**Analiza eksperymentalna nowych algorytmów sortowania w miejscu**

1. **Wstęp**

W mojej pracy badałem hybrydowe algorytmy sortujące, będące połączeniem wielu technik sortujących.

Algorytmy były analizowane w dwóch grupach

* 1. **Algorytmy deterministyczne** – podczas partycjonowania piwot jest wybierany w sposób deterministyczny
  2. **Algorytmy randomizowane** – piwot jest wybierany w sposób losowy

1. **Motywacja tworzenia algorytmów hybrydowych**

Zasadniczo głównym powodem tworzenia hybrydowych algorytmów sortujących jest poprawa wydajności algorytmów podstawowych. Łączymy algorytmy, aby efektywnie wykorzystać najlepsze części poszczególnych algorytmów oraz niwelować wady każdego z nich. Czasami wprowadzając nieznaczną modyfikację, jak na przykład wybór piwota jako mediana z trzech elementów zyskujemy poważną optymalizację

* 1. **Poprawa wydajności** – minimalna liczba wykonywanych operacji
  2. **Optymalizacja przypadku pesymistycznego** – logarytmiczno-liniowe działanie algorytmu, niezależnie od porządku danych wejściowych
  3. **Działanie w miejscu** – brak potrzeby alokacji dodatkowej pamięci, czas dostępu do pamięci, alokacji i dealokacji może być kosztowny,
  4. **Minimalizacja liczby porównań** – w praktycznych zastosowaniach sortujemy złożone struktury, dla których pojedyncza operacja porównania jest znacznie kosztowniejsza niż np. operacja przypisania

1. **Quick Sort**

Zanim przejdę do analizy algorytmów hybrydowych, chciałbym po krótce przedstawić problemy z jakimi musimy się zmierzyć stosując standardowe algorytmy sortujące.

* 1. **O(n2) czas działania dla przypadku pesymistycznego** – dla uporządkowanych danych wejściowych nie sprawdza się technika dziel i zwyciężaj wykorzystywana w algorytmie Quick Sort, głębokość drzewa wywołań rekurencyjnych jest asymptotycznie równa długości tablicy, sam algorytm Quick Sort działa w czasie kwadratowym
  2. **Duża liczba porównań** – algorytm wykonuje prawie 2 razy więcej operacji porównania w stosunku do operacji przypisania, problem w przypadku sortowania złożonych struktur danych

1. **Merge Sort**
   1. **Konieczność posiadania dodatkowej pamięci o wielkości O(n)**– Merge Sort potrzebuje dodatkowego bufora, istnieją implementacje dla których potrzebujemy n/2, nie wszystkie systemy posiadają zasoby aby móc to wykonać
   2. **Dodatkowy nakład czasowy związany z alokacją oraz zwalnianiem pamięci** – operacje alokacji i zwalniania miejsca na stercie są kosztowne
2. **Algorytmy z rodziny Quick Sort**

Tak jak wcześniej wspomniałem, czasami stosunkowo niewielka zmiana może znacząco poprawić wydajność algorytmów sortujących. Moją analizę rozpocząłem od badania wydajność różnych wersji algorytmu Quick Sort pod kątem optymalizacji dla przypadku pesymistycznego, z podziałem na metodę partycjonowania oraz strategię wyboru piwota.

* 1. **Partycjonowanie Hoare, losowy piwot** - najwydajniejszą z badanych metod jest partycjonowanie metodą Hoare przy wyborze piwota jako losowy element, dla uporządkowanych danych, czyli przypadku pesymistycznego w przypadku klasycznego Quick Sort, zmodyfikowany algorytm działa w czasie liniowo-logarytmicznym
  2. **Partycjonowanie Hoare, mediana z trzech** – w przypadku algorytmów deterministycznych, najwydajniejsze jest partycjonowanie metodą Hoare oraz wybór piwota jako mediana z trzech elementów, np. pierwszy, środkowy oraz ostatni element tablicy.

1. **QuickMerge Sort – rysunek**

Kolejnym badanym algorytmem był QuickMerge Sort.

* 1. **Sebastian Wild –** w swojej pracy zatytułowanej **QuickXsort – A Fast Sorting Scheme in Theory and Practice** autor przedstawił koncepcję połączenia

QuickMerge Sort to algorytm Hybrydowy, będący połączeniem algorytmów Quick Sort oraz Merge Sort.

Algorytm rozwiązuje problem z alokacją pamięci dla typowego algorytmu Merge Sort.

Celem tego algorytmu jest minimalizacja wykonywanych operacji porównania, przy równoczesnym zachowaniu działania w miejscu. Algorytm QuickMerge Sort to algorytm rekurencyjny składajacy się z trzech kroków

1. **Partycjonowanie** – na początku partycjonowana jest tablica wejściowa, przy użyciu metod deterministycznych lub niedeterministycznych
2. **Sortowanie mniejszej części algorytmem Merge Sort** – przy czym bufor stanowi druga część tablicy
3. **Sortowanie drugiej części** – rekurencyjne wywołanie algorytmu dla pozostałej podtablicy
4. **QuickMerge Sort – pseudokod**

Tutaj mamy pseudokod algorytmu QuickMerge Sort. Należy pamiętać, że po dokonaniu partycji, algorytm MergeSort stosujemy do mniejszej z otrzymanych podtablic. Dodatkowo, ponieważ bufor pamięci nie może zostać nadlisany, zamiast przenosić elementy, zamieniamy je miejscami stosując metodę swap.

QuickMerge Sort to algorytm rekurencyjny, składający się z trzech kroków.

1. **QuickMerge Sort – wykres**

Ponieważ składową częścią tego algorytmu jest partycjonowanie ciągu wejściowego, algorytm ten również był analizowany z podziałem na techniki deterministyczne oraz randomizowane. Algorytmy badałem pod kątem całkowitego kosztu wykonanych operacji na podstawie współczynnika kosztu, współczynnik kosztu określa ile razy operacja porównania jest droższa od operacji przypisania. Ma to sens, jeśli np. sortujemy złożone struktury, oraz operujemy na wskaźnikach do tych struktur. Wtedy przypisanie to podmiana wskaźników, zaś porównanie to dereferencja wszystkich składowych struktury.

Z przeprowadzonej analizy wynika, że w przypadku sortowania złożonych struktur danych, tzn. algorytm QuickMerge Sort jest znacznie wydajniejszy niż klasyczny algorytm Quick Sort. Najwydajniejszą kombinację otrzymano łącząc algorytm QuickMerge Sort z partycjonowaniem Lemuto przy wyborze piwota jako pseudo-mediana z trzech elementów lub mediana z trzech.

1. **Intro Sort – pseudokod**

Intro Sort, czyli sortowanie introspektywne, to algorytm hybrydowy będący połączeniem algorytmów

Quick Sort, Insertion Sort oraz Heap Sort. Działanie algorytmu dostosowuje się w zależności od rozmiaru

sortowanej tablicy oraz głębokości drzewa wywołań rekurencyjnych, oraz zasadniczo składa się z trzech niezależnych części. Najlepiej jest to widoczne w pseudokodzie.

* 1. **Insertion Sort** – jeśli długość tablicy jest mniejsza niż stała wartość wyznaczona eksperymentalnie, stosowany jest algorytm Insertion Sort, w mojej pracy wyznaczyłem tę stałą jako **20**
  2. **Heap Sort** – jeśli drzewo wywołań rekurencyjnych jest głębsze niż **2log(n)**, to rekurencja jest przerywana i stosowany jest algorytm Heap Sort. Dzięki temu, gdy na wejściu są uporządkowanedane, algorytm nie działa z czasem O(n2) lecz z czasem liniowo-logarytmicznym.
  3. **Quick Sort** – w pozostałych przypadkach stosowany jest algorytm Quick Sort

1. **Intro Sort – wykres**

Algorytm był analizowany pod kątem liczby wykonanych operacji w porównaniu do algorytmu Quick Sort.

W każdym z badanych przypadków algorytm Intro Sort okazał się wydajniejszy od algorytmu Quick Sort.

Najefektywniejszym algorytmem sortującym jest Intro Sort z partycjonowaniem metodą Hoare oraz wyborem piwota jako pseudo-mediana z dziewięciu elementów lub mediana z trzech.

1. **Zakończenie**

Dziękuję za uwagę.