Složitost algoritmů

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020





Osnova přednášky

Složitost algoritmů

- Motivace
- Analýza složitosti algoritmů

Motivace – kvalita řešení problémů

1. Máme navrhnout algoritmus potažmo software pro řešení problému.
2. Navrhneme algoritmus a implementujeme jej.
3. Máme zájem řešit problémy efektivně.
4. Naskýtá se otázka, jak kvalitní řešení problému jsme vlastně navrhli. Nejlepší? Nejhorší? Průměrné? Jak tedy měřit kvalitu řešení?
5. Kvalitu řešení budeme měřit pomocí tzv. **složitosti**. Složitost budeme chápat jako míru náročnosti našeho řešení na zdroje počítače.
6. Složitost problému budeme definovat jako složitost nejlepšího algoritmu řešícího daný problém.

Příklad

Problém – Vyvrtání díry do betonového panelu

Řešení – kávová lžička, ruční vrtačka, příklepová vrtačka, **vrtací kladivo s SDS upínáním vrtáku**

Motivace

1. **Algoritmus** je dosud chápán jako „mlýnek na maso“ – vložíme vstupní data, zatočíme klikou a dostaneme požadovaný výstup.
2. Je jasné, že různé mlýnky na maso se můžou kvalitativně lišit.
3. Máme zájem o vývoj kvalitních algoritmů a datových struktur.
4. Musíme tedy najít nástroj, který umožní měřit kvalitu algoritmů.

Nástroj pro měření kvality algoritmů se nazývá **složitost algoritmů**.

Pro měření složitosti algoritmů se okamžitě nabízí dvě měřítka:

- **prostor** – paměť potřebná pro výpočet a
- **čas** – doba běhu programu.

Motivace

1. V programech budeme používat i různé **datové struktury** – zásobníky, fronty, seznamy, množiny, tabulky atd.
2. Někdy si prostě nevystačíme s „proměnnou typu `int`“.
3. Datové struktury můžeme zjednodušeně chápat jako „skladiště“ informací.
4. Kvalitu datových struktur budeme měřit kvalitou operací pracujících s danou datovou strukturou, např. složitost nalezení prvku v tabulce, vložení prvku do tabulky, smazání prvku v tabulce atd.
5. Tímto získáme jednotný nástroj na měření kvality algoritmů a datových struktur.

Prostorová složitost algoritmu

- rozsah použité pracovní paměti (RAM či diskový prostor),
- lze ji jednoduše měřit v bajtech a násobcích,
- měření je poměrně přímočaré – lze provést jakousi inventuru programu,
- využívá stejné matematické postupy jako složitost časová.

Časová složitost algoritmu

- Doba běhu programu závisí typicky na mnoha faktorech. Jak tedy provést měření?
- Máme-li program řešící danou úlohu, můžeme na něm testovat jak dlouho program poběží pro různé velikosti vstupu a různé vstupy stejné velikosti.
- Můžeme sestrojit graf, jedna osa velikost vstupu, druhá osa doba běhu programu.
- Pozor nemusí to být funkce, pro jednu hodnotu na ose x může existovat více hodnot na ose y!
- Pro tutéž velikost vstupu může existovat více výsledků – nelze brát do úvahy jen prostou velikost vstupu.
- Například třídění posloupnosti čísel. Někdy stačí jen zkontrolovat, že posloupnost je již setříděná, jindy je potřeba ji celou „přeskládat“.
- Aby měl takový test smysl, musí být počet testů statisticky významný. Nelze dělat závěry z jednoho testu.

Časová složitost algoritmu

Obecně se dá říci:

- že doba běhu se bude zvyšovat s rostoucím velikostí vstupu a
- doba běhu bude záviset na HW a SW platformě, na které byly testy prováděny.

Úskalí experimentální analýzy časové složitosti programu

- Experimenty lze většinou provádět na omezeném vzorku dat (třídění 100 prvků, počet možných vstupů je 100!).
- Vybrat statisticky významný vzorek tak, aby poskytoval hodnověrné výsledky může být velice obtížné.
- Je velice problematické srovnávat měření na počítači A s operačním systémem B s počítačem C, který pracuje s operačním systémem D.
- A je také nutné algoritmus reálně implementovat a experimenty provést. Kdo to zaplatí?!

Časová složitost algoritmu

Metoda měření časové složitosti algoritmů:

- Musí brát do úvahy všechny možné vstupy.
- Lze srovnávat složitost dvou algoritmů nezávisle na HW a SW platformě
- analýzu algoritmu lze provést studiem formálního popisu algoritmu bez nutnosti provést detailní implementaci a provádět experimentální výpočty pomocí takto implementovaného programu

Časová složitost algoritmu

Po formální stránce směřujeme k tomu, že každému algoritmu přiřadíme funkci $f(n)$, která udává dobu běhu daného algoritmu vzhledem k velikosti vstupních dat n.
Doba běhu algoritmu – předpokládáme, že máme k dispozici hypotetický počítač, který přímo vykonává formálně popsané algoritmy.
Výsledkem našeho zkoumání je tvrzení typu
Algoritmus A pracuje s časovou složitostí úměrnou n^2.
Toto tvrzení znamená, že pokud bychom algoritmus implementovali a provedli měření doby běhu, tak pro žádný vstup nepřekročí doba běhu tohoto programu čas cn^2, kde c je konstanta závisející na HW a SW platformě.

Analýza složitosti algoritmů

Potřebné nástroje:

- měřítko (metrika) pro měření doby běhu algoritmu,
- model počítače, který bude vykonávat analyzované algoritmy,
- jazyk pro popis algoritmů,
- teoretický aparát pro zápis charakteristické doby běhu algoritmu.

Měření času

- jak změřit čas nezávisle na HW a SW platformě
- jak změřit čas teoreticky, bez implementace programu
- rychlost počítačů se postupně zvyšuje – zvyšuje se i rychlost (kvalita) algoritmů?
- dobu běhu algoritmu tedy nemůžeme měřit v obvyklých časových jednotkách
- dobu běhu algoritmu budeme měřit počtem provedených instrukcí

Primitivní operace

- nezávislé na programovacím jazyku
- předpokládáme vykonání v konstantním čase
- na různých HW a SW platformách budou časy vykonaní primitivních operací podobné
- získáváme odhad doby běhu programu na základě **počtu** primitivních operací

- přiřazení hodnoty do proměnné
- volání funkce
- provedení aritmetické operace
- porovnání dvou čísel
- indexování pole
- získání reference na objekt
- vrácení výsledku funkce

RAM stroj – Random Access Machine

- zjednodušený model počítače model počítače
- obsahuje jednoduchý procesor s jedním registrem (akumulátor)
- potenciálně neomezená paměť rozsahu $0 \ldots n$
- základní aritmetické operace
- přímé a nepřímé (pomocí obsahu akumulátoru) adresování paměti
- RAM stroj je ekvivalentní Turingovu stroji
- `http://en.wikipedia.org/wiki/RAM_machine`

Jazyk pro popis algoritmů

Nalezení maxima z posloupnosti čísel, zápis v C++

```
1  int Maximum(int[] a, int n)
2  {
3     int max = a[0];
4     for(int i = 0; i < n; i++)
5     {
6        if (a[i] > max)
7        {
8           max = a[i];
9        }
10    }
11    return max;
12 }
```

Počítání primitivních operací

1. inicializace proměnné max hodnotou `A[0]`,
2. dvě operace na začátku cyklu je proměnná `i` inicializována na 1,
3. jedna operace – před každým provedením cyklu je provedeno porovnání `i < n`, jedna operace n-krát,
4. tělo cyklu je provedeno $n - 1$ krát,
5. pokaždé je provedeno porovnání `max < A[i]`, což jsou dvě operace (indexování a porovnání)
6. pokud podmínka platí je provedeno přiřazení, což jsou dvě operace (indexování a přiřazení)
7. na konci těla cyklu se provede inkrementace proměnné `i`, což jsou dvě operace (přičtení a přiřazení)
8. celkově se v těle cyklu vykoná $4(n-1)$ až $6(n-1)$ operací
9. vrácení výsledku odpovídá jedné operaci

Počítání primitivních operací

Celkový počet primitivních operací je v nejlepším případě

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

v nejhorším případě je to

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$$

Nejlepší případ nastane pokud `A[0]` je maximem. Nejhorší případ nastane pokud jsou prvky v poli `A` setříděny vzestupně.

Nejlepší, nejhorší, průměrný případ

Z hlediska časové složitosti lze uvažovat tři případy:

nejlepší – není zajímavý, nic o podstatě algoritmu nevypovídá

průměrný – zajímavý, ale jde velice těžko analyzovat, je nutné brát do úvahy pravděpodobnosti výskytu jednotlivých možných vstupů

nejhorší – není potřeba prát do úvahy pravděpodobnost jak v předchozím případu, analyzuje se jednodušeji, v případě malé složitosti nejhoršího případu, je to pro nás dobrá zpráva

Budeme uvažovat vždy složitost v nejhorším případě, až na některé výjimky.

Notace „Velké O“

analýza komplexnějších algoritmů než je vyhledání maxima může být velice složitá
musíme se spokojit s odhadem
odhad by se od skutečné složitosti neměl lišit více než o nějakou multiplikativní konstantu
snaha detekovat hlavní faktory, části kódu, které ovlivňují složitost algoritmu
je nutný matematický aparát, který nám umožní skládat složitosti jednotlivých částí kódu a zahrnovat je pod nějaký jednotící celek

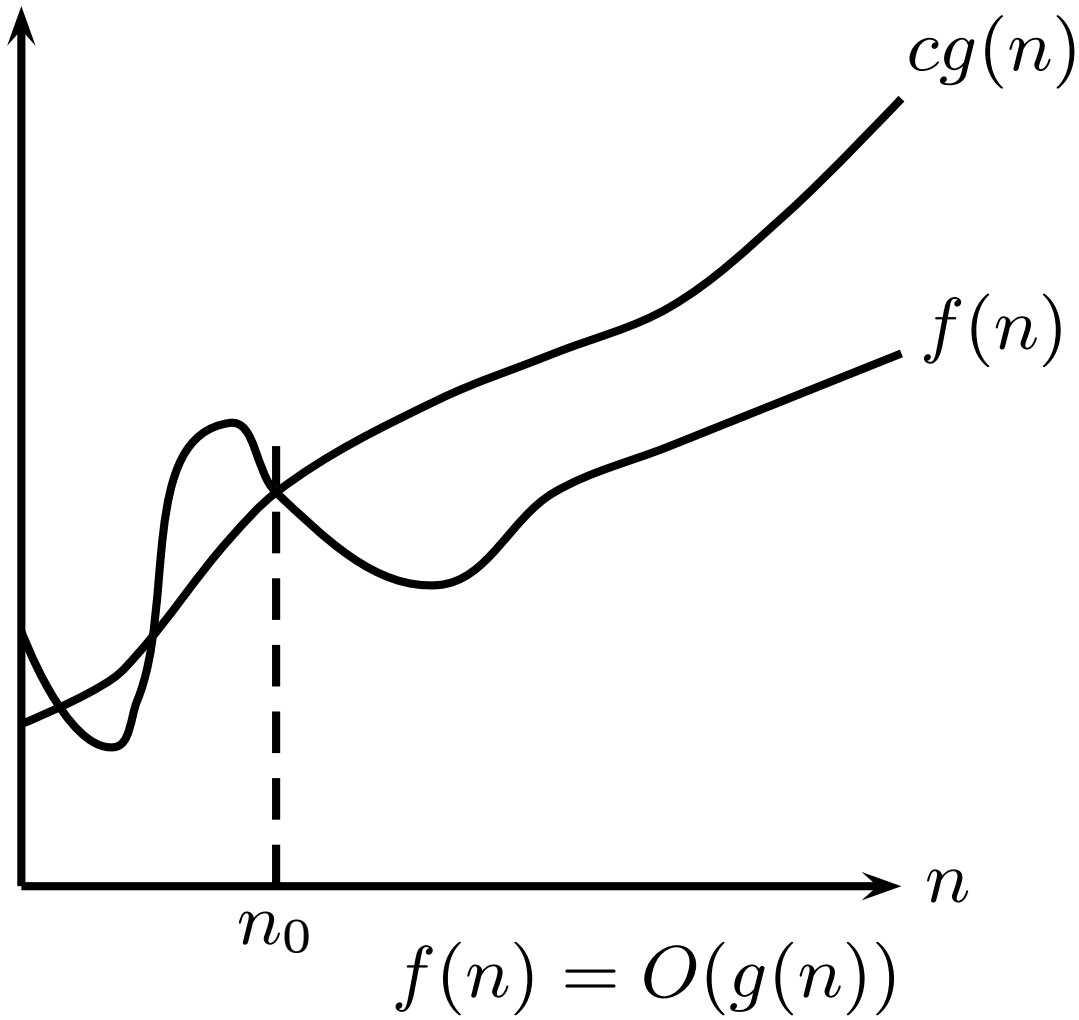
Notace „Velké O“

Definice

„Velké O“ Mějme funkce $f(n)$ a $g(n)$, kde $f(n), g(n) : N \to R$. Říkáme, že funkce $f(n)$ je $O(g(n))$, jestliže existuje reálná konstanta c, $c > 0$ a přirozené číslo $n_0 > 1$ takové, že $f(n) \leq cg(n)$ pro všechna $n \geq n_0$. Můžeme také říci, že f(n) je řádu g(n).

Tato definice nám umožňuje porovnávat funkce. Můžeme říci že jedna funkce „je menší než“ funkce jiná až na konstantní faktor (konstanta c z definice). Funkce lze takto porovnávat **asymptoticky**, čili kdy n roste k nekonečnu (požadavek pro $n \geq n_0$).

Notace „Velké O“



Notace „Velké O“ – příklad

Příklad

Výraz $7n - 2$ je $O(n)$.

Řešení

Podle definice „Velké O“ notace potřebujeme konstantu $c > 0$ a přirozené číslo $n_0 > 1$ takové, že $7n - 2 \leq cn$ pro všechna $n \geq n_0$. Je zřejmé, že lze vzít $c = 7$ a $n_0 = 1$.

Poznámka

Řešení je dokonce nekonečně mnoho, protože definici velkého O vyhovuje vzít za konstantu c libovolné reálné číslo větší nebo rovno 7 a za n_0 lze zvolit libovolné přirozené číslo větší nebo rovné 1.

Notace „Velké O“ – příklad

Časová složitost algoritmu vyhledání maxima v poli n čísel je $O(n)$.
Řešení: Víme že v nejhorším případě je pro vyhledání maxima potřeba $7n - 2$ primitivních operací a víme také že výraz $7n - 2$ je $O(n)$. Tudíž algoritmus vyhledání maxima v poli n čísel má složitost $O(n)$.

Význam asymptotického přístupu

Předpokládejme dále, že elementární operace vykonávané programy trvají $1ms$ a spočítejme, jak rozsáhlá data mohou jednotlivé programy zpracovat za sekundu, za minutu a za hodinu.

algortimus	složitost	1s	1min	1hod
$t_1(n)$	n	1000	$6 \cdot 10^4$	$3,6 \cdot 10^6$
$t_2(n)$	$n \log n$	140	4895	$2,0 \cdot 10^5$
$t_3(n)$	n^2	31	244	1897
$t_4(n)$	n^3	10	39	153
$t_5(n)$	2^n	9	15	21

Význam asymptotického přístupu

10-násobné zrychlení (nebo zvětšení doby)
lineární složitost – 10-násobným zvětšením rozsahu zpracovávaných dat
kvadratická složitost – 3-násobným zvětšením rozsahu zpracovávaných dat
exponenciální složitost – zvětšení rozsahu dat zhruba $\mathbf{o}\ 3,3$.
Dosažení rozsahu dat například $n = 50$ u algoritmu t_5 zrychlováním (nebo prodlužováním) výpočtu už vůbec nepřichází v úvahu.

Typické třídy asymptotických složitostí

1	konstantní
$\log n$	logaritmická
$\log^2 n$	
$\sqrt{n}$	
n	lineární
$n \log n$	lineárně logaritmická
n^2	kvadratická
n^3	kubická
n^k	obecně polynomiální
2^n	exponenciální

Třídy jsou uspořádány vzestupně.

Děkuji za pozornost