ALGORYTMY I STURKTURY DANYCH

**Specyfikacja poszczególnych algorytmów sortujących:**

1. **BubbleSort (Sortowanie bąbelkowe)**

**Złożoność:**

Czasowa: średnia, pesymistyczna: O(n2), optymistyczna: O(n)

Pamięciowa: O(1)

**Idea:** Lokalne zamiany między sąsiednimi elementami, jeśli nie są we właściwej kolejności. Jest to jedna z najprostszych implementacji algorytmu sortowania.

**Cechy zaimplementowanego algorytmu:**

- stabilność

- sortowanie w miejscu

- zastosowane usprawnienia:

a) Sprawdzanie czy ciąg nie jest już posortowany

b) Pamiętanie miejsca ostatniej zmiany (ograniczenie ilości porównań posortowanej części danych)

1. **InsertSort (Sortowanie przez wstawianie)**

**Złożoność:**

Czasowa: średnia, pesymistyczna: O(n2), optymistyczna: O(n)

Pamięciowa: O(1)

**Idea:** Do części już posortowanej dokładamy kolejny element, szukając dla niego poprawnej pozycji

**Cechy zaimplementowanego algorytmu:**

- stabilność

- sortowanie w miejscu

- dobre działanie dla małych list

- idea sortowania używana w listach wiązanych posortowanych

1. **SelectSort (Sortowanie przez wybór)**

**Złożoność:**

Czasowa: średnia, pesymistyczna: O(n2), optymistyczna: O(n2)

Pamięciowa: O(1)

**Idea:** Szukamy kolejnej wartości minimalnej oraz dostawiamy ją do wcześniej znalezionych wartości

**Cechy zaimplementowanego algorytmu:**

- niestabilne (nie zachowana kolejność wejściowa dla takich samych elementów)

- sortowanie w miejscu

- dużo porównań (ale za to mało przepisań/zamian – O(n))

- nieefektywność dla już uporządkowanych danych

1. **ShellSort (Sortowanie Shella):**

**Złożoność:**

Czasowa: zależy od ciągu odstępów – od **O(n log2n)** do **O(n2)**

Pamięciowa: O(1)

**Idea:** Sortowany zbiór dzielimy na podzbiory, których elementy są odległe od siebie w sortowanym zbiorze o pewien odstęp *h*. Każdy z tych podzbiorów sortujemy algorytmem przez wstawianie. Następnie odstęp zmniejszamy. Sortowanie powtarzamy i znów zmniejszamy odstęp, aż osiągnie on wartość 1. Wtedy sortujemy już normalnie za pomocą *InsertSort*. Dzięki początkowym dużym odstępom elementy są przesuwane w zbiorze bardziej efektywnie - na duże odległości.

**Cechy zaimplementowanego algorytmu:**

- efektywność dla odstępów:

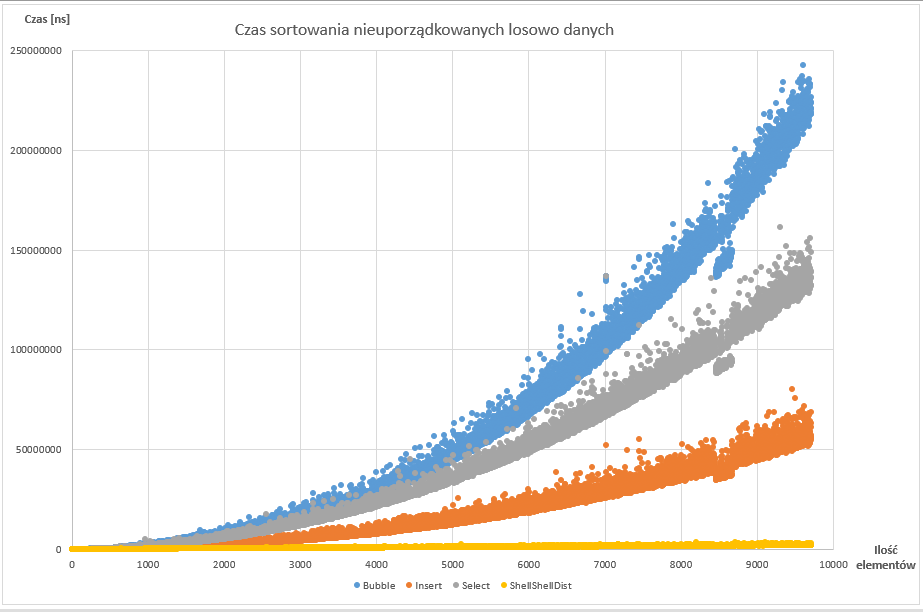
- zaproponowanych przez Shella – wzór na długość podziału (N/2k) = **O(n2)**

-odstępów zaproponowanych przez Sedgewicka – wzór na długość podziału (4k+3\*2k-1+1): **O(N4/3 )**

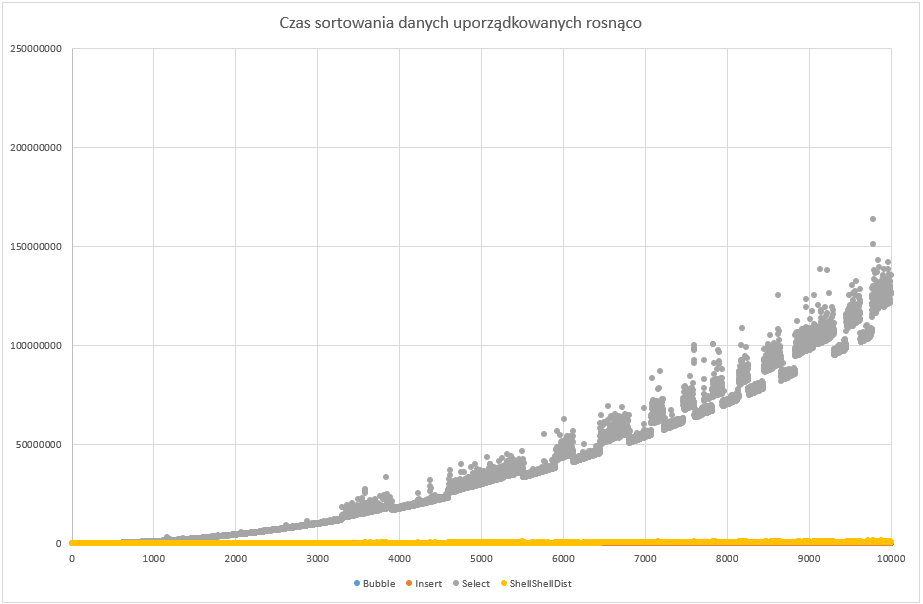
- niestabilne

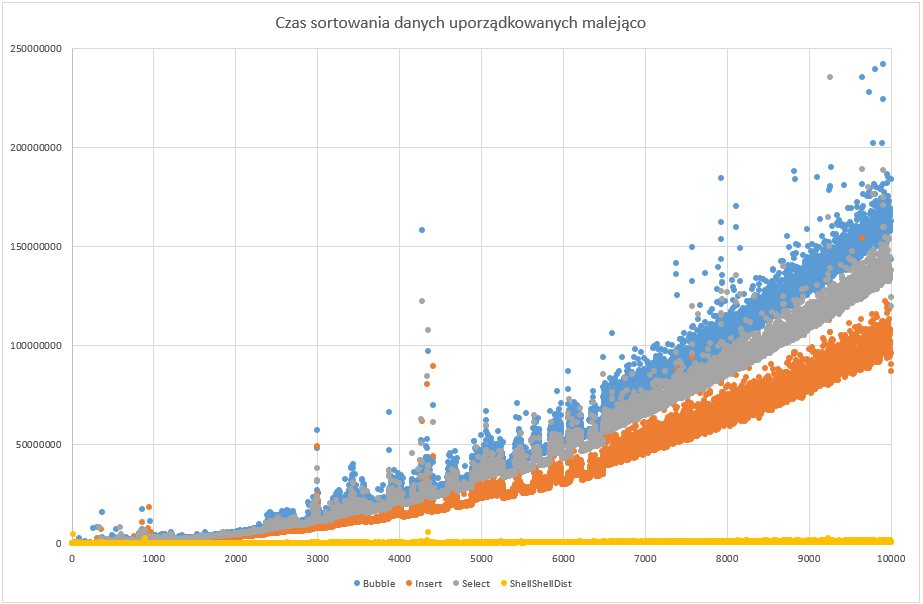
- sortowanie w miejscu

- rzadko stosowany ze względu na lepszą wydajność algorytmu sortującego *quickSort*

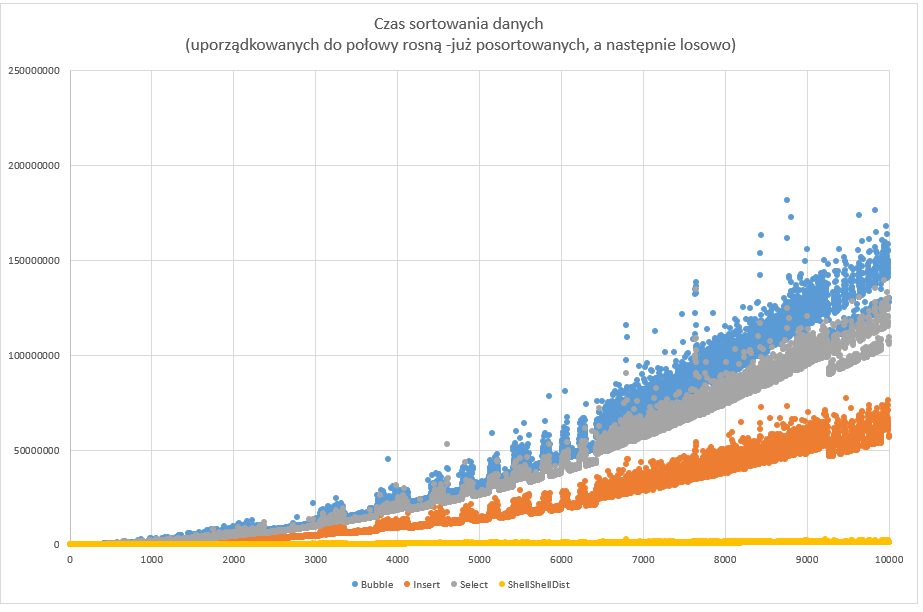
**Porównanie wydajności czasowej algorytmów:**

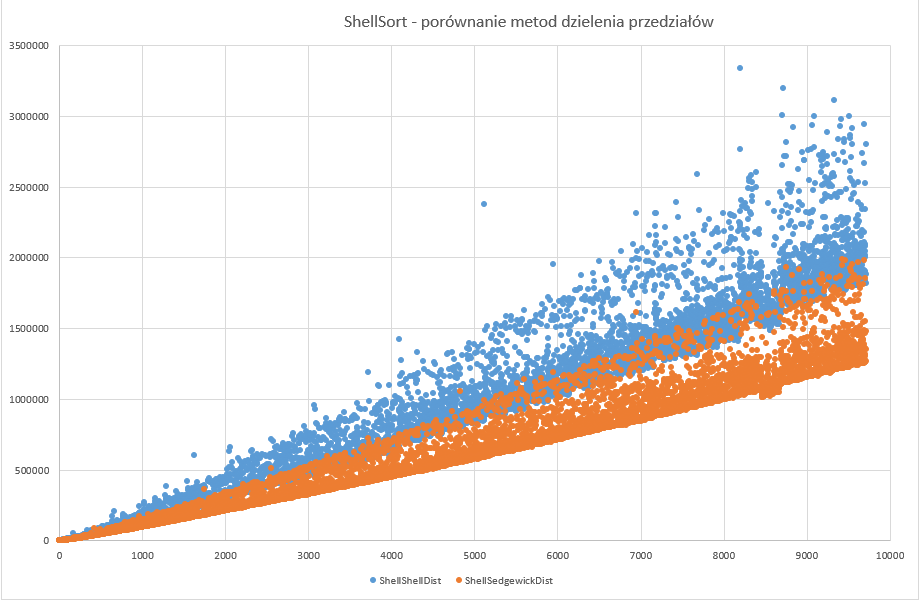
***Wykres1****: Dane losowe (pseudolosowe)*



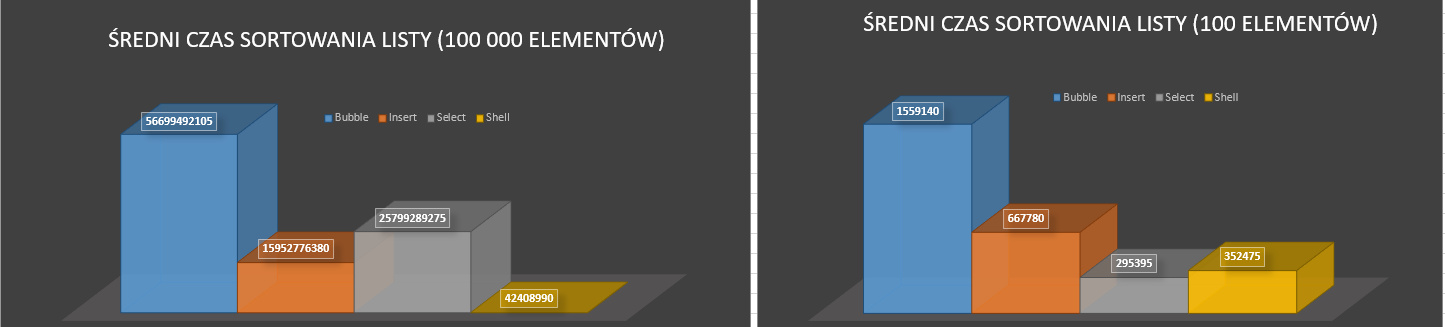
***Wykres2:*** *Dane uporządkowane rosnąco (już posortowane)*

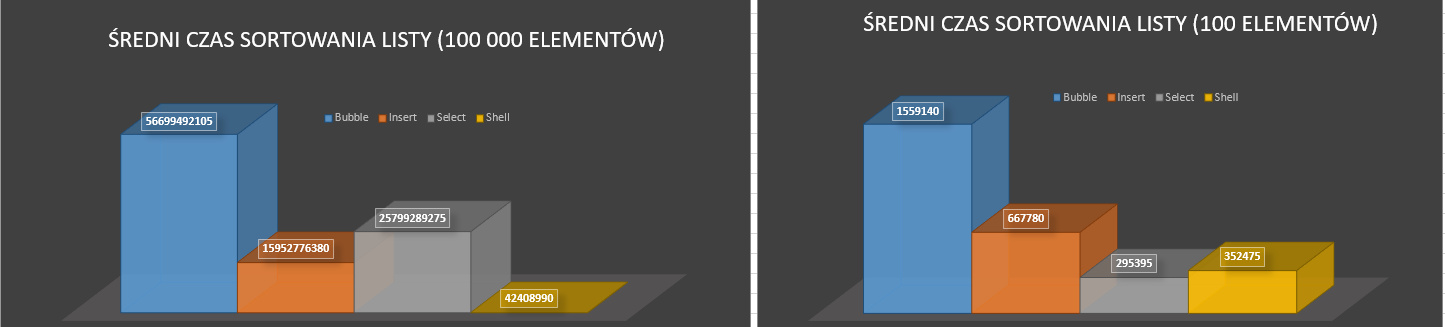
***Wykres3:*** *Dane uporządkowane malejąco (już posortowane)*

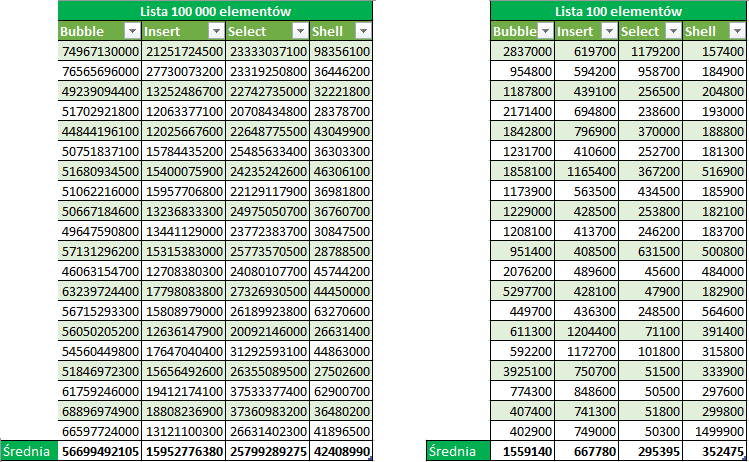
***Wykres4:*** *Dane uporządkowane do połowy rosnąco (już posortowane) a następnie losowo*



***Wykres5****: Porównanie efektywności sposobów dzielenia podziałów w implementacji algorytmu sortowania Shell’a*

**Tabelaryczne zestawienie wyników sortowania:**



****

**Uzasadnienie wyników i wnioski:**

Spośród wybranych implementacji sortowania, najlepiej radzi sobie *ShellSort*. Jak można się było tego spodziewać, najgorzej wypadł *BubbleSort* (jeden z najprostszych konceptualnie i implementacyjnie algorytmów). Spośród podanych dla dużych i średnich list najefektywniej jest wybrać implementację sortowania *Shella*, jednakże dla małych najlepiej poradzi sobie *insertSort* (mający przewagę przy małych listach nad *ShellSortem* w postaci stabilności sortowania i mniej skomplikowanych działań na strukturze danych wejściowych). *InsertSort* będzie również bardzo dobry wyborem dla sortowania danych już wstępnie już posortowany ShellSort pomimo swojej niskiej złożoności czasowej nie znalazł dużego zastosowania – został wyparty przez jeszcze szybszą implementację algorytmu sortującego – *quickSort*