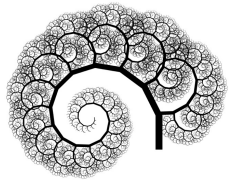
Algoritmy a programování

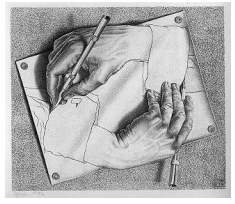
Řešení úloh rekurzí

Vojtěch Vonásek

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

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





```
1 def f(x):
2     return f(x-1)
3
4 def a(x):
5     b(x)
6
7 def b(x):
8     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

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

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

Jak vytvořit rekurzivní řešení nějaké úlohy?

- U některých úloh je již známé, typicky u matematických úloh, např:
  - rekurentí forma matematických řad: $F_n = F_{n-1} + F_{n-2}$
  - faktoriál: $n! = n \cdot (n-1)!$
  - suma řady $a_1, a_2, \ldots a_n$ je $\sum_{i=1}^{n} a_i = a_1 + \sum_{i=2}^{n} a_i$
  - binomický koeficient:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{a} \quad \binom{n}{0} = \binom{n}{n} = 1$$

  - U těchto forem je přepis do programovacího jazyka jednoduchý
- Jak ale použít (vymyslet) rekurzivní řešení pro nějakou úlohu?
  - Umístěte n královen na šachovnici $n \times n$ tak, aby se neohrožovaly
  - Výpis všech permutací ze vstupního listu objektů
  - Máme n věcí, každá má hmotnost a cenu, a musíme vybrat některé z nich tak, aby hmotnost vybraných nepřekročila danou váhu a zároveň byl součet cen vybraných nejvetší
  - a mnoho dalších ...

Rekurze: když známe rekurentní vztah

- Rekurentní vzhat naznačuje, jak zapíšeme do programovacího jazyka
- Nesmíme zapomenou na ukončovací podmínky

Příklad: faktoriál

- $n! = n \cdot (n-1)!$, víme-li, že $0! = 1! = 1$
- Nechť faktoriál implementuje funkce `fact(n)`. Vzorec přepíšeme `fact(n)= n * fact(n-1)`
- Funkce fact(n) vrací násobek n a volání `fact(n-1)`

```
1 def fact(n):
2     if n == 0 or n == 1: #basic case
3         return 1
4     return n * fact(n-1)
5
6 for i in range(6):
7     print(i,"! = ",fact(i), sep="")
```

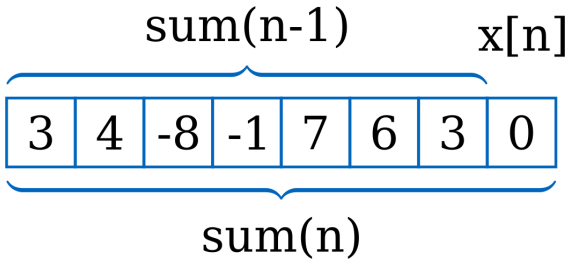
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Rekurze: když známe rekurentní vztah

- Rekurentní vzhat naznačuje, jak zapíšeme do programovacího jazyka
- Nesmíme zapomenou na ukončovací podmínky

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: $sum(1) = x_0$

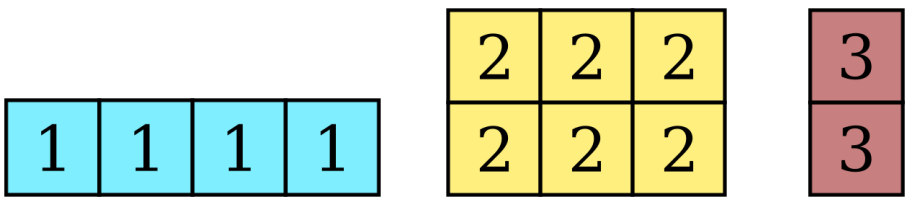



```
1 def sumRecursively(x): #x is list
2     if len(x) == 0:
3         return 0
4     if len(x) == 1:  #basic case
5         return x[0]
6     return sumRecursively(x[:-1]) + x[-1]
7
8 a = [2,4,6]
9 print( sumRecursively(a) )
```

```
12
```

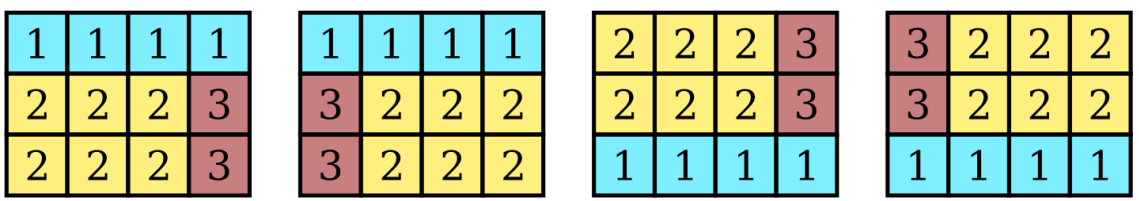
Skládání beden na paletu

- Máme *n* obdélníků a paletu; známe rozměry obdélníků a palety
- Úkolem je určit poskládání obdélníků tak, aby zcela zaplnily paletu (nebo říci, že to nejde)



1×4 2×3 2×1

- Všechna řešení na paletě 3×4



Odvození rekurzivního řešení

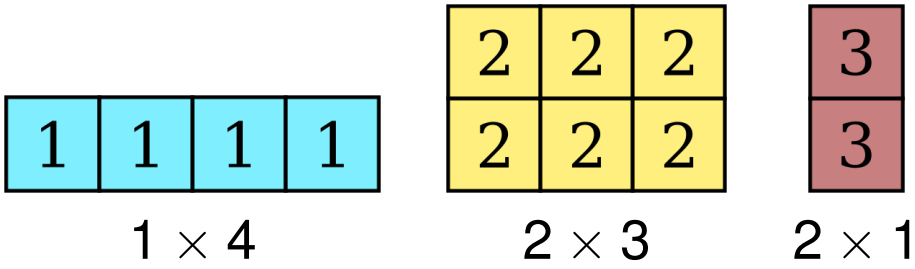
Skládání beden na paletu

- Máme *n* obdélníků a paletu; známe rozměry obdélníků a palety
- Úkolem je určit poskládání obdélníků tak, aby zcela zaplnily paletu (nebo říci, že to nejde)

Jak na to

- Jak budeme reprezentovat krabice a paletu?
  - Paleta je 2D pole, hodnoty: 0 = neobsazená, čísla > 0 jsou barva krabice
  - Jedna krabice: `box = [ rows, cols, color ]`
- Seznam krabic:

```
[ [rows1, cols1, color1],[rows2, cols2,color2], ... ]
```

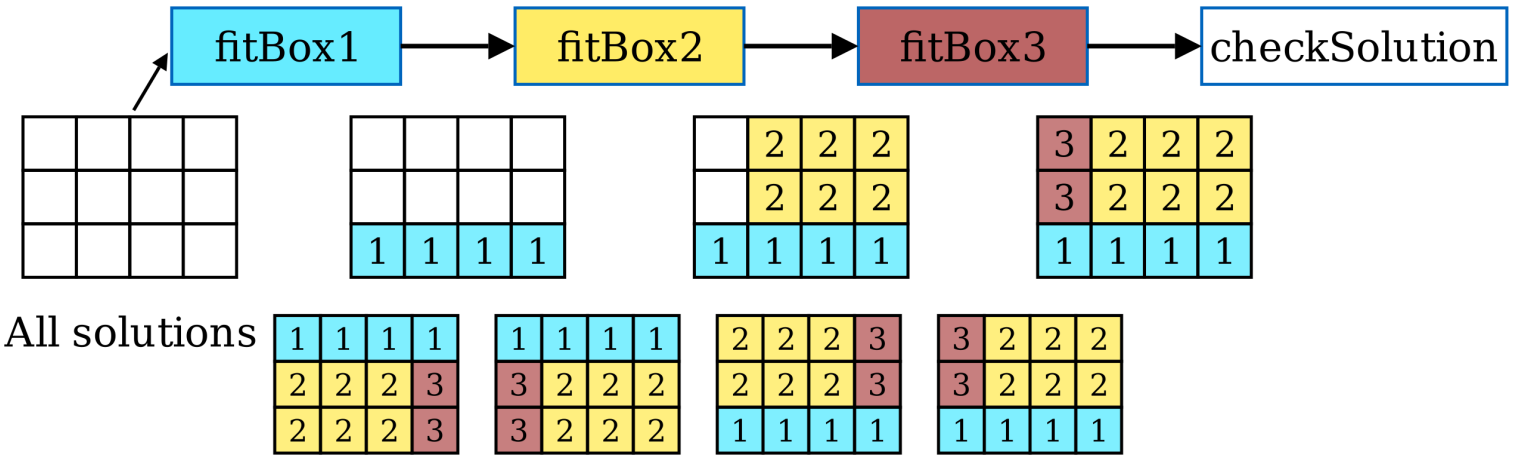


- `boxes = [ [1,4,1], [2,3,2], [2,1,3] ]`

Odvození rekurzivního řešení

Postup implementace nejprve si odvodíme řešení pro tři boxy

- Napíšeme funkce `fitBox1`, `fitBox2` a `fitBox3`
- Každá pracuje pouze s jedním boxem
- Všechny funkce sdílejí proměnnou pro zápis boxů
- `fitBox1` umístí první box na první volnou pozici, pak zavolá `fitBox2`
- `fitBox2` umístí druhý box na první volnou pozici, pak zavolá `fitBox3`
- `fitBox3` umístí třetí box na první volnou pozici, pak zavolá kontrolu řešení



Odvození rekurzivního řešení


```
1  def fitBox1(boxes, desk):
2      numBox = 0
3      boxRows, boxCols, cardColor = boxes[numBox]
4      deskRows, deskCols = deskSize(desk)
5      for row in range(deskRows):
6          for col in range(deskCols):
7              if canBePlaced(row, col, boxes[numBox], desk):
8                  writeBox(row, col, boxes[numBox], desk, cardColor)
9                  fitBox2(boxes, desk)
10                 writeBox(row, col, boxes[numBox], desk, 0)
```

- Funkce nejprve zjistí velikost krabice (`boxRows, boxCols`) a velikost palety (`deskRows, deskCols`)
- Projdeme všechny políčka `row, col` na paletě
- Pokud je možné vložit kartu (tj. celá se vejde na paletu a uvnitř je prázdno), pak:
  - Zapíšeme do palety, že jsme box umístili — zapíšeme hodnotu `cardColor`
  - Zavoláme `fitBox1`, která se stará o další postup
  - Poté odmažeme umístěnou kartu (tj. zápis hodnoty 0)

Odvození rekurzivního řešení


```
1  def fitBox2(boxes, desk):
2      numBox = 1
3      boxRows, boxCols, cardColor = boxes[numBox]
4      deskRows, deskCols = deskSize(desk)
5      for row in range(deskRows):
6          for col in range(deskCols):
7              if canBePlaced(row, col, boxes[numBox], desk):
8                  writeBox(row, col, boxes[numBox], desk, cardColor)
9                  fitBox3(boxes, desk)
10                 writeBox(row, col, boxes[numBox], desk, 0)
```

```
1  def fitBox3(boxes, desk):
2      numBox = 2
3      boxRows, boxCols, cardColor = boxes[numBox]
4      deskRows, deskCols = deskSize(desk)
5      for row in range(deskRows):
6          for col in range(deskCols):
7              if canBePlaced(row, col, boxes[numBox], desk):
8                  writeBox(row, col, boxes[numBox], desk, cardColor)
9                  checkSolution(boxes, desk)
10                 writeBox(row, col, boxes[numBox], desk, 0)
```

- Funkce `fitBox3` už jen volá kontrolu řešení (`checkSolution`)

Odvození rekurzivního řešení


```
1  def checkSolution(boxes, desk):
2      deskRows, deskCols = deskSize(desk)
3      isFull = True
4      for row in range(deskRows):
5          for col in range(deskCols):
6              if desk[row][col] == 0:
7                  isFull = False
8      if isFull:
9          print("Solution", desk)
10         allSolutions.append( copy.deepcopy(desk) )
```

- `checkSolution`: pokud paleta neobsahuje hodnotu 0, je řešení nalezeno

Odvození rekurzivního řešení

- Funkce `fitBox1`, `fitBox2` a `fitBox3` jsou si velmi podobné
- Stejdnotíme je do jedné funkce `fitBoxes` takto:
  - vstupem bude `numBox`, který určuje číslo krabice, která se aktuálně umisťuje
  - při volání `fitBoxes` zvýšíme číslo numBox o jedna
  - pokud `numBox` je roven počtu boxu, provedeme kontrolu řešení
  - nalezená řešení sdílíme v proměnné `allSolutions`

Odvození rekurzivního řešení


```
1  def fitBoxes(boxes, desk, numBox, allSolutions):
2      deskRows, deskCols = deskSize(desk)
3      if len(boxes) == numBox:
4          isFull = True
5          for row in range(deskRows):
6              for col in range(deskCols):
7                  if desk[row][col] == 0:
8                      isFull = False
9                      break
10         if isFull:
11             allSolutions.append( copy.deepcopy( desk ) )
12         return
13
14     for row in range(deskRows):
15         for col in range(deskCols):
16             if canBePlaced(row, col, boxes[numBox], desk):
17                 writeBox(row, col, boxes[numBox], desk, boxes[
                       numBox][2])
18                 fitBoxes(boxes, desk, numBox+1, allSolutions)
19                 writeBox(row, col, boxes[numBox], desk, 0)
```

- Nalezená řešení sdílíme v proměnné allSolutions

Odvození rekurzivního řešení


```
1      desk = [ [0]*4 for _ in range(3) ]
2      boxes = [ [1,4,1], [2,3,2], [2,1,3] ]
3      solutions = []
4      fitBoxes(boxes, desk, 0, solutions)
5      for solution in solutions:
6          print(solution)
```

```
[[1, 1, 1, 1], [2, 2, 2, 3], [2, 2, 2, 3]]
[[1, 1, 1, 1], [3, 2, 2, 2], [3, 2, 2, 2]]
[[2, 2, 2, 3], [2, 2, 2, 3], [1, 1, 1, 1]]
[[3, 2, 2, 2], [3, 2, 2, 2], [1, 1, 1, 1]]
```

Co dál:

- Jak zjistíme, že žádné řešené neexistuje?
- Co se stane, když v počátečním poli desk budou některá políčka mít zápornou hodnotu?
- Jak ukončit prohledávání ihned při nalezení prvního řešení?

Permutace

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

```
1 def printPermutation(prefix, items):
2     if len(items) == 0:
3         print(prefix, end="␣")  #print on one line
4     for i in range(len(items)):
5         printPermutation(prefix + items[i], items[:i]+items[i+1:])
6
7 y = ['a','b','c','d']
8 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

Permutace

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

```
1  globalResult = []
2
3  def makePermutation(prefix, items):
4      if len(items) == 0:
5          globalResult.append(prefix)
6      for i in range(len(items)):
7          makePermutation(prefix + items[i], items[:i]+items[i+1:])
8
9  y = ['a','b','c','d']
10 makePermutation("",y)
11 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']
```

Permutace

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

```
1  globalResult = []
2
3  def makePermutation(prefix, items):
4      if len(items) == 0:
5          globalResult.append(prefix)
6      for i in range(len(items)):
7          makePermutation(prefix + items[i], items[:i]+items[i+1:])
8
9  y = ['a','b','c','d']
10 makePermutation("",y)
11 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**

Permutace

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

```
1
2  def savePermutation(prefix, items, result):
3      if len(items) == 0:
4          result.append(prefix)
5      for i in range(len(items)):
6          savePermutation(prefix + items[i], items[:i]+items[i+1:],
               result)
7
8  y = ['a','b','c','d']
9  result = []
10 savePermutation("", y, result)
11 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

```
1
2  def savePermutation(prefix, items, result):
3      if len(items) == 0:
4          result.append(prefix)
5      for i in range(len(items)):
6          savePermutation(prefix + items[i], items[:i]+items[i+1:],
               result)
7
8  y = "XYZ"
9  result = []
10 savePermutation("", y, result)
11 print(result)
```

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

Rekurze: výpis pole

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

```
1  def printRecursively(x): #x is list
2      if len(x) != 0:
3          print(x[0], end="␣")
4          printRecursively(x[1:])
5      else:
6          print()  #empty list is printed as empty line
7
8  a = list( range(-10,10,3) )
9  print(a)
10 printRecursively(a)
11 print("*")
```

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

Mince

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

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  def allChanges(amount, coins, result, i):
2      if amount == 0:
3          for i in range(len(result)):
4              if result[i] != 0:
5                  print(coins[i],"CZK x",result[i], end=", " )
6          print()
7      else:
8          if coins[i] <= amount:
9              result[i] += 1
10             allChanges(amount - coins[i],coins, result, i)
11             result[i] -= 1
12         if i < len(coins)-1:
13             allChanges(amount, coins, result, i+1)
14
15 coins = [1,2,5,10]
16 s = [0]*len(coins)
17 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

```
1  def solve(amount, result, coins):
2      if amount == 0:
3          return
4      for i in range(len(coins)-1,-1,-1):
5          c = coins[i]
6          if c <= amount:
7              num = amount // c
8              amount %= c
9              result.append([num, c] )
10             solve(amount, result, coins[:i]+coins[i:])
11             break
12 result = []
13 solve(37, result, [1,2,5,10] )
14 for item in result: #item is [numberOfCoin, coin ]
15     number, coin = item
16     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

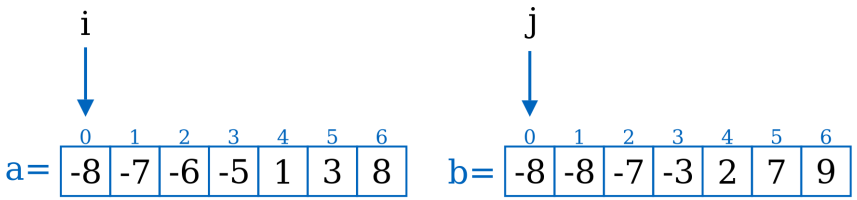
Mergesort


```
1  def mergeSort(a):
2      if len(a) <= 1:
3          return a
4      half = len(a) // 2
5      left = mergeSort(a[:half])  #sort the first half
6      right = mergeSort(a[half:]) #sort the second half
7      return joinSortedArrays(left,right)
8
9  def joinSortedArrays(a,b):
10     result = []    #new temporary array
11     i = 0
12     j = 0;
13     while i < len(a) and j < len(b):
14         if a[i] < b[j]:
15             result.append(a[i])
16             i += 1
17         else:
18             result.append(b[j])
19             j += 1
20     result += a[i:]
21     result += b[j:]
22     return result
```

Mergesort: spojení polí

- Spojení dvou seřazených polí

```
1  def joinSortedArrays(a,b):
2      result = []    #new temporary array
3      i = 0
4      j = 0;
5      while i < len(a) and j < len(b):
6          if a[i] < b[j]:
7              result.append(a[i])
8              i += 1
9          else:
10             result.append(b[j])
11             j += 1
12     result += a[i:]
13     result += b[j:]
14     return result
```



Mergesort

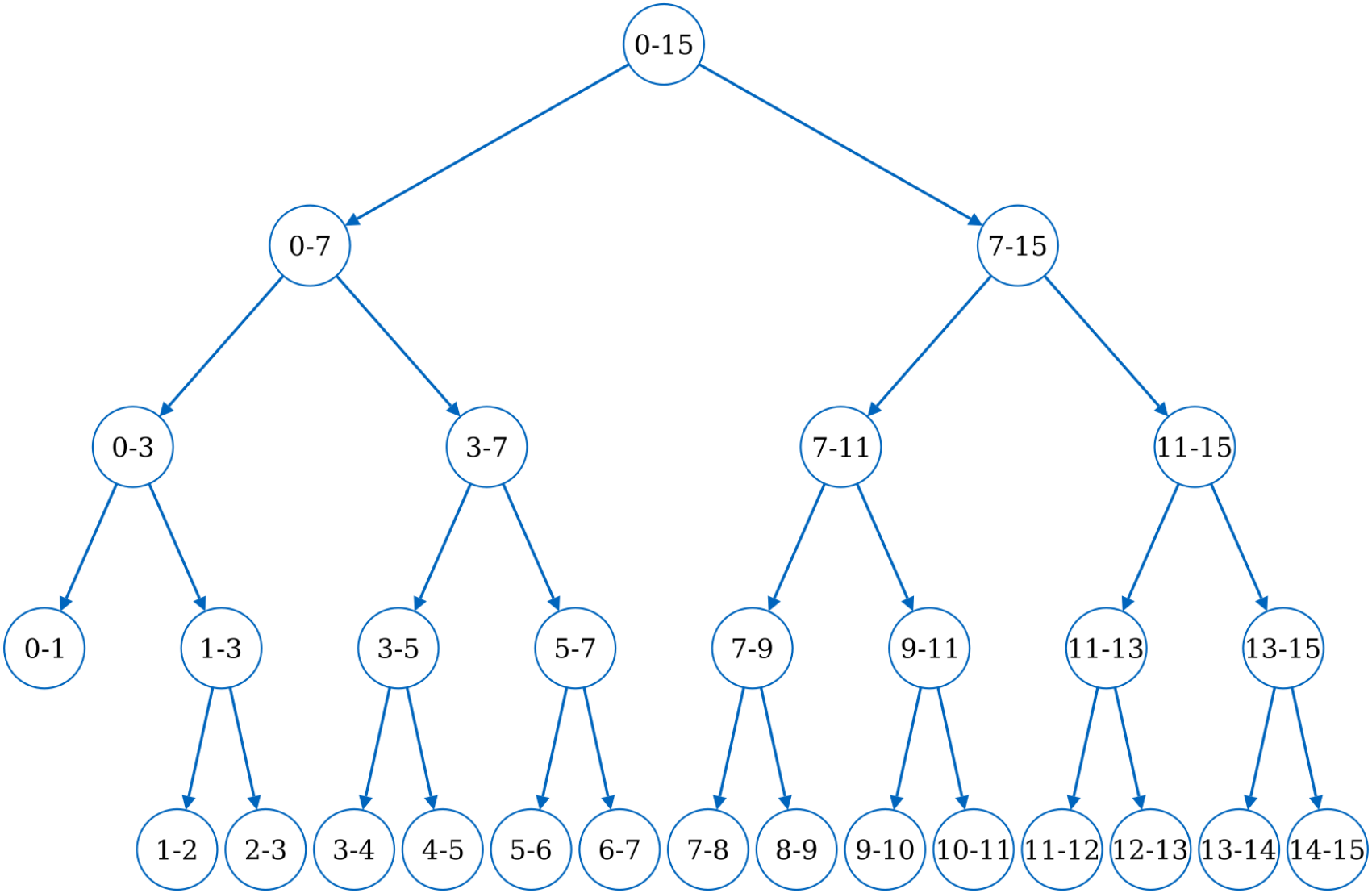

```
1 from mergeSort import mergeSort
2
3 a = [2,-1,0,5,7,7,1]
4 b = mergeSort(a)
5 print(a)
6 print(b)
```

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

https://www.youtube.com/watch?v=99lE5jwQXC8

Mergesort

- 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 $[\ldots a_j \ldots]$

Quick sort

- 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

```
1 def quickSortInternal(a, low, high):
2     if low >= 0 and high >=0 and low < high:
3         pivot = partition(a,low, high)
4         quickSortInternal(a, low, pivot)
5         quickSortInternal(a, pivot+1, high)
6
7 def quickSort(a):
8     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]`

```
1  def partition(a, low, high):
2      pivot = a[ (low + high) // 2 ]
3      i = low-1
4      j = high+1
5      while True:
6          i += 1
7          while a[i] < pivot:
8              i += 1
9          j -= 1
10         while a[j] > pivot:
11             j -= 1
12         if i >= j:
13             return j
14         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

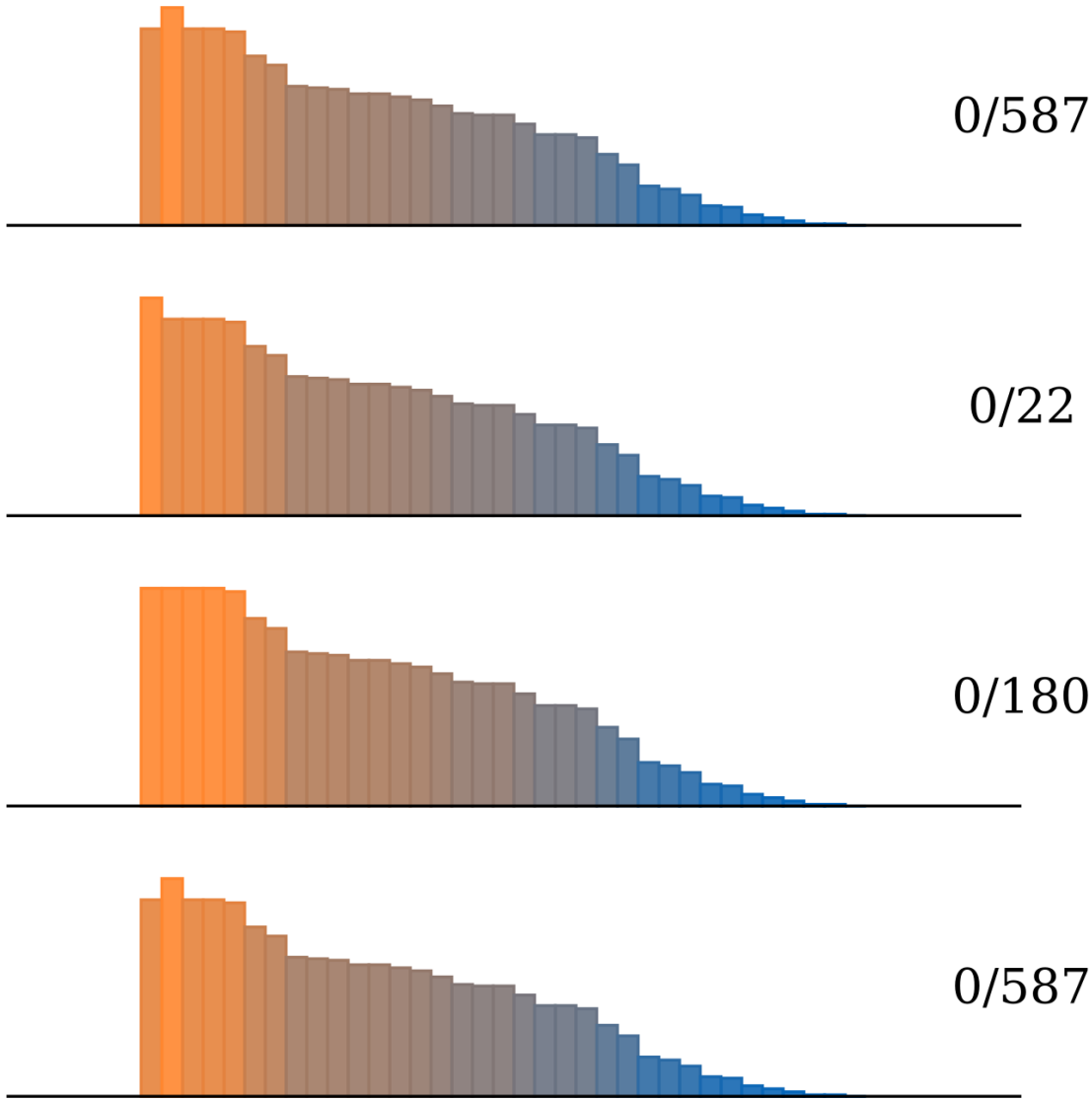
QuickSort + InsertionSort

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

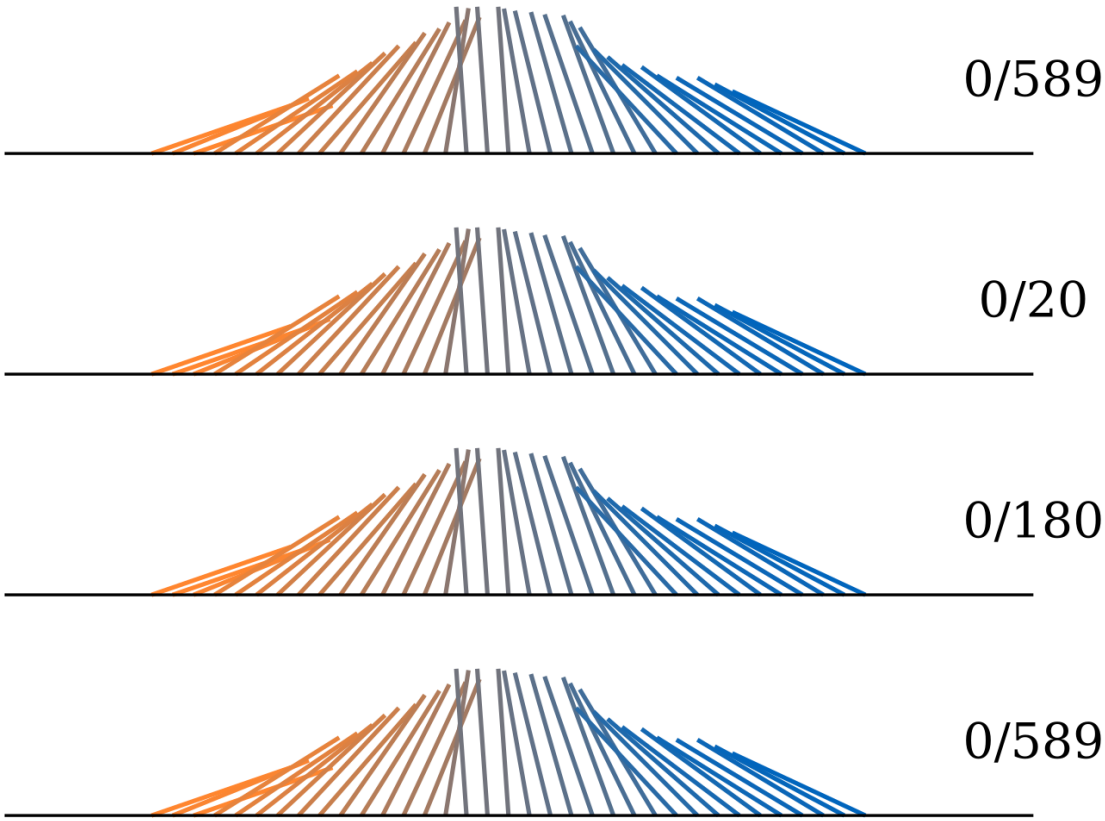
Rekurze: převod na nerekurzivní formu

- 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?