SORTOWANIE PRZEZ SCALANIE | JAVA

WSTĘP

Sortowanie przez scalanie jest jednym z prostszych rekurencyjnych algorytmów sortowania. W skrócie jego działanie polega na podzieleniu tablicy na dwie części, posortowanie każdej z nich a następnie ponowne scalenie tych tablic w jedną posortowaną całość. Główną zaletą tej metody sortowania jest jej złożoność czasowa – O(nlogn), która jest znacznie korzystniejsza dla dużej ilości danych do posortowania niż np. w przypadku sortowania przez wstawianie – O(n2).

Algorytm sortowania przez scalanie jest popularny przede wszystkim dzięki łatwości implementacji oraz swojej całkiem niezłej wydajności.

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 scalanie, którego implementacja znajduje się (w liniach 15-50 przesłanego pliku MergeSort.java) pod postacią dwóch funkcji *mergeSort* oraz *merge*.

*UWAGA: Dla uproszczenia założyłem w implementacji algorytmu, że działamy na typie całkowitoliczbowym, jednak warto zaznaczyć, że zarówno algorytm jak i cały program działałby analogicznie i równie poprawnie dla dowolnego innego porównywalnego typu.*

FUNKCJA MERGESORT

    static void mergeSort(int arr[], int l, int r) {

        if(l < r) {

            int q = (l+r)/2;

            mergeSort(arr, l, q);

            mergeSort(arr, q+1, r);

            merge(arr, l, q, r);

        }

}

Funkcja *mergeSort* przyjmuje 3 argumenty: tablicę arr[] oraz 2 liczby naturalne l i r. Zadaniem tej funkcji jest posortowanie podtablicy arr[] rozpoczynającej się na indeksie l a kończącej na indeksie r. Pierwsze jawne wywołanie przez nas tej funkcji najczęściej ma na celu posortowanie całej tablicy i odbywa się w następujący sposób (dla danej tablicy int A[]):

mergeSort(A, 0, A.length-1);

Po otrzymaniu argumentów powyższa funkcja sprawdza czy aby na pewno dostała do posortowania tablicę o co najmniej dwóch elementach (l < r). W innym przypadku funkcja po prostu nie robi nic. Warunek ten jest konieczny w celu wyjścia z rekurencji, o czym przekonamy się za chwilę. Jeżeli jednak faktycznie mamy do czynienia z co najmniej dwu-elementową tablicą zostanie wyznaczony jej środek (q), a następnie *mergeSort* zostaje rekurencyjnie wykonany kolejno dla pierwszej oraz drugiej podtablicy (w przypadku nieparzystej liczby elementu ta pierwsza będzie dłuższa o 1). Ostatnim krokiem jest scalenie obu już posortowanych tablic w jedną poprzez wywołanie funkcji *merge*.

FUNKCJA MERGE

    static void merge(int arr[], int l, int q, int r) {

        int sizeL = q - l + 1;

        int sizeR = r - q;

        int L[] = new int[sizeL+1];

        int R[] = new int[sizeR+1];

        for(int i = 0; i < sizeL; i++) {

            L[i] = arr[l+i];

        }

        for(int i = 0; i < sizeR; i++) {

            R[i] = arr[q+i+1];

        }

        L[sizeL] = R[sizeR] = Integer.MAX\_VALUE;

        for(int i = 0, j = 0, k = l; k <= r; k++) {

            if(L[i] <= R[j]) {

                arr[k] = L[i];

                i++;

            }

            else {

                arr[k] = R[j];

                j++;

            }

        }

    }

Funkcja *merge* jest odpowiedzialna za scalenie dwóch posortowanych tablic w posortowaną całość. Patrząc na listę argumentów przekazywanych tej funkcji możemy zauważyć, że pierwsza z posortowanych tablic do scalenia zaczyna się od indeksu l, a kończy na q, natomiast druga zaczyna na q+1 i kończy na r. Mając to na uwadze tworzymy 2 pomocnicze tablice (lewą – L[]   
oraz prawą – R[]) o odpowiedniej liczbie elementów powiększonej o 1. Ma to na celu po wypełnieniu obu tablic odpowiednimi wartościami dopisanie na ostatnim miejscu największej możliwej wartości (takiej, aby wszystkie wartości w przekazanej w argumencie tablicy były od niej mniejsze).

Kolejnym krokiem jest wejście do pętli for, która wybiera odpowiedni element i przepisuje go do tablicy arr[] na miejsce o indeksie k. Do tego celu potrzeba iteratorów i oraz j, które przechodzą odpowiednio po tablicy L[] oraz R[]. Za każdym razem wybieramy mniejszy z elementów – L[i] lub R[j], a następnie zwiększamy odpowiadający iterator i przechodzimy do kolejnego przebiegu pętli. W tym miejscu ukazuję się powód wpisania na ostatnie miejsca obu tablic nieosiągalnie dużej wartości – gdy dochodzimy do końca jednej tablicy, zostają przepisywane pozostałe elementy drugiej tablicy.

*UWAGA: Istnieje wiele innych sposobów implementacji funkcji scalającej. Dla przykładu jednym z równie popularnych metod jest zastąpienie ustawiania ostatnich elementów obu tablic na nieskończone wartości dołożeniem dodatkowych pętli. W takim sposobie główna pętla przepisująca wartości do arr[] kończy swoje działanie w momencie napotkania na koniec jednej z obu tablic, a następnie zostają wykonane dodatkowe pętle, które przepiszą pozostałe w tablicy L[] lub R[] elementy.*

ZŁOŻONOŚĆ CZASOWA ALGORYTMU

Złożoność czasowa funkcji *merge* jest liniowa, ponieważ mamy do czynienia z 3 pętlami   
o złożoności O(n) (a wszystkie operacje w tych pętlach zajmują czas stały) oraz operacjami o czasie stałym. Jeżeli założymy, że *mergeSort* zajmuje T(n) czasu otrzymamy równanie opisane rekurencyjnie (odpowiada wywołaniu *mergeSorta* na obu połowach tablicy):

Rozwiązując powyższe równanie otrzymujemy wynik T(n) = O(nlogn). Ta równość nie zależy od ułożenia danych wejściowych, tzn. czy tablica jest jakkolwiek posortowana czy nie. Dlatego też zarówno dla przypadku pesymistycznego, średniego jak i optymistycznego złożoność czasowa algorytmu sortowania przez scalanie jest równa O(nlogn).

WAŻNE CECHY ALGORYTMU

Ważną cechą sortowania przez scalanie jest fakt, że nie sortuje on w miejscu – potrzebna jest pomocnicza pamięć do wykonania operacji scalania.

Algorytm ten sortuje stabilnie – kolejność elementów o równych wartościach nie zostaje zmieniona.