Vyhledávání v seznamu

Úloha: Zjistěte, zda se v seznamu *a* nachází daná hodnota *x*,
a pokud ano, tak kde (je-li tam vícekrát, chceme první výskyt).

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)`

Základní algoritmus: **sekvenční vyhledávání**
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:     # stačí jednoduchá podmínka
    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

dosud: sekvenční průchod daty velikosti n $\rightarrow$ časová složitost O(n)

jiný přístup: **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 (bylo 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 hodnoty *x* najde *některý* z nich.

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 menší 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)*

CountSort (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() třídy List – řadí data na místě

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

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

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 je vždy
  $1 + 2 + \ldots + (n\text{-}2) + (n\text{-}1) = n.(n\text{-}1)/2$

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

- počet provedených **záměn** čísel v poli je nejvýše n-1,
  což celkovou 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)

```
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:

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

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

- v nejlepším případě (seřazené pole) se vykoná pouze n-1 porovnání a žádný posun → asymptotická časová složitost $\Theta(n)$

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

.

```
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:

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

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

- časová složitost v nejlepším případě (seřazené vstupní pole)
  je lineární – stačí jeden průchod polem

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
  → dostaneme pole uspořádaných dvojic

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

- takto pokračujeme dále, délku uspořádaných úseků zvyšujeme
  v každém kroku na dvojnásobek: 2, 4, 8, 16, 32, …
  (poslední úsek může být kratší)

- 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: $\Theta(n)$
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 $\Theta(n)$, neboť součet délek všech slévaných úseků je n a slévání má lineární časovou složitost vzhledem k délce slévaných úseků
$\rightarrow$ celková časová složitost $\Theta(n.\log n)$

Vnější třídění = třídění souborů

Úloha: setřídit rozsáhlá data uložená v souboru, která se nevejdou do pole. Nelze tedy použít většinu algoritmů vnitřního třídění.

V textovém souboru nelze indexovat, máme jen *sekvenční přístup* k datům – to postačuje pro iterativní implementaci mergesortu, když setříděné úseky budou v souborech místo v poli.

Základní algoritmus: **slévání (merge)**
- ze dvou setříděných souborů vytvoří jeden setříděný soubor v čase lineárně závislém na počtu všech slévaných čísel

12 16 20 21 28 37 50
13 15 25 32 40 42 48
$\rightarrow$ 12 13 15 16 20 21 25 28 32 37 40 42 48 50

Princip vnějšího třídění: setříděné úseky uložené v souborech se postupně slévají do větších a větších úseků, až vnikne jediný setříděný soubor obsahující všechna data

Výchozí úseky:

- délky 1 (**přímé slučování**)
  v jednotlivých krocích výpočtu z nich postupně vznikají uspořádané dvojice, čtveřice, osmice, … atd.
  v *K*-tém kroku výpočtu mají uspořádané úseky délku 2^K

- přirozeně existující (**přirozené slučování**)

  4 7 20 15 18 10 6 14 20 27 22 24 17 3 8

  4 7 20 15 18 10 6 14 20 27 22 24 17 3 8

- **předtřídit** úseky v poli některým algoritmem vnitřního třídění
  → zrychlení výpočtu
  (odpadne několik prvních kroků vnějšího třídění)

Organizace práce:

- každý úsek uložen v jednom v souboru
  → existuje pak ale mnoho souborů (nevhodné)

- všechny úseky za sebou v jednom souboru
  fáze rozdělovací – úseky se rozdělí střídavě do dvou souborů
  fáze slučovací – úseky se po dvou slévají a ukládají se do jednoho výsledného souboru
  → **dvoufázové třídění** – v každém kroku výpočtu je každé číslo dvakrát čteno a dvakrát zapisováno do souboru

- zrychlení výpočtu: **jednofázové třídění**
  – při slučování jsou setříděné úseky rovnou ukládány střídavě do dvou souborů (odpadne tedy samostatná rozdělovací fáze, každé číslo je v každém kroku výpočtu čteno a zapisováno jen jednou)

Časová složitost:

- třídíme n záznamů

- podstatný je počet provedených *vstupních a výstupních operací* (tj. čtení ze souboru a zápis do souboru), porovnávací operace vykonávané ve vnitřní paměti jsou řádově rychlejší

- v každém kroku výpočtu je každý záznam jednou nebo dvakrát čten a zapisován $\rightarrow$ jeden krok má vždy časovou složitost $\Theta(n)$

- přímé slučování: po k-tém kroku výpočtu mají úseky délku 2^k, rovnost $2^k=n$ nastane pro $k=\log_2 n$, tedy potřebný počet kroků výpočtu je $\Theta(\log n)$ $\rightarrow$ časová složitost celého algoritmu $\Theta(n.\log n)$

- přirozené slučování je v průměru o něco rychlejší, ale v nejhorším případě stejné jako přímé slučování (vstupní data mohou být uspořádána sestupně)

Možnosti zrychlení:

- **předtřídění** počátečních úseků v poli
  * zároveň rozdělovací fáze 1. kroku jednofázového třídění
  * co největší délka úseků, kolik nám dovolí vnitřní paměť
  * lze vytvářet i delší úseky – heapsort s postupným doplňováním haldy, resp. se dvěma haldami umístěnými v poli proti sobě

  $\rightarrow$ časová složitost $n.\log_2 x$, kde x je výchozí počet úseků, tj. stále $\Theta(n.\log n)$

- **jednofázové** třídění namísto dvoufázového

  $\rightarrow$ poloviční počet vstupních a výstupních operací

- **vyšší stupeň slučování *s***
  * tzn. při jednofázovém třídění sléváme z s vstupních do s výstupních souborů

  $\rightarrow$ časová složitost $n.\log_s n$, tj. stále $\Theta(n.\log n)$

Dolní odhad složitosti problému třídění
pro porovnávací algoritmy

Vstupní data pro úlohu třídění:
- jistá posloupnost *n* čísel (klíčů), čísla mohou být navzájem různá
- existuje *n!* možných uspořádání vstupních dat velikosti *n*

Obecný problém třídění:
o vstupních datech nic nevíme, není předem omezen rozsah hodnot, mohou to být čísla typu float (tzn. nelze jimi indexovat)
$\rightarrow$ při třídění můžeme hodnoty jedině vzájemně porovnávat

Strom všech možných průběhů výpočtu
nějakého třídicího algoritmu pro vstupní data velikosti *n*:
- kořen = počáteční stav
- binární strom, větvení = porovnání nějakých dvou čísel
  (dva možné výsledky)
- listy stromu = konec výpočtu (data setříděna)

Pro každá vstupní data se musí průběh výpočtu někde odlišit od ostatních $\rightarrow$ strom má minimálně ***n!*** **listů**.

Výška stromu výpočtů *h*
= počet provedených porovnání čísel při nejdelším výpočtu
= časová složitost algoritmu v nejhorším případě

Výška úplného binárního stromu se všemi k listy na poslední hladině je $\log_2 k$. Jiný binární strom s k listy musí mít výšku větší.

Uvažovaný strom všech možných výpočtů má tedy výšku $h \geq \log_2(n!)$. Časová složitost libovolného třídicího algoritmu (v nejhorším případě) nemůže být proto lepší než $\log_2(n!)$. Ukážeme, že to je $\Omega(n.\log n)$.

Známe konkrétní algoritmy s časovou složitostí $O(n.\log n)$, např. heapsort nebo mergesort, proto $\Theta(n.\log n)$ je i **složitost obecného problému vnitřního třídění v nejhorším případě**.

Přechod $\log_2(n!) \rightarrow n.\log n$
- podle Stirlingova vzorce
- bez použití Stirlingova vzorce:

$$n! \geq (2k).(2k-1) \ldots (k+1).k > k^{k+1} \geq k^k = (n/2)^{n/2}$$

pro n = 2k

$$n! \geq (2k+1).(2k) \ldots (k+1) > (k+1/2)^{k+1} > (k+1/2)^{k+1/2} = (n/2)^{n/2}$$

pro n = 2k+1

$$h \geq \log_2(n!) > \log_2((n/2)^{n/2}) = n/2 \,.\, \log_2(n/2) = n/2 \,.\, (\log_2 n - 1) \geq$$
$$\geq n/2 \,.\, (\log_2 n \;/\; 2) = n/4 \,.\, \log_2 n$$

↑

neboť $\log_2 n \;/\; 2 \geq 1$ pro $n \geq 4$