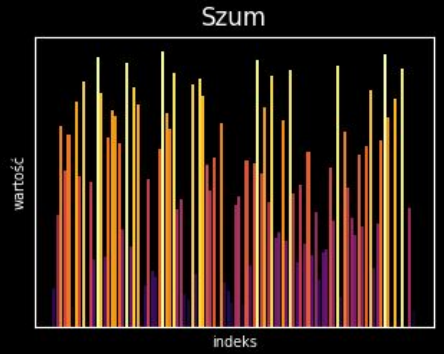
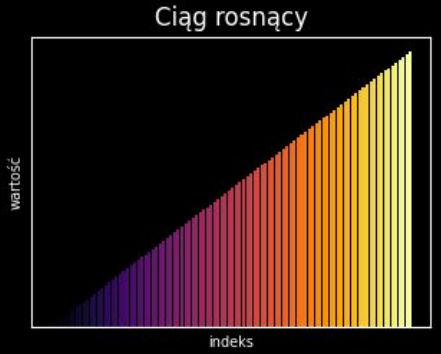
# Sprawozdanie

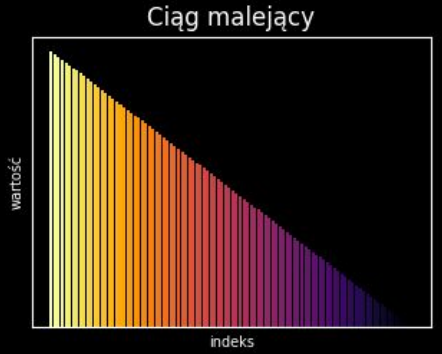
## Algorytmy sortujące

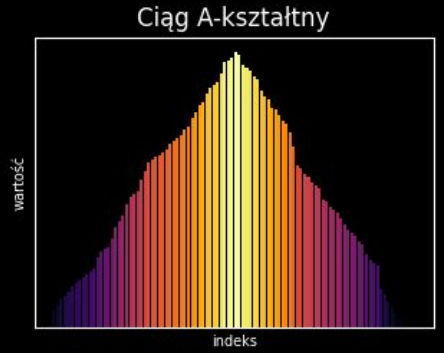
Jan Cieśla 155945

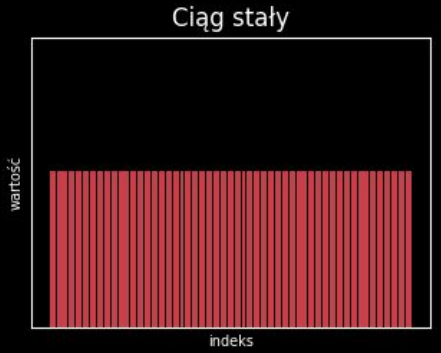
## Dane wejściowe



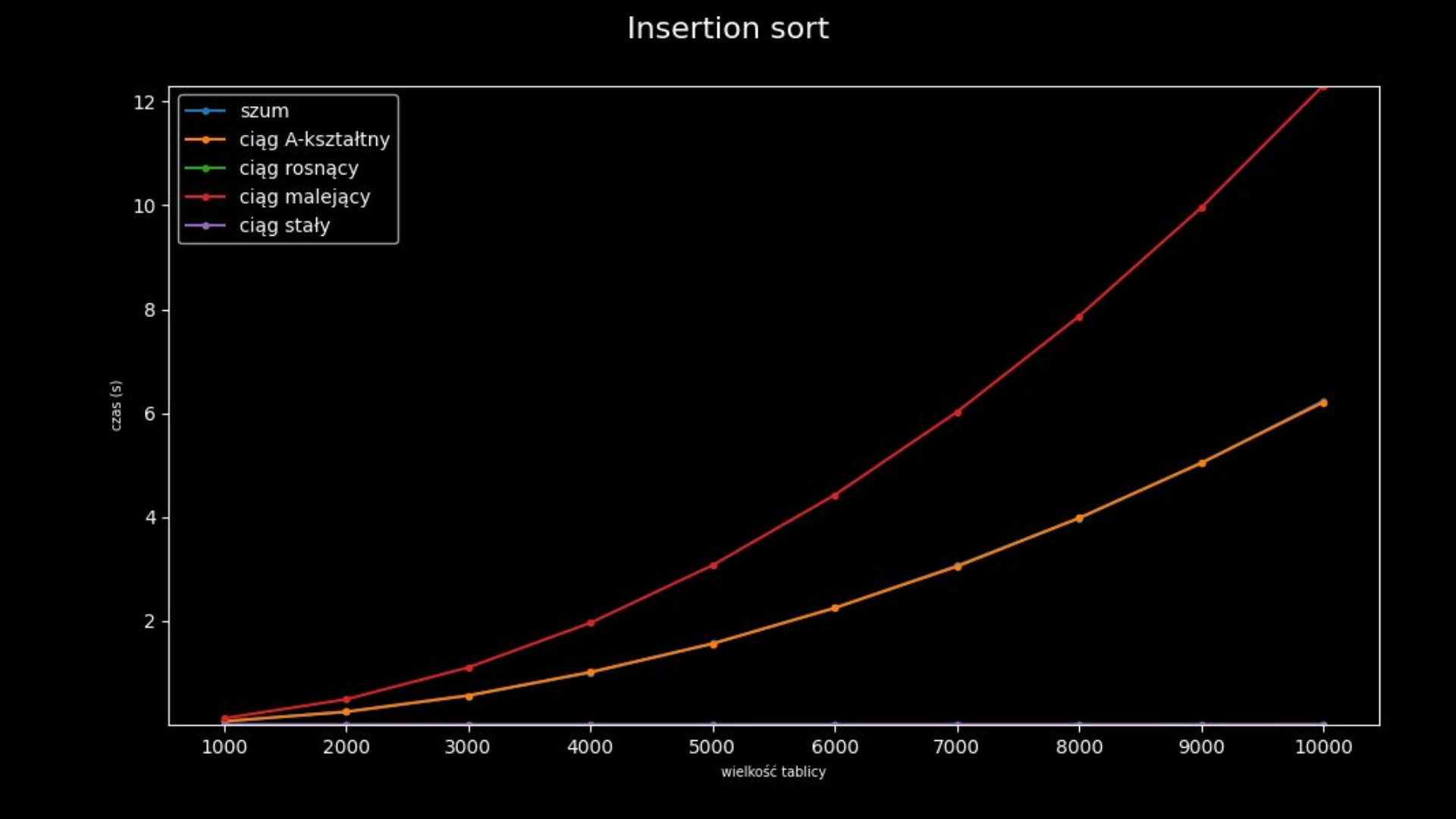



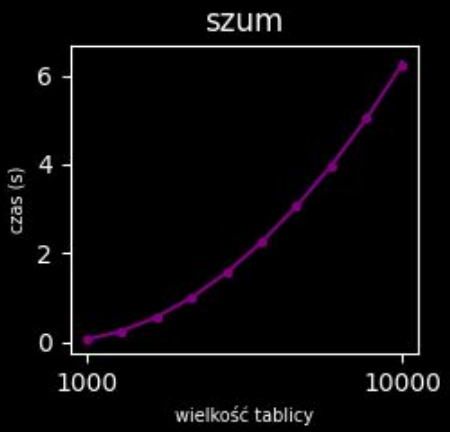



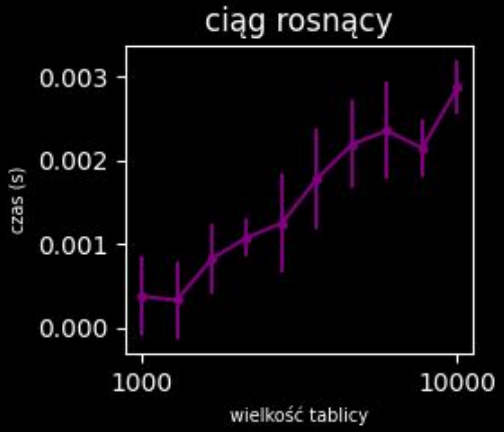

Dane wejściowe generowane są jako przekształcenie n-elementowego ciągu arytmetycznego o wyrazie minimalnym i maksymalnym odpowiednio -1000 i 1000.

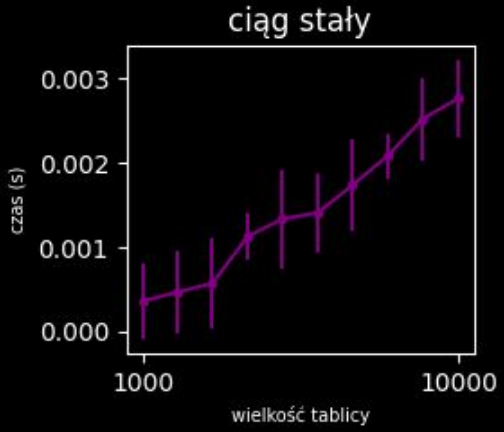
Do porównania efektywności algorytmów wybraliśmy n’y zaczynające się od 1000 zwiększające się co 1000 do 10000. Na wykresach prezentowane są uśrednione wyniki z 15 przebiegów. Zaznaczyliśmy także odchylenia standardowe. (Na oddzielnych wykresach dla czytelności)

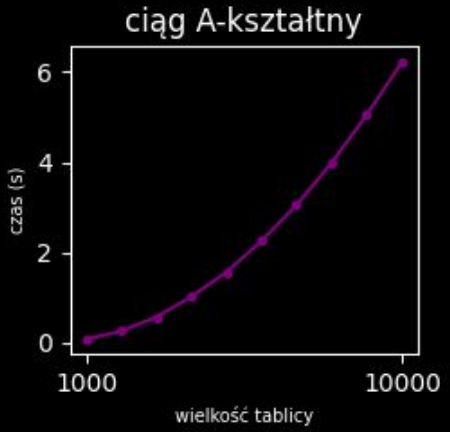

Insertion sort
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
czas (s)
wielkość tablicy
2
4
6
8
10
12
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

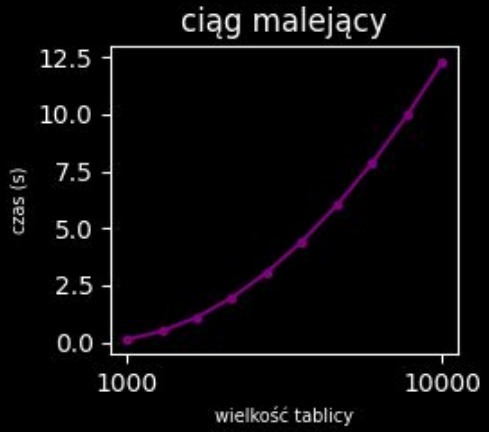
# Insertion sort zmienność wyników



# Złożoność obliczeniowa sortowania przez wstawianie

Przypadek optymistyczny

Czasowa: O(n)

Pamięciowa: O(1)

Przypadek średni

Czasowa: $O(n^2)$

Pamięciowa: O(1)

Przypadek pesymistyczny

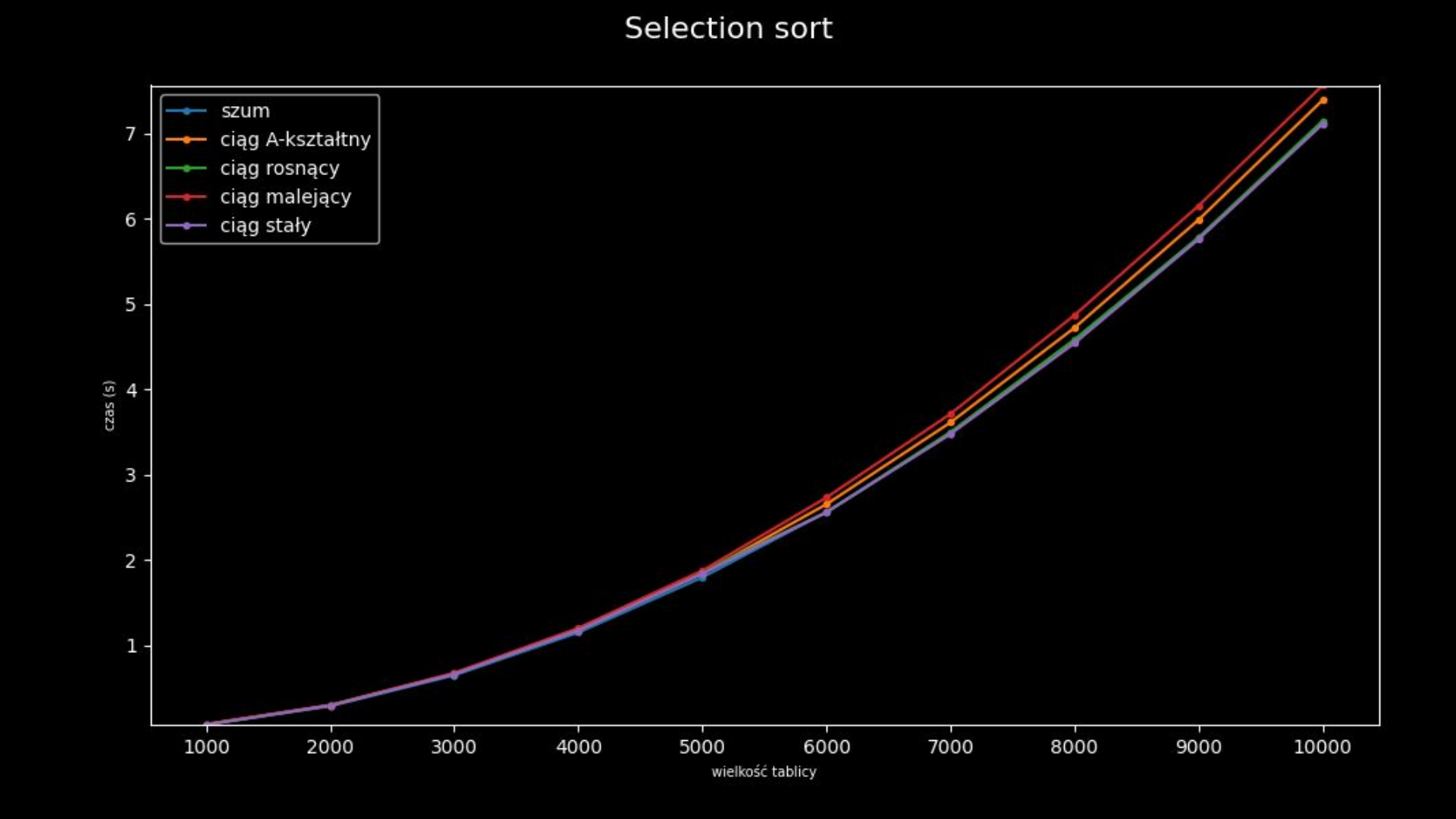
Czasowa: $O(n^2)$

Pamięciowa: O(1)

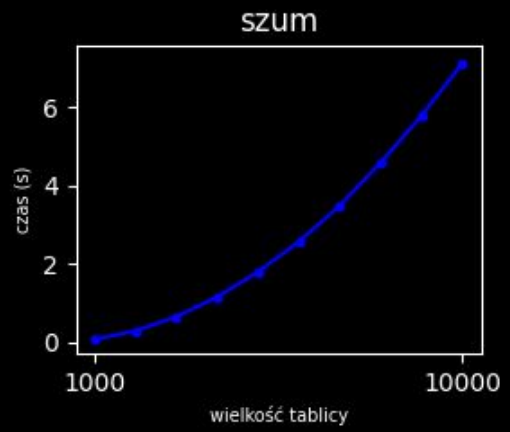
Algorytm ustawia kolejne elementy zbioru wejściowego na odpowiednie miejsce - odzwierciedla to ustawianie talii kart w odpowiedniej kolejności.

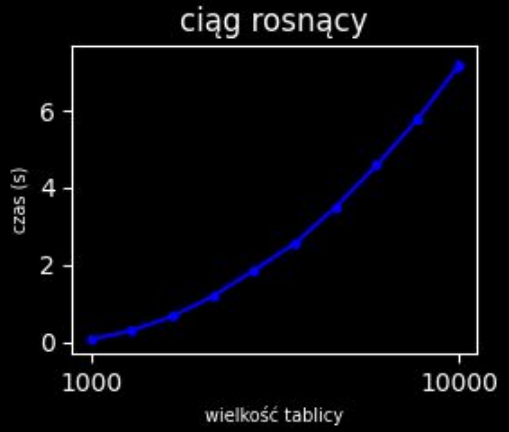
Algorytm jest stabilny oraz, co wynika z obserwacji, wydajniejszy dla zbiorów wstępnie posortowanych - wykona on wtedy najmniej porównań i zamian aby wstawić każdy element na odpowiednie miejsce.

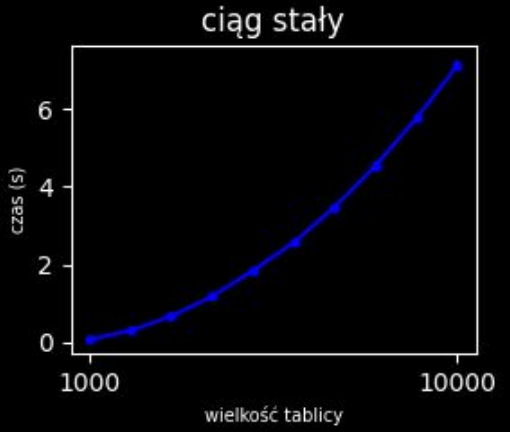
Na złożoność algorytmu składa się liczba porównań elementów oraz ich przestawienia w zbiorze. W przypadku optymistycznym czas wykonania algorytmu wyraża funkcja liniowa, więc złożoność redukuje się do O(n).

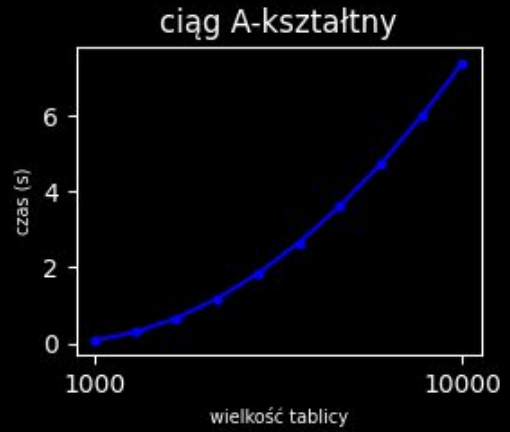

Selection sort
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
czas (s)
1
2
3
4
5
6
7
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
wielkość tablicy

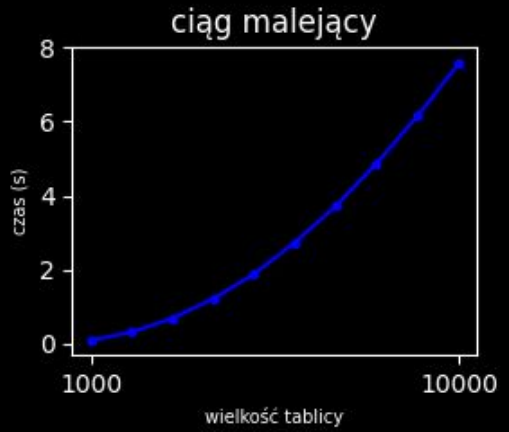
# Selection sort zmienność wyników



# Złożoność obliczeniowa sortowania przez wybór

Przypadek optymistyczny

Czasowa: $O(n^2)$

Pamięciowa: O(1)

Przypadek średni

Czasowa: $O(n^2)$

Pamięciowa: O(1)

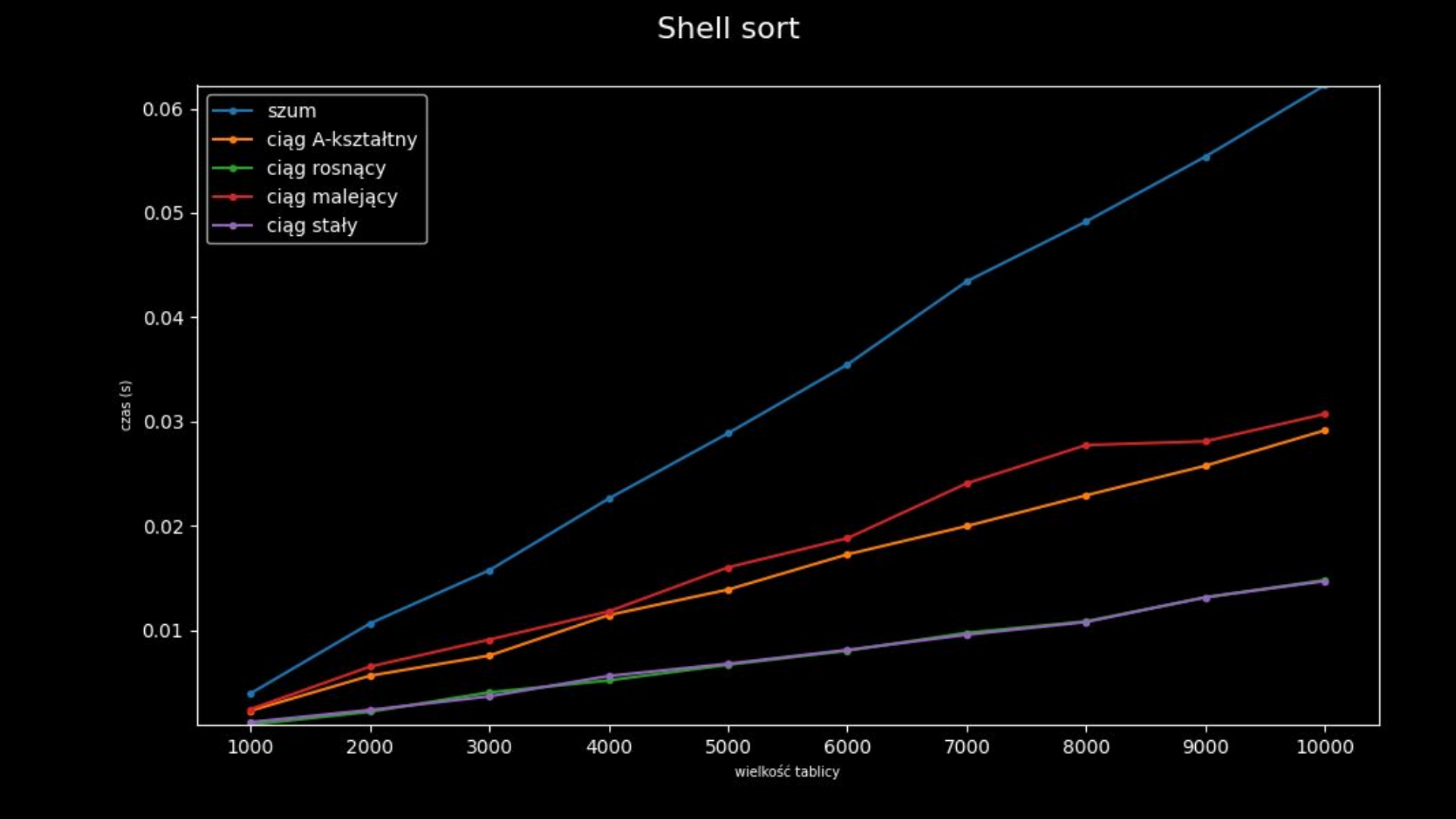
Przypadek pesymistyczny

Czasowa: $O(n^2)$

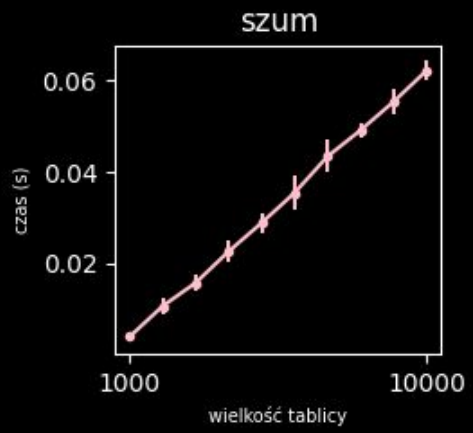
Pamięciowa: O(1)

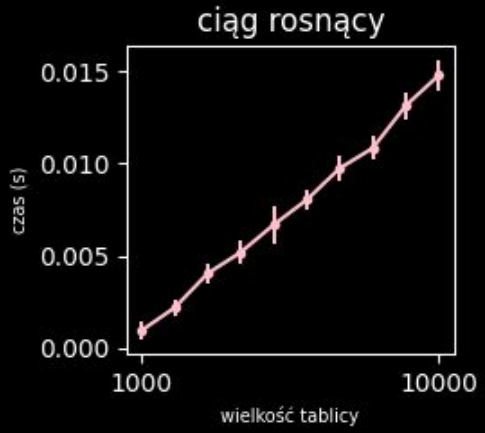
Algorytm dla każdego elementu wyszukuje minimalny z elementów o większych indeksach od aktualnego. Wykonuje on zatem (n-1) + (n-2) … + 1 porównań co skutkuje kwadratową złożonością czasową oraz brakiem potrzeby dodatkowej pamięci.

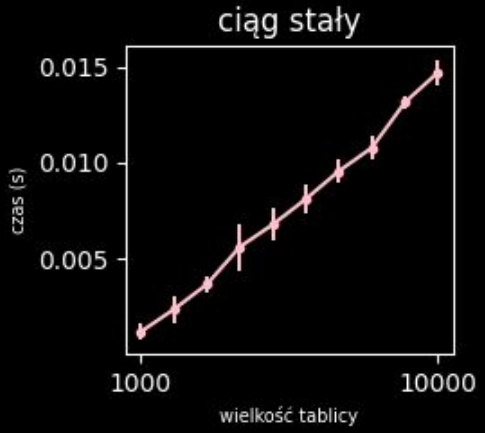
Sortowanie przez wybieranie działa “w miejscu” oraz ma identyczną wydajność w każdym przypadku, jest jednak NIESTABILNE odwróci ono kolejność identycznych elementów. Algorytm działa porównywalnie bez względu na dane wejściowe przez to, że zawsze wykona on tyle samo porównań (opisane w 1 akapicie) dla tablicy o ustalonej liczbie elementów.

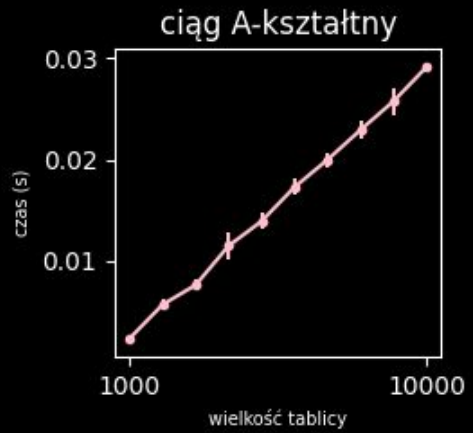

Shell sort
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
0.06
0.05
0.04
0.03
0.02
0.01
czas (s)
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
wielkość tablicy

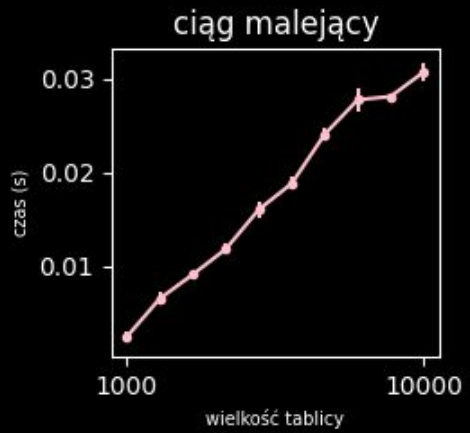
# Shell sort zmienność wyników



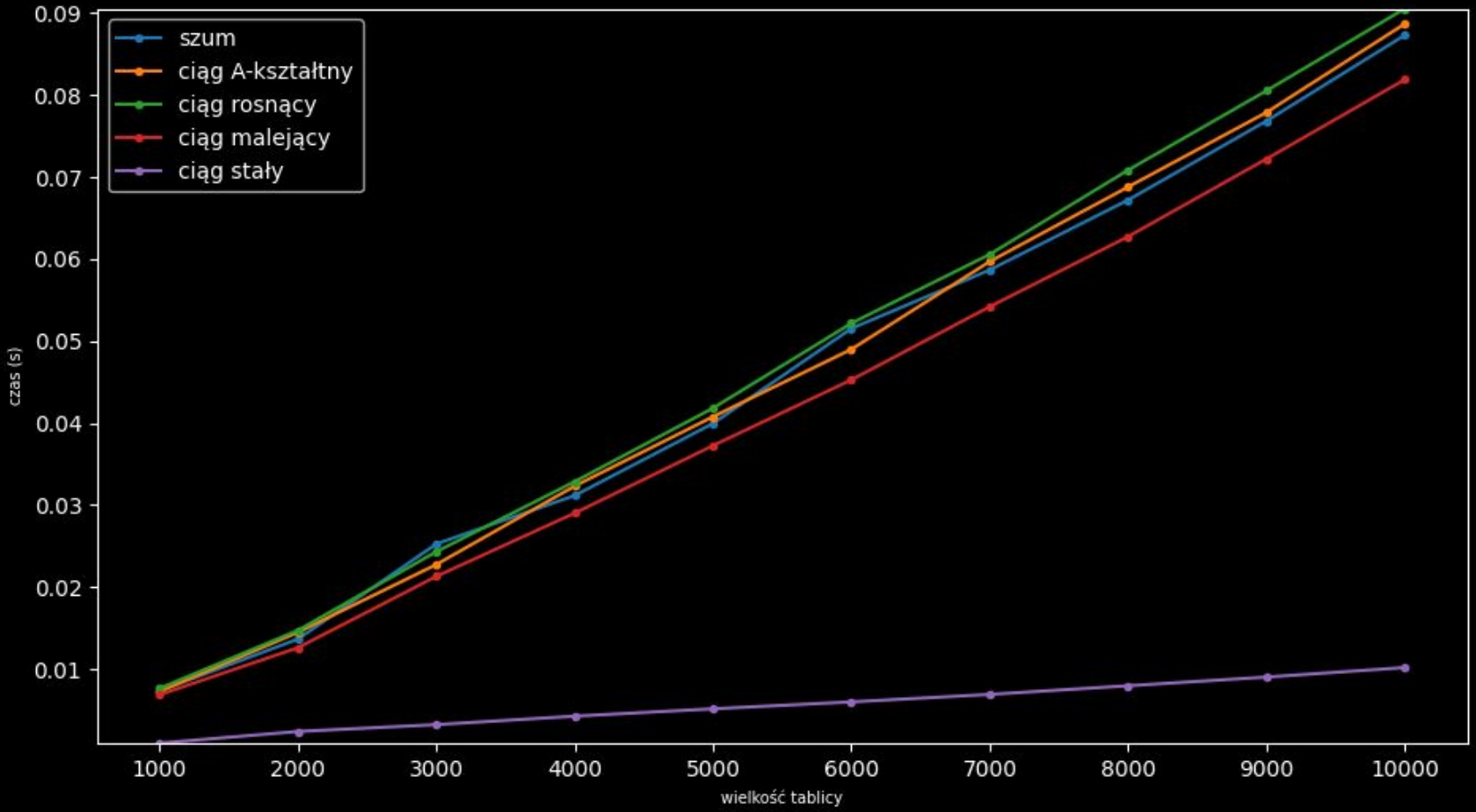
# Złożoność obliczeniowa sortowania Shella (przyrosty Sedgewicka)

| Przypadek optymistyczny | Przypadek średni | Przypadek pesymistyczny |
| --- | --- | --- |
| Czasowa: $O(n \cdot \log(n))$ | Czasowa: ? | Czasowa: $O(n^{4/3})$ \| $\Theta(n \cdot \log^2(n))$ |
| Pamięciowa: $O(1)$ | Pamięciowa: $O(1)$ | Pamięciowa: $O(1)$ |

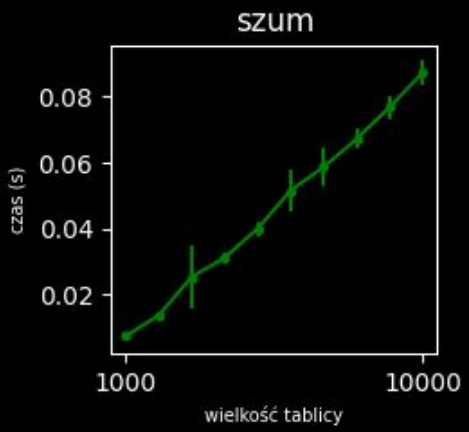
Jest to uogólnienie sortowania przez wstawianie, pozwalające na porównania i zamiany elementów położonych w zbiorze daleko od siebie. Algorytm stopniowo zmniejsza odstępy między sortowanymi elementami. Dzięki temu może je przenieść w docelowe miejsce szybciej niż *Insertion Sort*.

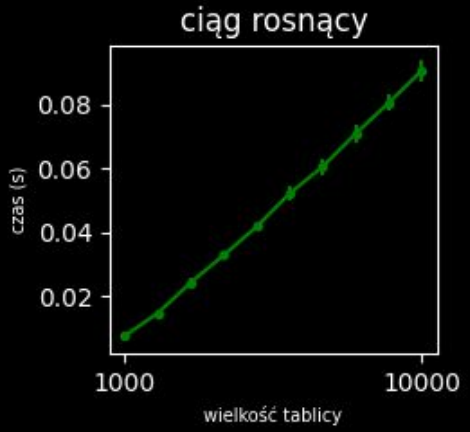
Wyznaczenie złożoności tego algorytmu zależy od użytych w nim przyrostów (odstępów) oraz od zastosowanego algorytmu bazowego (*Insertion / Bubble Sort*).

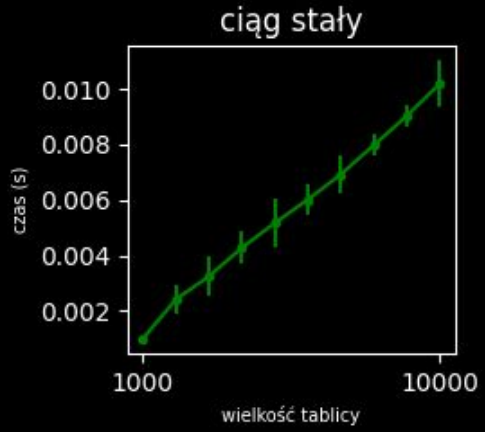
Możliwe jest uzyskanie przypadku pesymistycznego rzędu $\Theta(n \cdot \log^2(n))$, przyrosty są wtedy kolejnymi liczbami 3-gładkimi.

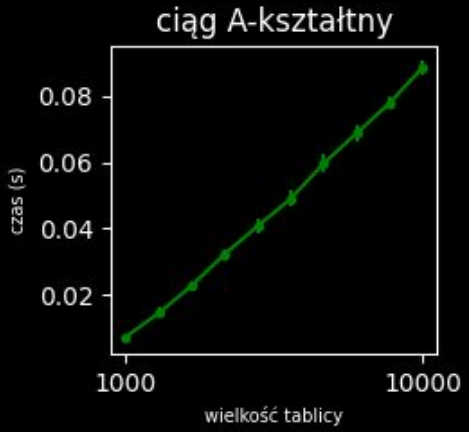

Heap sort
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
czas (s)
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
wielkość tablicy

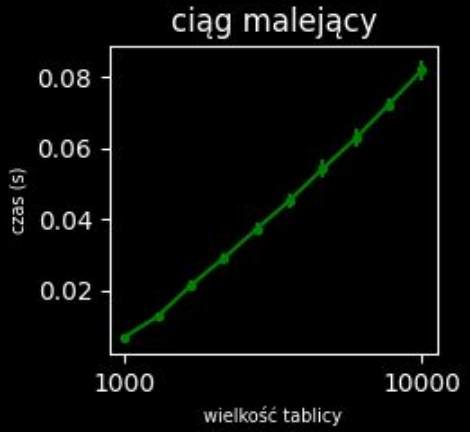
# Heap sort zmienność wyników



# Złożoność obliczeniowa sortowania przez kopcowanie

Przypadek optymistyczny

Czasowa: $O(n \cdot \log(n))$ | $O(n)$

Pamięciowa: $O(1)$

Przypadek średni

Czasowa: $O(n \cdot \log(n))$

Pamięciowa: $O(1)$

Przypadek pesymistyczny

Czasowa: $O(n \cdot \log(n))$

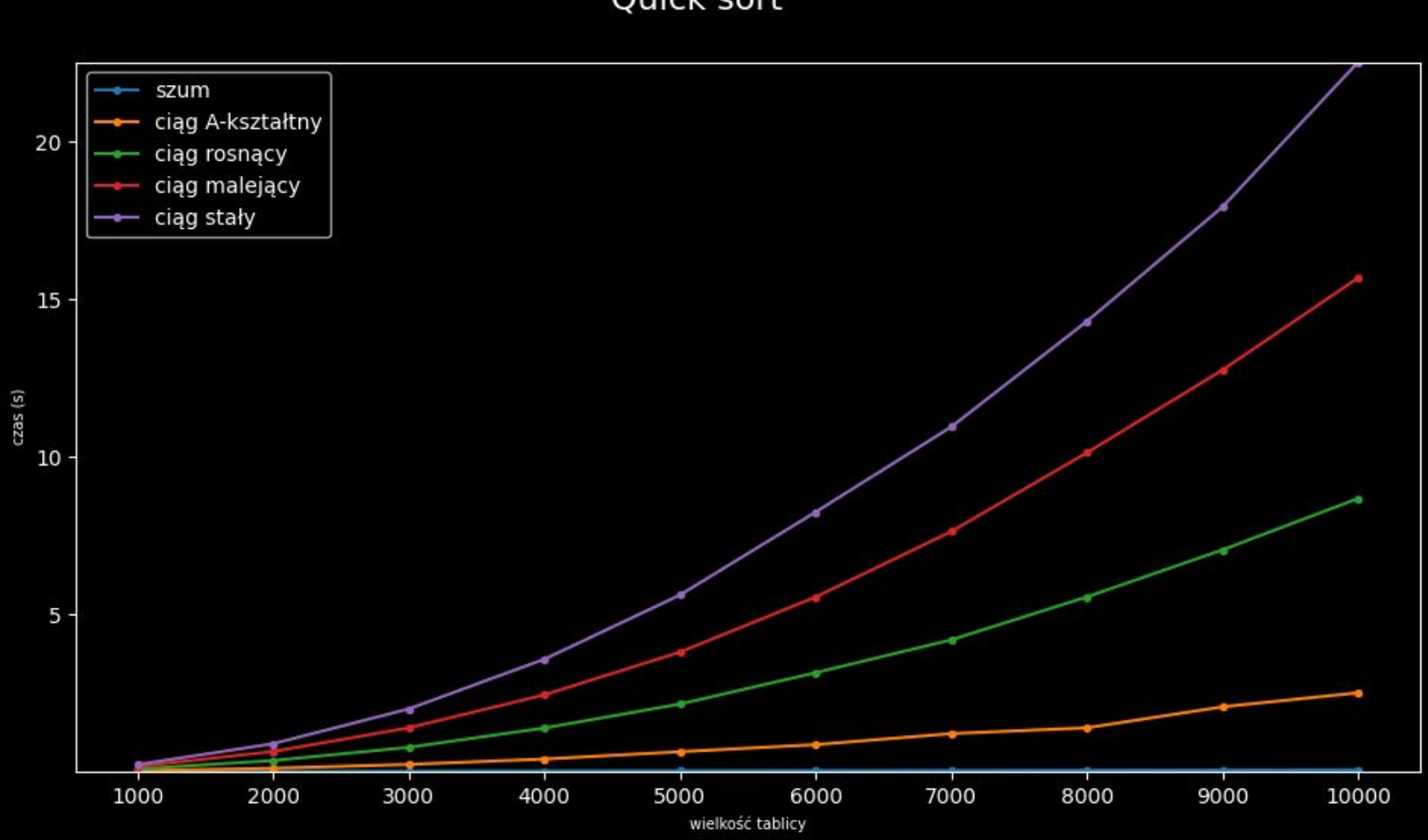
Pamięciowa: $O(1)$

Kopcowanie porównuje element z jego dziećmi w strukturze drzewa binarnego, procedura jest kontynuowana aż do osiągnięcia liści, zatem złożoność czasowa tej operacji to $O(\log(n))$. Nasza implementacja realizuje tą funkcje rekursywnie, możliwe jest jednak podejście iteracyjne, wtedy algorytm będzie działał ze złożonością pamięciową $O(1)$.

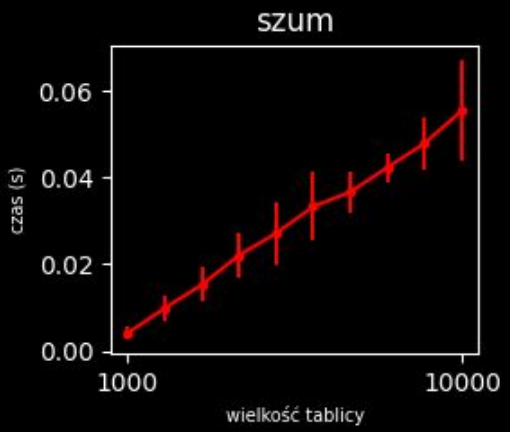
Tworzenie kopca wywołuje kopcowanie dla każdego elementu nie będącego liściem. Złożoność czasowa tego etapu to $O(n \cdot \log(n))$.

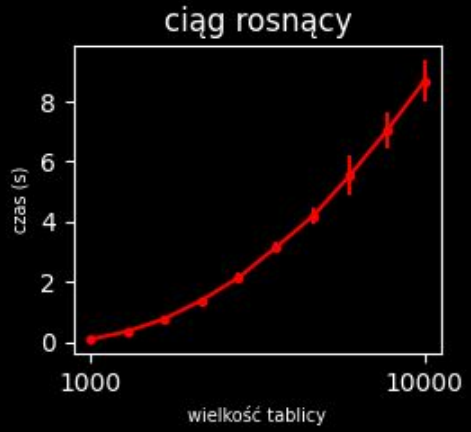
Sortowanie wywołuje kopcowanie dla każdego elementu, jednak ciągle zmniejsza się wielkość kopca a rośnie posortowana część tablicy. Sumarycznie złożoność czasowa to $O(n \cdot \log(n))$ a czasowa $O(1)$ (przy iteracyjnym kopcowaniu).

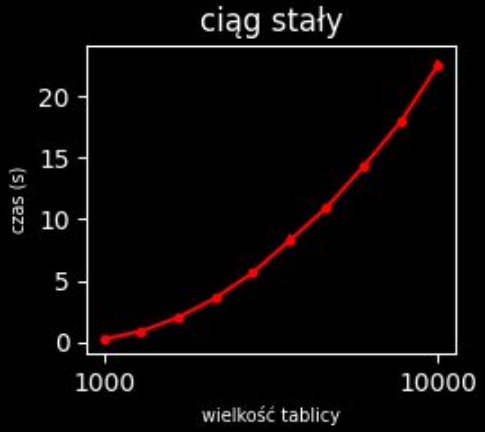
Algorytm jest NIESTABILNY, może być zaimplementowany tak aby sortował “w miejscu” i radzi sobie porównywalnie bez względu na typ danych, wyjątkiem są dane w których elementy są identyczne. Ogranicza to operację kopcowania, która w tym wypadku działa w stałym czasie, skutkując liniową złożonością algorytmu.

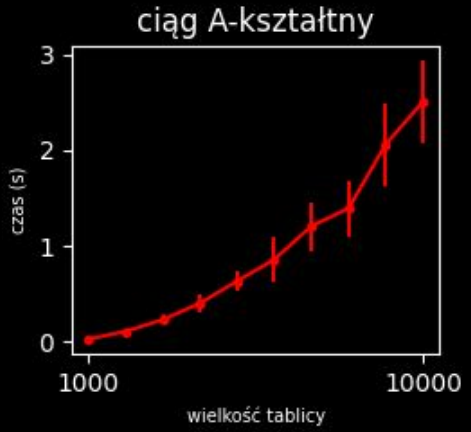

Quick sort
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
czas (s)
wielkość tablicy
5
10
15
20
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

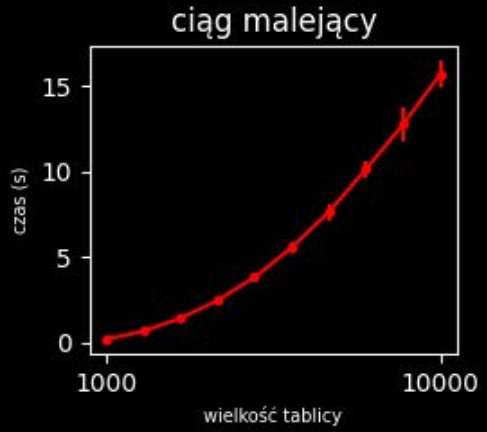
# Quick sort zmienność wyników



# Złożoność obliczeniowa sortowania szybkiego (pivot - pierwszy element)

| Przypadek optymistyczny | Przypadek średni | Przypadek pesymistyczny |
|---|---|---|
| Czasowa: $O(n \cdot \log(n))$ | Czasowa: $O(n \cdot \log(n))$ | Czasowa: $O(n^2)$ |
| Pamięciowa: $O(\log(n))$ | Pamięciowa: $O(\log(n))$ | Pamięciowa: $O(\log(n))$ |

Operacja podziału tablicy odbywa się w czasie liniowym - każdy element tablicy jest porównywany z wybranym pivotem.

Optymistycznie pivot zawsze będzie medianą zbioru, wtedy głębokość drzewa wywołań osiągnie najwyżej $\log_2 n$ stąd złożoność czasowa $O(n \cdot \log(n))$ i pamięciowa $O(\log(n))$.

W przypadku pesymistycznym pivot zawsze będzie najmniejszym/największym elementem w zbiorze. Algorytm wywoła się wtedy (n-1) razy stąd otrzymujemy złożoność czasową $O(n^2)$ i pamięciową $O(n)$.

Usprawniona wersja algorytmu po podziale tablicy najpierw obsługuje jej krótszą część, przez co ogranicza maksymalne zagłębienie rekurencji i zrównuje je z przypadkiem optymistycznym.

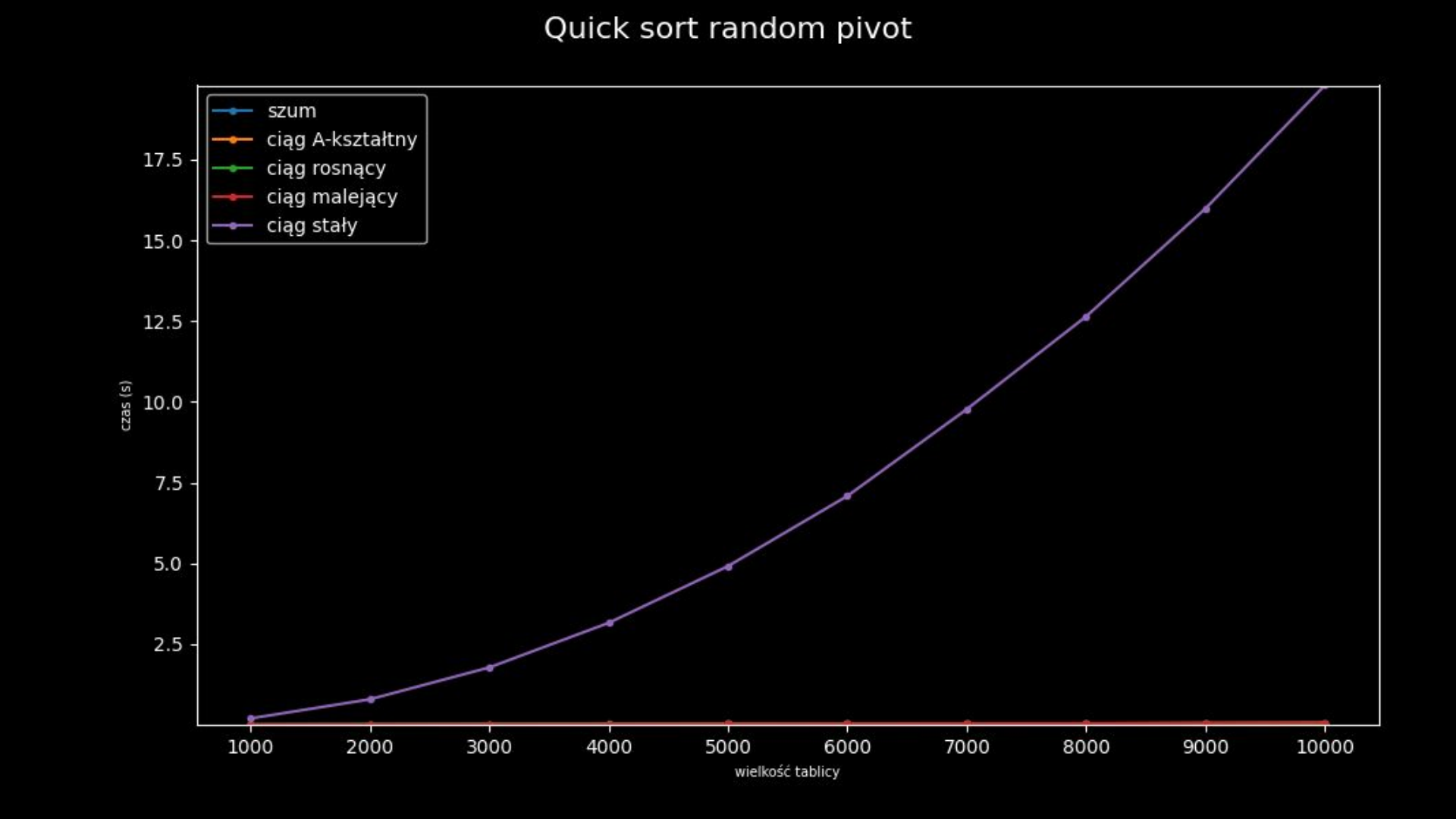
Algorytm świetnie radzi sobie z dużymi, nieposortowanymi danymi. Jakikolwiek porządek danych drastycznie zmniejsza jego wydajność. Dla danych posortowanych rosnąco, malejąco bądź stałych osiąga złożoność kwadratową. Sortowanie szybkie jest NIESTABILNE.

Różnica w czasie działania dla danych malejących, rosnących oraz stałych wynika z działania funkcji partition. W każdym z tych przypadków odbędzie się tyle samo porównań, jednak w przypadku danych rosnących wymagane jest (n-1)*2 zamian elementów w tablicy (zamiana elementu wybranego na pivot z ostatnim elementem tablicy i zamiana pivota z elementem wyznaczonym przez funkcję w tym przypadku zawsze indeks 0).

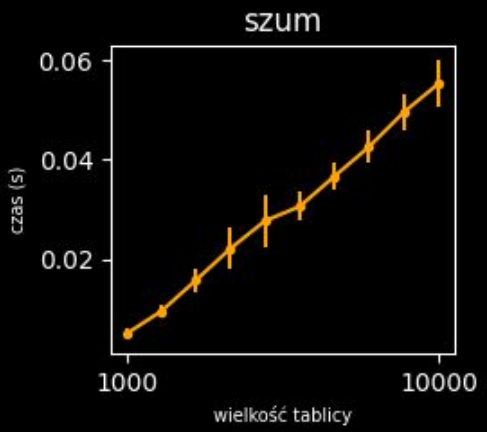
W przypadku danych malejących dla kolejnych rekurencyjnych wywołań sortowania szybkiego pivot będzie na zmianę największym i najmniejszym elementem dla najmniejszego sytuacja jest identyczna z ciągiem rosnącym, dla największego wymagane jest (n-1)+2 zamian. Dla całego sortowania wymagamy wtedy [(n+1) + ((n-2)+1)+ ((n-4)+1)...] + n//2 * 2 zamian.

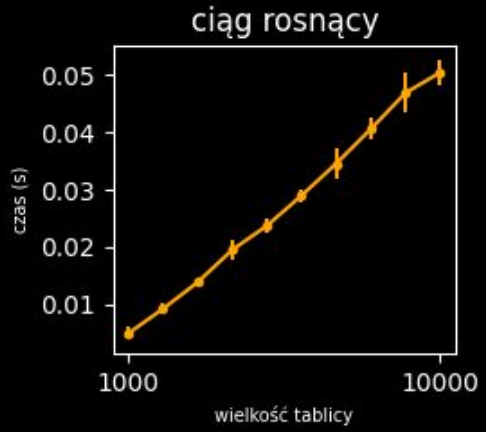
n parzyste: $\frac{n^2}{4} + 2n - 2$ n nieparzyste: $\frac{n+3}{2} \cdot \left\lceil \frac{n}{2} \right\rceil + 2 \cdot \left\lfloor \frac{n}{2} \right\rfloor - 2$

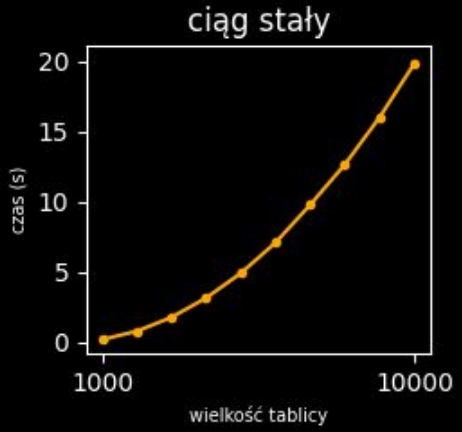
Podział stałych danych zawsze wymaga maksymalnej liczby zamian czyli (n+1)+n+n(-1)...+2 = $\frac{n^2 + 3n}{2} - 2$

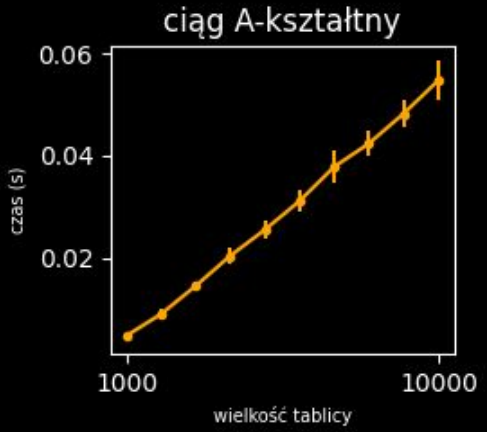

Quick sort random pivot
szum
ciąg A-kształtny
ciąg rosnący
ciąg malejący
ciąg stały
17.5
15.0
12.5
10.0
7.5
5.0
2.5
czas (s)
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
wielkość tablicy

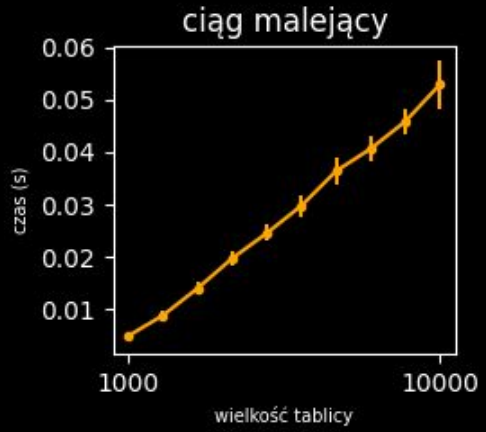
# Quick sort random pivot zmienność wyników



# Złożoność obliczeniowa sortowania szybkiego (pivot - losowy element)

Przypadek optymistyczny

Czasowa: $O(n \cdot \log(n))$

Pamięciowa: $O(\log(n))$

Przypadek średni

Czasowa: $O(n \cdot \log(n))$

Pamięciowa: $O(\log(n))$

Przypadek pesymistyczny

Czasowa: $O(n^2)$

Pamięciowa: $O(\log(n))$

Drobna zmiana w porównaniu do wersji algorytmu z pivotem jako pierwszy element tablicy zmniejsza prawdopodobieństwo wystąpienia przypadku pesymistycznego - wstępne posortowanie danych nie wpływa już na wydajność porządkowania. Algorytm jednak w dalszym ciągu nie radzi sobie z danymi w których poszczególne elementy często się powtarzają, bądź wszystkie są identyczne. Przy identycznych danych podział tablicy prowadzi wtedy do powstania nie posortowanej tablicy o długości n-1 i pivot'a ustawionego na ostatniej  pozycji.

W przypadku danych typu szum wykazuje się nieznacznie gorszą wydajnością niż bazowa wersja. (Przy takich danych pierwszy element tablicy ma identyczne prawdopodobieństwo jak każdy inny aby optymalnie podzielić zbiór, ale jego wybór nie zajmuje czasu jak wylosowanie pivot'a. Jest to delikatna różnica.)

# Porównanie wydajności