SORTOWANIE PRZEZ ZLICZANIE | JAVA

WSTĘP

Mechanizm **counting sorta**, jak sama nazwa wskazuje, opiera się na zliczaniu ilości wystąpień wszystkich wartości z danego przedziału. Jak wiadomo najlepsza możliwa złożoność czasowa dla algorytmów sortowania wykorzystujących porównania wynosi Ω(nlogn). Natomiast sortowanie przez zliczanie nie jest jednym z nich i posiada złożoność czasową O(n+k), gdzie n jest długością tablicy do posortowania, a k jest długością przedziału, z którego mogą być liczby w otrzymanej tablicy. Pomimo tak zachęcająco wyglądającej złożoności, ukrytych jest jednak również kilka wad.

IMPLEMENTACJA W JAVIE

W załączonym przeze mnie kodzie, napisanym w języku Java, znajduję się prosty program, który po uruchomieniu umożliwia komunikację z użytkownikiem. Program pozwala na własnoręczne wpisanie tablicy znaków całkowitoliczbowych (int) oraz wypisuje już posortowaną tablicę. Wszystko dzieje się za sprawą algorytmu sortowania przez wstawianie, którego implementacja znajduje się   
(w liniach 14-33 przesłanego pliku Main.java) pod postacią funkcji *countingSort*, która wygląda następująco:

1 void countingSort(int arr[], int max\_val) {

2       int[] counts = new int[max\_val+1];

3       for(int i = 0; i <= max\_val; i++) {

4           counts[i] = 0;

5       }

6       for(int i = 0; i < arr.length; i++) {

7           counts[arr[i]]++;

8       }

9       for(int i = 1; i <= max\_val; i++) {

10          counts[i] += counts[i-1];

11      }

12      int[] sorted = new int[arr.length];

13      for(int i = arr.length - 1; i >= 0; i--) {

14          sorted[counts[arr[i]] - 1] = arr[i];

15          counts[arr[i]]--;

16      }

17      for(int i = 0; i < arr.length; i++) {

18          arr[i] = sorted[i];

19      }

20  }

OPIS ZAIMPLEMENTOWANEGO ALGORYTMU

Działanie poszczególnych linijek widocznego powyższego kodu:

1 Aby posortować tablicę *countingSort* potrzebuje tablicę oraz informację o wszystkich możliwych wartościach w tej tablicy. W mojej implementacji algorytm zlicza ilość wystąpień każdej liczby całkowitej z przedziału od 0 do max\_val przekazywanego w argumencie do funkcji.

2 W drugiej linii zostaje utworzona tablica counts[] o max\_val+1 elementach do zliczania wystąpień wartości z przedziału od 0 do max\_val.

3-5 Wszystkie wartości w tablicy counts[] zostają wyzerowane.

6-8 Uzupełniamy teraz tablicę counts[] odpowiednimi liczbami odpowiadającym ilości wystąpień elementów o wartości równej indeksowi w tablicy counts[] (np. jeżeli w wejściowej tablicy znajdują się cztery 7 to counts[7] będzie równe 4).

9-11 Następnym krokiem jest zmodyfikowanie tablicy counts[], tak aby każdy element posiadał ilość wystąpień elementów mniejszych lub równych od siebie.

12 Tworzymy nową tablicę sorted[], do której będą wpisywane wartości z wejściowej tablicy w odpowiedni sposób, tak żeby otrzymać posortowaną tablice.

13-16 Zaczynając od końca wejściowej tablicy będziemy przepisywać wartości do końcowej tablicy. Załóżmy, że na początku chcemy poprawnie przepisać element o wartości 6. Jeżeli jest n elementów o wartości mniejszej lub równej 6 to wpisujemy 6 na n-te miejsce (czyli o indeksie równym n-1) w tablicy sorted[]. Dokonujemy tego w 14 linii. Następnie (w 15 linii) należy zmniejszyć o 1 wartość tablicy counts[] dotyczącą elementu, który przepisaliśmy, aby, jeżeli natrafimy na kolejny element o tej samej wartości, został on wpisany o jedno miejsce wcześniej.

17-19 Teraz pozostało tylko przepisać zawartość posortowanej tablicy sorted[] do oryginalnej tablicy, którą mieliśmy posortować (w naszym przypadku ).

WAŻNE CECHY ALGORYTMU

* *countingSort* potrzebuje dodatkowej pamięci do posortowania tablicy – nie działa „w miejscu”
* jest wykorzystywany jako algorytm pomocniczy w innych metodach sortowania, np. radix sort
* nie jest optymalny dla małej ilości liczb z dużego przedziału

ZŁOŻONOŚĆ CZASOWA ALGORYTMU

static void countingSort(int arr[], int max\_val) { O(max\_val + arr.length)

        int[] counts = new int[max\_val+1]; O(1)

        for(int i = 0; i <= max\_val; i++) { O(max\_val)

            counts[i] = 0; O(1)

        }

        for(int i = 0; i < arr.length; i++) { O(arr.length)

            counts[arr[i]]++; O(1)

        }

        for(int i = 1; i <= max\_val; i++) { O(max\_val)

            counts[i] += counts[i-1]; O(1)

        }

        int[] sorted = new int[arr.length]; O(1)

        for(int i = arr.length - 1; i >= 0; i--) { O(arr.length)

            sorted[counts[arr[i]] - 1] = arr[i]; O(1)

            counts[arr[i]]--; O(1)

        }

        for(int i = 0; i < arr.length; i++) { O(arr.length)

            arr[i] = sorted[i]; O(1)

        }

    }

Jeżeli przyjmiemy n jako długość wejściowej tablicy i k jako ustaloną ilość kluczy, które będziemy zliczać w wejściowej tablicy to złożoność czasowa algorytmu sortowania przez zliczanie jest równa O(n+k), co w praktyce oznacza, że złożoność ta jest liniowa względem większej z wartości n i k.