Projekt 2 - sortowanie

Aleksander Dygon - 151856

Wstęp

Celem tego projektu jest implementacja, przetestowanie i porównanie różnych metod sortowania tablic. W ramach projektu zostaną zbadane dwie grupy metod sortowania: I grupa metod - metody podstawowe (przez wstawianie, przez selekcję, sortowanie bąbelkowe) oraz II grupa metod - bardziej zaawansowane techniki sortowania (Quicksort, Sortowanie Shella, Sortowanie przez kopcowanie). Testy zostaną przeprowadzone na tablicach zawierających liczby całkowite z przedziału od -100 do 100.

Metody sortowania

I grupa metod

- Przez wstawianie
  Pseudokod:

```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

- Przez selekcję
  Pseudokod:

```
SELECTION-SORT(A)
    for i = 1 to A.length - 1
        minIndex = i
        for j = i + 1 to A.length
            if A[j] < A[minIndex]
                minIndex = j
        swap A[i] with A[minIndex]
```

- Sortowanie bąbelkowe
  Pseudokod:

```
BUBBLE-SORT(A)
    for i = 1 to A.length - 1
        for j = A.length downto i + 1
            if A[j] < A[j - 1]
                swap A[j] with A[j - 1]
```

II grupa metod

- Quicksort
  Pseudokod:

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] <= x
            i = i + 1
            swap A[i] with A[j]
    swap A[i + 1] with A[r]
    return i + 1
```

- Sortowanie Shella
  Pseudokod:

```
SHELL-SORT(A)
    n = A.length
    gap = n / 2
    while gap > 0
        for i = gap to n - 1
            temp = A[i]
            j = i
            while j >= gap and A[j - gap] > temp
                A[j] = A[j - gap]
                j = j - gap
            A[j] = temp
        gap = gap / 2
```

- Sortowanie przez kopcowanie
  Pseudokod:

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        swap A[1] with A[i]
        A.heapSize = A.heapSize - 1
        MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A)
    A.heapSize = A.length
    for i = floor(A.length / 2) downto 1
        MAX-HEAPIFY(A, i)

MAX-HEAPIFY(A, i)
    left = 2i
    right = 2i + 1
    if left <= A.heapSize and A[left] > A[i]
        largest = left
    else largest = i
```

```
if right <= A.heapSize and A[right] > A[largest]
    largest = right
if largest != i
    swap A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

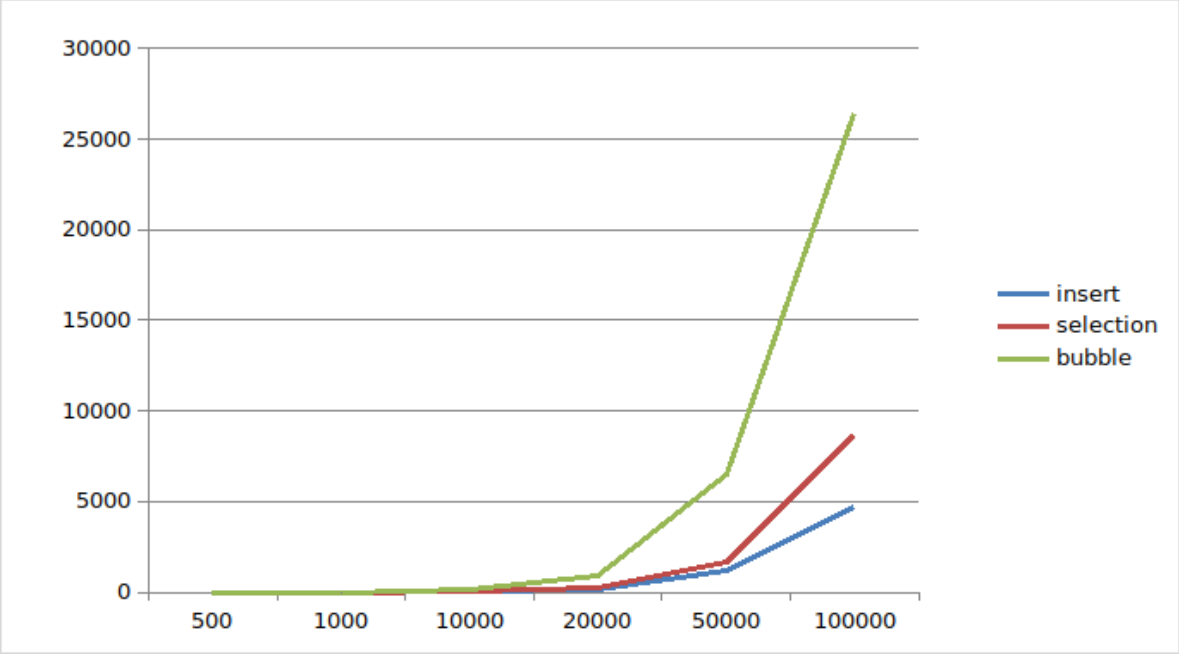
Testy sortowania

Sortowanie będzie testowane na trzech rodzajach danych wejściowych:

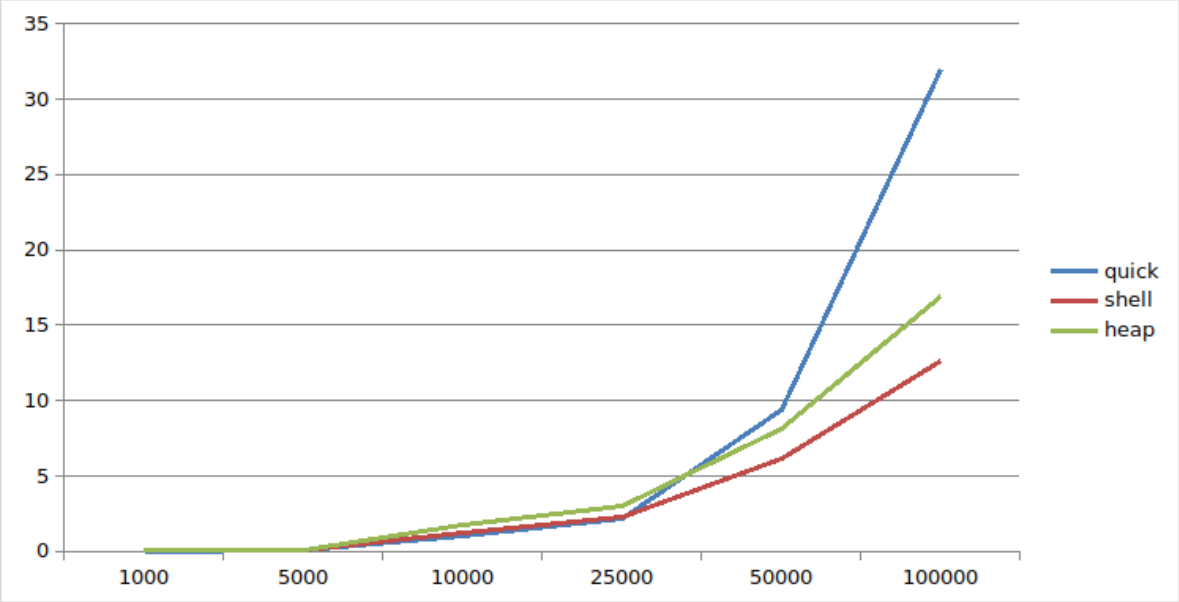
1. Dla danych wygenerowanych losowo
2. Dla danych posortowanych w kolejności odwrotnej (malejąco)
3. Dla danych posortowanych właściwie (rosnąco)

Dane wygenerowane losowo

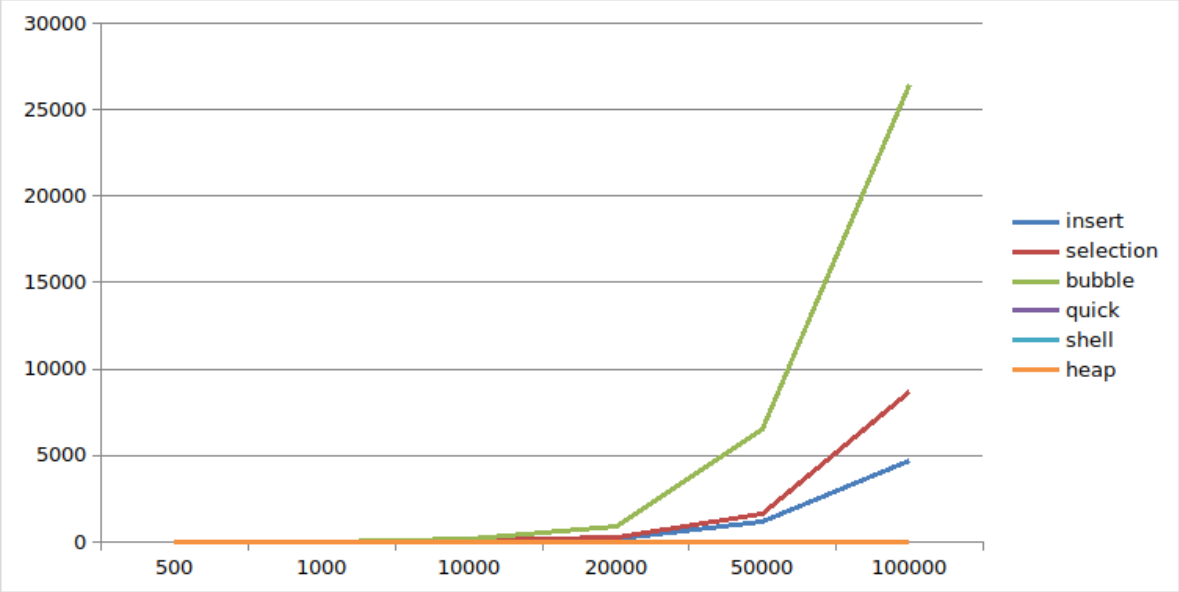
Wykres metod z grupy pierwszej:



Wykres metod z grupy drugiej:

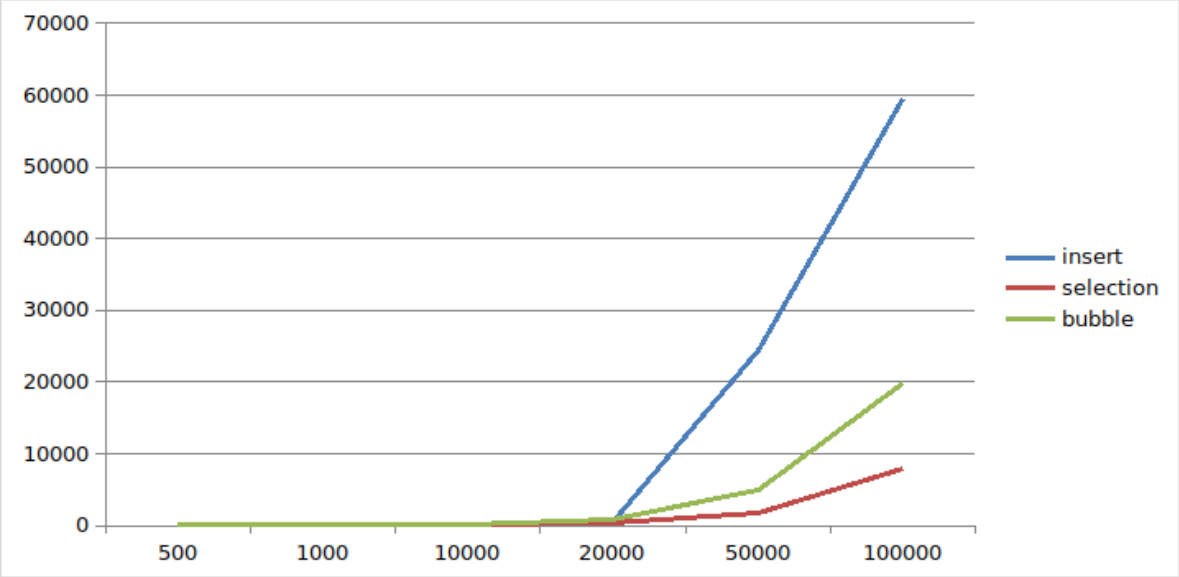


Wspólny wykres

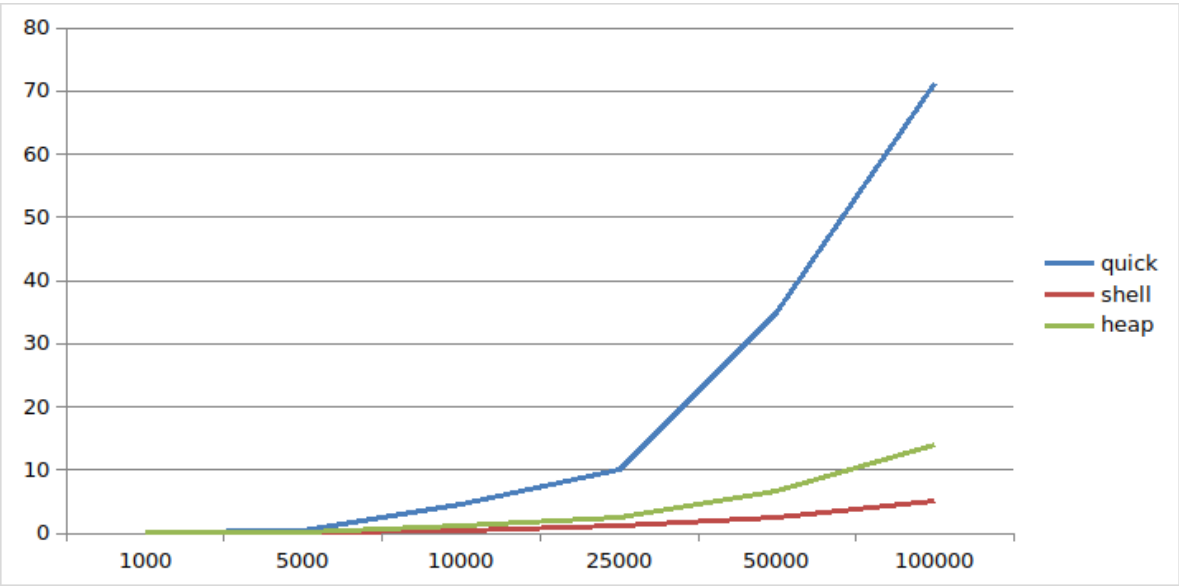


Dla danych posortowanych w kolejności odwrotnej (malejąco)

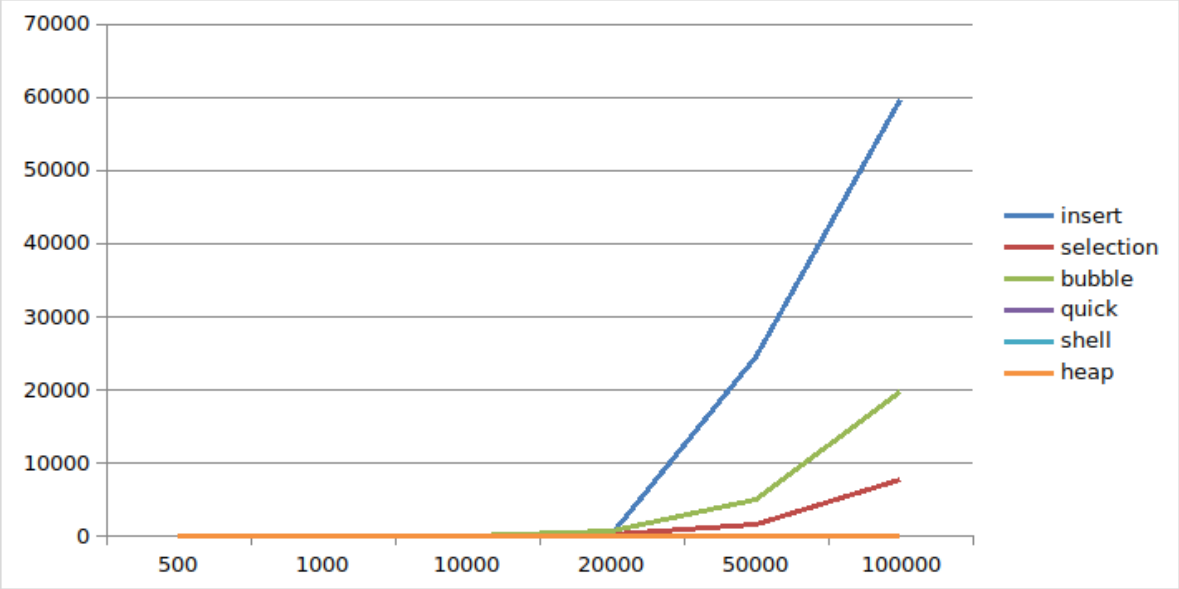
Wykres metod z grupy pierwszej:



Wykres metod z grupy drugiej:

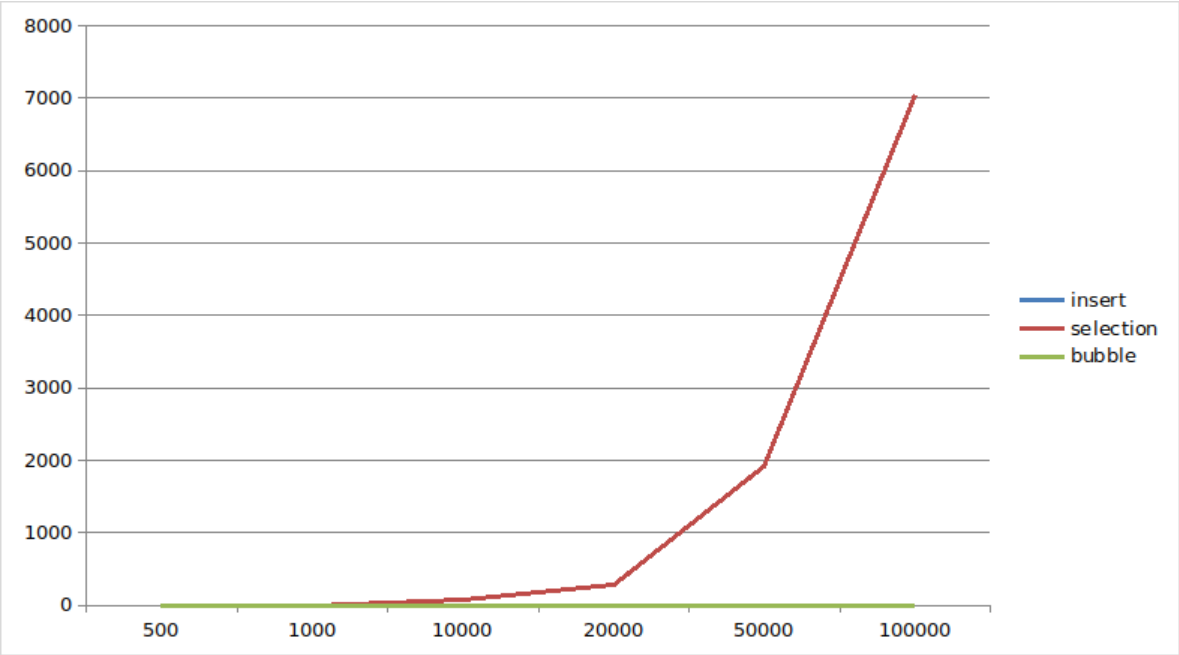


Wspólny wykres

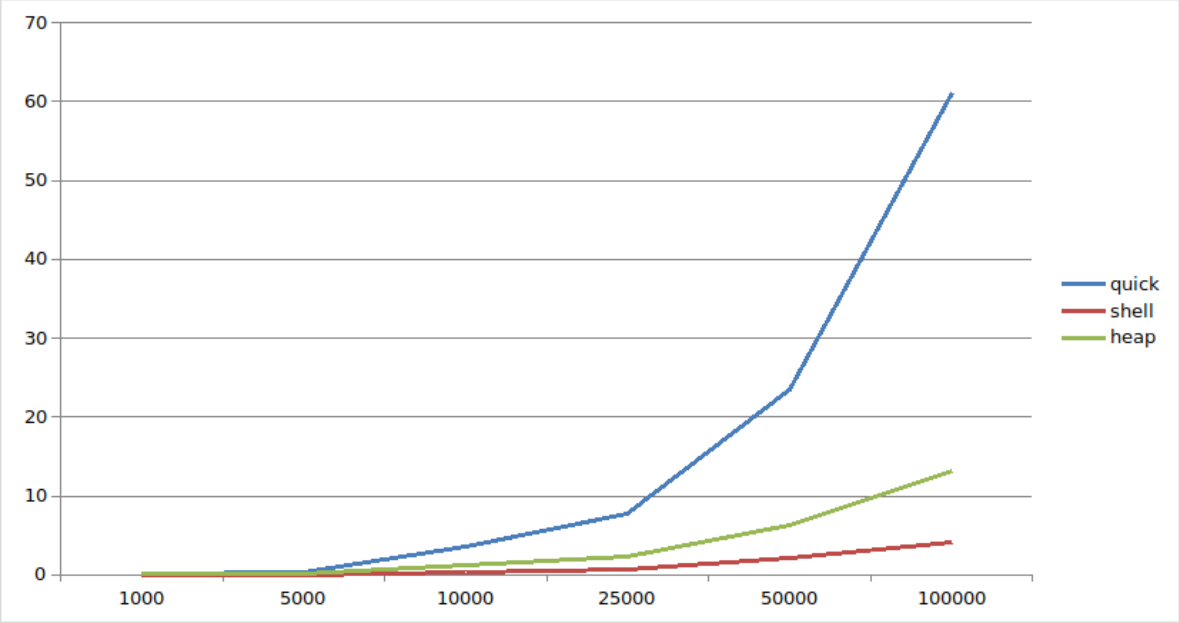


Dla danych posortowanych właściwie (rosnąco)

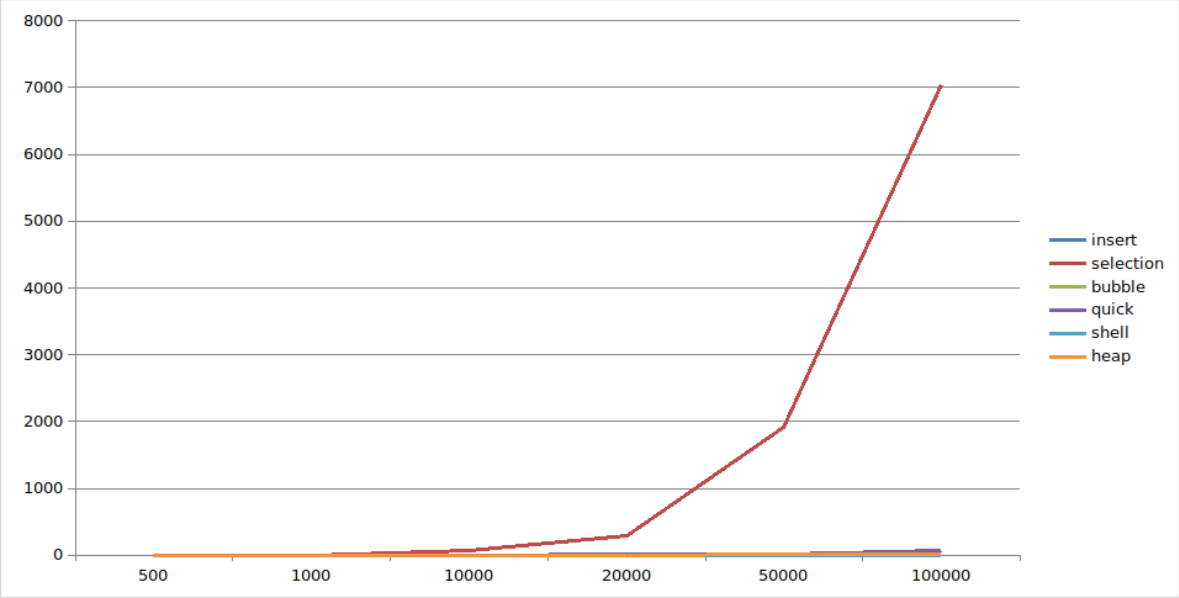
Wykres metod z grupy pierwszej:



Wykres metod z grupy drugiej:



Wspólny wykres



Podsumowanie

Algorytmy o złożoności $O(n^2)$ (INSERT, SELECTION, BUBBLE) są wyraźnie nieefektywne dla dużych zbiorów danych, co widać po gwałtownym wzroście czasu wykonania.
Quick Sort i Heap Sort są najbardziej efektywnymi algorytmami w analizie, szczególnie przy dużych zbiorach danych.
Shell Sort oferuje dobrą wydajność i może być preferowanym wyborem w przypadkach, gdzie stabilność czasu wykonania jest istotna, choć nie dorównuje efektywnością Quick Sort i Heap Sort.
Bubble Sort należy unikać w przypadku jakichkolwiek większych zbiorów danych z uwagi na ekstremalnie wysokie czasy wykonania.