

Helena Masłowska gr. 5.1  
indeks: 148182  
2 sem. Informatyka

---

# **Opracowanie algorytmów sortowania**

---

link do repozytorium:  
[https://github.com/HelenaMaslowska/Opracowanie\\_algorytmow\\_sortowania](https://github.com/HelenaMaslowska/Opracowanie_algorytmow_sortowania)

## **Spis treści**

1. Wstęp
2. Typy sortowań:
  - a. Naiwny
    - i. Bubble Sort
    - ii. Selection Sort
  - b. "Dziel i zwyciężaj"
    - i. Quick Sort
    - ii. Merge Sort
  - c. Inne algorytmy sortowania
    - i. Counting Sort
3. Wnioski i zestawienie algorytmów

## **Wstęp**

Celem niniejszego sprawozdania jest zaimplementowanie algorytmów sortowania w języku programowania Python i sprawdzenie efektywności wybranych algorytmów sortowania w zależności od rodzaju i liczby danych wejściowych. Zostaną porównane algorytmy takie jak Bubble Sort, Selection Sort, Quick Sort, Merge Sort, Counting Sort, a dane wejściowe zawierają od 10 do 10000 liczb. Testy zostały powtórzone 15 razy, aby móc bardziej oddanie wyrazić efektywność działania algorytmów.

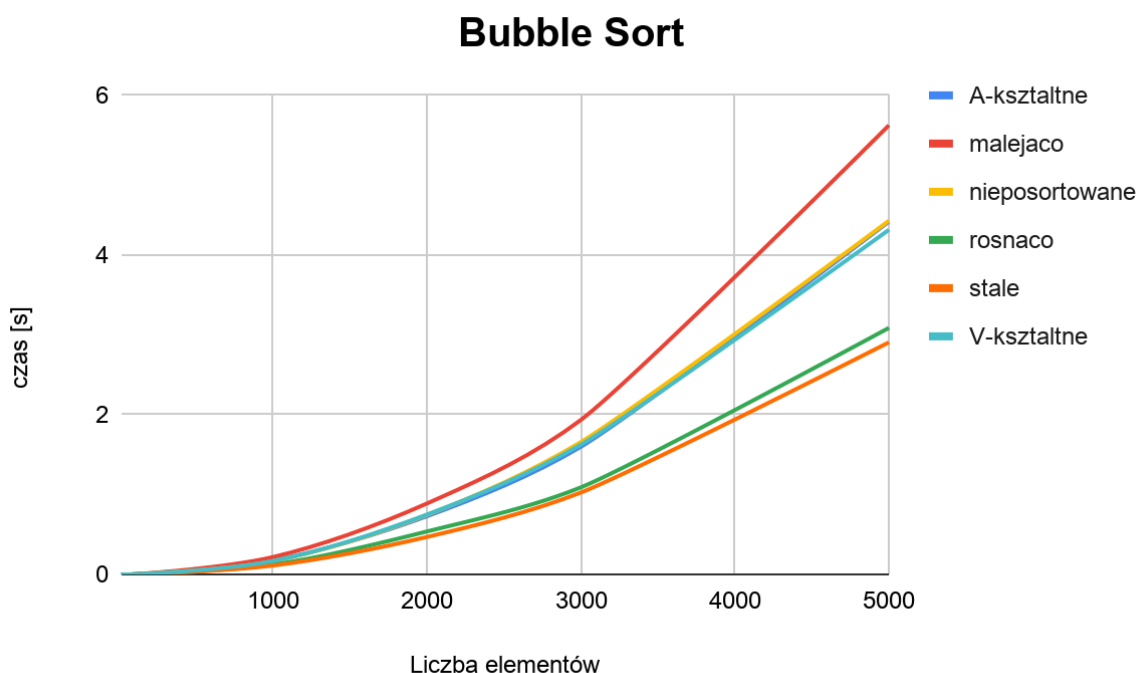
## Bubble Sort

In situ – w miejscu	TAK
Stabilny	TAK
Usytuowanie wśród podobnych algorytmów	Stabilny, in situ, najprostszy w zrozumieniu i implementacji Duża złożoność

### Rozwiązanie

Sprawdzamy 2 elementy obok siebie, z lewej stawiamy wartość mniejszą, z prawej większą, równych sobie wartości nie zmieniamy. Dodajemy indeks +1 i sprawdzamy dwie kolejne wartości. Powtarzamy poprzednie kroki  $n-1$  razy, gdzie  $n$  to wielkość tablicy.

**Zależność między ilością danych wejściowych a efektywnością algorytmu**



### Obserwacje i wnioski

Algorytm posortował elementy w dosyć długim czasie, jednak można zauważyć że dla danych stałych i rosnących algorytm zadziałał najszybciej.

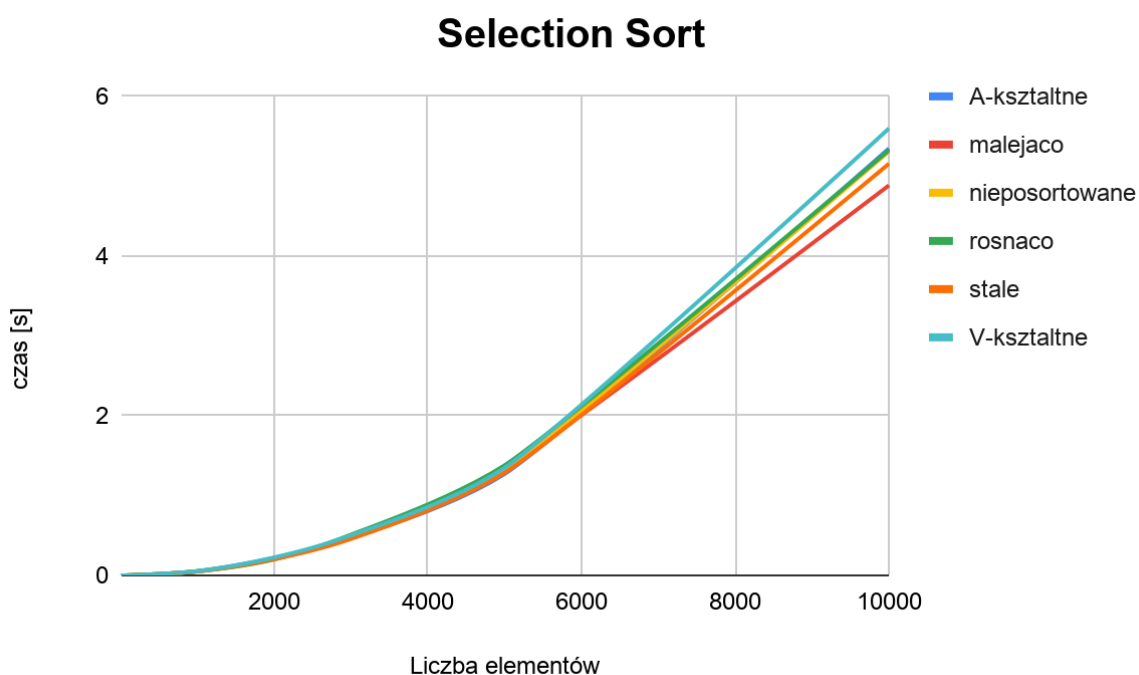
## Selection Sort

In situ – w miejscu	TAK
Stabilny	NIE
Usytuowanie wśród podobnych algorytmów	Łatwy w implementacji, in situ Złożoność obliczeniowa niezmienna Niestabilny – może się zmienić kolejność danych o takiej samej wartości, duża złożoność

### Rozwiązanie

Wybieramy najmniejszy element i ustawiamy go na prawidłowe miejsce (na początku na pierwsze, gdyż jest to najmniejszy element), następnie szukamy kolejnego najmniejszego elementu i ustawiamy go za pierwszym, już ustawionym, aż do uzyskania posortowania całej tablicy.

### Zależność między ilością danych wejściowych a efektywnością algorytmu



### Obserwacje i wnioski

Algorytm niezależnie od danych posortował elementy w dosyć długim czasie, ale krótszym od Bubble Sort, jednak niezależnie od wprowadzonych danych wejściowych.

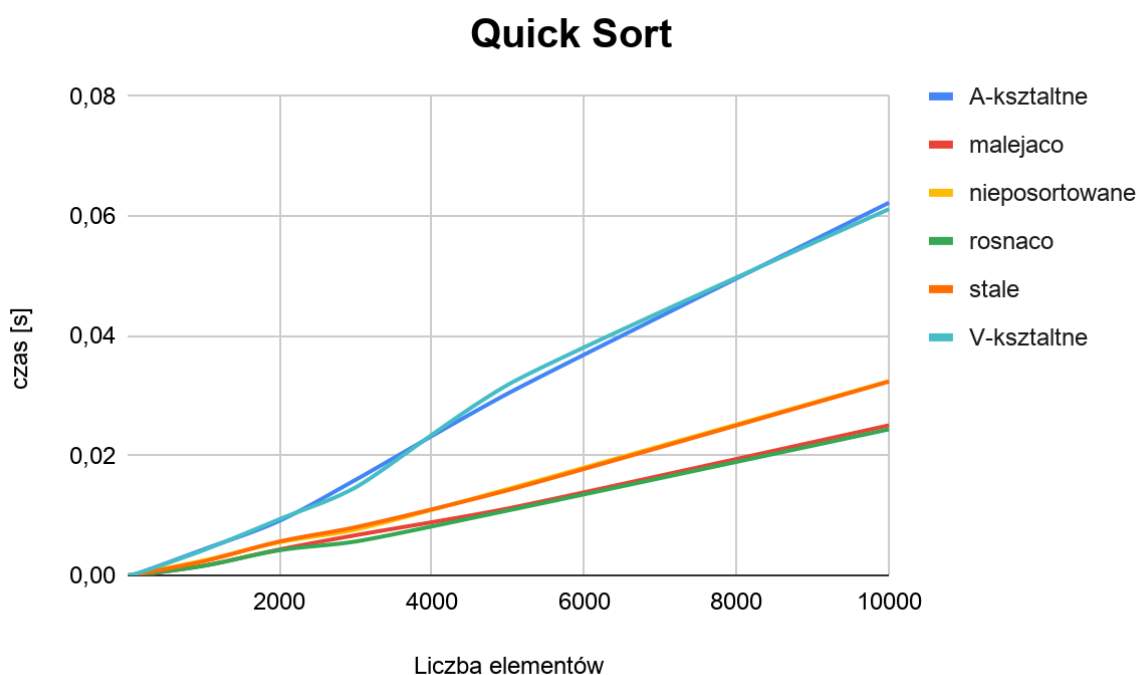
## Quick Sort

In situ – w miejscu	NIE
Stabilny	TAK
Usytuowanie wśród podobnych algorytmów	Stabilny, szybki dla różnych rozmiarów tablicy Rekurencja może spowodować przepełnienie stosu, największy problem stanowią tablice powyżej 10000 elementów

### Rozwiązanie

Najpierw wybieramy losowy element z tablicy P, który posłuży nam jako pivot, inaczej punkt odniesienia. Po lewej stronie muszą znaleźć się wszystkie elementy nie większe od P, a zaś po prawej pozostałe elementy, które są nie mniejsze od P. Następnie każdą z partycji sortujemy rekurencyjnie, po czym części łączymy ze sobą, uzyskując w ten sposób posortowaną tablicę.

### Zależność między ilością danych wejściowych a efektywnością algorytmu



### Obserwacje i wnioski

Istnieje bardzo duża rozbieżność między rodzajem danych a czasem działania algorytmu, co oznacza że czas działania algorytmu zależy od danych wejściowych. Najszybciej algorytm zadziałał dla danych posortowanych rosnąco i stałych.

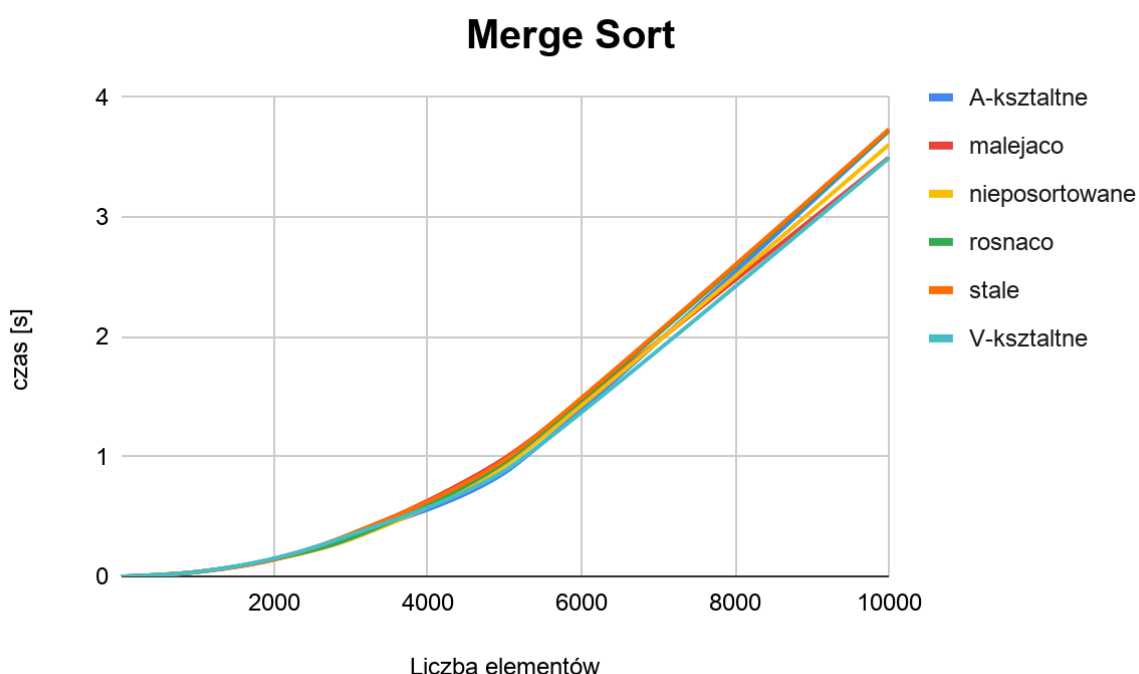
## Merge Sort

In situ – w miejscu	NIE
Stabilny	TAK
Usytuowanie wśród podobnych algorytmów	Stabilny, szybki Zużywa dużo pamięci

### Rozwiązanie

Dzielimy tablicę danych na dwie części, na każdej z nich wykonujemy Merge Sort (rekurencja), dopóki nie zostanie jeden element.

### Zależność między ilością danych wejściowych a efektywnością algorytmu



### Obserwacje i wnioski

Merge Sort posortował dane w podobnym czasie niezależnie od danych wejściowych, jednak można zauważyć różnicę w czasie między wyżej wymienionymi algorytmami naiwnymi a Merge Sortem.

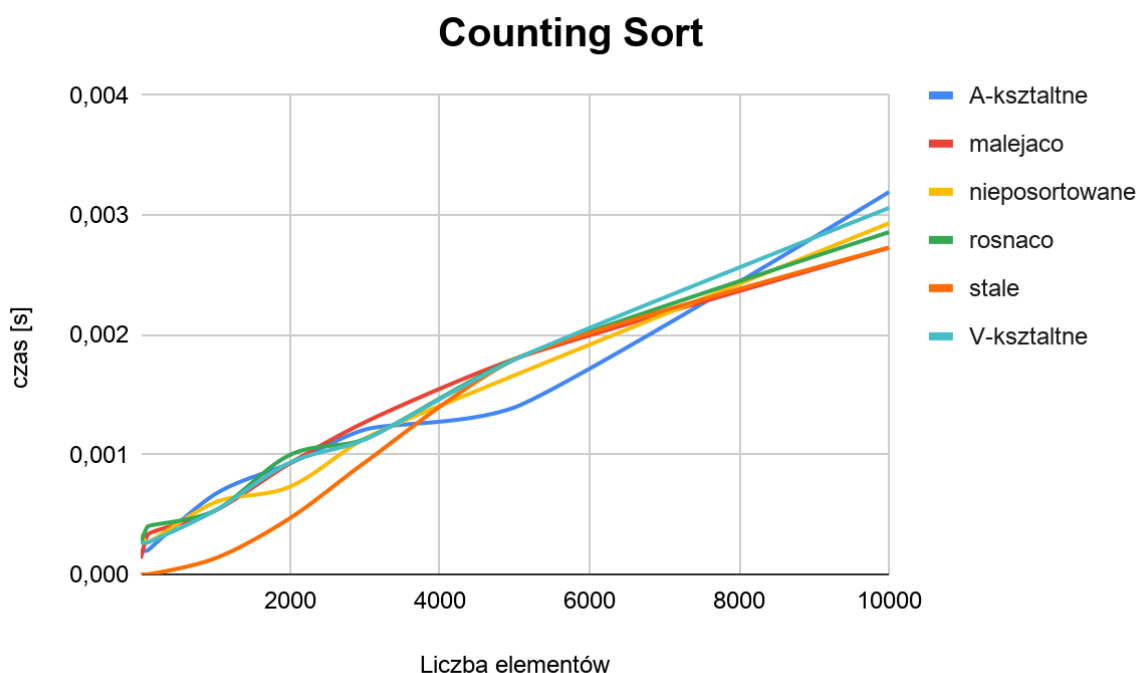
## Counting Sort

In situ – w miejscu	NIE
Stabilny	TAK
Usytuowanie wśród podobnych algorytmów	Stabilny, łatwy w implementacji Wartości danych liczbowych to liczby naturalne, zużywa dużo pamięci

### Rozwiązanie

Tworzymy nową tablicę, zliczamy ile jest elementów danej wartości i wpisujemy wynik do odpowiedniego indeksu w nowej tablicy (do indeksu, który jest równy wartości danego elementu). Wynik wypisujemy na podstawie nowej tablicy.

### Zależność między ilością danych wejściowych a efektywnością algorytmu



### Obserwacje i wnioski

Algorytm wyraźnie podniósł poprzeczkę jeśli chodzi o szybkość sortowania danych, posortował je w najkrótszym czasie. Ten rodzaj sortowania ma jednak pewne wymagania, które na dłuższą metę wykluczają go jako dobry algorytm, mianowicie nie działa przy elementach o wartości ujemnej.



## Wnioski i zestawienie algorytmów

Algorytmy sortowania przeważnie działają w podobnym czasie niezależnie od danych wejściowych i ciężko wybrać unikalny algorytm, który przeważa wszystkimi swoimi atutami. Najszybszym i najbardziej efektywnym algorytmem okazał się Quick Sort i Counting Sort (na wykresie poniżej zostało pokazane jak bardzo te algorytmy przeważają nad pozostałymi algorytmami), a najprostszym do zaimplementowania, choć mało wydajnym - Bubble Sort.

Na koniec przedstawione zostaje zestawienie wszystkich omawianych do tej pory sortowań, aby porównać ich efektywność.

### Zestawienie sortowań

