Algorytmy i Struktury Danych, 3. ćwiczenia

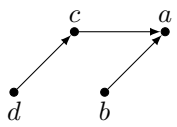
2019-10-16

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 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.

3 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.
Inne wyniki: istnieje również trochę bardziej skomplikowany algorytm, który pozwa na scalanie dwóch uporządkowanych ciągów w czasie liniowym, w stałej pamięci i dodatkowo **stabilnie** ([1]).

4 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:

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

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

Średnia liczba porównań - dolne ograniczenie

Udowodnij, że średnia liczba porównań niezbędnych do posortowania ciągu n-elementowego w modelu losowej permutacji wynosi co najmniej $n \log n - 1.45n$.
Rozwiązanie:
Wiemy, że dowolne drzewo decyzyjne dla problemu sortowania ma wysokość $h \geq \lceil \log_2 n! \rceil$.
Ze wzoru Stirlinga wiemy, że:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Stąd:

$$\log_2(n!) \approx \log_2 \left(\frac{n^n \cdot \sqrt{2\pi n}}{e^n} \right)$$

$$\log_2(n!) \approx n \log_2 n - n \log_2(e) + O(\log_2 n)$$

$$\log_2(n!) \approx n \log_2 n - 1.442695n + O(\log_2 n)$$

Bibliografia

[1] Viliam Geffert, Jyrki Katajainen i Tomi Pasanen. "Asymptotically efficient in-place merging". W: *Theor. Comput. Sci.* 237.1-2 (2000), s. 159–181. DOI: 10.1016/S0304-3975(98)00162-5. URL: https://doi.org/10.1016/S0304-3975(98)00162-5.