Algorytmy i Struktury Danych, 2. ćwiczenia

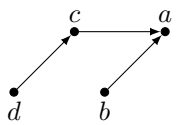
2018-10-10

Spis treści

1 Optymalne sortowanie 5–ciu elementów

Niech $A = (a, b, c, d, e)$.

- $compare(a, b)$, (bez straty ogólności, niech $a < b$)
- $compare(c, d)$, (niech $c < d$)
- $compare(a, c)$, (niech $a < c$)



- teraz wsortowujemy, e pomiędzy a, c, d,
  if $(e > c)$ **then** $compare(e, a)$ **else** $compare(e, d)$
- możemy otrzymać jeden z następujących częściowych porządków:

każdy z nich można posortować używając 2 porównań.

2 Sortowanie metodą Shella

Lemat 1 *Niech m, n, r będą nieujemnymi liczba całkowitymi i niech $(x_1, \ldots, x_{m+r})$ oraz $(y_1, \ldots, y_{n+r})$ będą dowolnymi ciągami liczbowymi takimi, że $y_i \leq x_{m+i}$ dla $1 \leq i \leq r$. Jeśli elementy x oraz y posortujemy niezależnie tak, że $x_1 \leq \ldots \leq x_{m+r}$ oraz $y_1 \leq \ldots \leq y_{n+r}$ to nadal będziemy mieli $y_i \leq x_{m+i}$ dla $1 \leq i \leq r$.*

Po posortowaniu element x_{m+i} jest większy bądź równy od co najmniej $m+i$ elementów z x, wśród nich jest co najmniej i elementów które przed sortowaniem były na pozycjach $m, \ldots, m+r$, każdy z tych elementów ma wśród y element od którego jest większy, stąd x_{m+i} jest większy bądź równy od i najmniejszych elementów y.

(pełny dowód jest w Knuth, tom III, strona 94)

Lemat 2 *Jeśli tablica jest h posortowana i k posortujemy, to nadal będzie h posortowana.*

Niech a_i i a_{i+h} elementy które po sortowaniu nie są h posortowane. Niech Y ciąg zawierający a_i, a_{i+k}, a_{i+2k}, ... Niech X ciąg zawierający a_{i+h}, a_{i+h+k}, a_{i+h+2k}, ... Po k posortowaniu ciągi Y i X są uporządkowane, z poprzedniego lematu mamy jednak, że $a_i \leq a_{i+h}$ — sprzeczność.

Lemat 3 *Liczba porównań wymagana przy h posortowaniu tablicy rozmiaru n wynosi $O(n^2/h)$.*

Mamy h ciągów, każdy o długości n/h — stąd całkowity czas wynosi $h \cdot n^2/h^2 = n^2/h$.

Lemat 4 *Liczba porównań wymagana przy h_i posortowaniu tablicy rozmiaru n, która jest h_{i+1} i h_{i+2} posortowana wynosi $O(nh_{i+1}h_{i+2}/h_i)$ (przy założeniu, że h_{i+1} i h_{i+2} są względnie pierwsze)*

Trzeba pokazać, że jeśli ciąg jest h_{i+1} i h_{i+2} posortowany, to jeśli $k \geq h_{i+1}h_{i+2}$, to $a_i \leq a_{i+k}$.

Lemat 5 *Dla ciągu $h = \{2^i - 1 : i \in N\}$ algorytm ShellSort ma złożoność $O(n\sqrt{n})$.*

(Knuth, tom III, strona 95)

Niech B_i koszt i–tej fazy, $t = \lceil \log n \rceil$. Dla pierwszy $t/2$ przebiegów $h \geq \sqrt{n}$, ponieważ koszt jednej fazy jest ograniczona przez $O(n^2/h)$ stąd sumaryczny koszt jest rzędu $O(n^{1.5})$. Dla pozostałych przebiegów możemy skorzystać z poprzedniego lematu, koszt pojedynczej fazy jest równy $B_i = O(nh_{i+2}h_{i+1}/h_i)$, więc sumaryczny koszt tych faz jest również rzędu $O(n^{1.5})$.

Lemat 6 *Dla ciągu $h = \{2^i 3^j : i, j \in N\}$ algorytm ShellSort ma złożoność $O(n \log^2 n)$.*

Wszystkich faz algorytmu jest $O(\log^2 n)$. Trzeba pokazać, że każda z faz zajmuje czas $O(n)$.

Obserwacja: jeśli ciąg jest 2 i 3–uporządkowany, to jego 1–posortowanie wymaga czasu $O(n)$.

Analogicznie jeśli ciąg jest $2i$ oraz $3i$–uporządkowany to jego i–posortowanie wymaga czasu $O(n)$.

3 Przesunięcie cykliczne tablicy

Function CyclicLeftShift(a, k)

$n := len(a)$
Reverse$(a,\ 1,\ n-k)$
Reverse$(a,\ n-k+1,\ n)$
Reverse$(a,\ 1,\ n)$

4 Scalanie w miejscu dla ciągów długości $\sqrt{n}$ i $n-\sqrt{n}$

Algorithm 1: Merge(A)

Dana jest tablica A zawierająca dwa uporządkowane rosnąco ciągi: $1..n-\sqrt{n}$ i $n-\sqrt{n}+1..n$.
Posortuj (używając alg. insertion sort) ciąg $n-2\sqrt{n}+1..n$
Scal ciąg $1..n-2\sqrt{n}$ i $n-2\sqrt{n}+1..n-\sqrt{n}$ używając obszaru $n-\sqrt{n}+1..n$ jako bufor
Posortuj (używając alg. insertion sort) ciąg $n-\sqrt{n}+1..n$

Scalanie dwóch uporządkowanych w ciągu przy użycia bufora:

```
def merge(A, n1, C):
    """scalanie uporzadkowanych A[0..n1) i A[n1..]
    przy uzyciu bufora C (dlugosci >= min(n1, |A|-n1))"""
    assert len(C) >= min(n1, len(A) - n1)
    n = len(A)
    if n1 > n - n1:
        # zamien A[0..n1) i A[n1..] - wersja uproszczona!
        reverse = lambda arr: arr[::-1]  # to nie jest O(1)
        A[:n1], A[n1:] = reverse(A[:n1]), reverse(A[n1:])
        A[:n] = reverse(A[:n])
        n1 = n - n1

    # zamien A[0..n1) i C
    for i in range(n1):
        A[i], C[i] = C[i], A[i]

    i, i1, i2 = 0, 0, n1
    while i1 < n1:
        # fragment A[0:i) jest juz uporzadkowany
        # pozostale elementy sa w C[i1:n1) i A[i2:n)
        if i2 == n or C[i1] <= A[i2]:
            A[i], C[i1] = C[i1], A[i]
            i1 += 1
        else:
            A[i], A[i2] = A[i2], A[i]
```

```
            i2 += 1
        i += 1
```

Uwaga! W tym kodzie dla uproszczenia zapisu użyłem do odwracania listy konstrukcji [::-1], która używa dodatkowej pamięci, jednak łatwo to zastąpić prostą pętlą odwracająca dany zakres.

5 Scalanie w miejscu

Knuth, Tom III, strona 698.

- podziel tablicę na bloki rozmiaru $\lceil\sqrt{n}\rceil$, — Z_1, Z_2, ..., Z_{m+2}, (blok Z_{m+2} może być mniejszy,
- zamień blok leżący na połączeniu dwóch ciągów, z blokiem Z_{m+1}, teraz każdy z bloków Z_1, ..., Z_m jest uporządkowany,
- posortuj używając selection–sort bloki, wg. pierwszego elementu z bloków (jeśli dwa bloki mają ten sam element początkowy, to porównaj elementy końcowe)
- scal Z_1, ..., Z_m używając Z_{m+1} jako bufora pomocniczego,

  Algorithm 2: Z-Merge(Z)

  foreach $i \in 1, \ldots, m-1$ **do**
  SimpleMerge(Z_i,Z_{i+1},Z_{m+1})

  (należy jeszcze pokazać, że taka procedura daje dobre uporządkowanie) — wskazówka: przed tym krokiem każdy element jest w inwersji z co najwyżej $\sqrt(n)$ innymi elementami bloków $Z_1, .., Z_{m+1}$
- dzielimy tablicę na trzy części: A, B, C, $|B| = |C| = 2\lceil\sqrt{n}\rceil$
- posortuj ostatnie $4 \cdot \lceil\sqrt{n}\rceil$ elementów (bloki B, C) używając InsertionSort (w rezultacie w bloku C znajdują się największe elementy w tablicy)
- scal bloki A i B używając C jako bufora
- posortuj blok C używając InsertionSort

Ćwiczenie: dlaczego używając selection-sort trzeba uwzględniać początki i końce bloków?

Na przykład dla ciągów (111,123),(111,145) (rozmiar bloku 3), sortując jedynie po początkach moglibyśmy otrzymać: (123,145,111,111), który przy scalaniu metodą opisaną w algorytmie nie da uporządkowanego ciągu.

6 Kodowanie permutacji

Zadanie: dla zadanego wektora inwersji permutacji $w(\pi)$, odkoduj oryginalną permutację π.
Gdzie:

$$w(\pi)[i] = |\{j : 1 \leq j < i \text{ oraz } \pi[j] > \pi[i]\}|$$

Rozwiązanie: `https://www.mimuw.edu.pl/~jrad/asd/inwersje.pdf`