představit, že jeden čítač je vyčleněn jako vstupní (vstupem je tedy jedno celé nezáporné číslo) a jeden je vyčleněn jako výstupní. Víc asi není k popisu modelu třeba dodávat.

Takto definovaný model *není* polynomiálně ekvivalentní Turingovým strojům; problém je v tom, že aritmetické operace s čísly (obsahy čítačů), jsou úměrné velikosti těchto čísel, nikoli velikosti jejich zápisu (což je, jak víme, 'exponenciální rozdíl'). Stačí ovšem přidat např. instrukce typu $c_j := c_j$ div 10 a $c_j := c_j * 10$; tyto modifikované stroje s čítači pak už *jsou* polynomiálně ekvivalentní Turingovým strojům.

6 Třída NPTIME. Polynomiální převeditelnost. NP-úplné problémy.

Stručný obsah: Příklady (běžných praktických) problémů, pro které jsou známy pouze exponenciální algoritmy. Nedeterministické Turingovy stroje. Třída NP (=NPTIME); její robustnost vůči výpočetním modelům. Polynomiální převeditelnost problémů. NP-úplnost. NP-úplné problémy. Otevřená otázka, zda P=NP.

Studijní cíle: Umět vysvětlit pojem NP-úplného problému a související pojmy (např. nedeterm. Turingův stroj) a dokazovat příslušnost k NP a NP-obtížnost u konkrétních problémů.

Čtenář jistě zná spoustu algoritmů (některé jsme uvedli i v rámci tohoto kursu), které patří do třídy PTIME. Připomeňme, že tato je chápána jako (horní) aproximace třídy prakticky zvládnutelných problémů.

Nyní uvedeme příklady několika problémů, pro které jsou známy exponenciální algoritmy, ale není známo, zda patří či nepatří do PTIME, neboli zda pro ně existují či neexistují polynomiální algoritmy.

Mezi takové problémy dlouho patřil dříve uvedený *problém prvočíselnosti* (v létě 2002 byl zveřejněn důkaz příslušnosti k PTIME), nadále tam ovšem patří problém SAT (splnitelnost booleovských formulí) a následující problémy:

Název: TSP (*problém obchodního cestujícího*) (ANO/NE verze)

Vstup: množina 'měst' $\{1, 2, \ldots, n\}$, přir. čísla ('vzdálenosti') d_{ij} $(i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n)$; dále číslo ℓ ('limit').
Otázka: existuje 'okružní jízda' dlouhá nejvýše ℓ, tj. existuje permutace $\{i_1, i_2, \ldots, i_n\}$ množiny $\{1, 2, \ldots, n\}$ tž. $d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1) \leq \ell$?

Název: HC (*problém hamiltonovského cyklu*)

Vstup: orientovaný graf G.
Otázka: existuje v G hamiltonovský cyklus (tj. uzavřená orientovaná cesta, procházející každým vrcholem právě jednou) ?

Název: HK (*problém hamiltonovské kružnice*)

Vstup: neorientovaný graf G.
Otázka: existuje v G hamiltonovská kružnice (tj. uzavřená cesta, procházející každým vrcholem právě jednou) ?

Název: IS (*problém nezávislé množiny*)

Vstup: neorientovaný graf G (o n vrcholech); číslo k $(k \leq n)$.
Otázka: existuje v G nezávislá množina velikosti k (tj. množina k vrcholů, z nichž žádné dva nejsou spojeny hranou) ?

Uvedené problémy jsou speciálního charakteru: jsou rozhodnutelné v polynomiálním čase *nedeterministickými* Turingovými stroji. (Jen stručně připomeňme, že u nedeterministického Turingova stroje může být v dané konfiguraci obecně možné provést několik vzájemně různých kroků – bezprostředně následující konfigurace tak není určena jednoznačně.)

Definice 6.1 *Daný problém P (typu ANO/NE) je rozhodován nedeterministickým Turingovým strojem M, jestliže všechny výpočty M jsou konečné a vydávají ANO nebo NE, a navíc platí:*

- *jestliže odpověď na otázku problému P pro vstup w je ANO, pak existuje (alespoň jeden) výpočet M nad w vydávající ANO,*
- *jestliže odpověď pro w je NE, pak všechny výpočty M nad w vydávají NE.*

Složitost nedeterministického Turingova stroje a příslušné třídy složitosti lze definovat takto:

Definice 6.2 Časová složitost *nedeterministického Turingova stroje M je zobrazení $T_M : \mathcal{N} \to \mathcal{N}$, kde $T_M(n)$ znamená maximální délku výpočtu pro vstup velikosti n.*
Třídou časové složitosti $N\mathcal{T}(f)$ *pro funkci $f : \mathcal{N} \longrightarrow \mathcal{N}$ rozumíme třídu těch problémů, které jsou řešeny nedeterministickými Turingovými stroji s čas. složitostí v $O(f)$.*

Čtenář si jistě snadno doplní definice pro prostorovou složitost.

Konzistentnější s předchozím textem by bylo, kdybychom při definování tříd $N\mathcal{T}(f(n))$ použili jako referenční model nedeterministické RAMy. Nám ovšem půjde především o třídu

$$\text{NPTIME} = \bigcup_{k=0}^{\infty} N\mathcal{T}(n^k)$$

tzv. problémů řešitelných v nedeterministickém polynomiálním čase. Její definice je podobně jako pro PTIME robustní (nezávislá na zvoleném 'rozumném' referenčním modelu).

Takto jsme se dostali k velmi známé dosud otevřené otázce, zda PTIME=NPTIME (dané otázce se často říká P-NP problém); přitom je ovšem zřejmé, že PTIME⊆NPTIME.

Přednáška 7

Uvedeme teď definici charakterizující 'nejtěžší' problémy v NPTIME. Pro tyto účely využijeme následující relaci $\triangleleft$ mezi problémy (v našem kontextu lze $P1 \triangleleft P2$ číst '$P1$ je nejvýš tak těžký jako $P2$'):

Definice 6.3 *Problém $P1$ je* polynomiálně převeditelný *na problém $P2$, označme $P1 \triangleleft P2$, jestliže existuje Tur. stroj M* s polynomiální časovou složitostí, *který pro libovolný vstup w problému $P1$ sestrojí vstup w' problému $P2$, přičemž platí, že odpověď na otázku problému $P1$ pro vstup w je stejná jako odpověď na otázku problému $P2$ pro vstup w'.*

Všimněme si následujících jednoduchých faktů:

- $((P1 \triangleleft P2) \wedge (P2 \triangleleft P3)) \Longrightarrow (P1 \triangleleft P3)$
- $((P1 \triangleleft P2) \wedge (P2 \in \text{PTIME})) \Longrightarrow (P1 \in \text{PTIME})$
- $((P1 \triangleleft P2) \wedge (P2 \in \text{NPTIME})) \Longrightarrow (P1 \in \text{NPTIME})$

Definice 6.4 *O problému P řekneme, že je* NP-těžký, *jestliže pro každý problém $P' \in NPTIME$ platí $P' \triangleleft P$. Je-li navíc $P \in NPTIME$, říkáme, že P je* NP-úplný.

Definovali jsme pojem NP-úplných problémů (tj. 'nejtěžších' problémů v NPTIME), ale dosud jsme neukázali, že aspoň jeden takový problém vůbec existuje. Tuto existenci ukazuje následující věta.

Věta 6.5 *(Cook, 1971) Problém SAT je NP-úplný.*

Důkaz: Problém příslušnosti SAT k NPTIME je zřejmý (příslušný nedeterministický Turingův stroj prostě 'zvolí' nějaké pravdivostní ohodnocení proměnných v zadané formuli

a ověří, zda při tomto ohodnocení je formule pravdivá; ke kladnému závěru má možnost dospět právě tehdy, když takové ohodnocení existuje).

Stačí tedy ukázat, že pro každý $P \in$ NPTIME platí $P \triangleleft$SAT (připomeňme, že se omezujeme na ANO/NE problémy).

Uvažujme tedy libovolný, ale dále pevný, problém $P \in$ NPTIME. Ten je nutně rozhodován nedeterministickým Turingovým strojem M s časovou složitostí $T_M(n) \leq p(n)$ pro určitý polynom p. Je potřeba ukázat, že existuje polynomiální algoritmus (přesněji řečeno Turingův stroj s časovou složitostí omezenou polynomiální funkcí), který k libovolnému vstupu w problému P zkonstruuje booleovskou formuli $\mathcal{F}_w$ (v konjunktivní normální formě), která je splnitelná právě tehdy, když odpověď na otázku P pro w je ANO.

Máme-li ovšem dán vstup w velikosti n, pak k rozhodnutí o odpovědi na otázku P pro w stačí zjistit, zda existuje posloupnost konfigurací $C_0, C_1, C_2, \ldots, C_{p(n)}$, která představuje možný přijímající (tj. končící odpovědí ANO) výpočet stroje M na vstupu w; všimněme si, že velikost konfigurací je rovněž nutně omezena hodnotou $p(n)$.

Není těžké (i když je to technicky pracné) zkonstruovat formuli zachycující schéma takového výpočtu, která je splnitelná právě tehdy, když příslušná posloupnost konfigurací existuje. (Podrobněji bude konstrukce probrána na cvičení.)

□

Máme-li dokázánu NP-úplnost jednoho problému, je možné ji využít k důkazu NP-úplnosti (či NP-obtížnosti) problémů dalších:

Tvrzení 6.6 *Jestliže $P_1 \triangleleft P_2$ a P_1 je NP-těžký, pak P_2 je rovněž NP-těžký; když je navíc P_2 v NPTIME, je NP-úplný.*

Důkaz: Tvrzení plyne snadno z faktu, že složení dvou polynomiálních funkcí je opět polynomiální funkce—byť vyššího stupně; jinými slovy: relace $\triangleleft$ je tranzitivní.

□

Demonstrováním příslušných převeditelností ukážeme na přednášce NP-úplnost několika již uvedených a dalších problémů. Např. ukážeme:

- SAT $\triangleleft$ IS
- IS $\triangleleft$ HC (referováno na cvičení)
- HC $\triangleleft$ HK
- HK $\triangleleft$ TSP
- SAT $\triangleleft$ 3-SAT
- 3-SAT $\triangleleft$ 3-CG

kde dosud nezmíněné problémy jsou definovány takto:

> *Název*: 3-SAT (*problém SAT s omezením na 3 literály*)
>
> *Vstup*: booleovská formule v konjunktivní normální formě, kde v každé klauzuli (tj. v každém konjunktu) jsou právě 3 literály (literál je buď proměnná nebo její negace).
> *Otázka*: je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá) ?

> *Název*: 3-CG (*problém barvení grafu třemi barvami*)
>
> *Vstup*: neorientovaný graf $G = (V, E)$.
> *Otázka*: lze G obarvit třemi barvami, tzn. existuje zobrazení $c : V \to \{col_1, col_2, col_3\}$ takové, že $\forall \{v_1, v_2\} \in E : c(v_1) \neq c(v_2)$?

Přednáška 8

Uvažujme ještě problém

> *Název*: ILP (*problém celočíselného lineárního programování*)
>
> *Vstup*: matice A typu $m \times n$ a sloupcový vektor b velikosti m, jejichž prvky jsou celá čísla.
> *Otázka*: existuje celočíselný sloupcový vektor x (velikosti n) tž. $Ax \geq b$?

Jedná se rovněž o NP-úplný problém. Snadno se ukáže, že je NP-těžký (např. převodem 3-SAT $\triangleleft$ ILP), ale na rozdíl od dříve uvedených problémů je obtížnější prokázat, že ILP $\in$ NPTIME. Zhruba řečeno, dá se ukázat, že pokud řešení nerovnosti $Ax \geq b$ existuje, existuje i řešení 'dostatečně malé'—jeho zápis je polynomiální vzhledem k zápisu A a b ; řešení se tedy dá v polynomiálním čase 'uhodnout' a ověřit.

Otázka P$\stackrel{?}{=}$NP

Když se hovoří o tzv. P-NP problému (slovo 'problém' zde není použito v našem technickém smyslu !), rozumí se tím otevřená otázka, zda PTIME je vlastní podtřídou NPTIME či zda jsou si tyto třídy rovny.

Obecně se má zato, že pro NP-úplné problémy neexistují polynomiální algoritmy; ovšem nikdo to zatím nedokázal. Všimněme si, že kdyby někdo objevil polynomiální algoritmus pro jeden NP-úplný problém, existovaly by polynomiální algoritmy pro všechny tyto problémy. Naopak když by někdo prokázal, že pro jeden konkrétní NP-úplný problém neexistuje polynomiální algoritmus, neexistoval by takový algoritmus pro žádný z NP-úplných problémů.

V praxi se tedy bere prokázání NP-úplnosti (či vlastně NP-obtížnosti) jako důkaz nezvládnutelnosti problému—trváme-li na zaručeném nalezení (nejlepšího možného) řešení. V úvahu pak přicházejí např. *aproximační algoritmy* (u optimalizační úlohy se např. může podařit sestavit rychlý algoritmus, který zaručeně nalezne řešení, jež je nejvýše dvakrát horší než optimální) či *pravděpodobnostní algoritmy* (využívají 'házení kostkou' a dávají rychle odpovědi, které však mohou být s určitou pravděpodobností nesprávné; jejich vícenásobným opakováním se ale dá docílit, že pravděpodobnost nesprávného výsledku je mizivá) – příklady takových algoritmů uvedeme v závěru kursu.

7 Další třídy časové i prostorové složitosti

Stručný obsah: Třídy PSPACE, EXPTIME, EXPSPACE. Idea rovnosti PSPACE=NPSPACE (Savitchova věta). Příklady PSPACE-úplných problémů. Dokazatelně nezvládnutelné problémy – příklady problémů s exponenciální či vyšší složitostí.

Studijní cíle: Umět vysvětlit probrané třídy složitosti, ilustrovat je příklady, vysvětlit pojem dokazatelně nezvládnutelných problémů.

Třída PSPACE

Předmětem našeho prvořadého zájmu je časová složitost algoritmů a problémů. Už v případě konkrétních algoritmů a problémů ovšem může mít dobrý smysl zkoumat také prostorovou (tj. paměťovou) složitost a speciálně vztah časové a prostorové složitosti (daný algoritmus může jít např. zrychlit jen za cenu zvýšení paměťové náročnosti a naopak).

Strukturální složitost samozřejmě zkoumá i třídy problémů vymezené prostorovou složitostí. Čtenář si jistě snadno doplní definice tříd PSPACE a NPSPACE a všimne si zřejmé inkluze PSPACE $\subseteq$ NPSPACE. Pro tyto třídy se ovšem ví, že platí i inkluze obrácená (a je tedy PSPACE = NPSPACE); to ihned plyne z následující věty (jen poznamenejme, že věta platí obecněji—pro naše účely však postačuje uvedené znění):

Věta 7.1 *(Savitch, 1970) Je-li problém P rozhodován nedeterministickým Turingovým strojem s prostorovou složitostí $O(n^k)$, pak je také rozhodován deterministickým Turingovým strojem s prostorovou složitostí $O(n^{2k})$.*

Důkaz: Nechť problém P je rozhodován nedeterministickým Turingovým strojem M_1 s prostorovou složitostí nejvýše $c_1 n^k$ (pro nějaké konstanty c_1, k). Všimněme si, že pro vstup w velikosti n může stroj M_1 vydat odpověď ANO právě tehdy, když existuje posloupnost konfigurací $C_0, C_1, C_2, \ldots, C_m$ popisující příslušný výpočet; velikost každé konfigurace je nejvýše rovna $c_1 n^k$. Takových konfigurací je ovšem nejvýše $c^{c_1 n^k}$ pro vhodně zvolené c (jako c lze zvolit součet počtu symbolů abecedy a počtu stavů stroje M_1). Jelikož je zbytečné,

aby se v uvedené posloupnosti konfigurací nějaká konfigurace opakovala, stačí uvažovat jen $m \leq c^{c_1 n^k}$.

Mělo by teď být jasné, jak lze M_1 simulovat deterministickým Turingovým strojem M_2 s prostorem $c_1 n^k \cdot c^{c_1 n^k}$ (ten prostě zkouší systematicky všechny možnosti a pro každou z nich zjišťuje, zda se jedná o zápis hledané posloupnosti $C_0, C_1, C_2, \ldots, C_m$).

Toto ovšem nestačí, neboť prostor používaný strojem M_2 je exponenciální. Základní idea ušetření prostoru spočívá v tom, že úkol ověření, zda z konfigurace C lze dosáhnout konfiguraci C' za 2^ℓ kroků se dá řešit systematickým generováním ('prostředních') konfigurací C'' a ověřováním, zda z C lze dosáhnout C'' za $2^{\ell-1}$ kroků a z C'' lze dosáhnout C' za $2^{\ell-1}$ kroků. K ověření zmíněných dvou podúkolů je ovšem možné použít tentýž prostor ! Uplatníme-li tuto myšlenku rekurzivně, není těžké vyvodit, že celkový potřebný prostor bude u upraveného M_2 nejvýše $c_1 n^k \cdot c_2 n^k$ pro vhodnou konstantu c_2 a tedy prostorová složitost M_2 je pak $O(n^{2k})$.

Detailněji bude důkaz probrán na cvičení.

□

Připomeňme, že pro lib. funkci f je $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$ (např. Turingův stroj očividně navštíví při výpočtu nejvýše tolik políček, kolik udělá kroků). Měl by teď už být zřejmý vztah

$$PTIME \subseteq NPTIME \subseteq PSPACE = NPSPACE.$$

Přes velké úsilí vědecké komunity, nemůžeme dosud vyloučit nejen možnost $PTIME = NPTIME$, ale dokonce ani $PTIME = PSPACE$, byť se tyto možnosti jeví velmi 'nepravděpodobnými'.

Podobně jako u NP-úplnosti, lze definovat tzv. PSPACE-úplné problémy; definici napíšeme obecněji:

Definice 7.2 *O problému P řekneme, že je $\mathcal{C}$-těžký, kde $\mathcal{C}$ je nějaká třída problémů, jestliže pro každý $P' \in \mathcal{C}$ platí $P' \triangleleft P$. Je-li navíc $P \in \mathcal{C}$, říkáme, že P je $\mathcal{C}$-úplný.*

Známým příkladem PSPACE-úplného problému je problém

> *Název*: QBF (*problém pravdivosti kvantifikovaných booleovských formulí*)
> *Vstup*: formule $(\exists x_1)(\forall x_2)(\exists x_3)(\forall x_4)\ldots(\exists x_{2n-1})(\forall x_{2n})\mathcal{F}(x_1, x_2, \ldots, x_{2n})$, kde $\mathcal{F}(x_1, x_2, \ldots, x_{2n})$ je booleovská formule v konjunktivní normální formě.
> *Otázka*: je daná formule pravdivá ?

Převeditelností z tohoto problému lze např. ukázat PSPACE-obtížnost různých deskových her.

Dokazatelně nezvládnutelné problémy

Jestliže prokážeme NP-obtížnost či PSPACE-obtížnost nějakého problému, říkáme, že je *prakticky nezvládnutelný* (intractable). Je totiž jasné, že navrhneme-li algoritmus, který řeší (přesně ten) daný problém, a neobjevíme-li přitom geniální 'trik', na který dosud nikdo nepřišel, bude mít algoritmus zřejmě exponenciální (či ještě horší) složitost. Obecně používat algoritmus (resp. odpovídající program) budeme moci jen na velmi malá vstupní data. Teoreticky ovšem pořád ještě možnost rychlého algoritmu (založeného na 'geniálním triku') existuje.

Poznámka. Samozřejmě je možné, že exponenciální algoritmus ve skutečnosti chceme použít jen na malá data, anebo ho třeba používáme na data, při nichž se neprojeví ona (worst case) exponenciální složitost. V tomto smyslu praktická nezvládnutelnost (obecného) problému ještě neznamená, že ho v praxi nemůže počítačový program úspěšně řešit v pro nás zajímavých případech. (Existují i jiné možnosti, jak v praxi úspěšně řešit i 'nezvládnutelný' problém. Zmíníme se o tom v partiích o aproximačních a pravděpodobnostních algoritmech.)

Známe ovšem i tzv. *dokazatelně nezvládnutelné* problémy (provably intractable problems), tj. ty, u nichž máme *dokázáno*, že pro ně neexistují polynomiální algoritmy. Definujeme-li např. třídy

$$\text{EXPTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(2^{n^k}) , \qquad \text{EXPSPACE} = \bigcup_{k=0}^{\infty} \mathcal{S}(2^{n^k})$$

pak EXPTIME-těžký či EXPSPACE-těžký problém je takovým dokazatelně nezvládnutelným problémem.

Dá se totiž ukázat, že inkluze PTIME⊂EXPTIME, PSPACE⊂EXPSPACE jsou skutečně vlastní (tzn., že neplatí rovnost). (My to zde ale dokazovat nebudeme.)

Např. následující problém je EXPSPACE-úplný:

> *Název*: RE^2 (*ekvivalence regulárních výrazů s mocněním*)
>
> *Vstup*: dva regulární výrazy, v nichž je možné použít mocnění (tzn. je možno psát α^2 místo $\alpha \cdot \alpha$).
> *Otázka*: reprezentují zadané výrazy tentýž jazyk ?

Poznamenejme, že stejný problém pro standardní regulární výrazy (bez mocnění) je PSPACE-úplný. (Připomeňme si, že složitost je funkcí velikosti vstupu; mocnění v reg. výrazech umožňuje exponenciálně zkrátit některé výrazy bez mocnění.) Čtenář by si měl být schopen vyvodit, že problém ekvivalence dvou *nedeterministických* konečných automatů je tedy také PSPACE-úplný. (Pro deterministické konečné automaty ovšem samozřejmě známe polynomiální algoritmus; připomeňme si, že přechodem od nedeterministického automatu k deterministickému se obecně nemůžeme vyhnout exponenciálnímu nárustu počtu stavů.)

Přednáška 9

Existují samozřejmě i dokazatelně těžší (superexponenciální) problémy. Ilustrujme je příkladem problému Presburgerovy aritmetiky (rozhodování pravdivosti formulí teorie sčítání).

> *Název*: ThAdd (*problém pravdivosti teorie sčítání*)
>
> *Vstup*: formule jazyka 1. řádu užívající jediný 'nelogický' symbol – ternární (tj. 3-ární) predikátový symbol $PLUS$ ($PLUS(x,y,z) \Leftrightarrow_{df} x+y=z$).
> *Otázka*: je daná formule pravdivá pro množinu $\mathcal{N} = \{0,1,2,\ldots\}$, kde $PLUS(a,b,c)$ je interpretováno jako $a+b=c$?

Zde vůbec není zřejmé, že existuje algoritmus, který daný problém řeší. Presburger ukázal takový algoritmus ve 20. létech 20. století (jedním z cílů tzv. Hilbertova programu bylo ukázat podobný algoritmus i s připuštěním predikátu pro násobení—nemožnost řešení tohoto úkolu ukázal později Gödel). Mnohem později bylo ukázáno, že každý algoritmus, řešící problém ThAdd má složitost minimálně 2^{2^n}.

Presburger ukázal algoritmus využitím tzv. metody eliminace kvantifikátorů. Výsledky teorie konečných automatů umožňují podat důkaz následující věty velice elegantně:

Věta 7.3 *Existuje algoritmus rozhodující problém ThAdd.*

Důkaz: (Idea.) Představme si třístopou pásku, kde v každé stopě je řetězec nul a jedniček, tj. binární zápis čísla. Na pásku samozřejmě můžeme hledět jako na jednostopou s tím, že povolené *symboly abecedy* jsou uspořádané *trojice* nul a jedniček. Snadno nahlédneme, že existuje konečný automat, který přijímá právě ta slova v 'abecedě trojic', která mají tu vlastnost, že součet čísla v první stopě s číslem v druhé stopě je roven číslu ve třetí stopě. Ihned je to jasné při čtení odzadu; pak si stačí připomenout, že regulární jazyky jsou uzavřeny na zrcadlový obraz.

Z dalších uzávěrových vlastností snadno vyvodíme, že pro lib. formuli $\mathcal{F}(x_1, x_2, \ldots, x_n)$ jazyka ThAdd která *neobsahuje kvantifikátory*, lze zkonstruovat kon. automat $A_{\mathcal{F}}$ přijímající právě binární zápisy těch n-tic čísel (na n-stopé pásce), pro které je $\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá.

Pro formuli $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je možné zkonstruovat automat, který přijímá právě binární zápisy těch $(n-1)$-tic čísel (dosazených za $x_1, x_2, \ldots, x_{n-1}$), pro které je $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá: automat pracuje na $(n-1)$-stopé pásce, ale simuluje $A_{\mathcal{F}}$ tak, že obsah n-té stopy nedeterministicky hádá ! (Pak ho samozřejmě lze převést na ekvivalentní deterministický automat.)

Jelikož $(\forall x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je ekvivalentní $\neg(\exists x_n)\neg\mathcal{F}(x_1, x_2, \ldots, x_n)$, načrtli jsme takto postup, který k formuli jazyka ThAdd v prenexní formě postupnou aplikací zmíněných

konstrukcí sestrojí konečný automat, který přijme prázdné slovo právě tehdy, když výchozí formule je pravdivá.

□

8 Rozhodnutelnost a nerozhodnutelnost problémů

Stručný obsah: (Algoritmická) rozhodnutelnost, nerozhodnutelnost a částečná rozhodnutelnost problémů. Church-Turingova teze (algoritmus = Turingův stroj). Rekurzivní a rekurzivně spočetné množiny (jazyky). Rekurzivní a částečně rekurzivní funkce. Idea univerzálního algoritmu (Turingova stroje).

Studijní cíle: Umět vysvětlit pojem rozhodnutelného, nerozhodnutelného a částečně rozhodnutelného problému a objasnit roli Church-Turingovy teze. Vše umět ilustrovat na příkladech problémů. Umět vysvětlit konstrukci univerzálního algoritmu.

Pro námi dosud uvažované problémy vždy existoval algoritmus, který příslušný problém řeší (rozhoduje). Chceme-li pojem algoritmické řešitelnosti (či rozhodnutelnosti) problémů uvést přesněji, lze např. podat tuto definici:

Definice 8.1 *Problém $P = (IN, OUT, p)$ ($p : IN \to OUT$) je* algoritmicky řešitelný*, jestliže existuje algoritmus, který pro libovolný vstup $w \in IN$ skončí a vydá jako výsledek $p(w)$. Jedná-li se o problém typu ANO/NE, říkáme, že je* algoritmicky rozhodnutelný*, nebo stručněji* rozhodnutelný*.*

Všimněme si, že se implicitně předpokládá, že vstupy a výstupy jsou 'finitní objekty' (či jsou takto kódovány); už jsme hovořili o tom, že stačí uvažovat kódování vstupů a výstupů řetězci symbolů z nějaké konečné abecedy.

Pojem algoritmické rozhodnutelnosti či algoritmické vyčíslitelnosti (řešitelnosti) lze přirozeně definovat např. pro množiny přir. čísel, jazyky, či funkce:

Definice 8.2 *Množina $\mathcal{M} \subseteq \mathcal{N}$ je* rozhodnutelná*, jestliže problém příslušnosti k $\mathcal{M}$ (Vstup: $n \in \mathcal{N}$; otázka: platí $n \in \mathcal{M}$?) je rozhodnutelný.*
Jazyk L v abecedě Σ (tedy $L \subseteq \Sigma^*$) je rozhodnutelný*, jestliže problém příslušnosti k L (Vstup: $w \in \Sigma^*$; otázka: platí $w \in L$?) je rozhodnutelný.*
Funkce $f : \mathcal{N} \to \mathcal{N}$ je algoritmicky *(někdy též* efektivně*)* vyčíslitelná*, jestliže 'problém výpočtu jejích hodnot' (Vstup: $n \in \mathcal{N}$; výstup $f(n)$) je algoritmicky řešitelný.*

Pojmy definované v předchozích definicích se odvolávaly k pojmu 'algoritmus'. Nahradíme-li v nich pojem 'algoritmus' pojmem 'Turingův stroj', uvádíme u definovaných pojmů výraz 'rekurzivní':

Definice 8.3 *Problém typu ANO/NE je* rekurzivní*, jestliže je rozhodován Turingovým strojem. Podobně zavádíme pojmy* rekurzivní jazyk*,* rekurzivní množina*,* rekurzivní funkce*.*

Poznámka. Pojem 'rekurzivní' je v této souvislosti ustálen z historických důvodů, které zde nebudeme rozebírat. Jen poznamenejme, že např. pojem 'rekurzivní funkce' nelze ztotožňovat s týmž pojmem v programovacích jazycích.

Jak ale čtenář zřejmě očekává, máme velmi dobré důvody přijímat následující tezi (axiom):

Church-Turingova 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 tytéž vstupy budou příslušné výstupy totožné.*

Důsledek 8.4 *Při přijetí Church-Turingovy teze lze ztotožňovat pojmy* rekurzivní *a* rozhodnutelný *(v případě (totální) funkce pojmy* rekurzivní *a* algoritmicky vyčíslitelná*).*

Jelikož pojem 'algoritmus' bereme jako pojem základní (podobně jako např. pojem 'množina') a nikoli odvozený (tj. definovaný pomocí základních a z nich odvozených pojmů), nelze Church-Turingovu tezi dokázat jako matematickou větu. Všimněte si ovšem, že v obráceném směru je platnost teze zřejmá. Tyto úvahy jsou už spíše logicko-filozofické povahy a zde je nebudeme dále rozebírat.

Pro nás je zde podstatné, že Church-Turingova teze, stručně řečeno, ztotožňuje pojmy algoritmus a Turingův stroj. Přitom (zatím) naše zkušenost a mnohé výsledky teorie vyčíslitelnosti potvrzují, že přijímání teto teze je rozumné.

Brzy ukážeme důkaz algoritmické nerozhodnutelnosti určitého problému (problému zastavení). Ve skutečnosti ovšem dokážeme, že tento problém není turingovsky rozhodnutelný, tj. není rekurzivní. Že je v tom případě (algoritmicky) nerozhodnutelný, vyplývá z Church-Turingovy teze; to se v takové souvislosti většinou explicitně neuvádí, ale měli bychom to mít na paměti.

Nejprve ale uvedeme definici širší třídy než je třída rozhodnutelných problémů:

Definice 8.5 *Problém typu ANO/NE je* částečně rozhodnutelný*, jestliže existuje algoritmus, který skončí právě pro ty vstupy problému, na něž je odpověď ANO. (Pro vstupy s odpovědí NE je běh algoritmu nekonečný).*
Podobně definujeme pojmy částečně rozhodnutelný jazyk*,* částečně rozhodnutelná množina*.*

Je zřejmé, že každý rozhodnutelný problém je i částečně rozhodnutelný (proč ?). Za chvíli uvidíme, že naopak to neplatí.

Ještě uvedeme analogický pojem pro funkce:

Definice 8.6 *Částečná funkce $\phi : \mathcal{N} \to \mathcal{N}$ ($Dom(\phi) \subseteq \mathcal{N}$) je* algoritmicky *(někdy též* efektivně*)* vyčíslitelná*, jestliže existuje algoritmus, jenž přijme jako vstup libovolné $n \in \mathcal{N}$, zastaví se právě tehdy, když $n \in Dom(\phi)$, a v tom případě vydá $\phi(n)$.*

Nahradíme-li opět pojem 'algoritmus' pojmem 'Turingův stroj', dostaneme definice *rekurzivně spočetného jazyka*, *rekurzivně spočetné množiny*, *částečně rekurzivní funkce*. Za předpokladu Church-Turingovy teze můžeme ovšem opět provést příslušná ztotožnění pojmů.

Určitý vztah mezi rozhodnutelností a částečnou rozhodnutelností uvádí následující věta (zformulujte si ji i pro ANO/NE problémy):

Věta 8.7 *(Post) Množina $A \subseteq \Sigma^*$ je rozhodnutelná právě když A i $\overline{A}$ jsou částečně rozhodnutelné.*

Důkaz: Důkaz je vcelku přímočarý (promyslete jej); hlavní myšlenka spočívá v tom, že k dvěma algoritmům (Turingovým strojům) lze zkonstruovat algoritmus (Turingův stroj), který je provádí 'paralelně', tj. 'střídavě sekvenčně'.

□

Univerzální algoritmus

Uvažujme na chvíli pračku. Je to vlastně zařízení, které realizuje určitý algoritmus. Podobně třeba automobil atd. A co počítač ? Ten také realizuje určitý algoritmus. Hlavní činnost ('proceduru') tohoto algoritmu lze ovšem nazvat *Proveď zadaný algoritmus (neboli program)* pro zadaný vstup (neboli data). Proto ten algoritmus realizovaný počítačem je vlastně

univerzální algoritmus

(schopný realizovat jakýkoli zadaný algoritmus). Formálně existenci takového univerzálního algoritmu potvrzuje následující věta. (Důkaz, tj. konstrukci univ. stroje, podrobně probereme na cvičení.)

> Značení $!M(w)$ znamená "stroj M se na vstup w zastaví" (tzn. jeho výpočet skončí, je-li v počáteční konfiguraci na pásce slovo slovo w).

Věta 8.8 *(O* univerzálním Turingovu stroji*.) Lze sestrojit Turingův stroj U takový, že pro libovolný Turingův stroj M s abecedou $\{0,1,\Box\}$ a libovolné $w \in \{0,1\}^*$ platí:*
1/ $!M(w) \Leftrightarrow !U(Kod(M) \cdot w)$ *a*
2/ jestliže $!M(w)$*, pak* $M(w) = U(Kod(M) \cdot w)$.

Přednáška 10

9 Nerozhodnutelné problémy

Stručný obsah: Metoda diagonalizace a důkaz nerozhodnutelnosti problému zastavení (otázky, zda $!M(w)$). (Algoritmická) převeditelnost problémů; využití pro důkaz (ne)rozhodnutelnosti. Postův korespondenční problém a aplikace na problémy z teorie (bezkontextových) jazyků. Problémy z logiky. Riceova věta o nerozhodnutelnosti netriviálních vstupně-výstupních vlastností algoritmů (programů).

Studijní cíle: Zvládnout prokazování nerozhodnutelnosti (a případně částečné rozhodnutelnosti) problémů v probraných a podobných příkladech. Umět aplikovat Riceovu větu.

Vzpomeňme si, že u NP-úplných problémů jsme nejprve dokázali NP-úplnost jednoho (konkrétně problému SAT) a pak jsme pomocí polynomiální převeditelnosti prokazovali NP-úplnost (resp. NP-obtížnost) dalších problémů.

Podobná situace je u nerozhodnutelných problémů. Když prokážeme nerozhodnutelnost jednoho, k prokázání nerozhodnutelnosti dalších už (většinou) stačí vhodně aplikovat tzv. algoritmickou převeditelnost.

Roli prvního (základního) nerozhodnutelného problému hraje problém zastavení:

> *Název*: HP (*Halting problem*)
>
> *Vstup*: Turingův stroj M (resp. jeho kód $Kod(M)$) s abecedou $\{0,1,\Box\}$ a slovo $w \in \{0,1\}^*$.
> *Otázka*: zastaví se M na w (tj. platí $!M(w)$) ?

Věta 9.1 *Problém HP je nerozhodnutelný.*

Důkaz: Připomeňme nejprve, že Turingovy stroje lze přímočaře kódovat řetězci nul a jedniček, tedy $Kod(M) \in \{0,1\}^*$.

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 nějaký Tur. stroj 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 samozřejmě možné také předpokládat pouze abecedu $\{0,1,\Box\}$.

Sestrojme nyní stroj D s abecedou $\{0,1,\Box\}$, 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 D přejde do nekonečného cyklu (a tedy se nezastaví); jestliže H skončí ve stavu q_{NE}, stroj D se zastaví (stav q_{NE} bude také jeho koncovým stavem).

Když ovšem nyní prozkoumáme, zda se D při spuštění na svůj vlastní kód $Kod(D)$ zastaví či nezastaví, dospějeme při obou možnostech k logickému sporu (zastaví se a nezastaví se současně).

□

Další věta plyne snadno z předchozí věty, věty o univerzálním Tur. stroji a Postovy věty:

Věta 9.2 *Problém HP je částečně rozhodnutelný, jeho doplňkový problém není (ani) částečně rozhodnutelný.*

Všimněme si, že v důkazu nerozhodnutelnosti problému zastavení (HP) jsme vlastně dokázali nerozhodnutelnost speciálního podproblému, který označíme

DHP (Diagonal Halting Problem)

Vstup: Stroj M daný svým kódem $Kod(M)$;

Otázka: Zastaví se M na svůj kód (tj. na slovo $Kod(M)$) ?

Poznámka. Metoda důkazu je v principu aplikací tzv. Cantorovy diagonalizační metody, jež má v teorii vyčíslitelnosti časté použití (Cantor ji mj. použil na důkaz toho, že neexistuje bijekce mezi množinou přirozených a množinou reálných čísel).

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). Více užitečný je ovšem následující pojem:

Definice 9.3 *Problém $P1$ je* (algoritmicky) převeditelný *na problém $P2$, označme $P1 \to P2$, jestliže existuje algoritmus A, který pro libovolný vstup w problému $P1$ sestrojí (tzn. skončí svůj výpočet a jako výstup vydá) vstup $P2$, označme jej $A(w)$, přičemž platí, že odpověď na otázku problému $P1$ pro vstup w je ANO právě tehdy, když odpověď na otázku problému $P2$ pro instanci $A(w)$ 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 Church-Turingovy teze jsou pojmy rekurzivní a algoritmické převeditelnosti totožné.

Užitečnost uvedeného pojmu pro naše účely vyslovuje následující tvrzení, jehož důkaz by měl být zřejmý (srovnejte se situací u NP-úplných, či NP-těžkých, problémů):

Tvrzení 9.4 *Je-li $P1 \to P2$ a problém $P1$ je nerozhodnutelný, je i problém $P2$ nerozhodnutelný.*

Příklad. Takto se např. prokáže nerozhodnutelnost problému UHP (Uniform Halting Problem. Vstup: Tur. stroj M; Otázka: Zastaví se M na každý vstup (tj. platí $\forall w :!M(w)$?.) Při prokázání $HP \to UHP$ 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)

Vstup: 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 $HP \to IPKP \to PKP$, kde problém $IPKP$ je zadán obdobně jako PKP, jen otázka se ptá, zda existuje iniciální řešení pro daný vstup. Hlavní myšlenka převeditelnosti $HP \to IPKP$ spočívá v následujícím: k danému M, w se sestrojí první členy seznamů $u_1 = \$$, $v_1 = \$q_0 w\$$ (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 tehdy, když M se zastaví na w.

(Podrobněji bude probráno na cvičení).

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 ?')

Myšlenka převodu je jednoduchá:

K instanci $u_1, u_2, \ldots, u_n$, $v_1, v_2, \ldots, v_n$ problému PKP sestrojíme gramatiky

$$G_1 : S \to u_1 S a_1 \mid u_2 S a_2 \mid \ldots \mid u_n S a_n \mid u_1 a_1 \mid u_2 a_2 \mid \ldots \mid u_n a_n$$

$$G_2 : S \to v_1 S a_1 \mid v_2 S a_2 \mid \ldots \mid v_n S a_n \mid v_1 a_1 \mid v_2 a_2 \mid \ldots \mid v_n a_n$$

kde $a_1, a_2, \ldots, a_n$ jsou nově přidané symboly.

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.

Poznámka. (Ne)rozhodnutelnost je také důležitou zkoumanou vlastností u logických teorií. Už jsme viděli příklad rozhodnutelné Presburgerovy aritmetiky (ThAdd). Zde jen poznamenejme, že přidáme-li vedle relačního symbolu PLUS ještě symbol MULT (s analogickou interpretací pro násobení), problém zjišťování pravdivosti formulí v množině přirozených čísel je pak nerozhodnutelný (což byl velmi významný a překvapivý výsledek dosažený Gödelem a Churchem ve třicátých létech dvacátého století).

Riceova věta

Na závěr této části uvedeme důležitou větu, jež ukazuje nerozhodnutelnost celé třídy problémů. Vyslovíme ji nejdříve v poněkud neformálním znění; slovo *algoritmus* zde 'pro praktičtější vyznění' nahrazujeme slovem *program* (což je algoritmus zapsaný v nějakém programovacím jazyku).

> *Jakákoli netriviální vlastnost programů týkající se výhradně jejich vstupně/výstupního chování je nerozhodnutelná (tj. množina všech programů s danou vlastností je nerozhodnutelná).*

Rozumí se, že vlastnost V je vstupně/výstupní právě tehdy, když každé dva programy, které realizují totéž vstupně/výstupní zobrazení (převádějí stejné vstupy na stejné výstupy) buď oba vlastnost V mají nebo ji oba nemají.
Vlastnost V je triviální, když ji mají buď všechny programy nebo ji nemá žádný program; jinak (tedy když ex. program, jenž V má, a ex. program, jenž V nemá) se V nazývá netriviální.

Vyslovíme uvedenou větu v přesnější formě (a pro Turingovy stroje). Omezíme se přitom (jak víme, bez velké újmy) na Turingovy stroje realizující (částečná) zobrazení typu $\{0,1\}^* \longrightarrow \{0,1\}^*$ (vstupem je řetězec nul a jedniček, při ukončení je neprázdný úsek pásky také řetězcem nul a jedniček). Označme množinu všech takových Turingových strojů jako ALL.

Věta 9.5 *(Rice) Nechť $\mathcal{A}$ je nějaká množina algoritmicky vyčíslitelných (částečných) zobrazení typu $\{0,1\}^* \longrightarrow \{0,1\}^*$. Pokud pro množinu*

$$\mathcal{M}_{\mathcal{A}} = \{M \mid M \text{ je kód Tur. stroje realizujícího zobrazení patřící do } \mathcal{A})\}$$

platí $\mathcal{M}_{\mathcal{A}} \neq \emptyset$ a $\mathcal{M}_{\mathcal{A}} \neq$ALL, pak $\mathcal{M}_{\mathcal{A}}$ je nerozhodnutelná.

Důkaz: Vezměme nějakou takovou $\mathcal{A}$, která není prázdná ani nezahrnuje všechny alg. vyčíslitelné funkce. Nechť nikde nedefinované zobrazení $\perp : \{0,1\}^* \longrightarrow \{0,1\}^*$ nepatří do $\mathcal{A}$ (opačný případ se řeší podobně). Nechť M_1 realizuje $\perp$, tedy $M_1 \notin \mathcal{M}_{\mathcal{A}}$, a nechť M_2 realizuje nějaké zobrazení z $\mathcal{A}$ (nutně takový existuje); tedy $M_2 \in \mathcal{M}_{\mathcal{A}}$.

Ukážeme, že problém DHP je převeditelný na $\mathcal{M}_{\mathcal{A}}$ (tj. na problém příslušnosti k $\mathcal{M}_{\mathcal{A}}$), čímž prokážeme nerozhodnutelnost $\mathcal{M}_{\mathcal{A}}$.

Algoritmus převodu $DHP \rightarrow \mathcal{M}_{\mathcal{A}}$ k danému stroji (kódu stroje) M (tj. ke vstupu problému DHP) 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_2.

Je zřejmé: když M se zastaví na $Kod(M)$ (tj. odpověď na daný vstup problému DHP je ANO), realizuje M' totéž zobrazení jako M_2 a tedy patří do $\mathcal{M}_{\mathcal{A}}$; když M se nezastaví na $Kod(M)$ (odpověď v DHP je NE), realizuje M' zobrazení $\perp$, tedy totéž jako M_1 a tedy do $\mathcal{M}_{\mathcal{A}}$ nepatří.

□

Přednáška 11

10 Další partie teorie a praxe algoritmů

Stručný obsah: Ilustrace problematiky v oblastech aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů. Mj. též podstata šifrování s veřejným klíčem (metodou RSA). Zmínka o nových, netradičních výpočetních modelech (kvantové výpočty, DNA-výpočty).

Studijní cíle: Umět vysvětlit základní principy v uvedených oblastech a ilustrovat je na příkladech.

10.1 Aproximační algoritmy

Setkali jsme se s řadou optimalizačních úloh – v takové úloze je hledáno optimum v jisté množině přípustných řešení (např. se jedná o minimální kostru ohodnoceného grafu, maximální nezávislou množinu vrcholů grafu, nejkratší okružní jízdu obchodního cestujícího apod.). Pro některé úlohy jsou známy rychlé (tj. polynomiální) algoritmy (např. pro zmíněnou minimální kostru); pro mnohé ovšem polynomiální algoritmy (přesněji řečeno, algoritmy s polynomiální časovou složitostí), které by vždy nalezly optimum, nemáme – a podle mnoha indicií tyto algoritmy ani neexistují (je tomu tak u problému maximální nezávislé množiny i u problému nejkratší okružní cesty; připomeňte si otázku $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$).

Co lze dělat, nemáme-li pro řešení úlohy použitelný algoritmus a přesto ji potřebujeme řešit v praxi ? Jednou z možností je použít *aproximační algoritmus*, čímž rozumíme rychlý (polynomiální) algoritmus, který alespoň aproximuje optimum – tj. vypočte nějaké přípustné

řešení (např. okružní cestu), které není pokud možno moc horší než optimum (tedy např. vypočtená okružní jízda není "o moc delší" než ta nejkratší možná).

Pro sestavení aproximačního algoritmu můžeme např. použít některou obecnou metodu vedoucí k rychlým algoritmům; např. okružní jízdu je možné sestavit "hltavým" (greedy) přístupem – jeho podstatu lze vyjádřit slovy "došel-li jsi už do města A, pokračuj dále do jemu nejbližšího města, které jsi dosud nenavštívil, atd.". Nebo lze např. použít nějakou heuristiku specifickou pro daný problém (tj. blíže nezdůvodněný postup, který dává v praxi uspokojivé výsledky nebo o kterém se alespoň domníváme, že takové výsledky může dávat).

Jak ale hodnotit kvalitu aproximačního algoritmu, tj. kvalitu jeho výstupů ? Určitou orientaci nám samozřejmě vždy může poskytnout experiment; např. srovnáním výstupů dvou různých aproximačních algoritmů pro vybrané (pro nás zajímavé) instance problému můžeme případně usoudit, který z algoritmů se pro naše účely jeví jako vhodnější. Byly a jsou ovšem vypracovávány i teoretické postupy, které umožňují vyjadřovat kvalitu aproximačních algoritmů na exaktnější bázi a navíc umožňují klasifikovat problémy podle míry jejich aproximovatelnosti. Zde si to budeme jen stručně ilustrovat na zmíněném problému obchodního cestujícího (Travelling SalesPerson); uvažujme jej v této formě:

> *Název*: TSP (*problém obchodního cestujícího*)
>
> *Vstup*: úplný neorientovaný graf s n vrcholy ("městy") (úplný znamená, že mezi každými dvěma různými vrcholy je hrana); každé hraně (i, j) je přiřazeno celé kladné číslo $d(i, j)$ (vzdálenost mezi městy i a j)
>
> *Výstup*: permutace $\pi = \{i_1, i_2, \ldots, i_n\}$ množiny $\{1, 2, \ldots, n\}$ (tj. "okružní cesta") tž. $D(\pi) = d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1)$ (tj. délka okružní cesty) je minimální možná.

Bude užitečné zvlášť uvažovat i podproblém problému TSP, který označíme Δ-TSP – pro jeho vstupy je vyžadováno splnění trojúhelníkové nerovnosti (tj. pro lib. vrcholy i, j, k je $d(i, j) \leq d(i, k) + d(k, j)$).

Poznámka. Připomeňme si standardní převod instance problému hamiltonovské kružnice (HK) na instanci (rozhodovacího problému příslušného k) TSP (při prokazování HK◁TSP); jestliže (u, v) je hrana ve výchozím grafu, položíme $d(u, v) = 1$, jinak položíme $d(u, v) = 2$. (Ve výchozím grafu G_1 s n vrcholy existuje hamiltonovská kružnice právě tehdy, když ve vytvořeném (úplném) grafu G_2 existuje okružní cesta délky n.) Jedná se vlastně o převod HK◁ Δ-TSP, což ukazuje, že už podproblém Δ-TSP je NP-těžký.

Cvičení. Zapište (pseudokódem) algoritmus $A1$, který k vstupu problému TSP sestrojí okružní cestu (tj. permutaci) hltavým přístupem, jak jsme se o tom zmínili výše.

Abychom se mohli vyjádřit o kvalitě algoritmu $A1$ (a dalších aproximačních algoritmů), zavedeme několik pojmů a označení. Pro vstup G problému TSP označme $m(G)$ délku nejkratší (tj. optimální) okružní cesty. Pro aproximační algoritmus A (pro problém TSP) označme $m_A(G)$ délku okružní cesty vydané algoritmem A pro vstup G. Je zřejmé, že absolutní chyba $m_A(G) - m(G)$ je nezáporná; ideální by bylo, aby byla co nejmenší. Všimněme si nejprve, že nemůžeme doufat, že by tato chyba byla omezená, a to ani v případě Δ-TSP:

Tvrzení 10.1 *Kdyby pro Δ-TSP existoval (polynomiální) aproximační algoritmus A s omezenou absolutní chybou (tj. existovalo by $k \in \mathbb{N}$ tak, že pro vš. instance G je $m_A(G) - m(G) \leq k$), pak P=NP.*

Důkaz: Předpokládejme, že takový algoritmus A a příslušné číslo k existují. Pak by následující polynomiální algoritmus vydával optimální řešení Δ-TSP (tj. nejkratší okružní cestu):

> - v dané instanci G problému Δ-TSP vynásob každé $d(i, j)$ číslem $k+1$; dostaneš tak instanci G' (trojúhelníková nerovnost zůstane zachována)
>
> - na G' spusť algoritmus A; ten nutně vydá optimální řešení (permutaci vrcholů, odpovídající nejkratší cestě) (proč ?), které je ovšem i optimálním řešením pro G (proč ?)

□

Když už nemůžeme zajistit omezení absolutní chyby $m_A(G) - m(G)$, můžeme alespoň omezit (nějakou konstantou) *aproximační poměr* $m_A(G)/m(G)$? Algoritmus $A1$ nesplňuje ani to:

Cvičení. Ukažte, že pro každé $c > 1$ a každé $n > 3$ existuje instance G s n vrcholy problému TSP tak, že $m_{A1}(G)/m(G) > c$. Trošku náročnější je pak ukázat, že pro každé $c > 1$ existuje pro nekonečně mnoho n instance G s n vrcholy problému Δ-TSP tak, že $m_{A1}(G)/m(G) > c$.

Aproximační poměr se u algoritmu $A1$ dá alespoň omezit pomalu rostoucí funkcí velikosti vstupu - v případě problému Δ-TSP; konkrétně se dá ukázat (n označuje počet vrcholů grafu G)

$$\frac{m_{A1}(G)}{m(G)} \;\leq\; \frac{1}{2}\,(\lceil \log n \rceil + 1)$$

To ale neznamená, že problém Δ-TSP není aproximovatelný s omezeným aproximačním poměrem. Zkusme sofistikovanější (myšlenkově náročnější) algoritmus $A2$ (formulujeme ho obecně pro TSP):

> ALGORITMUS $A2$
>
> Pro daný vstup TSP, tj. ohodnocený graf s n vrcholy:
>
> - sestroj minimální kostru grafu (podalgoritmem složitosti $O(n^2)$)
>
> - zvol lib. vrchol a z něj realizuj prohledání sestrojené kostry (která je stromem) do hloubky (depth-first search)

- pořadí navštívení jednotlivých vrcholů při onom prohledávání udává permutaci, která je výstupem algoritmu

Pro $A2$ snadno ukážeme, že v případě problému Δ-TSP je aproximační poměr omezený konstantou, konkrétně

$$\frac{m_{A2}(G)}{m(G)} \;\leq\; 2$$

Cvičení. Dokažte předchozí vztah; speciálně dokažte $c \leq m(G)$ a $m_{A2}(G) \leq 2c$, kde c je součet ohodnocení hran v minimální kostře.

Algoritmus $A2$ je tedy lepší než $A1$; v případě problému Δ-TSP nám vždy zaručuje nalezení okružní cesty, která je nejvýše dvakrát tak dlouhá jako nejkratší cesta (optimum).

Poznámka. Christofides (1976) ukázal ještě lepší algoritmus $A3$ pro Δ-TSP, pro nějž $m_{A3}(G)/m(G) \leq 3/2$. (Sestavení algoritmu a jeho ověření je ovšem náročnější.) Aproximační algoritmus s lepším aproximačním poměrem není znám, ani není známo, zda by existence takového algoritmu implikovala P=NP.

U obecného problému TSP nelze ovšem žádný r-aproximační algorimus, tj. algoritmus s aproximačním poměrem omezeným konstantou r, očekávat:

Tvrzení 10.2 *Jestliže pro TSP existuje r-aproximační algoritmus (pro nějaké $r \in \mathbb{N}$), pak $P = NP$.*

Důkaz: Předpokládejme, že existuje r-aproximační algoritmus A pro TSP.

Nyní v převodu instance problému hamiltonovské kružnice (HK) na instanci TSP zmíněném výše položme $d(u,v) = 1$, je-li (u,v) hrana výchozího grafu, a jinak položme $d(u,v) = 1 + nr$, kde n je počet vrcholů výchozího grafu. Algoritmus A spuštěný na vytvořené instanci problému TSP by de facto rozhodl, zda ve výchozím grafu je hamiltonovská kružnice (proč ?).

□

K dalšímu studiu problematiky aproximačních algoritmů lze doporučit především přehledovou monografii [A$^+$99]; některé aspekty jsou také rozebírány např. v [Hro99].

10.2 Pravděpodobnostní algoritmy

Existují rozhodovací (tedy ANO/NE) problémy, pro které neznáme rychlé (tj. polynomiální) algoritmy, a přesto je lze v praxi spolehlivě řešit nejen pro malé vstupy. Stačí slevit z absolutního nároku na řešící algoritmus, totiž z toho, aby algoritmus vždy vydal *zaručeně* (tedy stoprocentně) správnou odpověď. Přesněji řečeno, přestaneme vyžadovat, aby algoritmus nutně předepisoval *deterministický* proces, a připustíme náhodný prvek (házení

kostkou). Opakované běhy algoritmu na tentýž vstup pak mohou vydávat různé výstupy – každý z nich s určitou pravděpodobností.

K formalizaci pojmu pravděpodobnostního algoritmu můžeme použít jednoduchou verzi *pravděpodobnostního Turingova stroje* (probabilistic Turing machine, PTM); takový stroj $PM = (Q, \Sigma, \Gamma, \delta, q_0, F)$ si můžeme představit jako standardní Turingův stroj s tím, že $\delta(q,a)$ může být nyní dvouprvková množina – v tom případě se při výpočtu v konfiguraci $uqav$ volí jedna ze dvou možných bezprostředně následujících konfigurací náhodně – tak, že každá má pravděpodobnost zvolení 1/2 (můžeme si představit, že se zde hází mincí).

Podobně jako u nedeterministického Turingova stroje, odpovídá jednomu vstupu PTM strom výpočtů. Vrcholy stromu jsou ohodnoceny konfiguracemi; kořen je ohodnocen počáteční konfigurací a následníci vrcholu ohodnoceného konfigurací c jsou ohodnoceni právě bezprostředně následujícími konfiguracemi konfigurace c.

Každá hrana je přitom ohodnocena příslušnou pravděpodobností – hraně vycházející z vrcholu s jedním následníkem je přiřazena 1, každé ze dvou hran vycházejících z vrcholu s dvěma následníky je přiřazena 1/2. Každému vrcholu v je přiřazena pravděpodobnost jeho dosažení – tj. součin ohodnocení hran na cestě od kořene k vrcholu v.

Pak lze např. přesně definovat pravděpodobnost jevu, že daný PTM vydá na daný vstup x odpověď ANO – tato pravděpodobnost je dána součtem pravděpodobností vrcholů ve stromu výpočtu pro vstup x, které jsou ohodnoceny koncovými konfiguracemi, znamenajícími odpověď ANO (tedy přijímajícími konfiguracemi).

Nepůjdeme zde do dalších formálních detailů, ale budeme si pravděpodobnostní algoritmy ilustrovat na problému prvočíselnosti:

Název: *Prvočíselnost*

Vstup: přirozené číslo k (v dekadickém zápise)
Výstup: ANO, když k je prvočíslo, NE, když k je číslo složené.

Zmiňovali jsme už dříve, že přímočaré testování prvočíselnosti podle definice vede k algoritmu s exponenciální složitostí. Žádný polynomiální) *deterministický* algoritmus nebyl až do léta 2002 znám. (Teď už je znám, ale přesto má smysl kvůli větší rychlosti nadále uvažovat pravděpodobnostní algoritmus.)

Poznámka. Všimněme si, že je zřejmé, že doplňkový problém – tj. problém složenosti čísla – je v NP (proč ?). Využitím jistých výsledků teorie čísel se dá ukázat, že i problém prvočíselnosti je v NP. Tento problém byl dlouho jedním z mála "přirozených" problémů v NP (resp. v NP$\cap$co-NP), u kterých není znám polynomiální algoritmus a ani se neumí prokázat NP-úplnost. (Měli bychom upřesnit, že již dříve byla známa existence polynomiálního algoritmu, který by prvočíselnost rozhodoval za předpokladu, že platí tzv. Riemannova hypotéza – to se ovšem neví.)

S využitím poznatků teorie čísel lze sestavit rychlý pravděpodobnostní algoritmus A (existuje polynom p tak, že délka každé větve stromu výpočtu pro k je omezena $p(\log k)$; délka výpočtu je tedy omezena polynomiální funkcí velikosti vstupu), který má tyto vlastnosti

- jestliže n je prvočíslo, vydá A odpověď ANO s pravděpodobností 1 (NE vůbec nemůže vydat)

- jestliže n není prvočíslo, vydá A odpověď NE s pravděpodobností alespoň $1/2$.

Představme si nyní, že A zopakujeme pro dané n např. 50-krát. Když (aspoň) jednou vydá NE, víme, že n jistě není prvočíslo (a dále už A opakovat nemusíme). Když ve všech případech vydá ANO, tak n je prvočíslo s pravděpodobností větší než $1 - 2^{-50}$ – tedy "prakticky stoprocentně" (proč ?).

Naznačíme, jak algoritmus A vypadá. Mj. se opírá o tzv. *malou Fermatovu větu*:

Jestliže p je prvočíslo, tak pro každé $a, 0 < a < p$, platí

$$a^{p-1} \equiv 1 \pmod{p}$$

Poznámka. Náčrt důkazu: Rozvineme-li $(a+b)^p$ podle binomické věty, ověříme snadno, že výsledek modulo p je roven $a^p + b^p$, je-li p prvočíslo (ověřte !). Tedy (počítáno modulo p) máme $1^p \equiv 1$, $2^p \equiv (1+1)^p \equiv 1^p + 1^p \equiv 1+1 \equiv 2$ a obecně $a^p \equiv a$, z čehož plyne $a^{p-1} \equiv 1$.

Jako důkaz složenosti (tj. neprvočíselnosti) daného čísla m nejlépe slouží nějaký netriviální dělitel a čísla m (všichni známe rychlý algoritmus, kterým se přesvědčíme, že dané a skutečně dělí m). Z Fermatovy věty ovšem plyne důležité pozorování: když nalezneme pro dané m číslo $a, 0 < a < m$ tak, že $a^{m-1} \not\equiv 1 \pmod{m}$, tak jsme našli *svědka složenosti* čísla m (tedy svědka toho, že m není prvočíslo), byť tím nezískáme netriviálního dělitele m (jen jsme tak dokázali jeho existenci).

Poznámka. Je důležité si uvědomit, že počítání mocnin modulo m umíme rovněž velmi rychle, a sice tzv. metodou "opakovaného umocňování": počítaním $x_0 = a$, $x_1 = (x_0)^2 \bmod m$, $x_2 = (x_1)^2 \bmod m$, $x_3 = (x_2)^2 \bmod m$, ..., $x_i = (x_{i-1})^2 \bmod m$, ... dostáváme postupně a, $a^2 \bmod m$, $a^4 \bmod m$, $a^8 \bmod m$, ..., $a^{2^i} \bmod m$, ... Chceme-li pak např. znát a^{560}, vynásobíme $a^{512} \cdot a^{32} \cdot a^{16}$ (tj. $x_9 \cdot x_4 \cdot x_5$; vše počítáno samozřejmě modulo m).

Představme si nyní *hypoteticky*, že by platilo:

Hypotéza:
jestliže m není prvočíslo, tak alespoň jedna polovina z čísel $1, 2, \ldots, m-1$ jsou svědkové složenosti čísla m

Pak by kýžený pravděpodobnostní algoritmus A pro testování prvočíselnosti (který se pro zadané složené číslo mohl splést s pravděpodobností nejvýš $1/2$) byl nabíledni. (Jak by vypadal ?)

Jak ukazují další výsledky teorie čísel, situace není takto jednoduchá – uvedená hypotéza neplatí pro všechna složená čísla; definice svědka složenosti se ale dá upravit tak, aby hypotéza (pro nově definovaného svědka) platila !

Poznamenejme, že ač problém testování prvočíselnosti je takto uspokojivě vyřešen, nejsou známy žádné rychlé algoritmy pro nalezení prvočíselného rozkladu složeného čísla. Tedy ač např. umíme rychle zjistit, že dané 200-místné číslo je složené, nezaručuje nám to rychlé nalezení netriviálního dělitele. Speciálně, když nám někdo dá 200-místné číslo, které vytvořil jako součin dvou 100-místných prvočísel, nemáme (zatím ?) praktickou šanci tato prvočísla najít. (To, že tato "praktická šance" skutečně neexistuje, neumíme dokázat. V této souvislosti si všimněte zmínky o Shorově algoritmu v kapitolce o kvantových výpočtech !). V následující podkapitolce si ukážeme, k čemu se to používá.

Šifrovací systém s veřejným klíčem; RSA-kryptosystém

Stručně nastíníme podstatu jednoho používaného způsobu elektronické komunikace, při kterém se zpráva šifruje veřejně známým klíčem adresáta a který rovněž umožňuje tzv. 'elektronické podpisy'. Přitom bude zřejmá souvislost bezpečnosti popsaného způsobu s otázkami teorie složitosti.

Mějme doménu D (přípustných) zpráv; např. si představme množinu všech (zápisů) 200-místných dekadických čísel. (Čtenář si snadno představí kódování textu pomocí řetězce dekadických číslic; delší zprávy pak sestávají z více bloků.)

Dvěma permutacím (vzájemně jednoznačným zobrazením) $enc : D \to D$ a $dec : D \to D$ řekneme oboustranně komutativní šifrovací pár, dále jen *šifrovací pár*, jestliže

$$\forall m \in D : dec(enc(m)) = enc(dec(m)) = m$$

Přitom enc a dec jsou efektivně vyčíslitelné (to zde znamená, že pro ně existují rychlé algoritmy) a nemáme způsob, jak z algoritmu pro enc odvodit algoritmus pro dec.

Princip šifrovacího systému s veřejným klíčem, v němž spolu uživatelé komunikují, spočívá v tom, že každý uživatel X si vyrobí svůj šifrovací pár (enc_X, dec_X), zveřejní algoritmus pro enc_X a drží v tajnosti algoritmus pro dec_X.

Když chce uživatel A zaslat zprávu m uživateli B, vyhledá ve veřejném seznamu (algoritmus pro) enc_B a zašle mu $enc_B(m)$. B po obdržení aplikuje dec_B a získá $dec_B(enc_B(m)) = m$.

Oboustranná komutativnost u šifrovacího páru umožňuje *elektronické podpisy*: A zašle dvojici $enc_B(m)$, $enc_B(dec_A(m))$; B po přečtení $dec_B(enc_B(m)) = m$ zjistí, že zpráva má být od A (A uvede v m své jméno), druhá část $dec_B(enc_B(dec_A(m))) = dec_A(m)$ je pro něj nesrozumitelná – ovšem po vyhledání a aplikaci enc_A musí dostat m, čímž ověří pravost (když se např. jedná o podepsaný šek, může ho B předložit bance k proplacení; všimněte si, že B není schopen podpis A zfalšovat).

V RSA-systému si uživatel X vyrobí šifrovací pár podle následujícího algoritmu:

1. Zvol dvě náhodná 100-místná prvočísla p, q.

2. Spočítej součiny $n = pq$ a $\Phi(n) = (p-1)(q-1)$.

3. Urči (malé) e tž. $GCD(e, \Phi(n)) = 1$ (GCD označuje největší společný dělitel).
4. Vypočti d tž. $de \equiv 1(\bmod \Phi(n))$.
5. Zveřejni dvojici (e, n); šifrovací funkce je $enc_X(m) = m^e \bmod n$.
6. Drž v tajnosti (d, n); dešifrovací funkce je $dec_X(m) = m^d \bmod n$.

Bod 1 lze provést takto: Náhodně zvolíme 100-místné liché číslo, a otestujeme jeho prvočíselnost pravděpodobnostním algoritmem zmíněným výše. Když prvočíslem není, přičteme 2, znovu otestujeme, v negativním případě znovu přičteme 2 atd., dokud test prvočíselnosti neskončí pozitivně. Dá se ukázat, že prvočíslo bude "takřka jistě" nalezeno mezi prvními 200 testovanými čísly. (To lze odvodit z faktu, že funkce $\pi(n)$ udávající počet prvočísel mezi prvními n přirozenými čísly, je velmi dobře aproximována funkcí $n/\ln n$.)

Body 3. a 4. se dají řešit takto: postupně probíráme (kandidáty na číslo) e a Eukleidovým algoritmem ověřujeme zda, $GCD(e, \Phi(n)) = 1$; v kladném případě dostaneme (při určité variantě Eukleidova algoritmu) přímo lineární kombinaci $a \cdot e + b \cdot \Phi(n) = 1$ a tedy lze vzít $d = a$.

Z elementárních faktů teorie čísel se dá ukázat korektnost, tj. $\forall m : m^{ed} \equiv m \pmod n$.

Připomeneme-li si metodu "opakovaného umocňování", je zřejmé, že (de)šifrovat se dá velmi rychle.

Bezpečnost systému spočívá v tom, že se neví, jak d zjistit jinak, než pro n nalézt rozklad $n = p \cdot q$; jak jsme už zmínili, nejsou ale známy žádné algoritmy, které by byly v rozumné době schopné nalézt prvočíselný rozklad 200-místných čísel.

Další informace o problematice pravděpodobnostních algoritmů, šifrování apod. najde čtenář např. v [CLR90], [Sip97].

Přednáška 12

10.3 Paralelní algoritmy

V této části se jen letmo dotkneme oblasti paralelních algoritmů a paralelní výpočtové složitosti.

S rozvojem paralelních architektur, speciálně počítačů s více procesory, se přirozeně vynořila otázka, zda a které problémy lze řešit rychleji, použijeme-li k jejich řešení paralelní přístup, tedy algoritmus, který (na rozdíl od standardního sekvenčního algoritmu) počítá s více vykonavateli (procesory). Např. k problému

> *Název*: *Vzorek v textu*
>
> *Vstup*: vzorek p ('krátká' posloupnost symbolů) a text t ('dlouhá' posloupnost symbolů)
>
> *Výstup*: místa všech výskytů p v t

můžeme nějaký standardní algoritmus A nahradit paralelním

> - rozděl vstup t na dvě (přibližně stejně dlouhé) části t_1, t_2
> - spusť paralelně A na p, t_1 a A na p, t_2
> - zpracuj ('slej') výsledky obou paralelně běžících '(pod)procesů' (přitom ošetři rozhraní t_1, t_2) a vydej na výstup

Máme-li skutečně možnost paralelní běh implementovat (máme dva vykonavatele), doba výpočtu se v zásadě dvakrát zmenší oproti algoritmu A – za předpokladu, že přidaná činnost spojená s rozdělením práce a zkombinováním výsledků práce jednotlivých (pod)procesů je v celkovém kontextu zanedbatelná.

Čtenář snadno navrhne úpravu algoritmu pro případ tří, čtyř, či obecně k procesorů. Na druhé straně asi tuší, že chceme-li nějak postihnout a studovat paralelní výpočty obecně, nestačí se omezit na model s fixním počtem procesorů. Proto budeme počítat s (potenciálně) neomezeným paralelismem, konkrétně s modely umožňujícími nasadit na vstup velikosti n až $f(n)$ procesorů, kde f je nějaká (např. i exponenciální) funkce.

V případě hledání výskytů vzorku p v textu t délky m si můžeme představit nasazení m procesorů $1, 2, \ldots, m$, přičemž procesor i zjišťuje, zda se vzorek p nachází na pozici i (v kladném případě vydá i na výstup). Pomineme-li etapu rozdělení práce mezi procesory a řešení případných konfliktů na výstupu, je délka 'jádra' výpočtu celého systému uměrná délce ℓ vzorku p.

Můžeme ovšem také paralelizovat činnost ověření, zda p se vyskytuje na pozici i: na tento úkol nasadíme tým – ℓ-tici procesorů $(i, 1), (i, 2), \ldots, (i, \ell)$, kde procesor (i, j) zjišťuje, zda $p(j) = t(i + j - 1)$ (zde $p(j)$ označuje j-tý znak v p); v záporném případě např. zapíše 1 do jisté paměťové buňky b_i vyhrazené i-tému týmu (její hodnota byla na začátku 0). Po ukončení práce všech členů týmu 'manažer týmu' (např. $(i, 1)$) vydá na výstup i v případě, že $b_i = 0$. Takto je délka trvání 'jádra' výpočtu nezávislá na velikosti vstupu a lze ji omezit nějakou konstantou.

Abychom mohli úvahy o paralelních algoritmech a jejich výpočtové složitosti zpřesnit, potřebujeme nějaký konkrétní 'paralelní počítač', resp. jeho abstraktní model. Podobně jako nám v případě sekvenčních algoritmů posloužil model RAM, může nám zde posloužit PRAM (parallel RAM), definovaný níže. Na rozdíl od 'sekvenčního případu', kde RAM je skutečně obecně přijímaným referenčním modelem, v 'paralelním případě' je otázka 'toho pravého' modelu daleko komplikovanější. Nejdříve se ale soustředíme na PRAM a pak se ke zmíněné otázce vrátíme.

Stroj PRAM sestává z potenciálně nekonečného pole strojů (procesorů) RAM očíslovaných $0, 1, 2, \ldots$ a ze (sdílené) globální paměti; přitom

- každému procesoru je jeho jedinečné identifikační číslo přístupné jako hodnota speciálního registru PID (processor identifier)
- každý procesor může kromě své lokální paměti přistupovat také ke společné globální paměti (která je organizována stejně jako paměti lokální)

Přitom (je možné si představit centrální řídicí počítač, který zajišťuje, že)

- všechny procesory se řídí týmž algoritmem
- procesory pracují paralelně a synchronizovaně, t.j. v každém kroku PRAMu se provede jedna instrukce na každém RAMu

Důležitou technickou otázkou jsou případné konflikty při paralelním čtení ze / zapisování do stejné buňky (globální) paměti. Zde předpokládáme

- ze stejné buňky sdílené paměti může v rámci každé instrukce libovolný počet procesorů číst (CR = Concurrent Read)
- do stejné buňky sdílené paměti může v rámci každé instrukce libovolný počet procesorů zapisovat za předpokladu, že všechny zúčastněné procesory zapisují tutéž hodnotu (CW-C Concurrent Write Common).

Poznamenejme jen, že používaných variant PRAMu je více; např. typ CREW umožňuje souběžné čtení, ale zakazuje souběžné zapisování (Exclusive Write).

Jinak ještě dodejme, že vstupní data jsou ukládána v počátečním úseku globální paměti (kam se také uloží výstup).

Čtenář si teď může rozmyslet naprogramování výše uvedeného algoritmu pro hledání vzorku v textu pro PRAM. Důležitým bodem k promyšlení je využití PID procesorů k rozdělení práce (každý procesor na základě svého PID zjistí, v jakém týmu a na jakém úkolu má pracovat). My to zde provádět nebudeme, ale ilustrujeme na příkladu násobení (booleovských) matic.

Uvažujme nejdříve standardní násobení dvou matic $n \times n$; provádíme přitom $\Theta(n^3)$ číselných operací (což odpovídá složitosti standardního sekvenčního algoritmu). Můžeme-li nasadit n^2 procesorů, z nichž každý (nezávisle) spočítá jeden prvek výsledné matice, potřebný čas se sníží na $O(n)$. Nebudeme teď zkoumat, jak se dá postup vylepšit nasazením týmu na součet n čísel, ale zjednodušíme si situaci uvažováním násobení *booleovských* matic: máme spočítat $C := A \times B$, kde prvky jsou 0 a 1, přičemž $1 + 1 = 1$. Nasaďme n^2 týmů o n členech, tedy celkově n^3 procesorů – označme je (i, j, p) pro $1 \leq i, j, p \leq n$. Pokud každý procesor (i, j, p) vykoná (paralelně s ostatními) příkaz

> **if** $A(i, p)$ and $B(p, j)$ **then** $C(i, j) := true$

je úkol hotov !

Pseudokód celého programu, jímž se každý procesor řídí, je zachycen níže (čtenář samozřejmě vidí, jak by šel tento pseudokód přímočaře přepsat jako posloupnost skutečných instrukcí RAMu). Všimněte si, že pro konkrétní i, j provede příkaz $C(i, j) := true$ nula až n procesorů – jelikož všechny procesory ovšem přiřazují $C(i, j)$ tutéž hodnotu, jedná se o povolené souběžné zapisování (CW-C). Také je využito předpokladu, že matice C je na začátku vynulována (všechny prvky jsou $false$).

> VSTUP: Ve sdílené paměti je uloženo číslo n a dvourozměrná pole A, B, C : array $[1 \ldots n, 1 \ldots n]$ of *boolean*; všechny prvky C jsou přitom 0 (tj. $false$).
>
> VÝSTUP: Pole C obsahuje booleovský součin matic A, B
>
> POSTUP:
>
> **begin** { každý procesor začíná výpočtem (i, j, p) ze svého PID }
>
> $i := 1 +$ PID $div\ n^2$;
> $j := 1 +$ (PID $mod\ n^2$) $div\ n$;
> $p := 1 +$ PID $mod\ n$;
>
> **if** $A(i, p)$ and $B(p, j)$ **then** $C(i, j) := true$
> { píše 0 až n procesorů, ale vždy stejnou hodnotu }
>
> **end**

Takto máme tedy (paralelní) algoritmus, který vynásobí booleovské matice A, B typu $n \times n$ v konstantním čase při použití n^3 procesorů.

Zmínili jsme již, že v paralelním případě je otázka všeobecně přijímaného referenčního modelu daleko složitější než v případě sekvenčním. Nebudeme teď diskutovat nakolik je PRAM blízký či vzdálený reálným paralelním architekturám, jen zmíníme, že patří do tzv. druhé třídy výpočetních modelů (viz např. [vEB90]). *První počítačovou třídu* zde tvoří všechny tzv. rozumné sekvenční modely, určené tezí

> *Rozumné sekvenční modely* dokážou simulovat a být simulovány Turingovými stroji s nejvýš polynomiální časovou ztrátou a s nejvýš lineární prostorovou ztrátou.

Pozn.: Nebudeme se zde zabývat nuancí, zda se vyžaduje, aby obě podmínky platily u příslušné simulace zároveň.

Druhou počítačovou třídu pak tvoří tzv. rozumné paralelní modely, určené tzv. *paralelní tezí* (stručně: sekvenční prostor = paralelní čas):

> Problémy řešitelné na rozumných *sekvenčních* modelech v polynomiálně omezeném *prostoru* se dají řešit na *rozumných paralelních modelech* v polynomiálně omezeném *čase*.

Pozn.: Někdy se teze formuluje obecněji: $\mathcal{M}$ je rozumný paralelní model, jestliže pro libovolnou funkci F je

$$\bigcup_k \mathcal{M}\text{-TIME}(F^k(n)) = \bigcup_k \text{SPACE}(F^k(n))$$

(Připomeňme, že $F^k(n)$ označuje $(F(n))^k$.)

Poznamenejme, že má rozumný smysl uvažovat i jiné třídy počítačů, např. jisté slabší ale 'fyzikálně realizovatelnější' paralelní modely (viz např. [Wie92]).

Praxi nejbližší je zřejmě navrhování efektivních paralelních algoritmů pro konkrétní problémy. U pojmu efektivního (a prakticky realizovatelného) paralelního algoritmu došlo k určité shodě: *efektivním paralelním algoritmem* se rozumí paralelní algoritmus (je možno dosadit PRAM), který pracuje v polylogaritmickém prostoru (tj. s celkovou prostorovou složitostí $(\log n)^k$ pro něj. k) a s nasazením polynomiálního počtu procesorů. (Nasazení většího než polynomiálního počtu procesorů lze těžko považovat za zvládnutelné.) Třída problémů, které jsou řešitelné takovýmito efektivními paralelními algoritmy, se označuje NC (Nick's Class; název je podle Nicka Pippengera); poznamenejme, že definice třídy NC je robustní vzhledem k mnoha variantám modelů paralelních algoritmů.

Je snadné vyvodit $\text{NC} \subseteq \text{PTIME}$; opět se ale neumí dokázat, zda inkluze je vlastní (má se zato, že ano). Zhruba řečeno, NC obsahuje ty (sekvenčně zvládnutelné) problémy, které lze efektivně paralelizovat. Např. kniha [GR88] ukazuje řadu takových příkladů; problémy na grafech, třídění, analýza bezkontextových jazyků apod.

Pozn. Všimněme si, že nutnou podmínkou k existenci efektivního paralelního algoritmu pro daný problém je možnost jeho řešení v polylogaritmickém prostoru na sekvenčním stroji (viz paralelní tezi pro $F = \log$). Např. pro rozpoznávání bezkontextových jazyků bylo známo, že prostorová složitost je v $O(\log^2 n)$; netriviální efektivní paralelní algoritmus pro tento problém rozebírá také např. [CM90].

Jak jsme řekli, neumí se dokázat, že existují problémy v $\text{PTIME} - \text{NC}$, tzv. *vnitřně sekvenční* (inherently sequential) problémy. Nejpravděpodobnějšími kandidáty na takové problémy jsou tzv. PTIME-úplné (stručně P-úplné) problémy; *problém je P-úplný* jestliže je v P a každý jiný problém v P je na něj převeditelný (sekvenčním) algoritmem pracujícím v logaritmickém prostoru (konkrétněji: Turingovým strojem s jednou vstupní páskou, jež je 'read-only' a je na ní napsán vstup velikosti n, jednou 'read-write' pracovní páskou, na níž je možno navštívit $\log n$ políček, a jednou 'write-only' výstupní páskou).

Poznámka. Není těžké ukázat, že takováto LOGSPACE redukce je realizovatelná v polylogaritmickém čase s použitím polynomiálního počtu procesorů; je to tedy instance tzv. NC-redukce. Někdy se P-úplnost definuje v širším smyslu – pomocí NC-redukcí.

Příkladem P-úplného (a tedy pravděpodobně 'vnitřně sekvenčního') problému je problém vyhodnocení booleovského obvodu, a to i v monotónní verzi (bez negace), kterou uvedeme:

> *Název*: (mono) CVP (monotone Circuit Value Problem)
>
> *Vstup*: konečný acyklický orientovaný graf (obvod), který má jediný výstupní vrchol (nevychází z něj hrana), všechny jeho vstupní vrcholy (nevchází hrana) jsou ohodnoceny 0 nebo 1 a všechny nevstupní vrcholy jsou označeny $\wedge$ (and) či $\vee$ (or).
>
> *Výstup*: hodnota na výstupním vrcholu (hradle)

(Z kontextu je snad zřejmé, že hodnota hradla $\wedge$ je 1 právě když hodnota všech jeho předchůdců je 1 a hodnota hradla $\vee$ je 1 právě když hodnota aspoň jednoho jeho předchůdce je 1.)

Poznamenejme, že podobně jako u NP-úplných problémů a v jiných analogických situacích je nejtěžším prokázat P-úplnost nějakého (základního) problému. P-úplnost (resp. P-obtížnost) dalších problémů lze pak ukazovat LOGSPACE redukcemi (resp. NC-redukcemi) z už známých P-úplných problémů. (Dá se totiž ukázat, že složení dvou LOGSPACE-redukcí je také realizovatelné jako LOGSPACE-redukce, byť to není tak triviální jako v případě PTIME-redukcí; zkuste si rozmyslet proč.)

10.4 Distribuované algoritmy

Distribuované algoritmy také představují jistý druh paralelismu. Typické ovšem je, že běží zároveň na třeba i hodně vzdálených samostatných počítačích, které nejsou svázány s nějakým centrálním počítačem, a pracují *asynchronně* (nikoli 'v taktu'). Vzájemně komunikují zasíláním zpráv po síti, kterou jsou vzájemně propojeny.

Stručně jen načrtneme jeden problém z dané oblasti a řešení necháme na čtenáři. Podrobné informace o distribuovaných algoritmech lze nalézt např v [Lyn96].

Volba koordinátora

Zadání je převzato z [CP91]. Představme si procesory spojené v *kruhové síti*. Předpokládáme:

1. Každý procesor je připojen na dva obousměrné komunikační kanály, jejichž prostřednictvím si vyměňuje zprávy se svými dvěma sousedy.
2. Komunikace je asynchronní; každý procesor může v libovolném okamžiku vyslat zprávu jednomu ze svých sousedů. Zprávy jsou doručovány bezpečně (neztrácejí se ani nekomolí) a v pořadí, ve kterém byly odeslány (předpokládá se, že kanály jsou vybaveny na příjmu i vysílání vyrovnávací pamětí, organizovanou jako fronta).
3. Všechny procesory se řídí týmž algoritmem. Každý procesor je opatřen svým (zaručeně jedinečným) identifikačním číslem PID (jehož hodnota je mu dostupná). Žádný z procesorů neví, jak je kruh dlouhý ani jaká mají čísla jiní účastníci.

4. Všechny procesory jsou trvale připraveny přijímat a zpracovávat zprávy.

Zadání úlohy:

> Prostřednictvím výměny zpráv zjistit maximální identifikační číslo na kruhu. Proces zjišťování maxima, zvaný též 'volba koordinátora', může zároveň spustit libovolný počet procesorů. Proces skončí v okamžiku, kdy všechny procesory znají maximum a je doručena poslední zpráva, která byla s tímto procesem spojená.

Mírou velikosti zadání je počet n procesorů na kruhu. Přirozenou mírou složitosti algoritmu je celkový počet $M(n)$ vyslaných zpráv (komunikační složitost). Zkuste navrhnout (distribuovaný) algoritmus s pokud možno co nejmenším $M(n)$.

10.5 Nové výpočetní modely

Zde v podstatě jen poznamenáme, že provádění výpočtů (computation) není nutně omezeno jen na elektronické počítače, jak je známe.

Kvantové výpočty (Quantum computing)

Zhruba v 80. létech 20. století se začala diskutovat možnost realizace výpočtů (počítačů) na bázi kvantové mechaniky. Z těchto (fyzikálních) úvah vznikl jistý teoretický model počítače, pro který byly v 90. létech navrženy zajímavé algoritmy. Patrně nejznámější je existence *Shorova algoritmu pro prvočíselný rozklad čísel*, který pracuje (na kvantovém počítači) v polynomiálním čase – v případě praktické realizace by tak umožnil rozbití šifrovacích algoritmů užívaných pro bezpečnou výměnu zpráv, elektronické podpisy apod. (viz kapitolku o pravděpodobnostních algoritmech a šifrování).

Kvantový počítač, resp. jeho model, je v něčem podobný pravděpodobnostní verzi Turingova stroje (jednotlivé možnosti nemají ovšem přiřazeny pravděpodobnosti (reálná čísla), ale komplexní 'amplitudy'). Zdroj možných efektivních algoritmů je v tom, že kvantový stroj dokáže de facto najednou (a deterministicky) ověřovat exponenciálně mnoho možností, z nichž ovšem dostaneme k dispozici jen jednu v okamžiku 'pozorování' výpočtu; pointa je v tom, jak navrhnout algoritmus (a dobu pozorování), aby se pravděpodobnost pozorování 'dobrých' řešení blížila k jedničce, zatímco pravděpodobnost pozorování 'špatných' řešení šla k nule.

Zde téma nebudeme dále rozebírat a čtenáře odkazujeme např. na [Gru99].

DNA-výpočty (DNA-computing)

Experimentální výsledky (např. při řešení instancí některých NP-úplných problémů) byly dosaženy při výpočtech na 'biologické bázi'; čtenáře můžeme odkázat např. na [PRS98].

Reference

[A+99] Giorgio Ausiello et al. *Complexity and Approximation.* Springer Verlag, 1999.

[Chy84] Michal Chytil. *Automaty a gramatiky.* SNTL Praha, 1984.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[CM90] Michal Chytil and František Mráz. Složitost paralelních výpočtů. In *Sborník SOFSEM'90*, pages 157–186, 1990.

[CP91] Michal Chytil and Jan Pavelka. Algoritmy. MFF UK, Praha, 1991.

[GR88] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[Gru99] Jozef Gruska. *Quantum Computing.* McGraw-Hill, 1999.

[Har93] D. Harel. *Algorithmics (The Spirit of Computing).* Addison Wesley, 1993.

[Hro99] Juraj Hromkovič. Stability of approximation algorithms for hard optimization problems. In *Proc. SOFSEM'99*, volume 1725 of *Lecture Notes in Computer Science*, pages 29–47. Springer Verlag, 1999.

[HU78] J. Hopcroft and J. Ullman. *Formálne jazyky a automaty.* Alfa Bratislava, 1978.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[Kuč83] Luděk Kučera. *Kombinatorické algoritmy.* SNTL Praha, 1983.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[Man81] Zohar Manna. *Matematická teorie programů.* SNTL Praha, 1981.

[Mor84] J. Morávek. *Složitost výpočtů a optimální algoritmy.* Academia Praha, 1984.

[PRS98] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing: New Computing Paradigms.* Springer Verlag, 1998.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publish. Comp., 1997.

[vEB90] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 1. Elsevier, 1990.

[Wie92] Juraj Wiedermann. Weak parallel machines: a new class of physically feasible parallel machine models. In *Proc. MFCS'92*, volume 629 of *Lecture Notes in Computer Science*, pages 95–111. Springer Verlag, 1992.