

Sprawozdanie z Analizy Algorytmów Sortowania

Wojciech Ługowski (287292)

4 listopada 2025

Spis treści

1 Wstęp	2
2 Ciekawe fragmenty kodu	2
2.1 Scalanie Trójdrożne (ModMergeSort)	2
2.2 Kopiec Trójkowy (HeapSortTernary)	3
3 Wyniki i Wykresy	4
3.1 Skalowalność: $O(n^2)$ vs $O(n \log n)$	4
3.2 Wpływ wstępnego uporządkowania danych	4
3.3 Szczegółowa analiza modyfikacji	6
3.4 Analiza kosztów operacji (porównania vs przypisania)	9
4 Wnioski	11

1 Wstęp

W tym sprawozdaniu porównałem 6 algorytmów sortowania, aby sprawdzić, jak różnią się ich szybkością i jak na ich działanie wpływają wprowadzone modyfikacje.

Podzieliłem je na dwie grupy:

- Trzy klasyczne algorytmy: Insertion Sort (Sortowanie przez wstawianie), Merge Sort (Sortowanie przez scalanie) oraz Heap Sort (Sortowanie przez kopcowanie).
- Trzy autorskie modyfikacje tych algorytmów, mające na celu optymalizację: wstawianie parami (dla Insertion Sort), scalanie trójdorożne (dla Merge Sort) oraz sortowanie z użyciem kopca trójkowego (dla Heap Sort).

Wszystkie algorytmy zostały zaimplementowane w języku C++. Podczas testów mierzyłem czas ich wykonania oraz zliczałem kluczowe operacje: porównania elementów ('comp') i przypisywanie wartości ('sign'). Testy przeprowadziłem na zbiorach danych o różnej wielkości (od $N = 10$ do $N = 50000$) oraz o różnym stopniu wstępniego uporządkowania (dane losowe, posortowane rosnąco i posortowane malejąco).

2 Ciekawe fragmenty kodu

Bazowe implementacje Insertion Sort, Merge Sort czy Heap Sort są dobrze znane. Poniżej skupiam się na najciekawszych fragmentach kodu dotyczących wprowadzonych modyfikacji.

2.1 Scalanie Trójdorożne (ModMergeSort)

W przeciwieństwie do standardowego Merge Sort, który dzieli tablicę na dwie części, moja modyfikacja dzieli ją na trzy. Kluczową częścią jest funkcja MergeThree, która odpowiada za scalanie trzech już posortowanych podtablic w jedną. Proces ten realizowany jest w dwóch etapach:

1. Scalenie dwóch pierwszych podtablic do pomocniczej tablicy 'temp'.
2. Scalenie zawartości 'temp' z trzecią podtablicą z powrotem do głównej tablicy 'A'.

```
1 void MergeThree(int A[], int p, int s1, int s2, int k,
2                 long long &comp, long long &sign) {
3     int nTemp = s2 - p + 1;
4     int *temp = new int[nTemp];
5     int i = p, j = s1 + 1, t = 0;
6     while (i <= s1 && j <= s2) {
7         comp++;
8         if (A[i] <= A[j]) {
9             temp[t] = A[i];
10            sign++;
11            i++;
12            t++;
13        }
14        /// ...reszta warunkow dla scalania dwoch pierwszych czesci...
15        /// nastepnie cala zawartosc tablicy temp jest scalana z trzecia czescia (od
16        /// s2+1 do k)
17        /// z powrotem do tablicy glownej A. Dla zachowania czytelnosci pominiento
18        /// to w liscie.
19        delete[] temp;
20 }
```

Listing 1: Fragment implementacji funkcji MergeThree

2.2 Kopiec Trójkowy (HeapSortTernary)

Ta modyfikacja polega na zastąpieniu klasycznego kopca binarnego (gdzie każdy węzeł ma co najwyżej dwóch potomków) kopcem trójkowym (gdzie każdy węzeł może mieć do trzech potomków). Skutkuje to "pływszą" strukturą kopca. Wymagało to dostosowania funkcji HeapifyTernary, która teraz musi znajdować największy element spośród aż czterech: rodzica i jego trzech potencjalnych dzieci.

```
1 void HeapifyTernary(int A[], int n, int i, long long &comp, long long &sign) {
2     int p1 = Part_1(i); // Oblicza indeks pierwszego dziecka
3     int p2 = Part_2(i); // Oblicza indeks drugiego dziecka
4     int p3 = Part_3(i); // Oblicza indeks trzeciego dziecka
5     int largest = i;    // Zak adamy, że rodzic jest największy
6
7     if (p1 <= n) {
8         comp++;
9         if (A[p1] > A[largest]) largest = p1;
10    }
11   // ... podobne sprawdzenia dla p2 i p3 ...
12   // ... jeśli jest potrzeba zamiany, wykonujemy ja i rekurencyjnie wywołujemy
13   //       Heapify ...
14   if (largest != i) {
15       swap(A[i], A[largest]); sign++;
16       HeapifyTernary(A, n, largest, comp, sign);
17   }
}
```

Listing 2: Fragment implementacji funkcji HeapifyTernary

3 Wyniki i Wykresy

Ta sekcja przedstawia najważniejsze wyniki eksperymentów w postaci wykresów wygenerowanych w **LibreOffice Calc** na podstawie zebranych danych.

3.1 Skalowalność: $O(n^2)$ vs $O(n \log n)$

Pierwszy wykres doskonale ilustruje fundamentalną różnicę w klasach złożoności algorytmów sortowania, pokazując, jak czas wykonania rośnie wraz z rozmiarem danych N dla danych losowych.



Rysunek 1: Czas sortowania dla danych losowych. Algorytmy $O(n^2)$ (czerwony i żółty) wykazują gwałtowny wzrost czasu wykonania.

Na Rysunku 1 widać, że InsertionSort i jego modyfikacja (o złożoności $O(n^2)$) rosną kwadratowo, a ich czas wykonania drastycznie zwiększa się z każdym kolejnym rzędem wielkości N . Pozostałe algorytmy ($O(n \log n)$), takie jak MergeSort i HeapSort wraz z ich modyfikacjami, są tak wydajne, że ich linie leżą blisko osi X, demonstrując znacznie lepszą skalowalność.

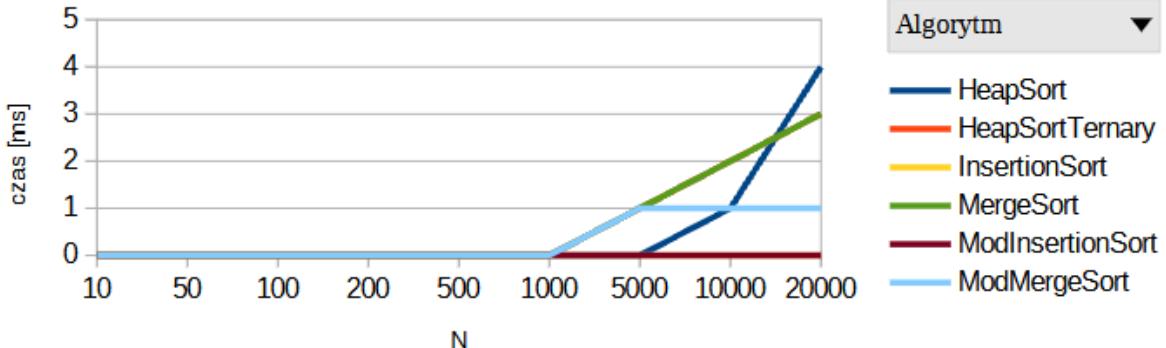
3.2 Wpływ wstępnego uporządkowania danych

Charakterystyka początkowego ułożenia danych może mieć ogromny wpływ na wydajność niektórych algorytmów. Poniższe wykresy analizują, jak algorytmy radzą sobie z danymi już posortowanymi (najlepszy przypadek dla Insertion Sort) oraz posortowanymi odwrotnie (najgorszy przypadek).

TypDanych | Posortowane ▼

Rozmiar N vs czas

dla danych posortowanych



N ▼

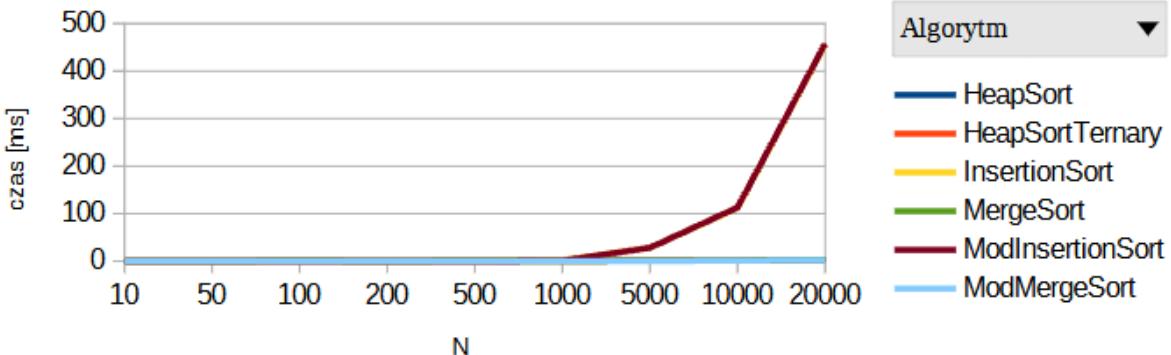
Rysunek 2: Czas sortowania dla danych posortowanych rosnąco (najlepszy przypadek dla InsertionSort).

Rysunek 2 ukazuje niezwykłą wydajność InsertionSort dla danych już posortowanych. Dzięki temu, że jego złożoność w najlepszym przypadku wynosi $O(n)$, stał się on najszybszym algorytmem ze wszystkich w tym scenariuszu.

TypDanych | Odwrotnie ▼

Rozmiar N vs czas

dla danych odwrotnie posortowanych



N ▼

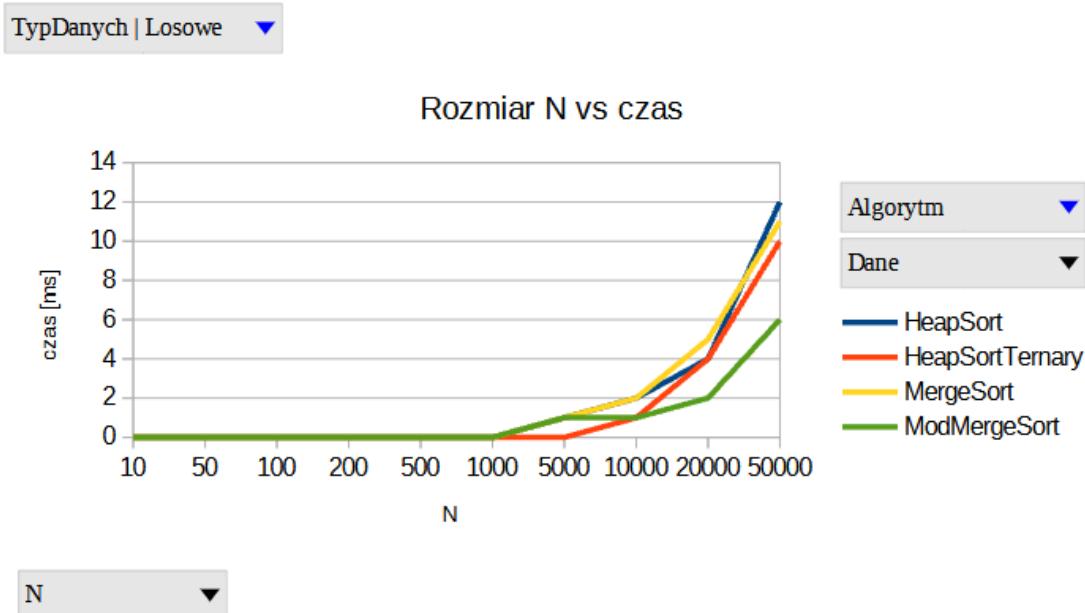
Rysunek 3: Czas sortowania dla danych posortowanych malejaco (najgorszy przypadek dla InsertionSort).

Z kolei Rysunek 3 ilustruje najgorszy przypadek dla InsertionSort – dane posortowane odwrotnie. W tej sytuacji algorytm jest ekstremalnie wolny. Interesującym wnioskiem jest to, że

algorytmy MergeSort i HeapSort (i ich modyfikacje) utrzymywały bardzo podobne czasy wykonania, niezależnie od wstępnego uporządkowania danych, co potwierdza ich stabilność.

3.3 Szczegółowa analiza modyfikacji

Aby dokładnie ocenić wpływ wprowadzonych modyfikacji, skupiono się na bezpośrednim porównaniu każdej pary algorytmów (oryginał vs modyfikacja) na danych losowych.

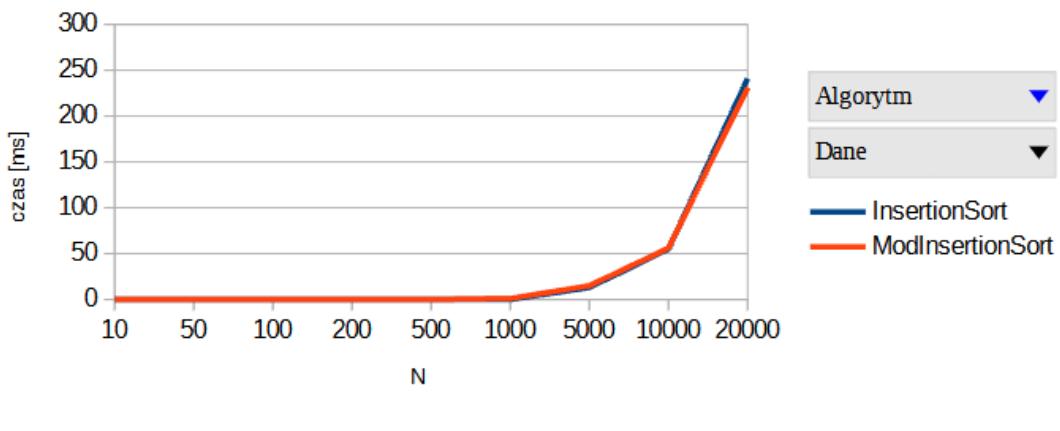


Rysunek 4: Porównanie czasów algorytmów $O(n \log n)$ dla danych losowych.

Rysunek 4 (będący "zbliżeniem" na algorytmy $O(n \log n)$ z Rysunku 1) wyraźnie pokazuje, że obie modyfikacje algorytmów o złożoności $O(n \log n)$ przyniosły poprawę. Zarówno ModMergeSort (linia jasnoniebieska) jak i HeapSortTernary (linia pomarańczowa) wykazują konsekwentnie niższe czasy wykonania niż ich bazowe wersje (MergeSort - granatowa i HeapSort - ciemnozielona).

TypDanych | Losowe ▼

Rozmiar N vs czas

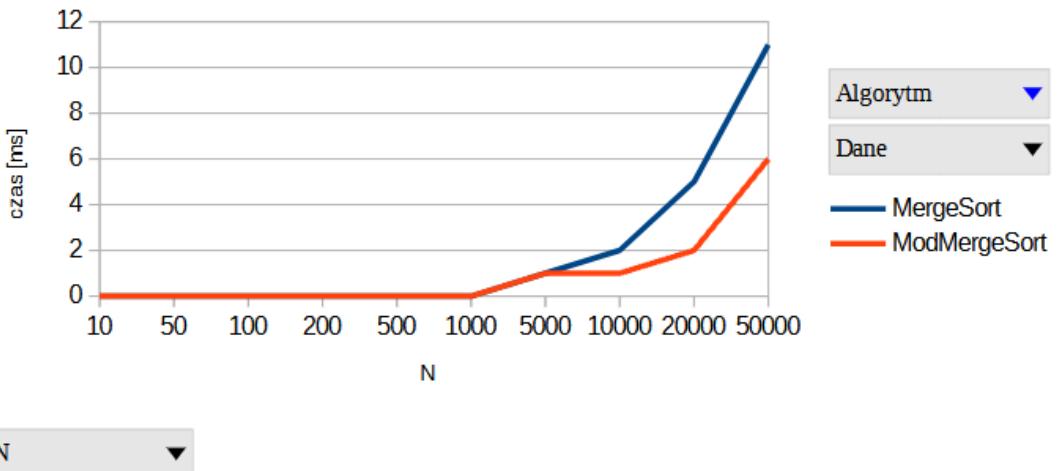


Rysunek 5: Porównanie czasu wykonania: InsertionSort vs ModInsertionSort dla danych losowych.

Jak widać na Rysunku 5, modyfikacja InsertionSort (sortowanie parami) nie przyniosła niesłychanego zysku wydajnościowego dla danych losowych. Czasy wykonania obu algorytmów są praktycznie identyczne, co potwierdzają także dane w tabeli 1. Dodatkowa złożoność nie przełożyła się na wymierne korzyści.

TypDanych | Losowe ▼

Rozmiar N vs czas

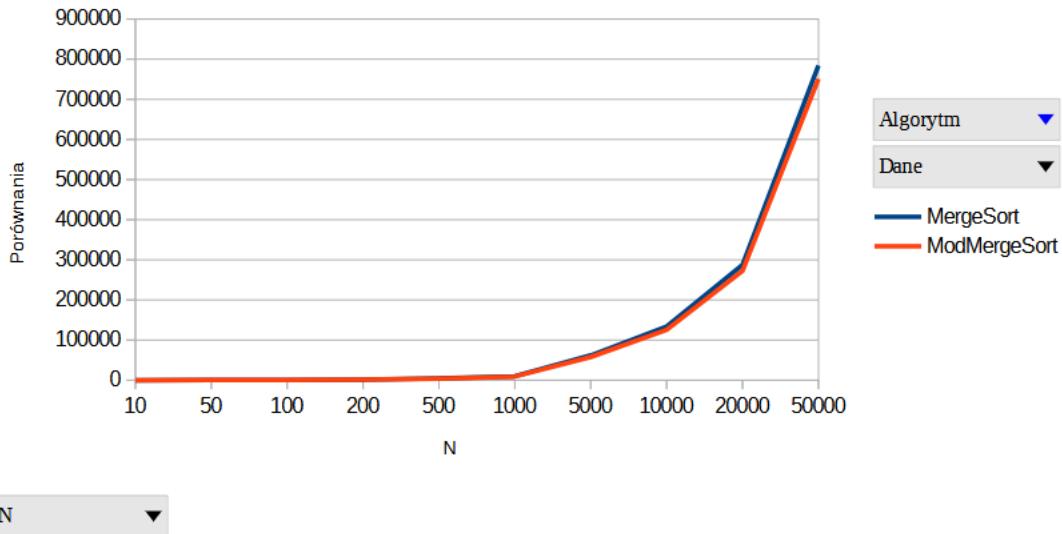


Rysunek 6: Porównanie czasu wykonania: MergeSort vs ModMergeSort dla danych losowych.

Rysunek 6 wyraźnie pokazuje, że modyfikacja ModMergeSort była konsekwentnie szybsza od standardowej wersji. Ten zysk w czasie jest bardzo obiecujący.

TypDanych | Losowe ▾

Analiza liczby porównań algorytmów

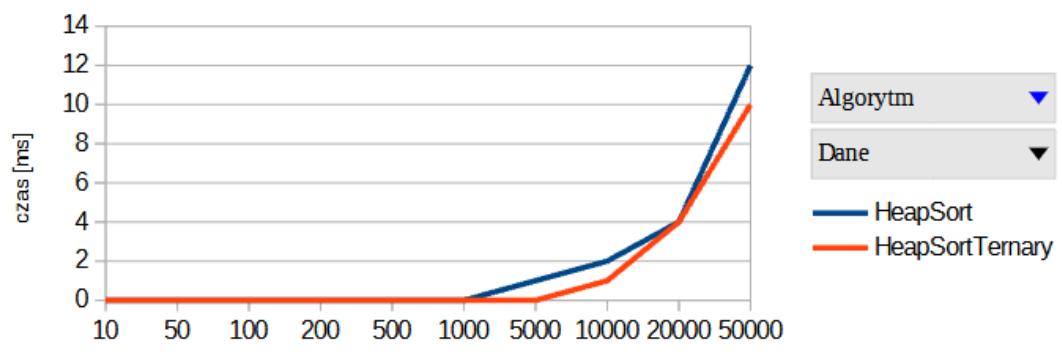


Rysunek 7: Porównanie liczby porównań: MergeSort vs ModMergeSort dla danych losowych.

Rysunek 7 wyjaśnia źródło przewagi ModMergeSort: algorytm ten wykonywał minimalnie mniej porównań dla każdej testowanej wielkości N . Mniejsza liczba porównań bezpośrednio przekłada się na krótszy czas wykonania.

TypDanych | Losowe ▾

Rozmiar N vs czas

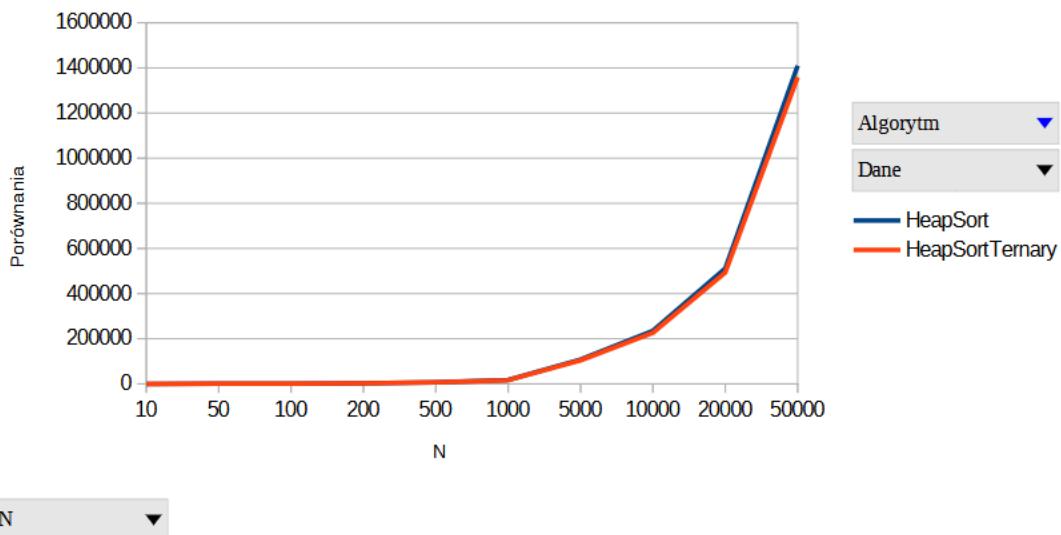


Rysunek 8: Porównanie czasu wykonania: HeapSort vs HeapSortTernary dla danych losowych.

W przypadku HeapSort (Rysunek 8), modyfikacja na kopiec trójkowy również okazała się szybsza od wersji binarnej. Podobnie jak przy ModMergeSort, zysk wydajności jest wyraźnie widoczny.

TypDanych | Losowe ▾

Analiza liczby porównań algorytmów



Rysunek 9: Porównanie liczby porównań: HeapSort vs HeapSortTernary dla danych losowych.

Co ciekawe, Rysunek 9 pokazuje, że liczba porównań dla obu wersji HeapSort była bardzo zbliżona. Oznacza to, że zysk wydajności kopca trójkowego musiał pochodzić z innego czynnika – jak to zostanie pokazane poniżej, była to drastycznie mniejsza liczba przypisań.

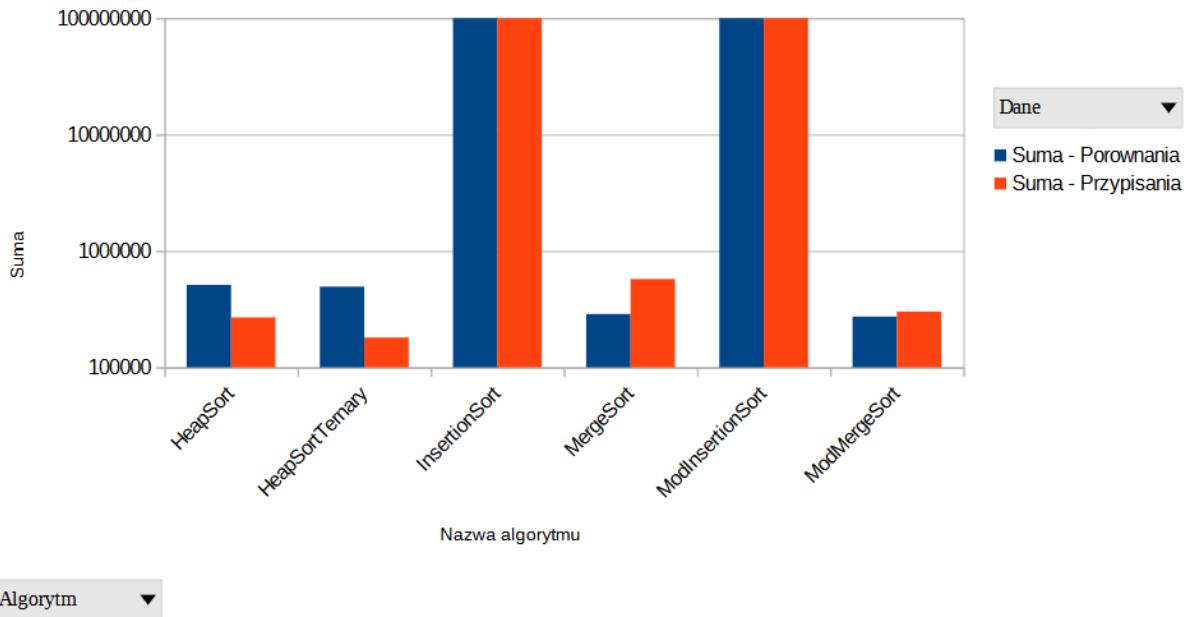
3.4 Analiza kosztów operacji (porównania vs przypisania)

Aby dogłębnie zrozumieć, dlaczego algorytmy różniły się wydajnością, przeanalizowano całkowitą liczbę wykonanych operacji porównania ('comp') oraz przypisania ('sign') dla testu z $N = 20000$ na danych losowych.

TypDanych | Losowe ▼ N | 20000 ▼

Suma porównań i przypisań

Dla N = 20000



Rysunek 10: Profil operacji (porównania i przypisania) dla N=20000 na danych losowych. Oś Y przedstawiona w skali logarytmicznej dla lepszej czytelności.

Rysunek 10 (z osią Y w skali logarytmicznej) jest kluczowy do zrozumienia mechanizmów stojących za zaobserwowanymi czasami. Wyraźnie widać, że:

- Algorytmy o złożoności $O(n^2)$ (InsertionSort i ModInsertionSort) wykonały o rzędy wielkości więcej operacji, co potwierdza ich nieefektywność dla dużych danych.
- Wśród algorytmów $O(n \log n)$, ModMergeSort wykonał zdecydowanie najmniej porównań, co jest zgodne z teorią, że podział na 3 zmniejsza głębokość rekursji potrzebnej do porównań.
- Z kolei HeapSortTernary wykonał znacznie mniej przypisań niż standardowy HeapSort. To sugeruje, że rzadsze przesuwanie danych (dzięki płytszej strukturze kopca trójkowego) skutecznie zrekompensowało fakt, że liczba porównań była zbliżona, prowadząc do ogólnego skrócenia czasu.

Poniższa tabela zbiera dokładne dane liczbowe dla tego kluczowego testu, pozwalając na precyzyjne porównania.

Tabela 1: Szczegółowe wyniki dla N=20000 (Dane losowe)

Algorytm	Czas (ms)	Porównania	Przypisania
InsertionSort	221	99563280	99583287
ModInsertionSort	221	99563280	99583289
MergeSort	4	287232	574464
ModMergeSort	2	272960	301320
HeapSort	4	510730	268324
HeapSortTernary	3	493320	180392

4 Wnioski

Przeprowadzone testy i analiza wykresów pozwoliły na sformułowanie następujących, kluczowych wniosków:

1. Klasa złożoności rzeczywiście ma duże znaczenie. Różnica między $O(n^2)$ a $O(n \log n)$ jest gigantyczna. Algorytmy InsertionSort nadają się tylko do bardzo małych danych.
2. Algorytmy "odporne" są bezpieczne. MergeSort i HeapSort działały tak samo szybko, nieważne czy dane były losowe, czy posortowane odwrotnie. To ich duża zaleta.
3. InsertionSort ma swoją niszę. Mimo że jest ogólnie wolny, to dla danych już posortowanych (albo prawie posortowanych) jest najszybszy ze wszystkich, bo działa w czasie liniowym.
4. Modyfikacje działają! Moje testy pokazały, że:
 - ModMergeSort faktycznie zmniejszyło liczbę porównań i czas wykonania.
 - HeapSortTernary mocno zmniejszył liczbę przypisań (przesunięć danych), co również przełożyło się na krótszy czas.
 - Modyfikacja InsertionSort nie dała żadnego zysku na danych losowych.
5. Nie ma jednego "najlepszego" algorytmu. To, który jest najlepszy, zależy od sytuacji. Jeśli dane są prawie posortowane, InsertionSort sprawuje się najlepiej. Jeśli zależy nam na jak najmniejszej liczbie przesunięć danych, HeapSortTernary był świetny.