Složitost algoritmů
Operační a paměťová složitost
Operační složitost v průměrném, nejhorším a nejlepším případě
Asymptotická složitost

Obsah

Složitost algoritmů

Pod složitostí algoritmu rozumíme dobu prováděného algoritmu (časovou složitost) a rozsah použité operační paměti (prostorovou složitost). Složitost závisí na velikosti vstupních dat, proto ji můžeme popsat jako funkci *T(n)*, kde číslo *n* udává velikost vstupních dat. Například *T(n) = an + b*, kde *a*, *b* jsou nějaké konstanty, je zápis lineární časové složitosti (složitost roste lineárně s rostoucí velikostí vstupů). Obvykle je pro odhad složitosti důležitý pouze typ funkční závislosti a nikoliv přesné hodnoty konstant - ty se zanedbávají.

Za efektivní algoritmy považujeme takové postupy, jejichž složitost je polynomiální (např. n^{127}, nikoliv exponenciální 2^n). Provádění exponenciálních algoritmů či dokonce algoritmů o složitosti *T(n) = n!* může, už jen při malém navýšení velikosti vstupních dat, trvat i mnoho tisíců a miliónů let. Čím je algoritmus složitější, tím menší je vliv navýšení výkonu procesoru při velké velikosti vstupu. Ani polynomiální algoritmy s velkým stupněm polynomu nebo s velkými vstupními daty nejsou příliš rychlé.

V zápisu *T(n) = an + b* je *a* multiplikativní konstanta, která v sobě zahrnuje počet operací ve strojovém kódu, který je třeba provést pro vyřešení jedné operace na úrovni vyššího programovacího jazyka. Aditivní konstanta *b* udává pouze konstantní nárůst složitosti nezávislý na velikosti vstupních dat. Velikost této konstanty závisí jen na daném počítačovém systému, který algoritmus provádí. Při značné velikosti vstupních dat (to je situace, která nás vzhledem k složitosti algoritmu zajímá, při malých vstupech je složitost všech algoritmů poměrně nízká a vyrovnaná) můžeme konstanty *a* a *b* zanedbat, protože vzhledem k velikosti *n* jsou nepatrné, a výsledná složitost *T(n)* tak závisí především na velikosti *n* - velikosti vstupních dat algoritmu. Vybereme-li ze všech konstant *a* závisejících na použitém kompilátoru, který překládá program, největší konstantu, kterou označíme *c*, platí *T(n) ≤ cn* pro všechna dostatečně velká *n*. Tuto skutečnost značíme *T = O(n)*, čímž vyjadřujeme asymptotické chování funkce *T*. Z této úvahy vychází matematická definice pojmů horní a dolní odhad složitosti algoritmu a složitost v průměrném případě (očekávaná složitost).

Operační a paměťová složitost

Operační složitost udává počet operací (často bývá označována také jako časová složitost), které je potřeba vykonat při běhu algoritmu. Dává nám tedy náhled na to, jak dlouho bude daný algoritmus zpracovávat konkrétní data.
Paměťová složitost udává nároky algoritmu na operační paměť (můžete se rovněž setkat s označením prostorová složitost). Obě tyto vlastnosti by měly být a jejich hodnot stanoveny, nicméně vždy bude záležet na konkrétním případě použití algoritmu, která ze složitostí má větší význam v daném kontextu. Dokonce se může stát, že paměťová složitost je zanedbána úplně a zkoumá se pouze časová složitost. Složitost algoritmu lze zkoumat na několika úrovních. Může nás zajímat:

- **složitost v nejlepším případě**

  nedává příliš smysl se touto hodnotou zabývat z pohledu operační složitosti, protože nevypovídá nic o vlastnostech algoritmu při běžném použití. Paměťová složitost v nejlepším případě může hrát roli při výběru algoritmu pro specifickou oblast použití (můžeme vyřadit algoritmy, které v nejlepším případě spotřebovávají více paměti, než máme k dispozici).
- **složitost v nejhorším případě**

  tento údaj o složitosti algoritmu je už pro nás zajímavější. Určuje, co se bude dít v případě, že vstupní data mají tu nejhorší možnou podobu. Tento údaj je důležitý hned z několika důvodů. Tato složitost nás bude zajímat v případě, že potřebujeme algoritmus, který pro daná data skončí vždy do určitého času, případně potřebujeme vědět, kolik maximálně může algoritmus potřebovat paměti.
- **složitost v průměrném případě**

  určuje nároky algoritmu na počet operací resp. operační paměť pro průměrná data. Co jsou průměrná data, bývá často velmi obtížné určit, proto určit tuto složitost není úplně jednoduché. Je potřeba provést analýzu vstupních dat a na základě nich určit složitost v průměrném případě.
- **složitost asymptotická**

  tento údaj nám dává informaci o tom, jak se bude chovat algoritmus pro hodně velká vstupní data.
  V takovém případě můžeme zanedbávat koeficienty, které vstupují do výpočtu složitosti a vztahovat složitost pouze v závislosti k n bez ohledu na to jestli např. složitost v nejhorším případě bude n^2+1000 pak pro hodně velká n můžeme konstantu 1000 zanedbat, protože s rostoucím n nám tato hodnota významně neovlivňuje chování algoritmu. Asymptotická složitost nám tedy umožnuje klasifikovat algoritmy do skupin a určuje, jak se bude algoritmus chovat s rostoucím n, tedy jak se bude zvyšovat doba běhu algoritmu nebo paměťová složitost s rostoucí velikostí vstupních dat.

Horní odhad složitosti algoritmu nás zajímá nejvíce, protože nám udává složitost algoritmu v nejhorším případě. Někdy ovšem algoritmus dosahuje této horní složitosti jen ve vzácných případech, v takovém případě pak o složitosti algoritmu vypovídá lépe složitost průměrná.

***Definice horního odhadu složitosti*: Řekneme, že *f(n)* (složitost algoritmu) je asymptoticky menší nebo rovna *g(n)* (funkční závislost udávající složitost), píšeme *f(n) = O(g(n)),* právě když existuje taková kladná konstanta *c* tak, že pro každou velikost dat *n* od určité hodnoty n_0 platí: $0 \leq f(n) \leq cg(n)$, neboli:**

$$f(n) = O(g(n)) \Leftrightarrow \exists c > 0\ \exists n_0 > 0\ \forall n \in N\ n \geq n_0 : 0 \leq f(n) \leq cg(n)$$

Např. zápis $f(n) = O(n^2)$, popř. $T(n) = O(n^2)$ nám udává kvadratickou horní složitost algoritmu, zápis $T(n) = O(\log n)$ logaritmickou atd.

Dolní odhad složitosti nám určuje ideální složitost daného algoritmu (nejrychlejší možné provedení), která nastává jen pro určité případy vstupních dat. Pokud o algoritmu víme, že má dolní odhad složitosti n^2, je jasné, že výsledek tímto algoritmem nedostaneme rychleji než v kvadratickém čase (v závislosti na velikosti vstupních dat). Dokázat dolní odhad složitosti úlohy je většinou podstatně těžší, než dokazovat odhad horní, protože musíme dokázat, že neexistuje algoritmus, který by daný problém řešil rychleji.

***Definice dolního odhadu složitosti (obdoba minulé definice)*: Řekneme, že *f(n)* (složitost algoritmu) je asymptoticky větší nebo rovna *g(n)*, píšeme *f(n) = Ω (g(n))* (Ω je velké omega), právě když existuje taková kladná konstanta *c* tak, že pro každou velikost dat *n* od určité hodnoty n_0 platí: $0 \leq cg(n) \leq f(n)$, neboli:**

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c > 0\ \exists n_0 > 0\ \forall n \in N\ n \geq n_0 : 0 \leq cg(n) \leq f(n)$$

Např. zápis *f(n) = Ω (n^3)*, popř. *T(n) = Ω (n^3)* nám udává kubickou dolní složitost algoritmu.

Složitost v průměrném případě (očekávaná složitost) se počítá jako střední hodnota náhodné složitosti *T(n)* při nějakém rozložení vstupních dat. Někdy může být i řádově lepší než složitost v nejhorším případě.

***Definice asymptoticky (přibližně) stejné složitosti*: Řekneme, že *f(n)* (složitost algoritmu) je asymptoticky stejná *jako g(n)*, píšeme *f(n) = Θ (g(n))* (Θ je velké theta), právě když existují takové kladné konstanty c_1, c_2 tak, že pro každou velikost dat *n* od určité hodnoty n_0 platí: $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, neboli:**

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2 > 0\ \exists n_0 > 0\ \forall n \in N\ n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Např. zápis *f(n) = Θ (n)*, popř. *T(n) = Θ (n)* nám udává, že složitost algoritmu *f(n)* je asymptoticky stejná jako *Θ (n)*, tedy lineární.

K nízké složitosti algoritmu přispívá nejen optimální využití paměti, ale také vhodná volba datových struktur, protože volba uložení dat nám určuje také časovou a paměťovou složitost práce s těmito daty a ta může ovlivnit i výslednou složitost celého algoritmu.

Řešené vzorové příklady k lepšímu pochopení výše uvedeného:

10.
Pro rostoucí spojité fukce f(x), g(x) platí $f(x) \in \Omega(g(x))$. Z toho plyne, že

a) $f(x) \in O(g(x))$
b) $f(x) \in \Theta(g(x))$
c) $g(x) \in \Theta(f(x))$
d) $g(x) \in \Omega(f(x))$
e) $g(x) \in O(f(x))$

Řešení
$f(x) \in \Omega(g(x))$ znamená, že f(x) roste rychleji nebo stejně rychle jako g(x). Tudíž g(x) roste pomaleji nebo stejně rychle jako f(x). To vyjadřujeme zápisem $g(x) \in O(f(x))$. Platí možnost e).

11.
Pro rostoucí spojité fukce f(x), g(x) platí $f(x) \in O(g(x))$. Z toho plyne, že

a) $f(x) \in \Theta(g(x))$
b) $f(x) \in \Omega(g(x))$
c) $g(x) \in \Theta(f(x))$
d) $g(x) \in \Omega(f(x))$
e) $g(x) \in O(f(x))$

Řešení
$f(x) \in O(g(x))$ znamená, že f(x) roste pomaleji nebo stejně rychle jako g(x). Tudíž g(x) roste rychleji nebo stejně rychle jako f(x). To vyjadřujeme zápisem $g(x) \in \Omega(f(x))$. Platí možnost d).

12.
Pokud funkce f roste asymptoticky rychleji než funkce g (tj. $f(x) \notin O(g(x))$), platí následující tvrzení

a) jsou-li v bodě x definovány obě funkce, pak $f(x) > g(x)$
b) rozdíl $f(x) - g(x)$ je vždy kladný
c) rozdíl $f(x) - g(x)$ je kladný pro každé $x > y$, kde y je nějaké dostatečně velké číslo
d) obě funkce f i g jsou definovány jen pro nezáporné argumenty
e) nic z předchozího

Řešení
Řeč je o asymptotické složitosti, takže vztah funkcí f(x) a g(x) blízko nuly není nijak podstatný. Varianty a) b) a d) žádají z tohoto hlediska příliš mnoho a tedy neplatí. Tvrzení úlohy $f(x) \notin O(g(x))$ říká, že od určitého okamžiku musí platit $f(x) > g(x)$, což je ekvivalentní variantě c).

13.
Pokud funkce f roste asymptoticky stejně rychle jako funkce g (tj. $f(x) \in \Theta(g(x))$), platí právě jedno následující tvrzení. Které?

a) jsou-li v bodě x definovány obě funkce, pak $f(x) = g(x)$
b) ani poměr $f(x)/g(x)$ ani poměr $g(x)/f(x)$ nekonverguje k nule s rostoucím x
c) rozdíl $f(x) - g(x)$ je kladný pro každé $x > y$, kde y je nějaké dostatečně velké číslo
d) obě funkce f i g jsou definovány jen pro nezáporné argumenty
e) nic z předchozího

Řešení

Třídy složitosti

Složitosti můžeme rozdělit do několika tříd složitosti. Nás boudou zajímat především třídy Pa NP.

Třída složitosti P (polynomial)

P je obvykle považována za třídu problémů, které jsou efektivněřešitelné v polynomiálním čase. Jejich časová složitost je $O(n^k)$.

Třída složitosti NP (nedeterministicky polynomiální)

NP úlohy umíme prozatím řešit jen tak že řešení „uhádneme“ a ověříme. Není však znám žádný efektivní algoritmus řešení (avšak není vyloučeno, že existuje). Pro ověření „uhádnutého“ výsledku ovšem existuje algoritmus s polynomiální složitostí. To znamená, že pro získání jednoho výsledku potřebujeme v podstatě polynomiální čas. Pro získání všech řešení potřebujeme tedy součet těchto časů.

Nedeterministický algoritmus pro rozhodovací úlohy má obvykle dvě fáze:

> **nedeterministická fáze -do paměti se zapíše uhádnuté řešení deterministická fáze -použije se deterministický algoritmus pro určení, zda toto řešení představuje opravdu řešení zadané úlohy**

Pozn.: Víme že P patří do NP, ale to, zda P je podmnožina NP, nebo P = NP (tedy jestli pro řešení NP existuje nějaký efektivní polynomiálně složitý algoritmus), to je problém století.

Příklad:

Máme množinu čísel jako např. {-9, -6, -2, 5, 8} a chceme vědět jestli součet nějakých čísel z této množiny je nula. V tomto případě je odpovědí podmnožina {-2, -6, 8}. Vypadá to jednoduše, ovšem pokud zadáme velkou množinu čísel, náročnost problému rapidně vzroste. Žádný rozumný algoritmus zatím pro tento případ nebyl nalezen.

Na druhé straně, pokud z množiny vyjmeme zmíněnou podmnožinu, je snadné ověřit, jestli je její součet nula nebo ne. NP problémy mají společné to, že pro všechny existuje nějaký ověřovací algoritmus, který je časově polynomiálně složitý.

Problém B je NP-těžký, pokud pro libovolný problém A ze třídy NP platí, že A je polynomiálně redukovatelný na B.

Problém je NP-úplný, pokud patří do třídy NP a je NP-těžký.