Algorytmy i Struktury Danych - zadanie 6.4.a

Maksymilian Zawartko

15 czerwca 2020

1 Zadanie

Ułóż algorytm sortujący w miejscu ciągi rekordów o kluczach ze zbioru 1, 2, 3.

2 Idea rozwiązania

Trzymaj indeks "i" (index_of_not_1) pierwszego elementu, co do którego zachodzi podejrzenie, że zaburza porządek w ciągu. W pierwszej fazie jest to pierwszy element, którego klucz nie jest 1. W tej też fazie trzymaj drugi index "j" (later_index_of_1) pierwszego elementu za "i", którego klucz jest 1. Zamieniaj elementy z indeksów "i" i "j" miejscami, zwiększaj oba indeksy, aż dojdziesz indeksem "j" do końca ciągu.
W drugiej fazie postępuj podobnie, ale "i" (index_of_not_2) indeksuj element, którego klucz nie jest 2, natomiast "j" (later_index_of_2) - ten, którego klucz jest 2.

3 Pseudokod

```
def sort(array):
    index_of_not_1 = 0
    while index_of_not_1 < len(array) and
            array[index_of_not_1].key == 1:
        index_of_not_1 += 1

    later_index_of_1 = index_of_not_1

    while later_index_of_1 < len(array):
        while later_index_of_1 < len(array) and
                array[later_index_of_1].key != 1:
            later_index_of_1 += 1

        if later_index_of_1 < len(array):
            array[index_of_not_1], array[later_index_of_1] =
```

```
            array[later_index_of_1], array[index_of_not_1]

        while index_of_not_1 < len(array) and
                array[index_of_not_1].key == 1:
            index_of_not_1 += 1

index_of_not_2 = index_of_not_1
while index_of_not_2 < len(array) and
        array[index_of_not_2].key == 2:
    index_of_not_2 += 1

later_index_of_2 = index_of_not_2

while later_index_of_2 < len(array):
    while later_index_of_2 < len(array) and
            array[later_index_of_2].key != 2:
        later_index_of_2 += 1

    if later_index_of_2 < len(array):
        array[index_of_not_2], array[later_index_of_2] =
            array[later_index_of_2], array[index_of_not_2]

        while index_of_not_2 < len(array) and
                array[index_of_not_2].key == 2:
            index_of_not_2 += 1
```

4 Uzasadnienie poprawności

To, że algorytm działa w miejscu jest oczywiste. Nie alokuje on żadnych nowych struktur danych; korzysta tylko z czterech (O(1)) dodatkowych zmiennych (index_of_not_1, index_of_not_2, later_index_of_1, later_index_of_2).

Uzasadnienia może wymagać fakt, że algorytm faktycznie sortuje. Widać to jednak po sposobie, w jaki działa. Wszystkie elementy ciągu przed indeksem index_of_not_1, a później index_of_not_2, są posortowane (najpierw znajdują się tam same jedynki (rekordy z kluczem 1 - dalej również używam tego skrótu myślowego), później spójny ciąg jedynek, a za nimi spójny ciąg dwójek). W pierwszej fazie algorytmu do ciągu jedynek na indeks index_of_not_1 ciągle przenoszone są jedynki, a to, co było w ich miejscu, jest przenoszone dalej w ciąg. W drugiej fazie w identyczny sposób dwójki przenoszone są na indeks index_of_not_1. Warto zwrócić uwagę, że w tej fazie w dalszej po indeksie index_of_not_1 części ciągu są już tylko dwójki i trójki (wszystkie jedynki zostały przeniesione na początek w pierwszej fazie).
Analogicznie, po drugiej fazie po indeksie index_of_not_1 są już same trójki, a więc ciąg wejściowy jest sortowany przez procedurę sort.

5 Złożoność algorytmu

O(n), gdzie n jest długością ciągu wejściowego, bo ograniczone przez n są zarówno zewnętrzne pętle while, jak i zagnieżdżone (każdy obrót zagnieżdżonej pętli przyczynia się do szybszego skończenia zewnętrznej, w której się znajduje, w przypadku pętli "while later_index_of_{1, 2} <len(array) and ..."; lub liczba obrotów zagnieżdżonej pętli nie zależy od liczby obrotów zewnętrznej, w przypadku pętli "while index_of_not_{1, 2} <len(array) and ...").