Řazení

IB111 Základy programování

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

7. listopadu 2025

Řazení

Řazení seznamů (nebo dat obecně)

- Oblíbené algoritmické téma.
- Anglicky „sorting“, česky někdy „třídění“ (nevhodné).
- Cílem je seřadit data v seznamu.
  - Obecněji i jiných datových strukturách s různými možnostmi přístupu k prvkům, možnostmi modifikace apod.
- 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.“
  - Složitost?
  - Stále zkoumané téma[1].

[1] https://techxplore.com/news/2022-12-scientists-python-function.html

Řadicí algoritmy

Vizualizace apod.

- https://visualgo.net/en/sorting
- http://www.sorting-algorithms.com
- https://en.wikipedia.org/wiki/Sorting_algorithm

Problém řazení seznamů

- Vstup: seznam čísel (nebo jiných porovnávatelných prvků[2]).
- Výstup:

[2] Existují algoritmy vyžadující bližší přístup k řazeným prvkům, např. jednotlivým číslicím – nad rámec IB111.

Problém řazení seznamů

- Vstup: seznam čísel (nebo jiných porovnávatelných prvků[2]).
- Výstup: seřazený seznam, který
  - obsahuje stejné prvky jako vstupní seznam,

[2] Existují algoritmy vyžadující bližší přístup k řazeným prvkům, např. jednotlivým číslicím – nad rámec IB111.

Problém řazení seznamů

- Vstup: seznam čísel (nebo jiných porovnávatelných prvků[2]).
- Výstup: seřazený seznam, který
  - obsahuje stejné prvky jako vstupní seznam,
  - ve stejném počtu.

- Příklad:
  - Vstup: [7, 0, -3, 1, 100, 17, 42, 0].
  - Výstup: [-3, 0, 0, 1, 7, 17, 42, 100].

[2] Existují algoritmy vyžadující bližší přístup k řazeným prvkům, např. jednotlivým číslicím – nad rámec IB111.

Řazení: První pokus

Řešení

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

- Je toto dobrý algoritmus?

Řazení: První pokus

Řešení

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

- Je toto dobrý algoritmus?
- Je to korektní algoritmus?
  - Dělá to, co po něm chceme?
- Je to efektivní algoritmus?
  - Kolik času zabere?
  - Kolik času zabere v nejhorším případě?

Řazení: První pokus

Řešení

- Zkoušíme systematicky všechna možná uspořádání prvků 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?
  - Kolik času zabere?
  - Kolik času zabere v nejhorším případě?

Řazení: První pokus

Řešení

- Zkoušíme systematicky všechna možná uspořádání prvků 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?
  - Kolik času zabere? Záleží na vstupu.
  - Kolik času zabere v nejhorším případě?

Řazení: První pokus

Řešení

- Zkoušíme systematicky všechna možná uspořádání prvků 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 popřemýšlet nad lepším řešením.
- Zvažte různé principy.
- Co nejefektivnější: čas, dodatečná paměť.
- Inspirace:
  - Jak byste seřadili karty?
  - Jak byste seřadili 200 kartiček se jmény lidí podle abecedy?
  - Jak by se seřadili lidé stojící v úzké chodbě podle abecedy?

Select Sort (řazení výběrem)

- Inspirace řazením zástupu lidí.

Základní princip

- Vybereme nejmenší prvek seznamu a zařadíme na 1. místo.
- Vybereme nejmenší prvek zbytku a zařadíme na 2. místo.
- Atd.

Select Sort (řazení výběrem)

- Inspirace řazením zástupu lidí.

Základní princip

- Vybereme nejmenší prvek seznamu a zařadíme na 1. místo.
- Vybereme nejmenší prvek zbytku a zařadíme na 2. místo.
- Atd.

- Jak implementovat?

Řazení výběrem (Select Sort)

select_sort.py

```
def select_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(size - 1):
        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[selected]
```

- Jaká je složitost tohoto algoritmu?
  - V nejlepším případě?
  - V nejhorším případě?

Řazení výběrem (Select Sort)

select_sort.py

```
def select_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(size - 1):
        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[selected]
```

- Jaká je složitost tohoto algoritmu?
  - V nejlepším případě? *Kvadratická.*
  - V nejhorším případě?

Řazení výběrem (Select Sort)

select_sort.py

```
def select_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(size - 1):
        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[selected]
```

- Jaká je složitost tohoto algoritmu?
  - V nejlepším případě? *Kvadratická.*
  - V nejhorším případě? *Kvadratická.*

Řazení vkládáním (Insert Sort)

- Inspirace řazením karet.

Základní princip

- Udržujeme si seřazenou část seznamu.
- Vezmeme jeden prvek z neseřazené části a zařadíme ho na správné místo do seřazené části.

Řazení vkládáním (Insert Sort)

- Inspirace řazením karet.

Základní princip

- Udržujeme si seřazenou část seznamu.
- Vezmeme jeden prvek z neseřazené části a zařadíme ho na správné místo do seřazené části.

- Jak implementovat?

Insert Sort (řazení vkládáním)

insert_sort.py

```
def insert_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(1, size):
        # my_list[0], …, my_list[i - 1] is sorted
        # the rest is not sorted yet
        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?
  - V nejhorším případě?

Insert Sort (řazení vkládáním)

insert_sort.py

```
def insert_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(1, size):
        # my_list[0], …, my_list[i - 1] is sorted
        # the rest is not sorted yet
        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? *Lineární.*
  - V nejhorším případě?

Insert Sort (řazení vkládáním)

insert_sort.py

```
def insert_sort(my_list: list[int]) -> None:
    size = len(my_list)
    for i in range(1, size):
        # my_list[0], …, my_list[i - 1] is sorted
        # the rest is not sorted yet
        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? *Lineární.*
  - V nejhorším případě? *Kvadratická.*

Řazení – další algoritmy (pro zajímavost)

Bubble Sort

- Opakovaně procházíme seznam.
- Najdeme-li špatně seřazenou dvojici prvků, prohodíme je.
- (Zdánlivě podobná složitost jako Insert Sort, ale prakticky velmi špatný algoritmus.)

Řazení – další algoritmy (pro zajímavost)

Bubble Sort

- Opakovaně procházíme seznam.
- Najdeme-li špatně seřazenou dvojici prvků, prohodíme je.
- (Zdánlivě podobná složitost jako Insert Sort, ale prakticky velmi špatný algoritmus.)

Quick Sort

- Vybereme tzv. pivota (jeden prvek).
- Rozdělíme seznam na: menší než pivot, pivot, větší než pivot.
- *Rekurzivně* seřadíme vzniklé menší seznamy.
- Složitost v nejhorším případě *kvadratická*.
- Průměrně zhruba $n \cdot \log n$, kde n je délka seznamu.
  - 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 zhruba $n \cdot \log n$.
  - To je teoreticky to nejlepší, co jde (pro algoritmy založené na porovnávání).
- Potřebuje extra prostor.

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 zhruba $n \cdot \log n$.
  - 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 často používá kombinace různých přístupů.
  - Python (do 3.10): *Timsort* (kombinace Merge a Insert Sortu).
  - C++: *Introsort* (kombinace Insert, Quick a Heap Sortu).

Řazení v Pythonu

- `my_list.sort()` – seřadí seznam,
  - tj. modifikuje zadaný seznam; nic nevrací.
- `sorted(my_list)` – vytvoří nový seřazený seznam ze zadaných prvků,
  - nemodifikuje původní seznam,
  - funguje i pro jiné datové struktury (množiny, klíče slovníků, …).
- Složitost řazení je v nejhorším případě zhruba $n \cdot \log n$.

Řazení ntic nebo seznamů

`sort_lists.py`

- Funguje podle operátoru < jako u jiných objektů.
- Použije se lexikografické pořadí.

Řazení ntic nebo seznamů

`sort_lists.py`

- Funguje podle operátoru `<` jako u jiných objektů.
- Použije se lexikografické pořadí.

Lexikografické uspořádání – připomenutí

- Na *dvojicích*:
  `(a, b) < (c, d)` je totéž, co
  `a < c or (a == c and b < d)`.
- Na *nticích stejné délky*:
  podobně; výsledek je podle první dvojice, která se nerovná.
- Na *nticích různé délky*/*seznamech*:
  podobně; je-li `s` vlastním prefixem (začátkem) `t`, pak `s < t`.

Různé způsoby řazení

`sorting.py`

Obrácené řazení – nejdřív `.sort()`, pak `.reverse()`.

Různé způsoby řazení

`sorting.py`

Obrácené řazení – nejdřív `.sort()`, pak `.reverse()`.

Řazení podle klíče (nebo i více klíčů)

- Klíč – vlastnost, podle které chceme řadit.
- Například: řazení seznamů podle jejich délky.

[3] Této technice se říká „decorate-sort-undecorate“ či „Schwartzian transform“.

Různé způsoby řazení

sorting.py

Obrácené řazení – nejdřív `.sort()`, pak `.reverse()`.

Řazení podle klíče (nebo i více klíčů)

- Klíč – vlastnost, podle které chceme řadit.
- Například: řazení seznamů podle jejich délky.
- Využijeme toho, že ntice jsou porovnávatelné[3].

```
def sorted_by_len(lists: list[list[int]]) \
        -> list[list[int]]:
    len_lists = [(len(s), s) for s in lists]
    len_lists.sort()
    return [s for _, s in len_lists]
```

[3] Této technice se říká „decorate-sort-undecorate“ či „Schwartzian transform“.

Různé způsoby řazení – podle klíče

`sorting.py`

- **Stabilní řazení** – zachovává pořadí sobě rovných prvků.
  - `sort()` v Pythonu je stabilní.
  - Stabilita nemusí být zachována při použití dekorace.

- Příklad: stabilní řazení seznamů podle jejich délky.

Různé způsoby řazení – podle klíče

`sorting.py`

- **Stabilní řazení** – zachovává pořadí sobě rovných prvků.
  - `sort()` v Pythonu je stabilní.
  - Stabilita nemusí být zachována při použití dekorace.

- Příklad: stabilní řazení seznamů podle jejich délky.
- Využijeme *indexů* do původního seznamu.

```
len_i = [(len(s), i)
         for i, s in enumerate(lists)]
```

- Jak **obrátit pořadí** číselného klíče?

Různé způsoby řazení – podle klíče

`sorting.py`

- **Stabilní řazení** – zachovává pořadí sobě rovných prvků.
  - `sort()` v Pythonu je stabilní.
  - Stabilita nemusí být zachována při použití dekorace.

- Příklad: stabilní řazení seznamů podle jejich délky.
- Využijeme *indexů* do původního seznamu.

```
len_i = [(len(s), i)
         for i, s in enumerate(lists)]
```

- Jak **obrátit pořadí** číselného klíče? Použít `-`.

Stabilní řazení vlastních objektů

Stabilní řazení vlastních objektů

- Opět využijeme indexů do původního seznamu.

```
def sorted_by_age(people: list[Person]) \
        -> list[Person]:
    ages_i = [(p.age, i)
              for i, p in enumerate(people)]
    ages_i.sort()
    return [people[i] for _, i in ages_i]
```

unique.py

Unikátní prvky (s použitím řazení)

unique.py

Unikátní prvky (s použitím řazení)

```
def unique_elements(my_list: list[int]) -> list[int]:
    result = []
    last: int | None = None
    # we don't want to change the input list
    for elem in sorted(my_list):
        if elem != last:
            result.append(elem)
            last = elem
    return result
```

- Jakou má toto řešení složitost?

Unikátní prvky (s použitím řazení)

```
def unique_elements(my_list: list[int]) -> list[int]:
    result = []
    last: int | None = None
    # we don't want to change the input list
    for elem in sorted(my_list):
        if elem != last:
            result.append(elem)
            last = elem
    return result
```

- Jakou má toto řešení složitost? Zhruba $n \cdot \log n$,
  - tj. takovou jako řazení.

`h_index.py`

Počítání *h-indexu*

- *h-index* – metrika publikační činnosti.
- Největší číslo h s vlastností „osoba má alespoň h publikací, které byly alespoň h-krát citovány“.

h_index.py

Počítání *h-indexu*

- *h-index* – metrika publikační činnosti.
- Největší číslo *h* s vlastností „osoba má alespoň *h* publikací, které byly alespoň *h*-krát citovány“.

```
def h_index(citations: list[int]) -> int:
    citations = sorted(citations)
    citations.reverse()

    h = 0
    while h < len(citations) and citations[h] > h:
        h += 1

    return h
```

K zamyšlení

Počítání *h-indexu*

- Zkuste si rozmyslet, proč je předchozí řešení korektní.
- Zkuste místo lineárního průchodu použít binární vyhledávání.
- Pro pokročilé: zkuste vymyslet řešení, které *nepoužije řazení* a bude mít v nejhorším případě *lineární* složitost.

K zamyšlení

Počítání *h-indexu*

- Zkuste si rozmyslet, proč je předchozí řešení korektní.
- Zkuste místo lineárního průchodu použít binární vyhledávání.
- Pro pokročilé: zkuste vymyslet řešení, které *nepoužije řazení* a bude mít v nejhorším případě *lineární* složitost.

Jaký je rozdíl mezi těmito dvěma příkazy?

- `my_list.sort()`
- `my_list = sorted(my_list)`