Digitální učební materiál

Číslo projektu:	CZ.1.07/1.5.00/34.0548
Název školy:	Gymnázium, Trutnov, Jiráskovo náměstí 325
Název materiálu:	VY_32_INOVACE_145_IVT
Autor:	Ing. Pavel Bezděk
Tematický okruh:	Algoritmy
Datum tvorby:	červenec 2013
Ročník:	4. ročník a oktáva
Anotace:	Algoritmus V. – Druhy výpočetní složitosti algoritmu
Metodický pokyn:	Při výuce nutno postupovat individuálně. Části DUM – „ Pro hloubavé“ jsou určeny pro zájemce o studium na technických a matematicko-fyzikálních oborech vysokých škol.

esf evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Autor	**Ing. Pavel Bezděk**		
Vytvořeno dne	**18. 7. 2013**		
Odpilotováno dne	**9. 12. 2013**	**ve třídě**	**8.Y**
Vzdělávací oblast	**Informatika a informační a komunikační technologie**		
Vzdělávací obor	**Informatika a výpočetní technika**		
Tematický okruh	**Algoritmus**		
Téma	**Algoritmus V. - Druhy výpočetních složitostí algoritmu**		
Klíčová slova	**Algoritmus, lineární složitost, kvadratická složitost**		

Druhy výpočetní složitosti algoritmů

Časová složitost

Časová složitost zkoumá **náročnost výpočtu v závislosti na velikosti argumentů**.

- **Konstantní složitost**

Algoritmy, jejichž výpočet trvá vždy konstantní počet kroků, mají **konstantní časovou složitost**.

Bez ohledu na velikost vstup. dat, vykoná algoritmus **konstantní počet kroků - k**.

- **Logaritmická složitost**

Existují i algoritmy, které **pracují déle než konstantně, ale kratčeji než lineárně**, jsou to **algoritmy s logaritmickou časovou složitostí - log(n)**.

Například když pro 5 vstupních prvků vykoná algoritmus 4 instrukce (kroky), pro 10 vstupních prvků vykoná algoritmus 5 instrukcí, pro 100 vstupních prvků vykoná algoritmus 11 instrukcí. Můžeme říct, že algoritmus má **logaritmickou** časovou složitost - **log(n)** .

- **Lineární složitost**

Algoritmus, jehož doba výpočtu **lineárně stoupá** s velikostí dat, má **lineární časovou složitost**.
Obecně tedy můžeme říct, že s každým dalším prvkem ve vstupních datech **vzroste časová složitost o konstantní počet kroků**.
Algoritmus má **lineární časovou složitost - n.**

- **Lineárně logaritmická složitost**

Můžeme mít i algoritmy pracující **déle než lineárně, ale méně než kvadraticky, logaritmy se složitostí n*log(n)**

- **Kvadratická složitost**

Předpokládejme, že porovnání dvou prvků trvá konstantní počet kroků. S každým prvkem seznamu se projdou ostatní prvky seznamu. Tedy se n-krát provede n kroků, kde **n** je délka seznamu **s**. Což nám dává složitost ***n*n***, neboli n^2. Budeme říkat, že algoritmus jehož výpočet trvá n^2 kroků, má **kvadratickou časovou složitost - n^2.**

- Dále ještě existují algoritmy, jejichž výpočet trvá n^3 kroků. Takové algoritmy mají **kubickou časovou složitost n^3 .**

Obecně o algoritmech se složitostí n^a, kde a je konstanta, říkáme, že mají **polynomiální časovou složitost - n^a .**

- **Exponenciální složitost**

Dále ještě existují algoritmy, jejichž **výpočet trvá a^n kroků (nejčastěji 2^n)**, kde **a** je konstanta nebo ***n* n* kroků.** Takové algoritmy mají exponenciální časovou složitost 2^n nebo n^n.

- **Faktoriální složitost**

Jestliže výpočet trvá n*(n-1)*(n-2)*(n-3)*……*4*3*2*1 kroků neboli **n!** (n-faktoriál), takové algoritmy mají faktoriální časovou složitost n! Je to složitost $a^n < n! < n^n$. Proto ji někdy řadíme mezi exponenciální složitosti.

Rychlé algoritmy - polynomiální časová složitost - n^a

Velmi rychlé algoritmy

- **Konstantní složitost** — **k** — **konstanta nebo číslo měnící se jen malém rozsahu s rostoucí velikostí dat**
- **Logaritmická složitost** — **log(n)**
- **Lineální složitost** — **n**
- **Lineárně logaritmická složitost** — **n*log(n)**

Pomalejší algoritmy

- **Kvadratická složitost** — n^2
- **Kubická složitost** — n^3

Velmi pomalé algoritmy

- nepolynomiální (exponenciální) časová složitost

- **Exponenciální složitost a^n např. 2^n**
- **Faktoriální složitost $n!$**
- **Exponenciální složitost n^n**

Jednoduché počítání složitosti

Časová a paměťová složitost dá určovat i intuitivně.

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.**

```
Program Maximum;
 const  N=10;
 type Pole = array[1..N] of integer;
 var  P: Pole;  Max,i: integer ;
   begin
        For i:=1 to N do begin  write(i,'.prvek pole: '); readln(P[i]);  end;
        Max:=P[1];
        For i:=2 to N  do
        begin   if P[i]>Max    then Max:=P[i]   end;
        writeln('Maximum= ',Max);
        repeat until keypressed;
   end.
```

Program maximální prvek pole v C++

```
#include <iostream>
//Maximalni prvek pole
using namespace std;
int i,maximum,pom;
const int index=10;
int pole[index];
int main()
{
 cout<<"Zadej prvky pole"<<endl;
 for (i=0;i<index;i++)
 {   cout<<"Zadej "<<i<<". prvek pole:";       cin >>pole[i];    }
 cout<< endl;
 cout<<"Zadali jste tyto prvky pole: "<<endl;
 for (i=0;i<index;i++)
 {   cout<<i<<". prvek pole: "<<pole[i]<<endl;   }
    cout << endl;
 pom=pole[0];
 for (i=1;i<index;i++)
 {   if (pom<pole[i]) pom=pole[i];   }
 maximum=pom;
 cout<<"Nejvetsi prvek pole je "<< maximum<<endl;
    return 0;
}
```

Časová a paměťová složitost hledání maxima

Algoritmus provede N −1 porovnání a max. N přiřazení.

Intuitivně **časová složitost bude lineárně záviset na N** , protože porovnání dvou čísel i přiřazení nám zabere jednotkový čas.

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.

Výpis násobků čísel od 1 do K

Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K: Uděláme jen zjednodušený zápis programu, který pro zjištění složitosti bude stačit:

```
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.**

Poznámka:
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.

Metoda „Kouknu a vidím“

Můžeme tuto metodu 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 máme **k cyklů a že každý běží od 1 do N** . Potom je **časová složitost** 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.

Pro hloubavé:

Složitost algoritmu v závislosti na více proměnných

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.

Častým příkladem, kde si **velikost vstupu potřebujeme rozdělit do více proměnných**, jsou **algoritmy pracující s grafy** . 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.

Použité zdroje

BÖHM, Martin. *Programátorská kuchařka: Recepty z programátorské kuchařky* [online]. Praha: KSP MFF UK Praha, 2011/2012 [cit. 2013-07-18]. KSP, Korespondenční seminář z programování: Programátorské kuchařky, 24. ročník KSP. Dostupné z: http://ksp.mff.cuni.cz/tasks/24/cook1.html