1. Úvod do algoritmů

Algoritmizace

Jiří Balun

Obsah

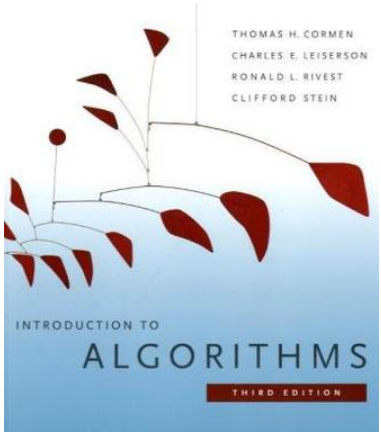
Úvod

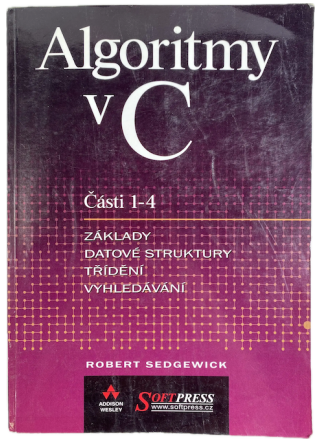
Organizační záležitosti

- **konzultační hodiny (kancelář 5.044):**
  - Pátek 8:00 – 9:00
  - případně v jiný čas po domluvě emailem
- **splnění předmětu:**
  - ústní zkouška + možná krátký test?
  - u ústní zkoušky si vylosujete otázku, poté budete mít čas na přípravu
  - cca 15 minut zkoušení
  - více se dozvíte na konci semestru

Literatura

- slajdy ke kurzu Algoritmizace z prezenčního studia od prof. Bělohlávka (odkaz **zde**)
- Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein, 2009)
- Algoritmy v C, části 1-4 (Sedgewick, 2003)





O kurzu 'Algoritmizace'

- jeden ze základních kamenů informatiky
- je součástí všech informaticky zaměřených oborů (MFF UK, ČVUT, VUT,...)
- slouží jako úvod pro další předměty jako např. „Základní algoritmy a datové struktury“
- za tento předmět je 6 kreditů, čemuž odpovídá jeho náročnost
- máme pouze 6 přednášek, proto část učiva připadne na samostudium
- je to teoretický předmět, ale všechny uvedené algoritmy si můžete sami naprogramovat

Proč studovat algoritmy?

Řešení praktických problémů

- algoritmy umožňují převést reálné úlohy na výpočet, který lze realizovat na počítači

Teoretický přínos

- studium algoritmů pomáhá porozumět hranicím toho, co je efektivně vypočitatelné

Efektivita

- různé algoritmy řeší tentýž problém s odlišnou časovou či paměťovou náročností

Škálovatelnost

- dobrý algoritmus umožňuje řešit velké instance problému, např. velké množství dat

Obecnost

- algoritmy nejsou vázány jen na informatiku, uplatňují se v ekonomii, medicíně, atd.

Inovace a konkurenceschopnost

- kvalita algoritmů často rozhoduje o úspěchu technologií a firem

Obsah kurzu

- základní pojmy: problém, algoritmus, pseudokód
- časová složitost algoritmu: v nejhorším a průměrném případě, O-notace
- základní datové struktury: pole, seznam, zásobník, fronta, strom
- metody řešení problému třídění
  - iterativní algoritmy: Insertion Sort, Selection Sort a Bubble Sort
  - rekurzivní algoritmy: Merge Sort, Quick Sort a Heap Sort
  - algoritmy nepoužívající porovnání: Bucket Sort, Radix Sort a Counting Sort
- pořádkové statistiky

Základní pojmy

Algoritmizace

Algoritmizace je proces tvorby postupu (algoritmu) pro řešení zadaného problému.

- v předchozím popisu je několik pojmů, které intutivně nějak chápeme, ale pro naše studium je potřeba je zpřesnit
- nejprve si tedy musíme odpovědět na tyto otázky:

Co je to algoritmus?

- Algoritmus je posloupnost instrukcí pro řešení problému.

Co je to instrukce?

- Instrukce je jednoznačný srozumitelný pokyn.

Co je to problém?

- Je přesně zadaná úloha, kterou určují vstupní data a k nim požadované výstupy.

Problém

Intuice: problém chápeme jako přiřazení, které každému přípustnému vstupu přirazuje odpovídající výstup.

Definice

Problém je specifikován:

1. množinou přípustných vstupů,
2. předpisem (z matematického pohledu jde o zobrazení), který pro každý přípustný vstup říká, jaké je odpovídající správné řešení (výstup).

- konkrétní hodnota vstupu se často označuje jako **instance** daného problému
- **rozhodovací** problém má výstupy jen "Ano" a "Ne"

Rozhodovací problém – test prvočíselnosti

Vstup: přirozené číslo n.
Výstup: "Ano", pokud je n prvočíslo, "Ne" v opačném případě.

Další příklady problémů

Problém – výpočet přepony pravoúhlého trojúhelníka

Vstup: dvě reálná čísla a, b.
Výstup: reálné číslo $c = \sqrt{a^2 + b^2}$.

Problém – třídění čísel

Vstup: posloupnost (nebo také n-tice) přirozených čísel $\langle a_0, \ldots, a_{n-1} \rangle$.
Výstup: permutace vstupní posloupnosti $\langle a'_0, \ldots, a'_{n-1} \rangle$ taková, že $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$.

Problém – hledání nejkratší cesty

Vstup: datová reprezentace mapy a GPS souřadnice dvou bodů na mapě.
Výstup: nejkratší cesta mezi vstupními body.

Instrukce

Intuice: instrukce je jednoznačný srozumitelný pokyn (nepřesná ale dostačující definice).

- předpokládáme, že existuje něco (například počítač) co instrukcím rozumí a je schopno je mechanicky (tj. bez dalšího přemýšlení) vykonávat
- tento vykonavatel instrukcí je schopen vykonávat instrukce tak, jak jsou předepsány algoritmem (tj. ve správném pořadí)

Příklady instrukcí:

- sečti čísla x a y (aritmetická instrukce)
- do proměnné x ulož číslo 5 (instrukce přiřazení)
- pokud je $x > 0$, pak zvyš hodnotu y o 1 (podmíněná instrukce)
- přečti číslo, které je na vstupu (vstupně/výstupní instrukce)
- vytiskni hodnotu proměnné x (vstupně/výstupní instrukce)
- pro každé $i = 1, 2, 3, 4, 5$ postupně proveď: vytiskni hodnotu i^2 (instrukce cyklu; v těle cyklu je vstupně/výstupní instrukce)

Algoritmus

Intuice: algoritmus řeší daný problém, pokud se pro každý vstup vykonáváním instrukcí podle algoritmu po konečném počtu kroků dobereme ke správnému výsledku.

Definice

Algoritmus je posloupnost instrukcí pro řešení problému.

Algoritmus řeší daný problém, pokud pro každý přípustný vstup I daného problému, jemuž odpovídá výstup O (tj. O je správný výstup pro vstup I) platí tyto podmínky:

1. vykonávání instrukcí podle algoritmu se vstupem I se po určité době zastaví (po konečném počtu kroků),
2. výstupem algoritmu je O.

- existuje řada podobných "definic" pojmu algoritmus, více či méně podrobných
- předpokládáme, že vstup I je na začátku vykonávání instrukcí zapsán na dohodnutém vstupním zařízení (soubor na disku, je zadán z klávesnice apod.)
- výstup O se objeví na dohodnutém výstupním zařízení (soubor na disku, obrazovka...)

Základní otázky o algoritmech

Správnost algoritmu:

- skončí navržený algoritmus pro každý přípustný vstup problému po konečném počtu kroků a se správným výstupem? Tj. jde vůbec o algoritmus řešící daný problém?
- chyby v algoritmech mohou mít vážné následky (např. řízení elektrárny atd.)

Složitost algoritmu:

- jak dlouho trvá vykonávání algoritmu?
- kolik paměti algoritmus potřebuje?
- složitost je jedno z měřítek, podle kterého můžeme porovnávat kvalitu algoritmů

Optimalita algoritmu:

- existuje lepší algoritmus?
- pro některé algoritmy lze dokázat, že lepší algoritmus neexistuje (a tedy nemá smysl hledat lepší algoritmus)

Popis algoritmů

Motivace: abychom mohli algoritmy studovat, musíme je nejprve umět nějakým srozumitelným způsobem popsat, přičemž se nabízí hned několik způsobů.

1. **přirozeným jazykem**
   - \+ snadno srozumitelné i laikům
   - – může být nejednoznačné a zdlouhavé
2. **zdrojovým kódem programovacího jazyka**
   - \+ jednoznačné
   - \+ z popisu je snadné vytvořit počítačový program
   - – obsahuje i zbytečné (nepodstatné) detaily
   - – nemusí být srozumitelné pro ty, kteří nemají zkušenost s daným programovacím jazykem
3. **pseudokódem** – jazyk blízký programovacímu jazyku, který je úspornější, protože neobsahuje tolik podrobností
   - \+ snadno pochopitelné i pro ty, kteří nemají zkušenosti s programováním
   - \+ je vytvořený přímo pro srozumitelný a úsporný popis algoritmů
   - \+ umožňuje snadný přepis do programovacích jazyků
   - – při implementaci algoritmu je třeba ho přepsat do programovacího jazyka
4. **existují další způsoby jako například vývojové diagramy atd.**

Popis přirozeným jazykem

- vhodné pro jednoduché a krátké algoritmy, u komplikovanějších algoritmů je tento popis nepřehledný a nejednoznačný
- nemá ustálené pojmenování pro instrukce, například "nastav proměnnou" a "ulož zpět do proměnné" je z našeho pohledu stejná operace přiřazení

Algoritmus pro výpočet n-té mocniny čísla x

1. Načti vstupní hodnoty: základ x a exponent n.
2. Nastav proměnnou $t \leftarrow 1$.
3. Opakuj n-krát:
   - vynásob t číslem x,
   - výsledek ulož zpět do t.
4. Po skončení opakování je v proměnné t uložen výsledek x^n.
5. Vrať jako výsledek hodnotu t.

Zdrojový kód

- jedná se o konkétní implementaci v nějakém programovacím jazyce
- nepřehledné, samotné řešení daného problému je skryté v technických detailech

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = * ( long * ) &y;   // evil floating point bit level hacking
  i  = 0x5f3759df - ( i >> 1 );  // what the f*ck?
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration
  return y;
}
```

Zdrojový kód: funkce pro rychlý výpočet inverzní druhé odmocniny z Quake III Arena.

Pseudokód

Proměnné

Intuice: v algoritmech si běžně potřebujeme uložit nebo pojmenovat nějakou hodnotu mezivýpočtu, k čemuž používáme tzv. **proměnné**.

- nebudeme řešit typy proměnných, tj. proměnná může nabývat libovolné smysluplné hodnoty, ať už je to číslo, logická hodnota, datová struktura...
- instrukce **přiřazení** slouží k uložení hodnoty do proměnné, značíme ji symbolem $\leftarrow$,
- časté pojmenování i, j pro indexy; m, n pro vstupní číselné hodnoty; A pro pole atd.

Příklady:

1: $i \leftarrow 42$
2: $j \leftarrow j + 1$
3: $n \leftarrow 0$
4: $m \leftarrow n - 666$
5: $temp \leftarrow$ `true`

Aritmetické instrukce

Intuice: máme k dispozici základní operace: $+$ (sčítání), $-$ (odčítání), $*$ (násobení), $/$ (dělení), $x \bmod y$ (zbytek po dělení čísla x číslem y), případně další.

- nebudeme řešit technickou stránku těchto operací, z našeho pohledu je každá taková operace jedna instrukce (přestože na konkrétním hardware je tomu jinak)
- například řádek 3 obsahuje: dvě instrukce násobení a po jedné instrukci sčítání a přiřazení (dohromady 4 instrukce)

Příklady:

1: $a \leftarrow 3$
2: $b \leftarrow 2 * 2$
3: $c \leftarrow (a * a) + (b * b)$
4: $r \leftarrow b \bmod a$

Logické hodnoty a instrukce

Intuice: máme dvě logické hodnoty `true` (pravda) a `false` (nepravda). Logické instrukce vrací logické hodnoty jako svůj výsledek, a v případě "and" a "or" je očekávají i na vstupu.

- and (konjunkce, logická spojka "a")
- or (disjunkce, logická spojka "nebo")
- $=$ a $\neq$ (test na rovnost a nerovnost)
- $<$ a $\leq$ (porovnání dle velikosti)

Příklady:

1. $i = 5$ bude mít hodnotu `true`, pokud hodnota proměnné i je 5, jinak `false`,
2. $(i = 5)$ and $(j < 10)$ bude mít hodnotu `true`, právě když oba výrazy $i = 5$ a $j < 10$ mají hodnotu `true`, jinak `false`,
3. $(i = 5)$ or $(j < 10)$ bude mít hodnotu `true`, právě když alespoň jeden z výrazů $i = 5$ a $j < 10$ má hodnotu `true`, jinak `false`.

Řízení programu

Intuice: řešení nějakého problému (nebo podproblému) budeme členit do samostatných algoritmů, viz níže je algoritmus pro výpočet druhé mocniny.

- algoritmus je definován svým jménem (zde např. square), vstupy a výstupy
- **return** ukončí vykonávání algoritmu a vrátí předanou hodnotu jako jeho výsledek

Algorithm square

Input $a \in \mathbb{N}$
Output a^2

1: **return** $a * a$

Poznámka: pokud máme takto definovaný algoritmus, můžeme jej používat při vytváření dalších algoritmů, zde například pomocí instrukce "square(a)", pro nějakou hodnotu a.

Větvení programu

Intuice: vykonání části kódu můžeme podmínit pomocí příkazu **if-then** a **if-then-else**.

if $condition$ **then** $expr_1 \ldots expr_k$

if $condition$ **then** $expr_1 \ldots expr_k$ **else** $expr'_1 \ldots expr'_\ell$

- $condition$ je logický výraz (musí tedy nabývat hodnotu `true` nebo `false`)
- výrazy $expr_1 \ldots expr_k$ (odsazená část kódu) se vykonají jen v případě, že podmínka $condition$ má hodnotu `true`, jinak se vykonají $expr'_1 \ldots expr'_\ell$ (v případě if-then-else)
- pro komplexnější podmínění můžeme přidat libovolný počet dalších **else if**–větvý

Algorithm min

Input $m, n \in \mathbb{N}$
Output minimum value from numbers m and n

```
1: if m < n then
2:     return m
3: else
4:     return n
```

Cyklus “for”

Intuice: příkaz cyklu použijeme v případě, že chceme nějakou část kódu vykonat vícekrát. Cyklus “for” má tuto syntaxi: **for** var **to** k **do** $expr_1 \dots expr_\ell$.

- var je tzv. **řídící proměnná** cyklu, která postupně nabývá hodnot od var do k (včetně)
- výrazy $expr_1$ až $expr_\ell$ tvoří **tělo** cyklu (odsazená část kódu), tj. kód který se zopakuje pro každou hodnotu var z daného rozsahu (jedno takové opakování je **iterace** cyklu)
- například algoritmus print-squares vytiskne (na obrazovku pomocí instrukce print) druhé mocniny čísel od 1 až do n, tj. pro $n = 4$ se vytisknou čísla 1, 4, 9 a 16

Algorithm print-squares

Input $n \in \mathbb{N}$
Output prints the squares of numbers 1 to n

1: **for** $i \leftarrow 1$ **to** n **do**
2: print(square(i))

Poznámka: řídící proměnnou je zde i, tělo cyklu tvoří pouze výraz na řádku 2.

Cyklus “while”

Intuice: příkaz cyklu “while” má tuto syntaxi: **while** $condition$ **do** $expr_1 \ldots expr_\ell$.

- tělo cyklu “while” se opakovaně vykonává dokud je splněna podmínka $condition$
- například v algoritmu power je podmínka $0 < n$, tj. tělo cyklu se opakuje, dokud je hodnota proměnné n větší než 0

Algorithm power

Input $x, n \in \mathbb{N}$
Output x^n

1: $temp \leftarrow 1$
2: **while** $0 < n$ **do**
3: $\quad temp \leftarrow temp * x$
4: $\quad n \leftarrow n - 1$
5: **return** $temp$

Poznámka: zde tělo cyklu tvoří výrazy na řádcích 3 a 4.

Analýza problému: největší společný dělitel

Příklad: Největší společný dělitel

Problém – nalezení největšího společného dělitele

Vstup: nezáporná celá čísla m a n, přičemž alespoň jedno z nich je větší než 0.
Výstup: $\gcd(m, n)$, tedy největší společný dělitel m a n (z angl. *greatest common divisor*).

- k dělí m (značíme $k|m$) právě tehdy, když existuje přirozené číslo ℓ takové, že $m = k\ell$
- $\gcd(m, n)$ je největší přirozené číslo, které dělí zároveň m i n (se zbytem 0)
- zbytek po dělení m číslem k budeme zapisovat $m \bmod k$, tj. operací modulo

Příklad

Výsledky $\gcd(m, n)$ pro konkrétní m a n:

- $\gcd(3, 5) = 1$
- $\gcd(3, 6) = 3$
- $\gcd(7, 0) = 7$
- $\gcd(12, 18) = \gcd(18, 12) = 6$
- a tak dále...

Naivní algoritmus

Intuice: začneme s $t = \min(m, n)$ a pokud $t|m$ a zároveň $t|n$, pak t je námi hledané číslo. Jinak snížíme t o 1 a celý proces opakujeme, dokud t není dělitelem obou čísel m i n.

Algorithm gcd-naive

Input $m, n \in \mathbb{N}$
Output greatest common divisor of m and n

1: $t \leftarrow \min(m, n)$
2: **if** $t = 0$ **then**
3: **return** $\max(m, n)$
4: **while** $m \bmod t \neq 0$ or $n \bmod t \neq 0$ **do**
5: $t \leftarrow t - 1$
6: **return** t

- řádky 2 a 3 řeší mezní případ, kdy $m = 0$ nebo $n = 0$ (nelze dělit nulou)
- podmínka na řádku 4 je neplatná (ukončí cyklus) právě tehdy, když $t|m$ a zároveň $t|n$
- algoritmus vždy zastaví, nejpozději pro $t = 1$, protože číslo 1 dělí každé celé číslo

Příklad: průběh algoritmu gcd-naive

Příklad

1: $t \leftarrow \min(m, n)$
2: **if** $t = 0$ **then**
3: **return** $\max(m, n)$
4: **while** $m \bmod t \neq 0$ or $n \bmod t \neq 0$ **do**
5: $t \leftarrow t - 1$
6: **return** t

Průběh algoritmu pro vstupy $m = 84$ a $n = 24$, tj. gcd-naive$(84, 24)$:

1. na řádku 1 se do t uloží 24,
2. podmínka na řádku 2 není splněna, a proto pokračujeme cyklem na řádcích 4-5,
3. podmínka na řádku 4 je splněna $(84 \bmod 24 \neq 0)$, proto se provede iterace cyklu,
4. při každé iteraci cyklu snižíme t o 1 a poté následuje opět ověření podmínky,
5. podmínka cyklu je zneplatněna až po 12 iteracích, konkrétně pro $t = 12$, jehož hodnota je vrácena jako výsledek gcd-naive$(84, 24)$.

Důkaz správnosti gcd-naive

Motivace: důkaz správnosti u většiny algoritmů nebudeme dělat příliš důsledně (je to komplikované), ale musíme si uvědomit, proč je to důležité.

Věta

Algoritmus gcd-naive je spravný.

Důkaz

Musíme ukázat, že algoritmus vždy najde správné řešení. Z teorie čísel víme, že platí:

1. největší společný dělitel dvou čísel $m, n \in \mathbb{N}$ je někde v intervalu $[0, \min(m, n)]$,
2. pokud je $m = 0$ nebo $n = 0$ (dle definice problému nemůže nastat $m = n = 0$), pak jejich největším společným dělitelem je větší z nich.

Situaci v bodu 2 evidentně řeší řádky 2-3, proto uvažujme 1, tj. že platí $m \neq 0$ a $n \neq 0$:

- nejprve je podmínka na řádku 4 otestována pro $t = \min(m, n)$,
- pokud t není $\gcd(m, n)$ snížíme t o 1 a následně jej znovu otestujeme v další iteraci,
- cyklus na řádcích 4-5 tedy prohledá celý interval $[1, \min(m, n)]$, v němž dle 1 se musí nacházet požadované řešení – to vrátíme jako výsledek $\gcd(m, n)$ na řádku 6. □

Euklidův algoritmus

Intuice: společné dělitele čísel m a n nezmění, když větší číslo nahradíme jejich zbytkem po dělení. Opakováním tohoto kroku se čísla rychle zmenšují, až zůstane $\gcd(m, n)$.

Algorithm gcd-Euclid

Input $m, n \in \mathbb{N}$
Output greatest common divisor of m and n

1: **while** $n \neq 0$ **do**
2: $\quad r \leftarrow m \bmod n$
3: $\quad m \leftarrow n$
4: $\quad n \leftarrow r$
5: **return** m

- jeden z nejstarších algoritmů (poprvé popsán Euklidem přibližně 300 př. n. l.)
- rozmyslete, co se stane, je-li na vstupu dvojice (m, n), kde $m < n$
- důkaz správnosti naleznete ve slidech od prof. Bělohlávka (slajdy 52-53)

Příklad: průběh algoritmu gcd-euclid

Příklad

1: **while** $n \neq 0$ **do**
2: $\quad r \leftarrow m \bmod n$
3: $\quad m \leftarrow n$
4: $\quad n \leftarrow r$
5: **return** m

Algoritmus demonstrujeme pro $m = 84$ a $n = 24$, tj. gcd-Euclid$(84, 24)$:

1. podmínka na řádku 1 je splněna $(24 \neq 0)$, a proto pokračujeme první iterací cyklu,
2. na řádcích 2–4 se do r uloží 12 (výsledek $84 \bmod 24$), dále se nastaví $m = 24$ a $n = 12$,
3. podmínka je opět splněna $(12 \neq 0)$, a proto pokračujeme druhou iterací cyklu,
4. na řádcích 2–4 se do r uloží 0 (výsledek $24 \bmod 12$), dále se nastaví $m = 12$ a $n = 0$,
5. další iterace cyklu nenastane, protože $n = 0$,
6. algoritmus pokračuje na řádku 5, kde je hodnota $m = 12$ vrácena jako výsledek.

Jak určit, který z algoritmů je lepší?

Provedeme analýzu jeho časové složitosti

- spočítáme/odhadneme počet instrukcí, které daný algoritmus vykonává
- ale tak jednoduché to nebude: pro různé vstupy se algoritmy mohou chovat různě, jak tedy určit vhodné kritéria pro jejich porovnání?
- podbrobněji se tomuto tématu budeme věnovat příště

Algoritmy porovnáme experimentálně

- spustíme je pro dostatečný vzorek dat a změříme dobu jejich vykonávání
- závislé na konkrétní implementaci, zvoleném programovacím jazyce atd.
- slouží hlavně pro potvrzení teoretických výsledků, my tomuto věnovat nebudeme

Naivní algoritmus vs Euklidův algoritmus

Motivace: jen pro zajímavost, zkusíme algoritmy analyzovat pro vstup $m = 84$ a $n = 24$.

Naivní algoritmus

- ověření speciálního případu na začátku (4 instrukce dohromady i s $\min(m, n)$)
- gcd-naive$(84, 24)$ provede 12 iterací cyklu (12krát se provede tělo a 13krát podmínka)
- tělo cyklu obsahuje jen výraz $t = t - 1$ (2 instrukce)
- ověření podmínky: $m \bmod t \neq 0$ or $n \bmod t \neq 0$ (5 instrukcí)
- 1 instrukce pro navrácení výsledné hodnoty algoritmu
- dostáváme tedy celkový počet $4 + (12 \cdot 2) + (13 \cdot 5) + 1 = 94$ instrukcí

Euklidův algoritmus (který je z těchto dvou opravdu ten lepší)

- gcd-Euclid$(84, 24)$ provede 2 iterace cyklu a 3krát se ověří podmínka
- na oveření podmínky stačí 1 instrukce, 4 instrukce pro tělo cyklu, a nakonec 1 return
- dostáváme tedy celkový počet $(2 \cdot 4) + 3 + 1 = 12$ instrukcí
- v tomto konkrétním případě je Euklidův algoritmus rychlejší, ale z jednoho experimentu nemůžeme udělat závěr (pro jiná čísla by mohl být naivní algoritmus rychlejší)

Porovnání algoritmů v Pythonu

```python
# runtime for 10000 pairs of random numbers: 0.9724s
def gcd_naive(m, n):
    t = min(m, n)

    if t == 0:
        return max(m,n)

    while m % t != 0 or n % t != 0:
        t -= 1

    return t


# runtime for 10000 pairs of random numbers: 0.0171s
def gcd_euclid(m, n):
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```