Analiza algorytmów sortujących

Tomasz Warzecha

1 Wstęp

W sprawozdaniu moim celem jest analiza i porównanie wydajności trzech głównych algorytmów sortowania wraz z ich modyfikacjami. Insertion sort, merge sort, heap sort zostały zaimplementowane, tak aby przeprowadzić na nich różne testy wydajności. Badania te skupiają się na generowaniu losowych danych, które następnie są używane do obliczenia czasu wykonania algorytmów oraz liczby operacji takich jak przypisania i porównania. Wyniki tych testów mają na celu porównanie algorytmów, a także pokazanie różnic między teoretycznymi, a praktycznymi wynikami.

2 Implementacja algorytmów

Algorytmy zostały zaimplementowane zgodnie z klasycznymi wersjami algorytmów sortujących, korzystając z pseudokodów zamieszczonych na stronie przez prof. Szymona Żeberskiego. Wprowadzone zostały także modyfikacje, które delikatnie zmieniają działanie algorytmów. Algorytmy zostały zaimplementowane dla tablic z liczbami typu `float`.

Najciekawsze fragmenty kodu

Poniżej przedstawię najciekawsze fragmenty kodu, wraz z objaśnieniami, tak aby łatwiej zrozumieć działanie algorytmów.

- **Insertion Sort:** Algorytm sortowania przez wstawianie polega na wybieraniu kolejnych elementów z nieposortowanej części tablicy i wstawianiu ich w odpowiednie miejsce wśród już posortowanych elementów.

```
for(int i = 1; i < n; i++) {
    float key = A[i];
    int j = i - 1;
    while(j >= 0 && A[j] > key) {
        A[j + 1] = A[j];
        j = j - 1;
    }
    A[j + 1] = key;
}
```

W powyższym kodzie zmienna `key` przechowuje aktualnie wybrany element do wstawienia. Pętla `while` przesuwa większe elementy o jedno miejsce w prawo, aż znajdzie odpowiednią pozycję dla elementu `key`.

- **Merge Sort:** Sortowanie przez scalanie to metoda, która dzieli tablicę na dwie części, sortuje je osobno, a następnie scala w jedną posortowaną całość.

```
if (L[i] <= R[j]) {
    A[l] = L[i];
    i++;
} else {
    A[l] = R[j];
    j++;
}
```

Tutaj, podczas scalania dwóch posortowanych podtablic `L` i `R`, wybierany jest mniejszy element i dodawany do głównej tablicy `A`. W ten sposób elementy są umieszczane we właściwej kolejności.

- **Heap Sort:** Sortowanie przez kopcowanie wykorzystuje strukturę danych zwaną kopcem, aby organizować elementy w taki sposób, że największy lub najmniejszy element można łatwo usunąć i umieścić na końcu tablicy.

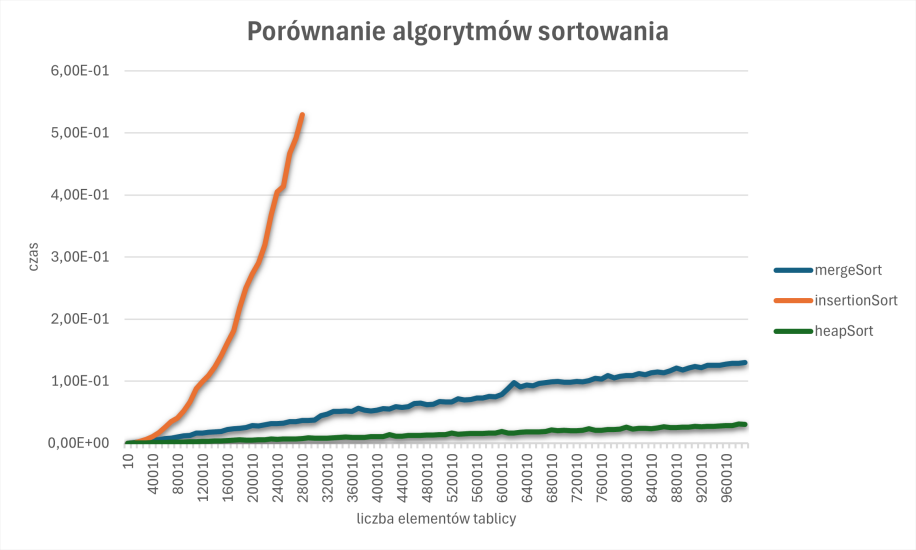
```
void heapify(float A[], int n, int i) {
    int largest;
    int l = left(i);
    int r = right(i);

    if (l < n && A[l] > A[i]) {
        largest = l;
    } else {
        largest = i;
    }

    if (r < n && A[r] > A[largest]) {
        largest = r;
    }

    if (i != largest) {
        float temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        heapify(A, n, largest);
    }
}
```

Funkcja `heapify` zapewnia, że poddrzewo naszego kopca jest poprawnie ułożone - wyszukuje największy element spośród korzenia i jego dzieci, a następnie zamienia elementy, jeśli korzeń nie jest największy.

3 Porównanie algorytmów

3.1 Porównanie czasów działania

Porównanie czasów wykonania algorytmów na różnych długościach tablic: od 10 do 100010. Liczby zostały przydzielone losowo z zakresu (-1000;1000). Dane zostały przedstawione na wykresie poniżej.



Z wykresu możemy odczytać, że czas wykonania algorytmu *insertion sort* jest znacząco większy od *heap sort* oraz *merge sort*. Jest to spowodowane faktem, że jego złożoność obliczeniowa wynosi $O(n^2)$, a nie $O(n \log n)$, jak w przypadku pozostałych algorytmów. Algorytm *merge sort* jest porównywalnie czasochłonny co *heap sort*; różnica widoczna na wykresie może wynikać z faktu, że w sortowaniu przez scalanie tworzymy pomocnicze tablice, zamiast operować na jednej, jak w przypadku sortowania przez kopcowanie.

Analiza liczby przypisań i porównań

W tabeli poniżej przedstawiono średnią ilość porównań i przypisań podczas wykonywania każdego sortowania, dla danego rozmiaru tablicy

Rozmiar	**Insertion Sort**	**Merge Sort**	**Heap Sort**
100	porównania: 5060 przypisania: 5258	porównania: 771 przypisania: 3105	porównania: 3175 przypisania: 4509
1000	porównania: 526996 przypisania: 528994	porównania: 10975 przypisania: 40917	porównania: 48090 przypisania: 69749
10000	porównania: 50457072 przypisania: 50477070	porównania: 143615 przypisania: 510837	porównania: 645690 przypisania: 944142

Table 1: przypisania i porównania użyte w każdym sortowaniu

Algorytm *Insertion Sort* wykazuje najwyższą liczbę porównań i przypisań, szczególnie dla większych tablic, co wynika z jego złożoności $O(n^2)$. Z kolei *Merge Sort* i *Heap Sort*, obie o złożoności $O(n \log n)$, wykonują znacznie mniej operacji, co czyni je bardziej wydajnymi przy większych rozmiarach danych. *Merge Sort* ma nieco niższą liczbę operacji niż *Heap Sort*, co może się kłócić z danymi z pierwszego wykresu, natomiast może być to spowodowane faktem, że większym obciążeniem dla komputera jest przechowywanie tablic w pamięci niż wykonywanie operacji.

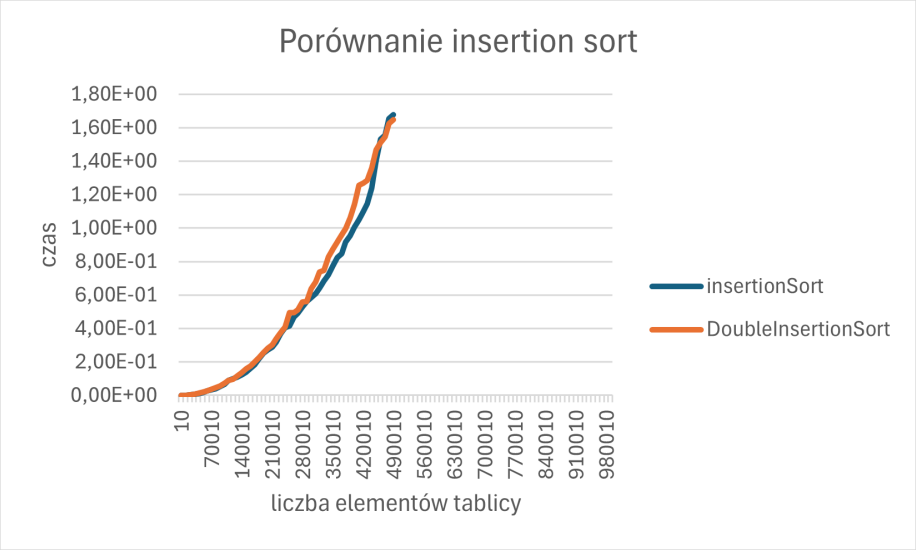
4 Modyfikacje algorytmów

Sortowanie przez wstawianie (podwójne)

Pierwszym zmodyfikowanym algorytmem jest insertion sort. Zmiana polega na tym, że zamiast jednego elementu bierzemy dwa, porównujemy je ze sobą, a następnie za pomocą pętli while szukamy miejsca na wstawienie dla mniejszego elementu i powtarzamy to samo dla większego elementu. Musimy dodac także obsługę parzystych tablic gdzie ostatni element sortujemy tak jak w standardowej wersji.

```
for (int i = 1; i < n - 1; i += 2) {
    float key1, key2;
    if (A[i] < A[i + 1]) {
        key1 = A[i];
        key2 = A[i + 1];
    } else {
        key1 = A[i + 1];
        key2 = A[i];
    }

    // sortowanie mniejszego klucza
    int j = i - 1;
    while (j >= 0 && A[j] > key1) {
        A[j + 1] = A[j];
        j--;
    }
    A[j + 1] = key1;

    //sortowanie wiekszego klucza
    int j2 = i;
    while (j2 > j && A[j2] > key2) {
        A[j2 + 1] = A[j2];
        j2--;
    }
    A[j2 + 1] = key2;
}
```

Niestety nasza modyfikacja nie dała nam lepszych rezultatów, co możemy zobaczyć na poniższym wykresie:

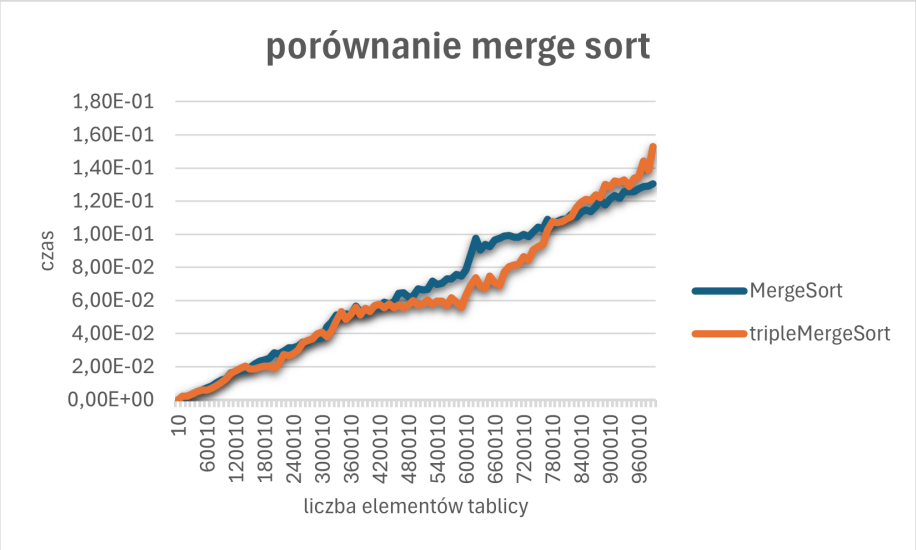


Potrójne sortowanie przez scalanie

W tej modyfikacji zamiast dzielić naszą tablice na dwie części, dzielimy na trzy, następnie porównujemy ze sobą elementy z trzech tablic i scalamy tak aby tworzyły coraz większe posortowane tablice.

```
for (i = 0; i < n1; i++) {
    L[i] = A[i + p];
}
for (j = 0; j < n2; j++) {
    S[j] = A[j + s1 + 1];
}
for (h = 0; h < n3; h++) {
    R[h] = A[h + s2 + 1];
}

i = 0;
j = 0;
h = 0;

for (int l = p; l <= k; l++) {
    if (L[i] <= S[j] && L[i] <= R[h]) {
        A[l] = L[i];
        i++;
    } else if (S[j] <= L[i] && S[j] <= R[h]) {
        A[l] = S[j];
        j++;
    } else {
        A[l] = R[h];
        h++;
    }
}
```

Tutaj również nasza modyfikacja nie dała lepszych rezultatów, czas działania jest mniej więcej taki sam:



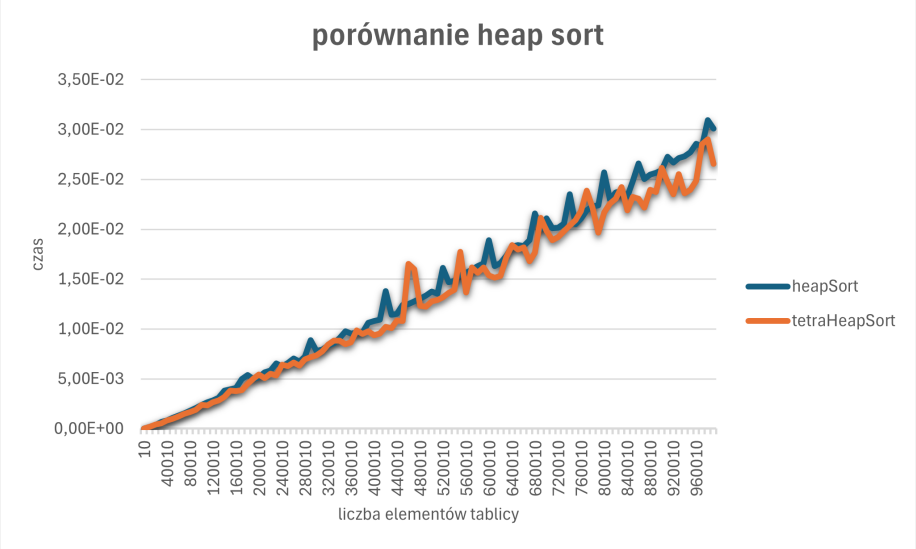
Sortowanie przez kopcowanie ternarnie

Tutaj chcemy przede wszystkim użyć drzewka terarnego zamiast binarnego. Tworzymy takie drzewko, a następnie porównujemy korzeń z trzema dziećmi, tak aby na miejscu korzenia znalazł się największy element. Po każdym przejściu, tak jak w standardowej wersji, największy element przesuwamy na koniec gdzie tworzy się posortowana tablica.

```
void heapifyT(float A[], int n, int i) {
    int largest;
    int l = leftT(i);
    int m = middleT(i);
    int r = rightT(i);

    if (l < n && A[l] > A[i]) {
        largest = l;
    } else {
        largest = i;
    }

    if (m < n && A[m] > A[largest]) {
        largest = m;
    }

    if (r < n && A[r] > A[largest]) {
        largest = r;
    }

    if (i != largest) {
        float temp = A[i];
```

```
23          A[i] = A[largest];
24          A[largest] = temp;
25          heapifyT(A, n, largest);
26      }
27  }
```

W tym przypadku zwyczajny heap sort radzi sobie delikatnie gorzej co widać na dużych tablicach, jednak nie jest to znacząca różnica.



5 Wnioski

Po przeprowadzeniu testów na trzech algorytmach sortujących (*Insertion Sort*, *Merge Sort* i *Heap Sort*), jak i na modyfikacjach, możemy wyciągnąć kilka wniosków.

- **Wydajność algorytmów:** *Insertion Sort* wypadł najgorzej, zwłaszcza przy większych zbiorach danych. Wynika to z tego, że ma on złożoność $O(n^2)$, przez co czas wykonania rośnie bardzo szybko w miarę wzrostu rozmiaru tablicy. Z kolei *Merge Sort* i *Heap Sort* mają złożoność $O(n \log n)$ i sprawdzają się dużo lepiej, szczególnie dla większych danych, *Merge Sort* wypada trochę gorzej niż *Heap Sort*, mimo ze ma mniej operacji, jednak może to być zależne od parametrów komputera na którym to testujemy.
- **Porównania i przypisania:** Jak widać, *Insertion Sort* ma zdecydowanie najwięcej porównań i przypisań, zwłaszcza przy dużych tablicach. Na szczęście *Merge Sort* i *Heap Sort* wykonują ich mniej, co sprawia, że są po prostu bardziej wydajne przy dużych zbiorach danych.

- **Zmodyfikowane algorytmy:** Zmodyfikowane wersje algorytmów, takie jak podwójne sortowanie przez wstawianie, potrójne sortowanie przez scalanie i ternarne sortowanie przez kopcowanie, niestety nie dały lepszych rezultatów. Wydaje się, że nie zmieniając zasadniczo złożoności algorytmów, nie osiągnęliśmy poprawy, a czas wykonania pozostał mniej więcej taki sam jak w przypadku wersji standardowych. Zatem można powiedzieć, że najlepiej pozostać przy klasycznych wersjach algorytmów

Podsumowując Jeśli chodzi o wybór algorytmu do dużych zbiorów danych, *Merge Sort* i *Heap Sort* to zdecydowanie lepszy wybór niż *Insertion Sort*. Zmodyfikowane wersje algorytmów (np. z większą liczbą elementów do porównań) nie sprawdziły się i nie poprawiły znacząco wyników. Można powiedzieć, że proste algorytmy takie jak *Merge Sort* są po prostu wystarczająco dobre do większości zastosowań.