Programy a algoritmy pracující s čísly

IB111 Úvod do programování skrze Python

2013

Připomenutí z minule

- proměnné, výrazy, operace
- řízení výpočtu: if, for, while
- funkce
- příklady: faktoriál, binární čísla, hádanka

Dnešní přednáška

- práce s čísly v Pythonu
- ukázky programů, ilustrace použití základních konstrukcí
- ukázky jednoduchých algoritmů, ilustrace rozdílu v efektivitě

Číselné typy

- `int` – celá čísla
- `float`
    - čísla s plovoucí desetinnou čárkou
    - reprezentace: báze, exponent
    - nepřesnosti, zaokrouhlování
- (`complex` – komplexní čísla)

Nepřesnosti

Přesná matematika:

$$((1 + \frac{1}{x}) - 1) \cdot x = 1$$

Nepřesné počítače:

```
>>> x = 2**50
>>> ((1 + 1.0 / x) - 1) * x
1.0
>>> x = 2**100
>>> ((1 + 1.0 / x) - 1) * x
0.0
```

Číselné typy – poznámky

- dělení: rozdíl `3/2` a `3/2.0`
- explicitní přetypování: `int(x)`, `float(x)`
- automatické „nafukování“ typu `int` (`long`):
  - viz např. `2**100`
  - pomalejší, ale korektní
  - rozdíl od většiny jiných prog. jazyků (běžné je „přetečení“)

Pokročilejší operace s čísly

Některé operace v knihovně `math`:

- zaokrouhlování: `round`, `math.ceil`, `math.floor`
- absolutní hodnota: `abs`
- `math.exp`, `math.log`, `math.sqrt`
- goniometrické funkce: `math.sin`, `math.cos`, ...
- konstanty: `math.pi`, `math.e`

použití knihovny: `import math`

Ciferný součet

- vstup: číslo x
- výstup: ciferný součet čísla x
- příklady:
  - $8 \to 8$
  - $15 \to 6$
  - $297 \to 18$
  - $11211 \to 6$

Ciferný součet: základní princip

opakovaně provádíme:

- dělení 10 se zbytkem – hodnota poslední cifry
- celočíselné dělení – „okrajování“ čísla

Ciferný součet – ukázka nevhodného programu

```
if n % 10 == 0:
      f = 0 + f
  elif n % 10 == 1:
      f = 1 + f
  elif n % 10 == 2:
      f = 2 + f
  elif n % 10 == 3:
      f = 3 + f
  elif n % 10 == 4:
      f = 4 + f
  ...
```

Ciferný součet – řešení

```python
def ciferny_soucet(n):
    soucet = 0
    while n > 0:
        soucet += n % 10
        n = n / 10
    return soucet
```

Collatzova posloupnost

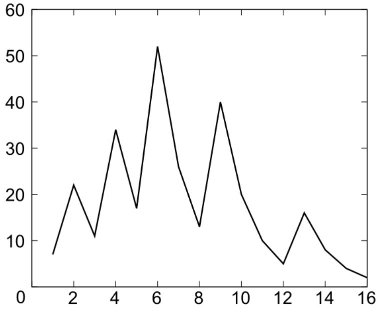
- vezmi přirozené číslo:
  - pokud je sudé, vyděl jej dvěma
  - pokud je liché, vynásob jej třemi a přičti jedničku
- tento postup opakuj, dokud nedostaneš číslo jedna

Collatzova posloupnost: výpis

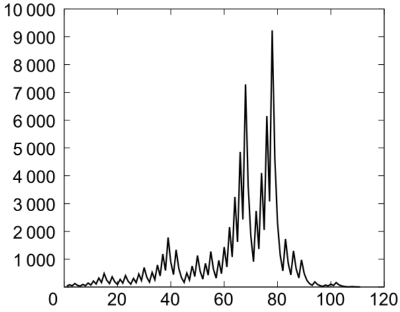
```
def collatz_vypis(n):
    while n != 1:
        print n,
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3*n + 1
    print 1
```

Collatzova posloupnost: příklady graficky

začínající číslem 7



začínající číslem 27



Bonus: Vykreslení grafu v Pythonu

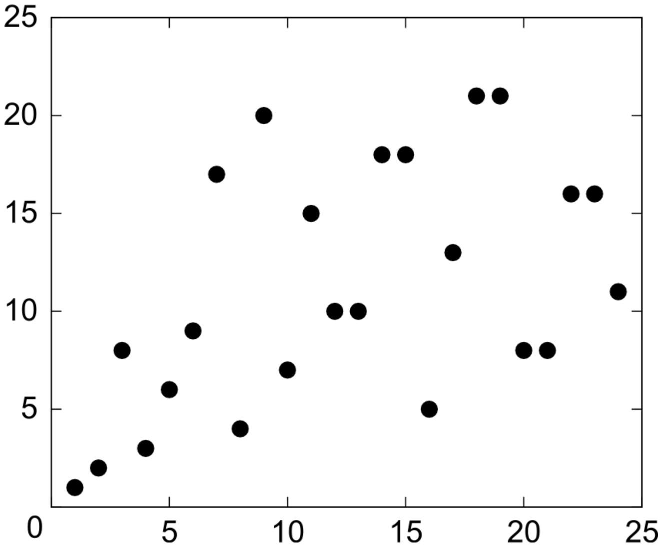
Využívá seznamy a knihovnu `pylab`

```
import pylab

def collatz(n):
    posloupnost = []
    while n != 1:
        posloupnost.append(n)
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3*n + 1
    posloupnost.append(1)
    return posloupnost

pylab.plot(collatz(7))
pylab.show()
```

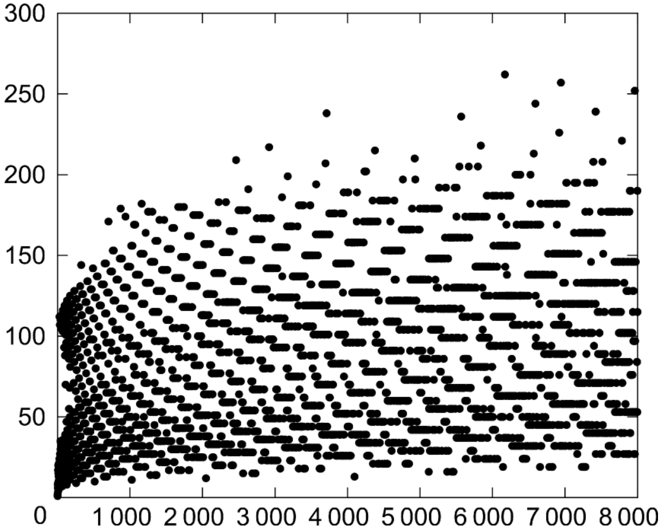
Collatzova posloupnost: délka posloupnosti

```
def collatz_delka(n):
    delka = 1
    while n != 1:
        if n % 2:
            n = 3*n + 1
        else:
            n = n / 2
        delka += 1
    return delka

def collatz_tabulka(kolik):
    for i in range(1, kolik+1):
        print i, collatz_delka(i)
```

Collatzova posloupnost: délka posloupnosti I



Collatzova posloupnost: délka posloupnosti II



Collatzova hypotéza

- Hypotéza: Pro každé počáteční číslo n, posloupnost narazí na číslo 1.
- experimentálně ověřeno pro velká n ($\sim 10^{18}$)
- důkaz není znám

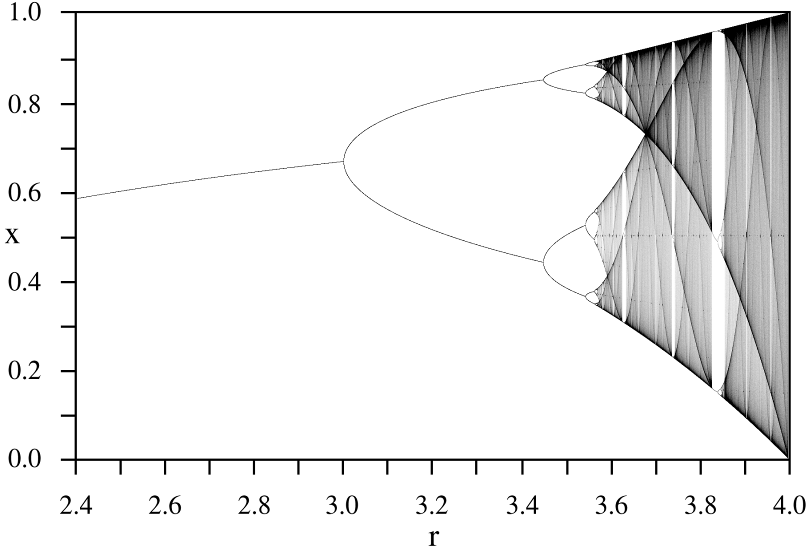
Collatzova posloupnost: experimenty

- upravte uvedenou funkci, aby nepočítala počet kroků, ale maximální číslo, které v průběhu výpočtu posloupnosti „potkáme“
- vykreslete graf těchto maximálních čísel

Logistická diferenční rovnice

$$x_{n+1} = 4 \cdot x_n \cdot (1 - x_n)$$

- jednoduchý úkol: výpis členů posloupnosti
- chaotické chování – citlivost k počátečním podmínkám (viz ukázka)
- zajímavé souvislosti: modelování populací, chaos, fraktály



Zdroj: Wikipedia

Největší společný dělitel

- vstup: přirozená čísla a, b
- výstup: největší společný dělitel a, b
- příklad: 180, 504

Jak na to?

Naivní algoritmus I

- projít všechny čísla od 1 do $min(a, b)$
- pro každé vyzkoušet, zda dělí a i b
- vzít největší

Naivní algoritmus II

- „školní“ algoritmus
- najít všechny dělitele čísel a, b
- projít dělitele, vybrat společné, vynásobit
- příklad:
    - $180 = 2^2 \cdot 3^2 \cdot 5$
    - $504 = 2^3 \cdot 3^2 \cdot 7$
    - $NSD = 2^2 \cdot 3^2 = 36$

Euklidův algoritmus: základ

základní myšlenka: pokud $a > b$, pak:

$$NSD(a, b) = NSD(a - b, b)$$

příklad:

krok	*a*	*b*
1	504	180
2	324	180
3	180	144
4	144	36
5	108	36
6	72	36
7	36	36
8	36	0

Operace modulo

- modulo = zbytek po dělení
- příklady:
  - $13 \bmod 5 = 3$
  - $28 \bmod 4 = 0$
  - $14 \bmod 3 = 2$
  - $18 \bmod 7 = ??$
  - $29 \bmod 13 = ??$

Euklidův algoritmus: vylepšení

vylepšená základní myšlenka: pokud $a > b$, pak:

$$NSD(a, b) = NSD(a \ mod \ b, b)$$

$krok$	a	b
1	504	180
2	180	144
3	144	36
4	36	0

Euklidův algoritmus: pseudokód

varianta s odčítáním, bez rekurze

```
def nsd(a,b):
    if a == 0:
        return b
    while b != 0:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Euklidův algoritmus: pseudokód

modulo varianta, rekurzivně

```
def nsd(a,b):
    if b == 0:
        return a
    else:
        return nsd(b, a % b)
```

Příklady

- 160, 75
- 57, 33

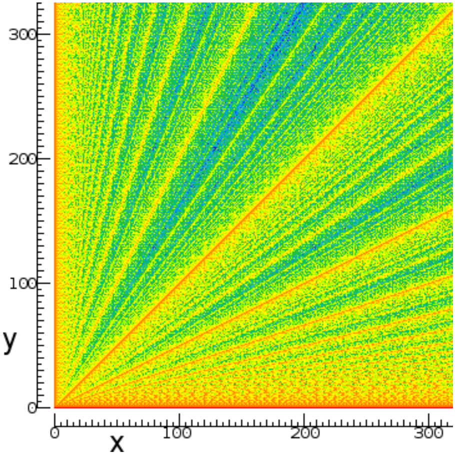
Příklad I – řešení

krok	*a*	*b*
1	160	75
2	75	10
3	10	5
4	5	0

Efektivita algoritmů

- proč byly první dva algoritmy označeny jako „naivní“?
- časová náročnost algoritmu:
  - naivní: exponenciální vůči počtu cifer
  - Euklidův: lineární vůči počtu cifer
- různé algoritmy se mohou výrazně lišit svou efektivností
- často rozdíl použitelné vs nepoužitelné
- více později (a v dalších předmětech)

Euklidův algoritmus – vizualizace



http://en.wikipedia.org/wiki/Euclidean_algorithm

Výpočet odmocniny

- vstup: číslo x
- výstup: přibližná hodnota $\sqrt{x}$

Jak na to?

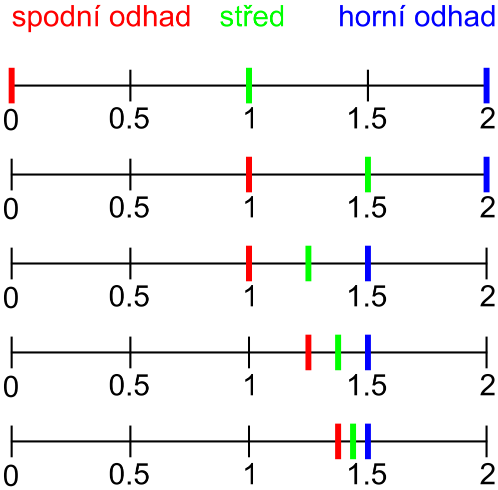
Výpočet odmocniny

- vstup: číslo x
- výstup: přibližná hodnota $\sqrt{x}$

Jak na to?

Mnoho metod, ukázka jedné z nich (rozhodně ne nejvíce efektivní)

Výpočet odmocniny: binární půlení



Výpočet odmocniny: binární půlení

```
def odmocnina(x, presnost = 0.01):
    horni_odhad = x
    spodni_odhad = 0
    stred = (horni_odhad + spodni_odhad) / 2.0
    while abs(stred**2 - x) > presnost:
        if stred**2 > x:
            horni_odhad = stred
        if stred**2 < x:
            spodni_odhad = stred
        stred = (horni_odhad + spodni_odhad) / 2.0
    return stred
```

Výpočet odmocniny – poznámky

- Funguje korektně jen pro čísla ≥ 1.
- Co program udělá pro čísla < 1?
- Proč?
- Jak to opravit?

Součet druhých mocnin

- Lze zapsat zadané číslo jako součet druhých mocnin?
- Příklad: $13 = 2^2 + 3^2$
- Která čísla lze zapsat jako součet druhých mocnin?

Součet druhých mocnin: řešení I

```
def soucet_ctvercu(n):
    for i in range(n):
        for j in range(n):
            if i**2 + j**2 == n:
                print n, "=", i**2, "+", j**2
```

- Program je zbytečně neefektivní. Proč?
- Jak upravit, abychom dostali výpis čísel, která lze zapsat jako součet čtverců?

Součet druhých mocnin: řešení II

```
def je_druha_mocnina(n):
    odmocnina = int(n**0.5)
    return odmocnina**2 == n

def soucet_ctvercu(n):
    for i in range(int(n**0.5) + 1):
        zbytek = n - i**2
        if je_druha_mocnina(zbytek):
            return True
    return False

def vypis_soucty_ctvercu(kolik):
    for i in range(kolik):
        if soucet_ctvercu(i):
            print i,
```

Podobné náměty

- variace: součet tří druhých mocnin, součet dvou třetích mocnin, . . .
- další náměty na posloupnosti: *The On-Line Encyclopedia of Integer Sequences*, `http://oeis.org/`

Náhodná čísla

- přesněji: *pseudo-náhodná* čísla
- opravdová náhodná čísla: `http://www.random.org/`
- bohaté využití v programování: výpočty, simulace, hry, ...
- Python
  - `import random`
  - `random.random()` – float od 0 do 1
  - `random.randint(a,b)` – celé číslo mezi a, b
  - mnoho dalších funkcí

Náhodná čísla: průměr vzorku

Vygenerujeme soubor náhodných čísel a vypočítáme průměrnou hodnotu:

```
def prumer_nahodnych(kolik, maximum = 100):
    soucet = 0.0
    for i in range(kolik):
        soucet += random.randint(0, maximum)
    return soucet / kolik
```

Jakou očekáváme hodnotu na výstupu? Jak velký bude rozptyl hodnot? (Názorná ukázka *centrální limitní věty*)

Simulace volebního průzkumu

- volební průzkumy se často liší; jaká je jejich přesnost?
- přístup 1: matematické modely, statistika
- přístup 2: simulace
- program:
  - vstup: reálné preference stran, velikost vzorku
  - výstup: preference zjištěné v náhodně vybraném vzorku

Simulace volebního průzkumu

```python
def pruzkum(vzorek, pref1, pref2, pref3):
    pocet1 = 0
    pocet2 = 0
    pocet3 = 0
    for i in range(vzorek):
        r = random.randint(1,100)
        if r <= pref1: pocet1 += 1
        elif r <= pref1 + pref2: pocet2 += 1
        elif r <= pref1 + pref2 + pref3: pocet3 += 1
    print "Strana 1:", 100.0 * pocet1 / vzorek
    print "Strana 2:", 100.0 * pocet2 / vzorek
    print "Strana 3:", 100.0 * pocet3 / vzorek
```

Poznámky ke zdrojovému kódu

- uvedené řešení není dobré:
  - „cut & paste“ kód
  - funguje jen pro 3 strany
- lepší řešení – využití seznamů

Výpočet π

- $\pi = 3.14159265359\ldots$
- Ale jak se na to přišlo?
- Jak vypočítat π?

Výpočet π

Příklady naivních metod:

- Gregoryho-Leibnizova řada:

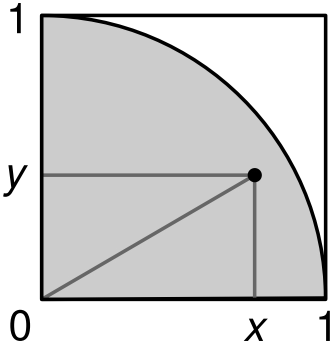
$$\pi = 4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

- Monte Carlo metoda – házení šipek do čtvrtdisku, Buffonova jehla

Výpočet π – Gregory-Leibniz

```python
def gregory_leibniz(n):
    soucet = 0.0
    znamenko = 1.0
    for k in range(1,n+1):
        soucet += znamenko/(2*k-1)
        znamenko *= -1
    return 4*soucet
```

Výpočet π – Monte Carlo

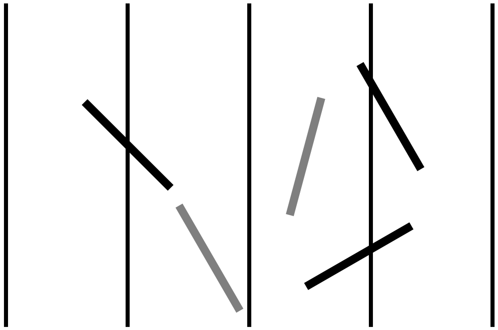


- obsah čtvrtdisku: $\pi/4$
- obsah čtverce: 1

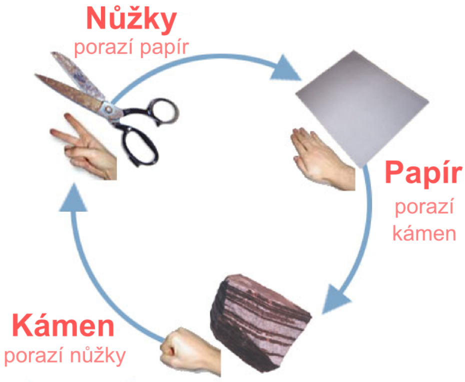
Výpočet π – Monte Carlo

```python
def monte_carlo_kruh(pocet_pokusu):
    zasahy = 0
    for k in range(pocet_pokusu):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1:
            zasahy += 1
    return 4.0 * zasahy / pocet_pokusu
```

Buffonova jehla



Kámen, nůžky, papír



http://cs.wikipedia.org/wiki/Kámen,_nůžky,_papír

KNP: strategie

```python
def strategie_rovnomerna():
    r = random.randint(1,3)
    if r == 1:
        return "K"
    elif r == 2:
        return "N"
    else:
        return "P"

def strategie_kamen():
    return "K"
```

KNP: vyhodnocení tahu

```
def vyhodnot(tah1, tah2):
    if tah1 == tah2:
        return 0
    if tah1 == "K" and tah2 == "N" or \
       tah1 == "N" and tah2 == "P" or \
       tah1 == "P" and tah2 == "K":
        return 1
    return -1
```

KNP: sehrání západu

```
def knp_hra(pocet_kol):
    body = 0
    for i in range(1, pocet_kol+1):
        print "Kolo ", i
        tah1 = strategie_rovnomerna()
        tah2 = strategie_rovnomerna()
        print "Tahy hracu:", tah1, tah2
        body += vyhodnot(tah1, tah2)
        print "Body hrace 1:", body
```

KNP: obecnější strategie

```python
def strategie(vahaK, vahaN, vahaP):
    r = random.randint(1, vahaK + vahaN + vahaP)
    if r <= vahaK:
        return "K"
    elif r <= vahaK + vahaN:
        return "N"
    else:
        return "P"
```

KNP: rozšiřující náměty

- turnaj různých strategií
- strategie pracující s historií
  - kopírování posledního tahu soupeře
  - analýza historie soupeře (hraje vždy kámen? $\rightarrow$ hraj papír)
- rozšíření na více symbolů (Kámen, nůžky, papír, ještěr, Spock)

Shrnutí

- operace s čísly, náhoda
- ukázky programů
- ukázky algoritmů, efektivita