Politechnika
Wrocławska

Algorytmy i złożoność obliczeniowa	
Temat: Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową	
Osoba wykonująca projekt: Mateusz Bukowski	
Termin zajęć: Czwartek (TP) 13:15-15:00	**Numer grupy:** 2

Spis treści

Wprowadzenie

Podczas przeprowadzania eksperymentu badane są **cztery algorytmy sortowania**:

- sortowania przez wstawianie
- sortowania przez kopcowanie
- sortowania Shella
- sortowania quicksort

Opis badanych algorytmów wraz z ich złożonościami obliczeniowymi:

1. Sortowanie przez wstawianie

Sortowanie przez wstawianie jest to algorytm sortowania, który polega na wstawianiu kolejnych elementów tablicy w odpowiednie miejsce posortowanej już części tablicy. Iterując po sortowanej tablicy, patrzy się czy dany element jest mniejszy czy większy od wcześniejszych, posortowanych elementów tablicy i w zależności od potrzeb użytkownika (czy ma być to tablica posortowana rosnąco czy malejąco) wstawia się dany element w odpowiednie miejsce (czyli od miejsca, gdzie ma się wstawić dany element, przesuwa się posortowane elementy o jeden indeks w prawo, a w to miejsce wstawia się dany element) i przeskakuje się na kolejny element tablicy.
Złożoność obliczeniowa dla najgorszego i średniego przypadku jest identyczna i wynosi **$O(n^2)$**, natomiast dla optymistycznego przypadku wynosi **$O(n)$.**

2. Sortowanie przez kopcowanie

Sortowanie przez kopcowanie jest to algorytm sortowania, który opiera się na tworzeniu specjalnych drzew binarnych zwanymi kopcami. Na samym początku z nieuporządkowanych elementów tablicy tworzy się kopiec, a następnie zamienia się ostatni liść z korzeniem. W dalszej części algorytmu odcina się ostatni liść i na nowo przywraca się własności kopca, odpowiednio układając tablicę. Proces ten powtarza się aż dojdzie się do ostatniego pozostałego elementu tablicy.
Złożoność obliczeniowa dla najgorszego, średniego i najlepszego przypadku jest identyczna i wynosi **$O(n \log n)$**.

3. Sortowanie Shella

Sortowanie Shella jest to algorytm sortowania, który polega na początkowym stworzeniu ciągu elementów według konkretniej funkcji dla danej złożoności obliczeniowej. Elementy te są długościami odstępów między elementami sortowanej tablicy. Następnie sortowana tablica dzielona jest na podzbiór składający się z elementów tablicy odległych o największy odstęp (czyli największy element stworzonego początkowo ciągu). Stworzony podzbiór sortowany jest algorytmem przez wstawianie i dalej tworzone oraz sortowane są po kolei podzbiory, dla coraz to mniejszych odstępów (coraz mniejszych elementów stworzonego ciągu).
Dla funkcji na wyraz ogólny ciągu odstępów, którą stworzył Shell najgorsza złożoność obliczeniowa wynosi **O(n²)**, natomiast dla funkcji na wyraz ogólny ciągu odstępów, którą stworzył Sedgewick złożoność obliczeniowa wynosi **O($n^{\frac{4}{3}}$)**.

4. Sortowanie quicksort

Sortowanie quicksort jest to algorytm sortowania, który polega na wykorzystaniu techniki dziel i zwyciężaj. Z sortowanej tablicy na początku wybierany jest pivot, który jest ustalonym elementem tablicy (prawym, lewym, środkowym lub losowym). W dalszej kolejności tablica ustawiana jest w taki sposób, że po lewej stronie pivota znajdują się elementy nie większe od niego, a po jego prawej stronie znajdują się elementy nie mniejsze od niego. Następnie w powstałych dwóch podtablicach wybiera się nowe pivoty i dzieli się je na kolejne podtablice według tego samego schematu. Podziały podtablic następują do momentu, aż każda podtablica będzie miała dokładnie jeden element.
Złożoność obliczeniowa dla najgorszego przypadku wynosi **O(n²)**, natomiast dla średniego i najlepszego przypadku wynosi ona **O($n \log n$)**.

Plan eksperymentu

Algorytmy sortowania

Podczas przeprowadzania eksperymentu badane są cztery algorytmy sortowania:

- sortowanie przez wstawianie
- sortowanie przez kopcowanie
- sortowanie Shella
- sortowanie quicksort

Dla algorytmu Shella badane są dwa przypadki dla dwóch różnych złożoności obliczeniowych **(O(n^2), O($n^{4/3}$))**:

1. Ciąg odstępów Shella – złożoność obliczeniowa O(n^2)
    - Wzór na wyraz ogólny ciągu: $\left\lfloor \frac{N}{2^k} \right\rfloor$, dla k≥1, gdzie N to ilość elementów sortowanej tablicy.
2. Ciąg odstępów Sedgewicka – złożoność obliczeniowa O($n^{\frac{4}{3}}$)
    - Wzór na wyraz ogólny ciągu: $\mathbf{4^k + 3 * 2^{k-1} + 1}$, dla k≥1, gdzie na początku ciągu występuje wartość 1.

Dla algorytmu quicksort badane są cztery przypadki dla czterech różnych sposobów wyboru pivota (**skrajnie lewy, skrajnie prawy, środkowy, losowy**).

Rodzaje sortowanych tablic

Eksperyment badający algorytmy sortowania jest przeprowadzany dla siedmiu różnych wielkości tablic:

- tablic zawierających **10 tysięcy** elementów
- tablic zawierających **20 tysięcy** elementów
- tablic zawierających **40 tysięcy** elementów
- tablic zawierających **50 tysięcy** elementów
- tablic zawierających **80 tysięcy** elementów
- tablic zawierających **100 tysięcy** elementów
- tablic zawierających **200 tysięcy** elementów

Dodatkowo sprawdzany jest także wpływ kolejności wygenerowanych elementów dla każdej wielkości tablicy. Algorytmy są badane dla tablic, w których elementy:

- wygenerowane są **losowo**
- posortowane są **rosnąco**
- posortowane są **malejąco**
- posortowane są w **33%**
- posortowane są w **66%**

Elementy dla tablic losowych, rosnących i malejących są generowane losowo **z przedziału od 0 do 10000** za pomocą użycia funkcji **rand()**, a następnie w zależności od żądanej kolejności elementów w tablicy, elementy są sortowane za pomocą algorytmu Shella (albo rosnąco, albo malejąco). Dla tablic posortowanych w 33% i w 66% na początku do pierwszych 33% lub odpowiednio 66% elementów przypisywane są znane wartości – od 0, inkrementując dla kolejnych elementów, aż do zapełnienia 33% lub 66% części tablicy. Dla dalszych elementów liczby są generowane losowo od liczby większej niż największa liczba z 33% lub odpowiednio 66% części tablicy.

Wszystkie algorytmy są sprawdzane dla elementów będących liczbami całkowitymi ze znakiem (**typ danych int**), natomiast dla algorytmu sortowania przez kopcowanie mierzone są dodatkowo średnie czasy dla tablic zbudowanych z elementów będących liczbami rzeczywistymi (**typ danych float**).

Każdy algorytm sortowania jest badany dla każdej wielkości tablicy oraz każdego sposobu jej generowania, poprzez **50-krotne** użycie sprawdzanego algorytmu dla każdej z tablic.

Badanie algorytmów polega na mierzeniu czasu sortowania danej tablicy, powtórzenia tego procesu 50-krotnie, a następnie wyliczeniu średniej zmierzonych czasów.
Czasy mierzone są za pomocą funkcji **chrono::high_resolution_clock** (od rozpoczęcia sortowania do zakończenia sortowania, nie licząc czasu generowania tablic).

Przebieg eksperymentu

Na początku odpowiednio generowana jest tablica o zadanym rozmiarze, w zależności od badanego wpływu sposobu generowania tablic (bez sortowania-losowa, rosnąco, malejąco, w 33%, w 66%). W dalszej kolejności tworzona jest kopia odpowiednio wygenerowanej tablicy, a później jest ona 50-krotnie sortowana każdym algorytmem po kolei (przez wstawianie, przez kopcowanie, shella dla dwóch złożoności, quicksort dla odpowiednich czterech pivotów). Dla każdego powtórzenia mierzony jest czas sortowania, następnie każdy z tych czasów zapisywany jest do pliku, a na koniec dla każdego algorytmu wyliczany jest średni czas sortowania dla danego rozmiaru tablicy i dla danego sposobu generowania tablicy. Proces ten jest powtarzany dla każdej wielkości tablicy i dla każdego typu wygenerowanej tablicy.

Dodatkowe informacje:

Kod źródłowy algorytmów sortowania wraz z menu został napisany obiektowo w języku C++. Aby zbadać czasy sortowań dla różnych typów danych zostały użyte szablony.

Dokładny opis badanych algorytmów na podstawie kodu źródłowego

Dla sortowania rosnącego:

1. **Sortowanie przez wstawianie**

   1.1. Wywoływana jest pętla for, która rozpoczyna się od drugiego elementu tablicy.

   1.2. Następnie do zmiennej pomocniczej zapisywana jest wartość aktualnego elementu.

   1.3. W dalszej części aktualny element tablicy porównywany jest z elementem poprzednim i jeśli jest on od niego mniejszy, aktualny element przyjmuje wartość poprzedniego elementu. Operacja ta powtarzana jest w stronę początku tablicy, aż poprzedni element będzie mniejszy bądź równy aktualnemu elementowi. Na samym końcu na miejsce obecnego elementu przypisujemy wartość ze zmiennej pomocniczej.

   1.4. Następnie przechodzi się do kolejnej iteracji pętli for i powtarza się krok 2 i 3. Powtarza się te czynności aż do ostatniego elementu tablicy.

2. Sortowanie przez kopcowanie

2.1. Wywoływana jest funkcja sortowanie_przez_kopcowanie, w której na samym początku wywoływana jest funkcja tworzenie_kopca. Funkcja ta, idąc od indeksu ostatniego rodzica aż do indeksu korzenia, wywołuje funkcję kopiec.

2.2. W funkcji kopiec porównywany jest rodzic wraz z jego potomkami. Z porównywanych wartości, największa z nich przypisywana jest do wartości rodzica poprzez zamianę z nim (o ile największej z nich nie ma rodzic).

2.3. Jeśli nastąpiła zamiana elementów, czyli jeśli któryś z potomków miał wartość większą od wartości rodzica, referencyjnie wywoływana jest funkcja kopiec dla indeksu elementu, który został zamieniony z rodzicem.

2.4. W dalszej części funkcji sortowanie_przez_kopcowanie wywoływana jest pętla for od ostatniego indeksu tablicy, w której na początku zamieniany jest korzeń z ostatnim liściem, a następnie wywoływana jest funkcja kopiec dla indeksu korzenia oraz dla ilości elementów sortowanej tablicy pomniejszonej z każdą iteracją o jeden. Czyli po każdej zamianie korzenia z ostatnim liściem, ostatni liść jest odcinany. Pętla for jest powtarzana aż dojdzie do zerowego indeksu sortowanej tablicy.

3. Sortowanie Shella

3.1. Na początku wyliczany jest według wzoru funkcji wyrazu ogólnego największy możliwy odstęp.

3.2. Następnie w pętli for rozpoczynającej się od wartości danego odstępu, do zmiennej pomocniczej zapisywana jest wartość aktualnego elementu i później porównywane są dwa elementy sortowanej tablicy. Element z indeksem równym odstępowi i elementem z indeksem mniejszym o dany odstęp. Jeśli ten drugi element jest większy, to do elementu pierwszego przypisywana jest wartość elementu drugiego. Sprawdzane jest także aż do końca czy nie znajduje się element tablic z indeksem mniejszym od indeksu drugiego elementu o odstęp. Jeśli istnieje to te kolejne dwa elementy są porównywane i w razie potrzeby zamieniane. Jeśli jednak nie istnieje to w miejsce elementu z najmniejszym istniejącym indeksem dla danego odstępu dla danej iteracji pętli for wpisywana jest wartość zmiennej pomocniczej.

3.3. W dalszej części przechodzi się do kolejnej iteracji pętli for i powtarza się czynności z kroku drugiego.

3.4. Jeśli pętla for dojdzie do wartości rozmiaru sortowanej tablicy, wyliczany jest kolejny (mniejszy) odstęp i powtarzane są kroki 2 i 3.

3.5. Ostatnią wartością odstępu, dla którego przeprowadzany jest proces sortowania jest wartość jeden.

4. Sortowanie quicksort

4.1. Wywoływana jest funkcja quicksort dla wszystkich elementów sortowanej tablicy.

4.2. W funkcji quicksort na początku wywoływana jest funkcja pivotIndex, w której tablica jest odpowiednio sortowana na elementy nie większe i nie mniejsze od pivota, a następnie zwracana jest wartość indeksu, na którym znajduje się pivot. Na samym początku funkcji pivotIndex wywoływana jest także funkcja pivot, która zwraca odpowiednie wartości (left, right, middle, random) w zależności od tego jaki rodzaj pivota został wybrany przez użytkownika na początku programu.

4.3. W dalszej części w funkcji quicksort, referencyjnie wywoływane są funkcje quicksort dla dwóch powstałych podtablic (podtablicy z elementami nie większymi od pivota i podtablicy z elementami nie mniejszymi od pivota).

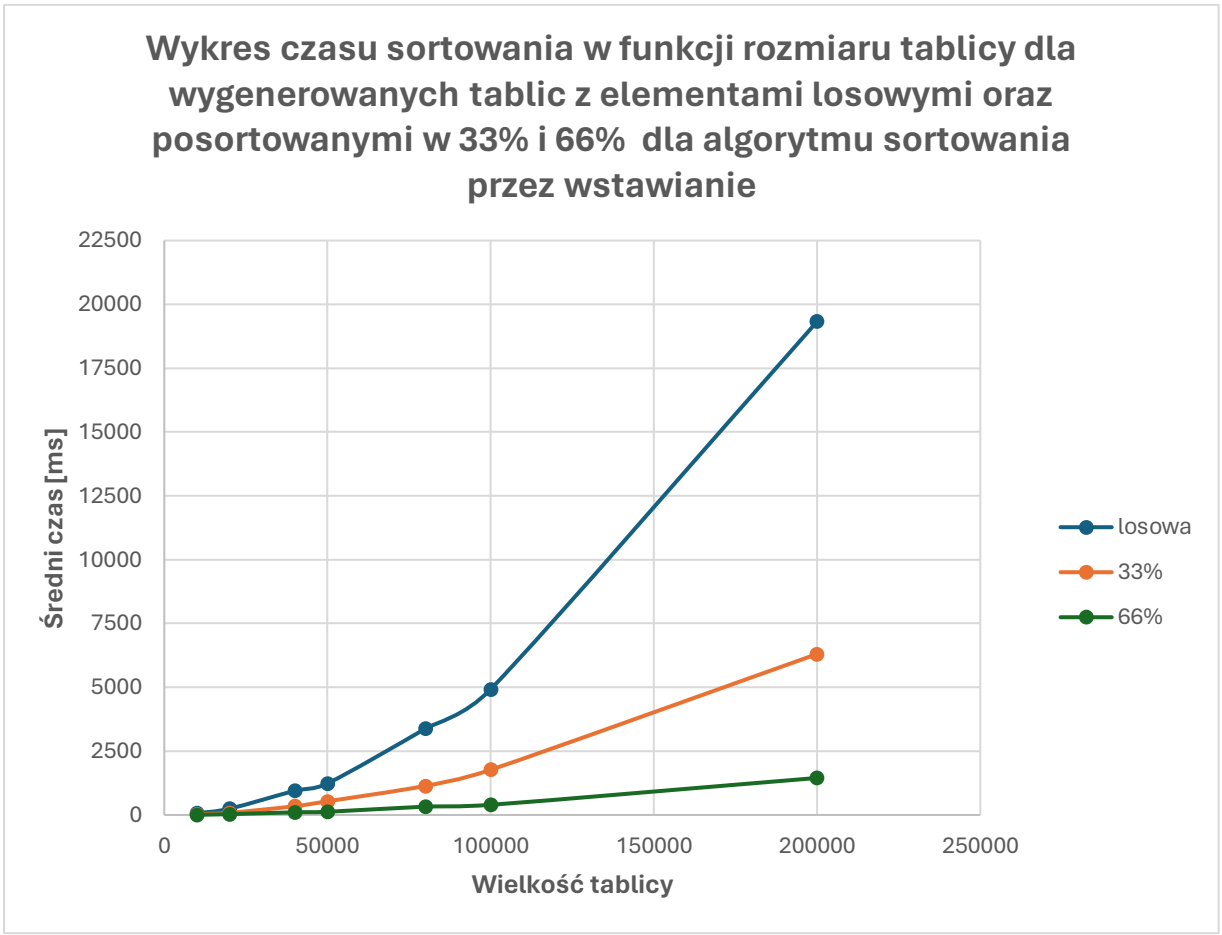
4.4. Dla kolejnych wywołań funkcji quicksort powtarzane są kroki 2 i 3, aż końcowe podtablice będą zbudowane z dokładnie jednego elementu.

Wyniki:

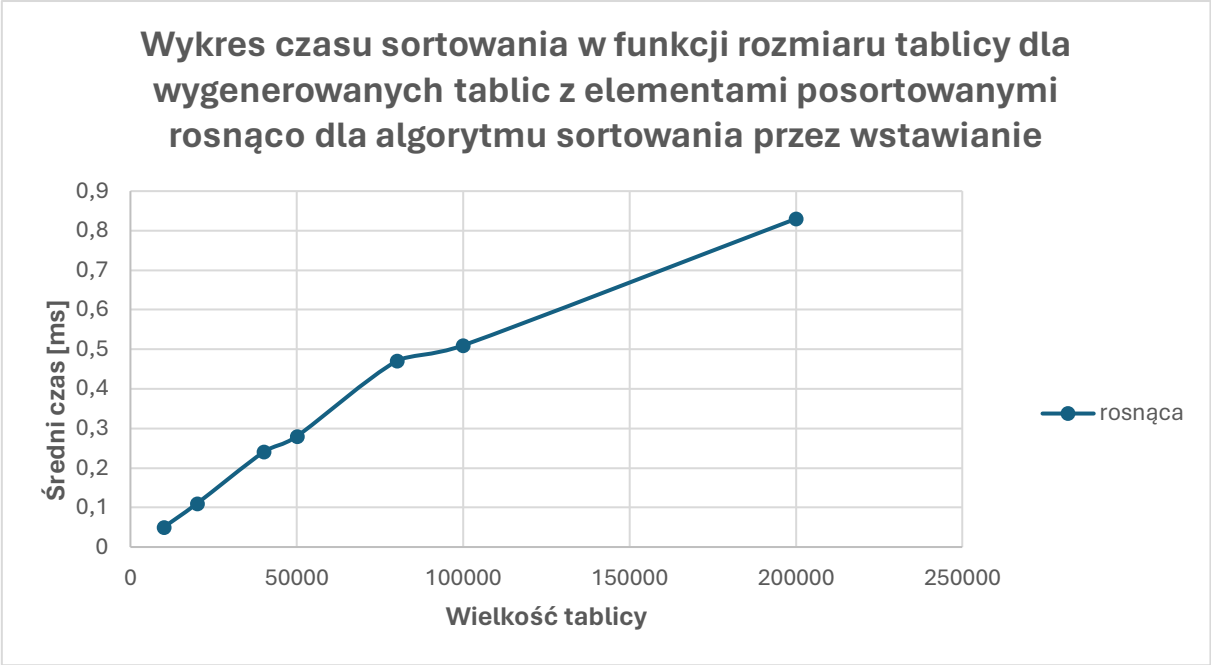
Wyniki dla algorytmu sortowania przez wstawianie:

Tabela 1: Tabela średnich czasów sortowania tablic algorytmem sortowania przez wstawianie

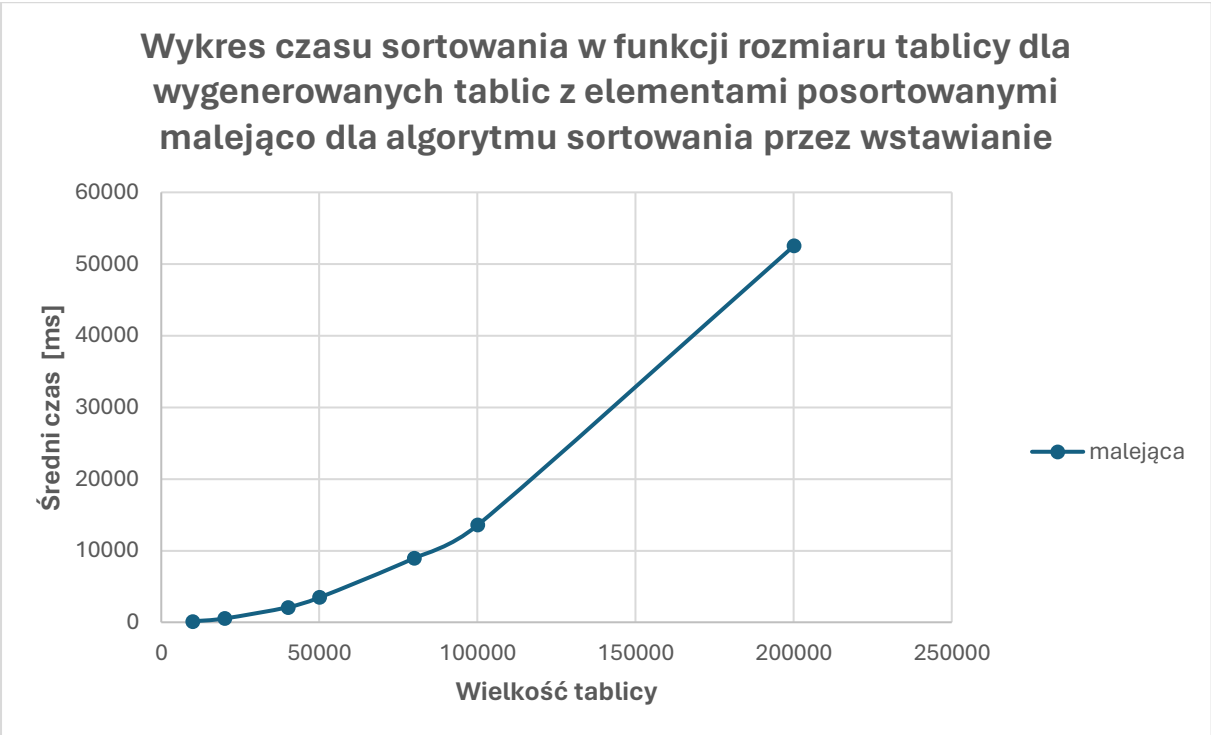
Średnie czasy [ms] sortowania tablic algorytmem sortowania przez wstawianie							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	67,20	245,74	948,74	1237,86	3381,20	4925,54	19326,82
posortowana rosnąco	0,05	0,11	0,24	0,28	0,47	0,51	0,83
posortowana malejąco	131,48	528,02	2094,34	3454,32	8952,26	13538,42	52496,67
posortowana w 33%	28,86	85,88	342,64	535,26	1140,70	1769,54	6302,94
posortowana w 66%	7,68	31,80	104,54	125,02	326,86	397,92	1450,50



Rysunek 1: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi w 33% i 66% dla algorytmu sortowania przez wstawianie



Rysunek 2: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami posortowanymi rosnąco dla algorytmu sortowania przez wstawianie



Rysunek 3: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami posortowanymi malejąco dla algorytmu sortowania przez wstawianie

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania przez wstawianie.

Dla tablic losowych zależność między średnim czasem sortowania a wielkością tablic jest kwadratowa. Jest to zgodne ze teoretyczną złożonością obliczeniową – $O(n^2)$.

Dla tablic częściowo posortowanych (w 33% i 66%), wyniki także zgadzają się z teoretyczną złożonością obliczeniową. Dodatkowo średnie czasy sortowania dla tych rodzajów tablic są już wielokrotnie niższe, od tych dla tablic losowych, co także jest prawidłowym i oczekiwanym zjawiskiem.

Dla tablic posortowanych malejąco zachodzi przypadek pesymistyczny, sortowanie zajmuje najwięcej czasu. Wyniki zgodne są z oczekiwaną złożonością - $O(n^2)$.

Dla tablic posortowanych rosnąco, zachodzi przypadek optymistyczny. Zależność między średnim czasem sortowania a wielkością tablic jest zbliżona do zależności liniowej. Jest to zgodne z oczekiwaną złożonością - $O(n)$.

Dla tablic (zwłaszcza tych z dużą ilością elementów), w których elementy są losowe, lub są posortowane malejąco algorytm sortowania przez wstawianie jest mało efektywny, jednak dla tablic z elementami posortowanymi rosnąco algorytm ten jest bardzo wydajny i skuteczny.

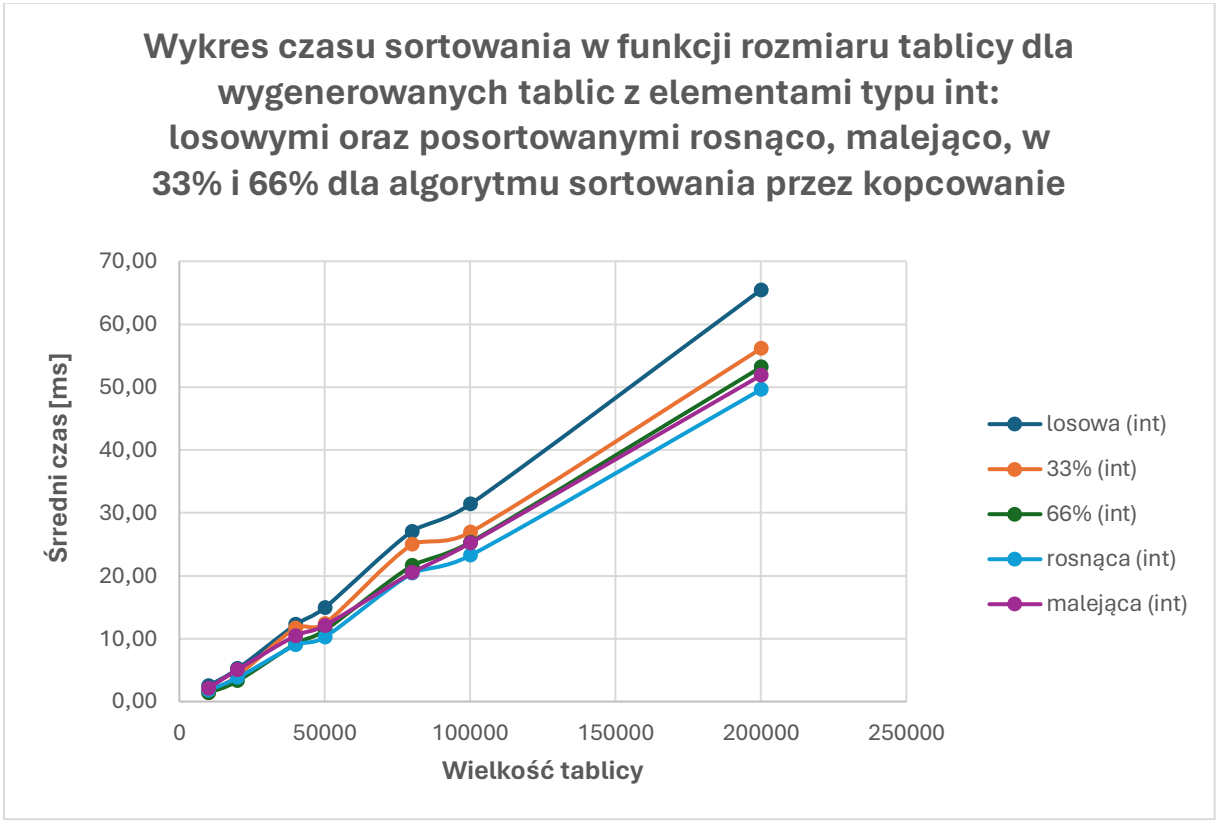
Wyniki dla algorytmu sortowania przez kopcowanie:

Tabela 2: Tabela średnich czasów sortowania tablic algorytmem sortowania przez kopcowanie dla liczb całkowitych

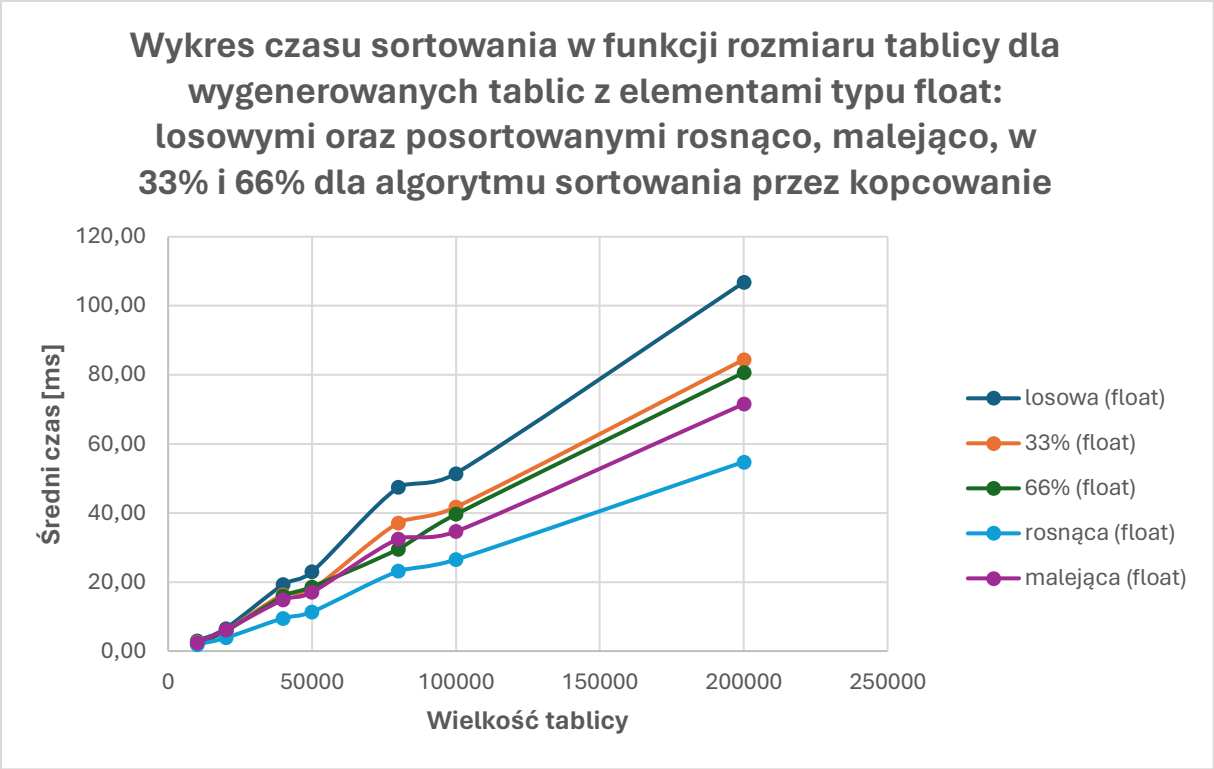
Średnie czasy [ms] sortowania tablic algorytmem sortowania przez kopcowanie							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa (int)	2,49	5,20	12,24	14,92	27,07	31,36	65,44
posortowana rosnąco (int)	1,78	3,78	8,98	10,24	20,37	23,21	49,60
posortowana malejąco (int)	2,08	5,04	10,48	12,08	20,48	25,18	51,86
posortowana w 33% (int)	1,42	4,10	11,70	12,38	24,96	26,90	56,12
posortowana w 66% (int)	1,36	3,26	9,20	11,32	21,58	25,32	53,14

Tabela 3: Tabela średnich czasów sortowania tablic algorytmem sortowania przez kopcowanie dla liczb rzeczywistych

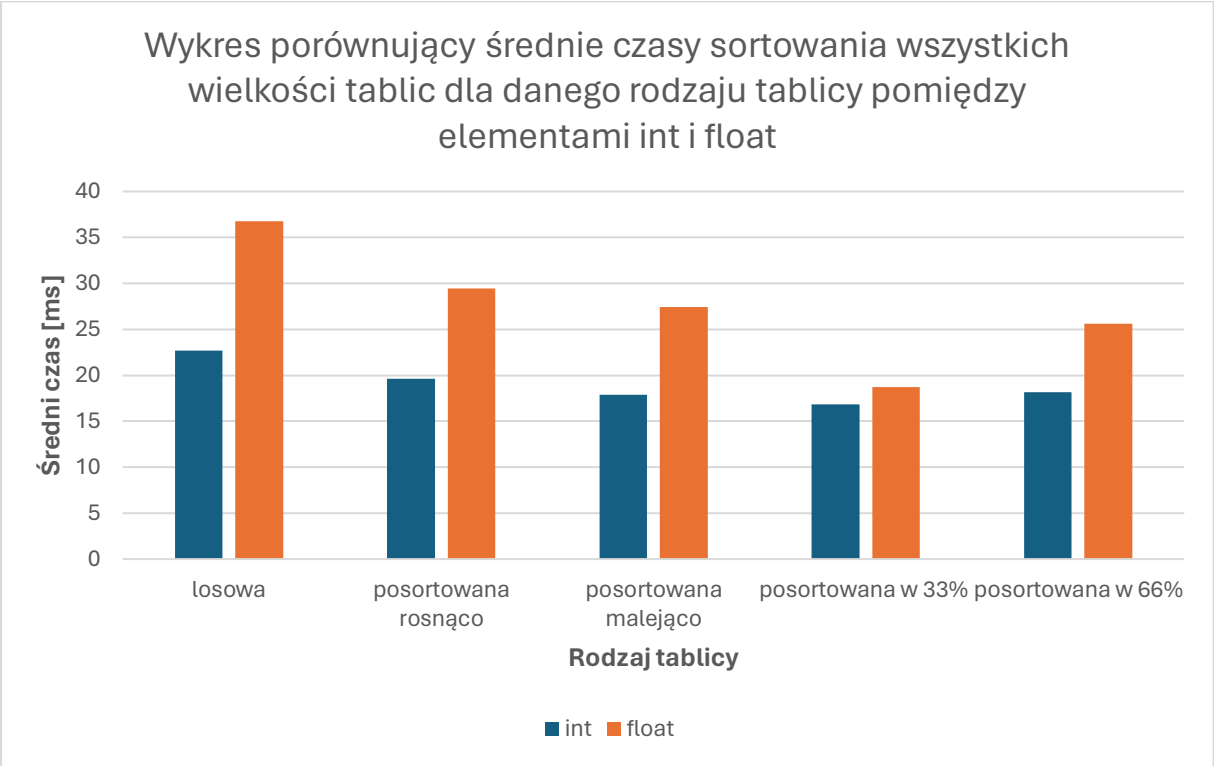
Średnie czasy [ms] sortowania tablic algorytmem sortowania przez kopcowanie							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa (float)	2,92	6,58	19,28	22,96	47,41	51,46	106,80
posortowana rosnąco (float)	1,92	3,84	9,52	11,36	23,12	26,48	54,78
posortowana malejąco (float)	2,45	6,12	14,86	17,02	32,50	34,64	71,52
posortowana w 33% (float)	2,54	5,92	16,40	17,82	37,16	41,72	84,40
posortowana w 66% (float)	2,14	5,76	15,86	18,62	29,56	39,60	80,64



Rysunek 4: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco i w 33% i 66% dla algorytmu sortowania przez kopcowanie – (int)



Rysunek 5: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco i w 33% i 66% dla algorytmu sortowania przez kopcowanie – (float)



Rysunek 6: Wykres porównujący średnie czasy sortowania wszystkich wielkości tablic łącznie dla danego rodzaju tablicy pomiędzy elementami int i float

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania przez kopcowanie.

Dla każdego rodzaju generowanej tablicy zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu liniowo logarytmiczna, co zgadza się z teoretyczną złożonością obliczeniową badanego algorytmu – $O(n\log n)$.

Najlepszym przypadkiem dla algorytmu sortowania przez kopcowanie jest przypadek, w którym sortowana jest tablica rosnąca – czas sortowania jest wtedy najkrótszy. Dla tablic posortowanych malejąco, w 33% i w 66% czas sortowania jest nieco dłuższy niż dla tablic posortowanych rosnąco, natomiast dla tablic losowych czas sortowania jest wyraźnie najdłuższy.

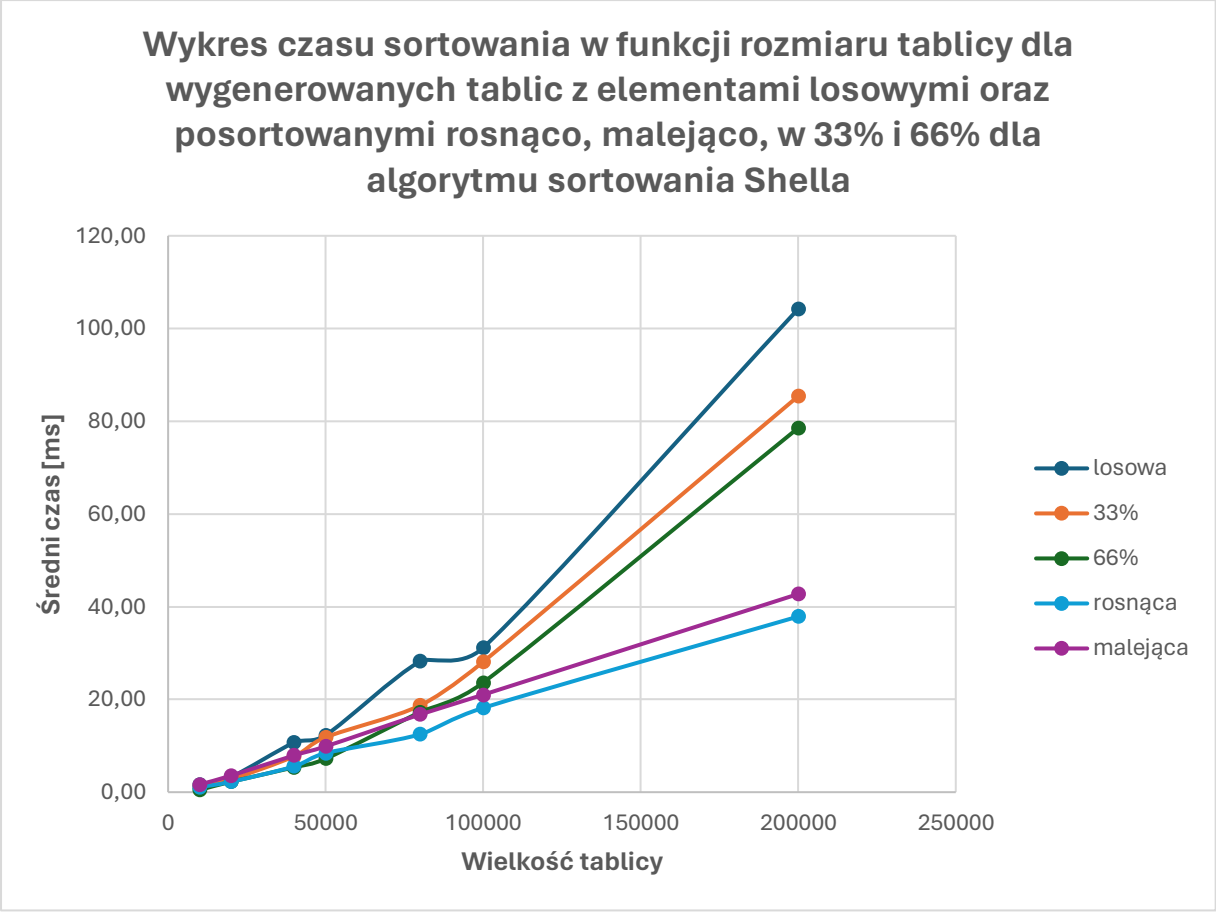
Sortowanie liczb rzeczywistych zdecydowanie wydłuża średni czas sortowania dla każdego rodzaju generowanej tablicy początkowej, w porównaniu z sortowaniem liczb całkowitych. Jednak mimo wydłużonego czasu sortowania, kolejność szybkości sortowania dla kolejnych rodzajów tablic jest zachowana.

Algorytm sortowania przez kopcowanie jest stabilnym i wydajnym algorytmem dla wszystkich rodzajów tablic początkowych.

Wyniki dla algorytmu sortowania Shella o złożoności $O(n^2)$:

Tabela 4: Tabela średnich czasów sortowania tablic algorytmem sortowania Shella ze złożonością $O(n^2)$

Średnie czasy [ms] sortowania tablic algorytmem sortowania Shella (złożoność O(n^2))							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	1,60	3,28	10,70	12,30	28,23	31,17	104,23
posortowana rosnąco	1,08	2,26	5,56	8,42	12,48	18,18	37,88
posortowana malejąco	1,67	3,58	7,94	9,88	16,76	21,02	42,74
posortowana w 33%	1,20	2,78	7,74	11,84	18,70	28,12	85,42
posortowana w 66%	0,48	2,26	5,36	7,22	17,28	23,64	78,56



Rysunek 7: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania Shella o złożoności obliczeniowej $O(n^2)$

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania Shella o złożoności $O(n^2)$.

Algorytm sortowania Shella jest ulepszonym algorytmem sortowania przez wstawianie, dlatego podobnie jak dla niego najkrótszy czas sortowania jest dla tablic posortowanych rosnąco. Wyjątkiem jest tutaj tablica posortowana malejąco, która średni czas sortowania ma bardzo zbliżony do czasu dla tablic posortowanych rosnąco. Wynika to z faktu, że wielokrotnie sortując kolejne podtablice złożone z elementów odległych od odstępy algorytm w dużej mierze posortuje już daną tablicę i przechodząc do ostatniego sprawdzania (całej tablicy), większość elementów będzie jest posortowana.

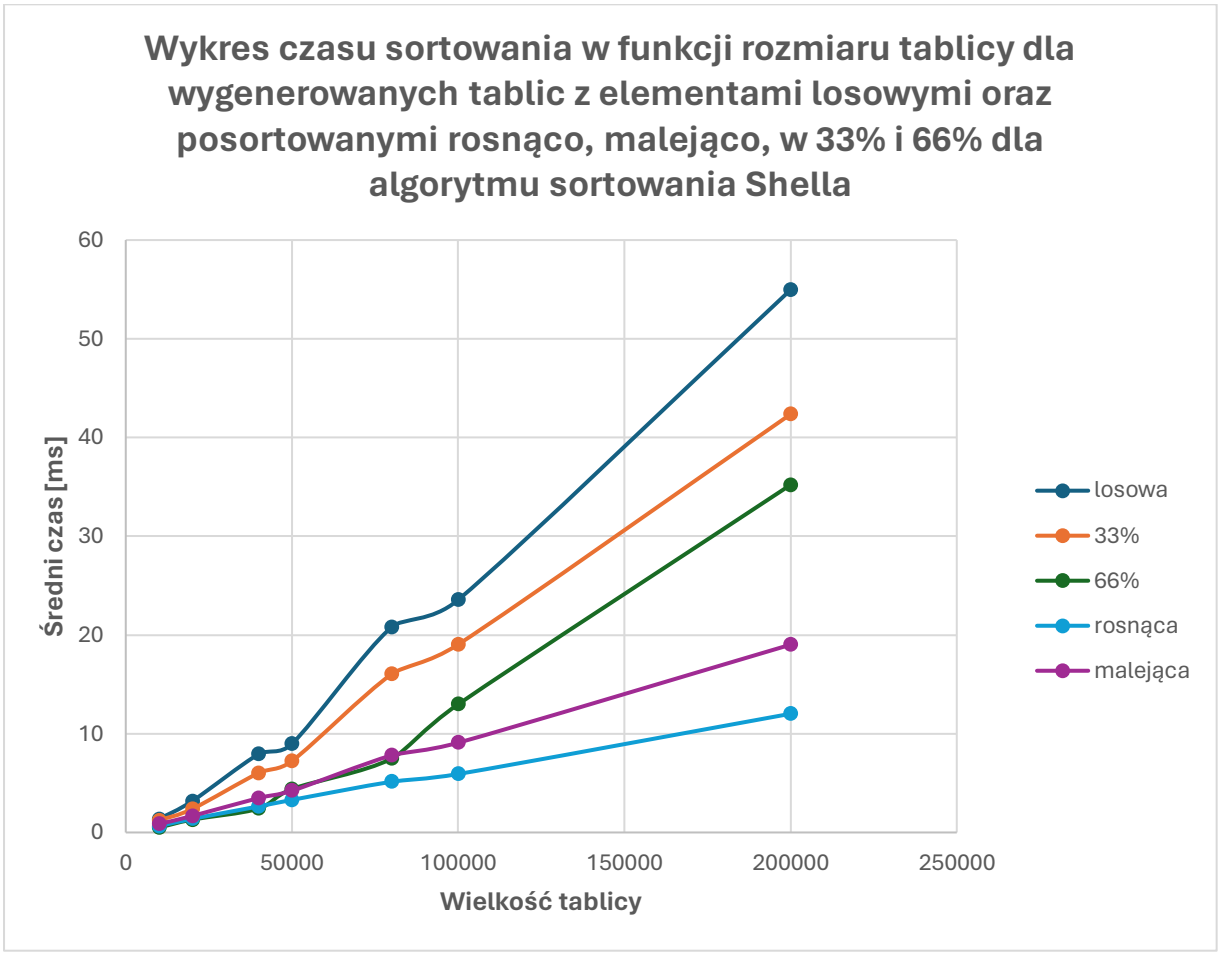
Dla tablic losowych oraz częściowo posortowanych średni czas sortowania jest znacznie dłuższy od poprzednich dwóch przypadków, a zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu kwadratowa. Jest to zgodne z oczekiwaną złożonością obliczeniową – $O(n^2)$.

Algorytm sortowania Shella o złożoności $O(n^2)$ jest dosyć stabilnym i wydajnym algorytmem dla wszystkich rodzajów tablic początkowych. Jednak najbardziej efektywny jest dla tablic z elementami posortowanymi rosnąco oraz malejąco.

Wyniki dla algorytmu sortowania Shella o złożoności O($n^{\frac{4}{3}}$):

Tabela 5: Tabela średnich czasów sortowania tablic algorytmem sortowania Shella ze złożonością O($n^{4/3}$)

Średnie czasy [ms] sortowania tablic algorytmem sortowania Shella (złożoność O(n^4/3))							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	1,36	3,18	7,94	8,98	20,81	23,56	54,96
posortowana rosnąco	0,64	1,38	2,64	3,3	5,14	5,92	12,02
posortowana malejąco	0,86	1,67	3,49	4,23	7,81	9,09	19,02
posortowana w 33%	1,23	2,38	6,03	7,24	16,04	19,04	42,36
posortowana w 66%	0,5	1,3	2,42	4,4	7,48	12,98	35,21



Rysunek 8: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania Shella o złożoności obliczeniowej O($n^{4/3}$)

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania Shella o złożoności $O(n^{4/3})$.

Algorytm sortowania Shella jest ulepszonym algorytmem sortowania przez wstawianie, dlatego podobnie jak dla niego najkrótszy czas sortowania jest dla tablic posortowanych rosnąco. Wyjątkiem jest tutaj tablica posortowana malejąco, która średni czas sortowania ma bardzo zbliżony do czasu dla tablic posortowanych rosnąco. Wynika to z faktu, że wielokrotnie sortując kolejne podtablice złożone z elementów odległych od odstępy algorytm w dużej mierze posortuje już daną tablicę i przechodząc do ostatniego sprawdzania (całej tablicy), większość elementów będzie jest posortowana.

Dla tablic losowych oraz częściowo posortowanych średni czas sortowania jest znacznie dłuższy od poprzednich dwóch przypadków, a na podwójny przyrost wielkości tablicy przyrost średniego czasu sortowania wynosi około 2,5 raza. Wyniki te są zgodne z oczekiwaną złożonością obliczeniową – $O(n^{4/3})$.

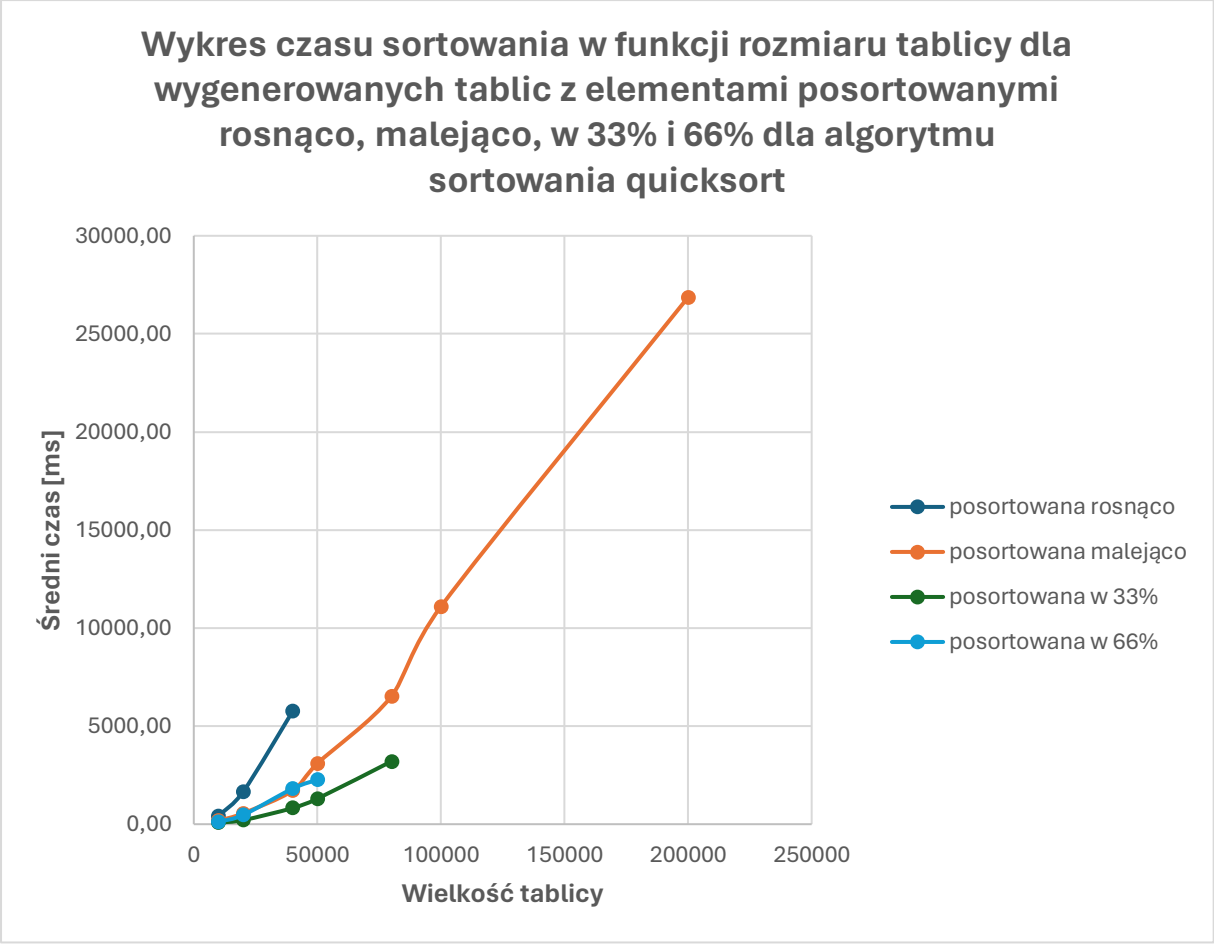
Algorytm sortowania Shella o złożoności $O(n^{4/3})$ jest dosyć stabilnym i wydajnym algorytmem dla wszystkich rodzajów tablic początkowych. Jednak najbardziej efektywny jest dla tablic z elementami posortowanymi rosnąco oraz malejąco.

Porównując algorytmy Shella dla dwóch różnych, badanych złożoności obliczeniowych, algorytm o złożoności $O(n^{4/3})$ jest zgodnie z oczekiwaniami znacznie wydajniejszy od algorytmu o złożoności $O(n^2)$. Średnie czasy sortowania są dla niego dużo krótsze.

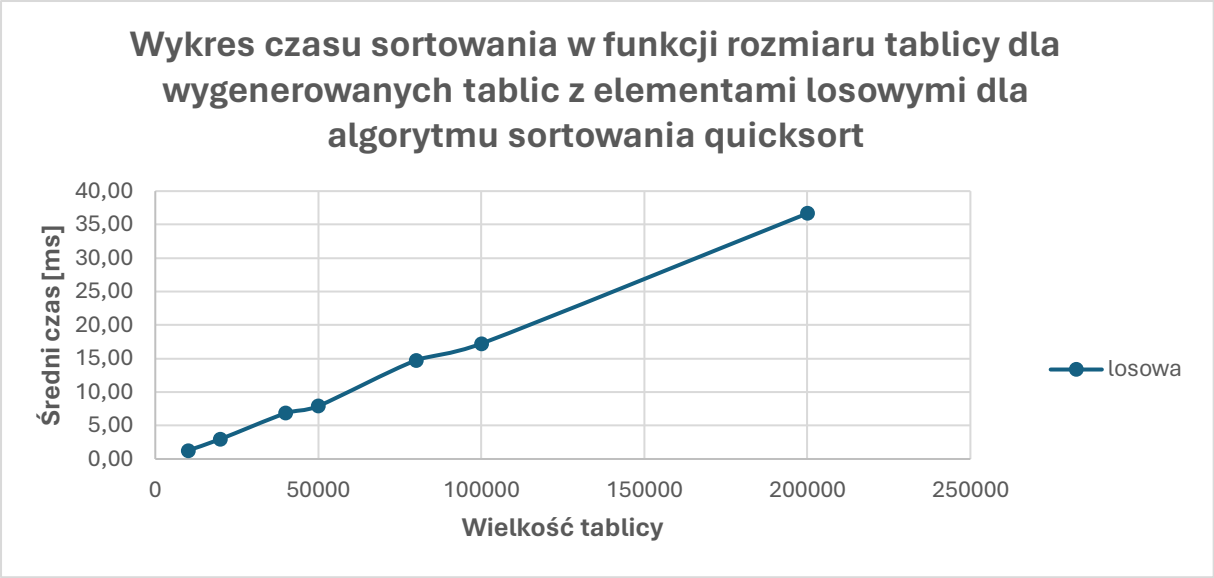
Wyniki dla algorytmu sortowania quicksort ze skrajnie lewym pivotem:

Tabela 6: Tabela średnich czasów sortowania tablic algorytmem sortowania quicksort ze skrajnie lewym pivotem

Średnie czasy [ms] sortowania tablic algorytmem quicksort ze skrajnie lewym pivotem							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	1,27	2,98	6,84	7,92	14,74	17,23	36,67
posortowana rosnąco	417,60	1660,00	5758,37	-	-	-	-
posortowana malejąco	177,40	557,72	1710,62	3091,88	6523,08	11096,72	26864,40
posortowana w 33%	81,6	208,64	825	1301,06	3206,18	-	-
posortowana w 66%	120,44	469,88	1817,06	2264,74	-	-	-



Rysunek 9: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania quicksort ze skrajnie lewym pivotem



Rysunek 10: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi dla algorytmu sortowania quicksort ze skrajnie lewym pivotem

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania quicksort.

Dla tablic losowych zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu liniowo logarytmiczna, co jest zgodne z oczekiwaną złożonością obliczeniową – $O(n \log n)$.

Dla tablic posortowanych rosnąco, malejąco oraz częściowo występują pesymistyczne przypadki algorytmu quicksort ze skrajnie lewym pivotem, dlatego ich zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu kwadratowa. Co zgadza się z teoretyczną złożonością obliczeniową dla pesymistycznych przypadków – $O(n^2)$.

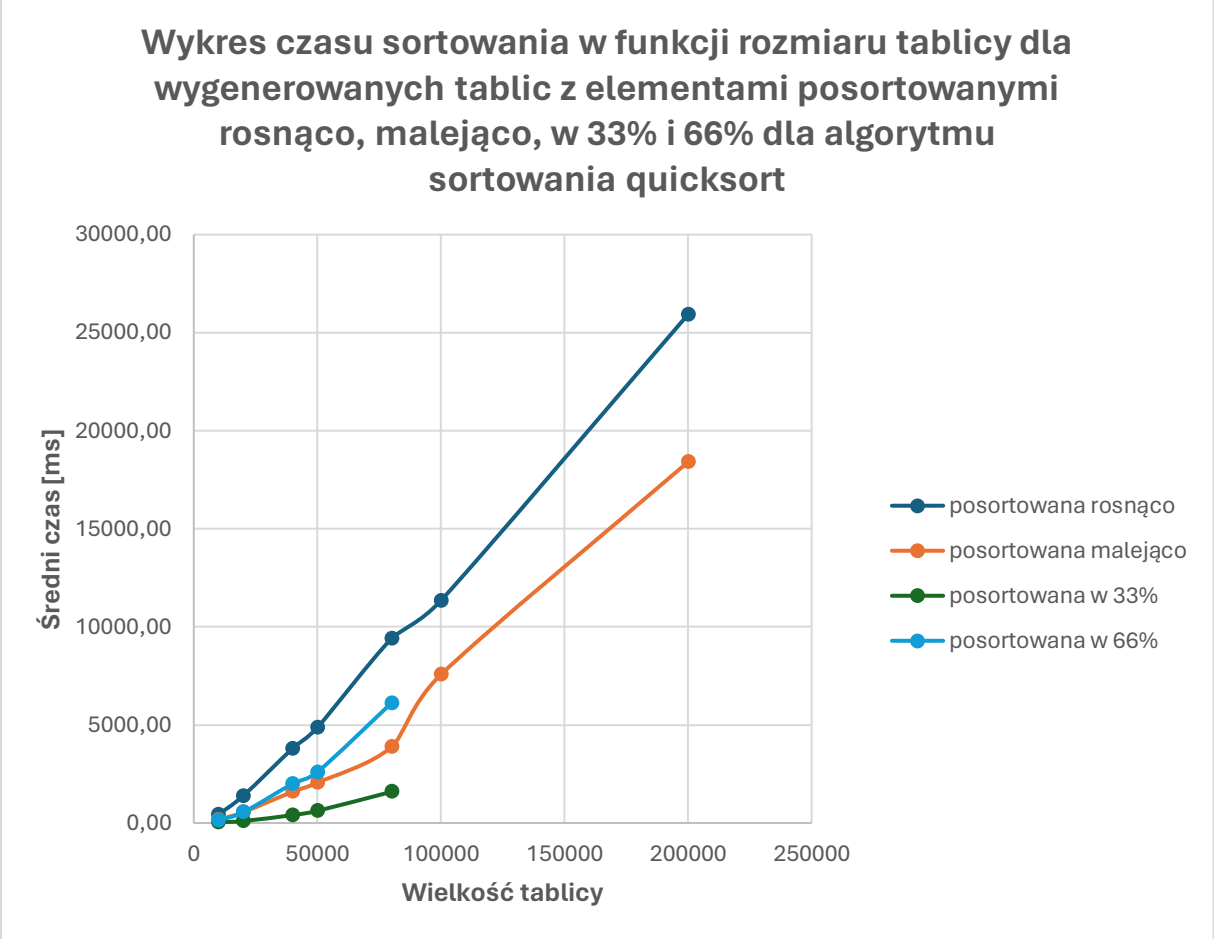
Dla tablic posortowanych rosnąco oraz częściowo, dla większej ilości elementów program automatycznie kończył działanie podczas próby posortowania tablicy. Wynika to z tego, że podczas sortowania występowało zjawisko przepełnienia stosu, ponieważ rekurencyjne wywołanie funkcji quicksort było powtarzane ogromną liczbę razy (co każdy jeden element), przez co stos, który przechowywał informacje o bieżących wywołaniach funkcji przepełniał się powodując automatyczne zatrzymanie działania programu.

Dla tablic losowych algorytm sortowania quicksort ze skrajnie lewym pivotem jest wydajnym algorytmem, natomiast dla reszty przypadków jest on mało efektywny.

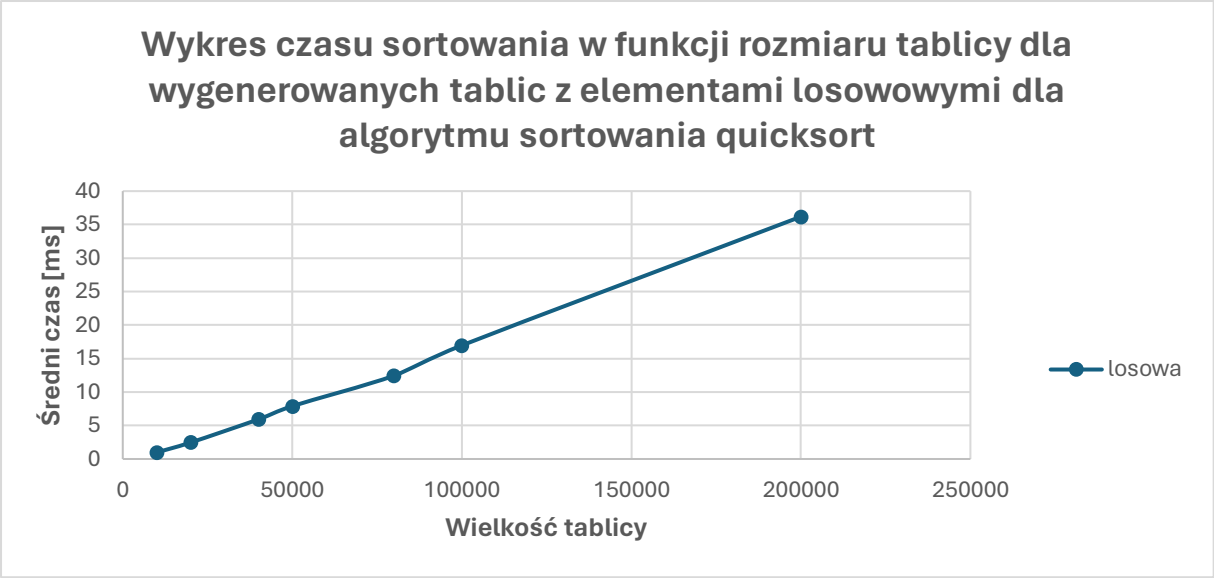
Wyniki dla algorytmu sortowania quicksort ze skrajnie prawym pivotem:

Tabela 7: Tabela średnich czasów sortowania tablic algorytmem sortowania quicksort ze skrajnie prawym pivotem

Średnie czasy [ms] sortowania tablic algorytmem quicksort ze skrajnie prawym pivotem							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	0,98	2,46	5,92	7,89	12,44	16,96	36,18
posortowana rosnąco	436,20	1388,14	3802,92	4869,68	9401,22	11343,12	25932,38
posortowana malejąco	173,60	551,72	1593,64	2052,44	3878,10	7566,26	18409,18
posortowana w 33%	35,54	100,82	402,66	620,1	1591,66	-	-
posortowana w 66%	142,44	547,44	2007,72	2596,52	6121,48	-	-



Rysunek 11: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania quicksort ze skrajnie prawym pivotem



Rysunek 12: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi dla algorytmu sortowania quicksort ze skrajnie prawym pivotem

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania quicksort.

Dla tablic losowych zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu liniowo logarytmiczna, co jest zgodne z oczekiwaną złożonością obliczeniową – $O(n \log n)$.

Dla tablic posortowanych rosnąco, malejąco oraz częściowo występują pesymistyczne przypadki algorytmu quicksort ze skrajnie prawym pivotem, dlatego ich zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu kwadratowa. Co zgadza się z teoretyczną złożonością obliczeniową dla pesymistycznych przypadków – $O(n^2)$.

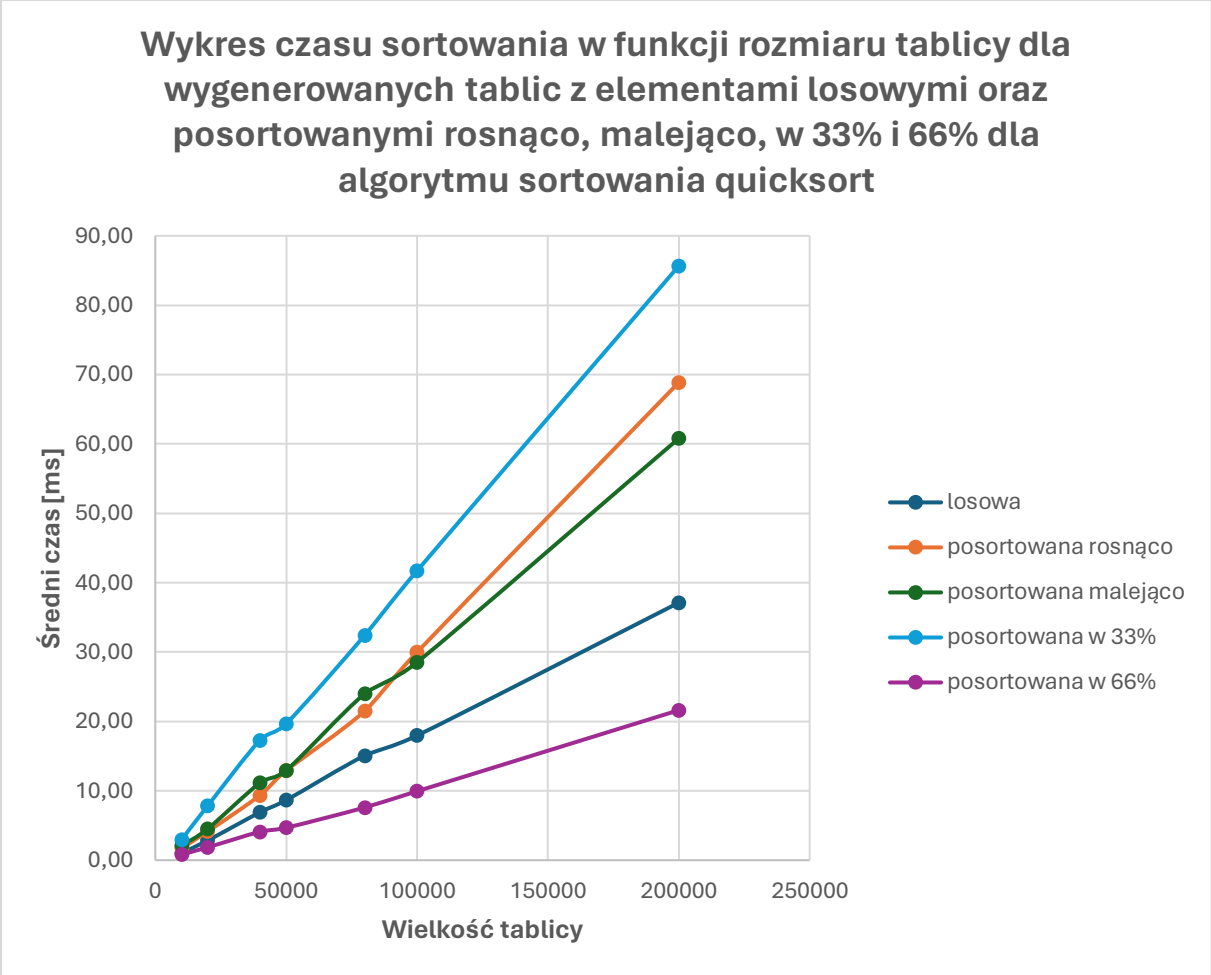
Dla tablic posortowanych częściowo, dla większej ilości elementów program automatycznie kończył działanie podczas próby posortowania tablicy. Wynika to z tego, że podczas sortowania występowało zjawisko przepełnienia stosu, ponieważ rekurencyjne wywołanie funkcji quicksort było powtarzane ogromną liczbę razy, przez co stos, który przechowywał informacje o bieżących wywołaniach funkcji przepełniał się powodując automatyczne zatrzymanie działania programu.

Podobnie jak dla algorytmu quicksort ze skrajnie lewym pivotem, dla tablic losowych algorytm sortowania quicksort ze skrajnie prawym pivotem jest wydajnym algorytmem, natomiast dla reszty przypadków jest on mało efektywny.

Wyniki dla algorytmu sortowania quicksort ze środkowym pivotem:

Tabela 8: Tabela średnich czasów sortowania tablic algorytmem sortowania quicksort ze środkowym pivotem

Średnie czasy [ms] sortowania tablic algorytmem quicksort ze środkowym pivotem							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	1,04	2,84	6,91	8,66	15,08	17,97	37,12
posortowana rosnąco	1,74	4,14	9,36	12,92	21,50	29,98	68,86
posortowana malejąco	2,08	4,52	11,16	12,94	24,00	28,52	60,80
posortowana w 33%	2,94	7,84	17,3	19,68	32,42	41,72	85,66
posortowana w 66%	0,83	1,86	4,1	4,68	7,6	9,94	21,62



Rysunek 13: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania quicksort ze środkowym pivotem

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania quicksort.

Dla każdego rodzaju początkowej tablicy zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu liniowo logarytmiczna, co jest zgodne z oczekiwaną złożonością obliczeniową – $O(n \log n)$.

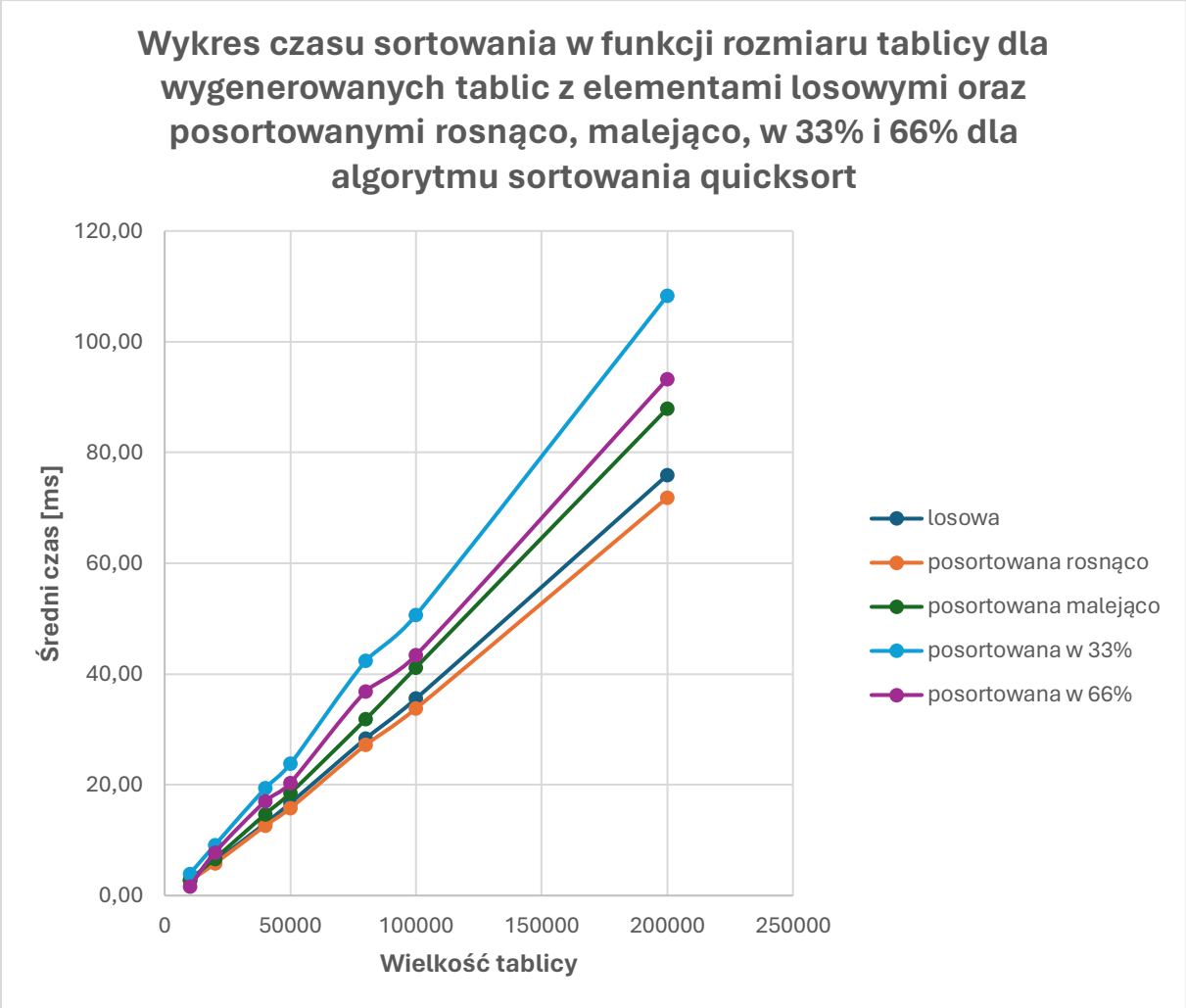
Najlepszym przypadkiem dla algorytmu sortowania quicksort ze środkowym pivotem jest przypadek, w którym sortowane są tablice posortowane w 66%. Średni czas sortowania dla tego przypadku jest wyraźnie najmniejszy. Drugim najlepszym przypadkiem jest sortowanie tablic losowych, a następnie posortowanych malejąco i rosnąco. Pesymistycznym przypadkiem jest przypadek, w którym sortowane są tablice posortowane w 33%. W porównaniu z najlepszym przypadkiem, średni czas sortowania dla tego przypadku jest wielokrotnie dłuższy.

Algorytm sortowania quicksort ze środkowym pivotem jest stabilnym i wydajnym algorytmem dla wszystkich rodzajów tablic początkowych.

Wyniki dla algorytmu sortowania quicksort z losowym pivotem:

Tabela 9: Tabela średnich czasów sortowania tablic algorytmem sortowania quicksort z losowym pivotem

Średnie czasy [ms] sortowania tablic algorytmem quicksort z losowym pivotem							
Tablica\Ilość elementów [tyś]	10	20	40	50	80	100	200
losowa	2,66	6,10	13,04	16,70	28,36	35,56	75,93
posortowana rosnąco	2,68	5,78	12,62	15,78	27,21	33,78	71,86
posortowana malejąco	2,90	6,66	14,70	18,50	31,82	41,18	87,92
posortowana w 33%	3,94	9,12	19,38	23,78	42,36	50,64	108,28
posortowana w 66%	1,58	7,78	17,04	20,36	36,87	43,4	93,26



Rysunek 14: Wykres czasu sortowania w funkcji rozmiaru tablicy dla wygenerowanych tablic z elementami losowymi oraz posortowanymi rosnąco, malejąco, w 33% i 66% dla algorytmu sortowania quicksort z losowym pivotem

Wnioski:

Analizując tabelę wyników oraz powyższe wykresy, można zauważyć, że otrzymane wyniki dla każdego rodzaju tablicy i dla każdego badanego typu danych są zgodne z oczekiwaną złożonością obliczeniową algorytmu sortowania quicksort.

Dla każdego rodzaju początkowej tablicy zależność między średnim czasem sortowania a wielkością tablic jest w przybliżeniu liniowo logarytmiczna, co jest zgodne z oczekiwaną złożonością obliczeniową – $O(n \log n)$.

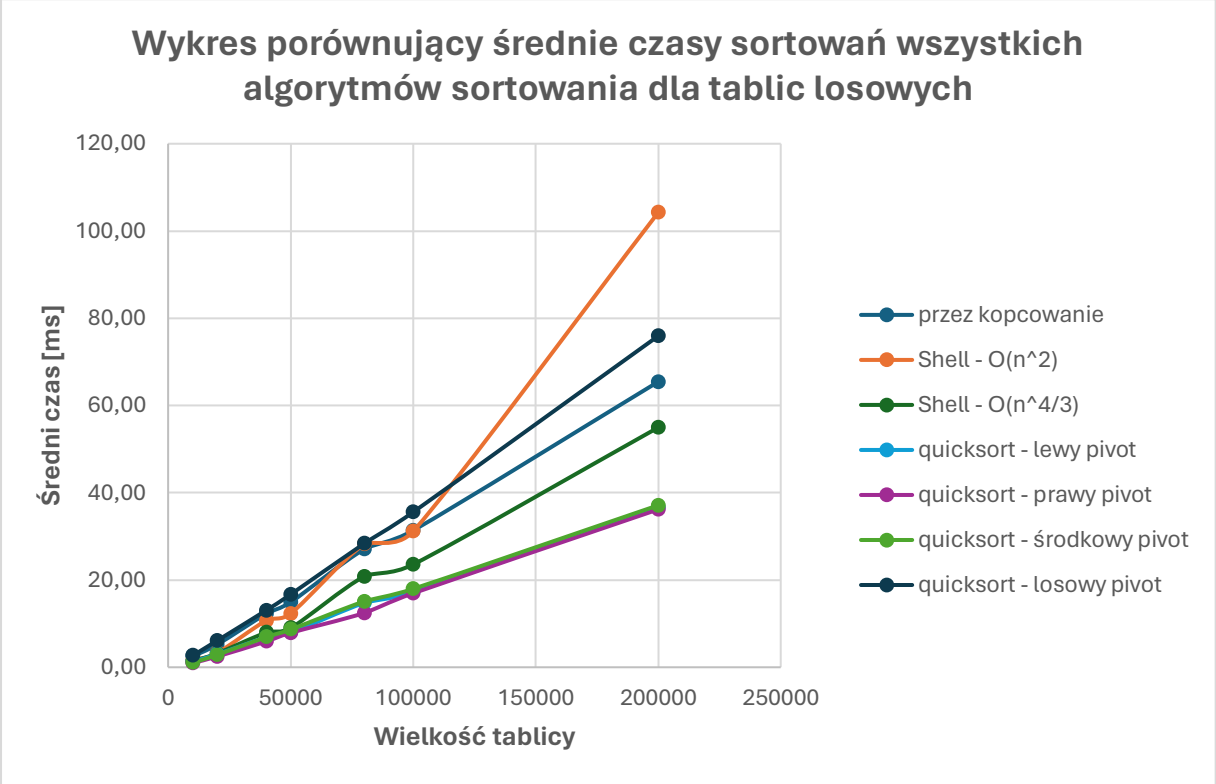
Najkrótszy średni czas sortowania dla algorytmu sortowania quicksort z losowym pivotem jest dla sortowania tablic posortowanych rosnąco oraz tablic losowych. Trochę mniej wydajne jest sortowanie tablic posortowanych malejąco i posortowanych w 66%. Średni czas sortowania jest lekko dłuższy od najlepszego przypadku. Natomiast pesymistycznym przypadkiem jest przypadek, w którym sortowane są tablice posortowane w 33%.

Algorytm sortowania quicksort z losowym pivotem jest stabilnym i wydajnym algorytmem dla wszystkich rodzajów tablic początkowych.

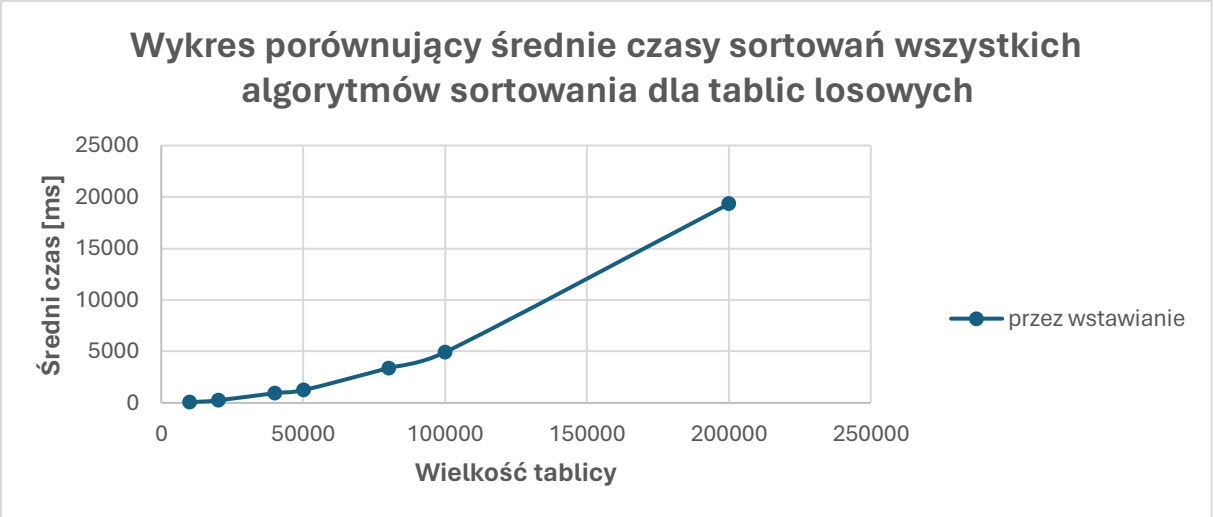
Porównując wszystkie cztery przypadki algorytmu sortowania quicksort, najbardziej uniwersalnym i wydajnym przypadkiem jest ten ze środkowym pivotem, drugim najbardziej efektywnym i stabilnym przypadkiem jest ten z losowym pivotem, natomiast przypadki ze skrajnymi pivotami są dla ogółu najmniej skuteczne, chyba że patrzy się jedynie na tablice losowe, to wtedy są one najwydajniejszymi przypadkami.

Podsumowanie

Porównanie średnich czasów sortowań wszystkich badanych algorytmów sortowania dla tablic losowych z elementami będącymi liczbami całkowitymi:



Rysunek 15: Wykres porównujący średnie czasy sortowań wszystkich algorytmów sortowania dla tablic losowych z wyłączeniem algorytmu sortowania przez wstawianie z powodu za wysokich czasów sortowań



Rysunek 16: Wykres średnich czasów sortowania dla algorytmu sortowania przez wstawianie dla tablic losowych

Końcowe wnioski:

Analizując wszystkie otrzymane wyniki można dojść do wniosku, że:

- Dla tablic losowych najefektywniejszym algorytmem jest algorytm sortowania quicksort ze skrajnymi pivotami.
- Dla tablic posortowanych rosnąco najefektywniejszym algorytmem jest algorytm sortowania przez wstawianie.
- Dla tablic posortowanych malejąco najefektywniejszym algorytmem jest algorytm sortowania Shella o złożoności obliczeniowej $O(n^{4/3})$.
- Dla tablic posortowanych w 33% najefektywniejszym algorytmem jest algorytm sortowania Shella o złożoności obliczeniowej $O(n^{4/3})$.
- Dla tablic posortowanych w 66% najefektywniejszym algorytmem jest algorytm sortowania quicksort ze środkowym pivotem.
- Najbardziej stabilnym algorytmem sortowania jest algorytm sortowania przez kopcowanie, dla którego średnie czasy sortowań dla każdego rodzaju generowanej tablicy są do siebie bardzo zbliżone i rosną zgodnie ze złożonością liniowo logarytmiczną przez co są też bardzo niskie. To czyni algorytm przez kopcowanie bardzo wydajnym i uniwersalnym algorytmem.
- Algorytm sortowania quicksort ze środkowym oraz loswym pivotem także są bardzo stabilnymi i efektywnymi algorytmami. Średnie czasy sortowań dla każdego rodzaju generowanej tablicy również są do siebie zbliżone i utrzymują się na niskim poziomie. To czyni ten algorytm dla tych dwóch przypadków również bardzo uniwersalnym i wydajnym algorytmem.

Bibliografia

[1] Prezentacje Doktora Jarosława Mierzwy

[2] https://youtu.be/2DmK_H7IdTo?si=3eensUz6lntuemGV

[3] https://eduinf.waw.pl/inf/alg/003_sort/index.php

[4] https://pl.wikipedia.org/wiki

[5] https://www.youtube.com/@MichaelSambol