Úvod do diskrétních struktur

část o algoritmech

Miroslav Kolařík

Obsah

1. O algoritmech
   - Intuitivně o algoritmech a jejich vlastnostech
   - K rekurzi
   - Konečné automaty
   - Složitost algoritmu

Obsah

1. O algoritmech
   - Intuitivně o algoritmech a jejich vlastnostech
   - K rekurzi
   - Konečné automaty
   - Složitost algoritmu

Intuitivní vymezení pojmu algoritmus

Algoritmus je přesný návod nebo postup, kterým lze vyřešit daný typ úlohy. Algoritmus obsahuje

1) hodnoty vstupních dat
2) předpis pro řešení
3) požadovaný výsledek, tj. výstupní data.

Pro zpřesnění pojmu algoritmus dodejme: je to předpis (konečný proces), který se skládá z uspořádané sady jednoznačných a proveditelných kroků a který zabezpečí, že na základě vstupních dat jsou poskytnuta požadovaná data výstupní.

Poznámky k vymezení pojmu algoritmus

Poznámka. Slovo „algoritmus“ pochází ze jména významného perského matematika al-Chorézmího (z 9. století našeho letopočtu).

Poznámka. Algoritmus bývá nahlížen také jako konečná posloupnost instrukcí pro vyřešení nějakého problému, které lze vykonávat mechanicky (jeho vykonávání tedy není podmíněno porozuměním tomu, proč a jak algoritmus funguje).

Poznámka. V analogii lze algoritmus přirovnat k „mlýnku na data“. Nasypeme-li do něj správná data a zameleme, obdržíme požadovaný výsledek. Přitom kvalita „mlýnku“ může být různá (o tzv. časové a paměťové náročnosti, viz dále).

Vybrané příklady algoritmů

- Erathostenovo síto (algoritmus na hledání prvních n prvočísel)
- Eukleidův algoritmus (na hledání NSD dvou přirozených čísel a jeho rozšířená verze, hledající navíc Bézoutovy celočíselné koeficienty)
- algoritmus na dělení mnohočlenů
- algoritmus na vyřešení kvadratické rovnice na množině $\mathbb{R}$
- algoritmus binárního vyhledávání
- Dijkstrův algoritmus (na hledání nejkratších cest v grafu)
- Kruskalův algoritmus (pro hledání minimální kostry grafu)
- algoritmy pro setřídění posloupnosti čísel (quicksort, mergesort, . . .)
- algoritmy na kompresi dat (LZ77, LZ78, LZW, Huffmanovo kódování, JPEG, MP3, . . .)
- šifrovací algoritmy (AES, DES, RSA, . . .)

Základní vlastnosti algoritmů

- **konečnost** (finitnost)
  každý algoritmus by měl skončit po konečném počtu kroků (v praxi je pochopitelně chtěno, aby požadovaný výsledek byl poskytnut v „rozumném“ čase, ne za milion let)
- **jednoznačnost** (determinovanost)
  každý krok algoritmu by měl být jednoznačně a přesně definován (v každé situaci by mělo být jasné, co a jak se má provést, jak má provádění algoritmu pokračovat)
- **obecnost** (hromadnost)
  algoritmus řeší celou třídu obdobných problémů (například jak obecně vypočítat součin dvou celých čísel) a neřeší jen jeden konkrétní problém (jak vypočítat kolik je $2 \cdot 6$).

K dalším typickým vlastnostem algoritmů patří:

- **rezultativnost** – algoritmus po zadání vstupních dat vrací nějaký výstup (například chybové hlášení, nebo třeba provede změnu parametrů v nějakém systému). (Algoritmus, který by nevydal žádný výsledek by byl v praxi k ničemu.)
- **korektnost** – je požadována a chtěna správnost algoritmem vydaného výsledku
- **opakovatelnost** – pro stejné vstupní údaje by měl algoritmus vracet stejné výsledky. (Při opakovaném řešení stejné kvadratické rovnice očekáváme totožné výstupy).[1]

[1] Existují výjimky: například v případě generování pseudonáhodných čísel by vracení jediného čísla nebylo žádoucí.

Poznámka: některé druhy algoritmů 1/2

- rekurzivní algoritmy – využívají (volají) sami sebe (později bude podrobněji)
- hladové algoritmy – k řešení se propracovávají po jednotlivých rozhodnutích, která jsou nevratná; například Kruskalův algoritmus pro hledání minimální kostry grafu
- algoritmy typu rozděl a panuj – dělí problém na menší podproblémy až po triviální podproblémy (které lze vyřešit přímo), dílčí řešení pak vhodným způsobem sloučí. (Jedná se o typický případ aplikace metody návrhu shora dolů.)
- pravděpodobnostní algoritmy – provádějí některá rozhodnutí náhodně či pseudonáhodně
- paralelní algoritmy – podstata tkví v rozdělení úlohy mezi více počítačů (respektive pro víceprocesorový počítač).

Poznámka: některé druhy algoritmů 2/2

- genetické algoritmy – pracují na základě napodobování evolučních procesů, postupným „pěstováním“ nejlepších řešení pomocí mutací a křížení
- algoritmy dynamického programování – postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů
- heuristické algoritmy – nekladou si za cíl nalézt nejlepší možné řešení; stačí jim nalézt dostatečně vhodné přiblížení. Používají se v situacích, kdy dostupné zdroje (nejčastěji čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné přesné algoritmy vůbec známy).

Poznámka: Jeden algoritmus může patřit zároveň do více skupin; například quicksort může být rekurzivní algoritmus typu rozděl a panuj.

Poznámka k reprezentaci algoritmů

Aby mohl být algoritmus vhodně reprezentován, je třeba stanovit úroveň podrobností jeho popisu (zejména s ohledem na to, komu je popisován). Laik potřebuje výrazně podrobnější popis než odborník.

V praxi je (v počítačové komunitě) pod pojmem **program**[2] obvykle označována formální reprezentace algoritmu, která je určena k tomu, aby ji realizoval počítač.

Reprezentace algoritmů vyžaduje nějakou formu jazyka. Základními „stavebními kameny“ jsou tzv. **primitiva**[3]. Souhrn přesně definovaných primitiv a pravidla umožňující tvoření složitějších prvků (z primitiv) tvoří **programovací jazyk**. Proces vývoje programu, jeho zakódování do kompatibilní podoby a vložení do stroje se nazývá **programování**.

---

[2] Aktivita provádění programu bývá označována jako **proces**.

[3] Každé primitivum má vlastní syntaxi (symbolickou reprezentaci) a sémantiku (zamýšlený význam).

Poznámka k přesné definici algoritmu

Definovat pojem algoritmus přesně je nemožné.[4] To si dobře uvědomovali průkopníci moderní informatiky (zejména Alonzo Church a Alan Turing). Turing proto definoval matematické modely jednoduchého univerzálního počítače (tzv. Turingovy stroje).

Churchova–Turingova teze

Každý algoritmus je možné realizovat nějakým Turingovým strojem.

Churchova–Turingova teze není věta, kterou by bylo možno dokázat v matematickém smyslu. Není totiž formálně definováno, co je to algoritmus. Jelikož je ale výpočet na Turingově stroji přesně definován, informatici jej všeobecně považují za definici pojmu algoritmus.

[4] Stejně tak nelze přesně vymezit třeba pojem stůl, viz přednáška.

Algoritmicky neřešitelné problémy

S přesně definovaným pojmem algoritmus (zavedeným přes Turingovy stroje) se můžeme ptát, co vše lze algoritmicky řešit. Velmi překvapivě (koncem třicátých let 20. století) objevil Church a nezávisle na něm i Turing, že existují jednoduché, přesně formulované výpočetní problémy, které nejsou Turingovými stroji řešitelné, tedy nejsou na počítači řešitelné žádným algoritmem.[5]

[5] Algoritmicky neřešitelných problémů existuje nespočetně mnoho, přičemž algoritmicky řešitelných je „jen“ spočetně mnoho.

Dva významné algoritmicky neřešitelné (nerozhodnutelné) problémy

- **Problém zastavení** (Halting problem.[a])
  Lze sestrojit algoritmus, který by o každém algoritmu uměl rozhodnout, zda jeho činnost skončí po konečném počtu kroků či nikoliv?
- **Problém rozhodnutí** (Entscheidungsproblem[b]).
  Existuje algoritmus, který by uměl rozhodnout, zda je dané matematické tvrzení v daném formálním jazyce pravdivé nebo nepravdivé?

---

[a] Turing v roce 1936 dokázal, že neexistuje obecný algoritmus, který by řešil problém zastavení pro všechny vstupy všech programů.

[b] Entscheidungsproblem je úloha, kterou poprvé předložil německý matematik David Hilbert roku 1928.

Obsah

1. O algoritmech
   - Intuitivně o algoritmech a jejich vlastnostech
   - K rekurzi
   - Konečné automaty
   - Složitost algoritmu

Rekurze

Rekurze[6] znamená sebeopakování. Velmi často se používá v matematice a informatice.

V programování rekurze představuje opakované vnořené volání stejné funkce (podprogramu); hovoříme o tzv. rekurzivní funkci. Každá rekurzivní funkce musí obsahovat ukončující podmínku, která určí, kdy se má rekurze zastavit. Po každém kroku volání sebe sama musí dojít ke zjednodušení problému. Pokud nenastane koncová situace, provede se rekurzivní krok.

[6] Zajímavou oblastí použití rekurze jsou fraktály – soběpodobné útvary, které mají na první pohled velmi složitý tvar, přestože jsou generovány opakovaným použitím jednoduchých pravidel.

O rekurzi jinými slovy (1/2)

Při popisu algoritmických struktur se často používají tzv. iterativní struktury, v nichž se sada instrukcí opakuje formou cyklu. Tzv. **rekurzivní struktury** představují významnou alternativu paradigmatu cyklů při implementaci opakovaných aktivit. Zatímco cyklus opakuje sadu instrukcí tak, že nejdříve sadu dokončí a poté ji znovu provede, při rekurzi se sada instrukcí opakuje jako podřízená úloha stejné sady.

Poznámka. Lze dokázat, že iterativní a rekurzivní struktury jsou z hlediska výpočetní síly ekvivalentní. Totiž, každý algoritmus využívající rekurzi lze přepsat do nerekurzivního tvaru při použití zásobníku. A také naopak, každý algoritmus používající struktury cyklu lze ekvivalentně přepsat do rekurzivní podoby.

O rekurzi jinými slovy (2/2)

Při provádění rekurzivní procedury, dochází k tzv. aktivaci této procedury. Tyto vnořené aktivace vznikají dynamicky a při postupu algoritmu nakonec opět mizí. V libovolném okamžiku může existovat více aktivací, pouze jedna z nich však aktivně postupuje. Ostatní aktivace čekají na ukončení jiné aktivace, aby mohly pokračovat.

Analogicky jako u řízení cyklů závisí rekurzivní systémy na testování koncové podmínky a jejich návrh musí zajistit, že tato podmínka bude dosažena. Rekurzivní procedura je obecně navržena tak, aby před požadavkem další aktivace testovala koncovou podmínku (základní případ). Pokud koncová podmínka není splněna, rutina vytvoří další aktivaci procedury a zadá jí úkol vyřešení upraveného problému, který je koncové podmínce blíže než úkol, který dostala aktuální aktivace. Jestliže je však aktivace splněna, procedura zvolí postup, který aktuální aktivaci ukončí bez vzniku dalších aktivací.

Možné dělení rekurze

Rozlišujeme dva základní typy dělení rekurze:

I. typ

1. **přímá rekurze** – nastává pokud podprogram volá přímo sám sebe
2. **nepřímá rekurze** – je situace, kdy vzájemné volání podprogramů vytváří „kruh“; například v příkazové části funkce *A* je volána funkce *B*, ve funkci *B* je volána funkce *C*, a ta volá funkci *A*

II. typ

1. **lineární rekurze** – nastává pokud podprogram při vykonávání svého úkolu volá sama sebe pouze jednou – vytváří se takto lineární struktura postupně volaných podprogramů
2. **stromová rekurze** – nastává pokud se funkce nebo procedura v rámci jednoho vykonávání svého úkolu vyvolá vícekrát (vzniklou strukturu je v tomto případě možné znázornit jako strom).

Humor o rekurzi

Některé definice rekurze v sobě zahrnují prvky humoru a parodie na výkladové slovníky:

Cyklus nekonečný
    viz Nekonečný cyklus

Nekonečný cyklus
    viz Cyklus nekonečný

Tento žertík v sobě nese i poučení o nesprávném užívání rekurze: je vidět, že zde chybí ukončující podmínka.

Příklady

Příklad

Nejčastějším příkladem rekurzivního postupu je výpočet faktoriálu $N!$, který lze pro $N \in \mathbb{N}_0$ spočítat dle vztahu $N! = N \cdot (N-1)!$, přičemž $0! = 1$.

faktorial(N):
    pokud $N <= 0$, potom vysledek $= 1$,
    jinak vysledek $= N *$ faktorial $(N-1)$

Ukázkou vhodného použití rekurze je průchod stromem. Také třeba rekurzivní verze Eukleidova algoritmu nebo třídícího algoritmu quicksort.

Poznámka: Nerekurzivní řešení dané úlohy může být efektivnější, většinou však ztrácí na přehlednosti. Vhodnou metodou je iterace:

Příklad

faktorial(N):
    pokud $N < 0$, potom konec,
    jinak
        vysledek $= 1$,
        pro i od 1 do N proveď
            vysledek $=$ vysledek $* i$,
konec

Poznámka: Pro programátora je rekurze velmi užitečný nástroj, je to však také nástroj nebezpečný a při jeho použití je třeba velká obezřetnost. Mechanické použití rekurze vede totiž poměrně často k algoritmům s exponenciální časovou složitostí, které jsou velice pomalé a prakticky použitelné jen pro malé vstupy.

Fibonacciho posloupnost je dobrou ukázkou, jak lze řešit rekurzivní úlohu různými a různě efektivními způsoby.

Příklad

Určete N-té Fibonacciho číslo pro dané přirozené číslo N, je-li $\mathrm{Fib}(1) = 0$, $\mathrm{Fib}(2) = 1$ a $\mathrm{Fib}(N+2) = \mathrm{Fib}(N+1) + \mathrm{Fib}(N)$ pro $N \geq 1$.
Nekonečná posloupnost Fibonacciho čísel tedy začíná čísly
0 1 1 2 3 5 8 13 21 34 55 89 144 ...
Toto řešení má však exponenciální časovou složitost a provádí se v něm velké množství zbytečných výpočtů.

Rozšíření rekurzivního algoritmu jen o jedno pomocné pole F přispěje k zásadnímu zrychlení výpočtu. Do tohoto pole jsou postupně ukládána všechna již jednou vypočítaná Fibonacciho čísla. Před každým rekurzivním voláním je zkontrolováno, zda by takové volání nebylo zbytečné (jestli už hledaná hodnota není v poli F). Takto jednoduše modifikovaný algoritmus má rázem lineární časovou složitost.

Poznámka: Příklad určení *N*-tého Fibonacciho čísla lze řešit také bez využití rekurze a bez pomocného pole. Postupuje se přes dvě proměnné, ve kterých jsou uchovávány poslední dvě hodnoty Fibonacciho posloupnosti. (Paměťová složitost je v tomto případě konstantní.)

Obsah

1. O algoritmech
   - Intuitivně o algoritmech a jejich vlastnostech
   - K rekurzi
   - Konečné automaty
   - Složitost algoritmu

Konečný automat

Konečný automat je teoretický výpočetní model primitivního počítače používaný v informatice pro studium vyčíslitelnosti a obecně formálních jazyků. Konečné automaty se používají pro zpracování regulárních výrazů, například jako součást lexikálního analyzátoru v překladačích a uplatňují se třeba také při návrhu sekvenčních logických obvodů.

Konečný automat

Konečný automat lze chápat jako abstraktní model určitého specifického typu výpočtu. Výpočet probíhá diskrétním způsobem se symboly. Konečný automat sestává z několika přechodů a z konečné (odtud název) množiny stavů, přičemž v danou chvíli se automat nachází právě v jediném ze svých stavů (v tzv. aktuálním stavu). Jeden z jeho stavů je tzv. počáteční (výchozí) a určitá podmnožina všech jeho stavů vymezuje tzv. koncovou množinu stavů. Na vstupu automat obdrží tzv. vstupní slovo, které se skládá ze symbolů, a které automat zleva doprava postupně čte. Právě na základě symbolů, které čte ze vstupu a na základě tzv. přechodové funkce[7] může mezi svými stavy přecházet. Pokud po přečtení celého slova skončí v jednom z koncových stavů, dané slovo přijímá (v opačném případě jej zamítá).

[7] Přechodová funkce definuje výše zmíněné přechody – přiřazuje danému (aktuálnímu) stavu a symbolu na vstupu stav následující.

Dříve než definujeme potřebné pojmy, vyřešíme dva následující příklady.

Příklad

Nechť $T = \{a, b, c\}$. Sestavte konečný automat, který rozpozná slova, která obsahují podřetězec *abc*.

Řešení: na přednášce.

Příklad

Nechť $T = \{0, 1\}$. Sestavte konečný automat, který přijímá regulární jazyk řetězců, které vyjadřují binární číslo dělitelné třemi (beze zbytku).

Řešení: na přednášce.

Definice

Abeceda T je libovolná neprázdná konečná množina symbolů. **Řetězec (slovo)** α je libovolná konečná posloupnost symbolů abecedy T. Neprázdný řetězec (slovo) $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ budeme stručně zapisovat ve tvaru $\alpha = \alpha_1\alpha_2\ldots\alpha_n$. **Prázdný řetězec (prázdné slovo)** budeme značit ε. Počet symbolů v řetězci (ve slově) je jeho **délkou** a označuje se $|\alpha|$. V souladu s tím $|\varepsilon| = 0$. Řetězec (slovo) α je **podřetězec (podslovo)** řetězce (slova) β, jestliže je „souvislou“ podčástí β. **Zřetězením** řetězců (slov) $\alpha = \alpha_1\alpha_2\ldots\alpha_k$ a $\beta = \beta_1\beta_2\ldots\beta_l$ rozumíme řetězec (slovo) $\alpha\beta = \alpha_1\alpha_2\ldots\alpha_k\beta_1\beta_2\ldots\beta_l$. V souladu s tím $\alpha\varepsilon = \alpha = \varepsilon\alpha$. Pro $n \in \mathbb{N}$ symbol α^n označuje **n-násobné zřetězení** α; pro $n = 0$ je $\alpha^n = \varepsilon$.

Definice

Pozitivní uzávěr T^+ abecedy T je množina všech neprázdných řetězců (slov) symbolů množiny T. **Uzávěr** T^* je množina $T^+ \cup \{\varepsilon\}$. Libovolná podmnožina uzávěru T^* je **(formální) jazyk L** nad (abecedou) T. Tedy $L \subseteq T^*$.

Příklad

Pro $T = \{0,1,2\}$, $\alpha = 101122$ a $\beta = 112$ je $|\alpha| = 6$ a $|\beta| = 3$. Dále $\alpha\beta = 101122112$, $\beta^3 = 112112112$ a $\alpha^0 = \varepsilon$. Přitom $|\alpha\beta| = 6+3 = 9$ a $|\alpha^0\beta^3| = 0+9 = 9$. Zřejmě β je podřetězcem (podslovem) α, ale ne naopak.

Příklad

- Pro $T = \{a\}$ je uzávěr $T^* = \{a\}^* = \{a^n; n \in \mathbb{N}_0\}$.
- Pro $T = \{a,b\}$ je pozitivní uzávěr $T^+ = \{a,b\}^+$ množina všech konečných neprázdných řetězců (slov) sestávajících výhradně ze symbolů a nebo b, například: a, bab, a^5, b^3a^2.

Příklad

- Konečným (formálním) jazykem L nad $T = \{a,b\}$ je třeba množina $L = \{\varepsilon, a, b, aa, ab, bb\} \subseteq T^*$.
- Pro $T = \{a,b,c\}$ je $L = \{a^n b^n c^n; n \in \mathbb{N}_0\} \subseteq T^*$ (formálním) jazykem nad abecedou T.
- $\mathbb{Z}$ je (formální) jazyk nad $T = \{-,0,1,2,3,4,5,6,7,8,9\}$.

Deterministický konečný automat (DKA)

Definice

Deterministický konečný automat[a] je pětice $\mathscr{A} = \langle T, Q, \delta, q_0, F \rangle$, kde

- T je konečná neprázdná množina vstupních symbolů (abeceda)
- Q je konečná množina stavů
- δ je tzv. **přechodová funkce**, $\delta : Q \times T \to Q$, popisující pravidla přechodů mezi stavy
- q_0 je **počáteční stav**, $q_0 \in Q$
- F je **množina koncových stavů**, $F \subseteq Q$.

[a] Existují i tzv. **nedeterministické konečné automaty** s přechodovou funkcí $\delta : Q \times T \to 2^Q$. Lze o nich dokázat, že mají stejnou výpočetní sílu jako deterministické konečné automaty.

Popis činnosti deterministického konečného automatu:

- Na počátku se automat nachází v počátečním stavu q_0 a na vstupu má nějaké slovo $w \in T^*$.
- Dokud není $|w| = 0$, tak se v každém kroku odebere nejlevější symbol x ze vstupního slova w a z aktuálního stavu q přejde automat (podle přechodové funkce) do stavu $\delta(q, x)$.
- Skončí-li automat po přečtení (zpracování) celého vstupního slova w v koncovém stavu, pak daný vstup přijímá (rozpoznává). V případě, že automat přečte celé vstupní slovo w a nenachází se v koncovém stavu, dané slovo nepřijímá (zamítá). Automat také slovo w nepřijímá, když pro aktuální stav q a odebraný symbol x neexistuje funkční hodnota $\delta(q, x)$, tedy, pokud není možný další přechod.

Poznámka. Konečný automat lze velmi názorně reprezentovat (mírně modifikovaným) orientovaným grafem – stavy jsou vrcholy, přechody jsou vyznačeny hranami ohodnocenými vstupními symboly a koncové stavy jsou vyznačeny dvojitým kroužkem.

Příklad

Nechť $T = \{0,1\}$ je abeceda. Graficky znázorněte DKA přijímající všechna slova, která obsahují lichý počet jedniček a libovolný počet nul.

Řešení. Pro vyřešení úlohy nám stačí uvažovat automat se dvěma různými stavy: s počátečním stavem q_0 a s jediným koncovým stavem q_1. Přechodová funkce δ je definována takto: $\delta(q_0,0) = q_0$, $\delta(q_0,1) = q_1$, $\delta(q_1,0) = q_1$, $\delta(q_1,1) = q_0$. Grafické znázornění na přednášce.

Příklad

Nechť množina $T = \{a, b, c, d\}$ je abeceda. Sestavte deterministický konečný automat, který přijme všechna slova, která obsahují podřetězec *dbca* nebo podřetězec *dbcd*.

Řešení. Pro vyřešení úlohy nám stačí automat s pěti stavy: q_0, q_1, q_2, q_3, q_4, kde q_0 je počáteční stav a q_4 je (jediný) koncový stav. Přechodová funkce δ je pak dána následovně:

$$
\begin{array}{lllll}
\delta(q_0,a)=q_0, & \delta(q_1,a)=q_0, & \delta(q_2,a)=q_0, & \delta(q_3,a)=q_4, & \delta(q_4,a)=q_4,\\
\delta(q_0,b)=q_0, & \delta(q_1,b)=q_2, & \delta(q_2,b)=q_0, & \delta(q_3,b)=q_0, & \delta(q_4,b)=q_4,\\
\delta(q_0,c)=q_0, & \delta(q_1,c)=q_0, & \delta(q_2,c)=q_3, & \delta(q_3,c)=q_0, & \delta(q_4,c)=q_4,\\
\delta(q_0,d)=q_1, & \delta(q_1,d)=q_1, & \delta(q_2,d)=q_1, & \delta(q_3,d)=q_4, & \delta(q_4,d)=q_4.
\end{array}
$$

Doporučený úkol. Daný DKA graficky znázorněte.

Definice

Množinu všech řetězců (slov) $w \in T^*$ přijímaných DKA nazveme **jazykem** $L(A)$ rozpoznávaným tímto automatem.

Příklad

Pro abecedu $T = \{a, b\}$ vytvořte DKA,

(a) který přijímá pouze dva řetězce: a, ba, tedy $L(A) = \{a, ba\}$.

(b) jehož $L(A) = \{a^n; n \in \mathbb{N}_0\} \cup \{(ba)^n; n \in \mathbb{N}\}$.

Řešení. Jednoduché.

Poznámka.

Množina všech řetězců (slov), které daný DKA přijme, tvoří tzv. **regulární jazyk**.

Obsah

1. O algoritmech
   - Intuitivně o algoritmech a jejich vlastnostech
   - K rekurzi
   - Konečné automaty
   - Složitost algoritmu

Již víme, že existují algoritmicky neřešitelné (nerozhodnutelné) problémy, pro které nemá smysl zkoušet algoritmy konstruovat. Příkladem je dříve zmíněný problém zastavení (halting problem). Jedná se o sestrojení algoritmu, který by o každém algoritmu uměl rozhodnout, zda jeho činnost skončí po konečném počtu kroků či nikoliv.

Redukcí z problému zastavení se dá ukázat nerozhodnutelnost celé řady problémů, které se týkají ověřování chování programů:

- Vydá daný program pro nějaký vstup odpověď „Ano“?
- Zastaví se daný program pro libovolný vstup?
- Dávají dva dané programy pro stejné vstupy stejný výstup?

Dále se budeme zabývat **algoritmicky řešitelnými problémy**, jejich časovými a pameťovými nároky. Bude nás tedy zajímat efektivita algoritmů z hlediska rychlosti výpočtu a velikosti potřebné (operační) paměti.

Složitost algoritmu

Složitost každého algoritmu může být studována buď z hlediska **paměťové náročnosti** nebo z hlediska **časové náročnosti**. Paměťovou náročností rozumíme požadavek na velikost paměti počítače, jež je zapotřebí k provedení výpočtu. Podobně časovou náročností rozumíme čas potřebný pro výpočet. Tento čas se obvykle neměří v časových jednotkách, ale počtem provedených elementárních kroků algoritmu.[8]

Poznámka. Dále se budeme věnovat pouze časové složitosti[9], neboť k paměťové složitosti se přistupuje analogicky.

---

[8] Jak víte, při studiu algoritmů se rozlišuje **časová složitost v nejhorším případě** a **časová složitost v průměrném případě**.

[9] Složitost je funkce závislá na velikosti vstupních dat algoritmu. U funkcí popisujících časovou složitost budeme uvažovat pouze jejich **řádovou velikost**, tedy například složitosti lišící se konstantním násobkem budeme považovat za stejné.

Notace „velké O“

Definice – řádové porovnávání funkcí

Nechť f, g jsou dvě funkce, které přiřazují přirozeným číslům reálná čísla. Pak řekneme, že funkce f **roste řádově nejvýše** jako funkce g (f je třídy $O(g)$), píšeme $f(n) \in O(g(n))$, právě když existují čísla $K > 0$ a $n_0 \in \mathbb{N}$ taková, že pro každé přirozené číslo $n \geq n_0$ platí $f(n) \leq K \cdot g(n)$.

Definice

Řekneme, že algoritmus má **polynomiální (polynomickou) časovou složitost**, právě když existuje polynom p takový, že $f(n) \leq p(n)$ pro všechna $n \in \mathbb{N}$.

Polynomiální jsou tedy všechny algoritmy, jejichž funkci časové složitosti můžeme shora omezit polynomem $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$, kde $a_k, a_{k-1}, \ldots, a_2, a_1, a_0 \in \mathbb{N}_0$ a také $k \in \mathbb{N}_0$.

Příklad

Ověřte, že platí

(a) $\log_2 n \in O(n)$

(b) $5n^3 + 3n^2 + 7 \in O(n^3)$.

Řešení.

(a) Podle definice je $\log_2 n \in O(n)$, pokud existuje konstanta $K > 0$ a $n_0 \in \mathbb{N}$ tak, že $\log_2 n \leq Kn$ pro každé $n \in \mathbb{N}$, $n \geq n_0$. V tomto případě stačí zvolit třeba $K = 100$ a $n_0 = 1$.

(b) Podobně, $5n^3 + 3n^2 + 7 \in O(n^3)$, pokud existuje kladná konstanta K a přirozené číslo n_0 tak, že $5n^3 + 3n^2 + 7 \leq Kn^3$ pro každé $n \in \mathbb{N}$, $n \geq n_0$. Z nerovnice

$$K \geq 5 + \frac{3}{n} + \frac{7}{n^3},$$

vidíme, že stačí například položit $K = 6$ a $n_0 = 4$.

Příklad

$3n^3 - n^2 + 2n \in O(n^3)$, neboť $3n^3 - n^2 + 2n \leq Kn^3 \Leftrightarrow$
$n^3(K-3) + n^2 - 2n \geq 0$, což pro $K = 4$ dává $n^3 + n^2 - 2n \geq 0 \Leftrightarrow$
$n(n+2)(n-1) \geq 0$, což platí $\forall n \geq n_0 = 1$.

Příklad*

Zjistěte, zda $n^2 \in O(n\ln^2 n)$.

Řešení. Poznamenejme, že asymptotickou notaci lze definovat i přes výpočet limity a to následovně:

$$f(n) \in O(g(n)) \quad \Longleftrightarrow \quad \lim_{n\to+\infty} \frac{f(n)}{g(n)} \in \langle 0; +\infty).$$

Platí:

$$\lim_{n\to+\infty} \frac{n}{ln^2 n} = +\infty,$$

odkud $n^2 \notin O(n\ln^2 n)$.

Příklad

Některé typické příklady časové složitosti (od nejrychlejší po nejpomalejší):

- $O(1)$ – **konstantní** (indexování prvků v poli)
- $O(\log N)$ – **logaritmická** (vyhledání prvku v seřazeném poli metodou půlení intervalu)
- $O(N)$ – **lineární** (vyhledání prvku v neseřazeném poli sekvenčním vyhledáváním)
- $O(N \log N)$ – **lineárnělogaritmická** (seřazení pole N čísel dle velikosti; třídění sléváním, třídění haldou)
- $O(N^2)$ – **kvadratická** (třídění N čísel dle velikosti; třídění přímým výběrem, bublinkové třídění)
- ...
- $O(2^N)$ – **exponenciální** (Fibonacciho posloupnost řešená pomocí stromové rekurze)
- $O(N!)$ – **faktoriálová** (řešení problému obchodního cestujícího hrubou silou).

Definice

Nechť $f, g : \mathbb{N} \to \mathbb{R}$ jsou funkce. Pak píšeme

- $f(n) \in O(g(n)) \Leftrightarrow \exists K > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq K \cdot g(n)$
  a říkáme, že funkce f roste řádově nejvýše jako funkce g.
- $f(n) \in \Omega(g(n)) \Leftrightarrow \exists k > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \geq k \cdot g(n)$
  a říkáme, že funkce f roste řádově aspoň jako funkce g.
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ a $f(n) \in \Omega(g(n))$
  a říkáme, že funkce f roste řádově stejně jako funkce g;
  nebo, že funkce f a g jsou řádově ekvivalentní (neboli asymptoticky ekvivalentní).

Příklad

Dokažte, že $\frac{n^2-1}{n+1} \in \Theta(n)$.

Řešení.

$$k \cdot n \leq \frac{n^2-1}{n+1} \leq K \cdot n$$

$$k \cdot n \leq n-1 \leq K \cdot n$$

$$k \leq 1 - \frac{1}{n} \leq K$$

Je to splněno například pro $k = \frac{1}{2}$, $K = 1$ a pro $\forall n \geq n_0 = 2$.

Příklad

Dokažte, že $3n^2 \in \Theta(n^2)$, přičemž $3n^2 \notin \Theta(n)$ a $3n^2 \notin \Theta(n^3)$.

Řešení. Jednoduché.

Uvažme počítač, u nějž provedení jedné instrukce trvá jednu nanosekundu. Následující tabulka ukazuje délky trvání výpočtu, spustíme-li na takovém počítači algoritmus o řádové složitosti $f(n)$ se vstupními daty velikosti n.

$f(n)$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$	$n = 1000$
n	$20ns$	$40ns$	$60ns$	$80ns$	$0,1\mu s$	$1\mu s$
$n \log n$	$86ns$	$0,2\mu s$	$0,35\mu s$	$0,5\mu s$	$0,7\mu s$	$10\mu s$
n^2	$0,4\mu s$	$1,6\mu s$	$3,6\mu s$	$6,4\mu s$	$10\mu s$	$1ms$
n^4	$0,16ms$	$2,56ms$	$13ms$	$41ms$	$0,1s$	$16,8min$
2^n	$1ms$	$16,8min$	$36,6let$			
$n!$	$77let$					

Předchozí tabulka potvrzuje oprávněnost představy: prakticky použitelný algoritmus je algoritmus s nejvýše polynomickou časovou složitostí.[10]

Předchozí představu rámcově potvrzuje i další tabulka, která popisuje, jak se zvětší rozsah zpracovatelných úloh v případě zvětšení výpočetní rychlosti použitého počítače 100x a 1000x, jestliže původně bylo možno v daném časovém limitu zpracovat vstupní data o velikosti $n = 100$.

[10] Nelze to však brát jako dogma. Viz následující dva příklady:

- $f_1(n) = 2^{100} \cdot n$,
- $f_2(n) = 2^{n^{0,0001}}$ ($= 2^{10}$ pro $n = 10^{10^4}$).

$f(n)$	zrych. výp. $1x$	zrych. výp. $100x$	zrych. výp. $1000x$
n	100	10000	100000
$n \log n$	100	5362	43150
n^2	100	1000	3162
n^4	100	316	562
2^n	100	106	109
$n!$	100	100	101

Z tabulek je vidět, že už pro algoritmy s exponenciální časovou složitostí je typická existence mezní velikosti vstupních dat, nad níž je úloha prakticky neřešitelná i při zvýšení rychlosti počítače o několik řádů.

Reference

- Část o složitosti byla zpracována s využitím textu
  P. Martinka: Základy teoretické informatiky, Olomouc 2006.