Algorytmy i Struktury Danych, 2. ćwiczenia

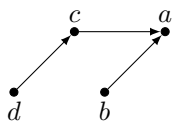
2017-10-13

Spis treści

1 Optymalne sortowanie 5–ciu elementów

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

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



- teraz wkładamy, e pomiędzy a, c, d,
  if $(e > c)$ then $compare(e, a)$ else $compare(e, d)$
- możemy otrzymać jeden z następujących posetów:

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 $h \cdot n^2/h^2 = n^2/h$. qued

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 $a_i \leq a_{i+k}$. qued

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 $O((\log n)^2)$. Trzeba pokazać, że każda z nich zajmuje $O(n)$ czasu. Obserwacja — jeśli ciąg jest 2 i 3 uporządkowany, to jego 1–posortowanie wymaga $O(n)$ czasu. Analogicznie jeśli ciąg jest $2i$ i $3i$ posortowany to jego i posortowanie wymaga $O(n)$.

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 $n - \sqrt{n} + 1..n$

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? Rozwiązanie: np. 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 Zadania z klasówek

Zadanie 1

W tym zadaniu rozważamy rekurencyjny algorytm sortowania przez scalanie, w którym scalanie dwóch posortowanych ciągów odbywa się w sposób klasyczny: na swoją docelową pozycję trafia mniejszy z początkowych elementów scalanych ciągów.

Przykład Podczas scalania ciągów [2, 4, 5, 8] oraz [1, 3, 6, 7] porównywane są kolejno 2 z 1, 2 z 3, 4 z 3, 4 z 6, 5 z 6, 8 z 6 oraz 8 z 7.

Zaprojektuj liniowy algorytm, który sprawdzi, czy w wyniku wykonania algorytmu sortowania przez scalanie na danej permutacji p[1..n] liczb naturalnych $\{1, \ldots, n\}$, porównane zostaną ze sobą zadane, dwie różne liczby a i b ze zbioru $\{1, \ldots, n\}$.

Rozwiązanie: 1. Znajdź wspólny poziom rekurencji na której te elementy są scalane
2. Dla a i b wyznacz pierwszy większy/mniejszy element a', a'', b', b''
3. Sprawdź cośtam cośtam.

Zadanie 2

Udowodnij, że jeśli algorytm sortujący tablicę A[1..n] porównuje i zamienia wyłącznie elementy odległe co najwyżej o 2015 (tzn. jeśli porównuje A[i] z A[j], to |i-j| <= 2015), to jego pesymistyczny czas działania jest co najmniej kwadratowy.

Rozwiązanie: Zauważmy, że zamiana elementów odległych o co najwyżej 2015, może zmniejszyć liczbę inwersji o $O(1)$. Ponieważ tablica może zawierać $O(n^2)$ inwersji, stąd czas działania dowolnego algorytmu o tej własności będzie $\Omega(n^2)$.

Zadanie 3

Dana jest tablica a[1..n] parami różnych elementów pochodzących ze zbioru z liniowym porządkiem. Należy posortować tablicę a rosnąco. Jedyną operacją służącą do porównywania elementów między sobą jest funkcja ile(x,y), której wynikiem jest liczba całkowita k zdefiniowana tak, że

$$|k| = |\{1 \leq i \leq n : \min(x, y) \leq a[i] \leq \max(x, y)\}|.$$

Wartość k jest ujemna tylko wtedy, gdy x jest mniejsze od y. Udowodnij, że każdy algorytm sortujący a wywoła funkcję ile w pesymistycznym przypadku co najmniej n-1 razy.

Zaproponuje algorytm sortowania a w miejscu za pomocą co najwyżej $O(n)$ wywołań funkcji ile i $O(n)$ zamian.

Rozwiązanie: 1. Znajdź minimum
2. Przejdź po wszystkich elementach i wstaw je w odpowiednie miejsce (uwaga! to musi być w miejscu)