QUICKSORT | JAVA

WSTĘP

Sortowanie szybkie (quicksort) jest jednym z najbardziej wydajnych i popularnych algorytmów sortowania w dzisiejszych czasach. Quicksort jest rekurencyjnym algorytmem, który działa zgodnie ze strategią „dziel i zwyciężaj”.

METODA PARTITION

Sercem sortowania szybkiego jest dodatkowa metoda *partition.* W skrócie przetasowuje ona elementy tablicy tak, aby powstały kolejno od lewej 3 części – lewa podtablica, pivot,   
prawa podtablica. Pivot jest jedną, wybraną wcześniej wartością z tablicy. *partition* sprawia, że wszystkie wartości w lewej podtablicy są mniejsze od pivota, a z prawej podtablicy odpowiednio większe od pivota. Istnieje bardzo wiele metod implementacji funkcji *partition.* Mogą się one różnić wyborem pivota (co samo w sobie daje ogromną gamę możliwości) oraz sposobem odpowiedniego „układania” elementów względem niego. Ja w swoim kodzie zdecydowałem się na napisanie partition zgodnie z popularną metodą partycjonowania Lomuto.

QUICKSORT - IMPLEMENTACJA W JAVIE

void quickSort(int A[], int p, int r) {

        if(p < r) {

            int q = partition(A, p, r);

            quickSort(A, p, q-1);

            quickSort(A, q+1, r);

        }

    }

Metoda *quickSort* ma na celu posortowanie tablicy. Jeżeli dostanie na wejściu niepustą,   
ponad 1-elementową tablicę wykorzystuje ona funkcję partycjonującą, wywołując *partition* na tablicy oraz przypisując zwróconą wartość do zmiennej q. Następnie na lewej podtablicy od początku do elementu q-1 oraz na prawej pod tablicy od elementu q+1 do końca zostaje rekurencyjnie wywołany *quickSort.*

PARTITION (LOMUTO) – IMPLEMENTACJA W JAVIE

int partition(int A[], int p, int r) {

        int pivot = A[r];

        int i = p-1;

        for(int j = p; j < r; j++)

        {

            if(A[j] <= pivot)

            {

                i++;

                int t = A[i];

                A[i] = A[j];

                A[j] = t;

            }

        }

        A[r] = A[i+1];

        A[i+1] = pivot;

        return i+1;

    }

Pierwszym krokiem jest wybranie pivota – w naszym przypadku jest to ostatni element w tablicy. Później przechodzimy po wszystkich elementach części tablicy A, którą mamy partycjonować. W tej pętli korzystamy z dodatkowej zmiennej (int i), która przechowuje nam indeks ostatniego elementu mniejszego lub równego pivotowi. W pętli, jeżeli natrafimy na element nie większy od pivota, zwiększamy i o 1 oraz dokonujemy zamiany tego elementu z elementem A[i].   
Po zakończeniu pętli for pozostaje tylko zamienić i+1-szy element z naszym pivotem, dzięki czemu uzyskujemy końcowy, oczekiwany efekt.

ZŁOŻONOŚĆ CZASOWA ALGORYTMU

* **Przypadek pesymistyczny** występuje, gdy funkcja *partition* za każdym razem wybiera na pivota najmniejszą lub największą wartość w tablicy. W takiej sytuacji będzie ona wywoływana n-1 razy za każdym razem dla tablicy o mniejszej o 1 liczbie elementów. Daje nam to złożoność **O(n2)**.
* Z **przypadkiem optymistycznym** mamy do czynienia, kiedy z każdym wywołaniem *partition* tablica zostaje podzielona na dwie, równe części. Oznacza to, że *quickSort* zostanie uruchomiony log2n razy, z czego implikuje złożoność **O(nlogn)** dla przypadku optymistycznego.
* W **średnim przypadku** *quickSort* również cechuje się złożonością czasową **O(nlogn)**.

PODSUMOWANIE

Quicksort pomimo złożoności czasowej O(n2) w pesymistycznym przypadku jest jednym z najbardziej wydajnych algorytmów sortowania dzięki sprzyjającej złożoności przypadku średniego. Sortowanie szybkie sortuje rekurencyjnie, w miejscu, ale nie stabilnie.