Číselné soustavy

- převod **z dvojkové soustavy** na číselnou hodnotu
- algoritmus: Hornerovo schéma

$$110010 \approx 1.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 0.2^0 =$$
$$= ((((1.2 + 1).2 + 0).2 + 0).2 + 1).2 + 0 \quad = 50$$

- převod **z šestnáctkové soustavy** na číselnou hodnotu

$$A1F \approx A.16^2 + 1.16^1 + F.16^0 =$$
$$= 10.16^2 + 1.16^1 + 15.16^0 =$$
$$= (10.16 + 1).16 + 15 \quad = 2591$$

```python
def bin_int(s):
    """
      převod binárního zápisu čísla (string s)
      na číselnou hodnotu
    """
    n = 0
    for i in range(len(s)):
        n = n * 2 + int(s[i])
    return n
```

```python
def hex_int(s):
    """
      převod hexadecimálního zápisu čísla (string s)
      na číselnou hodnotu
    """
    cifry={'A':10, 'B':11, 'C':12, 'D':13, 'E':14, 'F':15,
           'a':10, 'b':11, 'c':12, 'd':13, 'e':14, 'f':15}
    n = 0
    for i in range(len(s)):
        if s[i] in "0123456789":
            n = n * 16 + int(s[i])
        else:
            n = n * 16 + cifry[s[i]]
    return n
```

Jiné řešení
– kratší zápis, ale jsou povolena pouze velká písmena A, B, C, D, E, F:

```
def hex_int(s):
    """
      převod hexadecimálního zápisu čísla (string s)
      na číselnou hodnotu
    """
    cifry = "0123456789ABCDEF"
    n = 0
    for i in range(len(s)):
        n = n * 16 + cifry.index(s[i])
    return n
```

$110010 \approx 1.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 0.2^0 =$

$= ((((1.2 + 1).2 + 0).2 + 0).2 + 1).2 + 0 = 50$

- převod číselné hodnoty **do dvojkové soustavy**
- algoritmus: Hornerovo schéma využité v opačném směru
  posloupnost zbytků při celočíselném dělení dvěma
  tvoří **odzadu** dvojkový zápis čísla
  $\rightarrow$ připojování dvojkových cifer do stringu **zleva**

50 : 2 = 25, zb. 0
25 : 2 = 12, zb. 1
12 : 2 = 6, zb. 0
6 : 2 = 3, zb. 0
3 : 2 = 1, zb. 1
1 : 2 = 0, zb. 1

```
def int_bin(n):
    """
      převod čísla do dvojkové soustavy
      - obrácené Hornerovo schéma
    """
    s = ""
    while n > 0:
        s = str(n % 2) + s
        n //= 2
    return s
```

Pozor na časovou složitost – při této implementaci bude kvadratická vzhledem k délce vytvářeného binárního zápisu (při řetězení se vždy vytváří nový string, což vyžaduje lineární čas).

Řešení s lineární složitostí: jednotlivé cifry vkládat do seznamu (na konec seznamu), potom celý seznam otočit a převést na string.

```python
def int_hex(n):
    """
      převod čísla do šestnáctkové soustavy
      - obrácené Hornerovo schéma
    """
    s = ""
    cifry = "0123456789ABCDEF"
    while n > 0:
        s = cifry[n % 16] + s
        n //= 16
    return s
```

Rychlé umocňování

aplikace dvojkové soustavy

Úloha: spočítat hodnotu x^n, kde

- n je (velké) kladné celé číslo
- x může být reálné číslo (nebo třeba také matice)

Řešení:

1. přímočaře v čase $\Theta(n)$:

```
def mocnina1(x, n):
    """výpočet x**n lineárně"""
    v = 1
    for i in range(n):
        v *= x
    return v
```

2. rychleji v čase $\Theta(\log n)$:

postupně počítáme hodnoty $x, x^2, x^4, x^8, \dots$

a vhodné z nich násobíme do výsledku

Které hodnoty x^k jsou ty vhodné?

Pozorování: $x^{25} = x^{16} . x^8 . x^1$, neboť 25 = 16 + 8 + 1

- to je jednoznačný rozklad čísla *n* na součet mocnin dvojky

- jsou to ty mocniny dvojky, kde je jednička v binárním zápisu čísla *n*

Postup je tedy podobný, jak převod čísla do dvojkové soustavy.

```python
def mocnina2(x, n):
    """výpočet x**n rychleji"""
    v = 1
    while n > 0:
        if n % 2 == 1:
            v *= x
        x *= x
        n //= 2
    return v
```

Časová složitost $\Theta(\log n)$ – počet opakování while-cyklu.

Vyhledávání v seznamu

Úloha: Zjistěte, zda se v seznamu ***a*** nachází daná hodnota ***x***,
a pokud ano, tak kde.

Python:

test výskytu provádí operátor **in** `if x in a`

nebo také metoda count() `a.count(x)`

pozici prvního výskytu určuje metoda index() `a.index(x)`

(pokud tam není → chyba)

Základní algoritmus:

jeden průchod polem – časová složitost O(N)

1. for-cyklus

```
j = -1
for i in range(len(a)):
    if a[i] == x:
        j = i

if j == -1:
    print("Není tam")
else:
    print("Je na pozici", j)
```

Jednoduché, ale nešikovné (cyklus pokračuje i po nalezení ***x***).
V případě více výskytů najde poslední, nikoliv první výskyt.

2. for-cyklus s výskokem

```
j = -1
for i in range(len(a)):
    if a[i] == x:
        j = i
        break

if j == -1:
    print("Není tam")
else:
    print("Je na pozici", j)
```

Obdobné řešení, ale cyklus zbytečně nepokračuje po nalezení ***x***.
V případě více výskytů najde první.

3. while-cyklus

```
i = 0
while i < len(a) and a[i] != x:
    i += 1

if i == len(a):
    print("Není tam")
else:
    print("Je na pozici", i)
```

Vhodně zvolená složená podmínka a zkrácené vyhodnocování logických výrazů (zleva doprava, dokud není rozhodnuto o výsledku)

4. cyklus řízený proměnou typu boolean

```
i = 0
dalsi = True       #zpracovávat další prvek?
while dalsi:
    if a[i] == x:
        dalsi = False
        print("Je na pozici", i)
    elif i == len(a)-1:
        dalsi = False
        print("Není tam")
    else:
        i += 1
```

5. vyhledávání pomocí zarážky

→ zjednodušení podmínky ve while-cyklu

```
a.append(x)           # přidat zarážku (dočasně)
i = 0
while a[i] != x:
    i += 1
del a[-1]             # zrušit zarážku

if i == len(a):
    print("Není tam")
else:
    print("Je na pozici", i)
```

Hodnota ***x*** je v seznamu ***a*** vždy nalezena
– pokud tam původně nebyla, tak se najde v zarážce.

Rychlejší vyhledávání

A) dosud: ***sekvenční*** průchod daty velikosti N $\rightarrow$ časová složitost O(N)

B) ***binární vyhledávání (půlení intervalů)***
- data musí být uspořádaná
- vždy porovnat hledanou hodnotu s prostředním prvkem zkoumaného úseku, polovinu úseku „zahodit“
- postupně dostáváme úseky délky N, $N/2$, $N/4$, $N/8$, …, 1
- po K krocích zbývá úsek velikosti $N/2^K$, hledáme K takové, aby $N/2^K = 1$

  $\rightarrow$ počet půlení $K = \log_2 N$, tedy časová složitost algoritmu O(log N)

*Příklad (historický)**:*** pražský telefonní seznam bytových stanic
- cca 430 000 jmen (v roce 1995)
- rychlost hledajícího člověka 1 jméno za sekundu
- sekvenční hledání: 5 dní a nocí x binární hledání: 20 sekund

6. binární vyhledávání
→ půlení intervalů v uspořádaném seznamu

```
i = 0                          # začátek úseku
j = len(a) - 1                 # konec úseku
k = (i + j) // 2               # střed úseku
while a[k] != x and i <= j:
    if x > a[k]:
        i = k + 1
    else:
        j = k - 1
    k = (i + j) // 2

if x == a[k]:
    print("Je na pozici", k)
else:
    print("Není tam")
```

V případě více výskytů najde některý z nich.

Výpočet druhé odmocniny

- aplikace algoritmu půlení intervalů:
  numerický iterační výpočet druhé odmocniny se zvolenou přesností

- místo půlení v uspořádaném seznamu hodnot
  budeme půlit interval na číselné ose, ve kterém musí ležet výsledek

- hledaný výsledek stále leží v uvažovaném intervalu <*dolni*, *horni*>
  a velikost tohoto intervalu se v každém kroku zmenší na polovinu,
  až bude menší než zvolená přesnost výpočtu *eps*

```python
def odmocnina(n):
    eps = 0.0001        # zvolená přesnost výpočtu
    if n > 1:
        dolni, horni = 1, n
    else:
        dolni, horni = 0, 1
    stred = (dolni + horni)/2
    while horni - dolni >= eps:
        if stred**2 < n:
            dolni = stred
        else:
            horni = stred
        stred = (dolni + horni)/2
    return stred

print(f"{odmocnina(900): 0.3f}")
```

Řazení dat v poli

= vnitřní třídění (terminologicky nepřesné, ale užívané)

Úloha: uspořádat prvky pole podle velikosti
(od nejmenšího po největší)

Přímé metody

SelectSort – třídění výběrem, přímý výběr
InsertSort – třídění vkládáním, přímé zatřiďování
BubbleSort – třídění záměnami, bublinkové třídění

- jednoduchý zápis programu
- třídí „na místě“ (tzn. nepotřebují další datovou strukturu velikosti N)
- časová složitost $O(N^2) \rightarrow$ vhodné pro malá data

Rychlejší metody

MergeSort – třídění sléváním
QuickSort – třídění rozdělováním
HeapSort – třídění haldou, haldové třídění

- časová složitost O(*N*.log *N*)

Přihrádkové metody pro data speciálních vlastností

CountingSort – třídění počítáním
BucketSort – přihrádkové třídění
RadixSort – víceprůchodové přihrádkové třídění

- „lineární“ časová složitost (ale nejen vzhledem k *N* – bude později)

Python: sám umí řadit
– standardní funkce sorted() – vytvoří setříděnou kopii

```
>>> a = [5, 2, 8, 1, 9, 0]
>>> sorted(a)
[0, 1, 2, 5, 8, 9]
>>>
```

- nebo metoda sort() – třídí na místě

```
>>> a.sort()
```

- lze řadit seznam čísel podle hodnot
  nebo seznam stringů abecedně
  (a také třeba n-tice, slovníky, množiny)
- lze řadit jakékoliv objekty podle vlastního kritéria – parametr key
- lze řadit vzestupně nebo sestupně – parametr revrese

Použitý algoritmus: *TimSort* (Tim Peters 2002)

- hybridní algoritmus MergeSort / InsertSort
- vyvinuto pro Python, používá také Java
- využívá existence uspořádaných úseků v datech

SelectSort (třídění výběrem, přímý výběr)

Algoritmus:

založíme prázdný výsledný seznam
dokud zadaný vstupní seznam není prázdný
- projdeme vstupní seznam a najdeme v něm nejmenší číslo
- odebereme ho ze seznamu
  a vložíme na konec výsledného uspořádaného seznamu

Implementace na místě v poli:

- *p*ole se dělí na setříděný úsek (vlevo) a nesetříděný úsek (vpravo)
- na začátku tvoří všechny prvky nesetříděný úsek
- v nesetříděném úseku pole se vždy najde nejmenší prvek
  a vymění se s prvním prvkem tohoto úseku,
  tím se setříděný úsek prodlouží o jeden prvek

7 4 2 9 5
2 4 7 9 5
2 4 7 9 5
2 4 5 9 7
2 4 5 7 9

modře – hotovo (setříděný úsek)
červeně – minimum ze zbývajících hodnot

```
def trid_vyberem(a):
    for i in range(len(a)-1):
        # umístit číslo na pozici "i"
        k = i
        for j in range(i+1, len(a)):
            if a[j] < a[k]:
                k = j
        if k > i:
            a[k], a[i] = a[i], a[k]
```

Časová složitost:

- celkem uděláme n-1 průchodů polem
- postupně procházíme úseky délky n, n-1, n-2, …, 2
- počet provedených porovnání čísel je tedy postupně

  n-1, n-2, n-3, …, 1
- celkový počet provedených porovnání čísel:

  $1 + 2 + \ldots + (n-2) + (n-1) = n.(n-1)/2$

→ asymptotická časová složitost $\Theta(n^2)$
(ve všech případech)

- počet provedených záměn čísel v poli je nejvýše n-1,

  což časovou složitost neovlivní

InsertSort (třídění vkládáním, přímé zatřiďování)

Algoritmus:

založíme prázdný výsledný seznam
dokud zadaný vstupní seznam není prázdný
- vezmeme první číslo ze vstupního seznamu
- odebereme ho ze seznamu a vložíme do výsledného uspořádaného seznamu na správné místo, kam patří

Implementace na místě v poli:

- pole se dělí na setříděný úsek (vlevo) a nesetříděný úsek (vpravo)
- na začátku je setříděný úsek tvořen pouze prvním prvkem pole
- první prvek nesetříděného úseku se vždy zařadí
  do setříděného úseku na místo, kam patří,
  tím se setříděný úsek prodlouží o jeden prvek

realizace: prvky setříděného úseku se posouvají o jednu pozici
doprava, dokud je třeba

7 4 2 9 5
4 7 2 9 5
2 4 7 9 5
2 4 7 9 5
2 4 5 7 9

modře – hotovo (setříděný úsek)
červeně – první ze zbývajících hodnot
(zatřiďovaný prvek)

```python
def trid_vkladanim(a):
    for i in range(1, len(a)):
        # vkládáme číslo z pozice "i"
        x = a[i]
        j = i-1
        while j >= 0 and x < a[j]:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = x
```

Časová složitost:

- celkem vykonáme n-1 vkládání čísla (průchodů polem)
- postupně procházíme úseky délky 1, 2, 3, …, n-1
- počet provedených porovnání a posunů čísel v poli je tedy postupně nejvýše 1, 2, 3, …, n-1 (může být menší, někdy se neprojde celý úsek)
- celkový počet provedených operací v nejhorším případě:

$$1 + 2 + \ldots + (n-2) + (n-1) = n.(n-1)/2$$

→ asymptotická časová složitost $\Theta(n^2)$

- časová složitost v nejlepším případě (seřazené vstupní pole) je pouze $\Theta(n)$ – stačí jeden průchod polem, žádné přesuny dat

BubbleSort (třídění záměnami, bublinkové třídění)

Základní myšlenka:
Pole je seřazeno vzestupně právě tehdy, když pro každou dvojici jeho sousedních prvků platí, že levý z nich je menší než pravý (nebo jsou stejné).

Algoritmus (zároveň implementace na místě v poli):
- opakovaně procházíme celým polem, porovnáváme sousední prvky a jsou-li špatně, vzájemně je vyměníme
- když při průchodu nenarazíme na žádnou špatnou dvojici sousedů, ukončíme výpočet

.

```python
def trid_bublinkove(a):
    setrideno = False
    while not setrideno:
        setrideno = True
        for j in range(len(a)-1):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
                setrideno = False
```

Jiná možnost implementce:

- každý průchod může být vždy o jeden krok kratší než předchozí (neboť největší prvek tříděného úseku se dostal až na konec úseku) → vždy stačí nejvýše *N*-1 průchodů

```
def trid_bublinkove(a):
    for i in range(len(a)-1):    # počítadlo průchodů
        for j in range(len(a)-i-1):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
```

Časová složitost:

- celkem vykonáme nejvýše n-1 průchodů polem
  (neboť při každém z nich se alespoň jedno číslo správně umístí)
- postupně procházíme úseky délky n, n-1, n-2, …, 2
- počet provedených porovnání a případných prohození čísel
  je tedy postupně n-1, n-2, n-3, …, 1
  (může být menší, někdy se čísla neprohazují,
  někdy stačí méně průchodů a pole je seřazeno)
- celkový počet provedených operací v nejhorším případě:

$$1 + 2 + \ldots + (n-2) + (n-1) = n.(n-1)/2$$

→ asymptotická časová složitost $\Theta(n^2)$

- časová složitost v nejlepším případě (seřazené vstupní pole)
  je pouze $\Theta(n)$ – stačí jeden průchod polem, žádné přesuny dat

Možnosti dalšího zrychlení:

- při příštím průchodu polem stačí jít jen do místa poslední uskutečněné výměny
  → rychlejší zkracování průchodů, stačí méně průchodů

```
def trid_bublinkove(a):
    vymena = len(a)-1
    while vymena > 0:
        pruchod = vymena   # kam až procházíme seznam
        vymena = 0
        for j in range(pruchod):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
                vymena = j    # místo poslední výměny
```

- třídění přetřásáním – pole se prochází střídavě zleva a zprava

MergeSort (třídění sléváním) – iterativní implementace

- nejprve v poli porovnáme dvojice sousedních prvků a uspořádáme je
$\rightarrow$ dostaneme pole uspořádaných dvojic

- sléváme první a druhou dvojici do uspořádané čtveřice,
třetí a čtvrtou dvojici do další uspořádané čtveřice, atd.
$\rightarrow$ dostaneme pole uspořádaných čtveřic

- takto pokračujeme dále, délku uspořádaných úseků zvyšujeme
v každém kroku na dvojnásobek: 2, 4, 8, 16, 32, …

- algoritmus končí, když délka uspořádaného úseku dosáhne *N*
(tzn. v poli je jediný setříděný úsek)

```
def mergesort(a):
    """
      třídění sléváním - iterativní verze
    """
    n = len(a)              # délka vstupního seznamu
    temp = [None] * n       # alokuje pomocný seznam

    # postupně slévá sousední úseky délek 1,2,4,...
    usek = 1
    while usek < n:
        for zacatek in range(0, n-usek, 2*usek):
            stred = zacatek + usek - 1
            konec = min(stred + usek, n-1)
            merge(a, zacatek, stred, konec, temp)
        usek *= 2
```

```
def merge(a, zac, stred, kon, temp):
    """
      sleje a[zac..stred] s a[stred+1..kon]
      do a[zac..kon] pomocí temp[zac..kon]
    """
    i = zac              # začátek prvního úseku
    j = stred+1          # začátek druhého úseku
    k = zac              # začátek výsledného seznamu

    # sleje a[zac..stred] s a[stred+1..kon] do temp
    while i <= stred and j <= kon:
        if a[i] < a[j]:
            temp[k] = a[i]
            i += 1
        else:
            temp[k] = a[j]
            j += 1
        k += 1
```

```
    if i <= stred:          # zbytek prvního úseku
        temp[k:kon+1] = a[i:stred+1]
    else:                   # zbytek druhého úseku
        temp[k:kon+1] = a[j:kon+1]

    # výsledek zkopíruje zpět do seznamu a
    a[zac:kon+1] = temp[zac:kon+1]

# konec definice funkce merge()
```

Prostorová složitost: O(*N*)
algoritmus potřebuje druhé pomocné pole na slévání úseků, nepracuje tedy „na místě“ jako předchozí algoritmy

Časová složitost:
velikost úseků se zdvojnásobuje $\rightarrow$ provede se $\log_2 N$ kroků výpočtu, v každém z nich se vykoná práce O(*N*), neboť součet délek všech slévaných úseků je *N* a slévání má lineární časovou složitost vzhledem k délce slévaných úseků
$\rightarrow$ celková časová složitost **O(*N*.log *N*)**