Algorytmy i Struktury Danych, 3. ćwiczenia

2018-10-17

Spis treści

1 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.

2 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.

3 Budowa kopca w algorytmie HeapSort

Kopiec możemy budować idąc od dołu do góry. Zauważmy, że ostatnie $n/2$ elementów spełnia warunki kopca, pozostaje jedynie poprawić porządek w pierwszych $n/2$ elementach.

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  DownHeap(i)
end for

Koszt budowy kopca możemy opisać wzorem:

$$\frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 4 + \frac{n}{16} \cdot 6 + \ldots = n \cdot \sum_{i=1} \frac{i}{2^i} =$$

$$n \cdot (\sum_{i=1} \frac{1}{2^i} + \sum_{i=2} \frac{1}{2^i} + \sum_{i=3} \frac{1}{2^i} + \ldots) = n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots) = 2n$$

4 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 mniejszy/większy element a', a'', b', b''

3. Sprawdź, czy a i b są porównywane podczas scalania ciągów (a', a, a'') i (b', b, b'').

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)