IB113 Úvod do programování a algoritmizace
Přednáška 6

Algoritmy a složitost; vyhledávání a řazení

Nikola Beneš

23. říjen 2017

Co bude dnes?

Příklady jednoduchých algoritmů

- použití zajímavých myšlenek
- lehký náhled na (časovou) složitost algoritmů

Vyhledávání a řazení

- ukázky a porovnání různých způsobů řešení

Z minula

Dělitelé čísla

```
def divisors(num):
    result = []
    for divisor in range(1, num + 1):
        if num % divisor == 0:
            result.append(divisor)
    return result
```

- jak *složité* je toto řešení?
  - kolik času (zhruba) zabere (vzhledem k danému vstupu)
- dalo by se nějak vylepšit?
  - počítat jen do `num // 2` (a přidat `num` na konec)
  - kolik času zabere potom?

Dělitelé: vylepšení

- jiné vylepšení?
  - využít toho, že `divisor` a `num // divisor` jsou oba dělitelé
  - menší z těchto dvou dělitelů je $\leq$ odmocnině z `num`

```
def divisors(num):
    limit = int(math.sqrt(num))
    result = []
    for divisor in range(1, limit + 1):
        if num % divisor == 0:
            result += [divisor, num // divisor]
    return result
```

- má tento kód nějaké problémy?
- kolik (zhruba) času zabere?
- co kdybychom chtěli seřazený seznam dělitelů?

Příklad: Otrávená studna

- máme osm studen, víme, že (právě) jedna z nich je otrávená
- laboratorní rozbor
  - pozná přítomnost jedu ve vodě
  - je drahý
- kolik rozborů potřebujeme k tomu, abychom s jistotou našli otrávenou studnu?

Řešení: stačí *tři* rozbory

- každý rozbor nám umožní zmenšit „okruh podezřelých“ na polovinu
  - můžeme smíchat vodu z více studní

Příklad: Otrávená studna

- máme osm studen, víme, že (právě) jedna z nich je otrávená
- laboratorní rozbor
  - pozná přítomnost jedu ve vodě
  - je drahý
  - je navíc časově náročný
- kolik rozborů potřebujeme k tomu, abychom s jistotou našli otrávenou studnu?
  - všechny rozbory chceme dělat zároveň

Řešení: stále stačí *tři* rozbory

- jeden rozbor: voda ze studní č. 1, 2, 3, 4
- druhý rozbor: voda ze studní č. 1, 2, 5, 6
- třetí rozbor: voda ze studní č. 1, 3, 5, 7
- jak poznáme, která studna je otrávená?

Příklad: Hádání čísla

- myslím si přirozené číslo mezi 1 a 1000 (včetně)
- povoleny pouze otázky typu „Je myšlené číslo menší než N?“
- kolik otázek potřebujete na odhalení čísla?
- *těžší*: kolik *předem formulovaných otázek* potřebujete?
- kdybyste měli pouze k dispozici pouze K otázek,
  v jakém rozsahu jste schopni čísla hádat?

Příklad: Hádání čísla

Řešení

- půlení intervalu
- N čísel: potřebujeme cca $\log_2 N$ otázek
- K otázek: umíme vyřešit rozsah velikosti 2^K

Řešení těžší varianty (předem formulované otázky)

- dotazy na bity v bitovém zápisu
- totéž jsme dělali u druhé varianty otrávené studny
- opět stačí cca $\log_2 N$ otázek

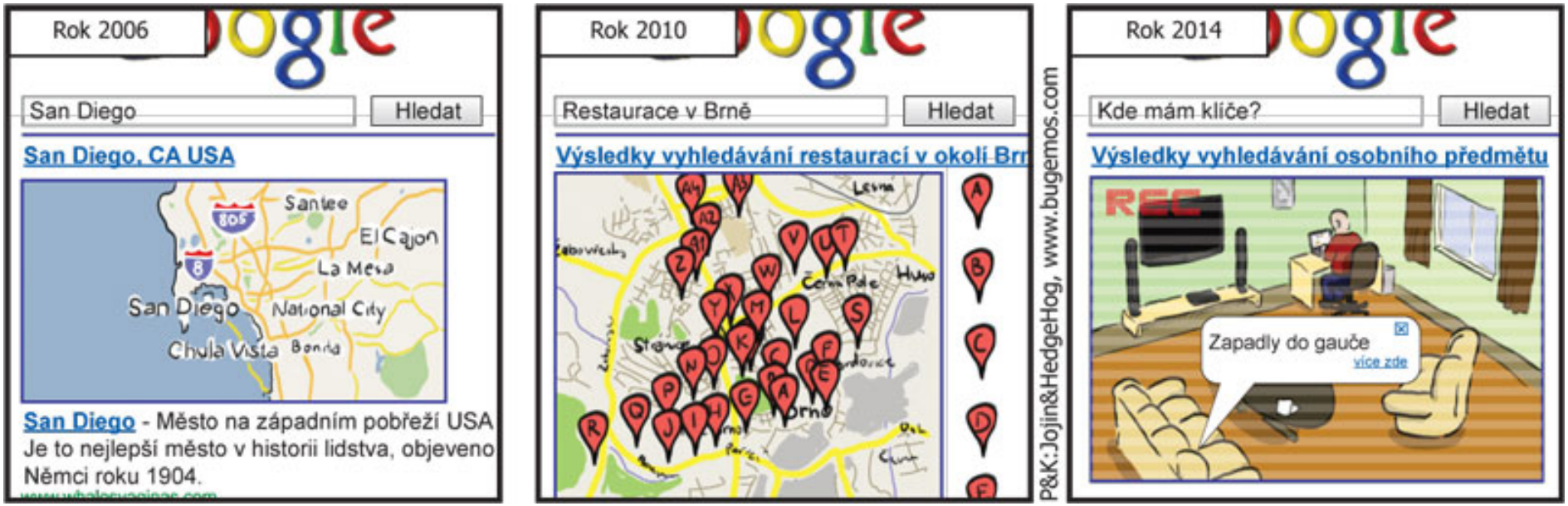
Připomenutí: logaritmus

$$x = b^y \text{ právě tehdy, když } y = \log_b(x)$$

$$\log_{10}(1000) = 3 \qquad \log_3(81) = 4$$
$$\log_2(16) = 4 \qquad \log_2(2) = 1$$
$$\log_2(1024) = 10 \qquad \log_5(1) = 0$$
$$\log_2(\sqrt{2}) = 0.5 \qquad \log_{0.5}(4) = -2$$

- $\log_b(x \cdot y) = \log_b(x) + \log_b(y)$
- $b^{\log_b(x)} = x$

Vyhledávání



- častý problém:
  - web, slovník, informační systémy
  - dílčí krok v mnoha algoritmech

zdroj obrázku: `http://www.bugemos.com/`

Vyhledávání v obecném seznamu

- vstup: seznam čísel + dotaz (číslo)
- výstup: odpověď `True`/`False`, jestli je číslo v seznamu
- alt. výstup: index čísla v seznamu (nějaká speciální hodnota, např. `None`, pokud číslo v seznamu není)

- v nejhorším případě musíme projít celý seznam
- časová složitost: cca délka seznamu
- jde to lépe? v obecném seznamu ne

Vyhledávání

Vyhledávání v seřazeném seznamu

- vstup: seřazený seznam čísel + dotaz (číslo)
- výstup: odpověď `True`/`False`, jestli je číslo v seznamu, příp. index

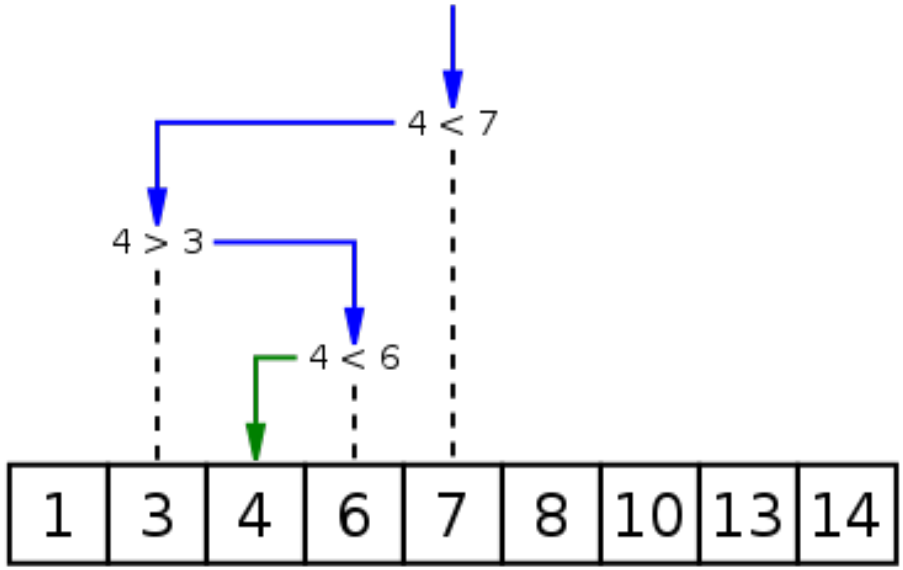
Řešení

- „naivní“ řešení: procházení celého seznamu
  - složitost cca délka seznamu (*lineární*)
  - použitelné pro krátké seznamy
  - nepoužitelné pro delší seznamy
- „rozumné“ řešení: půlení intervalu
  - složitost *logaritmická* k délce seznamu
  - jak implementovat?

Vyhledávání

Binární vyhledávání

- půlení intervalu (podobně jako hádání čísel, výpočet odmocniny, ...)
- podíváme se na *prostřední* prvek seznamu
  - podle jeho hodnoty buď skončíme, nebo pokračujeme doleva nebo doprava
  - udržujeme si „dolní“ a „horní“ mez



Vyhledávání

```
def binary_search(haystack, needle):
    lower_bound = 0
    upper_bound = len(haystack) - 1
    while lower_bound <= upper_bound:
        middle = (lower_bound + upper_bound) // 2
        if needle == haystack[middle]:
            return True

        if needle < haystack[middle]:
            upper_bound = middle - 1
        else:
            lower_bound = middle + 1

    return False
```

Složitost algoritmů (letmo)

Časová složitost

- funkce, která *délce vstupu* přiřazuje *počet kroků*
- typicky měříme nejhorší případ
  - (existují i jiné možnosti)
- je třeba si ujasnit:
  - co jsou základní kroky, které počítáme
  - co je *velikost vstupu*

- při počítání se seznamy (většinou):
  - základní kroky jsou operace s jednotlivými prvky
  - velikost vstupu je délka seznamu

Složitost algoritmů (letmo)

Asymptotická časová složitost

- zajímá nás pouze rychlost růstu funkcí
- třída $\mathcal{O}(f(n))$ – funkce, které rostou stejně rychle jako $f(n)$ nebo pomaleji
- zanedbáváme nedůležité části složitostní funkce
  - násobení konstantou
  - přičítání konstanty
  - nevedoucí členy polynomů, apod.
- příklady:
  - třída $\mathcal{O}(n)$ zahrnuje všechny *lineární* funkce $(n,\ 3n+7,\ 1000n+9,\ \ldots)$
  - třída $\mathcal{O}(n^2)$ zahrnuje všechny *kvadratické* funkce $(n^2,\ 6n^2+11,\ 20n^2-17n+9,\ n^2+n+3,\ \ldots)$
  - třída $\mathcal{O}(\log n)$ zahrnuje všechny logaritmické funkce (o základu > 1)

Vyhledávání a datové struktury

- seřazený seznam
  - umíme rychle vyhledávat, ale přidávání prvků je pomalé
  - proč? potřebujeme udržovat seřazenost a přidávání dovnitř seznamu je lineární

- jiné datové struktury pro rychlé vyhledávání, přidávání i ubírání prvků (pro zajímavost)
  - vyhledávací stromy
  - hašovací tabulky
  - v Pythonu: uvidíme později

Řadicí algoritmy

- oblíbené algoritmické téma
- anglicky „sorting algorithms“, česky někdy „třídicí algoritmy“
- cílem je seřadit data v seznamu
- existuje mnoho různých algoritmů
- většina programovacích jazyků má ve své standardní knihovně funkci `sort()` nebo podobnou

- proč se tím tedy zabýváme?
  - ukázka programů se seznamy
  - algoritmy s různou myšlenkou
  - „programátorská tradice“

Řadicí algoritmy

Vizualizace apod.

- `http://sorting.at`
- `http://www.sorting-algorithms.com`
- google: `sorting algorithms`

- `https://www.youtube.com/watch?v=lyZQPjUT5B4`

Řadicí algoritmy

Problém řazení

- vstup: seznam čísel (nebo obecněji nějakých porovnávatelných prvků)
- výstup: seřazený seznam, který
  - obsahuje stejné prvky, co vstupní seznam
  - ve stejném počtu

- příklad:
  - vstup: [7, 0, -3, 1, 100, 17, 42, 0]
  - výstup: [-3, 0, 0, 1, 7, 17, 42, 100]

Řazení: První pokus

Řešení

- zkoušíme systematicky všechna možná uspořádání prvků v seznamu
- pro každé z nich ověříme, jestli je seznam seřazený

- je toto dobrý algoritmus?
- je to korektní algoritmus? ano
  - dělá to, co po něm chceme? ano
- je to efektivní algoritmus? ne
  - kolik času zabere? záleží na vstupu
  - kolik času zabere v nejhorším případě?
    exponenciálně mnoho vzhledem k délce seznamu

Řazení: Lepší řešení

- zkuste vymyslet lepší řešení
- různé principy
- co nejefektivnější
- inspirace:
  - jak řadíte karty?
  - jak byste seřadili 200 kartiček se jmény lidí podle abecedy?
  - jak by se seřadili lidé stojící v úzké chodbě podle abecedy?

Řazení výběrem (Select Sort)

- vyberu nejmenší prvek seznamu a zařadím jej na správné místo
- vyberu nejmenší prvek zbytku a zařadím jej na správné místo
- atd.

- jak implementovat?
  - vytváření nového seznamu
  - modifikace původního seznamu
  - kterou variantu preferujeme?
  - co se bude snadněji implementovat?
  - co bude efektivnější?

Řazení výběrem (Select Sort)

```
def select_sort(my_list):
    size = len(my_list)
    for i in range(size):
        selected = i

        for j in range(i + 1, size):
            if my_list[j] < my_list[selected]:
                selected = j

        my_list[selected], my_list[i] = my_list[i], my_list[se
```

- jaká je složitost tohoto algoritmu
  - v nejlepším případě? $\mathcal{O}(n^2)$
  - v nejhorším případě? $\mathcal{O}(n^2)$

Řazení vkládáním (Insert Sort)

- inspirace řazením karet
- udržuji si seřazenou část
- vezmu jeden prvek z neseřazené části a zařadím jej do seřazené části

- jak implementovat?
  - vytvoření nového seznamu
  - modifikace původního seznamu
  - kterou variantu preferujeme?
  - co se bude snadněji implementovat?
  - co bude efektivnější?

Řazení vkládáním (Insert Sort)

```
def insert_sort(my_list):
    size = len(my_list)
    for i in range(1, size):
        # my_list[0:i - 1] je seřazená část
        # my_list[i:] je zatím neseřazená
        current = my_list[i]
        j = i
        while j > 0 and my_list[j - 1] > current:
            my_list[j] = my_list[j - 1]
            j -= 1
        my_list[j] = current
```

- jaká je složitost tohoto algoritmu
  - v nejlepším případě? $\mathcal{O}(n)$
  - v nejhorším případě? $\mathcal{O}(n^2)$

Řazení – další algoritmy (pro zajímavost)

Bubble Sort

- opakovaně procházím seznam
- pokud najdu dvojici prvků, která je špatně seřazená, prohodím je
- (teoreticky podobná složitost jako Insert Sort, ale prakticky velmi špatný algoritmus)

Quick Sort

- vyberu tzv. pivota (jeden prvek)
- rozdělím seznam na: menší než pivot, pivot, větší než pivot
- rekurzivně seřadím vzniklé menší seznamy
- složitost v nejhorším případě $\mathcal{O}(n^2)$
- v průměrném případě $\mathcal{O}(n \log n)$
  - závisí na způsobu výběru pivota
- v praxi funguje většinou velmi dobře

Řazení – další algoritmy (pro zajímavost)

Merge Sort

- rozdělíme seznam na dvě poloviny
- rekurzivně seřadíme vzniklé menší seznamy
- dva seřazené seznamy spojíme do jednoho: „merge“
- složitost v nejhorším případě $\mathcal{O}(n \log n)$
  - to je teoreticky to nejlepší, co jde (pro algoritmy založené na porovnávání)
- potřebuje extra prostor

... a jiné

- zatím neznáme ideální řadicí algoritmus
  - několik podmínek, žádný nesplňuje všechny
- v praxi se dost často používá kombinace různých přístupů
  - Python: *Timsort* (kombinace Merge a Insert Sortu)
  - C++: *Introsort* (kombinace Insert, Quick a Heap Sortu)

Vyhledávání a řazení v Pythonu

- `x in my_list` – test, zda je `x` v seznamu
  - prohledává lineárně celý seznam
- `my_list.index(x)` – pozice `x` v seznamu
- `my_list.count(x)` – počet výskytů `x` v seznamu
- `my_list.sort()` – seřadí seznam
  - tj. modifikuje zadaný seznam

- `sorted(my_list)` – vytvoří seřazený seznam ze zadaných prvků
  - nemodifikuje zadaný seznam
  - funguje i pro jiné datové struktury (řetězce, ntice, ...)

- datové struktury, v nichž se snadněji vyhledává (uvidíme příště)

Různé způsoby řazení

```
s = ["abeceda", "elektrika", "letadlo", "James Bond", "oko"]

print(s)
print(sorted(s))
print(sorted(s, reverse=True))
print(sorted(s, key=str.lower))
print(sorted(s, key=len))

def count_es(text):
    return text.count("e")

print(sorted(s, key=count_es))

# nebo lépe s použitím anonymní funkce (tzv. lambda)
print(sorted(s, key=lambda x: x.count("e")))
```

Použití řazení

Přesmyčky

- chceme poznat, jestli jsou dva řetězce vzájemnými přesmyčkami (anagramy)
- příklad: *chleba* – *blecha*, *reklama* – *karamel*
- jak na to?

```
def anagram(word1, word2):
    return sorted(word1) == sorted(word2)
```

Použití řazení

Unikátní prvky

- chceme ze seznamu vypsat všechny jeho prvky, ale bez opakování
- přímočaré řešení: opakované procházení seznamu (s použitím `in`)
- lepší řešení: vybírat ze seřazeného seznamu
- ještě lepší řešení: použít jinou datovou strukturu

Použití řazení

Unikátní prvky

```
def unique_elements(my_list):
    result = []
    for element in my_list:
        if element not in result:
            result.append(element)
    return result
```

- jakou má toto řešení složitost? $\mathcal{O}(n^2)$

Použití řazení

Unikátní prvky

```
def unique_elements(my_list):
    result = []
    new_list = sorted(my_list)  # nechceme měnit vstup
    for i in range(len(new_list)):
        if i == 0 or new_list[i - 1] != new_list[i]:
            result.append(new_list[i])
    return result
```

- jakou má toto řešení složitost? $\mathcal{O}(n \log n)$

Použití řazení

Unikátní prvky

```
def unique_elements(my_list):
    return list(set(my_list))
```

- uvidíme příště

Závěr

Co jsme se dnes dozvěděli?

- (letmo) jak se uvažuje o složitosti algoritmů
- princip půlení intervalu a jeho využití v algoritmech
- binární vyhledávání
- problém řazení, řadicí algoritmy
  - Insert Sort, Select Sort
- řazení v Pythonu

Co bude příště?

- trochu obecněji o datových typech
- další datové typy v Pythonu a jejich použití
  - množina, slovník