

# Analýza algoritmů. Složitost.

Matematické algoritmy (11MAG)

Jan Přikryl

Ústav aplikované matematiky  
ČVUT v Praze, Fakulta dopravní

7. přednáška 11MAG  
pondělí 11. listopadu 2013

verze: 2013-11-11 15:52



# Obsah přednášky

## ① Úvod

### Analýza algoritmů

## ② Vývojová stádia algoritmu

## ③ Porovnání variant

## ④ NP-úplné problémy



# Algoritmy a programy

Co je co

## Algoritmus:

- myšlenka řešení nějakého problému
- konečný počet kroků řešení
- vyjadřujeme nejčastěji slovně nebo pseudokódem

## Program:

- implementace algoritmu ve zvoleném programovacím jazyce

Bez algoritmu nelze napsat program.



# Analýza složitosti algoritmů

Proč nás to vlastně zajímá

Ideální kombinace: **efektivní algoritmus + efektivní implementace**

Analýza algoritmu:

- poskytne **předpověď výkonnosti** algoritmu
- je **vhodnější než experimenty**
- umožní výběr **vhodné varianty** řešení

Asymptotická složitost **paměťová** × složitost **časová**



# Analýza složitosti algoritmů (2)

## Předpověď chování

**Analýza efektivity:** Identifikace efektivních a neefektivních částí algoritmu umožní soustředit se na ty části, jejichž úprava přinese nejvyšší nárůst výkonu.

### Předpověď výkonnosti programu

Velké projekty potřebují apriorní odhad výkonnosti pro daný hardware – je třeba učinit odhad bez znalosti detailů programového kódu.

Identifikace úzkých míst a jejich vhodné ošetření ještě před vlastním naprogramováním.



# Analýza složitosti algoritmů (2)

## Předpověď chování

**Analýza efektivity:** Identifikace efektivních a neefektivních částí algoritmu umožní soustředit se na ty části, jejichž úprava přinese nejvyšší nárůst výkonu.

### Předpověď výkonnosti programu

Velké projekty potřebují apriorní odhad výkonnosti pro daný hardware – je třeba učinit odhad bez znalosti detailů programového kódu.

Identifikace úzkých míst a jejich vhodné ošetření ještě před vlastním naprogramováním.



# Analýza složitosti algoritmů (3)

## Ověření vlastností

### Vhodnější než experimenty

Experimenty ověří chování pouze ve vybraných krizových případech – místo „dokázali jsme, že daný algoritmus funguje správně“ lze pouze tvrdit: „**nenalezli jsme způsob, jak prokázat, že algoritmus je špatně**“.

Záruky funkčnosti poskytuje jedině **formální analýza**.

### Výběr vhodné varianty

Ne vždy je nejvhodnější varianta ta, jenž je v počtu instrukcí nejefektivnější a tedy nejrychlejší – *např. implementace pro jednočipový počítač × pracovní stanice*.



# Analýza složitosti algoritmů (3)

## Ověření vlastností

### Vhodnější než experimenty

Experimenty ověří chování pouze ve vybraných krizových případech – místo „dokázali jsme, že daný algoritmus funguje správně“ lze pouze tvrdit: „**nenalezli jsme způsob, jak prokázat, že algoritmus je špatně**“.

Záruky funkčnosti poskytuje jedině **formální analýza**.

### Výběr vhodné varianty

Ne vždy je nejvhodnější varianta ta, jenž je v počtu instrukcí nejefektivnější a tedy nejrychlejší – *např. implementace pro jednočipový počítač × pracovní stanice*.





# Obsah přednášky

## 1 Úvod

## 2 Vývojová stádia algoritmu

Algoritmus pomocí explicitního řešení

Rekurzivní algoritmus

Dynamické programování

Maticová varianta pomocí opakovaného mocnění

## 3 Porovnání variant

## 4 NP-úplné problémy



# Ilustrační příklad

## Fibonacciho posloupnost

Poprvé popsána italským matematikem Leonardem z Pisy, známým také jako Fibonacci (1202).

Růst populace králíků za poněkud idealizovaných podmínek.

Číslo  $F[n]$  popisuje velikost populace po  $n$  měsících, předpokládáme-li, že

- první měsíc se narodí jediný pár,
- nově narozené páry jsou produktivní od druhého měsíce svého života,
- každý měsíc zplodí každý produktivní pár jeden další pár,
- králíci nikdy neumírají, nemají predátory.



# Ilustrační příklad

## Stavy populace králíků

Fibonacciho číslo	Stav
$F[1] = 1$	začínáme s jedním párem
$F[2] = 1$	ještě jsou příliš mladí
$F[3] = 2$	tento měsíc již zplodí první potomky
$F[4] = 3$	druhý pár potomků
$F[5] = 5$	první potomci třetí generace

Obecně

$$F[n + 2] = F[n + 1] + F[n]$$

Problém (Výpočet Fibonacciho posloupnosti)

*Jak efektivně zjistit  $F[n]$  pro zvolené  $n$ ?*



# Explicitní řešení

## Přímé vyjádření $F[n]$

Explicitní nerekurzivní vztah pro  $n$ -tý člen Fibonacciho posloupnosti je

$$F[n] = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}},$$

kde  $\phi$  je hodnota zlatého řezu,

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,61803398874989 \dots \approx 1,618.$$

### Algoritmus 1

**Require:**  $n$

**Ensure:**  $F[n]$

$$F[n] \leftarrow \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$$



# Explicitní řešení

Přímé vyjádření  $F[n]$

Explicitní nerekurzivní vztah pro  $n$ -tý člen Fibonacciho posloupnosti je

$$F[n] = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}},$$

kde  $\phi$  je hodnota zlatého řezu,

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,61803398874989 \dots \approx 1,618.$$

## Algoritmus 1

**Require:**  $n$

**Ensure:**  $F[n]$

$$F[n] \leftarrow \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$$



# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?



# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?



# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?





# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?



# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?



# Analýza Algoritmu 1

Aneb co všechno může dopadnout špatně

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1,61803^n - 0,61803^n}{2,23606}.$$

Jaké budou výsledky?

$F[2] = 1,00000$  je v pořádku,

$F[3] = 1,78884$  zaokrouhlíme na 2,

$F[20] = 6764,69$  ještě zaokrouhlíme na 6765,

$F[21] = 10945,4$  už zaokrouhlíme na 10945 místo 10946,

$F[25] = 75020,6$  by mělo být 75025.

Existuje přesnější varianta výpočtu?



# Rekurzivní algoritmus

## Výpočet podle definice

Hodnoty  $F(0)$  až  $F(2)$  předpočítáme, zbytek lze s jejich pomocí vyjádřit.

### Algoritmus 2

**Require:**  $n \geq 0$

**Ensure:**  $y = F(n)$

- 1: **if**  $n = 0$  **then**
- 2:    $y \leftarrow 0$
- 3: **else if**  $n \leq 2$  **then**
- 4:    $y \leftarrow 1$
- 5: **else**
- 6:    $y \leftarrow F(n - 1) + F(n - 2)$
- 7: **end if**

Jak efektivní je daný algoritmus?



# Rekurzivní algoritmus

## Výpočet podle definice

Hodnoty  $F(0)$  až  $F(2)$  předpočítáme, zbytek lze s jejich pomocí vyjádřit.

### Algoritmus 2

**Require:**  $n \geq 0$

**Ensure:**  $y = F(n)$

- 1: **if**  $n = 0$  **then**
- 2:    $y \leftarrow 0$
- 3: **else if**  $n \leq 2$  **then**
- 4:    $y \leftarrow 1$
- 5: **else**
- 6:    $y \leftarrow F(n - 1) + F(n - 2)$
- 7: **end if**

Jak efektivní je daný algoritmus?



# Rekurzivní algoritmus

Výpočet podle definice

Hodnoty  $F(0)$  až  $F(2)$  předpočítáme, zbytek lze s jejich pomocí vyjádřit.

## Algoritmus 2

**Require:**  $n \geq 0$

**Ensure:**  $y = F(n)$

1: **if**  $n = 0$  **then**

2:    $y \leftarrow 0$

3: **else if**  $n \leq 2$  **then**

4:    $y \leftarrow 1$

5: **else**

6:    $y \leftarrow F(n - 1) + F(n - 2)$

7: **end if**

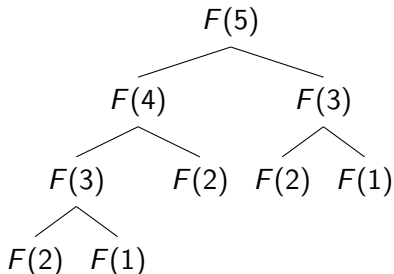
Jak efektivní je daný algoritmus?



# Rekurzivní algoritmus

## Efektivita

Posloupnost výpočtu  $F(5)$ :



Počet kroků pro  $F(n)$  je  $\tau(n) = 3 \cdot F(n) - 2$ .



# Dynamické programování

## Varianta „shora dolů“

Technika matematické optimalizace.

Dekompozice problému na identické podproblémy.

Dva základní přístupy:

- **shora dolů** – řešíme podproblémy postupně a pamatujeme si řešení
- **zdola nahoru** – vyřešíme všechny potřebné podproblémy a skládáme je

### Algoritmus 3

**Require:**  $n \geq 0$

**Ensure:**  $y = F[n]$

```
1: Alokuj  $[f_1, f_2, \dots, f_n]$ 
2:  $f[2] \leftarrow f[1] \leftarrow 1$ 
3: for  $i = 3$  to  $n$  do
4:    $f_i \leftarrow f_{i-1} + f_{i-2}$ 
5: end for
6:  $y \leftarrow f_n$ 
```





# Dynamické programování

## Varianta „shora dolů“

Technika matematické optimalizace.

Dekompozice problému na identické podproblémy.

Dva základní přístupy:

- **shora dolů** – řešíme podproblémy postupně a pamatujeme si řešení
- **zdola nahoru** – vyřešíme všechny potřebné podproblémy a skládáme je

### Algoritmus 3

**Require:**  $n \geq 0$

**Ensure:**  $y = F[n]$

```
1: Alokuj  $[f_1, f_2, \dots, f_n]$ 
2:  $f[2] \leftarrow f[1] \leftarrow 1$ 
3: for  $i = 3$  to  $n$  do
4:    $f_i \leftarrow f_{i-1} + f_{i-2}$ 
5: end for
6:  $y \leftarrow f_n$ 
```



# Dynamické programování

## Varianta „zdola nahoru“

Algoritmus 3 potřebuje pole  $n$  prvků pro uchování minulých členů posloupnosti.  
Jde to ale i bez něj.

### Algoritmus 4

**Require:**  $n \geq 0$

**Ensure:**  $y = F[n]$

```
1: if  $n = 0$  then  
2:    $y \leftarrow 0$   
3: else  
4:    $a \leftarrow 1, b \leftarrow 1$   
5:   for  $i = 3$  to  $n$  do  
6:      $c \leftarrow a + b$   
7:      $a \leftarrow b, b \leftarrow c$   
8:   end for  
9:    $y \leftarrow b$   
10: end if
```



# Dynamické programování

## Varianta „zdola nahoru“

Algoritmus 3 potřebuje pole  $n$  prvků pro uchování minulých členů posloupnosti.  
Jde to ale i bez něj.

### Algoritmus 4

**Require:**  $n \geq 0$

**Ensure:**  $y = F[n]$

```
1: if  $n = 0$  then  
2:    $y \leftarrow 0$   
3: else  
4:    $a \leftarrow 1, b \leftarrow 1$   
5:   for  $i = 3$  to  $n$  do  
6:      $c \leftarrow a + b$   
7:      $a \leftarrow b, b \leftarrow c$   
8:   end for  
9:    $y \leftarrow b$   
10: end if
```



# Maticová varianta

## Jiná explicitní forma

Pro členy Fibonacciho posloupnosti platí také

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Použitím opakovaného mocnění snížíme počet kroků na  $\mathcal{O}(\log n)$ .

*Důkaz:* Indukcí pro  $n \rightarrow n + 1$ .



# Maticová varianta

Pomocí opakovaného mocnění

## Algoritmus 5

**Require:**  $n \geq 0$

**Ensure:**  $y = F(n)$

1: **if**  $n = 0$  **then**

2:      $y \leftarrow 0$

3: **else**

4:      $\mathbf{M} \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

5:      $\mathbf{M} \leftarrow$   
        $\text{matpow}(\mathbf{M}, n - 1)$

6:      $y \leftarrow M[0, 0]$

7: **end if**

## Algoritmus 5a

**Require:**  $n \geq 0, \mathbf{A}[2 \times 2]$

**Ensure:**  $\mathbf{B} = \text{matpow}(\mathbf{A}, n)$

1: **if**  $n > 1$  **then**

2:      $\mathbf{B} \leftarrow \text{matpow}(\mathbf{A}, n/2)$

3:      $\mathbf{B} \leftarrow \mathbf{B}\mathbf{A}$

4: **end if**

5: **if**  $n$  je liché **then**

6:      $\mathbf{B} \leftarrow \mathbf{A} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

7: **end if**



# Asymptotická složitost

Velká  $\mathcal{O}$  notace

Jakým způsobem se bude chování algoritmu měnit v závislosti na velikosti (počtu, objemu) vstupních dat?

Dva základní typy:

- **časová složitost** – vliv na dobu výpočtu
- **paměťová složitost** – nároky na operační paměť

Značíme:

- $\mathcal{O}(N)$  – lineární složitost,
- $\mathcal{O}(N^2)$  – kvadratická složitost,
- $\mathcal{O}(\log N)$  – logaritmická složitost.



# Asymptotická složitost

Velká  $\mathcal{O}$  notace

## Vliv asymptotické časové složitosti

Pro  $\mathcal{O}(N^2)$  má zdvojnásobení objemu vstupních dat za následek čtyřnásobnou dobu vykonávání algoritmu.

Pro  $\mathcal{O}(\log N)$  může mít čtyřnásobný počet dat na vstupu za následek dvojnásobnou dobu vykonávání algoritmu.

Pro  $\mathcal{O}(1)$  je doba vykonávání algoritmu nezávislá na velikosti vstupu.

## Vliv asymptotické paměťové složitosti

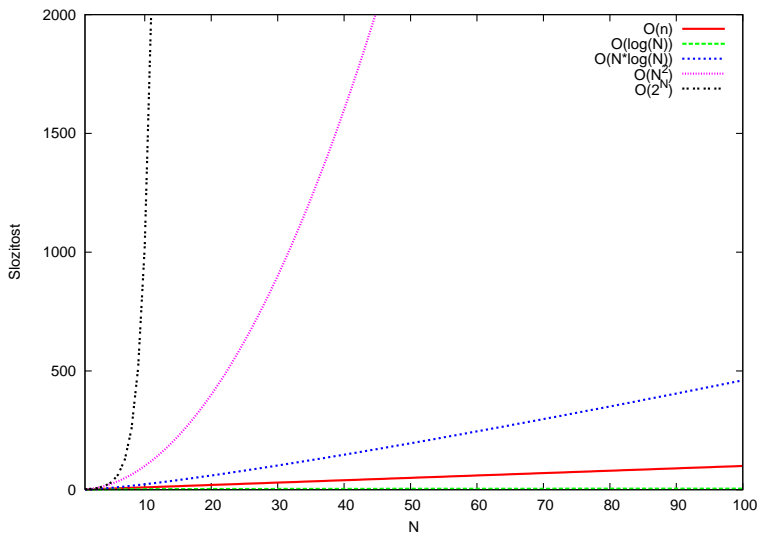
Pro  $\mathcal{O}(N)$  má zdvojnásobení velikosti vstupu za následek dvojnásob vysoké nároky na operační paměť.

Pro  $\mathcal{O}(2^N)$  čtyřnásobná velikost vstupu zosminásobí paměťové nároky.



# Asymptotická složitost

## Obrázek





# Obsah přednášky

- 1 Úvod
- 2 Vývojová stádia algoritmu
- 3 Porovnání variant**
- 4 NP-úplné problémy



# Porovnání variant

## Místo závěru

Algoritmus	Paměťové nároky	Časová složitost
1	$O(1)$	$O(\log N)$
2	$O(N)$	$O(F(N))$
3	$O(N)$	$O(N)$
4	$O(1)$	$O(N)$
5	$O(\log N)$	$O(\log N)$



# Obsah přednášky

- 1 Úvod
- 2 Vývojová stádia algoritmu
- 3 Porovnání variant
- 4 NP-úplné problémy



# Rozhodovací úlohy, třída P

Jako **rozhodovací úlohu** označujeme úlohu, jejímž řešením jsou výroky „ANO“ respektive „NE“. Jako výstup v případě počítače můžeme uvažovat hodnoty true a false nebo 1 a 0.

Běžné úlohy v matematice lze snadno převést na rozhodovací úlohy.

## Definice (Třída P)

Rozhodovací úloha  $L$  náleží do třídy P, pokud existuje *deterministický* Turingův stroj, který tuto úlohu rozhodne v polynomiálním čase.



# Příklady rozhodovacích úloh v třídě P

Minimální kostra grafu – existuje kostra s ohodnocením menším, než  $c$ ?

Nejkratší cesta v grafu – existuje cesta mezi dvěma uzly s ohodnocením menším, než  $c$ ?

Lineární programování – existuje  $\arg \max_{\mathbf{x}} \mathbf{w}^T \mathbf{x} > c$  za daných omezujících podmínek?

Kompresce dat (LZW) – přidá komprese řetězce  $s$  do slovníku slovo  $t$ ?



# Třída NP

## Definice

Rozhodovací úloha  $L$  náleží do třídy NP, pokud existuje *nedeterministický* Turingův stroj, který tuto úlohu rozhodne v polynomiálním čase.

**Nederministický Turingův stroj:** vstupům může odpovídat více, než jedna jediná akce (sekvence  $\Rightarrow$  strom instrukcí).

Platí  $P \subseteq NP$ .



# Příklady rozhodovacích úloh v třídě NP

Všechny úlohy třídy P.

Izomorfismus grafu – lze dané dva grafy nakreslit stejně?

Faktorizace čísel – pro dané  $n$  a  $k$ , existuje  $f : 1 < f < k, f \mid n$ ?

Všechny NP-úplné úlohy.



# Třída NP-úplných úloh

## Nejtěžší ze třídy NP

Třída NP-úplných problémů je třídou rozhodovacích úloh, pro něž platí následující definice:

### Definice

Rozhodovací úloha  $L$  je NP-úplná, pokud náleží do třídy NP a zároveň jde o úlohu NP- těžkou.

Co to znamená:

- jakékoliv řešení  $L$  lze ověřit v polynomiálním čase,
- Jakýkoliv problém z třídy NP lze převést na  $L$  transformací vstupů opět v polynomiálním čase.

Tyto typy úloh umíme řešit pouze **přibližně!**





# Příklady NP-úplných problémů

Problém batohu – lze zabalit batoh tak, aby jeho hmotnost nepřesáhla  $m$  a cena věcí byla alespoň  $c$ ?

Problém obchodního cestujícího – existuje v grafu hamiltonovská kružnice o délce nejvýše  $c$ ?

Obarvení grafu – lze uzly daného grafu obarvit nejvýše  $c$  barvami tak, aby sousedící uzly neměly stejnou barvu?

Problém čínského listonoše (pouze na smíšeném grafu) – existuje v grafu eulerovská kružnice o délce nejvýše  $c$ ?

