1. Eukleidův algoritmus *(zapsaly T. Klimošová a K. Böhmová)*

Problém hledání největšího společného dělitele

Cílem je najít největšího společného dělitele zadaných čísel $a_0, b_0 \in N$, dále jen $\gcd(a_0, b_0)$, tedy Greatest Common Divisor.

Algoritmus: (Eukleidův algoritmus verze 1.0; snad 200 let před Eukleidem)

1. *Vstup:* čísla a_0, b_0.
2. $a \leftarrow a_0, b \leftarrow b_0$.
3. while $a \neq b$:
4. if $a > b \Rightarrow a \leftarrow a - b$
5. else $\Rightarrow b \leftarrow b - a$
6. *Výstup:* $a = \gcd(a_0, b_0)$.

Důkaz správnosti:

1) EA se zastaví

V každém průchodu cyklem se buď a zmenší o hodnotu b anebo se b zmenší o hodnotu a. Proto součet $a+b$ klesne pokaždé alespoň o 1. Tedy celkem se uskuteční nanejvýš $a + b$ průchodů cyklem a pak algoritmus skončí.

2) EA vydá $\gcd(a_0, b_0)$

Při důkazech správnosti algoritmu se často používá *invariant.* To je něco, co se v průběhu algoritmu nemění (obvykle hodnota nějakého výrazu nebo jeho vlastnost, např. dělitelnost dvěma).

Dokážeme, že platí invariant: $\gcd(a, b) = \gcd(a_0, b_0)$.

Algoritmus při svém běhu postupně mění hodnoty a a b. Tedy $(a_0, b_0) \rightarrow (a_1, b_1) \rightarrow \ldots \rightarrow (a_k, b_k)$. A jakmile je splněna rovnost $a_k = b_k$, tak skončí. My chceme ukázat, že platí $\gcd(a_0, b_0) = \gcd(a_1, b_1) = \ldots = \gcd(a_k, b_k)$.

K tomu bude třeba několik pozorování:

Pozorování 1: $\gcd(a, a) = a$ (pro $a > 0$).

Nahlédneme snadno. Největší společný dělitel jediného čísla je ono samo.

Pozorování 2: $\gcd(a, b) = \gcd(b, a)$.

Taktéž jednoduché. Při určování největšího společného dělitele nám na pořadí čísel nezáleží.

Pozorování 3: $\gcd(a, b) = \gcd(a \pm b, b)$.

Dokážeme silnější věc – platnost ekvivalence $k|a, k|b \Leftrightarrow k|a, k|(a \pm b)$.

$\Rightarrow$: Jelikož $k|a, k|b$, dá se a vyjádřit jako k-násobek nějakého a'. Tedy $a = k \cdot a'$. Obdobně $b = k \cdot b'$, pro nějaká čísla $a', b' \in N$. Takže součet $a \pm b$ si můžeme rozepsat jako $(a' \pm b') \cdot k$ z čehož je již vidět, že $k|(a \pm b)$.

$\Leftarrow$: Podobně $k|(a \pm b), k|b \Leftrightarrow k|a$.

Uvědomíme si, že tato tři pozorování nám stačí k tomu, aby invariant platil. Algoritmus tedy opravdu vydá $\gcd(a_0, b_0)$. ♡

3) **Rychlost algoritmu**

Algoritmus provede v nehorším případě řádově $a+b$ iterací. To je velmi pomalé, a proto vyvstává otázka, dá-li se nějak zrychlit.

Všimněme si, že opakované odčítání jednoho čísla od druhého lze nahradit výpočtem zbytku po dělení. Algoritmus tedy mužeme upravit následovně:

Algoritmus: (Eukleidův alg. v2.0; neznámý Eukleidův pokračovatel, neznámo kdy)

1. *Vstup:* čísla a_0, b_0.
2. $a \leftarrow a_0, b \leftarrow b_0$.
3. while $b \neq 0$:
4. $\quad a \leftarrow a \bmod b$
5. $\quad a \leftrightarrow b$
6. *Výstup:* $a = \gcd(a_0, b_0)$.

Důkaz správnosti:

Algoritmus určitě vydá správný výsledek, protože počítá totéž, co předchozí verze. Ukážeme, že počet kroků, který provede verze 2.0, je řádově $\log(\max(a, b))$. K tomu se nám bude hodit následující:

Lemma: Za dva průchody cyklem se buď hodnota a nebo hodnota b zmenší alespoň dvakrát.

Důkaz: Rozebereme, co se stane po dvojím projití cyklem. Na začátku máme čísla a_0 a b_0. Bez újmy na obecnosti předpokládejme, že $a_0 > b_0$. Po prvním průchodu získáme $a_1 = b_0; b_1 = a_0 \bmod b_0$, což je méně než b_0. Po druhém průchodu: $a_2 = b_1; b_2 = a_1 \bmod b_1 = b_0 \bmod b_1$. Chceme ukázat, že když je $b_1 < b_0$, tak $b_0 \bmod b_1 \leq b_0/2$.

- Buď platí $b_1 \leq b_0/2$. Pak jistě $b_0 \bmod b_1 < b_1 \leq b_0/2$.
- Nebo nastane druhá možnost $b_1 > b_0/2$. Pak tedy $b_0 \bmod b_1 = b_0 - b_1$, což je taktéž maximálně $b_0/2$. ♡

Nyní stačí lemma šikovně využít: Dvojic průchodů, které zmenšují hodnotu a, je nejvýše $\lceil \log_2 a \rceil$. Obdobně těch snižujících hodnotu b je nejvýše $\lceil \log_2 b \rceil$. Celkový počet průchodů tedy je nejvýše $2(\lceil \log_2 a \rceil + \lceil \log_2 b \rceil)$.

Ještě bychom chtěli dokázat, že náš horní odhad na počet průchodů není příliš velkorysý. Zkusme se podívat, co se stane, když algoritmus necháme spočítat $\gcd(F_n, F_{n-1})$, kde F_i je i-té Fibonacciho číslo. Tehdy všechna modula budou ve skutečnosti odčítání a algoritmus bude počítat $\gcd(F_n, F_{n-1}) \to \gcd(F_{n-1}, F_{n-2}) \to \ldots \to \gcd(1, 1)$.

Díky exponenciální velikosti Fibonacciho čísel tento výpočet bude opravdu trvat logaritmicky dlouho vzhledem k velikosti většího ze zadaných čísel. (Zároveň je to také pěkný důkaz toho, že Fibonacciho čísla jsou navzájem nesoudělná.)

Výpočetní model

Když chceme studovat algoritmy, měli bychom se začít zaobírat otázkou, co takový algoritmus vlastně je. Žádná obecně uznávaná definice algoritmu neexistuje, nadefinujeme si proto alespoň výpočetní model a za algoritmy budeme považovat programy v tomto modelu.

Základní vlastnosti výpočetního modelu:

V první řadě potřebujeme modelu sdělit, s čím má pracovat, a pak se dovědět, jak to dopadlo. Bez újmy na obecnosti můžeme považovat n čísel za *vstup* a za *výstup* m čísel (můžeme se omezit na čísla, neboť cokoli jiného umíme do čísel zakódovat).

Dále by měl výpočetní model umět provádět *elementární operace*, což jsou nejen ty aritmetické ($+, -, *, /, \mathrm{mod}, \ldots$) a logické (negace, and, or, ...), ale také řídící operace (skoky, podmíněné skoky, halt, ...) a práci s pamětí.

Definice:

- *Čas běhu algoritmu* $t(x)$ pro vstup x měříme jako počet elementárních operací, které program provedl při zpracování vstupu x.
- *Prostor běhu algoritmu* $s(x)$ je analogicky počet paměťových buněk spotřebovaných při výpočtu se vstupem x.
- *Časová složitost* (v nejhorším případě) je:

$$T(n) := \max\{t(x); x \text{ je vstup délky } n\}.$$

- *Prostorová složitost* (v nejhorším případě) je:

$$S(n) := \max\{s(x); x \text{ je vstup délky } n\}.$$

Díra v definici: Co jsou to čísla? Když mohou být libovolně velká, pak se dá podvádět při počítání časové i prostorové složitosti. Napravit to můžeme buďto tak, že vstup budeme měřit v bitech a operace s k-bitovými čísly bude trvat řádově $\log k$ kroků, nebo že omezíme čísla na polynomiálně velká vzhledem k velikosti vstupu. Pro normální algoritmy tyto dvě definice dávají časové složitosti lišící se vždy logaritmem velikosti vstupu, takže pro srovnávání algoritmů nemusíme tento rozdíl uvažovat. Pro tuto přednášku si vybereme omezení čísel na polynomiálně velká.

Panorama složitostí

Abychom měli představu o tom, jak použitelné jsou algoritmy se kterou časovou složitostí, zkusme si pár složitostí srovnat za předpokladu, že náš počítač za sekundu provede 10^9 operací:

funkce / $n =$	10	20	30	50	100	1 000	1 000 000
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	5.6 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	50 ns	100 ns	1 μs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	147 ns	282 ns	664 ns	10 μs	20 ms
n^2	100 ns	400 ns	900 ns	2.5 μs	100 μs	1 ms	1 000 s
n^3	1 μs	8 μs	27 μs	125 μs	1 ms	1 s	10^9 s
2^n	1 μs	1 ms	1 s	10^6 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	10^6 s	10^{18} s	10^{32} s	10^{64} s	10^{158} s	10^{2567} s	$\approx \infty$

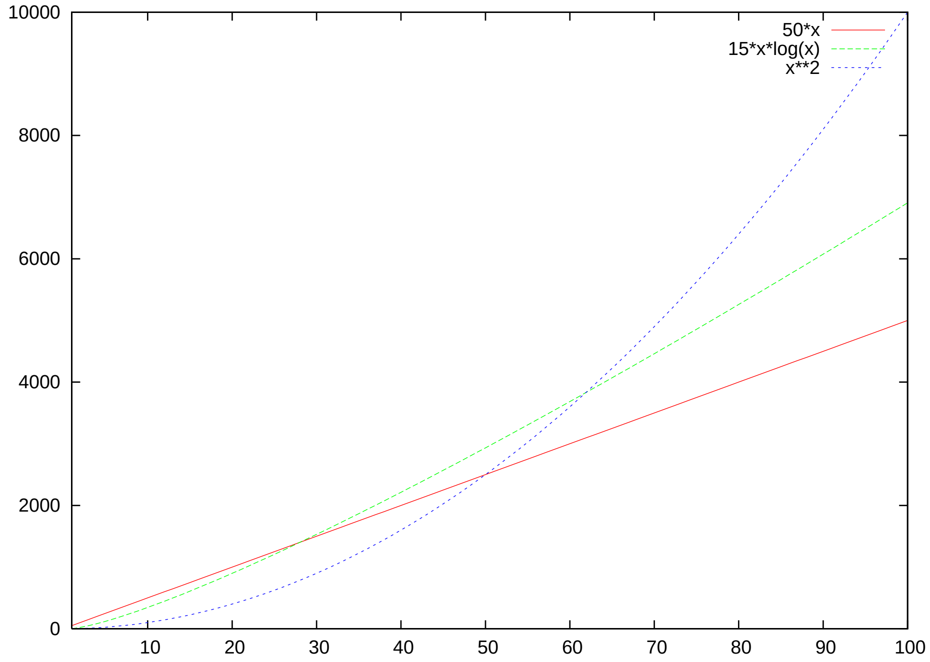
Pro představu: 1 000 s je asi tak čtvrt hodiny
1 000 000 s je necelých 12 dní
10^9 s je 31 let
10^{18} s je asi tak stáří Vesmíru.

Počítače se nám ale zrychlují[1] – není tedy lepší chvíli počkat než vymýšlet lepší algoritmus? :-) Podívejme se, jak velký vstup dokážeme zpracovat za 10^6 a $2 \cdot 10^6$ operací:

funkce	10^6 op.	$2 \cdot 10^6$ op.	změna
n	1 000 000	2 000 000	2-krát více
n^2	1 000	1 414	$\sqrt{2} \approx 1.41$-krát více
n^3	100	126	$\sqrt[3]{2} \approx 1.26$-krát více
2^n	20	21	o 1 více

Na některých časových složitostech se tedy exponenciální zrychlování počítačů projevuje jen minimálně.

Jak je vidět na následujícím grafu, konstanty v časové složitosti vůbec nehrají příliš velkou roli, proto se vyplácí najít nejdříve algoritmus, jehož složitost je asymptoticky dobrá, a teprve pak optimalizovat konstanty.



[1] „Každé dva roky se výkon počítačů zdvojnásobí“ – Gordon Moore (prý)

V praxi také často pomůže zpracovávat velké vstupy asymptoticky rychlým algoritmem a pro malé vstupy přepnout na „hloupý“ algoritmus, který má malé multiplikativní konstanty.