Sprawozdanie z listy 1

Maja Kołakowska

287360

Spis treści

0.1 Wstęp

W tej pracy zajmiemy się analizą i porównaniem wybranych algorytmów sortowania. Rozpatrzone zostaną klasyczne metody: sortowanie przez wstawianie, sortowanie przez scalanie („dziel i zwyciężaj") oraz sortowanie przez kopcowanie. Dla każdego z wymienionych algorytmów zostaną zaimplementowane ich zmodyfikowane wersje:

- Insertion Sort, który wstawia jednocześnie dwa elementy do tablicy,
- Merge Sort z podziałem tablicy na trzy części zamiast dwóch,
- Heap Sort wykorzystujący kopiec ternarny.

Wszystkie podstawowe algorytmy zostały zaimplementowane na podstawie pseudokodów wstawionych prez prof. Szymona Żeberskiego. Celem pracy jest porównanie efektywności podstawowych implementacji algorytmów z ich zmodyfikowanymi odpowiednikami. Analiza zostanie przeprowadzona z wykorzystaniem trzech wskaźników: czasu wykonania, liczby porównań oraz liczby przypisań.

Celem pracy jest porównanie efektywności podstawowych implementacji algorytmów z ich zmodyfikowanymi odpowiednikami. Analiza zostanie przeprowadzona z wykorzystaniem trzech wskaźników: czasu wykonania, liczby porównań oraz liczby przypisań.

0.2 Opis metody badawczej

Zajmiemy się trzema wskaźnikami: czasem, liczbą porównań oraz liczbą przypisań. Dla każdego algorytmu przeprowadzimy 100 testów, z których następnie wyciągniemy średnie wartości naszych wskaźników. Badanie przeprowadzimy na tablicach zawierających losowe liczby. Rozmiary tablic to: 5000, 10000, 20000, 40000, 60000, 80000, 100000 oraz 150000 elementów. Wielkości te zostały dobrane z myślą o najwolniejszym algorytmie. Dla Insertion Sorta te dane są „duże", natomiast dla Merge oraz Heap Sorta będą „małe" ze względu na ilość czasu potrzebną dla tych danych.

0.3 Insertion Sort

Insertion Sort jest jednym z prostszych algorytmów — zarówno pod względem zobrazowania (często porównywany do sortowania talii kart na ręce), jak

i implementacji kodu. Sortowanie przez wstawianie jest efektywnym algorytmem dla małych tablic. Jego złożoność czasowa wynosi $O(n^2)$, a w najlepszym wypadku $O(n)$.

Natomiast zmodyfikowany Insertion Sort bierze na raz dwa elementy, które najpierw porównujemy między sobą, a potem wstawiamy w odpowiednie miejsce w posortowanej tablicy. Redukuje to w ten sposób liczbę porównań i przypisań. Na Rysunku 1 przedstawiono fragment kodu implementującego tę modyfikację.

```
void insertionSortDouble(int arr[], int n) {
    if (n<=1) return;
    for (int i = 0; i < n; i+=2) {
        if (i+1<n) {
            if (arr[i] > arr[i+1]) {
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
            int maly = arr[i];
            int duzy = arr[i+1];
            int j = i-1;
            while ((j >= 0) && arr[j] > duzy) {
                    arr[j+2] = arr[j];
                    j = j-1;
                }
            arr[j+2] = duzy;
            while ((j >= 0) && arr[j] > maly) {
                    arr[j+1] = arr[j];
                    j = j-1;
                }
            arr[j+1] = maly;
        }
        else {
            int key = arr[i];
            int j = i-1;
            while ((j >= 0) && arr[j] > key) {
                    arr[j+1] = arr[j];
                    j = j-1;
            }
            arr[j+1] = key;
        }
    }
}
```

Rysunek 1: Fragment kodu zmodyfikowanego Insertion Sort

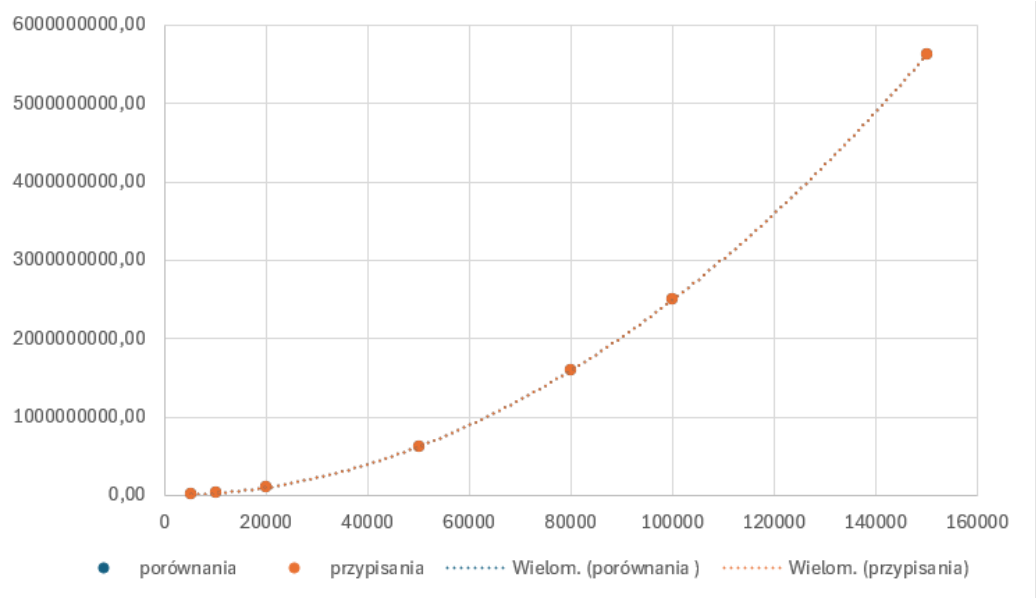
0.3.1 Pomiary dla sortowania przez wstawianie

Dla każdego z siedmiu podanych uprzednio rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, porównań i przypisań. Wyniki tych pomiarów prezentuje Tabela 1.

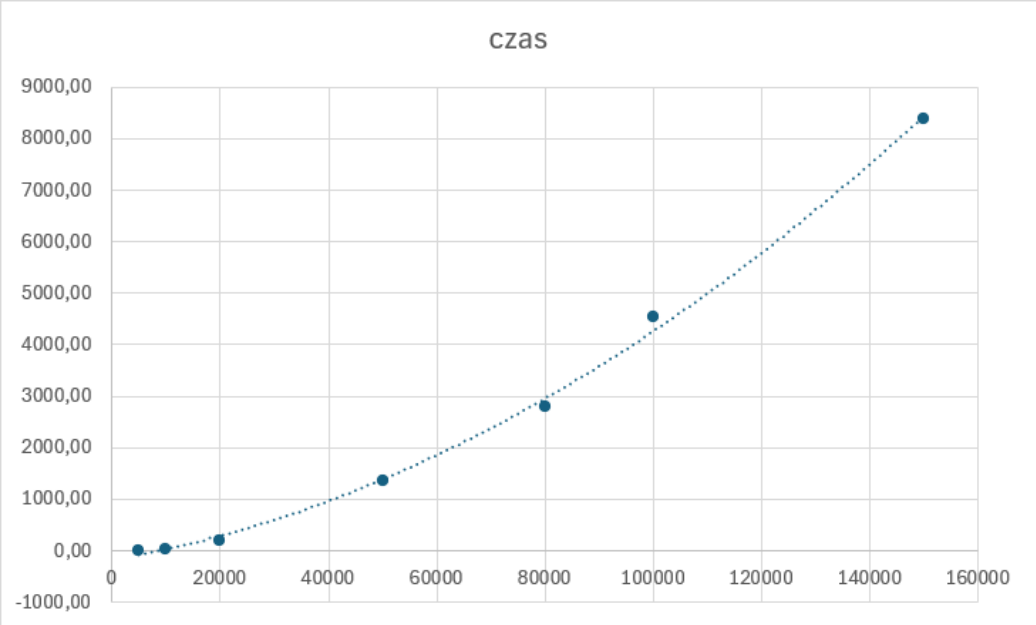
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	4 173 230.00	4 174 490.00	7.08
10 000	16 681 200.00	16 683 700.00	21.21
20 000	66 681 800.00	66 686 700.00	96.91
50 000	416 844 000.00	416 857 000.00	604.52
80 000	1 066 900 000.00	1 066 920 000.00	1 552.93
100 000	1 666 710 000.00	1 666 740 000.00	2 435.82
150 000	3 750 060 000.00	3 750 100 000.00	5 446.79

Tabela 1: Wyniki pomiarów dla Insertion Sort

Wyniki przedstawiono również na Rysunkach 2 i 3.



Rysunek 2: Wykres przypisań i porównań dla Insertion Sort



Rysunek 3: Wykres czasu wykonania dla Insertion Sort

Oba wykresy mają wyznaczone linie trendu wielomianowe. Dla liczby porównań i przypisań linia ta opisana jest równaniem: $y = 0{,}2503x^2 - 30{,}269x + 327459$, natomiast dla czasu wykonania – równaniem: $y = 3 \times 10^{-7}x^2 + 0{,}0182x - 179{,}09$. Możemy więc ograniczyć tę funkcję od góry inną funkcją kwadratową. Oznacza to, że sortowanie ma złożoność kwadratową, zgodnie z naszymi założeniami. W taki sam sposób rosną zarówno liczba porównań, jak i liczba przypisań. Możemy zatem uznać, że algorytm ten sortuje w zadowalającym czasie jedynie tablice o niewielkiej liczbie elementów.

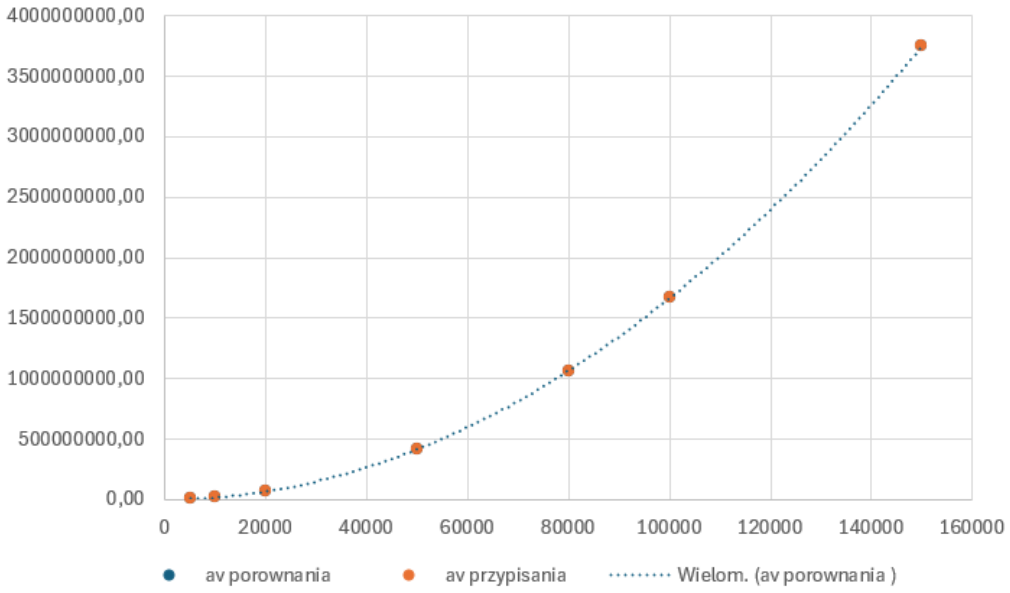
0.3.2 Pomiary dla sortowania przez wstawianie z modyfikacją

Dla każdego z siedmiu podanych uprzednio rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, porównań i przypisań. Wyniki tych pomiarów prezentuje Tabela 2.

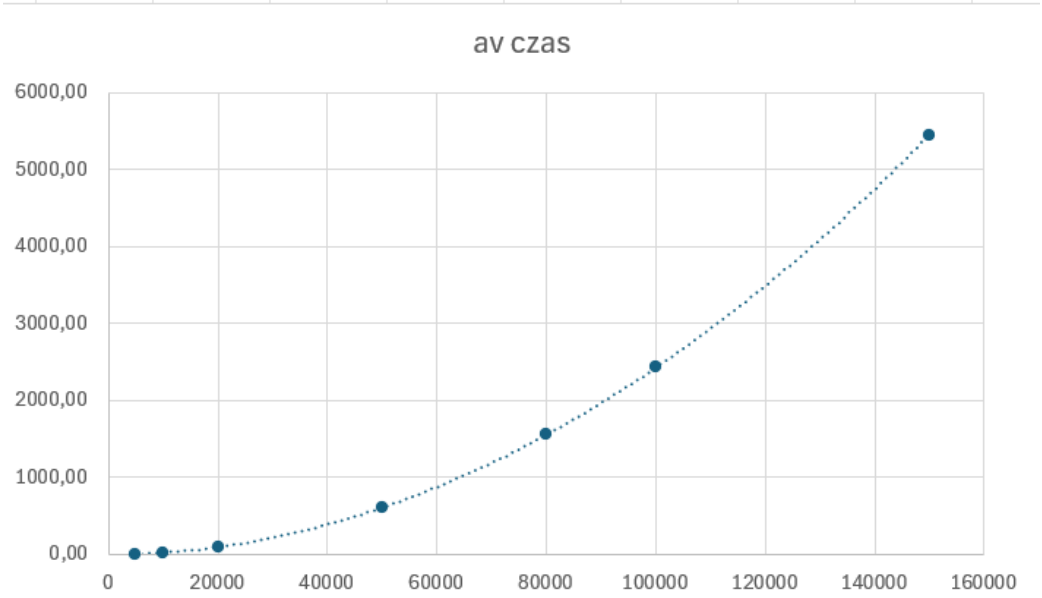
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	4 173 230.00	4 174 490.00	7.08
10 000	16 681 200.00	16 683 700.00	21.21
20 000	66 681 800.00	66 686 700.00	96.91
50 000	416 844 000.00	416 857 000.00	604.52
80 000	1 066 900 000.00	1 066 920 000.00	1 552.93
100 000	1 666 710 000.00	1 666 740 000.00	2 435.82
150 000	3 750 060 000.00	3 750 100 000.00	5 446.79

Tabela 2: Wyniki pomiarów dla Insertion Sort z modyfikacjami

Wyniki przedstawiono również na Rysunkach 3 i 4.



Rysunek 4: Wykres przypisań i porównań dla Insertion Sort zmodyfikowanego

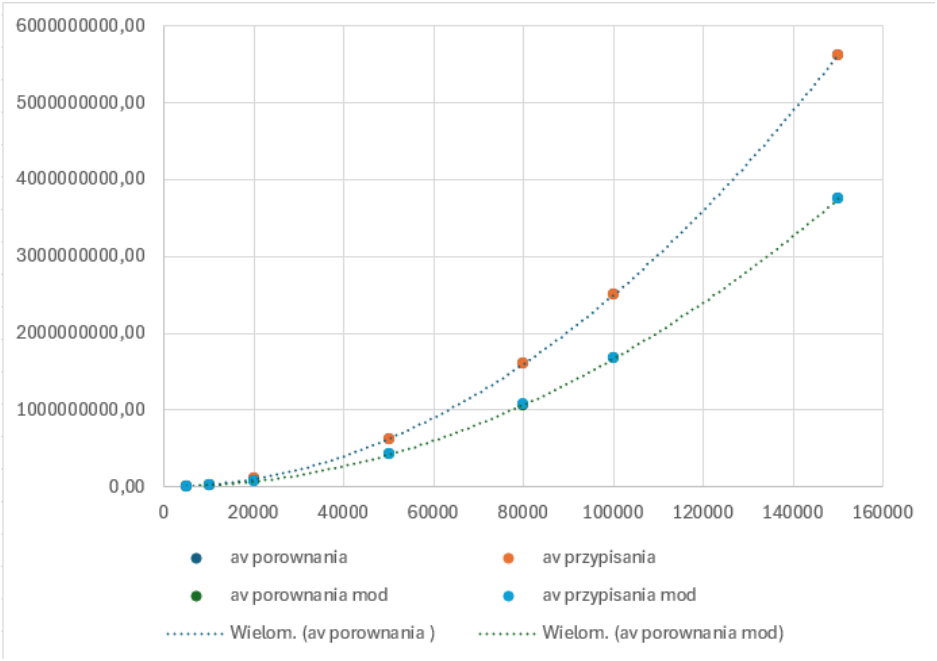


Rysunek 5: Wykres czasu wykonania dla Insertion Sort zmodyfikowanego

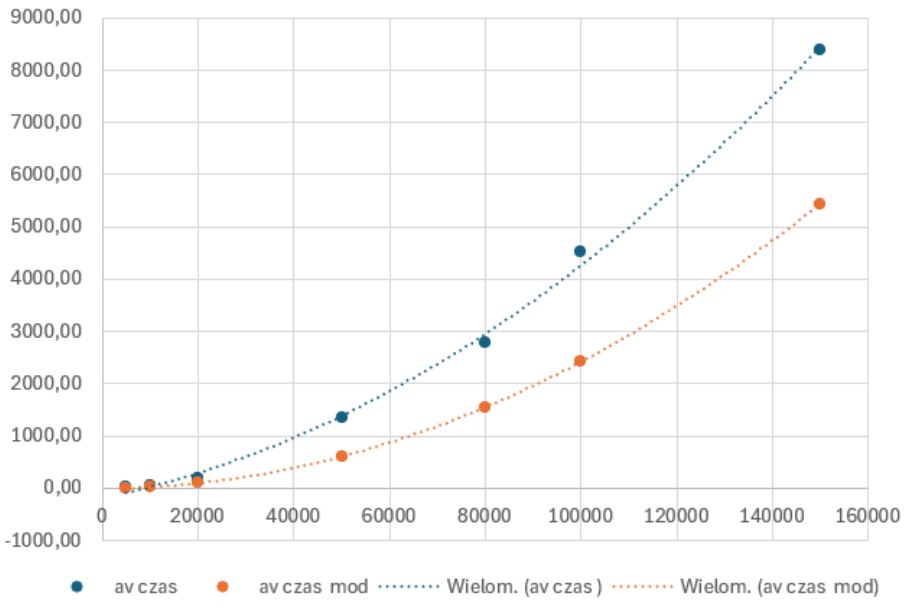
Podobnie jak w poprzednim przypadku, oba wykresy mają wyznaczone linie trendu wielomianowe. Dla liczby porównań i przypisań linia ta opisana jest równaniem: $y = 0{,}1666x^2 + 4{,}4488x - 23737$, natomiast dla czasu wykonania: $y = 2 \times 10^{-7}x^2 + 2 \times 10^{-4}x - 3{,}7455$.

0.3.3 Porównanie algorytmów i wnioski

Na Rysunkach 5 oraz 6 przedstawiono wykresy zbiorcze odpowiadające punktom 0.3.1 oraz 0.3.2. Z przedstawionych wyników wynika, że zarówno pod względem liczby porównań, przypisań, jak i czasu wykonania, zmodyfikowana wersja algorytmu jest zdecydowanie bardziej wydajna. Przykładowo, dla $n = 1\,000\,000$ klasyczny insertion sort potrzebuje około 5,3 minuty, natomiast jego zmodyfikowana wersja jedynie 3,37 minuty. Nie zmienia to jednak faktu, że oba algorytmy nie nadają się do sortowania dużych tablic.



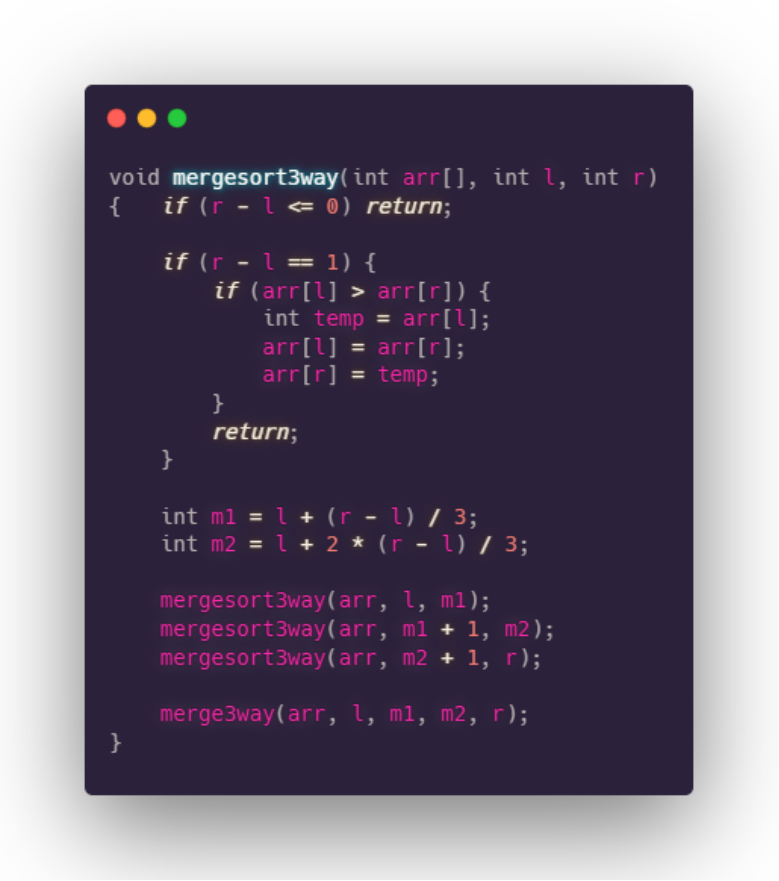
Rysunek 6: Wykres porównań i przypisań dla obu Insertion Sortów



Rysunek 7: Wykres czasu wykonania dla obu Insertion Sortów

0.4 Merge Sort

Sortowanie przez scalanie to klasyczny algorytm wykorzystujący metodę *dziel i zwyciężaj*. Algorytm ma strukturę rekurencyjną i działa w trzech krokach: dzieli zestaw danych na dwie części, stosuje sortowanie przez scalanie dla każdej z nich osobno, a następnie łączy posortowane podciągi w jeden posortowany ciąg. Złożoność obliczeniowa tego algorytmu wynosi $O(n \log n)$. Zmodyfikowana wersja sortowania dzieli zestaw danych na trzy części, a następnie postępuje w analogiczny sposób jak klasyczny algorytm. Na Rysunku 7 przedstawiono proces scalania w zmodyfikowanym algorytmie. Ta wersja również powinna charakteryzować się złożonością obliczeniową $O(n \log n)$, jednak może wykonywać więcej porównań niż przypisań ze względu na większą liczbę operacji porównywania podczas scalania.



```
void mergesort3way(int arr[], int l, int r)
{   if (r - l <= 0) return;

    if (r - l == 1) {
        if (arr[l] > arr[r]) {
            int temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;
        }
        return;
    }

    int m1 = l + (r - l) / 3;
    int m2 = l + 2 * (r - l) / 3;

    mergesort3way(arr, l, m1);
    mergesort3way(arr, m1 + 1, m2);
    mergesort3way(arr, m2 + 1, r);

    merge3way(arr, l, m1, m2, r);
}
```

Rysunek 8: Fragment kodu zmodyfikowanego Merge Sort

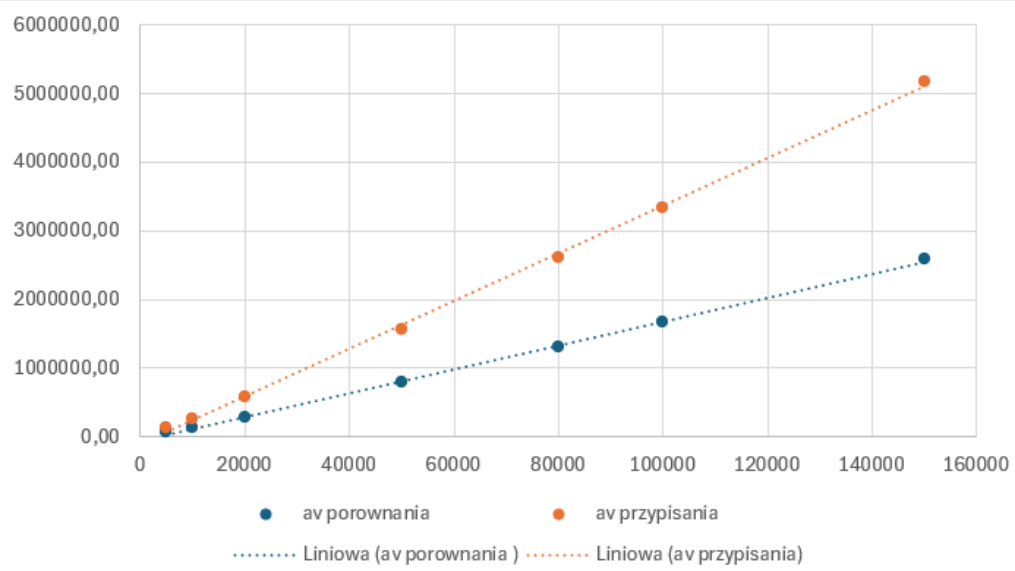
0.4.1 Pomiary dla sortowania przez scalanie

Dla każdego z siedmiu podanych wcześniej rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, liczby porównań oraz przypisań. Wyniki tych pomiarów przedstawia Tabela 3.

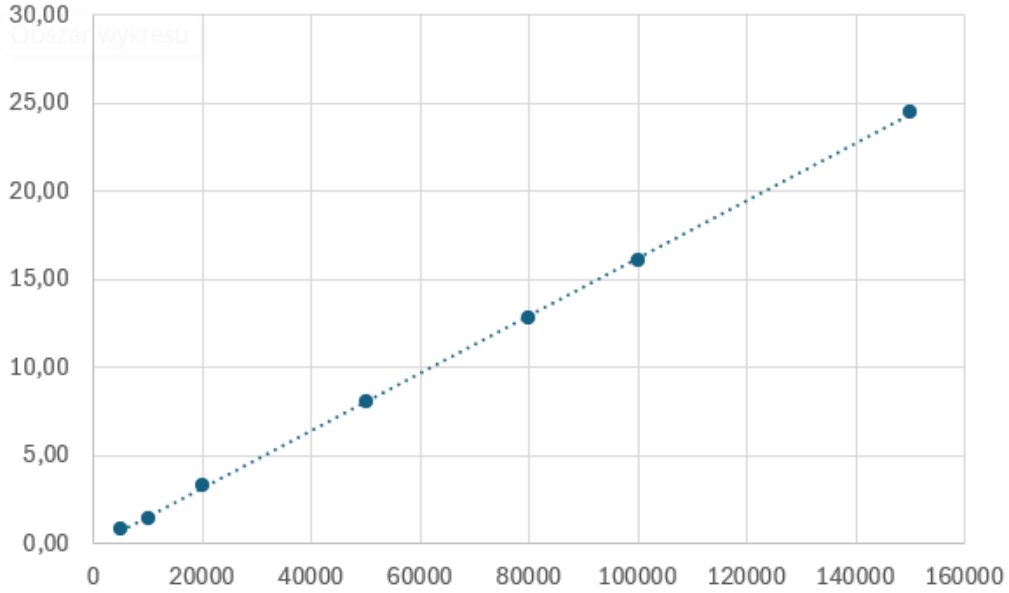
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	61 808.00	123 616.00	0.79
10 000	133 616.00	267 232.00	1.48
20 000	287 232.00	574 464.00	3.29
50 000	784 464.00	1 568 930.00	8.11
80 000	1 308 930.00	2 617 860.00	12.81
100 000	1 668 930.00	3 337 860.00	16.11
150 000	2 587 860.00	5 175 710.00	24.49

Tabela 3: Wyniki pomiarów dla Merge Sort

Wyniki przedstawiono również na Rysunkach 9 i 10.



Rysunek 9: Wykres porównań i przypisań dla Merge Sort



Rysunek 10: Wykres czasu wykonania dla Merge Sort

Funkcję czasu aproksymowano funkcją liniową, ponieważ jest ona zbliżona do funkcji liniowo-logarytmicznej dla małych rozmiarów danych. Wyniki w tabeli są jednak bardziej zbliżone do funkcji liniowej, dlatego możemy uznać, że dla niewielkich danych złożoność jest lepsza, niż zakładaliśmy. Warto również zauważyć, że sortowanie przez scalanie wykonuje zdecydowanie więcej przypisań niż porównań, co było zgodne z założeniami. Obie te zależności zostały przybliżone funkcjami liniowymi.

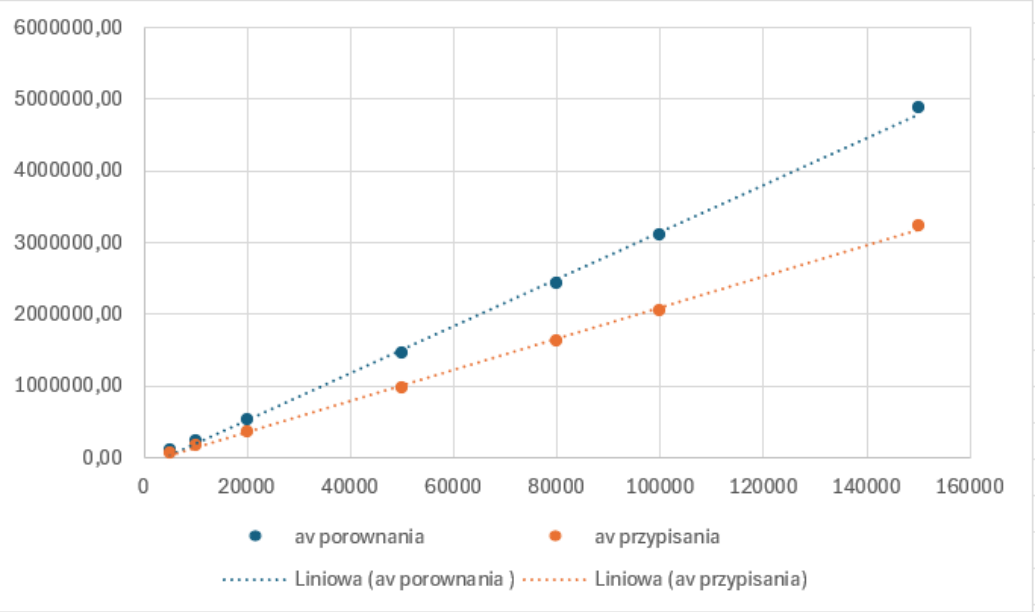
0.4.2 Pomiary dla sortowania przez scalanie z modyfikacją

Dla każdego z siedmiu podanych wcześniej rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, liczby porównań oraz przypisań. Wyniki tych pomiarów przedstawia Tabela 4.

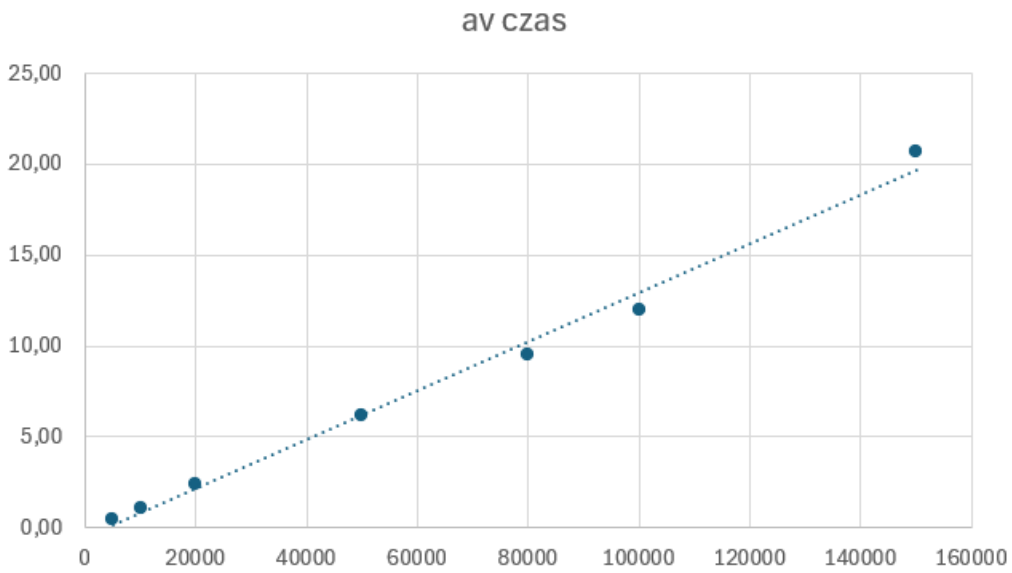
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	110 435.00	76 091.90	0.51
10 000	241 450.00	165 156.00	1.11
20 000	530 273.00	360 470.00	2.37
50 000	1 458 220.00	977 392.00	6.17
80 000	2 440 130.00	1 631 450.00	9.54
100 000	3 112 040.00	2 061 350.00	11.96
150 000	4 874 750.00	3 232 130.00	20.67

Tabela 4: Wyniki pomiarów dla zmodyfikowanego Merge Sort (3-way)

Wyniki przedstawiono również na Rysunkach 11 i 12.



Rysunek 11: Wykres porównań i przypisań dla Merge Sort(3-way)

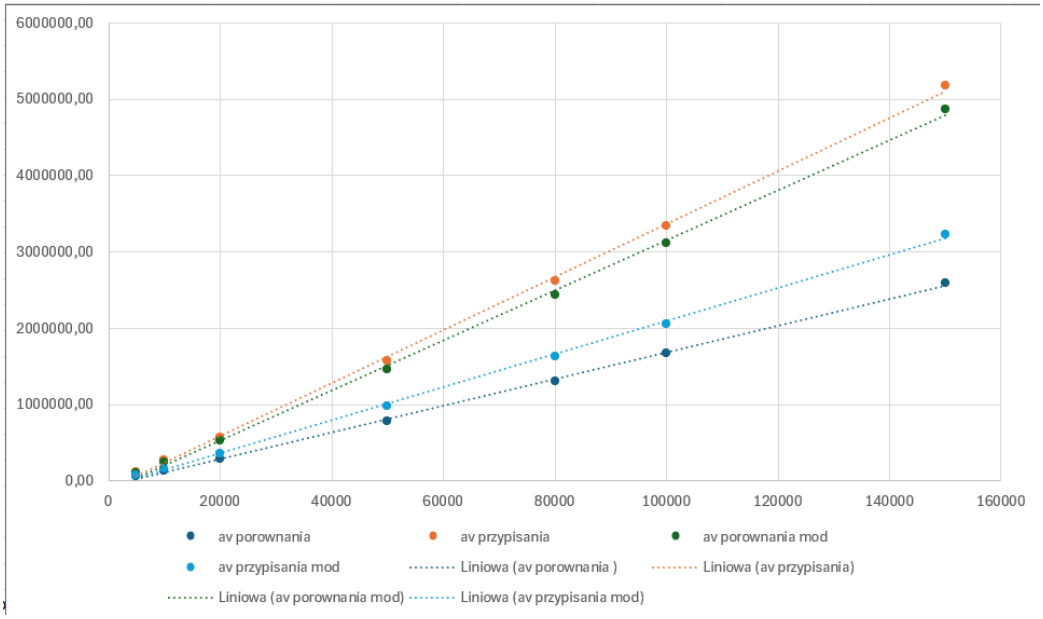


Rysunek 12: Wykres czasu wykonywania dla Merge Sort(3-way)

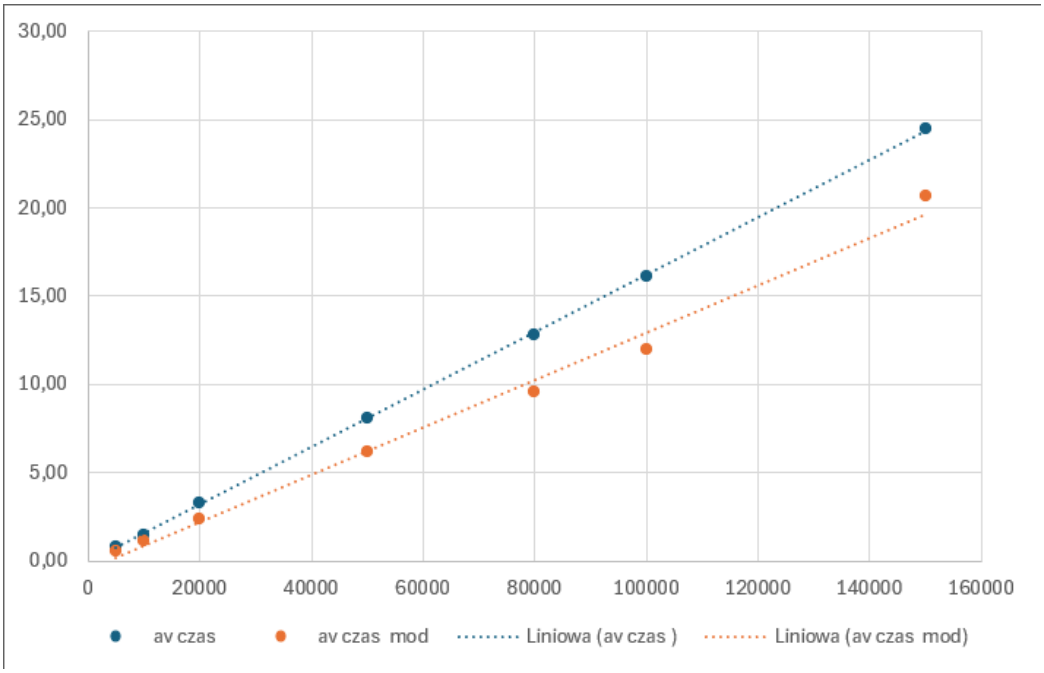
W tym przypadku funkcja czasu również została aproksymowana funkcją liniową, jednak wyniki w tabeli wskazują, że wartości rosną liniowo. Możemy więc stwierdzić, że ten algorytm ma lepszą złożoność obliczeniową, niż zakładaliśmy. Sortowanie przez scalanie z podziałem na trzy części ma więcej porównań niż przypisań, w przeciwieństwie do klasycznego merge sorta. Obie funkcje zostały przybliżone funkcjami liniowymi.

0.4.3 Porównanie algorytmów i wnioski

Na Rysunkach 13 oraz 14 znajdują się wykresy zbiorcze odpowiadające wykresom z punktów 0.4.1 oraz 0.4.2. Na wykresach widać, że oba algorytmy działają w podobnym czasie, jednak zmodyfikowana wersja jest nieco szybsza. Ma jednak zdecydowanie więcej porównań niż podstawowa wersja, natomiast klasyczna wersja ma znacznie więcej przypisań. Mimo wszystko, dla losowych wartości i małych rozmiarów tablic, lepiej wypada zmodyfikowany merge sort.



Rysunek 13: Wykres porównań i przypisań dla obu Merge Sortów



Rysunek 14: Wykres czasu wykonywania dla obu Merge Sortów

0.5 Heap Sort

Sortowanie przez kopcowanie jest bardziej zaawansowanym algorytmem, ponieważ wykorzystuje strukturę danych — kopiec. Największy element kopca znajduje się w korzeniu, co można wykorzystać do wybierania kolejnych elementów w odpowiedniej kolejności. Po usunięciu elementu z korzenia należy naprawić strukturę kopca, a następnie powtarzać operację, aż wszystkie elementy będą posortowane. Sortowanie przez kopcowanie ma złożoność obliczeniową $O(n \log n)$.

Heap sort wykorzystujący kopiec trójarny działa podobnie, jednak każdy węzeł kopca ma trzech potomków zamiast dwóch. Dzięki temu drzewo kopca jest niższe, co może zmniejszyć liczbę porównań podczas sortowania. Budowa kopca i jego naprawianie przebiegają analogicznie do kopca binarnego, z uwzględnieniem trzeciego dziecka (na Rysunku 15 znajduje się fragment kodu). Złożoność obliczeniowa tego algorytmu wynosi $O(n \log n)$, chociaż ma on potencjał działać szybciej niż heap sort binarny.

```
void zkopcuj3(int arr[], int n, int i) {
    int largest = i;
    int dzieciak1 = 3*i + 1;
    int dzieciak2 = 3*i + 2;
    int dzieciak3 = 3*i + 3;

    if (dzieciak1 < n && arr[dzieciak1] > arr[largest]) {
        largest = dzieciak1;
    }
    if (dzieciak2 < n && arr[dzieciak2] > arr[largest]) {
        largest = dzieciak2;
    }
    if (dzieciak3 < n && arr[dzieciak3] > arr[largest]) {
        largest = dzieciak3;
    }

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        zkopcuj3(arr, n, largest);
    }
}
```

Rysunek 15: Fragment kodu zmodyfikowanego Merge Sort

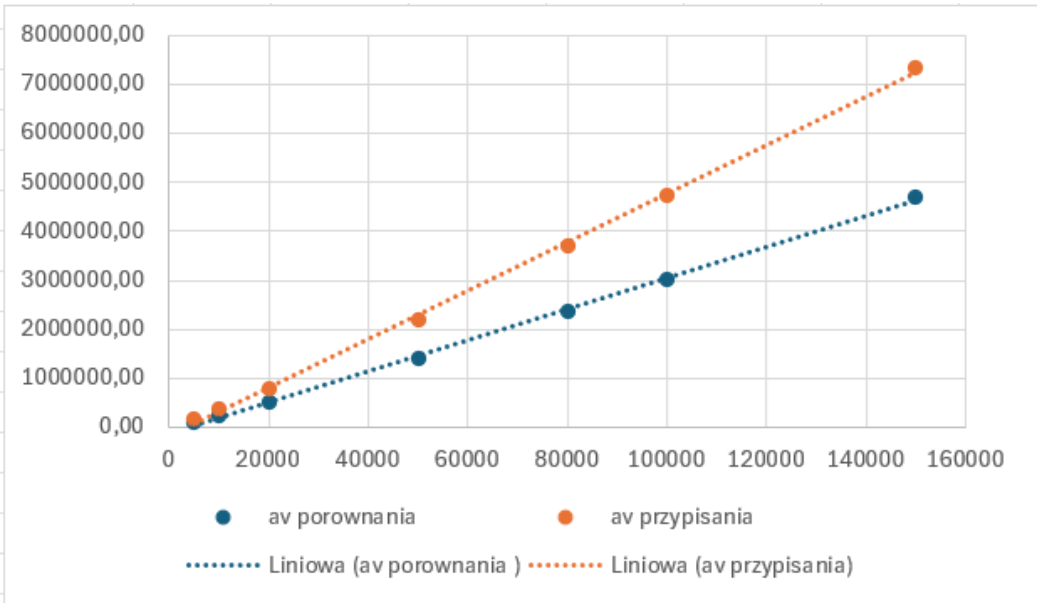
0.5.1 Pomiary dla sortowania przez kopcowanie

Dla każdego z siedmiu podanych wcześniej rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, liczby porównań oraz przypisań. Wyniki tych pomiarów przedstawia Tabela 5.

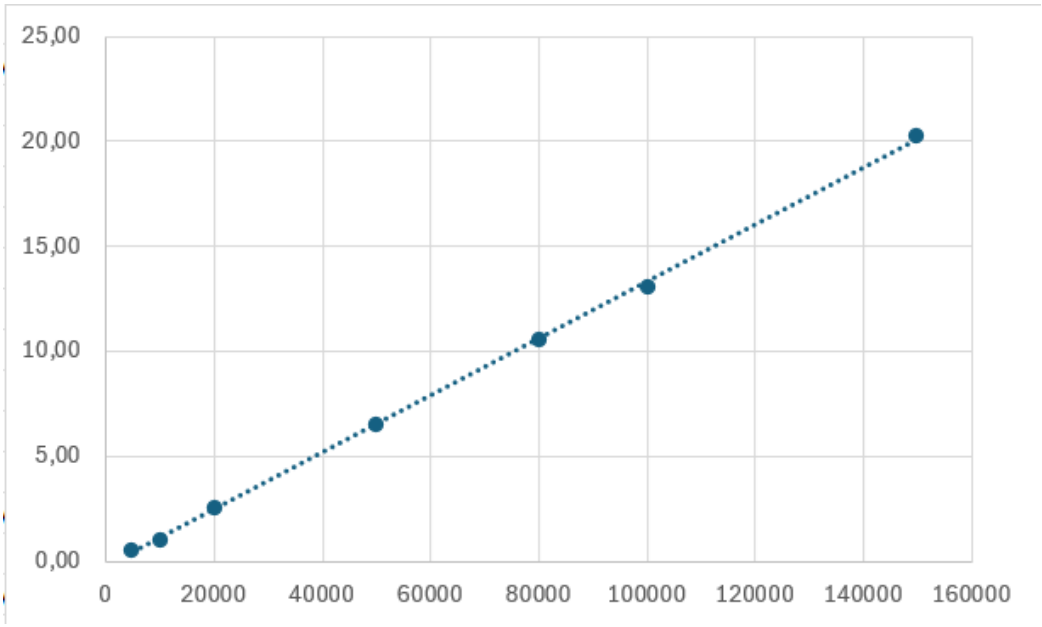
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	107 689.00	171 308.00	0.53
10 000	235 374.00	372 597.00	1.05
20 000	510 734.00	805 167.00	2.59
50 000	1 409 820.00	2 212 440.00	6.50
80 000	2 363 010.00	3 700 900.00	10.59
100 000	3 019 650.00	4 724 860.00	13.11
150 000	4 700 630.00	7 344 030.00	20.27

Tabela 5: Wyniki porównań i przypisań pomiarów dla Heap Sort

Wyniki przedstawiono również na Rysunkach 16 i 17.



Rysunek 16: Wykres porównań i przypisań dla Heap Sort



Rysunek 17: Wykres czasu wykonywania dla Heap Sort

Funkcja czasu została aproksymowana funkcją liniową, ponieważ jest ona zbliżona do funkcji liniowo-logarytmicznej dla małych rozmiarów danych. Jednak wyniki w tabeli rosną szybciej niż typowa funkcja liniowa, więc możemy stwierdzić, że przewidywana złożoność obliczeniowa jest zgodna z wynikami badań. Warto też zauważyć, że sortowanie przez kopcowanie ma zdecydowanie więcej przypisań niż porównań. Obie te zależności zostały przybliżone funkcjami liniowymi.

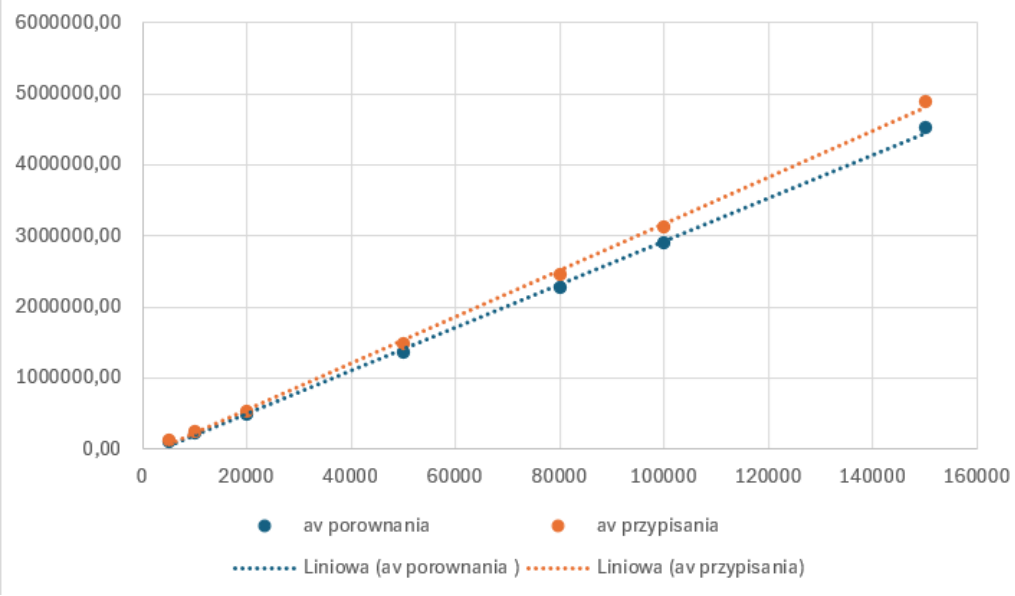
0.5.2 Pomiary dla sortowania przez kopcowanie z modyfikacją

Dla każdego z siedmiu podanych wcześniej rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, liczby porównań oraz przypisań. Wyniki tych pomiarów przedstawia Tabela 6.

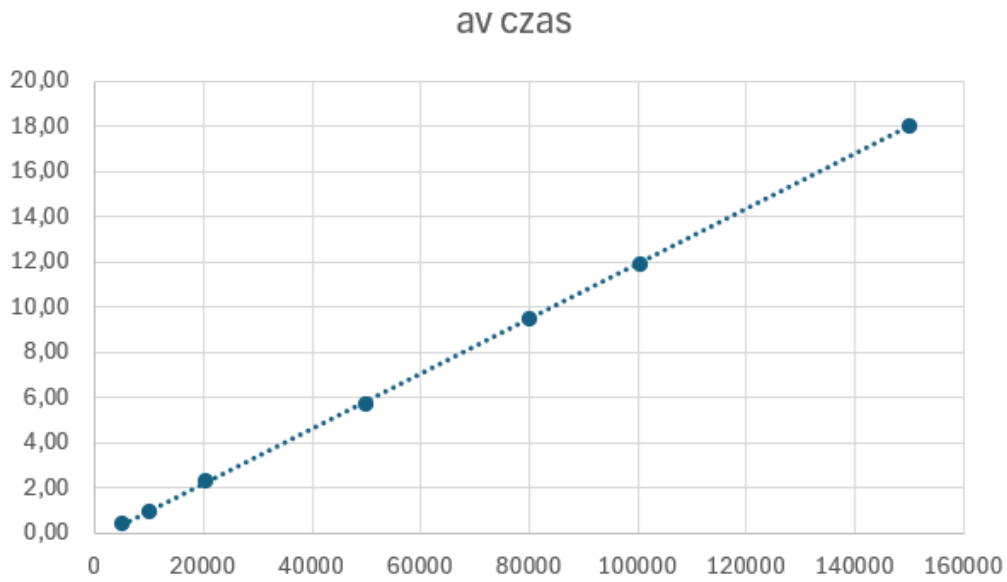
Rozmiar	Porównania	Przypisania	Czas [ms]
5 000	104 368.00	116 362.00	0.43
10 000	226 583.00	250 522.00	0.95
20 000	493 447.00	541 351.00	2.29
50 000	1 358 710.00	1 478 390.00	5.71
80 000	2 270 510.00	2 462 010.00	9.53
100 000	2 897 370.00	3 136 760.00	11.95
150 000	4 526 100.00	4 885 260.00	18.02

Tabela 6: Wyniki pomiarów dla zmodyfikowanego Heap Sort (kopiec ternarny)

Wyniki przedstawiono również na Rysunkach 18 i 19.



Rysunek 18: Wykres porównań i przypisań dla Heap Sort (kopiec ternarny)

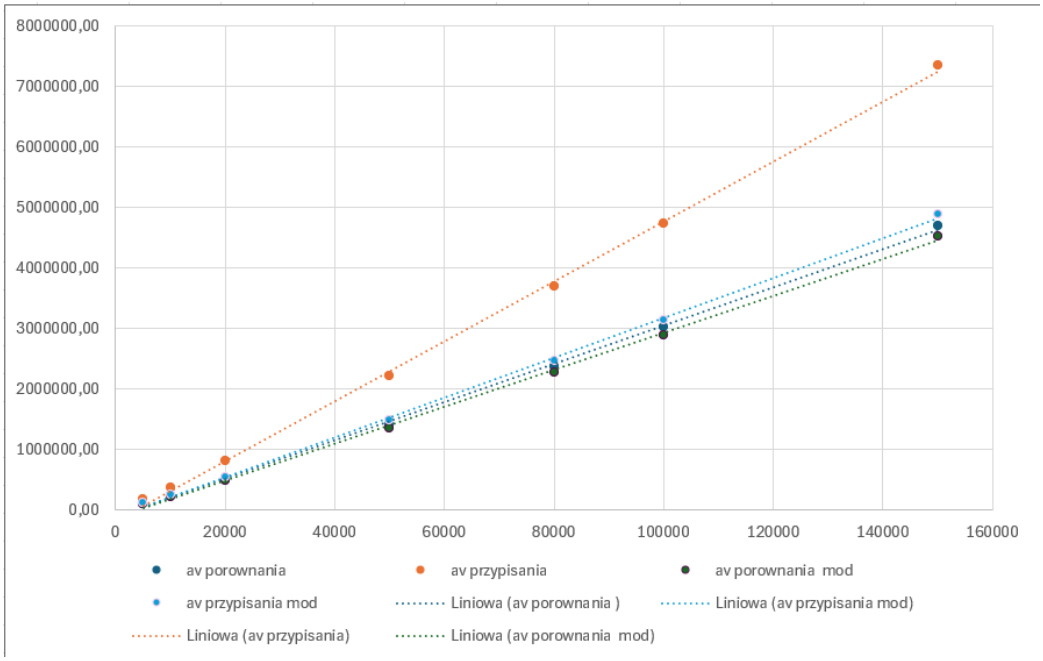


Rysunek 19: Wykres czasu wykonywania dla Heap Sort (kopiec ternarny)

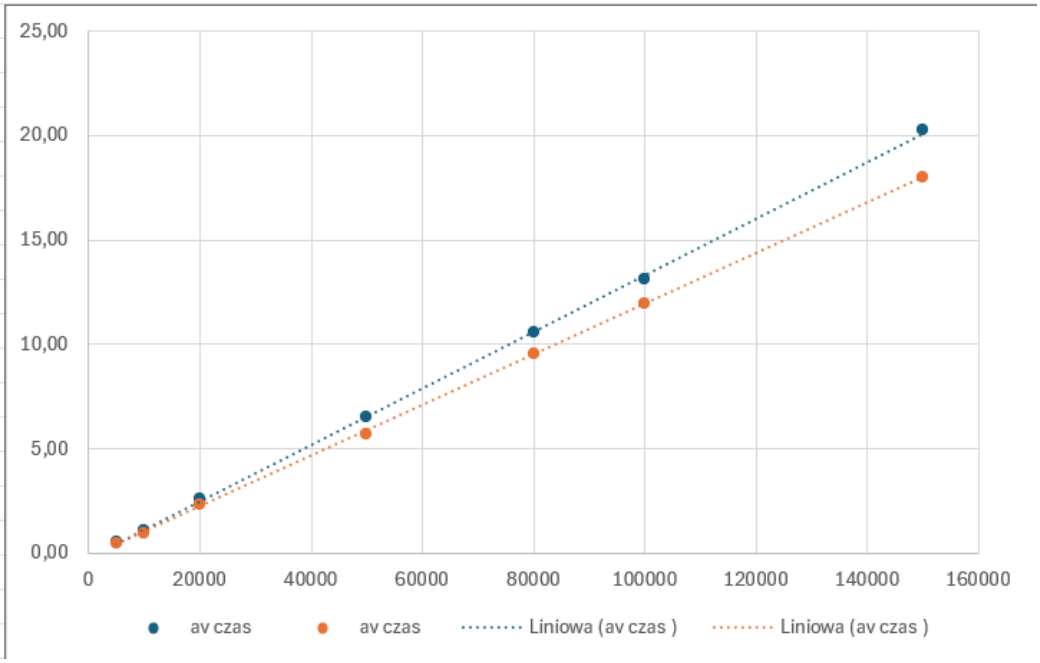
W tym przypadku funkcja czasu również została aproksymowana funkcją liniową, jednak wyniki w tabeli wskazują, że wartości rosną liniowo. Możemy więc stwierdzić, że ten algorytm ma lepszą złożoność obliczeniową, niż zakładaliśmy. Sortowanie przez kopcowanie trójarne ma podobne wartości liczby porównań i przypisań. Obie te funkcje zostały przybliżone funkcjami liniowymi.

0.5.3 Porównanie algorytmów i wnioski

Na Rysunkach 20 oraz 21 znajdują się wykresy zbiorcze dla wykresów z punktów 0.5.1 oraz 0.5.2. Na wykresach widać, że algorytmy mają podobne parametry, a największa różnica dotyczy liczby przypisań. Dla większych tablic zaczynamy również obserwować różnice w czasach wykonywania algorytmów. Można więc stwierdzić, że zmodyfikowana wersja algorytmu jest bardziej efektywna, zwłaszcza dla dużych zbiorów danych.



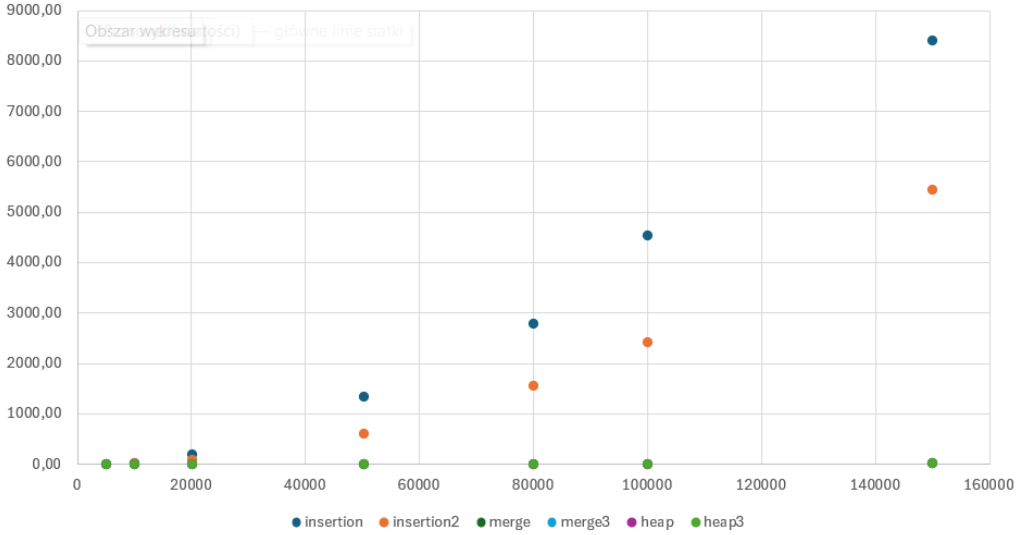
Rysunek 20: Wykres porównań i przypisań dla obu Heap Sortów



Rysunek 21: Wykres czasu wykonywania dla obu Heap Sortów

0.6 Porównanie wszystkich algorytmów sortowania

Na Rysunku 22 przedstawiono wykres porównujący czasy wykonywania algorytmów opisanych w podpunktach 0.3–0.5. Jak możemy zauważyć, insertion sort i jego modyfikacja są najbardziej czasochłonne i nadają się jedynie dla naprawdę małych zestawów danych. Pozostałe algorytmy mają zbliżone wartości czasów – są bardzo szybkie i działają dobrze nawet dla dużych zbiorów, większych niż w naszym badaniu. Ciężko jednak jednoznacznie wskazać najefektywniejszy algorytm, ponieważ zależy to od konkretnych potrzeb oraz dostępnych zasobów pamięci.



Rysunek 22: Wykresczasu wykonywania dla wszystkich algorytmów

0.7 Wnioski

Z naszego badania wynika, że lepiej wypadły zmodyfikowane wersje algorytmów. Zwłaszcza dla insertion sorta zmodyfikowana wersja w każdym aspekcie radziła sobie lepiej. W przypadku merge sorta zmodyfikowany algorytm osiągnął lepsze wyniki, z wyjątkiem liczby porównań. Dla heap sorta również zmodyfikowany algorytm wypadł korzystniej. Należy jednak zauważyć, że dla dwóch ostatnich algorytmów wybrano zbyt mało oraz zbyt małe wartości testowe. Przy ponownym przeprowadzaniu badania warto zastosować większe rozmiary danych, aby uzyskać rzeczywistą złożoność oblicze-

niową pokrywającą się z teoretycznymi założeniami, czego nie udało się w pełni osiągnąć w obecnym eksperymencie.