Algorytmy i Złożoność Obliczeniowa

Prowadzący zajęcia: dr inż. Marcin Łopuszyński
Termin zajęć: Sroda godz. 15:15 tydzień nieparzysty

Autor sprawozdania:
- Kamil Wojcieszak 264487

Projekt 1:

Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową

1. Wstęp teoretyczny

Insertion sort

Insertion sort jest jednym z prostych algorytmów sortowania, który działa na zasadzie przeglądania elementów tablicy i wstawiania każdego z nich we właściwe miejsce w posortowanej części tablicy. Algorytm ten jest często używany do sortowania niewielkich zbiorów danych lub jako część bardziej złożonych algorytmów sortowania.

Kroki algorytmu Insertion sort:

Początek:

- Na początku przyjmujemy, że pierwszy element tablicy jest posortowany.
- Zakładamy, że tablica składa się z jednego posortowanego elementu, więc możemy rozpocząć od drugiego elementu.

Iteracja po tablicy:

- Iterujemy przez każdy element tablicy, zaczynając od drugiego.
- Dla każdego elementu, którym obecnie iterujemy, porównujemy go z elementami w posortowanej części tablicy.

Wstawianie elementu w odpowiednie miejsce:

- Jeśli napotkamy element mniejszy lub równy od obecnego elementu, przesuwamy ten element w prawo, aby zrobić miejsce dla obecnego elementu.
- Powtarzamy ten proces aż do momentu, gdy napotkamy element większy od obecnego lub dojdziemy do początku posortowanej części tablicy.

Wstawianie elementu:

- Po znalezieniu odpowiedniego miejsca dla obecnego elementu, wstawiamy go w to miejsce.

Kontynuacja:

- Powtarzamy powyższe kroki dla każdego kolejnego elementu tablicy, aż wszystkie elementy zostaną posortowane.

Koniec:

- Po zakończeniu iteracji przez wszystkie elementy tablicy, cała tablica zostanie posortowana.

Algorytm ten ma złożoność czasową $O(n^2)$, gdzie n to liczba elementów w tablicy. Insertion sort jest prosty i łatwy do zrozumienia, ale dla dużych zbiorów danych może być mniej wydajny od bardziej zaawansowanych algorytmów sortowania, takich jak quicksort czy mergesort. Jednakże, dla małych zbiorów danych lub w przypadku, gdy tablica jest już prawie posortowana, insertion sort może być skutecznym wyborem.

QuickSort

To algorytm sortowania dziel i zwyciężaj, który działa w średnim przypadku w czasie $O(nlogn)$, gdzie n to liczba elementów do posortowania. Niemniej jednak, w najgorszym przypadku, jego złożoność może wynosić $O(n^2)$, ale prawdopodobieństwo wystąpienia takiego przypadku jest niskie.

Algorytm Quicksort polega na podziale tablicy na mniejsze fragmenty oraz rekursywnym sortowaniu każdego z tych fragmentów. Kluczowym elementem Quicksort jest wybór elementu podziałowego, który często jest nazywany „pivotem". Ten element jest wykorzystywany do podziału tablicy na dwie części: jedną zawierającą elementy mniejsze lub równe pivotowi, a drugą zawierającą elementy większe od pivotu.

Kroki algorytmu Quicksort:

Wybór pivota: Wybiera się element z tablicy, który będzie służył jako pivot. Istnieje wiele metod wyboru pivota, ale najczęściej stosowaną jest metoda wyboru pierwszego, ostatniego lub środkowego elementu.

Podział tablicy: Tablica jest przeglądana, a elementy są porównywane z pivotem. Elementy mniejsze od pivota są umieszczane po lewej stronie, a większe po prawej. W wyniku tego kroku pivot znajduje się już na swojej prawidłowej pozycji w posortowanej tablicy, ale elementy po jego lewej stronie nie są jeszcze posortowane.

Rekursywne sortowanie podtablic: Proces sortowania jest rekursywnie powtarzany dla lewej i prawej podtablicy. Każda z tych podtablic jest sortowana w ten sam sposób poprzez wybór pivota, podział tablicy i rekursywne sortowanie mniejszych fragmentów.

Złączenie wyników: Po zakończeniu sortowania wszystkich podtablic, wyniki są łączone w celu uzyskania całkowicie posortowanej tablicy.

Wybór pivota może znacząco wpłynąć na wydajność algorytmu Quicksort. W najgorszym przypadku, gdy pivot jest wybierany zawsze jako najmniejszy lub największy element, sortowanie może osiągnąć złożoność $O(n^2)$, szczególnie gdy tablica jest już posortowana lub prawie posortowana. Jednakże, jeśli pivot jest wybierany w sposób umiejętny, na przykład jako losowy element lub jako element środkowy, to algorytm osiąga oczekiwaną złożoność $O(n\,log(n))$ w średnim przypadku.

Heap sort

Heap sort jest algorytmem sortowania, który operuje na strukturze danych zwanej kopcem. Jest to algorytm efektywny i stabilny, który działa w czasie $O(n\,logn)$, co oznacza, że jego złożoność czasowa rośnie w sposób logarytmiczny wraz ze wzrostem liczby elementów do posortowania.

Kroki działania algorytmu heap sort:

1. Budowa kopca

- Pierwszym krokiem w algorytmie jest utworzenie kopca z listy elementów do posortowania. Kopiec jest drzewem binarnym spełniającym pewne właściwości. W przypadku kopca używanego w heap sort, jest to tzw. kopiec binarny maksymalny.

- Kopiec binarny maksymalny to drzewo binarne, w którym dla każdego węzła rodzica jego wartość jest większa lub równa wartości jego dzieci. Wartości te są uporządkowane w taki sposób, że korzeń drzewa zawiera największą wartość.

- Aby utworzyć kopiec, można wykorzystać procedurę zwanej "heapify", która naprawia właściwości kopca w przypadku jego naruszenia.

2. Sortowanie

- Po zbudowaniu kopca, korzeń drzewa zawiera największą wartość w zbiorze danych. Zamieniamy wartość korzenia z ostatnim elementem w zbiorze, aby przenieść największy element na jego właściwe miejsce na końcu zbioru.

- Następnie zmniejszamy rozmiar kopca o 1 (ignorując ostatni, posortowany element) i stosujemy operację naprawy kopca (heapify) na pozostałych elementach, aby przywrócić właściwości kopca.

- Powtarzamy ten proces, aż kopiec będzie miał tylko jeden element. W takiej sytuacji cała tablica zostanie posortowana.

Shell sort

Shell Sort, to algorytm sortowania, który jest ulepszeniem sortowania przez wstawianie. Został zaproponowany przez Donalda Shella w 1959 roku. Jest to algorytm in-place, co oznacza, że operuje na danych bez konieczności tworzenia dodatkowych struktur danych. Shell Sort łączy strategię podziału danych z metodą sortowania przez wstawianie.

Kroki algorytmu Shell Sort:

1. Określenie sekwencji dzielącej: Pierwszym krokiem algorytmu Shell Sort jest określenie sekwencji dzielącej, która jest używana do podziału listy na podlisty. Zazwyczaj stosuje się sekwencje malejące, które pozwalają na coraz to mniejsze podziały. Najpopularniejszą sekwencją jest sekwencja dzieląca w postaci $\{ n/2^k \}$, gdzie n to liczba elementów na liście, a k to liczba kroków.

2. Sortowanie podlist: Następnie algorytm Shell Sort sortuje elementy w każdej podliście za pomocą sortowania przez wstawianie. Pierwsza podlista obejmuje co drugi element, druga podlista obejmuje elementy oddalone o $n/2^k$, trzecia podlista obejmuje elementy oddalone o $n/2^{k+1}$ itd., gdzie k to liczba kroków w sekwencji dzielącej.

3. Skalowanie sekwencji dzielącej: Po każdym przejściu przez listę, sekwencja dzieląca jest skalowana, czyli zmniejszana, aby przyspieszyć zbliżanie się elementów do ich właściwych pozycji.

4. Finalne sortowanie przez wstawianie: Ostatecznie, gdy sekwencja dzieląca zawiera tylko jeden krok, sortowanie przypomina bardziej zwykłe sortowanie przez wstawianie, ale dzięki wstępnemu posortowaniu podlist przez poprzednie kroki, elementy znajdują się już w pobliżu swojej prawidłowej pozycji, co przyspiesza ten proces.

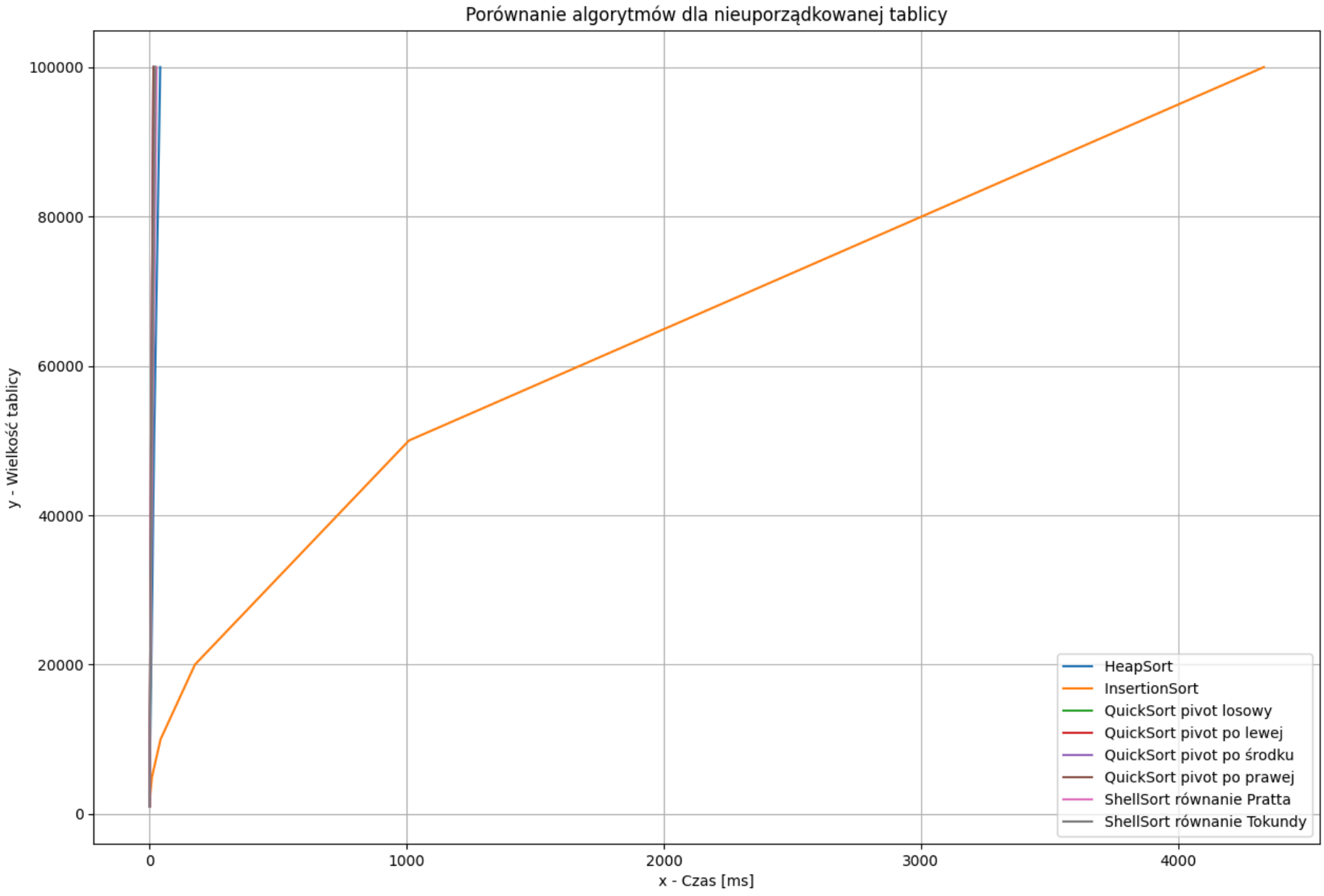
Oto kluczowe cechy Shell Sort:

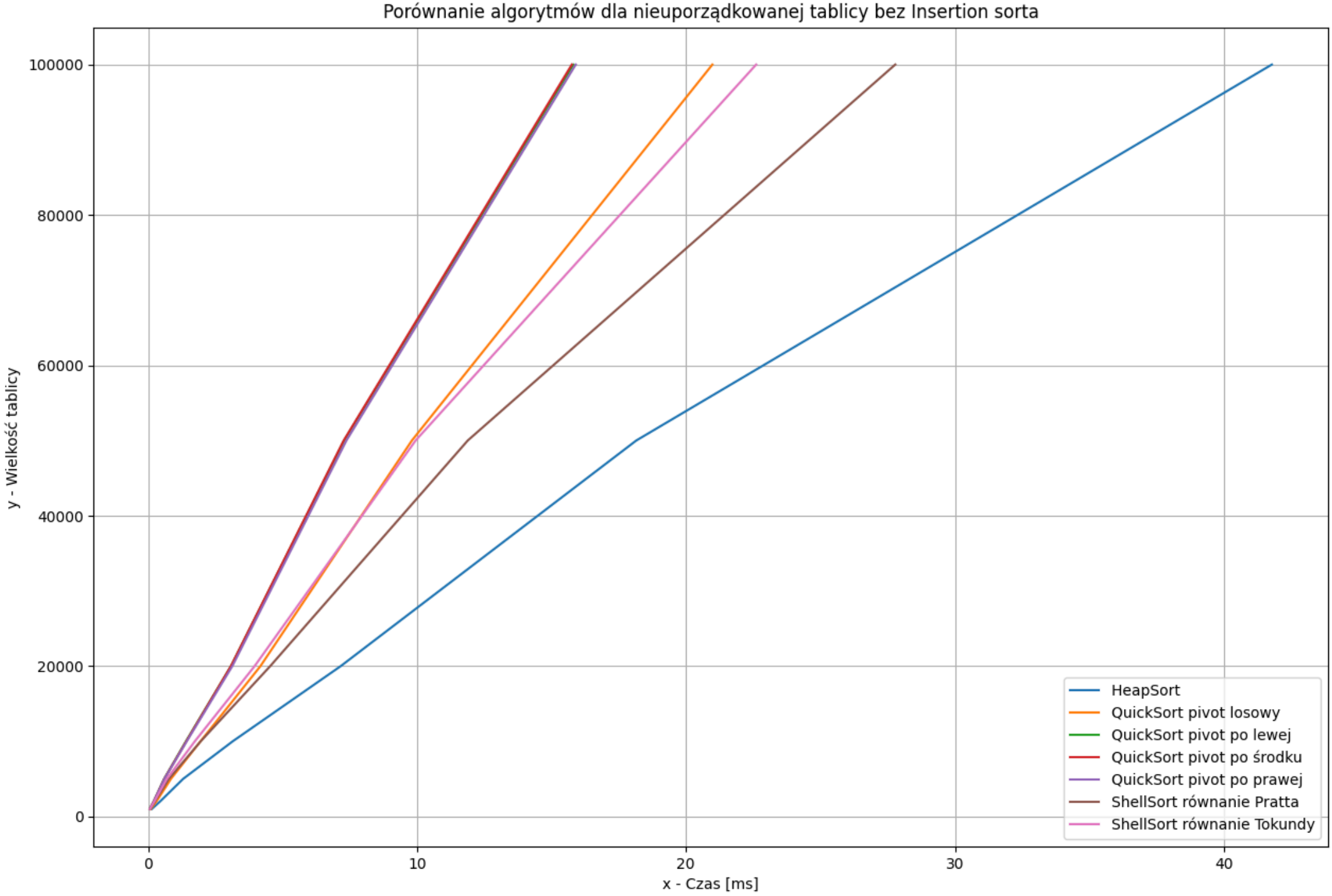
Złożoność czasowa algorytmu Shell sort jest trudna do teoretycznego oszacowania, ponieważ zależy od sekwencji dzielącej. Jednakże, w praktyce jest to często wybierany algorytm sortowania dla małych i średnich zbiorów danych.

2. Plan eksperymentu.

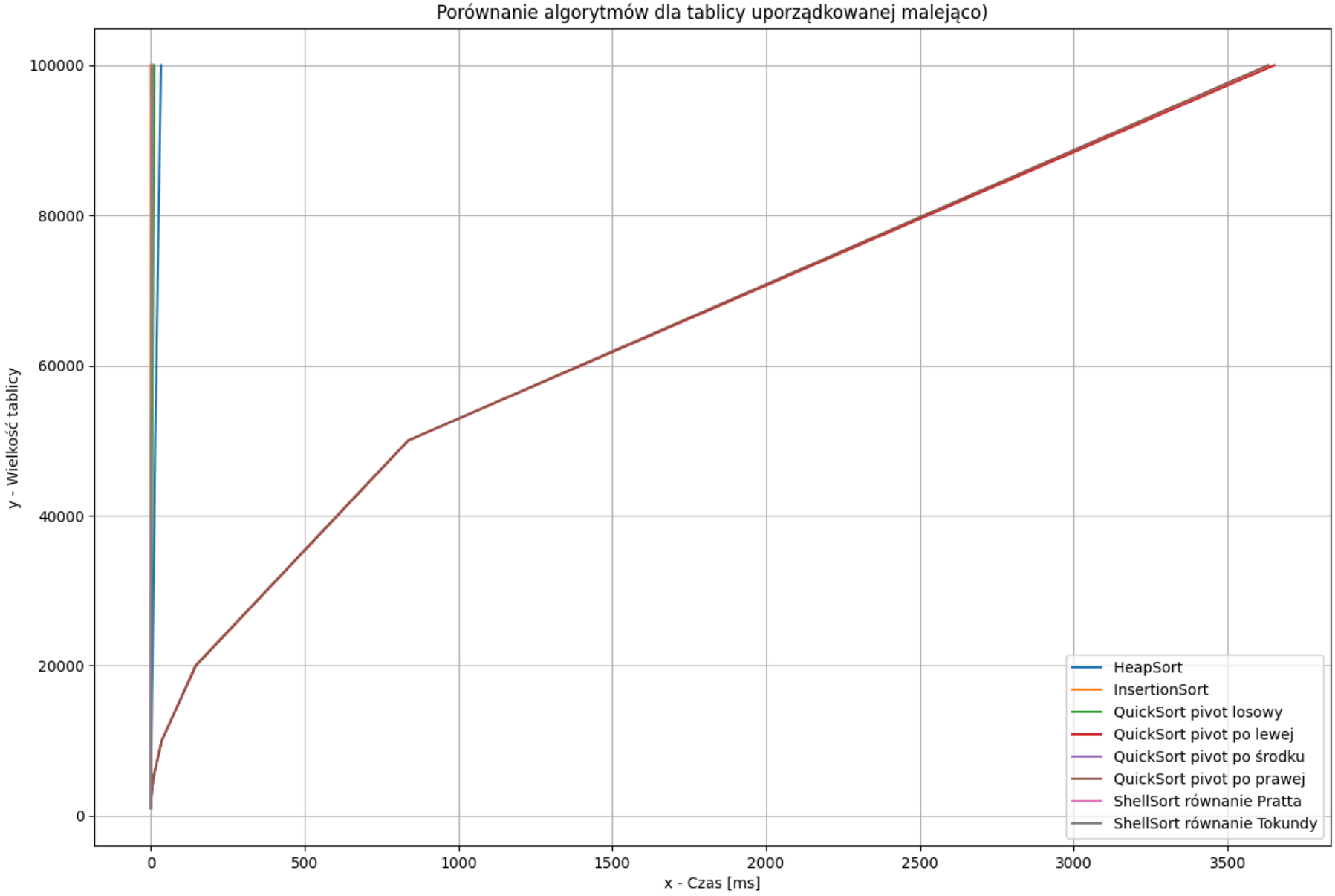
Analiza efektywności działania algorytmu została przeprowadzana dla 7, różnej wielkości tablicy (1000, 2000, 5000, 10000, 20000, 50000, 100000). Dla każdego z zadanych wielkości został wykonany algorytm i policzony czas wykonania. Wszystkie wyniki zostały zaprezentowane na wykresach poniżej.

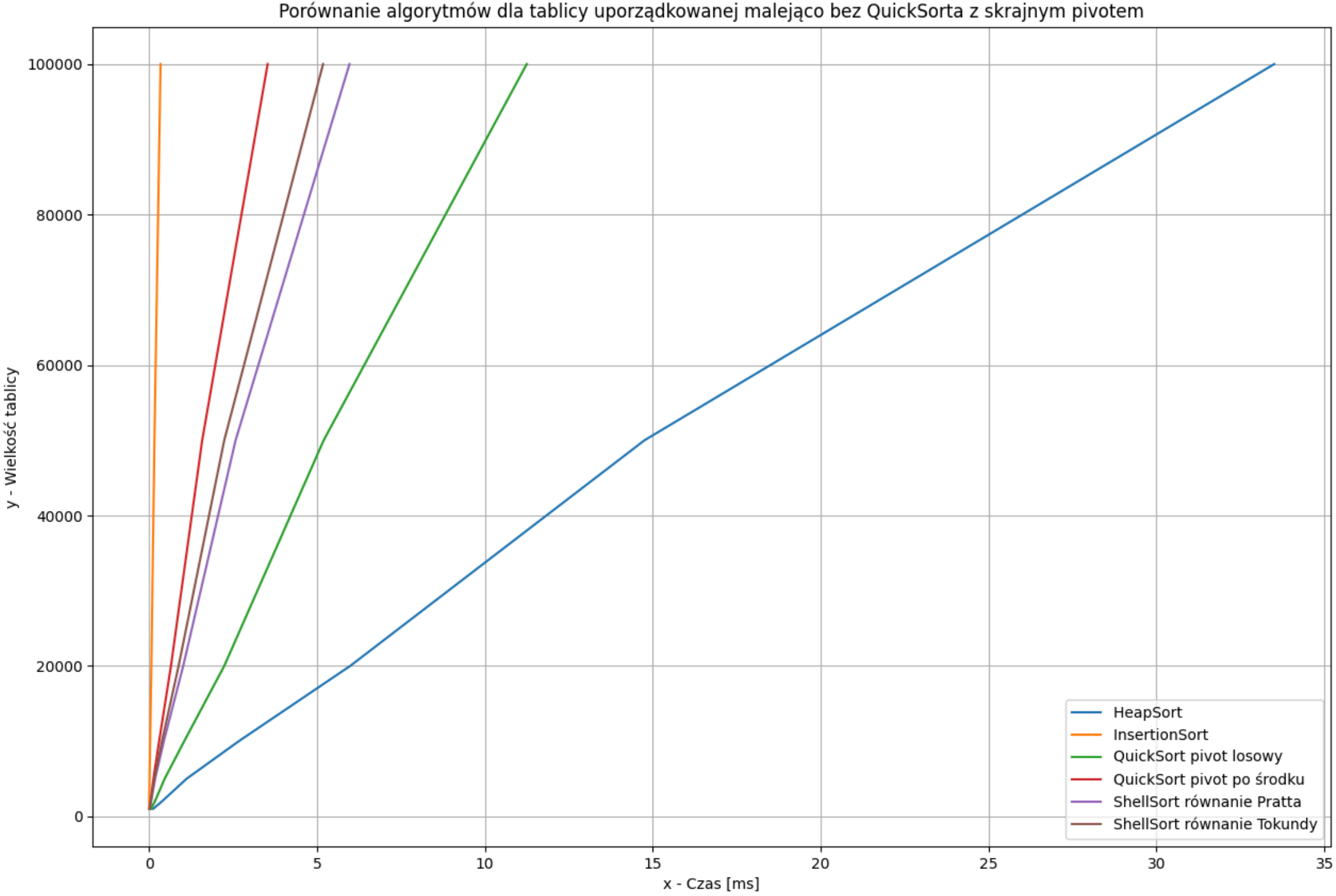
Eksperyment 1 - badanie czasu wykonania dla nieuporządkowanej tablicy



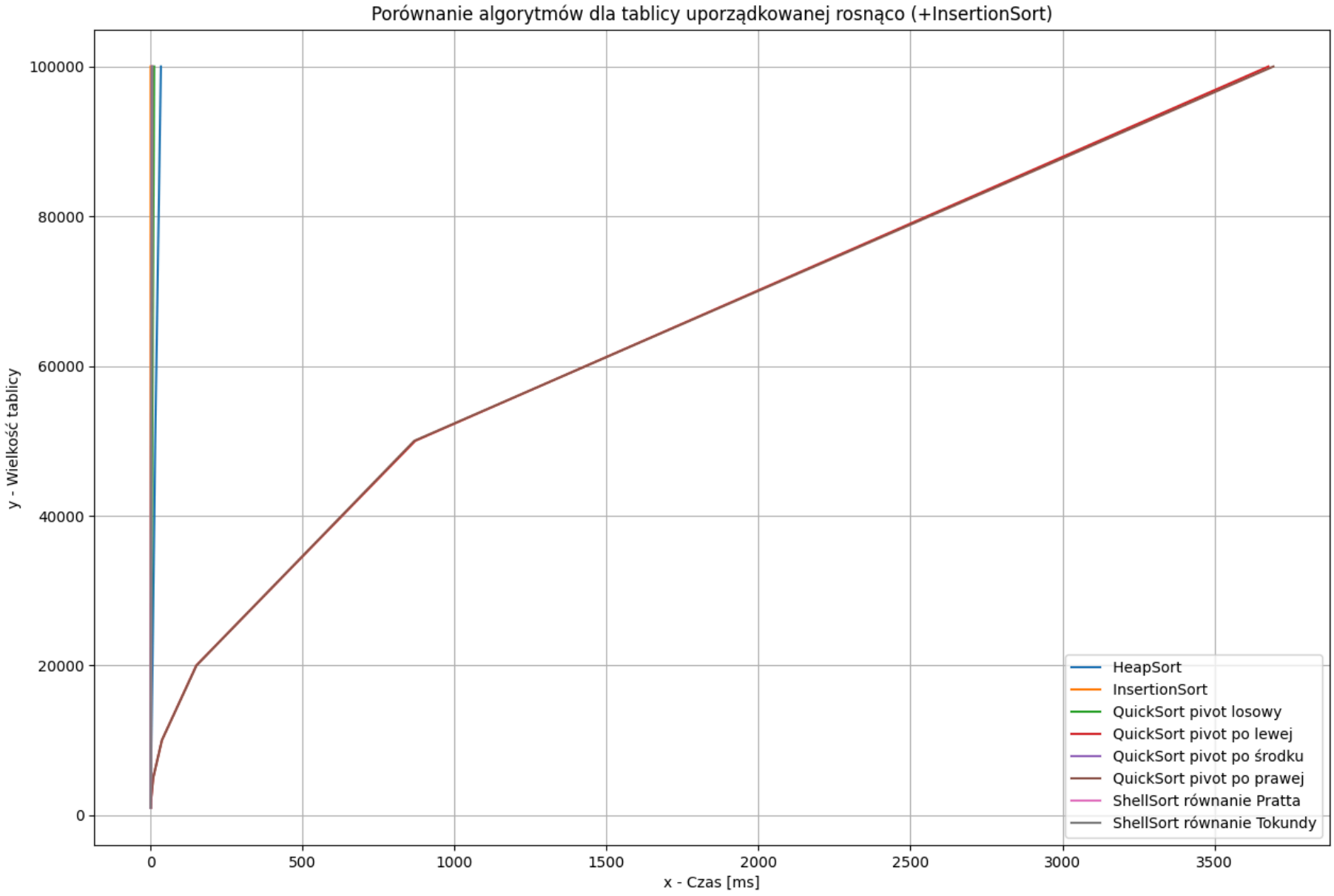

Porównanie algorytmów dla nieuporządkowanej tablicy bez Insertion sorta
y - Wielkość tablicy
x - Czas [ms]
0
20000
40000
60000
80000
100000
0
10
20
30
40
HeapSort
QuickSort pivot losowy
QuickSort pivot po lewej
QuickSort pivot po środku
QuickSort pivot po prawej
ShellSort równanie Pratta
ShellSort równanie Tokundy

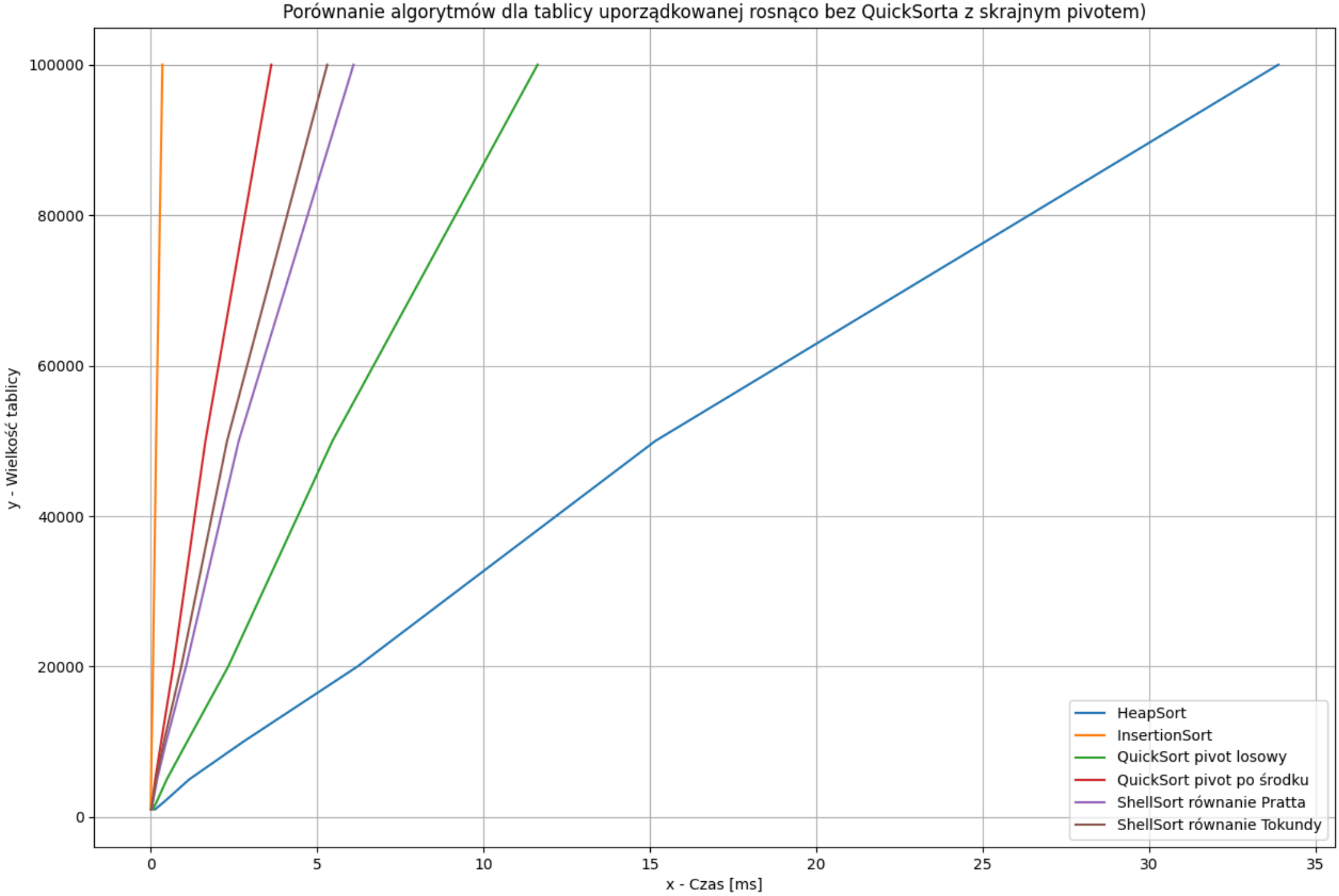
Eksperyment 2 - badanie czasu wykonania dla tablicy uporządkowanej malejąco



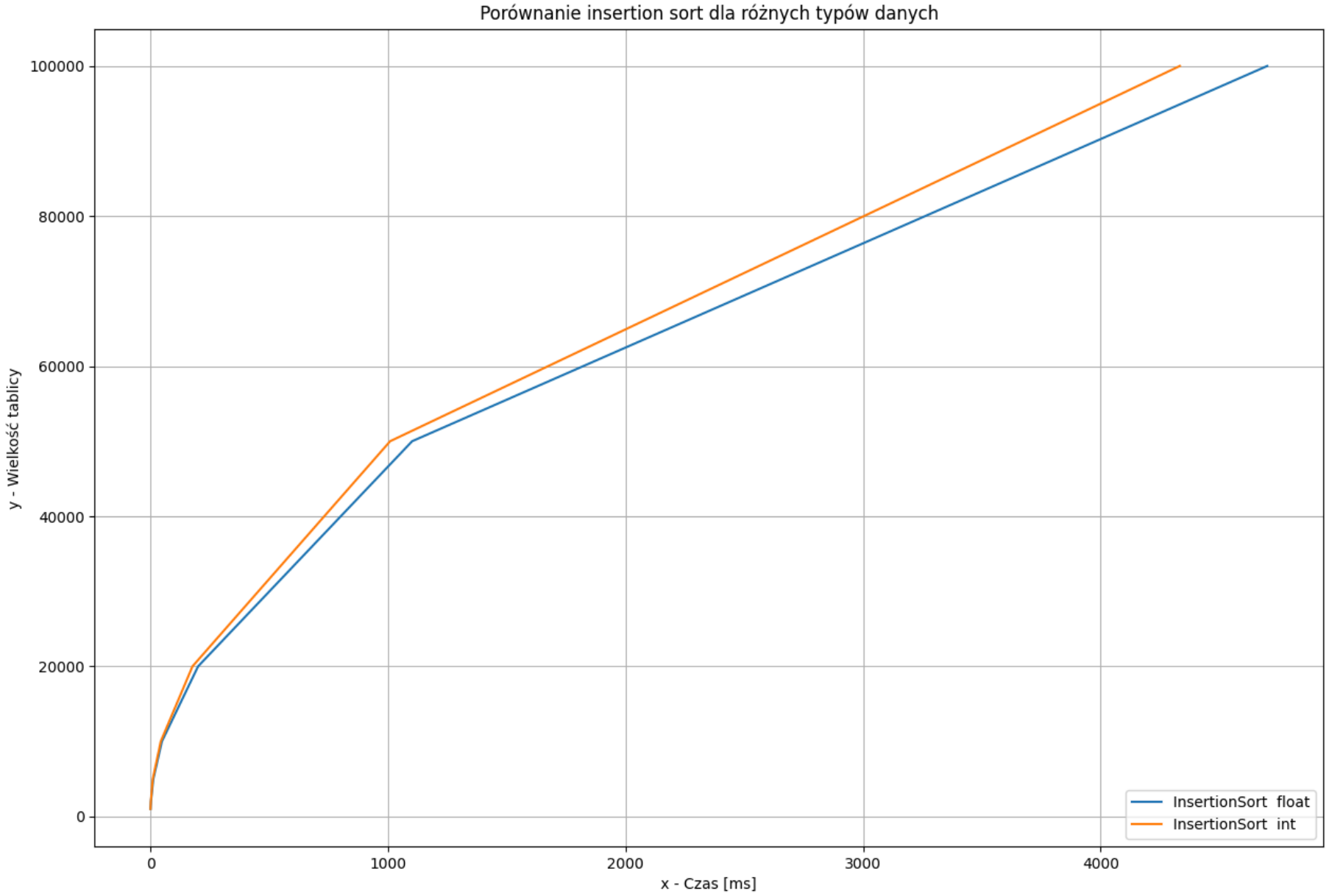

Porównanie algorytmów dla tablicy uporządkowanej malejąco bez QuickSorta z skrajnym pivotem
y - Wielkość tablicy
x - Czas [ms]
0
20000
40000
60000
80000
100000
0
5
10
15
20
25
30
35
HeapSort
InsertionSort
QuickSort pivot losowy
QuickSort pivot po środku
ShellSort równanie Pratta
ShellSort równanie Tokundy

Eksperyment 3 - badanie czasu wykonania dla tablicy uporządkowanej rosnąco




Porównanie algorytmów dla tablicy uporządkowanej rosnąco bez QuickSorta z skrajnym pivotem)
y - Wielkość tablicy
x - Czas [ms]
0
20000
40000
60000
80000
100000
0
5
10
15
20
25
30
35
HeapSort
InsertionSort
QuickSort pivot losowy
QuickSort pivot po środku
ShellSort równanie Pratta
ShellSort równanie Tokundy

Eksperyment 4 - badanie czasu wykonania dla różnych typów danych



Eksperyment 1 wnioski

Dla nieposortowanej tablicy zdecydowanie najgorszym algorytmem okazał się Insertion sort. Insertion sort jest algorytmem o złożoności czasowej O(n2), co oznacza, że jego czas działania rośnie kwadratowo wraz ze wzrostem liczby elementów w tablicy. Dla większych zbiorów danych, gdzie liczba elementów jest znacząca, czas wykonania insertion sort może być znacząco dłuższy niż dla innych algorytmów sortowania, takich jak quicksort, heapsort czy shell sort, które mają lepszą złożoność czasową. Najlepszym algorytmem okazał się quicksort z sztywno ustalonym pivotem.

Eksperyment 2 i 3 wnioski

Wyniki insertion sort były zdecydowanie najlepsze. Kiedy tablica jest już uporządkowana, insertion sort daje lepsze rezultaty niż inne algorytmy.

2 powody dla których insertion sort działa dobrze w przypadku tablicy uporządkowanej:

Mała liczba zamian: W przypadku tablicy już posortowanej, wstawianie nowych elementów na odpowiednie miejsce w posortowanej części tablicy (w tym przypadku od największego do najmniejszego) wymaga tylko niewielkiej liczby zamian, co jest efektywne dla insertion sort.

Niski koszt operacji: W przypadku uporządkowanej tablicy, każdy nowy element, który dodajemy, będzie mniejszy bądź większy (w zależności od uporządkowania) od poprzedniego. W związku z tym, gdy insertion sort porównuje nowy element z elementami w posortowanej części, może łatwo znaleźć właściwe miejsce do wstawienia bez konieczności wykonywania wielu zamian.

Z drugiej strony quicksort z pivotem ustawionym na skrajnie lewo lub prawo działał najwolniej.

2 powody dlaczego quicksort działa gorzej w przypadku tablicy uporządkowanej malejąco:

Wybór pivota: W quicksort, wybór pivota ma duże znaczenie dla wydajności algorytmu. W przypadku tablicy posortowanej, jeśli pivota wybieramy jako skrajnie lewy lub prawy element, to będziemy mieć podział na jedną dużą grupę i jedną pustą, co prowadzi do najgorszego przypadku dla quicksort.

Liczba rekurencji: Quick sort w najgorszym przypadku może mieć złożoność czasową $O(n^2)$, gdy pivota wybieramy niefortunnie. W przypadku tablicy posortowanej malejąco, ten niefortunny wybór pivota może prowadzić do licznych rekurencji i długiego czasu wykonania.

Eksperyment 4 wnioski

Wydajność algorytmu insertion sort dla danych typu int w porównaniu z danymi typu float może być szybsza z kilku powodów. Różnica w szybkości leży najprawdopodobniej w szybkości porównania danych do siebie. Porównania liczb całkowitych są prostsze i szybsze niż porównania liczb zmiennoprzecinkowych, które wymagają dodatkowych kroków do porównania bitów mantysy i eksponenty.