Základní pojmy

1 Pojmy problém a algoritmus

V této kapitole zavedeme dva základní pojmy potřebné pro kurz, problém a algoritmus. Slovo problém má v běžném jazyce široký význam. Naznačuje, že se nacházíme v neuspokojivé situaci (např. před nevyluštěnou křížovkou) a chceme ji, ať už je naše motivace jakákoliv, změnit, vyřešit problém (např. vyluštit křížovku). V kurzu se budeme zabývat specifickým významem slova. Omezíme se na problémy, které lze přesně specifikovat. To znamená, že můžeme přesně a objektivně říci, v čem problém spočívá a co je jeho řešením. Navíc, seskupíme konkrétní problémy stejného typu do skupin, kterým pak říkáme algoritmický problém. Konkrétním problémům pak říkáme instance algoritmického problému. Vyluštit křížovku z New York Times z 3. března 1986 je instancí algoritmického problému, který bychom mohli nazvat „Vyluštit křížovku“. Tedy, abychom přesně specifikovali algoritmický problém, musíme určit všechny jeho instance, a pro každou z instancí řešení, které je správné. Přirozeně, nutnost přesné specifikace vstupů i výstupu je motivována hlavně tím, že chceme řešit pomocí počítače, který není schopen interpretovat vágní formulace. Bez přesné formulace správného řešení, zase nemůžeme ověřit zda je řešení spočtené počítačem správné a to ani mechanicky, ani formálně.

V následujícím se budeme držet zvyklostí z oblasti zabývajících se algoritmy, a z teoretické informatiky, zejména z teorie složitosti. Potřebujeme následující definice.

Definice 1. *Abeceda* Σ je konečná neprázdná množina. Prvkům Σ říkáme symboly. *Slovo (řetězec)* nad abecedou Σ je uspořádaná sekvence znaků z Σ. Délka slova $x \in \Sigma$ je délka této sekvence. Množinu všech slov nad abecedou Σ označujeme jako Σ^*. Jazyk L nad Σ je libovolná podmnožina Σ^*.

Příklad 1. Mějme $\Sigma_{Bool} = \{0, 1\}$. Sekvence 0001, 1110, 00 jsou slovy nad touto abecedou, sekvence 12, $\#43k$ nejsou. Σ^* je pak množina $\{0, 1, 00, 01, 10, 11, 000, \dots\}$.

Slova nad vhodně zvolenou abecedou použijeme pro reprezentaci instancí problému i pro reprezentaci správných řešení. Ačkoliv se tento přístup může zdát trochu nepřirozený, přináší řadu výhod. Poskytuje základ jednotného a přesného aparátu pro uvažování. Také do jisté míry koresponduje s realitou — data v počítači jsou sekvence jedniček a nul, tedy slova nad Σ_{Bool}. Poskytují také základ pro výpočet složitosti, velikost instancí pak můžeme měřit pomocí délky slov, které je reprezentují. Je důležité si uvědomit, že reprezentace instancí různými způsoby může vést k zásadně odlišným složitostem některých algoritmů (příkladem budiž nedávno objevený algoritmus pro testování prvočíselnosti). Analogickou situaci znáte z programování, volba vhodných datových struktur může mít zásadní vliv na rychlost běhu programu. Definice následují.

Definice 2. Nechť Σ je abeceda. Pak *algoritmický problém* definujeme jako dvojici $\langle L, R\rangle$, kde $L \subseteq \Sigma^*$ je množina instancí daného problému, a $R \subseteq \Sigma^* \times \Sigma^*$ je relace popisující vztah mezi instancemi a správnými řešeními. Tedy pokud $\langle x, y\rangle \in R$, pak y je správným řešením x.

Během kurzu se nebudeme zabývat technickými podrobnostmi kódování instancí problémů, ani vstupů. Nicméně, můžeme tak učinit jenom předpokladem, že kódování bude zvoleno správně, i když jej neuvedeme. Chtěl bych upozornit, že tak činíme pouze proto, abychom se nemuseli zabývat technickými detaily, které zbytečně komplikují výklad. Při popisu konkrétních algoritmických problémů tedy nebudeme uvádět instance a řešení problému pomocí slov nad příslušnými abecedami, ale jako konkrétní objekty (čísla, grafy apod).

Příklad 2. Pro ukázku si problém testování prvočíselnosti uvedeme ve verzích bez kódování pomocí řetězců i s jeho kódováním.

(a) Bez použití řetězců můžeme problém zavést následovně. Množina instancí je dána $\{n \mid n \in N, n \geq 1\}$, správné řešení je ANO, pokud je n prvočíslo, jinak NE.

(b) Zavedeme-li kódování nad Σ_{Bool} takové, že zakódováním n je slovo skládající se z n znaků 1, kódováním ANO je slovo 1 a kódováním NE je slovo 0, pak je problém popsán pomocí dvojice $\langle L, R\rangle$ takové, že:

$L = \{1^n | n \in N, n \geq 1\}$, kde a^n je slovo složené z n znaků a, navíc $\langle 1^n, 1\rangle \in R$, pokud je n prvočíslo, $\langle 1^n, 0\rangle \in R$, pokud n není prvočíslo.

V literatuře jsou studovány dva běžně se vyskytující typy problémů, rozhodovací problémy a optimalizační problémy. U obou skupin můžeme vyjádřit vztah mezi instancemi a správnými řešeními jiným způsobem, než přesným popisem R.

Definice 3. Problém $\langle L, R\rangle$ je *rozhodovacím problémem*, pokud je množina všech možných výsledků dvouprvková, tj $|\{y \mid (x, y) \in R \text{ pro } x \in L\}| = 2$. Jeden z prvků této množiny pak označujeme jako rozhodnutí ANO, druhý z prvků jako rozhodnutí NE.

Rozhodovací problém je problémem zjištění, jestli daná instance má požadovanou vlastnost. Pokud ji má, očekáváme odpověď ANO, pokud ji nemá, očekáváme odpověď NE. Protože R je v tomto případě jednoduché, můžeme problém reprezentovat pomocí jazyka. Slova, která kódují instance, pro které je odpověď ANO, do jazyka patří, slova, která kódují instance, pro které je odpověď NE, do jazyka nepatří.

Příklad 3. (a) Problém testování prvočíselnosti je rozhodovací problém.

(b) Typickým rozhodovacím problémem je problém splnitelnosti formulí výrokové logiky v konjunktivní normální formě (CNF). Připomeňme si, že formule je v CNF, pokud je konjunkcí *klausulí*. Klausule je disjunkcí *literálů*, z nichž každý je buď výrokovou proměnnou nebo její negací. Formule $(x \vee y) \wedge (y \vee z)$ je v CNF, kdežto formule $(x \wedge y) \vee (x \to y)$ není v CNF. Každou formuli výrokové logiky lze převést do CNF. Řekneme, že formule φ je splnitelná, pokud existuje ohodnocení výrokových proměnných takové, že φ je pravdivá. Například formule $(x \vee y)$ je splnitelná, protože pro ohodnocení $x = 1, y = 1$ je pravdivá. Oproti tomu formule $(x \wedge \neg x)$ splnitelná není, protože není pravdivá pro žádné ohodnocení proměnných. Problém splnitelnosti formulí výrokové logiky, který označujeme jako SAT, je problémem určení toho, zda je formule splnitelná. Problém je tedy určen jazykem {zakódování $\varphi \mid \varphi$ je splnitelná formule v CNF}.

Definice 4. Nechť Σ je abeceda. *Optimalizační problém* je čtveřice $\langle L, sol, cost, goal\rangle$, kde $L \in \Sigma^*$ je množina instancí daného problému, $sol : L \to \mathcal{P}(\Sigma^*)$ je zobrazení přiřazující instanci problému množinu vhodných řešení; $cost : L \times \mathcal{P}(\Sigma^*) \to \mathbb{Q}$ je zobrazení přiřazující každé instanci a vhodnému řešení této instance jeho cenu; *goal* je buď minimum (pak mluvíme o minimalizačním problému) nebo maximum (pak mluvíme o maximalizačním problému). Řekneme, že $y \in \Sigma^*$ je *optimálním řešením* instance $x \in L$, pokud $cost(x, y) = goal\{cost(y, x) \mid x \in sol(x)\}$.

Optimalizační problém je problémem výběru řešení s nejlepší cenou z množiny vhodných řešení dané instance. Vhodná řešení obdržíme pomocí zobrazení *sol*, cenu každého z nich určuje *cost*. Nakonec *goal* nám říká, jestli chceme najít to s největší nebo s nejmenší cenou.

Příklad 4. (a) Úloha batohu (KNAPSACK), jejíž definice následuje, je optimalizačním. problémem. Neformálně můžeme popsat KNAPSACK například následovně. Zloděj se vloupal do banky a chce si odnést zlatý písek. Má k dispozici batoh, který má omezený objem. Zlatý písek je v bance uskladněn v pytlících, které mohou mít různý objem. Každý z pytlíku může zloděj buď přesypat celý do batohu, nebo jej nesmí vůbec otevírat. Samozřejmě, cílem zloděje je odnést si v batohu co největší objem zlatého písku. Formálně je úloha batohu určena následovně:

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$sol(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leq b\}$
Cena řešení:	$cost(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Čísla w_i odpovídají objemům jednotlivých pytlů s pískem, b je kapacita batohu. Pro instanci $I = (b, w_1, \ldots, w_5)$, kde $b = 29, w_1 = 3, w_2 = 6, w_3 = 8, w_4 = 7, w_5 = 12$, existuje jediné optimální řešení $C = \{1, 2, 3, 5\}$ s cenou $cost(C, I) = 29$. Lze snadno ověřit, že všechna ostatní přípustná řešení mají menší cenu.

(b) Problém třídění, kterým jste se zabývali v ALM1, lze také formulovat jako optimalizační problém. Pro sekvenci prvků $a = a_1, \ldots, a_n$ a jejich uspořádání $\leq$ je počet inverzí definován jako $Inv(a) = |\{(a_i, a_j) \mid i <$

$j, a_i \not\leq a_j\}|$, tedy jako počet dvojic prvků, které jsou nejsou uspořádány ve správném pořadí. Pak je problém třídění definován následovně:

Problém třídění	
Instance:	$a = a_1, \ldots, a_n$, uspořádání $\leq$
Přípustná řešení:	$sol(a)$ = množina všech permutací prvků $a_1, \ldots, a_n$
Cena řešení:	pro $x \in sol(a)$ je $cost(a, x) = Inv(x)$
Cíl:	minimum

Tento kurz je hlavně o algoritmech. Překvapivě si ale neřekneme přesnou formální definici. Její vysvětlení totiž vyžaduje poměrně velký prostor, navíc neexistuje pouze v jedné formě. Existující definice se od sebe více či méně výrazně liší (lze ale dokázat určitou formu jejich ekvivalence). V následujícím si tedy popíšeme algoritmus neformálně.

Algoritmus je konečná procedura zapsaná ve vhodném symbolickém jazyce ve tvaru jednoznačných instrukcí (instrukce je chápána jako elementární krok, ne jako instrukce procesoru na počítači, i když instrukce procesoru je také elementárním krokem). Instrukce jsou prováděny za sebou v jednoznačném pořadí (tj. v každém kroku víme, která instrukce je následující) a k jejich provedení není potřeba žádná inteligence, intuice, lze je provádět strojově. Po skončení procedury dostaneme výsledek. Na algoritmus se můžeme podívat i z jiného úhlu: Algoritmus je konečná množina pravidel, pomocí kterých se dynamicky mění stav systému. Stav systému si lze představit jako množinu proměnných nebo kus paměti. Každá z instrukcí pak mění hodnotu jedné proměnné (nebo jedné paměťové buňky) v závislosti na tom, v jakém stavu se systém právě nachází. Existuje také pravidlo, které určí, že vývoj systému je u konce. Tento pohled na algoritmus je užitečný, protože zdůrazňuje vlastnosti, které nejsou z prvního pohledu patrné: Algoritmus potřebuje ke své činnosti prostor (paměť) a jednou instrukcí může změnit pouze limitovanou část daného prostoru.

Algoritmus danému vstupu (instanci nějakého problému) přiřazuje výstup. Z vnějšku jej lze tedy chápat jako zobrazení. Formálně zapíšeme fakt, že algoritmus A pro vstup x vrátí y jako $A(x) = y$. Řekneme, že A řeší problém $\langle L, R\rangle$ pokud $(x, A(x)) \in R$ pro každé $x \in L$, tedy pokud pro každou instanci problému vrací A správné řešení.

U optimalizačních problémů, pro které neznáme efektivní algoritmus pro nalezení optimálního řešení, se často spokojíme s algoritmem, který místo optimální řešení vrací řešení mu blízké, ale současně pracuje efektivně. Takový algoritmus nazýváme *aproximačním algoritmem.*

2 Úvod do časové složitosti

Algoritmus potřebuje ke svému běhu dva základní zdroje, prostor a čas. V této kapitole se budeme zabývat časovou složitostí, tedy zkoumáním toho, kolik času běh algoritmu zabere.

Časová složitost (dále jen složitost) algoritmu A je funkce $Time_A : \mathbb{N} \to \mathbb{N}$ přiřazující velikosti vstupní instance počet instrukcí, které musí algoritmus A během výpočtu provést. Protože obecně nemusí algoritmus pro dvě různé stejně velké instance provést stejné množství instrukcí, potřebujeme počty instrukcí pro instance stejné velikosti nějakým způsobem agregovat. Existuje celá řada přístupů, nejčastější jsou následující dva.

Definice 5. Označme si $t_A(x)$ počet instrukcí, které algoritmus A provede pro instanci $x \in L$. Časová složitost v *nejhorším případě* je definovaná

$$Time_A(n) = \max\{t_A(x) \mid x \in L, |x| = n\}.$$

Složitost v průměrném případě je pak

$$Time_A(n) = \frac{\sum_{x \in L, |x|=n} t_A(x)}{|\{x \mid |x| = n\}|}.$$

Složitost v nejhorším případě je typicky snazší nalézt. Důvodem je skutečnost, že nalézt $t_A(x)$ pro všechny instance potřebné délky je obtížné (obtížné může být i pouhé nalezení těchto instancí), kdežto „nejhorší případ“

se hledá snáze. Složitost v průměrném případě lze většinou odhadnout s použitím pravděpodobnostních metod (které jsou ale mimo rozsah tohoto kurzu).

Jedním z využití složitosti (mimo zřejmé využití k porovnání algoritmů) je rozdělení problémů na prakticky řešitelné a na praktický neřešitelné. Jak ovšem přiřadit složitost k problému?

Definice 6. Nechť U je algoritmický problém.

- $O(g(n))$ je *horním omezením složitosti* problému U, pokud existuje algoritmus A řešící U takový, že $Time_A(n) \in O(g(n))$.
- $\omega(f(n))$ je *dolním omezením složitosti* problému U, pokud pro každý algoritmus B řešící U platí, že $Time_B(n) \in \omega(f(n))$

V předchozí definici se vyskytuje asymptotická notace, kterou se zabývá následující kapitola. Čtenáři, kterému je tato notace neznámá, doporučuji, aby si následující kapitolu prošel před pokračováním ve čtení té současné.

Za horní omezení složitosti problému bereme složitost nejlepšího známého algoritmu. Nalézt dolní omezení složitosti je oproti tomu mnohem komplikovanější. Musíme totiž vyloučit existenci algoritmu s lepší složitostí než dolní hranicí. To je netriviální a pro naprostou většinu problémů je taková hranice neznámá (vyloučíme-li například hranici danou tím, že algoritmus musí přečíst celý vstup). Příkladem problému, pro který je tato hranice známá je například třídění porovnáváním, které má dolní omezení složitosti $O(n \log n)$.

Problémy rozdělujeme na prakticky řešitelné a prakticky neřešitelné pomocí jejich horního omezení složitosti. Rozdělení je následující

– problém považujeme za praktický řešitelný, pokud pro je jeho horní omezení složitost $O(p(n))$, kde $p(n)$ je polynom. Tedy, problém je praktický řešitelný, pokud pro něj existuje algoritmus s nejhůře polynomickou složitostí.

– ostatní problémy považujeme za prakticky neřešitelné.

Zamysleme se nyní nad důvody, proč zvolit za hranici praktické řešitelnosti zrovna polynomickou složitost. Důvodů je hned několik, uvedeme dva. Polynom je největší funkce, která „neroste rychle.“ U tohoto důvodu předpokládáme, že stupeň polynomu nebude extrémně velký (např. 100). U většiny známých algoritmů s polynomickou složitostí, není stupeň polynomu vyšší než 10 (možná i méně). Druhým důvodem je to, že polynomy jsou uzavřené vzhledem k násobení (vynásobením dvou polynomů dostaneme zase polynom). Pokud uvnitř algoritmu A spouštíme nejvíce polynomický počet krát algoritmus B s nejhůře polynomickou složitostí, má tento algoritmus také nejhůře polynomickou složitost (přitom samozřejmě předpokládáme, že pokud bychom považovali složitost algoritmu B při analýze A za konstantu, bude složitost A nejhůře polynomická). V podstatě tedy můžeme efektivní algoritmy skládat do sebe (analogicky volání funkcí při programování) a získaný algoritmus bude také efektivní.

Poznámka 1. V teorii složitosti označujeme množinu problémů, které jsou prakticky řešitelné, jako třídu **P**. Další významnou třídou problémů je **NP**. Do této třídy patří problémy, pro které existuje algoritmus, který ověří efektivně (tj. v polynomickém čase) správnost řešení. To znamená, že pokud máme řešení, umíme v polynomickém čase spočítat, jestli je toto řešení správné. Dobrý příkladem je luštění křížovky. Pokud by existoval polynomický algoritmus pro luštění křížovky, pak by tento problém patřil do **P**. Pokud by existoval polynomický algoritmus, který zkontroluje, zdali je křížovka vyplněna správně, patří luštění křížovky do **NP**. Je snadné dokázat, že $\mathbf{P} \subseteq \mathbf{NP}$. Stále však nevíme, platí-li inkluze v druhém směru (pokud ano, jsou obě třídy shodné). Za vyřešení tohoto otevřeného problému známého jako „**P** vs **NP** problém“ je vypsána odměna milion amerických dolarů a úspěšného řešitele čeká nehynoucí sláva.

3 Asymptotická notace

Vyjádřit (rozuměj zapsat jako vzoreček) složitost algoritmu je často komplikované a pracné, a navíc to ztěžuje její použití. Proto se při analýze algoritmů a uvažování o složitostech používá jejich zjednodušená reprezentace. Složitost algoritmu vyjadřujeme v tzv. asymptotické notaci. Zjednodušení spočívá v zanedbání částí, které vyjádření komplikují, a v ponechání toho nejdůležitějšího, což je popis hlavního trendu rychlosti růstu složitosti

(jako rychlosti růstu funkce). Asymptotická notace existovala a byla studována ještě předtím, než našla použití v analýze algoritmů. Toto použití navrhl a prosadil především Donald Knuth.

Definice 7. $O(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladné konstanty c a n_0 takové, že $0 \leq g(n) \leq cf(n)$ pro všechna $n \geq n_0$.

$\Omega(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladné konstanty c a n_0 takové, že $g(n) \geq cf(n) \geq 0$ pro všechna $n \geq n_0$.

$\Theta(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladní konstanty c_1, c_2 a n_0 takové, že $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$.

Zápis $g(n) = O(f(n))$ je příkladem toho, čemu Knuth říká jednostranná rovnost (v následujících úvahách lze zaměnit Ω a Θ za O). Rovnítko v ní interpretujeme jinak, než je zvyklé (což je zřejmé, na obou stranách rovnosti nejsou objekty stejného typu). Korektně by měl být vztah zapsán jako $g(n) \in O(f(n))$, nicméně použití rovnítka se již zažilo a stalo se de facto standardem.

Pokud $g(n) = O(f(n))$, tak funkce $g(n)$ roste od určitého n nanejvýš tak rychle jako $f(n)$. To, že jde o rychlost růstu a ne o konkrétní hodnoty funkcí, lze vypozorovat z použití konstanty c v definici. Vynásobením c můžeme totiž $f(n)$ posouvat po ose y nahoru a dolů, jak se nám hodí. Obdobná úvaha platí i pro $g(n) = \Omega(f(n))$, s tím rozdílem, že nyní $g(n)$ roste alespoň tak rychle jako $f(n)$. Pokud $g(n) = \Theta(f(n))$, obě funkce rostou stejně rychle.

Příklad 5. Čtenář může snadno ověřit následující

a) $x^2 + 1 = O(x^3)$

b) $x = \Omega(\log x)$

c) $x^3 + x = \Theta(x^3)$

d) $5 = \Omega(1)$ □

Pro pochopení významu rovností, ve kterých se asymptotická notace nachází na levé i na pravé straně, si stačí uvědomit, že výraz, ve kterém se tato notace nachází představuje množinu funkcí. Tedy například výraz $n^3 + O(n)$ je množina

$$\{n^3 + f(n) \mid f(n) \in O(n)\}.$$

Protože na obou stranách $=$ jsou množiny a jedná se o jednosměrnou rovnost, interpretujeme $=$ jako $\subseteq$. Pro aritmetiku s množinami funkcí platí intuitivní pravidla. Uvážíme-li množiny funkcí S a T, pak platí

$$\begin{aligned} S + T &= \{g + h \mid g \in S, h \in T\}, \\ S - T &= \{g - h \mid g \in S, h \in T\}, \\ S \cdot T &= \{g \cdot h \mid g \in S, h \in T\}, \\ S \div T &= \{g \div h \mid g \in S, h \in T\}. \end{aligned}$$

Skládání funkce s množinou funkcí provedeme obdobně,

$$f(O(g(n))) = \{f(h(n)) \mid h \in O(g(n))\}.$$

Například $\log O(n^2)$ je množina logaritmů z funkcí omezených shora kvadrikou. Pokud jsou asymptotické výrazy vnořeny, pak množiny sjednotíme, tedy například

$$O(g(n) + O(f(n))) = \bigcup \{O(h) \mid h \in g(n) + O(f(n))\}.$$

Příklad 6. Čtenář může snadno ověřit následující

a) $c \cdot O(f(n)) = O(f(n))$

b) $O(O(f(n)) = O(f(n))$

c) $O(f(n)) + O(f(n)) = O(f(n))$

d) $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$ □

Reference

[1] Donald Knuth. The Art of Computer Programming, Volume I. Addison-Wesley, 1997

[2] Donald Knuth. Selected papers on analysis of algorithms. Center for the study of language an information, 2000

[3] Cormen et. al. Introduction to algorithms. The MIT press. 2008.

[4] Donald Knuth Concrete mathematics: A foundation for computer science. Addison-Wesley professional, 1994

[5] John Kleinberg, Éva Tardes. Algorithm design. Addison-Wesley, 2005.

[6] U. Vazirani et. al. Algorithms. McGraw-Hill, 2006.

[7] Steve Skiena. The algorithm design manual. Springer, 2008.

[8] Juraj Hromkovič. Algorithmics for hard problems. Springer, 2010.

[9] Oded Goldreich. Computational complexity: a conceptual perspective. Cambridge university press, 2008.