

Vyhledávání, řazení, složitost

IB111 Úvod do programování

2016

Výpočet odmocniny

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

Jak na to?

Výpočet odmocniny

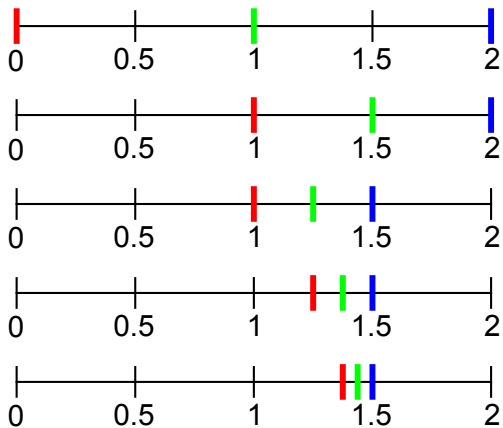
- vstup: číslo x
- výstup: přibližná hodnota \sqrt{x}

Jak na to?

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

Výpočet odmocniny: binární půlení

spodní odhad střed horní odhad



Výpočet odmocniny: binární půlení

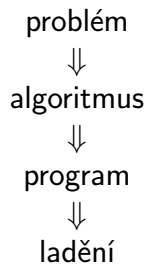
```
def square_root(x, precision = 0.01):  
    upper = x  
    lower = 0  
    middle = (upper + lower) / 2  
    while abs(middle**2 - x) > precision:  
        if middle**2 > x:  
            upper = middle  
        if middle**2 < x:  
            lower = middle  
        middle = (upper + lower) / 2  
    return middle
```

Výpočet odmocniny – chyba

Drobný problém: Program není korektní.

Kde je chyba?

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



Poznámka o ladění

- laděním se nebudeme (na přednáškách) příliš zabývat
- to ale neznamená, že není důležité...

Ladění je dvakrát tak náročné, jak psaní vlastního kódu. Takže pokud napíšete program tak chytře, jak jen umíte, nebudete schopni jej odladit. (Brian W. Kernighan)

Do průšvihů nás nikdy nedostane to, co nevíme. Dostane nás tam to, co víme příliš jistě a ono to tak prostě není. (Y. Berry)

- ladící výpisy
 - např. v každé iteraci cyklu vypisujeme stav proměnných
 - doporučeno vyzkoušet na ukázkových programech ze slidů
- použití debuggeru
 - dostupný přímo v IDLE
 - sledování hodnot proměnných, spuštěných příkazů, breakpointy, ...
 - více: cvičení, pozdější přednáška

Otrávené studny

- 8 studen, jedna z nich je otrávená
- laboratorní rozbor
 - dokáže rozpoznat přítomnost jedu ve vodě
 - je drahý
- kolik rozborů potřebujeme?
- jak určit otrávenou studnu?

Otrávené studny II

- 8 studen, jedna z nich je otrávená
- laboratorní rozbor
 - dokáže rozpoznat přítomnost jedu ve vodě
 - je drahý
 - je časově náročný (1 den)
- jak určit otrávenou studnu za 1 den pomocí 3 paralelních rozborů?

Otrávené studny: řešení

Řešení s využitím binárních čísel

studna	kód	studna	kód
A	000	E	100
B	001	F	101
C	010	G	110
D	011	H	111

test	přidělené studny
1	B, D, F, H
2	C, D, G, H
3	E, F, G, H

Vyhledávání: hra

- Myslím si přirozené číslo X mezi 1 a 1000.
- Povolená otázka: „Je X menší než N ?“
- Kolik otázek potřebujete na odhalení čísla?
- Kolik *předem formulovaných* otázek potřebujete?
- Mezi kolika čísla jste schopni odhalit skryté číslo na K otázek?

Vyhledávání: řešení

- „dynamické otázky“: půlení intervalu
- „předem formulované otázky“: dotazy na bity v bitovém zápisu (stejně jako u studen)
- N čísel: potřebujeme $\log_2 N$ otázek
- K otázek: rozlišíme mezi 2^K čísla

Připomenutí: logaritmus

$$x = b^y \Leftrightarrow y = \log_b(x)$$

$$\log_{10}(1000) = 3$$

$$\log_2(16) = 4$$

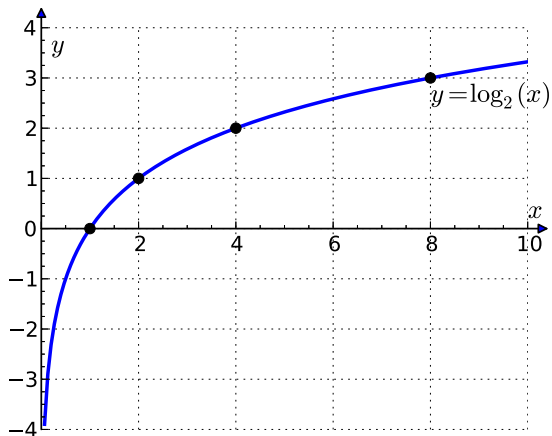
$$\log_2(1024) = 10$$

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

<http://www.khanacademy.org/math/algebra/logarithms>

<http://khanovaskola.cz/logaritmy/uvod-do-logaritmu>

Logaritmus – graf



$$\log_3(81) = ?$$

$$\log_2(2) = ?$$

$$\log_5(1) = ?$$

$$\log_{10}(0.1) = ?$$

$$\log_2(\sqrt{2}) = ?$$

$$\log_{0.5}(4) = ?$$

vyhledávání v (připravených) datech je velmi častý problém:

- web
- slovník
- informační systémy
- dílčí krok v algoritmech

Vyhledávání: konkrétní problém

- vstup: seřazená posloupnost čísel + dotaz (číslo)
- výstup: pravdivostní hodnota (True/False)
příp. index hledaného čísla v posloupnosti (-1 pokud tam není)

příklad:

- vstup: 2, 3, 7, 8, 9, 14 + dotaz 8
- výstup: True, resp. 3 (číslování od nuly)

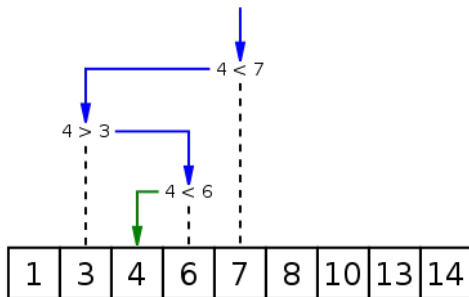
Vyhledávání a logaritmus

- naivní metoda = průchod seznamu
 - **lineární** vyhledávání, $O(n)$
 - pomalé (viz např. databáze s milióny záznamů)
 - jen velmi krátké seznamy
- základní rozumná metoda = půlení intervalu
 - **logaritmický** počet kroků (vzhledem k délce seznamu), $O(\log(n))$

Vyhledávání: půlení intervalu

- binární vyhledávání
- podobné jako: hra s hádáním čísel, aproximace odmocniny
- podíváme se na prostřední člen \Rightarrow podle jeho hodnoty pokračujeme v levém/pravém intervalu
- udržujeme si „horní mez“ a „spodní mez“

Binární vyhledávání – ilustrace



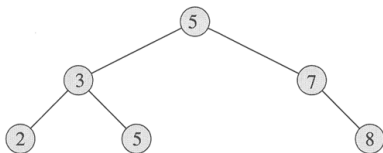
Wikipedia

Vyhledávání: program

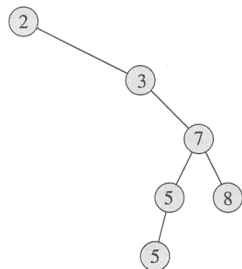
```
def binary_search(value, alist):
    lower_bound = 0
    upper_bound = len(alist) - 1
    while lower_bound <= upper_bound:
        middle = (lower_bound + upper_bound) // 2
        if alist[middle] == value:
            return True
        elif alist[middle] > value:
            upper_bound = middle - 1
        else:
            lower_bound = middle + 1
    return False
```

Vyhledávání, přidávání, ubírání

- seřazený seznam – rychlé vyhledávání, ale pomalé přidávání prvků
- rychlé vyhledávání, přidávání i ubírání prvků – datová struktura slovník; vyhledávací stromy, hašovací tabulky
- více později / v IB002



(a)



(b)

Řadicí algoritmy: terminologická poznámka

- anglicky „sorting algorithm“
- česky používáno: řadicí algoritmy nebo třídící algoritmy
- řadicí vesměs považováno za „správnější“

- mnoho různých algoritmů pro stejný účel
- většina programovacích jazyků má vestavěnou podporu (funkce `sort()`)

Proč se tím tedy zabýváme?

Proč se tím tedy zabýváme?

- 1 ukázka programů se seznamy
- 2 ilustrace algoritmického myšlení, technik návrhu algoritmů
- 3 typický příklad drobné změny algoritmu s velkým dopadem na rychlost programu
- 4 hezky se to vizualizuje a vysvětluje
- 5 tradice, patří to ke vzdělání informatika
- 6 občas se to může i hodit

Řadicí algoritmy: komentář

- zde důraz na jednoduché algoritmy, základní použití seznamů, intuici
- detailněji v IB002 Algoritmy a datové struktury I
 - pokročilejší algoritmy
 - důkazy korektnosti
 - složitost formálně

Doporučené zdroje

- <http://www.sorting-algorithms.com/>
 - animace
 - kódy
 - vizualizace
- <http://sorting.at/>
 - elegantní animace
- více podobných: Google → sorting algorithms
- A na zpestření: <http://www.youtube.com/watch?v=1yZQPjUT5B4>

Řadící algoritmy: problém

- vstup: posloupnost (přirozených) čísel
např. 8, 2, 14, 3, 7, 9
- výstup: seřazená posloupnost
např. 2, 3, 7, 8, 9, 14

pozn. většina zmíněných algoritmů aplikovatelná nejen na čísla, ale na „cokoliv, co umíme porovnávat“

Pokus č. 1

- zkusíme systematicky všechna možná uspořádání prvků
- pro každé z nich ověříme, zda jsou prvky korektně uspořádány
- je to dobrý algoritmus?

Co vy na to?

- zkuste vymyslet
 - řadicí algoritmus
 - co nejvíce různých principů
 - co nejefektivnější algoritmus
- možná inspirace: jak řadíte karty?

n – délka vstupní posloupnosti

	počet operací
jednoduché algoritmy	$O(n^2)$
složitější algoritmy	$O(n \log(n))$

Bublíkové řazení (Bubble sort)

- „probublávání“ vyšších hodnot nahoru
- srovnávání a prohazování sousedů
- po i iteracích je nejvyšších i členů na svém místě

Bublňkové řazení: program

```
def bubble_sort(a):  
    n = len(a)  
    for i in range(n):  
        for j in range(n-i-1):  
            if a[j] > a[j+1]:  
                tmp = a[j]  
                a[j] = a[j+1]  
                a[j+1] = tmp
```

invariant cyklu: $a[n-i-1:]$ ve finální pozici

Bublinkové řazení: příklad běhu

[8, 2, 7, 14, 3, 1]

[2, 7, 8, 3, 1, 14]

[2, 7, 3, 1, 8, 14]

[2, 3, 1, 7, 8, 14]

[2, 1, 3, 7, 8, 14]

[1, 2, 3, 7, 8, 14]

Implementační detail: prohazování prvků

- prohození hodnot dvou proměnných a , b
- na slidech psáno „běžným“ způsobem pomocí pomocné proměnné: $t = a$; $a = b$; $b = t$
- Python umožňuje zápis: $a, b = b, a$

Řazení výběrem (Select sort)

- řazení výběrem
- projdeme dosud neseřazenou část seznamu a vybereme nejmenší prvek
- nejmenší prvek zařadíme na aktuální pozici (výměnou)

Řazení výběrem: program

```
def select_sort(a):  
    for i in range(len(a)):  
        selected = i  
        for j in range(i+1, len(a)):  
            if a[j] < a[selected]: selected = j  
        tmp = a[i]  
        a[i] = a[selected]  
        a[selected] = tmp
```

Řazení vkládáním (Insert sort)

- podobně jako „řazení karet“
- prefix seznamu udržujeme seřazený
- každou další hodnotu zařadíme tam, kam patří

Řazení vkládáním: program

```
def insert_sort(a):  
    for i in range(1, len(a)):  
        current = a[i]  
        j = i  
        while j > 0 and a[j-1] > current:  
            a[j] = a[j-1]  
            j -= 1  
        a[j] = current
```

Význam proměnných

- proměnná `selected` u řazení výběrem
 - **index** vybraného prvku
 - používáme k indexování, `a[selected]`
- proměnná `current` u řazení vkládáním
 - **hodnota** „posunovaného“ prvky
 - `a[j] = current`
- v našich případech mají stejný typ (`int`), ale jiný význam a použití (záměna = častý zdroj chyb)
- zřejmější pokud řadíme třeba řetězce

Quicksort

- rekurzivní algoritmus
- vybereme „pivota“ a seznam rozdělíme na dvě části:
 - větší než pivot
 - menší než pivot
- obě části pak nezávisle seřadíme (rekurzivně pomocí quicksortu)
- pokud máme smůlu při výběru pivota, tak je stejně pomalý jako předchozí
- v průměrném případě je rychlý – *quick*
 $O(n \log(n))$

Řazení slučováním (Merge sort)

- rekurzivní algoritmus
- seznam rozdělíme na dvě poloviny a ty seřadíme (pomocí Merge sort)
- ze seřazených polovin vyrobíme jeden seřazený seznam – „zipování“
- vždy efektivní – $O(n \log(n))$

Specifické předpoklady – efektivnější algoritmus

- předchozí algoritmy využívají pouze operaci porovnání dvou hodnot
- aplikovatelné na cokoliv, co lze porovnávat, žádné další předpoklady
- s doplňujícími předpoklady můžeme dostat nové algoritmy (obecný princip)
- řazení (krátkých) čísel → Radix sort

Radix sort

- seznam („krátkých“) čísla
- postupujeme od nejméně významné cifry k nejvýznamnější
- seřadíme čísla podle dané cifry = rozdělení do 10 „kyblíčků“ (jednoduché, rychlé)

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms.

Složitost trochu podrobněji

- složitost algoritmu – jak je algoritmus výpočetně náročný
- časová, prostorová
- měříme počet **operací** nikoliv čas na konkrétním stroji
- vyjadřujeme jako funkci délky vstupu
- O notace – zanedbáváme konstanty
- např. $O(n)$, $O(n \log(n))$, $O(n^2)$

Ilustrace rozdílů v složitosti

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10 ²⁰ cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10 ⁵⁸ cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10 ¹³⁵ cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10 ⁵⁹⁸ cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10 ¹²¹⁴ cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10 ²⁴⁴⁷ cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10 ⁴⁹¹³ cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10 ⁹⁸⁴⁵ cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10 ¹⁹⁷⁰⁹ cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10 ³⁹⁴³⁸ cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10 ⁷⁸⁸⁹⁴ cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10 ¹⁵⁷⁸⁰⁸ cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10 ³¹⁵⁶³⁴ cent

Table 1.1 Running times for different sizes of input. “nsec” stands for nanoseconds, “ μ ” is one microsecond and “cent” stands for centuries.

Složitost řadících algoritmů

n – délka vstupní posloupnosti

	počet operací
jednoduché algoritmy	$O(n^2)$
složitější algoritmy	$O(n \log(n))$

Python: Seznam funkcí

Pro zajímavost: v Pythonu můžeme mít třeba i seznam funkcí

```
def test_sorts():  
    for sort in [ bubble_sort, insert_sort, select_sort ]:  
        a = [41, 71, 46, 15, 97, 44, 30, 11]  
        sort(a)  
        print(a)
```

Vyhledávání a řazení v Pythonu

- `x in seznam` – test přítomnosti `x` v seznamu
- `seznam.index(x)` – pozice `x` v seznamu
- `seznam.count(x)` – počet výskytů `x` v seznamu
- `seznam.sort()` – seřadí položky seznamu
- `sorted(seznam)` – vrátí seřazené položky seznamu (ale nezmění vlastní proměnnou)

pro řazení používá Python Timsort – kombinaci řazení slučováním a vkládáním

Různé způsoby řazení

```
s = [ "prase", "Kos", "ovoce", "Pes", "koza",  
      "ovce", "kokos" ]  
  
print(sorted(s))  
print(sorted(s, reverse = True))  
print(sorted(s, key = str.lower))  
print(sorted(s, key = len))  
print(sorted(s, key = lambda x: x.count("o")))
```

Unikátní prvky, nejčastější prvek

- máme seznam prvků, např. výsledky dotazníku (oblíbený programovací jazyk):
["Python", "Java", "C", "Python", "PHP",
"Python", "Java", "JavaScript", "C",
"Pascal"]
- chceme:
 - seznam unikátních hodnot
 - nejčastější prvek

Unikátní prvky, nejčastější prvek

- máme seznam prvků, např. výsledky dotazníku (oblíbený programovací jazyk):
["Python", "Java", "C", "Python", "PHP", "Python", "Java", "JavaScript", "C", "Pascal"]
- chceme:
 - seznam unikátních hodnot
 - nejčastější prvek
- přímočaře: opakované procházení seznamu
- efektivněji: seřadit a pak jednou projít
- elegantněji: využití pokročilých datových struktur / konstrukcí

Unikátní prvky

```
def unique(alist):
    alist = sorted(alist)
    # rozdílné chování od alist.sort() !!
    result = []
    for i in range(len(alist)):
        if i == 0 or alist[i-1] != alist[i]:
            result.append(alist[i])
    return result
```

```
def unique(alist):
    return list(set(alist))
```

Nejčastější prvek přímočaře

```
def most_common(alist):
    alist = sorted(alist)
    max_value, max_count = None, 0
    current_value, current_count = None, 0
    for value in alist:
        if value == current_value:
            current_count += 1
        else:
            current_value = value
            current_count = 1
        if current_count > max_count:
            max_value = current_value
            max_count = current_count
    return max_value
```

Nejčastější prvek sofistikovaněji

```
def most_common(alist):  
    return max(alist, key=alist.count)
```

Stack Overflow diskuze:

<http://stackoverflow.com/questions/1518522/python-most-common-element-in-a-list>

- přesmyčky = slova poskládaná ze stejných písmen
- úkol: rozpoznat, zda dvě slova jsou přesmyčky
- vstup: dva řetězce
- výstup: True/False
- příklady:
 - odsun, dusno → True
 - kostel, les → False
 - houslista, souhlasit → True
 - ovoce, ovace → False

Přesmyčky řešení

- seřadíme písmena obou slov
- přesmyčky \Leftrightarrow po seřazení identické
- implementace za využití sorted přímočará

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

Rozměňování peněz

- vstup: částka X
- výstup: vyplacení částky pomocí co nejméně mincí (bankovek)
- předpokládejme „klasické“ hodnoty peněz: 1, 2, 5, 10, 20, 50, 100, ...
- příklady:
 - $29 \rightarrow 2, 2, 5, 20$
 - $401 \rightarrow 1, 200, 200$

Rozměňování peněz

- **hladový algoritmus:** „použij vždy nejvyšší minci, která je menší než cílová částka“
- cvičení – naprogramovat
- funguje pro klasické hodnoty
- nefunguje pro obecný případ – najděte konkrétní příklad
- zkuste vymyslet algoritmus pro obecný případ

- vyhledávání: půlení intervalu, rekurze
- řadicí algoritmy:
 - jednoduché (kvadratické): bublinkové, výběrem, vkládáním
 - složitější ($n \cdot \log(n)$, rekurzivní): quick sort, slučování
- příklady