C2142 Návrh algoritmů pro přírodovědce

4. Řazení

Tomáš Raček

Jaro 2022

Sekvenční vyhledávání

Problém. Uvažme problém nalezení prvku v poli. Tento lze zřejmě řešit s lineární složitostí.

```
def search(array, k):
    for i in range(len(array)):
        if array[i] == k:
            return i

    return None
```

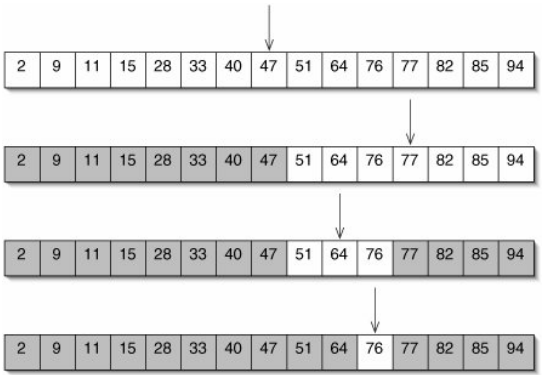
Zamyšlení. Pokud by bylo pole seřazeno, vyhledávání v něm by mohlo být snazší.

- Jak náročné je vyhledávání v seřazeném poli?
- Jak náročné je seřadit pole?

Binární vyhledávání I

Myšlenka. Pole je seřazeno ⇔ pro každý prvek pole platí, že hodnoty vlevo od něj jsou menší nebo rovny tomuto prvku a vpravo od něj větší nebo rovny.

Příklad. Vyhledávání čísla 76.



Binární vyhledávání II

Rekurzivní implementace binárního vyhledávání:

```
def binary_search(A, k, i_min, i_max):
        if i_max < i_min:
                return None

        mid = (i_min + i_max) // 2
        if k == A[mid]:
                return mid
        elif k < A[mid]:
                return binary_search(A, k, i_min, mid - 1)
        else:
                return binary_search(A, k, mid + 1, i_max)
```

Složitost. V každé iteraci se velikost prohledávané oblasti zmenší na polovinu. Složitost binárního vyhledávání je tedy $O(\log n)$.

Problém řazení

Neformální definice:

Vstup: Posloupnost prvků $a_1, \ldots, a_n$ délky n

Výstup: Neklesající permutace vstupní posloupnosti, tj.

$$\forall i \in \{1, \ldots, n-1\} : a_i \leq a_{i+1}$$

Poznámka k definici. Formální definice pracuje s prvky tvaru (a_i, D_i), kde a_i jsou klíče (podle kterých se řadí) a D_i pak další data, která mají význam v praktických aplikacích. Při analýze algoritmů je většinou zanedbáme ($D_i = \emptyset$).

Poznámka k terminologii. Někdy se nesprávně používá pro řazení výraz třídění, které značí ale spíše seskupování objektů podle daných vlastností.

Selection sort

Myšlenka. Najdi v poli nejmenší prvek a vyměň jej s prvkem na první pozici. Opakuj postup pro zbytek pole (bez prvního prvku).

```
def selection_sort(A):
    for i in range(len(A)):
        min_idx = i
        for j in range(i, len(A)):
            if A[min_idx] > A[j]:
                min_idx = j
        A[min_idx], A[i] = A[i], A[min_idx]
```

Složitost. Vnější `for` cyklus zjevně $O(n)$, vnitřní má však různé délky (n, n - 1,..., 1). Celkem tedy 1 + 2 + ...+ n $\in O(n^2)$.

Bubble sort

Myšlenka. Porovnávám postupně prvky na sousedních pozicích. Pokud jsou vůči sobě dva v nesprávném pořadí, prohodím je.

Důsledek. Po první iteraci „probublá“ největší prvek na konec pole.

```
def bubble_sort(A):
    for i in range(len(A)):
        for j in range(len(A) - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
```

Složitost. $O(n^2)$ podle stejné úvahy jako pro selection sort. Složitost lze vylepšit na příznivých datech. Jak?

Insertion sort

Myšlenka. Rozdělme (virtuálně) pole na dvě části: (1) už seřazenou a (2) dosud neseřazenou. Vždy první prvek z (2) zařazujeme korektně do (1).

```
def insertion_sort(A):
    for i in range(len(A)):
        item = A[i]
        j = i
        while j > 0 and item < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = item
```

Složitost. Pro nejhorší případ (pole seřazeno v opačném pořadí) je složitost $O(n^2)$.

Složitost problému řazení

Idea důkazu. Uvažme posloupnosti složené pouze z čísel 1,...,n, kde se každé číslo vyskytuje nejvýše jednou (= permutace této množiny). Takových je zjevně n!.

Asociativní řadicí algoritmy mohou provádět pouze porovnání dvojice prvků.

Korektní řadicí algoritmus musí pro každou takovouto posloupnost provést jiný výpočet.

Libovolný asociativní řadicí algoritmus musí tyto výpočty odlišit, tj. provést alespoň $\log_2(n!)$ binárních testů.

Důsledek. Dolní odhad složitosti řazení je $\Omega(n \log n)$.

Quick sort

Myšlenka. Vyberu prvek pole. Vlevo od něj přesunu všechny menší prvky, vpravo od něj větší. Obě tyto části pak řadím dále rekurzivně stejným postupem.

```
def quick_sort(array):
    if len(array) <= 1:
        return array
    else:
        pivot = array[0]
        smaller = [x for x in array[1:] if x < pivot]
        bigger = [x for x in array[1:] if x >= pivot]
        return quick_sort(smaller) + [pivot] + quick_sort(bigger)
```

Složitost. V nejhorším případě (libovolně seřazené pole) bude hloubka rekurzivního volání $O(n)$, v každé úrovni je potřeba rozdělit prvky do dvou částí se složitostí $O(n)$. Celkem tedy $O(n^2)$.

Merge sort

Idea. Uvažme dvě již seřazené posloupnosti. Jak z nich vytvořit jednu seřazenou?

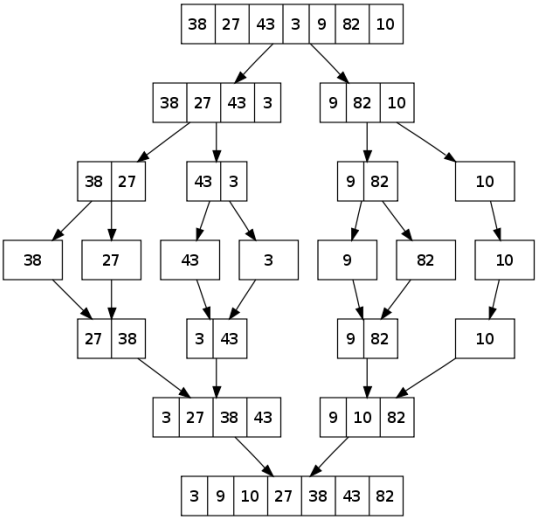
Funkce merge(A, B)

- porovnává vždy první prvky obou posloupností, menší z nich přesune do výstupní posloupnosti

Merge sort

- seřazenou posloupnost délky n získám ze dvou seřazených posloupností délky $n/2$ aplikací funkce merge
- analogicky posloupnosti velikosti $n/2$ „slévám“ ze dvou seřazených posloupností o velikosti $n/4$
- posloupnosti délky 1 jsou implicitně seřazeny

Merge sort – ukázka



Merge sort – implementace

```
def merge(A, B):
    if len(A) == 0:
        return B
    if len(B) == 0:
        return A

    if A[0] < B[0]:
        return [A[0]] + merge(A[1:], B)
    else:
        return [B[0]] + merge(A, B[1:])

def merge_sort(A):
    if len(A) <= 1:
        return A

    mid = len(A) // 2
    return merge(merge_sort(A[:mid]), merge_sort(A[mid:]))
```

Merge sort – složitost

Složitost merge sortu

- funkce merge má složitost $\mathcal{O}(n)$
- počet úrovní volání funkce funkce merge je $\mathcal{O}(\log n)$
- složitost merge sortu je $\mathcal{O}(n \log n)$

Pozorování

- dolní odhad složitosti problému řazení je $\Omega(n \log n)$
- složitost merge sortu je $\mathcal{O}(n \log n)$

Závěr. Merge sort je optimální algoritmus pro problém řazení se složitostí $\mathcal{O}(n \log n)$.

Stabilita řadicích algoritmů

Definice. Řadicí algoritmus je stabilní, pokud neprohazuje prvky se stejným klíčem.

Příklad. Uvažme seznam lidí seřazených podle jména. Pokud jej sežadíme dále stabilním algoritmem podle příjmení, získáme seznam seřazený podle příjmení a jména.

	A	B
1	**Jméno**	**Příjmení**
2	Marek	Dostál
3	Petr	Klíč
4	Petr	Dostál
5	Prokop	Buben
6	Tomáš	Fuk

Přirozenost řadicích algoritmů

Definice. Řadicí algoritmus je přirozený, pokud dokáže využít předuspořádání vstupní posloupnosti.

Důsledek. Přirozené řadicí algoritmy řadí částečně seřazené posloupnosti v lepším čase.

Příklad. Modifikace algoritmu bubble sort:

```
def bubble_sort(A):
    for i in range(len(A)):
        swapped = False
        for j in range(len(A) - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                swapped = True

        if not swapped:
            return
```

Součet čísel

Experiment. Určete součet zadaného souboru reálných čísel.

Řešení. V Pythonu například funkce sum nebo explicitně:

```
total = 0
for item in array:
    total += item
```

Pozorování. Uvažme stejnou sadu čísel, jen v jiném (zcela náhodně zvoleném) pořadí.

```
array2 = sorted(array, key = lambda k : random.random())
```

Zřejmě platí:

```
sum(array) == sum(array2)
```

Nebo ne?

Asociativita sčítání reálných čísel

Příklad. Uvažme výpočet 1,11 + 0,001 s přesností na 3 platné číslice.

Poznámka. Počet platných desítkových číslic pro typ float je průměrně 7, pro typ double pak 16.

- 1,11 + 0,001 = 1,11
- korektní výsledek v rámci zvolené přesnosti

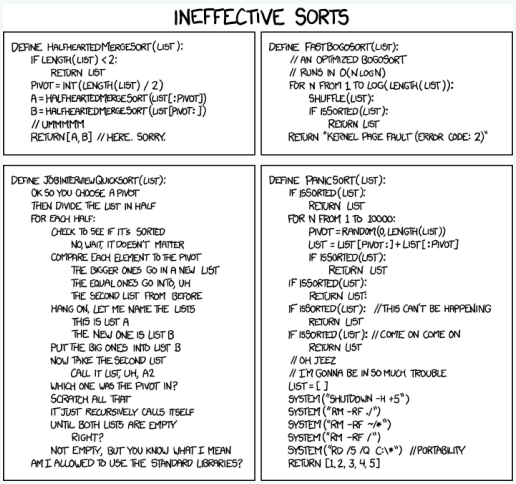
Srovnejme

- 1,11 + 0,001 + ...+ 0,001
- 1,11 + (0,001 + ...+ 0,001)

Závěr. Sčítání reálných čísel není asociativní.

Doporučení. Pokud je přesnost důležitá, sčítám čísla v pořadí podle jejich (absolutní) velikosti.

xkcd.com/1185/



StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.