Základy algoritmizace

2. Problémy, algoritmy, data

doc. Ing. Jiří Vokřínek, Ph.D.
Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Základy algoritmizace

- Dnes:
  - Řešení problému a algoritmus
  - Rozklad problému na podproblémy
  - Ukládání dat – pole
  - Funkce a anotace typů

Výpočet největšího společného dělitele

- Úloha

  Najděte největší společný dělitele přirozených čísel *x* a *y*.

- Řešení
  - Vstup: dvě přirozená čísla *x* a *y*
  - Výstup: přirozené číslo *d* – největší společný dělitel *x* a *y*
  - Základní varianta:

```
def getGreatestCommonDivisor(x, y):
    if (x<y) :
        d = x
    else:
        d = y
    while ((x % d != 0) or (y % d != 0)):
        d = d - 1
    return d
```

Pro (51851814,5185152) 185110 iterací cyklu while

Výpočet největšího společného dělitele

- Místo výběru menšího z čísel *x* a *y* vnoříme do těla hlavního cyklu dva cykly zmenšující hodnoty aktuálních hodnot *x* a *y*

```
def getGreatestCommonDivisorLoops(x, y):
    while (x != y):
        while (x>y):
            x -= y
        while (y>x):
            y -=x
    return x
```

Pro (51851814,5185152) 17650 iterací vnitřních cyklů while

- Vnitřní cykly počítají nenulový zbytek po dělení většího čísla menším

Výpočet největšího společného dělitele

- Vnitřní cykly můžeme nahradit přímým výpočtem zbytku po dělení

```
def getGreatestCommonDivisorEuclid(x, y):
    remainder = x % y
    while (remainder != 0):
        x = y
        y = remainder
        remainder = x % y
    return y
```

Pro (51851814,5185152) 4 iterace cyklu while

- Euklidův algoritmus
  - Určíme zbytek po dělení daných čísel
  - Zbytkem dělíme dělitele a určíme nový zbytek, až dosáhneme nulového zbytku
  - Poslední nenulový zbytek je největší společný dělitel

Funkce

Funkce

- Znovupoužitelný blok kódu
- Provádí jednu konkrétní funkci
- Podporuje modularitu programované aplikace
- Přijímá množinu argumentů
- Může vracet výsledek

```
def printme( parameters ):
   print(parameters)
   return

printme("Hello!")
```

Funkce

- Argumenty jsou vždy předávány referencí!
- Cokoli změníme v těle funkce, má vliv i mimo

```
def changeme( mylist ):
    mylist.append([1,2,3,4])
    print(mylist)
    return

list = [10,20,30]
changeme(list)
```

[10, 20, 30, [1, 2, 3, 4]]

- Pokud ale změníme referenci, nepřenese se mimo funkci

```
def changeme( mylist ):
    mylist = [1,2,3,4]
    print(mylist)
    return

list = [10,20,30]
changeme(list)
print(list)
```

[1, 2, 3, 4]

[10, 20, 30]

Funkce

- Funkce mají omezený „scope“ proměnných
- Proměnné definované v rámci funkce mají pouze lokální scope
- Proměnné definované mimo funkce mají globální scope (jsou přístupné odevšad)
- Vzájemně se překrývají, ale neovlivňují

```
def test():
    scope = "local"
    print(scope)
    return

scope = "global"
print(scope)
test()
print(scope)
```

global
local
global

Funkce

- Funkce mají omezený „scope“ proměnných
- Proměnné definované v rámci funkce mají pouze lokální scope
- Proměnné definované mimo funkce mají globální scope (jsou přístupné odevšad)
- Vzájemně se překrývají, ale neovlivňují

```
def test(scope):
    scope = "local"
    print(scope)
    return scope

scope = "global"
print(scope)
scope = test(scope)
print(scope)
```

```
global
local
local
```

Funkce

- Pojmenované argumenty
  - Nezáleží na pořadí
- Defaultní argumenty
  - Nemusí být uvedeny
  - Nesmí za ním být žádný non-default!

```
def doSomething( first , second = "none" ):
    print(first + second)
    return;

doSomething( second=2, first=1 )                3
doSomething( second="two", first="one" )        onetwo
doSomething("some")                             somenone
```

Funkce

- Proměnné počty parametrů (*tuple)
  - Formální argumenty jsou následovány proměnnou s hvězdičkou
  - Pokud nejsou předány, seznam je prázdný

```
def doSomething( *args):
    print(len(args))
    return;

doSomething( 1,2,3,4,5 )                5
doSomething()                           0
```

- Anonymní funkce
  - Nepoužívá def
  - Jednořádková funkce, vrací jen vyhodnocení výrazu
  - Nepracuje s globálními proměnými

http://www.tutorialspoint.com/python/python_functions.htm

Funkce

- Typová kontrola
  - Pomocí anotací
  - Kontrola nástrojem MyPy (nebo přímo v IDE)
  - Vhodné k definování „rozhraní“

```
numberVariable: int
def doSomething(number: int, name: str) -> bool:
```

  - Knihovna typing
    - př. vlastní typ – alias

```
from typing import List
IntegerMatrix = List[List[int]]
```

    - př. návratový typ, který může být i None

```
from typing import Optional
def generateRandomMatrix() -> Optional[integerMatrix]:
```

Funkce

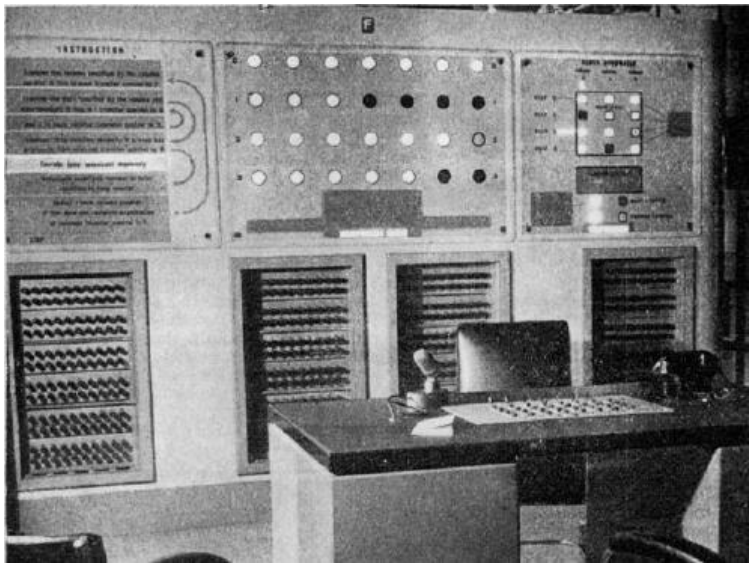
- Praktická doporučení
  - Funkce by měla být krátká – dělat jen jednu věc
  - Malý počet argumentů
  - Jméno volíme jako sloveso, př. computeFactorial(n)
  - Název funkce i argumentů by měl být samo-vypovídající
  - Snažíme se vyvarovat přepínání činnosti funkce hodnotou vstupních parametrů

Jak najít správný algoritmus?

Rozklad problému na podproblémy

- Postupný návrh programu rozkladem na podproblémy
  - Zadaný problém rozložíme na podproblémy
  - Pro řešení podproblémů zavedeme abstraktní příkazy
  - S abstraktními příkazy sestavíme hrubé řešení
  - Abstraktní příkazy realizujeme jako funkce

- Příklad – hra NIM
  - První „počítačová“ hra (1951)
  - Hra dvou hráčů v odebírání herních kamenů (nebo zápalek)




THE
GAME
CALLED
NIM
THREE

Hra NIM

- Pravidla
  - Hráč zadá počet herních kamenů
  - Pak se střídá se strojem v odebírání
    - Lze odebrat 1, 2, nebo 3 kameny
  - Prohraje ten, kdo odebere poslední herní kámen
- Dílčí problémy
  - Zadání počtu kamenů
  - Odebrání kamenů hráčem
  - Odebrání kamenů strojem

Příklad průběhu hry

- Zadej počet kamenů (15 až 35): 18
- Stav hry
  - Počet kamenů ve hře
  - Kdo je na tahu – počítač (P) nebo hráč (H)
- Průběh
  1. Počet kamenů 18; Kolik odeberete? **1** (H)
  2. Počet kamenů 17; Odebírám **1** (P)
  3. Počet kamenů 16; Kolik odeberete? **3** (H)
  4. Počet kamenů 13; Odebírám **1** (P)
  5. Počet kamenů 12; Kolik odeberete? **3** (H)
  6. Počet kamenů 11; Odebírám **1** (P)
  7. Počet kamenů 8; Kolik odeberete? **3** (H)
  8. Počet kamenů 5; Odebírám **1** (P)
  9. Počet kamenů 4; Kolik odeberete? **3** (H)
  10. Počet kamenů 1; Odebírám **1** (P)

- Hráč vyhrál, počítač odebral poslední kámen

NIM – pravidla pro odebírání

- Počet kamenů nevýhodných pro protihráče je 1, 5, 9, …
- Obecně $4n + 1$, kde n je celé nezáporné číslo
- Stroj musí z počtu kamenů K odebrat x kamenů, aby platilo

$$K - x == 4n + 1$$

  - $x == (K - 1) - 4n$, tj. hledáme zbytek po dělení 4
  - $x = (K - 1) \% 4$
- Je-li $x == 0$, je okamžitý počet kamenů pro stroj nevýhodný a bude-li protihráč postupovat správně, stroj prohraje

NIM – hrubý návrh řešení

```
numberOfStones = getNumberOfStones(limits)
machine = True

while (numberOfStones):
    if machine:
        machineTurn()
    else:
        humanTurn()
    machine = not machine
if machine: print("Machine wins!")
else: print("Human wins!")
```

- Podproblémy `getNumberOfStones`, `machineTurn` a `humanTurn` reprezentují abstraktní příkazy, které implementujeme jako funkce vracející počet kamenů k odebrání

NIM – podrobnější návrh

- Funkce na zadání počtu kamenů v rámci limitu
  - Ošetříme vstup na rozsah hodnot
  - Možno ošetřit proti špatnému zadání

Zkuste si sami

```
def getNumberOfStones(min: int, max: int) -> int:
    n = int(input("Enter number of stones in range ["
                  + str(min)+ "," + str(max) + "]: "))
    if n < min: return min
    if n > max: return max
    return n
```

NIM – podrobnější návrh

- Funkce pro tah hráče
  - Akceptujeme jen validní tahy – 1, 2 nebo 3 kameny
  - Vstupy plně ošetřeny

```
def humanTurn(n: int) -> int:
    r = ""
    while (r != "1" and r != "2" and r!= "3"):
        r = input("Number of stones is "
            + str(n) +
            ". How many stones you will take: ")
    return int(r)
```

NIM – podrobnější návrh

- Funkce pro tah stroje
  - Pomocí výpočtu optimální strategie

```
def machineTurn(n: int) -> int:
    r = (n-1) % 4
    if r == 0: r = 1
    print("Number of stones is "
          + str(n) + ". I take: " + str(r))
    return r
```

NIM – podrobnější návrh

- Finální program
  - Vyzkoušejte si naprogramovat a otestovat pro případ, kdy začíná počítač nebo člověk

Kdo vyhrává při aplikování optimální strategie?

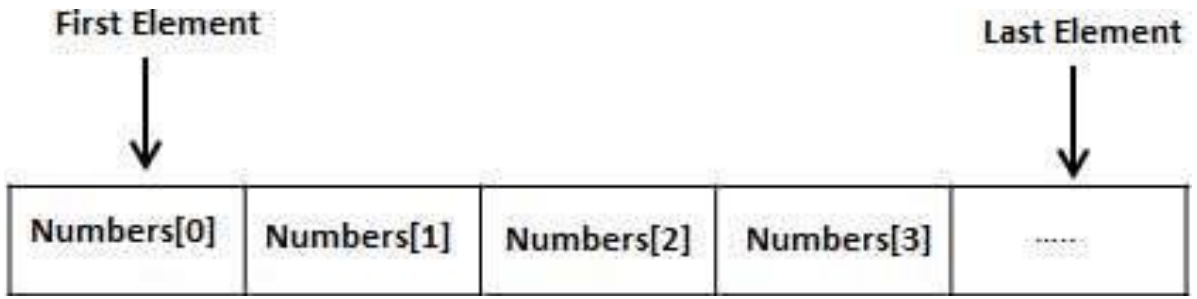
```
MIN_S = 15
MAX_S = 35
machine = True

numberOfStones = getNumberOfStones(MIN_S, MAX_S)
while (numberOfStones > 0):
    if machine:
        numberOfStones -= machineTurn(numberOfStones)
    else:
        numberOfStones -= humanTurn(numberOfStones)
    machine = not machine
if machine: print("Machine wins!")
else: print("Human wins!")
```

Když proměnná nestačí

Stav hry

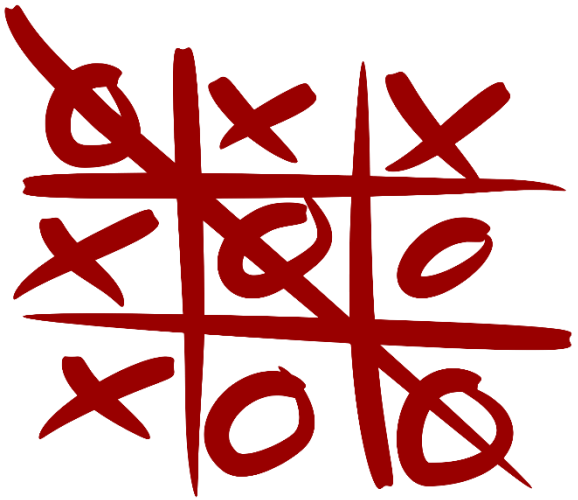
- U hry NIM je stav udržován v jedné proměnné
- Varianta „multiple-heap“ musí udržovat více stavových proměnných
  - heap1 = 10
  - heap2 = 20
  - heap3 = 30
- Co když máme tisíc hromádek?
  - Použijeme pole



- Vícerozměrná data?

Pole

- Vícerozměrná data
  - „pole polí“
  - Používáme seznamy, či tuple
- Matice, šachovnice, …
- Příklad reprezentace – piškvorky (tic-tac-toe)



Tic Tac Toe

- Jak se liší od hry NIM?
  - Reprezentace dat
  - Složitější výpočet strategie
  - Test konce hry
  - Test přípustnosti tahu
  - …

```
sizeOfGame = getSizeOfGame(limits)
machine = True
while (not gameFinished()):
    if machine:
        machineTurn()
    else:
        humanTurn()
    machine = not machine
if machine: print("Machine wins!")
else: print("Human wins!")
```

Tic Tac Toe

- Kostra hry

```
from typing import List
from typing import Optional
Game = List[List[str]]

def printGame(game: Game) -> None:

def getGame(*size: int) -> Game:

def isAllowed(game: Game, x: int, y: int) -> bool:

def boardFull(game: Game) -> bool:

def humanTurn(game: Game, char: str) -> None:

def machineTurn(game: Game, char: str) -> None:

def gameFinished(game: Game) -> Optional[str]:

def play(machine: bool, game: Game) -> None:
```

Tic Tac Toe

- Příklad vykreslování hry

```
def printGame(game: Game) -> None:
    for row in game:
        print("| ", end = '')
        for char in row:
            print(char+" | ", end = '')
        print()
```

```
| X | - | - | O | - | X | O | - | - | - |
| - | - | - | - | O | - | X | X | - | O |
| - | - | X | - | O | - | - | - | - | - |
| - | O | - | - | - | - | - | X | - | - |
| X | X | X | X | - | - | X | - | - | X |
| - | - | - | - | - | - | - | O | - | - |
| - | O | X | O | - | O | - | - | O | O |
| - | - | X | - | - | - | X | O | - | - |
| - | - | - | - | - | - | X | - | - | O |
| - | X | O | - | - | - | - | O | - | O |
```

Tic Tac Toe

- Příklad generování hry

```
def getGame(*size: int) -> Game:
    n = int(input("Enter size of the game (minimum 3): "))
    if n < 3: n = 3
    game = []
    for x in range(n):
        row = []
        for y in range(n):
            row.append("-")
        game.append(row)
    return game
```

- Nebo

```
def getGame(*size: int) -> Game:
    n = int(input("Enter size of the game (minimum 3): "))
    if n < 3: n = 3
    return [["-" for x in range(n)] for x in range(n)]
```

Tic Tac Toe

- Příklad testu tahu

```
def isAllowed(game: Game, x: int, y:int) -> bool:
    if x<0 or x >= len(game): return False
    if y<0 or y >= len(game): return False
    return game[x][y] == "-"
```

- Příklad testu remízy

```
def boardFull(game: Game) -> bool:
    for row in game:
        for cell in row:
            if cell == "-": return False
    return True
```

Tic Tac Toe

- Tah hráče

```
def humanTurn(game: Game, char: str) -> None:
    x,y = -1, -1
    while (not isAllowed(game, x, y)):
        printGame(game)
        x = int(input("Enter X position of your move: "))
        y = int(input("Enter Y position of your move: "))
    game[x][y] = char
```

Tic Tac Toe

- Tah počítače

```
import random

def machineTurn(game: Game, char: str) -> None:
    x,y = -1, -1
    while (not isAllowed(game, x, y)):
        x = random.randint(0,len(game)-1)
        y = random.randint(0,len(game)-1)
    game[x][y] = char
```

- Kde je AI????

Udělejte si v klidu doma

- Zacyklí se?

Statistika vs. pseudo-náhodná čísla

Tic Tac Toe

- Test konce hry

```
def gameFinished(game: Game) -> Optional[str]:
    for x in range(len(game)):
        for y in range(len(game)):
            char = game[x][y]
            if char == "-": continue
            if (y < len(game)-2 and game[x][y+1] == char
                and game[x][y+2]) == char:
                return char
            if (x< len(game)-2 and game[x+1][y] == char
                and game[x+2][y] == char):
                return char
            # add other two directions
```

Tic Tac Toe

- A celá hra ….

```
machine = True
game = getGame()
while True:
    if boardFull(game):
        print("No winner!")
        break;
    winner = gameFinished(game)
    if not winner:
        if machine:
            machineTurn(game,"O")
        else:
            humanTurn(game, "X")
        machine = not machine
    else:
        if winner == "O": print("Machine wins!")
        else: print("Human wins!")
        break
```

Tic Tac Toe

- Hra funguje?
- Nedostatky:
  - Test pouze 3 kamenů
  - Nekontrolujeme diagonály
  - Slabá AI
  - Není někde chyba? Jak to budeme testovat?
- Dodělejte/opravte si sami ;-)
- Měli jsme dobrou specifikaci hry?

Je dobré si pořádně promyslet co program má dělat a za jakých podmínek

Práce s polem

- Seznamy a řetězce – první přednáška
- Mohou být vnořené
- Tvoření pomocí append nebo zřetězení

```
array = [0 for x in range(n)]
```

- Přístup přes indexy
- Možno modifikovat obsah
- Iterace pomocí cyklu for nebo while

```
for x in array: doSomething(x)


i = 0
while i<len(array):
    doSomething(array[i])
    i+=1
```

Funkce – připomínka

- Praktická doporučení
  - Funkce by měla být **krátká – dělat jen jednu věc**
  - Malý počet argumentů
  - **Jméno** volíme jako sloveso, př. computeFactorial(n)
  - Název funkce i argumentů by měl být **samo-vypovídající**
  - Snažíme se vyvarovat přepínání činnosti funkce hodnotou vstupních parametrů

Základy algoritmizace

- Dnes:
  - Řešení problému a algoritmus
    - Příklad – výpočet společného dělitele, 1 funkce ≈ 10 řádek kódu
  - Rozklad problému na podproblémy
    - Příklad – hra NIM, 4 funkce ≈ 40 řádek kódu
  - Ukládání dat – pole
    - Příklad – hra Tic Tac Toe, 8 funkcí ≈ 80 řádek kódu
  - Funkce
  - Anotace typů

Příště vyhledávání a řazení