Složitost

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítka, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat, že aritmetické operace, přiřazování, porovnávání apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bajtů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry algoritmu určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách.

Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Jednoduché počítání složitosti

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly posloupnosti, a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:

```
posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
  Jestliže posl[i] > max:
    max = posl[i]
Vypiš max
```

Není těžké nahlédnout, že algoritmus provede maximálně $N-1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N, protože porovnání dvou čísel nám zabere „jednotkový čas“ a paměťová složitost bude také na N záviset lineárně, protože každé číslo z posloupnosti budeme uchovávat v paměti. Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K:

```
Pro i = 1 až K:
  Pro j = 1 až K:
    Vypiš i*j a mezeru
  Přejdi na nový řádek
```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2. Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K.

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme N^k.

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N. Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N. To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N, na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména

ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S, podstatných jmen je A a přídavných jmen B.

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka). V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M, kde N je počet vrcholů grafu a M je počet jeho hran.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N.

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B o složitosti N^2. Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A, ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A.

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. *asymptotickou časovou složitost.*

Představme si, že máme algoritmus se složitostí $N^2/4 + 6N + 12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $N^2 + N + 1$.
- Pro velká N je $N + 1$ oproti N^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost N^2. Obecně škrtáme všechny členy, které jsou pro dost velké N menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor $\mathcal{O}$ (velké O), díky kterému budeme umět popsat, co přesně naše „škrtání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \to \mathbb{R}_0^+$ a $g : \mathbb{N} \to \mathbb{R}_0^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do nezáporných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(g)$.

A zde je použití: $n^2/4 + 6n + 12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\ldots)$. Někdy také říkáme, že se konstanty a méně významné členy v $\mathcal{O}$ ztrácí.

Poznámky

- Uvědomte si, že je notace nadefinovaná tak, že omezuje shora, takže nejen, že platí $n^2/2 \in \mathcal{O}(n^2)$, ale také třeba $n^2/2 \in \mathcal{O}(n^5)$. Na první pohled je to neintuitivní – můžeme tvrdit, že *QuickSort* běží v $\mathcal{O}(2^n)$ a mít pravdu. Copak by byl problém definovat tak, aby její význam nebyl „funkce až na konstanty shora omezená touto funkcí“, ale „funkce je až na (rozdílné) konstanty shora i zdola omezená touto funkcí“?

  Samozřejmě to vyjádřit můžeme! Definovat lze skoro cokoliv a existuje dokonce zaběhnutá notace $f \in \Theta(g)$, která tuto skutečnost (omezení zdola i shora) vyjadřuje. Samostatné omezení zdola (až na konstantu) se značí Ω a jeho definice je velmi podobná definici $\mathcal{O}$.

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu *BubbleSort* (bublinkovém třídění), o kterém se můžete dočíst v kuchařce o třídicích algoritmech. Funguje tak, že se dívá na všechny dvojice sousedních prvků, a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

```
BubbleSort(pole, N):
  Opakuj:
    setříděno = 1
    Pro i = 1 až N-1:
      Jestliže pole[i] > pole[i+1]:
        p = pole[i]
        pole[i] = pole[i+1]
        pole[i+1] = p
        setříděno = 0
  Skonči, až bude setříděno = 1
```

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám největší neseřazená hodnota „probublá“ na začátek hodnot, které jsou již na seřazené správném místě, a ostatní se posunou o jednu pozici doleva. Rozmyslete

si, proč. Průchodů je proto nejvýše N a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy nastat, pokud necháme setřídit klesající posloupnost. Tam provedeme přesně N průchodů (v posledním se jen ověří, zda je posloupnost seřazená).

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídicího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídicí algoritmus QuickSort (opět viz třídicí kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

- $\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)
- $\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do $\mathcal{O}$-čka“.
- $\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)
- $\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)
- $\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)
- $\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)
- $\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsat, dostaneme $\mathcal{O}(N \cdot 2^N)$)
- $\mathcal{O}(N!)$ – *faktoriálová,* $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ (nalezení všech permutací N prvků, tedy třeba všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální.* Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k. Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří například i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$ (každý algoritmus, který seběhne v čase $\mathcal{O}(\log N)$, seběhne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které dnes běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	100 ns	1 μs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 μs	20 ms
n^2	100 ns	400 ns	900 ns	100 μs	1 ms	1 000 s
n^3	1 μs	8 μs	27 μs	1 ms	1 s	10^9 s
2^n	1 μs	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Karel Tesař a Martin Mareš