Algoritmy a programování

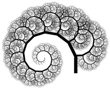
Rekurze

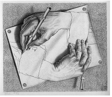
Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

Rekurze

Rekurze: definice problému pomocí jednoduší varianty stejného problému, odkaz sama na sebe





Implementační pohled

- Rekurzivní volání je pokud funkce volá sebe samu
- Více funkcí se volá navzájem

```python
def f(x):
    return f(x-1)

def a(x):
    b(x)

def b(x):
    a(x)
```

Algoritmický pohled

- Rekurze je způsob řešení problémů
- Rozděl a panuj (Divide and Conquer)
  - řešení problému je založeno na řešení jednodušší varianty stejného problému
- Rekurzivní definice matematických funkcí

$$x_{n+1} = rx_n(1 - x_n)$$

$$F_n = F_{n-1} + F_{n-2}$$
$$F_0 = F_1 = 1$$

- Funkce volá samu sebe
- Bez ukončovací podmínky vzniká nekonečný cyklus
- Prakticky je počet volání omezen operačním systémem ($\sim 10^3$ volání)
- Překročení tohoto limitu vede na chybu programu

```
def f(x):    #nekonecna rekurze
    return f(x-1)

f(10)
```

```
f(x)
[Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

- Při použití rekurze je třeba vždy uvést ukončovací podmínku

- Při použití rekurze je třeba vždy uvést ukončovací podmínku

```python
def f(n):
    if n > 0:
        return 1+f(n-1)
    return 0

print( f(-1) )
print( f(0) )
print( f(5) )
print( f(6) )
```

```
0
0
5
6
```

Rekurze: omezení počtu volání

- Argument se snižuje, `f(n)` → `f(n-1)` → `f(n-2)` → ...
- Je třeba zespoda omezit na Nmin

```
def f(n):
    if n > Nmin:
        f(n-1)
    else
        return
```

- Argument se zvyšuje, `f(n)` → `f(n+1)` → `f(n+2)` → ...
- Je třeba shora omezit na Nmax

```
def f(x):   #pocet volani je omezen
    if x < Nmax:
        return f(x+1)
    else:
        return
```

Rekurze: faktoriál

- Přímá definice: $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- Rekurzivní definice: $n! = n \cdot (n-1)!$
- Součástí rekurzivní definice je tzv. základní (bázový) případ: $0! = 1! = 1$
- Základní případ slouží jako ukončovací podmínka

```python
def f(n):
    if n == 0 or n == 1: #basic case
        return 1
    return n * f(n-1)

for i in range(6):
    print(i,"! = ",f(i), sep="")
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

- Přímá definice: $x^n = \prod_{i=1}^{n} x = \underbrace{x \cdot x \cdots x}_{n}$
- Rekurzivní definici: $x^n = x \cdot x^{n-1}$
- Základní případ: $x^0 = 1$

```python
def prod_iterative(x,n):
    result = 1
    for _ in range(n):
        result = result * x
    return result


print(prod_iterative(10,0))
print(prod_iterative(2,10))
print(prod_iterative(2**(0.5),2))
```

```
1
1024
2.0000000000000004
```

Rekurze: umocňování

- Přímá definice: $x^n = \prod_{i=1}^{n} x = \underbrace{x \cdot x \cdots x}_{n}$
- Rekurzivní definici: $x^n = x \cdot x^{n-1}$
- Základní případ: $x^0 = 1$

```python
def prod_recursive(x,n):
    if n == 0:
        return 1
    return x * prod_recursive(x,n-1)

print(prod_recursive(10,0))
print(prod_recursive(2,10))
print(prod_recursive(2**(0.5),2))
```

```
1
1024
2.0000000000000004
```

- Vstup je řetězec, úkolem je uložit (vypsat) ho v opačném pořadí
- ahoj $\rightarrow$ joha

Klasické řešení přes cyklus

- Projdeme string pozpátku, např. for cyklem

```python
def reverseIterative(x):
    result = ""
    for i in range(len(x)-1,-1,-1):
        result += x[i]
    return result

print( reverseIterative("PYTHON") )
```

```
NOHTYP
```

- Vstup je řetězec, úkolem je uložit (vypsat) ho v opačném pořadí
- ahoj → joha

Rekurzivní přístup

- Základní případ: pokud vstup je prázdný string, vracíme prázdný string
- Jinak: otočíme znaky `x[1:]` a přidáme k nim první znak `x[0]`

```python
def reverseRecursive(x):
    if len(x) == 0:
        return ""
    return reverseRecursive(x[1:]) + x[0]

print( reverseRecursive("PYTHON") )
```

```
NOHTYP
```

- Existuje řetězec, kde rekurzivní řešení selže?

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

- Každý cyklus lze nahradit rekurzí

Klasický cyklus $0, 1, \ldots n-1$

```python
def countUp(n):
    for i in range(n):
        print(i)

countUp(5)
```

```
0
1
2
3
4
```

- Každý cyklus lze nahradit rekurzí

Rekurzivní přístup výpočtu $0, 1, \ldots, n-1$

- Vnitřní funkce `countUpInner` volá sebe sama dokud aktuální hodnota je menší než maximum
- Uživatel může zavolat buď `countUpInner(n,0)` nebo `countUpRecursive(n)`

```
def countUpInner(maxvalue, actualvalue):
    if actualvalue < maxvalue:
        print(actualvalue)
        countUpInner(maxvalue,
            actualvalue+1)

def countUpRecursive(n):
    countUpInner(n,0)

countUpRecursive(5)
```

```
0
1
2
3
4
```

- Jaká je nevýhoda rekurzivního řešení oproti for+range ?

Permutace

- Výpis všech permutací M
- Řešení rekurzí
- Pro každý prvek $m_i \in M$:
  - najdi všechny permutace $M \setminus \{m_i\}$
  - před každou přidej m_i

```python
def printPermutation(prefix, items):
    if len(items) == 0:
        print(prefix, end=" ")  #print on one line
    for i in range(len(items)):
        printPermutation(prefix + items[i], items[:i]+items[i+1:])

y = ['a','b','c','d']
printPermutation("",y)
```

```
abcd abdc acbd acdb adbc adcb bacd badc bcad bcda bdac bdca cabd
    cadb cbad cbda cdab cdba dabc dacb dbac dbca dcab dcba
```

- Program pouze vypisuje, ale neukládá výsledek

- Upravíme předchozí program tak, aby ukládal nalezené permutace
- Místo `print(prefix)` uložíme do pole výsledků
- Použijeme globální proměnnou `globalResult`

```python
globalResult = []

def makePermutation(prefix, items):
    if len(items) == 0:
        globalResult.append(prefix)
    for i in range(len(items)):
        makePermutation(prefix + items[i], items[:i]+items[i+1:])

y = ['a','b','c','d']
makePermutation("",y)
print(globalResult)
```

```
['abcd', 'abdc', 'acbd', 'acdb', 'adbc', 'adcb', 'bacd', 'badc', '
    bcad', 'bcda', 'bdac', 'bdca', 'cabd', 'cadb', 'cbad', 'cbda',
    'cdab', 'cdba', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']
```

- Upravíme předchozí program tak, aby ukládal nalezené permutace
- Místo `print(prefix)` uložíme do pole výsledků
- Použijeme globální proměnnou `globalResult`

```python
globalResult = []

def makePermutation(prefix, items):
    if len(items) == 0:
        globalResult.append(prefix)
    for i in range(len(items)):
        makePermutation(prefix + items[i], items[:i]+items[i+1:])

y = ['a','b','c','d']
makePermutation("",y)
print(globalResult)
```

- Skrytý předpoklad: pole `globalResult` existuje a je prázdné
- Pokud by došlo k volání `savePermutation` z jiné rekurzivní funkce, hrozí přepsání dat
- Použití globálních proměnných není vhodné, **snažíme se nepoužívat**

- Správné řešení: použijeme další argument funkce `savePermutation`, do kterého budeme ukládat výsledek

```python

def savePermutation(prefix, items, result):
    if len(items) == 0:
        result.append(prefix)
    for i in range(len(items)):
        savePermutation(prefix + items[i], items[:i]+items[i+1:],
            result)

y = ['a','b','c','d']
result = []
savePermutation("", y, result)
print(result)
```

- Nepoužívá globální proměnné
- Je zaručena existence pole pro výsledky (při prvním volání `savePermutation`)

Permutace

- Program hledá permutaci pole, ale funguje i na řetězce

```python

def savePermutation(prefix, items, result):
    if len(items) == 0:
        result.append(prefix)
    for i in range(len(items)):
        savePermutation(prefix + items[i], items[:i]+items[i+1:],
            result)

y = "XYZ"
result = []
savePermutation("", y, result)
print(result)
```

```
['XYZ', 'XZY', 'YXZ', 'YZX', 'ZXY', 'ZYX']
```

Rekurze: výpis pole

- Vypíšeme první prvek a dále rekurzivně zbytek pole dokud je vstup neprázdný

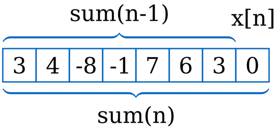
```python
def printRecursively(x): #x is list
    if len(x) != 0:
        print(x[0], end=" ")
        printRecursively(x[1:])
    else:
        print()  #empty list is printed as empty line

a = list( range(-10,10,3) )
print(a)
printRecursively(a)
print("*")
```

```
[-10, -7, -4, -1, 2, 5, 8]
-10 -7 -4 -1 2 5 8
*
```

Součet řady x_i, $i = 0, \ldots, n-1$

- Přímá definice: $s = x_0 + x_1 + \ldots + x_{n-1}$
- Rekurzivní definice: $sum(n) = sum(n-1) + x_n$
- Základní případ: $sum(1) = x_0$




```python
def sumRecursively(x): #x is list
    if len(x) == 0:
        return 0
    if len(x) == 1:  #basic case
        return x[0]
    return sumRecursively(x[:-1]) + x[-1]

a = [2,4,6]
print( sumRecursively(a) )
```

```
12
```

Mince

- Vstupem je hodnota a seznam mincí, úkolem je určit všechny kombinace mincí, které dávají požadovanou hodnotu
- Příklad: $c = (1, 2, 5)$, $amount = 5$ $\quad (5 \times 1\mathrm{CZK})$ nebo $(3 \times 1\mathrm{CZK} + 1 \times 2\mathrm{CZK})$ nebo $(1 \times 1\mathrm{CZK} + 2 \times 2\mathrm{CZK})$ nebo $(1 \times 5\mathrm{CZK})$

Rekurzivní řešení: skládáme částku *amount* z mincí $c_i, c_{i+1}, \ldots c_n$

- Zkusíme minci c_i, snížíme částku na *amount* $- c_i$, řešíme s mincemi $c_i, c_{i+1}, \ldots c_n$
- Nebo: nepoužijeme c_i, řešíme úlohu *amout* s mincemi $c_{i+1}, \ldots, c_n$

```
def allChanges(amount, coins, result, i):
    if amount == 0:
        for i in range(len(result)):
            if result[i] != 0:
                print(coins[i],"CZK x",result[i], end=", " )
        print()
    else:
        if coins[i] <= amount:
            result[i] += 1
            allChanges(amount - coins[i],coins, result, i)
            result[i] -= 1
        if i < len(coins)-1:
            allChanges(amount, coins, result, i+1)

coins = [1,2,5,10]
s = [0]*len(coins)
allChanges(12, coins, s, 0)
```

Mince: postup

Rekurzivní řešení: skládáme částku *amount* z mincí $c_i, c_{i+1}, \ldots c_n$

- Zkusíme minci c_i, snížíme částku na $amount - c_i$, řešíme s mincemi $c_i, c_{i+1}, \ldots c_n$
- Nebo: nepoužijeme c_i, řešíme úlohu *amout* s mincemi $c_{i+1}, \ldots, c_n$

```
1 CZK x 12,
1 CZK x 10, 2 CZK x 1,
1 CZK x 8, 2 CZK x 2,
1 CZK x 7, 5 CZK x 1,
1 CZK x 6, 2 CZK x 3,
1 CZK x 5, 2 CZK x 1, 5 CZK x 1,
1 CZK x 4, 2 CZK x 4,
1 CZK x 3, 2 CZK x 2, 5 CZK x 1,
1 CZK x 2, 2 CZK x 5,
1 CZK x 2, 5 CZK x 2,
1 CZK x 2, 10 CZK x 1,
1 CZK x 1, 2 CZK x 3, 5 CZK x 1,
2 CZK x 6,
2 CZK x 1, 5 CZK x 2,
2 CZK x 1, 10 CZK x 1,
```

Mince:

- Hledáme nejmenší počet mincí (o známých hodnotách), které poskládají vstupní částku
- Příklad: mince $(1, 2, 5)$, částka 10 CZK, řešení: 2 x 5 CZK (jiné řešení bude potřebovat více mincí)
- Greedy (“hladové”) řešení: preferujeme sumu poskládat z mincí vyšší hodnoty

```python
def solve(amount, result, coins):
    if amount == 0:
        return
    for i in range(len(coins)-1,-1,-1):
        c = coins[i]
        if c <= amount:
            num = amount // c
            amount %= c
            result.append([num, c] )
            solve(amount, result, coins[:i]+coins[i:])
            break
result = []
solve(37, result, [1,2,5,10] )
for item in result: #item is [numberOfCoin, coin ]
    number, coin = item
    print(coin, " CZK x ", number)
```

Mince:

- Hledáme nejmenší počet mincí (o známých hodnotách), které poskládají vstupní částku
- Příklad: mince $(1, 2, 5)$, částka 10 CZK, řešení: 2 x 5 CZK (jiné řešení bude potřebovat více mincí)
- Greedy ("hladové") řešení: preferujeme sumu poskládat z mincí vyšší hodnoty

```
10  CZK x  3
5  CZK x  1
2  CZK x  1
```

Mergesort

- Třídící algoritmus využívající principle divide-and-conquer
- Pole je rozděleno na dvě poloviny
- Každá se setřídí (rekurzivně)
- Výsledné pole jsou spojeny

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

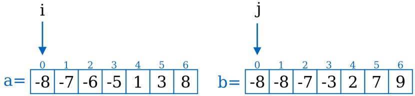
```python
def mergeSort(a):
    if len(a) <= 1:
        return a
    half = len(a) // 2
    left = mergeSort(a[:half])  #sort the first half
    right = mergeSort(a[half:]) #sort the second half
    return joinSortedArrays(left,right)

def joinSortedArrays(a,b):
    result = []    #new temporary array
    i = 0
    j = 0;
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
    result += a[i:]
    result += b[j:]
    return result
```

Mergesort: spojení polí

- Spojení dvou seřazených polí

```python
def joinSortedArrays(a,b):
    result = []    #new temporary array
    i = 0
    j = 0;
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1
    result += a[i:]
    result += b[j:]
    return result
```



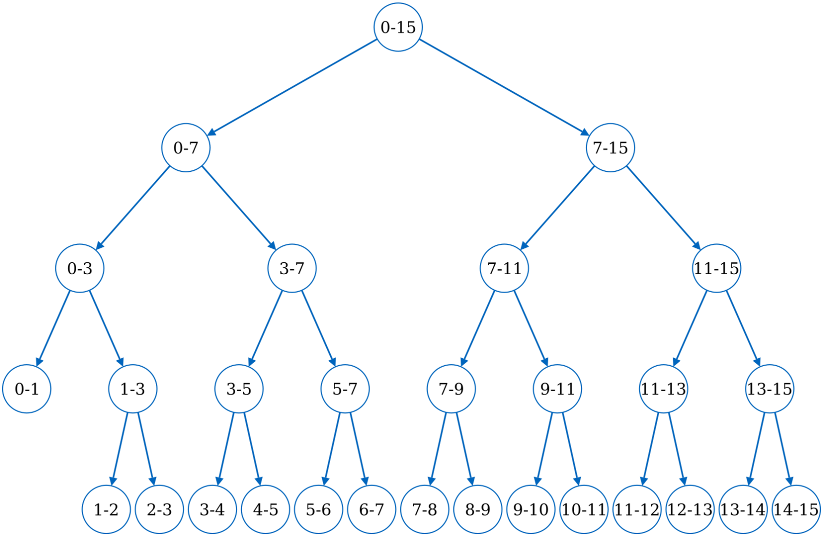
```python
from mergeSort import mergeSort

a = [2,-1,0,5,7,7,1]
b = mergeSort(a)
print(a)
print(b)
```

```
[2, -1, 0, 5, 7, 7, 1]
[-1, 0, 1, 2, 5, 7, 7]
```

https://www.youtube.com/watch?v=991E5jwQXC8

- Graf volání `mergeSort(array[start:end])` pro seřazení pole o délce $n = 16$ prvků



Mergesort: vlastnosti

- Algoritmus potřebuje pomocné pole
- Velikost tohoto pole je n
- Řazení není in-place
- Mergesort je stabilní
- Spojení dvou polí — složitost $\mathcal{O}(n)$
- Počet úrovní je $\sim \log_2 n$
- Složitost $\mathcal{O}(n \log n)$

Quick sort

- Rychlé třídění (T. Hoare, 1959)
- V poli určíme jeden prvek — pivot
- Partitioning — částečné setřídění tak aby prvky “před” pivotem byly menší než pivot (a prvky “za” pivotem budou větší)

$$x = [\underbrace{\ldots a_i \ldots}_{a_i \leq p} p \underbrace{\ldots a_j \ldots}_{a_j \geq p}]$$

- Rekurzivně setřídíme obě části $[\ldots a_i \ldots]$ a $[\ldots a_j \ldots]$

- Rekurzivní volání QuickSort
- Uživatel používá `quickSort(a)`, kde `a` je pole
- Rekurze je řešena funkcí `quickSortInternal(a,low, high)`, kde `low` a `high` určuje část pole pro seřazení

```
def quickSortInternal(a, low, high):
    if low >= 0 and high >=0 and low < high:
        pivot = partition(a,low, high)
        quickSortInternal(a, low, pivot)
        quickSortInternal(a, pivot+1, high)

def quickSort(a):
    quickSortInternal(a, 0, len(a)-1)
```

Quick sort: částečné setřídění

- Vstupem je pole, první prvek (low) a poslední prvek (high)
- Pivot je v polovině rozsahu low:high
- Procházíme prvky zleva (od low) dokud `a[i] < pivot`
- Pak procházíme prvky zprava (od high) dokud `a[j] > pivot`
- Pokud se indexy `i` a `j` potkají, menší z nich je nový pivot
- Jinak vyměníme prvky `a[i]` a `a[j]`

```python
def partition(a, low, high):
    pivot = a[ (low + high) // 2 ]
    i = low-1
    j = high+1
    while True:
        i += 1
        while a[i] < pivot:
            i += 1
        j -= 1
        while a[j] > pivot:
            j -= 1
        if i >= j:
            return j
        a[i], a[j] = a[j], a[i]
```

- Výsledkem částečného setřídění je nový pivot
- Platí, že prvky před pivotem jsou menší nebo rovno než pivot (a prvky za pivotem jsou větší nebo rovno pivot)

Quick sort: vlastnosti

- In-place třídění
- Rekurzivní postup
- Quicksort není stabilní (většina implementací)
- Quicksort je jednoduchý na pochopení, ale je snadné udělat chybu při implementaci (další přednáška)
- Průměrná složitost $\mathcal{O}(n \log n)$
- Nejhorší složitost $\mathcal{O}(n^2)$

Quick sort: varianty

Introsort

- Quicksort + detekce + heapsort

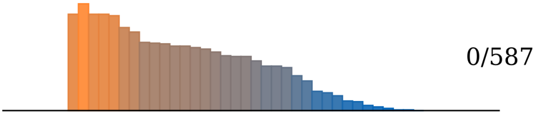
QuickSort + InsertionSort

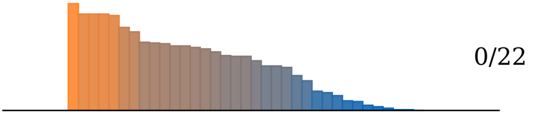
- Hlavní třídění probíhá QuickSortem, pokud při rekurzivním volání dojde k řazení krátkého pole ($\sim$ 10 položek), přepne se na InsertionSort

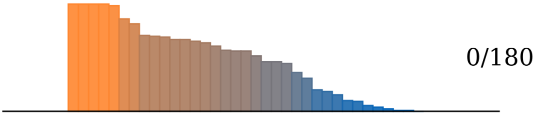
Rekurze: převod na nerekurzivní formu

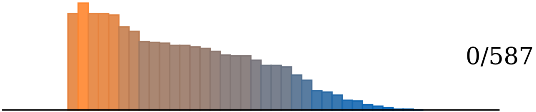
- Každé rekurzivní řešení je možné zapsat nerekurzivně, ale může to být těžké
- Je třeba pamatovat si kontext jednotlivých volání (vnitřní proměnné a argumenty volání funkcí)
- Zde se využívá datová struktura zásobník (viz další přednášky)

Opačně seřazené pole Poznáte algoritmy podle průběhu?

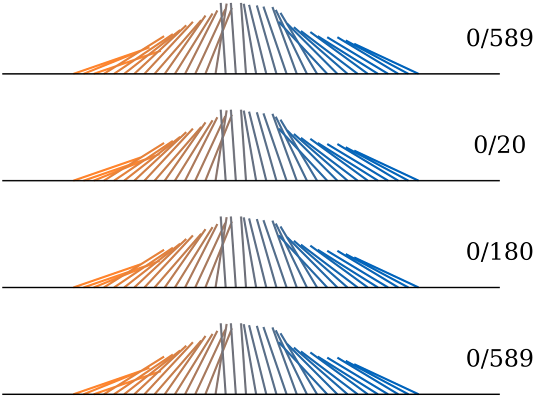








Opačně seřazené pole Poznáte algoritmy podle průběhu?



Opačně seřazené pole Poznáte algoritmy podle průběhu?