Podstawy programowania i przetwarzania danych

Podstawowe algorytmy sortowania za pomocą porównań

Małgorzata Śleszyńska-Nowak
malgorzata.nowak@pw.edu.pl
http://pages.mini.pw.edu.pl/~sleszynskam/

Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

Spis treści

Problem sortowania za pomocą porównań

Problem sortowania za pomocą porównań

Niech $(D, <=)$ będzie zbiorem z totalnym praporządkiem (relacja, która jest zwrotna, przechodnia i całkowita, nie musi być antysymetryczna – intuicyjnie: to, że $student_1 \leq student_2$ i $student_2 \leq student_1$ nie musi oznaczać, że $student_1 = student_2$).

Wejście: lista `t` $= [t_0, t_1, \ldots, t_{n-1}] \in D^n$

Wyjście: przepermutowana wersja listy wejściowej,

$$t_{\sigma(0)}, t_{\sigma(1)}, \ldots, t_{\sigma(n-1)},$$

gdzie σ jest permutacją porządkującą `t`, tj. taka, że:

$$t_{\sigma(0)} <= t_{\sigma(1)} <= \ldots <= t_{\sigma(n-1)}.$$

Zazwyczaj po prostu zmieniamy kolejność elementów w t (np. poprzez zamianę miejscami w świadomy sposób dopóki nie otrzymamy posortowanej listy) **w miejscu** (bez użycia dodatkowej pamięci).

```
1   def sort3(t):
2       """ sortuje 3-elementową listę """
3       assert len(t) == 3
4       if not t[0] <= t[1]:
5           t[0], t[1] = t[1], t[0]
6       if not t[1] <= t[2]:
7           t[1], t[2] = t[2], t[1]
8       if not t[0] <= t[1]:
9           t[0], t[1] = t[1], t[0]
10
11  t = [3, 2, 1]
12  sort3(t)
13  assert t[0] <= t[1] and t[1] <= t[2] # t[0] <= t[2] przez przechodniość
```

Jeśli nie chcemy modyfikować t, to działajmy na kopii t.

W języku Python mamy wbudowane sortowania (i ich używamy w praktyce):

```
t = [5, 2, 3, 1, 4]
s_t = sorted(t) # zwraca posortowaną kopię, nie zmienia listy t
print(s_t)      # [1, 2, 3, 4, 5]
t.sort()        # metoda, sortuje listę t
print(t)        # [1, 2, 3, 4, 5]
```

używany algorytm: Tim-sort (hybryda algorytmów, które poznamy na następnym wykładzie)

Ogólnie, permutacja indeksów n-elementowej listy jest bijekcją σ: $\{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$.

Taka permutacja określa pewne przestawienie elementów: element pod indeksem i jest zastępowany przez ten pod indeksem $\sigma(i)$.

Dwuliniowa notacja Cauchy'ego:

$$\begin{pmatrix} 0 & 1 & \ldots & n-1 \\ \sigma(0) & \sigma(1) & \ldots & \sigma(n-1) \end{pmatrix}$$

Jednoliniowa notacja:

$$\begin{pmatrix} \sigma(0) & \sigma(1) & \ldots & \sigma(n-1) \end{pmatrix}$$

i w ten sposób możemy przechowywać permutację jako listę.

Permutacja określa pewne przestawienie elementów: element pod indeksem i jest zastępowany przez ten pod indeksem $\sigma(i)$.

Niech $\mathtt{t} = [50, 30, 40, 10, 20]$

Permutacja porządkująca to:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 & 0 \end{pmatrix}$$

$\sigma(0)$ – indeks najmniejszego elementu $\sigma(1)$ – indeks drugiego najmniejszego elementu $\sigma(n-1)$ – indeks największego elementu

Tak więc posortowana wersja $\mathtt{t}$ to $[t_3, t_4, t_1, t_2, t_0]$.

*Sortowanie w numpy:

```
import numpy as np
x = np.array([50, 30, 40, 10, 20])
y = np.sort(x)                # posortowana kopia (zobacz również x.sort())
# x: array([50, 30, 40, 10, 20])
# y: array([10, 20, 30, 40, 50])
sigma = np.argsort(x)         # permutacja porządkująca
# sigma: array([3, 4, 1, 2, 0])
x[sigma]                      # x[sigma[0]], x[sigma[1]], ..., x[sigma[n-1]]
# array([10, 20, 30, 40, 50])
```

Zwróćmy uwagę, że permutacja porządkująca może nie być unikalna:

Niech $\mathtt{t} = [1, 2, 1, 3, 2, 1]$

Mówimy, że algorytm sortujący jest **stabilny**, jeśli generuje on permutację porządkującą σ taką że dla wszystkich $i < j$:

$$t_{\sigma(i)} == t_{\sigma(j)} \rightarrow \sigma(i) < \sigma(j)$$

„Stabilna" permutacja $\mathtt{t}$:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 2 & 5 & 1 & 4 & 3 \end{pmatrix}$$

Stabilne sortowanie jest przydatne np. gdy chcemy posortować dane po więcej niż jednym kryterium. Sortujemy wtedy najpierw po jednym kryterium, a następnie po drugim. Sortowanie po drugim nie może nam zniszczyć porządku pomiędzy równymi elementami (wg drugiego kryterium), gdyż są one posortowane (i nierówne) względem pierwszego kryterium.

Posortowanie wg pierwszego kryterium (kolumna `tip`):

```
#      tip     sex
#  0  1.01  Female
#  1  1.66    Male
#  3  3.31    Male
#  2  3.50    Male
#  4  3.61  Female
#  5  4.71    Male
```

Posortowanie wg drugiego kryterium (kolumna `sex`):

```
#      tip     sex
#  0  1.01  Female
#  4  3.61  Female
#  1  1.66    Male
#  3  3.31    Male
#  2  3.50    Male
#  5  4.71    Male
```

Do czego może się przydać sortowanie?

- ładne wypisywanie (aby łatwiej czytało się człowiekowi),
- wyszukiwanie (np. wyszukiwanie binarne),
- znajdowanie statystyk pozycyjnych (min, max, median, kwartyle i inne kwantyle),
- znajdowanie najbliższych sąsiadów,
- usuwanie duplikatów,
- grupowanie danych, obliczanie funkcji agregujących w podgrupach,
- itp.

Algorytmy sortowania (podstawowe)

Dość absurdalny algorytm **Bogosort**.

```
1  Dla każdej permutacji σ zbioru {0, ..., n − 1}:
2      Jeśli t_σ(0)<=t_σ(1)<= ... <=t_σ(n−1):
3          Zwróć [t_σ(0), t_σ(1), ..., t_σ(n−1)]
```

Złożoność czasowa: $O(n \cdot n!)$, operacja dominująca: <=.

Poprawny, acz bezużyteczny. Można lepiej.

Kryteria oceny algorytmów sortowania:

- liczba porównań
- liczba przypisań (np. zamian jak `t[i],t[j] = t[j],t[i]`), zapisywanych jako $t_i \leftrightarrow t_j$
- wykorzystywana pamięć (poza wejściowym `t` i wyjściowym σ)
- stabilność

Zaczynamy od podstawowych (niezbyt skomplikowanych i niekoniecznie wydajnych) algorytmów sortowania. Wszystkie mają złożoność czasową $O(n^2)$ (ze względu na porównania). Podstawowe algorytmy są używane dla małych ilości danych (powiedzmy $n \leq 20$) – okazuje się, że wtedy są wydajniejsze od bardziej skomplikowanych podejść.

Sortowanie przez wybór (ang. *selection sort*). Główna idea:

1. znajdź najmniejszy element, zamień go z elementem na indeksie 0,
2. znajdź drugi najmniejszy element, zamień go z elementem na indeksie 1,
3. itd.

Pseudokod:

```
1  Dla i = 0, 1, ..., n − 2:
2      j = argmin_{k=i,i+1,...,n−1} t_k
3      t_i ↔ t_j
```

Warunek poprawności: po i-tej iteracji elementy na indeksach $0, 1, \ldots, i$ są na swoich ostatecznych pozycjach.

Przykładowa implementacja:

```
1  def selection_sort(t):
2      """sortuje w miejscu"""
3      n = len(t)
4      for i in range(n-1):
5          j = i
6          for k in range(i+1, n):
7              if not t[j] <= t[k]:
8                  j = k
9          t[i], t[j] = t[j], t[i]
```

Analiza algorytmu:

	najlepszy przypadek	pesymistyczny przypadek
Porównania <=	$n(n-1)/2^{\dagger}$	$n(n-1)/2^{\dagger}$
Zamiany $\leftrightarrow$	$n-1^{*}$	$n-1^{\$}$
Pamięć	$O(1)$	$O(1)$

$*$ – możemy użyć pojedynczej instrukcji `if` i mamy $= 0$

$\dagger$ – $(n-1)+(n-2)+\ldots+1 = (1+(n-1))(n-1)/2 = n(n-1)/2 = \Theta(n^2)$

\$ – optymalnie

Czy algorytm jest stabilny? Nie. Kontrprzykład: `t = [1, 1, 0]`.

Przykładowa implementacja zwracająca permutację porządkującą:

```
1  def selection_argsort(t):
2      """znajduje permutację porządkującą"""
3      n = len(t)
4      s = list(range(n)) # [0, 1, ..., n-1]
5      for i in range(n-1):
6          j = i
7          for k in range(i+1, n):
8              if not t[s[j]] <= t[s[k]]:
9                  j = k
10         s[i], s[j] = s[j], s[i]
11     return s
```

Sortowanie przez wstawianie (ang. *insertion sort*). Główna idea:

1. załóżmy, że pierwszych kilka elementów jest już posortowanych
2. pojawia się nowy element
3. gdzie powinniśmy wstawić ten nowy element, żeby cała sekwencja nadal była posortowana?

Pseudokod:

```
1   Dla i = 1, 1, ..., n − 1:
2       Znajdź największe j ≤ i takie że t_{j−1}<=t_i  # niech t_{−1} = −∞
3       Wstaw t_i pomiędzy [t_0, t_1, ..., t_{j−1}, t_j, ..., t_{i−1}]
4       tak że pierwsze i elementy to [t_0, t_1, ..., t_{j−1}, t_i, t_j, ..., t_{i−1}]
```

Warunek poprawności: po i-tej iteracji zachodzi $t_0 <= t_1 <= \ldots <= t_{i-1}$ – pierwsze i elementów jest zawsze posortowanych (ale być może nie są na swoich ostatecznych pozycjach).

Przykładowa implementacja:

```
1  def insertion_sort(t):
2      """sortuje w miejscu"""
3      for i in range(1, len(t)):
4          j, tcur = i, t[i]
5          while j > 0:
6              if t[j-1] <= tcur:
7                  break
8              t[j] = t[j-1]
9              j -= 1
10         t[j] = tcur
```

Analiza algorytmu:

	najlepszy przypadek	pesymistyczny przypadek
Porównania <=	$n-1$	$n(n-1)/2$
Przypisania =	$n-1$	$O(n^2)$
Pamięć	$O(1)$	$O(1)$

Czy algorytm jest stabilny? Tak.

Poprzez konstrukcję: nie pozwalamy, aby element równy poprzednikowi był wstawiony przed nim.

Sortowanie bąbelkowe (ang. *bubble sort*). Algorytm, który zna i wymienia w pierwszej kolejności każdy. Niezbyt użyteczny, za to łatwo się go implementuje i rozumie sposób działania.

Główna idea: porównaj (i zamień, jeśli trzeba) sąsiadujące pary elementów (czy $t_i \leq t_{i+1}$?)

Pseudokod:

```
1  Dla i = 0, 1, ..., n − 1:
2      Dla j = 0, 1, ..., n − i − 1:
3          Jeśli t_j > t_{j+1}:  t_j ↔ t_{j+1}
```

Warunek poprawności: po i-tej iteracji ostatnie i elementów jest na swojej ostatecznej pozycji.

Przykładowa implementacja:

```
1  def bubble_sort(t):
2      """sortuje w miejscu"""
3      n = len(t)
4      for i in range(1, n):
5          for j in range(n-i):
6              if not t[j] <= t[j+1]:
7                  t[j], t[j+1] = t[j+1], t[j]
```

Analiza algorytmu:

	najlepszy przypadek	pesymistyczny przypadek
Porównania <=	$n(n-1)/2^*$	$n(n-1)/2$
Zamiany $\leftrightarrow$	0	$n(n-1)/2$
Pamięć	$O(1)$	$O(1)$

$*$ – można zejść do $= n-1$ jeśli zakończymy algorytm gdy nie wykona żadnej zamiany w wewnętrznej pętli

Czy algorytm jest stabilny? Tak.

Poprzez konstrukcję: nie pozwalamy, aby element równy następnikowi był wstawiony za nim.

Uwagi końcowe

- Sortowanie przez wybór ma optymalną liczbę zamian
- Sortowanie przez wstawianie jest jednym z najszybszych algorytmów sortowania małych tablic
- Sortowanie przez wstawianie jest naprawdę szybkie dla prawie posortowanych danych (częsty przypadek)
- Sortowanie przez wstawianie jest często używane aby przyspieszyć bardziej skomplikowane algorytmy (patrz poniżej)
- Sortowania przez wstawianie jest stabilne, lubimy to
- Sortowanie przez wstawianie ma wartą uwagi modyfikację: sortowanie (niestabilne) Shella (dla ciekawskich)

Dziękuję

malgorzata.nowak@pw.edu.pl