Kapitola 2

Základní řadící algoritmy, správnost programu

Co si zde procvičíme?

- Nahlédneme do základů teorie složitosti
- Vrátíme se k základním řadícím algoritmům insertion sort a selection sort, které jistě již znáte, a podíváme se na ně podrobněji
- Budeme se zabývat důkazem správnosti algoritmu:
  - Seznámíme se s pojmem invariant a jeho základními vlastnostmi
  - Ukážeme si, jak invariant správně formulovat
  - Naučíme se ověřit platnost invariantu v průběhu programu
- Zaměříme se na základní analýzu složitosti programu a určení, kolikrát se při běhu programu zopakuje ten který řádek
- Ukážeme si, jak prosté zanoření příkazu může výrazně ovlivnit složitost algoritmu i přesto, že jeho funkčnost zůstane zachována
- Seznámíme se rovněž s algoritmy pro lineární a binární vyhledávání

2.1 Teorie složitosti

Teorie složitosti je obor, který se zabývá **výpočetními problémy** a klasifikací jejich složitosti včetně hledání vzájemných vztahů. Specifikace výpočetního problému má dvě části:

- definice vstupu (vstupních dat),
- definice výstupu (výstupních dat).

Instance výpočetního problému je problém specifikovaný pro daný konkrétní vstup. Instanci problému si nesmíme zaměňovat se samotným problémem a procesem jeho řešení.

V této kapitole se budeme zabývat problémy řazení čísel. V těchto problémech je na vstupu posloupnost $a_1, a_2, \ldots, a_n$ čísel a na výstupu je permutace těchto čísel $a_1', a_2', \ldots, a_n'$, kde požadujeme

$$a_1' \leq a_2' \leq \cdots \leq a_n'.$$

Algoritmus je - řečeno neformálně - dostatečně přesný popis postupu výpočtu.

Algoritmus řešící nějaký výpočetní problém je správný, pokud pro každý vstup skončí správným výsledkem, výstupem. Správnost algoritmu určujeme pomocí metody **invariantu**.

Invariant programu je vlastnost, která se během vykonávání programu nemění. Její platnost ověřujeme ve třech fázích:

- na začátku programu,
- při vykonávání programu (po každém jednotlivém kroku),
- na konci programu

Invariant by se měl vztahovat k danému programu (algoritmu). Používá se k důkazu správnosti algoritmu, protože na jeho konci dává smysluplnou informaci o výsledku výpočtu.

2.2 Insertion sort a invariant programu

Algoritmus řazení vkládáním (**insertion sort**) pracuje tak, že postupně prochází prvky a každý další zařadí na správné místo do již seřazené posloupnosti.

Úloha 2.2.1. Insertion sort

a) Je dán program, který řadí čísla použitím algoritmu insertion sort:

```
1       def insertion_sort(a):
2           for j in range(1, len(a)):
3               key = a[j]
4               i = j - 1
5               while i >= 0 and a[i] > key:
6                   a[i + 1] = a[i]
7                   i = i - 1
8               a[i + 1] = key
```

Ověřte platnost následujícího invariantu:
Na začátku každé vnější iterace pro j obsahuje podposloupnost $\{a_0, ..., a_{j-1}\}$ prvky $\{a_0, ..., a_{j-1}\}$ původní posloupnosti $\{a_0, ..., a_{n-1}\}$ v uspořádaném pořadí.
Je invariant platný na začátku algoritmu (pro $j = 0$)? Platí během celého programu, tj. po každém řádku? Dává na konci programu (pro $j = n - 1$) informaci o seřazení posloupnosti?
Odůvodněte, že uvedený invariant neplatí na řádcích 6-8 algoritmu.

b) Upravíme invariant následovně:
Na začátku každé vnější iterace pro j obsahuje podposloupnost $\{a_0, ..., a_j\}$ prvky $\{a_0, ..., a_j\}$ původní posloupnosti $\{a_0, ..., a_{n-1}\}$ v uspořádaném pořadí.
Ověřte platnost invariantu pro daný program. V případě, že je invariant porušen, označte řádky programu, na nichž k porušení invariantu dochází.

Návod: Vypište si obsah pole a po každém řádku programu. Ve výpisu si označte index j a kontrolujte platnost invariantu.

Řešení: Podle algoritmu insertion sort řadíme pole $[2, 7, 5, 6, 4, 1]$. Aktuálně je $j = 2$, tj. chceme zařadit prvek 5 do již seřazené posloupnosti $\{2, 7\}$. Získáme následující výpis obsahu pole po jednotlivých řádcích (podle specifikace invariantu řádky vnějšího cyklu opomíjíme):

Řádek	**Obsah pole**					
			j			
2	2	7	5	6	4	1
3	2	7	5	6	4	1
4	2	7	5	6	4	1
5	2	7	5	6	4	1
8	2	5	7	6	4	1

Když se vrátíme k invariantu, jehož platnost máme ověřit, zaměříme se na sloupeček j. Pozorujeme, že na řádku 2 až 5 je porušeno pravidlo uspořádání posloupnosti $\{2, 7, 5\}$. Daný invariant platí na řádku 8.

Úloha 2.2.2. Program pro řazení insertion sort upravíme následovně:

```
1    def insertion_sort1(a):
2        for j in range(1, len(a)):
3            key = a[j]
4            i = j - 1
5            while i >= 0 and a[i] > key:
6                a[i + 1] = a[i]
7                i = i - 1
8                a[i + 1] = key
```

Lze nový program použít pro seřazení pole a?
Ověřte platnost původního invariantu:
Na začátku každé vnější iterace pro j obsahuje podposloupnost $\{a_0, ..., a_{j-1}\}$ prvky $\{a_0, ..., a_{j-1}\}$ původní posloupnosti $\{a_0, ..., a_{n-1}\}$ v uspořádaném pořadí.

Řešení: Algoritmus lze využít k řazení prvků pole - argumentujte, že tomu tak je. Invariant není pro uvedenou verzi algoritmu platný. Dokažte to!

Úloha 2.2.3. Vraťme se k původní verzi programu pro řazení insertion sort (viz úloha 2.2.1). Předpokládáme, že pole a obsahuje n prvků. Spočtěte pro jednotlivé řádky algoritmu, kolikrát se při běhu programu vykonají.

Návod: To, kolikrát se vykoná ten který řádek, lze zjistit i programově - pomocí počítadla pro jednotlivý řádek. Pro analýzu je nutné řešit úlohu pro nejhorší případ, tj. v tomto případě posloupnost řazená sestupně.
Řešení: Tabulka uvádí horní odhad počtu opakování jednotlivých řádků programu:

Pokud všechny výrazy v pravém sloupci tabulky sečteme, lze pro $n \geq 10$ výsledek shora ohraničit hodnotou $4n^2$.Pokud bychom stejný výpočet

`def insertion_sort(a):`	–
`    for j in range(1, len(a)):`	n
`        key = a[j]`	$n-1$
`        i = j - 1`	$n-1$
`        while i >= 0 and a[i] > key:`	$\sum_{i=1}^{n-1}(i+1) = \frac{(n+2)(n-1)}{2}$
`            a[i + 1] = a[i]`	$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
`            i = i - 1`	$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
`        a[i + 1] = key`	$n-1$

provedli pro seřazenou posloupnost (nejlepší případ), lze výsledek shora ohraničit hodnotou $5n$. Zamyslete se nad tím, co toto zjištění znamenají pro celkovou složitost algoritmu insertion sort.

Úloha 2.2.4. Je dána rekurzivní verze algoritmu insertion sort:

```
def insertion_sort_rec(a, n):
    if n > 0:
        insertion_sort_rec(a, n - 1)
        key = a[n]
        i = n - 1
        while i >= 0 and a[i] > key:
            a[i + 1] = a[i]
            i = i - 1
        a[i + 1] = key
```

Z konstrukce programu pro `insertion_sort_rec` lze časovou složitost vyjádřit následujícím vztahem:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n-1) + c_2 n, & n > 0 \end{cases}$$

Vysvětlete, jak uvedený vztah vznikl.
Dokažte matematickou indukcí, že $T(n) \leq c_3 n^2$. c_1, c_2, c_3 jsou konstanty nezávislé na n.

2.3 Selection sort

Algoritmus řazení výběrem (**selection sort**) pracuje tak, že vyhledá nejmenší prvek v neseřazené části posloupnosti a vymění ho s prvním prvek neseřazené části. Tím se seřazená část posloupnosti zvětší o jeden prvek.

Úloha 2.3.1. Selection sort

a) Je dán program, který řadí čísla použitím algoritmu selection sort:

```
1  def selection_sort(a):
2      for j in range(0, len(a) - 1):
3          smallest = j
4          for i in range(j + 1, len(a)):
```

```
5                   if a[i] < a[smallest]:
6                       smallest = i
7               a[j], a[smallest] = a[smallest], a[j]
```

Ověřte platnost následujícího invariantu:
Na začátku každé vnější iterace pro j obsahuje podposloupnost $\{a_0, ..., a_{j-1}\}$ j nejmenších prvků původní posloupnosti $\{a_0, ..., a_{n-1}\}$ a tyto prvky $\{a_0, ..., a_{j-1}\}$ jsou uspořádané.

b) Upravíme invariant následovně:
Na začátku každé vnější iterace pro j obsahuje podposloupnost $\{a_0, ..., a_j\}$ $j+1$ nejmenších prvků původní posloupnosti $\{a_0, ..., a_{n-1}\}$ a tyto prvky $\{a_0, ..., a_j\}$ jsou uspořádané.
Ověřte platnost invariantu pro daný program. V případě, že je invariant porušen, označte řádky programu, na nichž k porušení invariantu dochází.

> *Řešení:* Invariant platí pouze po výměně prvků na posledním (sedmém) řádku programu. Ukažte to na příkladu.

Úloha 2.3.2. Složitost algoritmu selection sort
Předpokládáme, že pole a obsahuje n prvků.

a) Spočtěte pro jednotlivé řádky algoritmu selection sort, kolikrát se při běhu programu vykonají. Vypočtěte celkovou složitost algoritmu v závislosti od hodnoty n.

b) Vypočtěte celkovou složitost algoritmů `selection_sort1` a `selection_sort2` (viz níže, algoritmy se liší úrovní vnoření posledního řádku) v závislosti od hodnoty n. Opět nejprve určete, kolikrát se během výpočtu budou opakovat jednotlivé řádky a získané hodnoty sečtěte.
Výsledek ověřte programově - definicí počítadla pro jednotlivé řádky.

```
1       def selection_sort1(a):
2           for j in range(0, len(a) - 1):
3               smallest = j
4               for i in range(j + 1, len(a)):
5                   if a[i] < a[smallest]:
6                       smallest = i
7                   a[j], a[smallest] = a[smallest], a[j]
```

```
1       def selection_sort2(a):
2           for j in range(0, len(a) - 1):
3               smallest = j
4               for i in range(j + 1, len(a)):
5                   if a[i] < a[smallest]:
6                       smallest = i
7                       a[j], a[smallest] = a[smallest], a[j]
```

c) Který z algoritmů `selection_sort`, `selection_sort1` a `selection_sort2` je podle výpočtu nejrychlejší? Výsledek ověřte opakovaným spouštěním programů pro vygenerované pole a velikosti $n = 1000, 2000, ..., 10000$. Pomocí knihovny `time` odměřte dobu výpočtu programu. Odpovídá výsledek měření výpočtům?

2.4 Lineární vyhledávání

Lineární vyhledávání (nazývané také **sekvenční vyhledávání**) funguje na principu postupného procházení všech prvků seznamu.
Znalost vlastností posloupnosti prvků (například informace o řazení posloupnosti) může ovlivnit rychlost vyhledání prvku v posloupnosti.

Úloha 2.4.1. Lineární vyhledávání

a) Je dán program, který nalezne hodnotu v v poli a, pokud tato hodnota v poli existuje a vrátí index prvního výskytu hodnoty v v poli.

```
1    def linear_search(a, v):
2        for i in range(len(a)):
3            if a[i] == v:
4                return i
5        return -1
```

Ověřte platnost následujícího invariantu:
Podposloupnost $\{a_0, ..., a_{i-1}\}$ *neobsahuje prvek* v.
Na základě platnosti invariantu diskutujte správnost programu.

b) Upravíme invariant následovně:
Podposloupnost $\{a_0, ..., a_i\}$ *neobsahuje prvek* v.
Ověřte platnost invariantu pro daný program. V případě, že je invariant porušen, označte řádky programu, na nichž k porušení invariantu dochází.

Úloha 2.4.2. Složitost algoritmu lineárního vyhledávání

a) Spočtěte pro jednotlivé řádky algoritmu lineárního vyhledávání, kolikrát se při běhu programu vykonají. Úlohu řešte v závislostí na délce n posloupnosti.
Rozlište případy, kdy je hledaný prvek prvním nebo posledním prvkem posloupnosti, kdy se nachází uprostřed posloupnosti, a také, kdy se v posloupnosti nenachází.

b) Vypočtěte celkovou složitost algoritmu `linear_search1`. Stanovte invariant a ověřte, že algoritmus slouží k vyhledání prvků v poli. Určete, kolikrát se budou opakovat jednotlivé řádky algoritmu.

```
1    def linear_search1(a, v):
2        found = -1
3        for i in range(len(a)):
4            if a[i] == v:
5                found = i
6        return found
```

Nápověda: Formulujte invariant ve vztahu k hodnotě proměnné `found`.

c) Předpokládejme, že vyhledáváme v seřazeném poli a. Použijeme algoritmus **binárního vyhledávání** / viz program dále. Pro jednotlivé řádky programu spočtěte, kolikrát se při běhu programu vykonají.

```
1   def binary_search(a, v):
2       begin, end = 0, len(a)
3       while begin <= end:
4           mid = (begin + end) // 2
5           if a[mid] == v:
6               return mid
7           if a[mid] < v:
8               begin = mid + 1
9           else:
10              end = mid - 1
11      return -1
```

d) Který z algoritmů `linear_search`, `linear_search1` a `binary_search` je podle předpokladu (výpočtu) nejrychlejší? Výsledek ověřte opakovaným spouštěním programů pro vygenerovaná pole a velikosti $n = 1000, 2000, ..., 10000$ a pro hodnotu v, která se v daném poli nachází, resp. nenachází.

Pomocí knihovny `time` odměřte dobu výpočtu programu. Odpovídá výsledek měření výpočtům?