Vyhledávání, řazení, složitost

IB111 Úvod do programování skrze Python

2015

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 čísly 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 čísly

Připomenutí: logaritmus

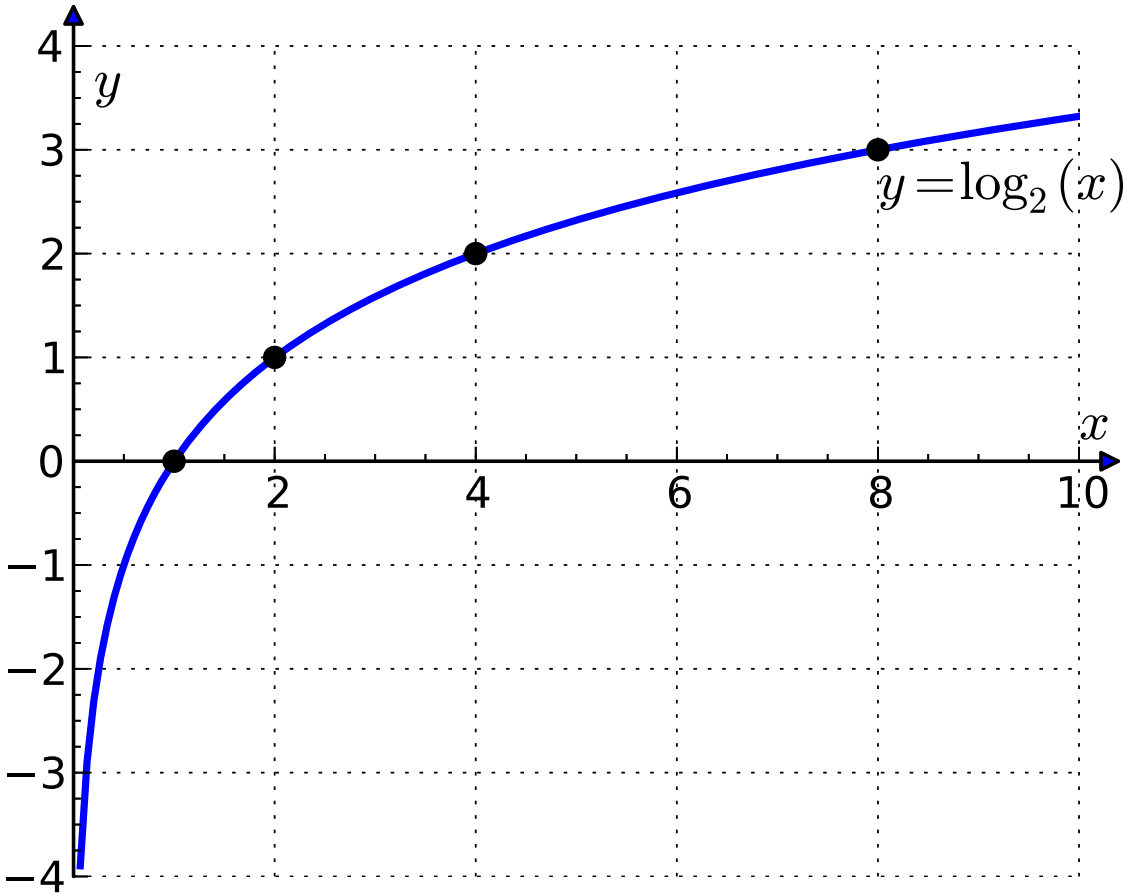
$$x = b^y \Leftrightarrow y = \log_b(x)$$

$$\begin{aligned}
\log_{10}(1000) &= 3 \\
\log_2(16) &= 4 \\
\log_2(1024) &= 10 \\
\log_b(xy) &= \log_b(x) + \log_b(y)
\end{aligned}$$

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

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

Logaritmus – graf



Logaritmus – test

$$
\begin{aligned}
\log_3(81) &= ? \\
\log_2(2) &= ? \\
\log_5(1) &= ? \\
\log_{10}(0.1) &= ? \\
\log_2(\sqrt{2}) &= ? \\
\log_{0.5}(4) &= ?
\end{aligned}
$$

Vyhledávání: motivace

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)

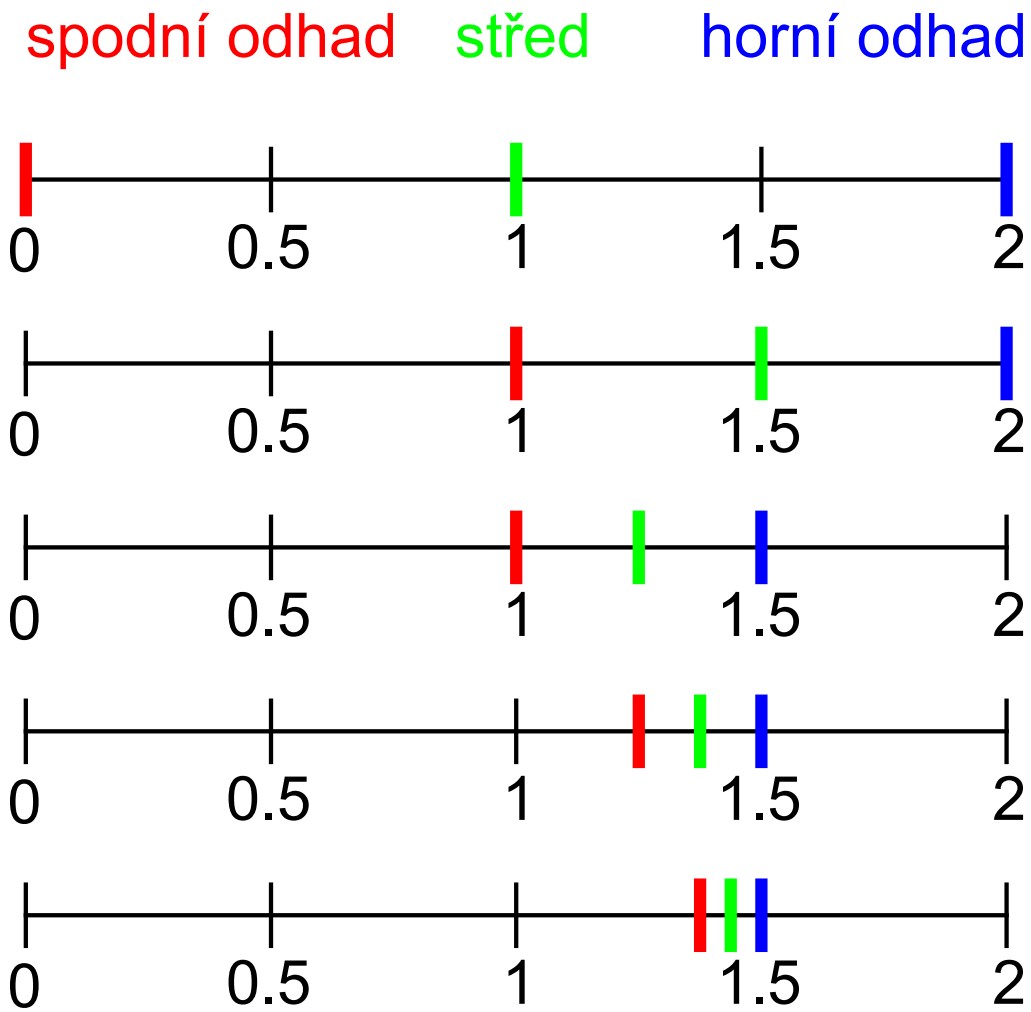
Vyhledávání a logaritmus

- 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“

Výpočet odmocniny: binární půlení



Výpočet odmocniny: binární půlení

```
def odmocnina(x, presnost = 0.01):
    horni_odhad = x
    spodni_odhad = 0
    stred = (horni_odhad + spodni_odhad) / 2.0
    while abs(stred**2 - x) > presnost:
        if stred**2 > x:
            horni_odhad = stred
        if stred**2 < x:
            spodni_odhad = stred
        stred = (horni_odhad + spodni_odhad) / 2.0
    return stred
```

Vyhledávání: program

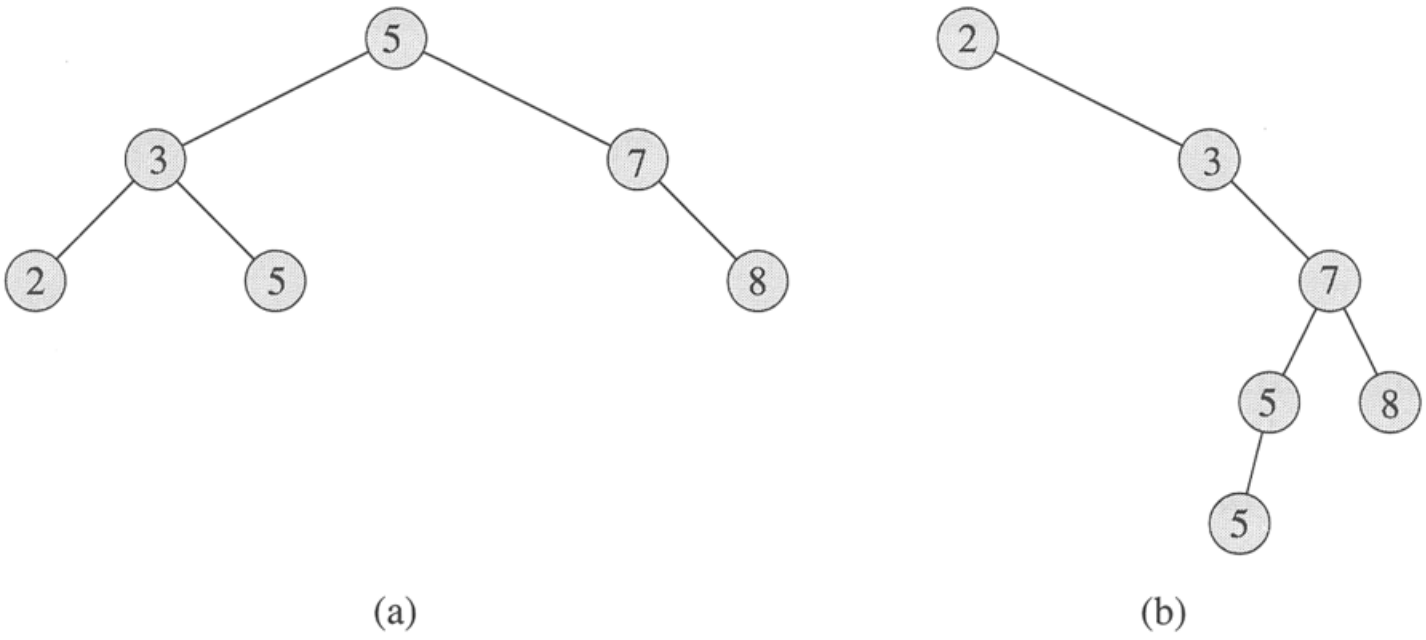
```
def binarni_vyhledavani(hodnota, seznam):
    spodni_mez = 0
    horni_mez = len(seznam) - 1
    while spodni_mez <= horni_mez:
        stred = (spodni_mez + horni_mez) / 2
        if seznam[stred] == hodnota:
            return True
        elif seznam[stred] > hodnota:
            horni_mez = stred - 1
        else:
            spodni_mez = stred + 1
    return False
```

Vyhledávání: rekurzivní varianta

```
def binarni_vyhledavani(hodnota, seznam,
                        spodni_mez, horni_mez):
    if spodni_mez > horni_mez:
        return False
    stred = (spodni_mez + horni_mez)/2
    if seznam[stred] < hodnota:
        return binarni_vyhledavani(hodnota, seznam,
                                   stred+1, horni_mez)
    elif seznam[stred] > hodnota:
        return binarni_vyhledavani(hodnota, seznam,
                                   spodni_mez, stred-1)
    else:
        return True
```

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



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

Řadicí algoritmy: terminologická poznámka

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

Řadicí algoritmy: komentář

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

Proč se tím tedy zabýváme?

Řadicí algoritmy: komentář

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 $\rightarrow$ sorting algorithms
- A na zpestření: `http://www.youtube.com/watch?v=lyZQPjUT5B4`

Řadicí 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 aplikovatená nejen na čísla, ale na „cokoliv, co umíme porovnávat“

Pokus č. 1

- zkoušíme 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?

Složitost

n – délka vstupní posloupnosti

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

Bublinkové řazení (Bubble sort)

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

Bublinkové řazení: program

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

Řazení vkládáním (Insert sort)

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

Řazení vkládáním: program

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

Význam proměnných

- proměnná `vybrany` u řazení výběrem
  - index vybraného prvku
  - používáme k indexování, `a[vybrany]`
- proměnná `aktualni` u řazení vkládáním
  - hodnota „posunovaného“ prvky
  - `a[j] = aktualni`
- 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)

Quick sort

- 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 – $O(n \log(n))$

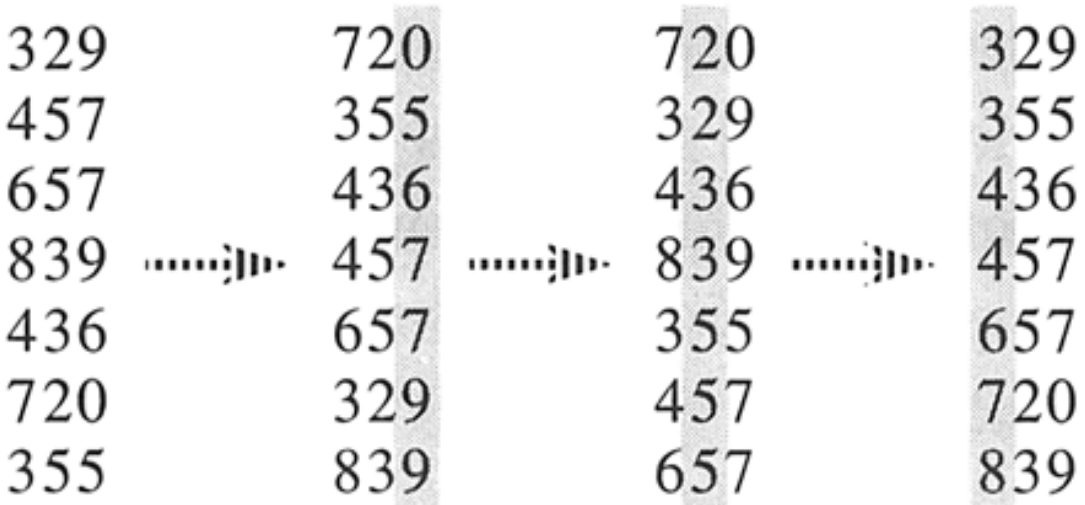
Radix sort

- 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)

Radix sort

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

Radix sort ilustrace



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.

M. H. Alsuwaiyel: Algorithms, Design Techniques and Analysis.

Složitost řadicích algoritmů

n – délka vstupní posloupnosti

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

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 unikatni(seznam):
    seznam = sorted(seznam)
    # rozdilne chovani od seznam.sort()
    vystup = []
    for i in range(len(seznam)):
        if i == 0 or seznam[i-1] != seznam[i]:
            vystup.append(seznam[i])
    return vystup

def unikatni(seznam):
    return list(set(seznam))
```

Nejčastější prvek

```
def nejcastejsi(seznam):
    seznam = sorted(seznam)
    max_prvek, max_pocet = None, 0
    akt_prvek, akt_pocet = None, 0
    for prvek in seznam:
        if prvek == akt_prvek:
            akt_pocet += 1
        else:
            akt_prvek = prvek
            akt_pocet = 1
        if akt_pocet > max_pocet:
            max_prvek = akt_prvek
            max_pocet = akt_pocet
    return max_prvek
```

Nejčastější prvek sofistikovaněji

```
def nejcastejsi(seznam):
    return max(seznam, key=seznam.count)
```

Stack Overflow diskuze:

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

Přesmyčky

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

Přesmyčky řešení

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

```
def presmycky(slovo1, slovo2):
  return sorted(slovo1) == sorted(slovo2)
```

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

Shrnutí

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