Algorytmy i złożoność obliczeniowa - badanie złożoności obliczeniowej algorytmów sortowania

Grzegorz Wszoła

Zadanie projektowe 1	
Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową	
Prowadzący	Jarosław Mierzwa
Termin zajęć	11:15 WT NP
Termin oddania	08.04.2025

Spis treści

1 Wstęp - badane algorytmy

1.1 Insertion sort

Sortowanie przez wstawianie (Insertion sort) - prosty algorytm sortowania działający na zasadzie dzielnia tablicy na dwie części - posortowaną i nieposortowaną. Następnie wstawiania elementy w odpowiednie miejsce w już posortowanej części tablicy iterując po części nieposortowaniej. Działa iteracyjnie – pobiera kolejny element i przesuwa go w lewo, aż znajdzie właściwą pozycję.

1.1.1 Złożoność obliczeniowa

Dla tablicy posiadającej n elementów

- Najlepszy przypadek - gdy tablica jest już posortowana (w tym przypadku rosnąco)

$$(n-1) \in O(n)$$

- Dla danych losowych elementy wstawiają się w środek posortowanej części tablicy, co daje średnią liczbę przesunięć $\frac{n}{2}$ na każdą iterację.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \cdot \frac{(n-1)n}{2} = \frac{(n-1)n}{4} \in O(n^2)$$

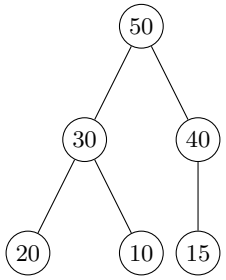
Złożoność pamięciowa algorytmu to O(1), ponieważ algorytm wykonywany jest w miejscu nie potrzebuje dodatkowej pamięci i jest stabilny (zachowuje kolejność równych elementów).

1.2 Heap Sort

Sortowanie przez kopcowanie polega na strukturze danych nazywanej kopiec (heap), działa w dwóch głównych etapach budowaniu/odbudowywaniu kopca i sortowaniu.
Struktura danych kopiec (heap) to pełne drzewo binarne, które spełnia własność kopca, gdzie każdy węzeł:[1]

- Jest zawsze większy niż jego węzły potomne, a klucz węzła korzenia jest największy spośród wszystkich innych węzłów. Ta własność nazywana jest własnością kopca maksymalnego (max heap property).
- Jest zawsze mniejszy niż jego węzły potomne, a klucz węzła korzenia jest najmniejszy spośród wszystkich innych węzłów. Ta własność nazywana jest własnością kopca minimalnego (min heap property).

W tym przypadku zastosowany został kopiec maksymalny. Największy element (korzeń kopca) jest zamieniany z ostatnim elementem i usuwany z kopca, a następnie przeprowadzana jest operacja heapify dla przywrócenia własności kopca. Proces powtarza się, aż kopiec zostanie opróżniony.



Rysunek 1: Przykład kopca binarnego maksymalnego

[1] `https://www.programiz.com/dsa/heap-data-structure`

1.2.1 Złożoność obliczeniowa

Wysokość kopca binarnego z n elementami wynosi $logn$. Złożoność obliczeniową algorytmu Heap Sort możemy podzielić na dwa etapy:

- Budowa kopca – operacja heapify w najgorszym przypadku musi przejść przez całą wysokość kopca, co ma złożoność obliczeniową $O(\log n)$. Jeżeli przeprowadzimy tę operację dla każdego węzła od dolnych do górnych poziomów, całkowity czas budowy kopca wyniesie $O(n)$.
- Sortowanie – podczas sortowania będziemy musieli przejść przez każdy element kopca, przeprowadzając $n-1$ operacji wymiany i heapify dla każdego z tych elementów. Każda operacja heapify ma złożoność $O(\log n)$, więc całkowity czas sortowania wyniesie $O(n \log n)$.

Z tego wynika, że całkowita złożoność obliczeniowa algorytmu Heap Sort wynosi $O(n \log n)$ w najgorszym przypadku. Jego złożoność pamięciowa to O(1) oraz nie jest on stabilny.

1.3 Sortowanie Shella

Sortowanie Shella to algorytm sortowania, który jest ulepszeniem sortowania bąbelkowego. Zamiast porównywać sąsiednie elementy, sortowanie Shella porównuje elementy w odległych od siebie parach, zmniejszając stopniowo tę odległość, aż do momentu, gdy wykonuje porównania sąsiednich elementów.

1. Zaczynamy od wyboru sekwencji skoków (w programie został użyty zwykły algorytm shella z dzieleniem na 2 i algorytm Sedgewicka $gap_k = 3 \cdot gap_k + 1$)
2. Następnie algorytm porównuje i zamienia elementy jak w zwykłym sortowaniu bąbelkowym
3. Zmniejsza 'gap' pomiędzy porównywanymi elementami

1.3.1 Złożoność obliczeniowa

Z powodu użycia dwóch różnych sekwencji w tym algorytmie występują dwie różne złożoności obliczeniowe, zależne od doboru przyrostów:

- Dla sekwencji dzielenia przez 2 $\left(\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1\right)$, czyli oryginalnej wersji Shella, złożoność w najgorszym przypadku wynosi $O(n^2)$.
- Dla sekwencji Sedgewicka (naprzemienne wartości wyliczane według wzoru $9 \cdot 4^k - 9 \cdot 2^k + 1$ lub alternatywnego $4^k + 3 \cdot 2^{k-1} + 1$):
  - Złożoność w najgorszym przypadku: $O(n^{4/3})$
  - Średnia złożoność: $O(n \log^2 n)$, co czyni ją jedną z bardziej efektywnych sekwencji.

Zależnie od sekwencji, złożoność Shell Sort waha się od $O(n^2)$ do nawet $O(n \log n)$ dla najlepszych empirycznie wyznaczonych wartości przyrostów. Złożoność pamięciowa wynosi $O(1)$, jednak algorytm nie jest stabilny.[2]

1.4 Quick Sort

Quick Sort to wydajny algorytm sortowania oparty na strategii "dziel i rządź". Działa poprzez wybór elementu zwanego pivotem, a następnie podział tablicy na dwie części: mniejsze od pivota i większe od pivota. Proces jest rekurencyjnie powtarzany dla obu części, aż tablica zostanie posortowana.
Pivot można wybrać na różne sposoby w programie użyte zostały metody:

- maksymalny lewy
- maksymalny prawy
- losowy
- środkowy

[2] Jako pomoc została użyta strona: `https://www.baeldung.com/java-shell-sort`

Wybór pivota ma wpływ na złożoność obliczeniową algorytmu. Na przykład wybór skrajnie lewego elementu jako pivota w przypadku tablicy posortowanej w odwrotnej kolejności może znacznie wydłużyć czas sortowania, prowadząc do najgorszej złożoności $O(n^2)$

1.4.1 Złożoność obliczeniowa

Jeśli jako pivot wybieramy zawsze pierwszy lub ostatni element, złożoność Quick Sort może ulec pogorszeniu. Wtedy algorytm nie dzieli tablicy równomiernie, tylko wydziela jedną bardzo małą część (z jednym elementem) i resztę tablicy. Oznacza to, że zamiast podziału $O(logn)$ (jak w optymalnym przypadku), mamy n poziomów rekurencji, a każde wywołanie wykonuje $O(n)$ operacji podziału:

$$O(n + (n-1) + (n-2) + ... + 1) = O(n^2)$$

Jednak dla średniego przypadku gdy tablica nie jest wcześniej posortowana jest zbliżona do $O(nlogn)$

Przy losowym wyborze pivotu, w przeciętnym przypadku każda iteracja dzieli tablicę na części w mniej więcej równych proporcjach. Oznacza to, że mamy około $logn$ poziomów rekurencji (bo za każdym razem tablica zmniejsza się o połowę), a każde wywołanie wykonuje $O(n)$ operacji podziału, więc złożoność obliczeniowa wynosi $O(nlogn)$

Mimo losowego wyboru, istnieje mała szansa, że pivoty będą wybierane w sposób bardzo nierówny, co może prowadzić do $O(n^2)$, ale w praktyce zdarza się to bardzo rzadko.

Wybieranie środkowego elementu tablicy daje podobne korzyści jak losowy pivot.
Dzięki temu, że pivot znajduje się bliżej środka, tablica jest zazwyczaj dzielona bardziej równomiernie, co prowadzi do optymalnej wydajności $O(nlogn)$.
Najgorszy przypadek może wystąpić, jeśli tablica jest posortowana, a Quick Sort wybiera zawsze środkowy element przedziału, który może nie zapewniać idealnego podziału. Jednak prawdopodobieństwo takiej sytuacji jest mniejsze niż w przypadku pivota skrajnego.

2 Plan eksperymentu

2.1 Rozmiary tablic

Jako rozmiary tablic zostały użyte wielokrotności potęgi dwójki (łącznie 7 przypadków).

$$\begin{gathered} 10 \cdot 10^3 \\ 20 \cdot 10^3 \\ 40 \cdot 10^3 \\ \vdots \\ 320 \cdot 10^3 \\ 640 \cdot 10^3 \end{gathered}$$

2.2 Sposób generowania tablic

Tablice generowane są w sposób dynamiczny, a dane są losowe lub wczytywane z pliku.

2.2.1 Generowanie losowych danych

Dane generowane są losowo używając rozkładu normalnego. Następnie tworzone są kopie tablicy, które spełniają inne warunki zadania (tablica posortowana w 33%, 66% itd.)

```
template<typename T>
    static T* generateRandomArray(int size, T minValue = 0, T maxValue = 0) {
        if (maxValue <= minValue) {
            maxValue = static_cast<T>(size - 1);
        }
        std::random_device rd;
        std::mt19937 gen(rd());
        if constexpr (std::is_integral<T>::value) {
            std::uniform_int_distribution<int> dist(static_cast<int>(minValue), static_cast<int>(
                maxValue));
            auto arr = new T[size];
            for (int i = 0; i < size; ++i) {
                arr[i] = static_cast<T>(dist(gen));
            }
            return arr;
        }
        else if constexpr (std::is_floating_point<T>::value) {
            std::uniform_real_distribution<T> dist(minValue, maxValue);
            auto arr = new T[size];
            for (int i = 0; i < size; ++i) {
                arr[i] = dist(gen);
            }
            return arr;
        }
        return nullptr;
    }
```

Listing 1: Metoda generująca losowe tablice na podstawie rozkładu normalnego

```
template<typename T>
    static T** generateTableCopies(int size, T* intTable) {
        auto tableCopies = new T*[5];
        auto tableCopy = new T[size];
        auto table33p = new T[size];
        auto table66p = new T[size];
        auto tableSorted = new T[size];
        auto tableReverseSorted = new T[size];
        memcpy(tableCopy, intTable, sizeof(T)*size);
        memcpy(table33p, intTable, sizeof(T)*size);
        memcpy(table66p, intTable, sizeof(T)*size);
        memcpy(tableSorted, intTable, sizeof(T)*size);
        memcpy(tableReverseSorted, intTable, sizeof(T)*size);
        std::sort(table33p, table33p + size/3);
        std::sort(table66p, table66p + (2 * size / 3));
        std::sort(tableSorted, tableSorted + size);
        std::sort(tableReverseSorted, tableReverseSorted + size, std::greater<T>());
        tableCopies[0] = tableCopy;
        tableCopies[1] = table33p;
        tableCopies[2] = table66p;
        tableCopies[3] = tableSorted;
        tableCopies[4] = tableReverseSorted;
        return tableCopies;
    }
```

Listing 2: Metoda generująca w części lub całkowicie posortowane tablice używając standardowej biblioteki C

2.3 Metoda mierzenia czasu

Czas mierzony jest obliczany za pomocą std::chrono::high_resolution_clock::now();, przykład użycia:

```
static void ShellSort(T* a, int size, double &time, bool type) {
        auto start = std::chrono::high_resolution_clock::now();
        .
        . //reszta funkcji
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::milli> elapsed = end - start;
        time = elapsed.count();
```

Listing 3: Przykład użycia funkcji chrono

2.4 Rodzaj danych

Badania zostały przeprowadzone dla różnych typów danych, w tym int oraz float, w celu analizy wpływu typu danych na czas wykonania algorytmów.

2.5 Komputer użyty podczas testów oraz warunki przeprowadzania

- Procesor działający podczas testów na prędkości 4.4GHz
- 8GB RAM
- System operacyjny Linux (Ubuntu)
- Temperatury procesora wachały się pomiędzy 60, a 90 stopni $°C$
- Każda z pętli była testowana 20 (aby czas sortowania był racjonalny) razy za każdym razem generując nowe dane i wyliczona była średnia arytmetyczna z wyników czasu

3 Przebieg eksperymentu

Dla każdego algorytmu zostało przeprowadzone sortowanie na 5 różnych typach tablicy (losowa, 33%posortowana, 66% posortowana, posortowana oraz posrtowana malejąco) oraz dwóch typów danych (int, float)

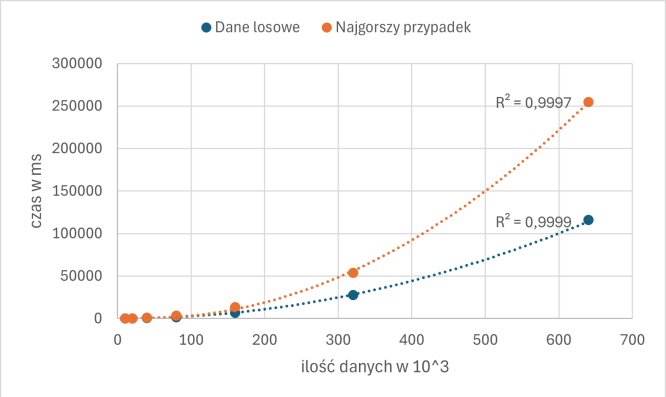
3.1 Insertion Sort

Wartości teoretyczne dla sortowania przez wstawianie: średni/najgorszy - $O(n^2)$, najlepszy - $O(n)$

Tabela 1: Zmierzony czas dla sortowania przez wstawianie

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	25,55	23,25	14,54	0,01516	53,59
20	111,07	96,11	59,81	0,03080	236,71
40	447,29	392,87	246,25	0,0623	865,37
80	1747,07	1510	931,87	0,1154	3352
160	6855,01	6110,49	3817,27	0,2590	13611,3
320	27667,4	24471,3	15251,5	0,5490	53717,3
640	115939	104242	66916,1	1,138	254866

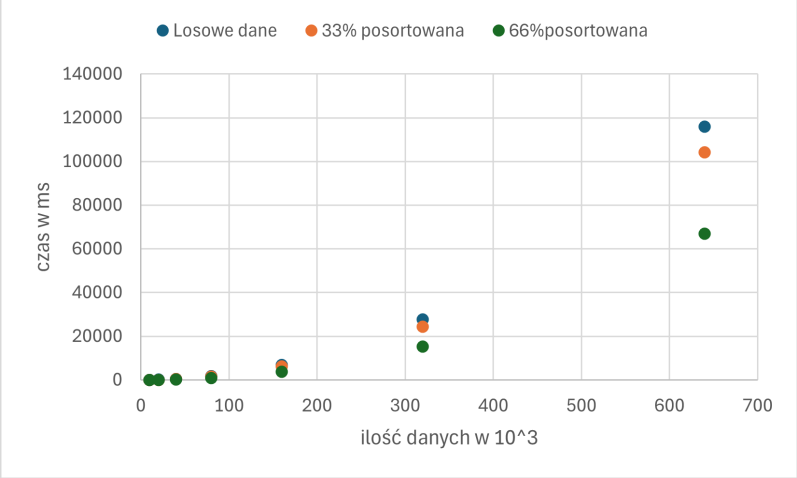
Na wykresie zarówno dla danych losowych, jak i najgorszego przypadku widzimy krzywe o kształcie charakterystycznym dla funkcji kwadratowej, a współczynnik determinacji R^2 bliski 1 potwierdza dobre dopasowanie modelu kwadratowego do wyników pomiarów.



Rysunek 2: Wykres czasu w ms do ilości danych

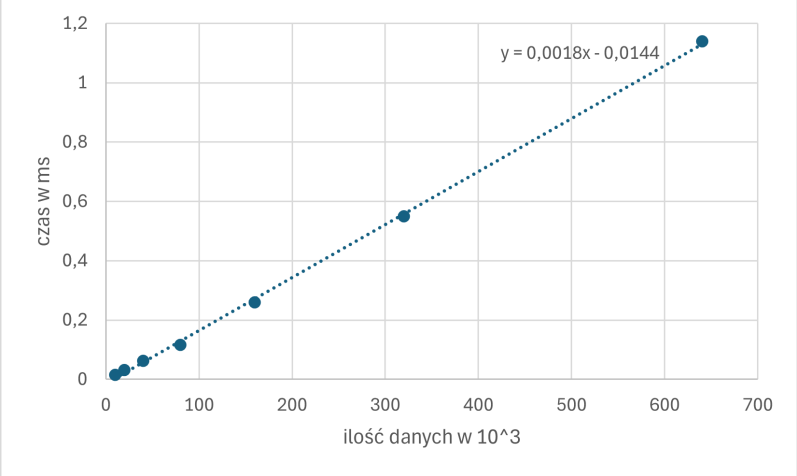
Jedną z zalet sortowania przez wstawianie jest jego efektywność w przypadku częściowo uporządkowanych danych. Gdy tablica jest wstępnie posortowana (lub zawiera tylko kilka nieuporządkowanych elementów) algorytm działa znacznie szybciej niż w przypadku całkowicie losowego rozkładu wartości. W najelpszy przypadku jest to jedno przejście przez tablice $O(n)$.

Z poniższego wykresu możemy zaobserwować skrócenie czasu sortowania dla tablic posortowanych w większej części.



Rysunek 3: Wykres czasu do ilości danych dla trzech różnych stopni posortowania

Insertion sort posiada złożoność obliczeniową $O(n)$ dla już posortowanej tablicy co widać na poniższym wykresie



Rysunek 4: Wykres czasu do ilości danych dla posortowanej tabeli

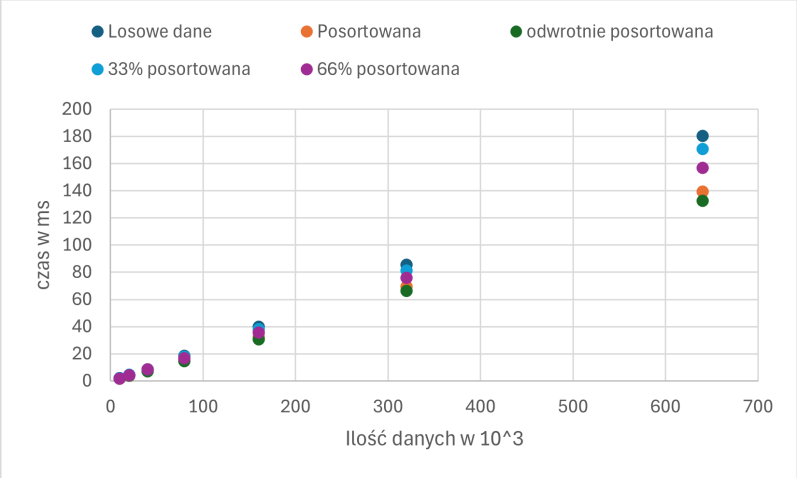
3.2 Heap Sort

Sortowanie przez kopcowanie charakteryzuje się stabilną złożonością $O(nlogn)$, niezależnie od początkowego układu danych, choć w praktyce mogą występować niewielkie różnice w czasie wykonania.

Tabela 2: Zmierzony czas dla sortowania przez kopcowanie

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	1,957	1,931	1,789	1,666	1,600
20	4,311	4,417	4,188	3,953	3,826
40	8,679	8,497	8,362	7,305	7,131
80	18,670	17,745	16,757	15,696	14,687
160	39,953	38,193	35,428	32,466	30,754
320	85,629	81,351	76,007	69,308	66,396
640	180,372	170,598	156,849	139,283	132,638

Z wykresu można zauważyć, że początkowy układ danych nie ma istotnego wpływu na czas sortowania, choć sortowanie tablic już uporządkowanych (w obu kierunkach) jest nieco szybsze. Wynika to z faktu, że w takich przypadkach operacje przywracania własności kopca są mniej kosztowne niż dla danych losowych.



Rysunek 5: Wykres czasu do ilości danych dla wszytskich wariantów algorytmu heap sort

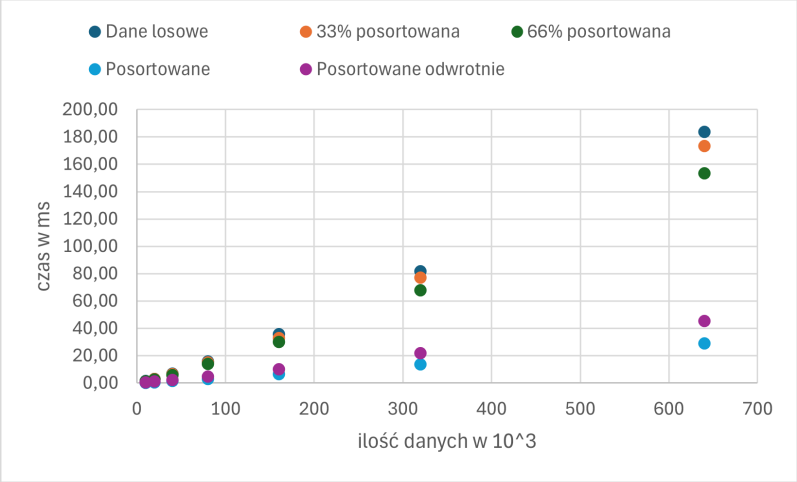
3.3 Shell Sort

Złożoność sortowania Shella zależy w dużym stopniu od wyboru algorytmu, który redukuje "gap" pomiędzy kolejnymi skokami

Tabela 3: Zmierzony czas dla sortowania Shella

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
algorytm $\frac{n}{2}$					
10	1,39	1,34	1,23	0,30	0,48
20	3,16	2,97	2,71	0,66	1,05
40	6,95	6,61	5,95	1,63	2,25
80	15,64	15,14	14,08	3,03	4,77
160	35,62	33,09	29,99	6,41	10,12
320	81,75	77,07	68,01	13,65	21,79
640	183,66	173,52	153,51	28,92	45,36
algorytm Sedgewicka					
10	1,23	1,10	1,03	0,21	0,37
20	2,51	2,42	2,32	0,47	0,79
40	5,73	5,35	5,11	1,01	1,66
80	12,93	12,35	11,24	2,15	3,49
160	28,40	27,40	25,19	4,64	7,54
320	63,26	60,69	56,05	9,81	15,41
640	140,58	135,49	125,14	21,06	32,24

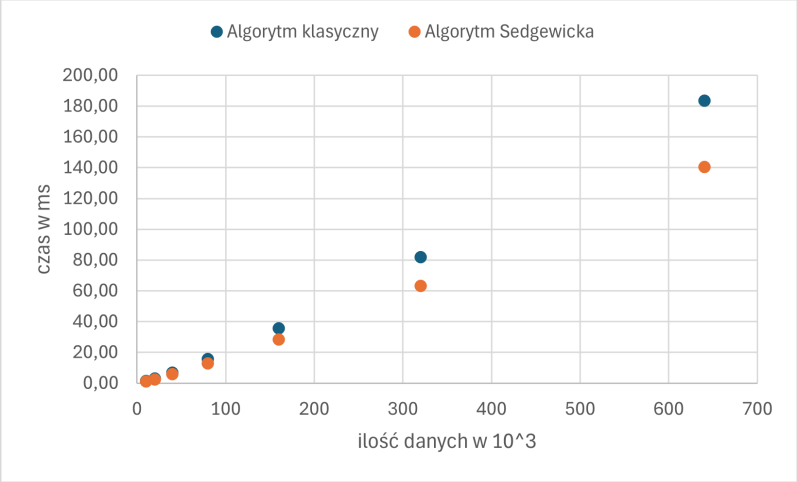
Dla zwykłego sortowania Shella czas sortowania jest zbliżony do sortowania przez wstawianie, czyli $O(n^2)$ jednak można zaobserwować jedną ciekawą zależność. Dla tablicy odwrotnie posortowanej wyniki są dużo krótsze. Wynika to prawdopodobnie z dużych przerw pomiędzy porównywanymi liczbami.



Rysunek 6: Wykres czasu do ilości danych dla zwykłego sortowania Shella

Dużą różnice jednak wprowadzają różne algorytmy obliczania przerwy. Dla algorytmu Sedgewicka średni przypadek jest taki sam jednak najgorszy możliwy to $O(n^{\frac{3}{4}})$

Można również porównać szybkość pomiędzy algorytmami



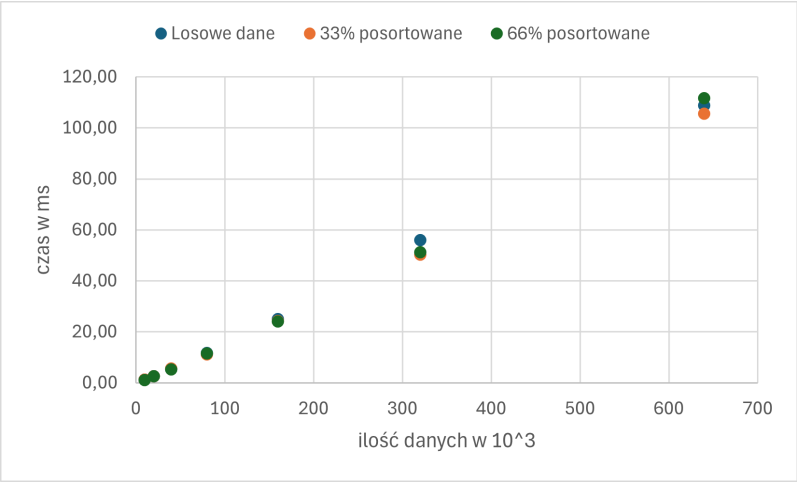
Rysunek 7: Wykres czasu do ilości danych, porównanie algorytmu liczącego przerwę

Jak widać im większa jest liczba danych tym większą różnicę widać pomiędzy algorytmami. Aktualnie najlepsze ciągi do przerwy są wyliczane empirycznie jednak nie zostały tutaj porównywane.

3.4 Quick Sort

Najbardziej popularny algorytm z powodu dużej szybkości i złożoności średniej $O(nlogn)$. Wynik jednak bardzo zależy od wyboru pivotu.

Ogólnie średni przypadek wynosi $O(nlogn)$ co widać na poniższym wykresie



Rysunek 8: Wykres pokazujący średnią złożoność obliczeniową

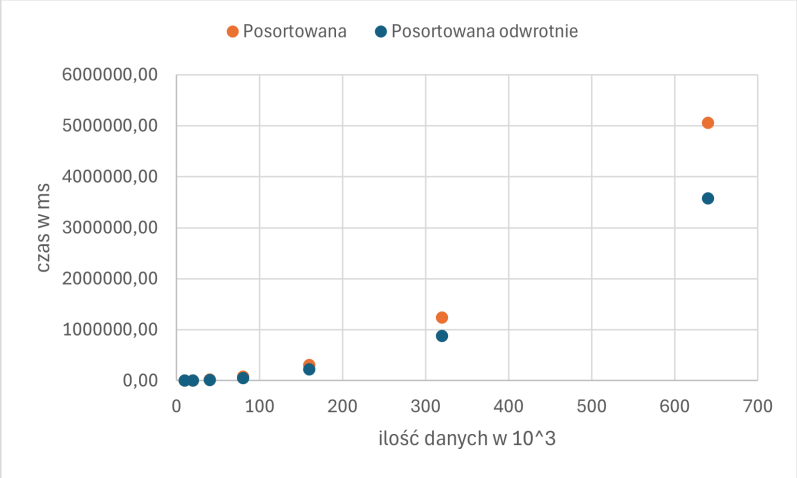
3.4.1 Pivot maksymalny prawy

Jak widać wybranie pivota jako maksymalnie prawego elementu daje bardzo słabe wyniki patrząc na szczególne przypadki tablicy posortowanej i odwrotnie posortowanej (Wyniki nie są dokładnie z powodu przekroczenia stacku rekursji i użycie wersji iteratywnej, która jest wolniejsza).

Tabela 4: Zmierzony czas dla sortowania Quick Sort z wyborem maksymalnie prawego pivota

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	1,17	1,22	1,11	1288,63	924,11
20	2,58	2,57	2,57	5219,34	3509,46
40	5,42	5,57	5,21	19570,20	13738,60
80	11,68	11,20	11,59	77165,70	54500
160	24,92	24,34	23,97	308789	219160
320	56,01	50,23	51,29	1237350	876792
640	108,75	105,52	111,68	5063050	3576580

W najgorszych przypadkach algorytm osiąga złożoność zbliżoną do $O(n^2)$, co widać na poniższym wykresie



Rysunek 9: Wykres porównujący najgorsze przypadki dla quick sort z prawym pivotem

3.4.2 Pivot maksymalnie lewy

Podobnie jak w przypadku maksymalnie prawego pivota ta wersja algorytmu bardzo źle radzi sobie z danymi posortowanymi, z powodu nierównych podziałów tablicy przy obieraniu pivota.

Tabela 5: Zmierzony czas dla sortowania Quick Sort z wyborem maksymalnie lewego pivota

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	1,19	1,31	1,51	309,36	954,43
20	2,55	2,75	3,13	1171,70	3537,64
40	5,44	5,96	7,26	4714,32	13847,20
80	11,44	13,69	13,69	18735,60	55617,70
160	24,45	26,96	29,95	75152,90	221110
320	53,11	59,78	64,26	300770	891394
640	108,67	123,88	136,18	1199090	3620870

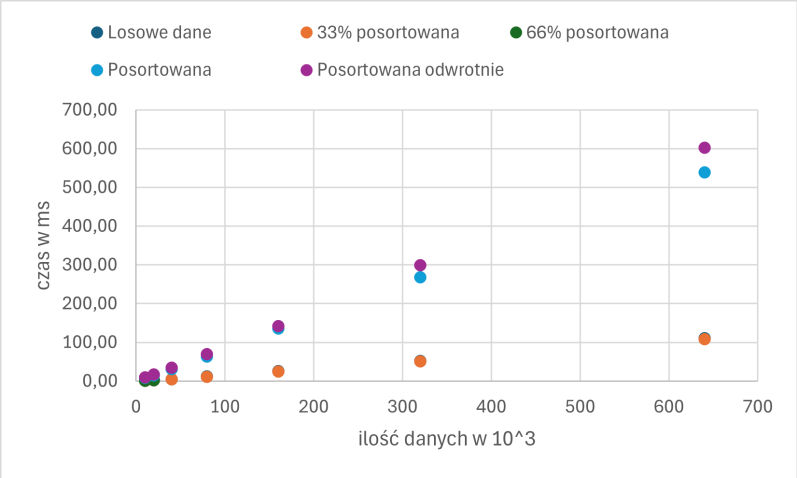
3.4.3 Pivot środkowy

Jedna z lepszych możliwości wyboru pivota, która zwiększa szanse ze tablica będzie podzielona w bardziej równomierny sposób.

Tabela 6: Zmierzony czas dla sortowania Quick Sort z wyborem środkowego pivota

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	1,20	1,13	1,18	8,19	9,22
20	2,63	2,53	2,56	15,32	17,24
40	5,42	5,28	5,26	30,97	34,95
80	12,11	11,26	11,22	64,04	69,80
160	25,87	24,65	26,06	135,75	142,50
320	52,13	50,52	51,63	267,70	299,51
640	110,60	108,38	105,51	538,97	602,07

Przy wyborze pivota środkowego najgorszy przypadek nadal zbliżony jest do $O(n^2)$, jednak wartości nie są tak zawrotne jak w przypadku wyboru prawego/lewego pivota



Rysunek 10: Wykres pokazujący szybkość quick sort z pivotem środkowym

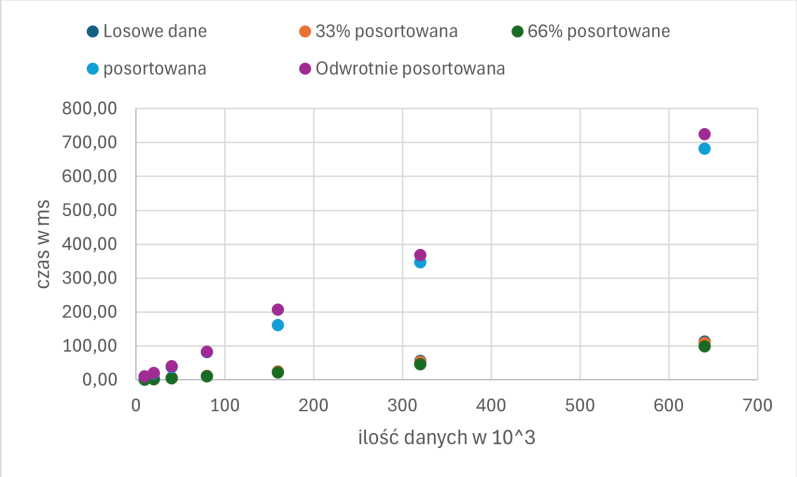
3.4.4 Pivot losowy

Jedną z lepszych metod wyboru pivota jest losowy wybór, ponieważ zwiększa szansę na bardziej równomierny podział tablicy. Dzięki temu algorytm unika najgorszego przypadku złożoności O(n^2), który występuje, gdy podziały są bardzo nierówne.

Tabela 7: Zmierzony czas dla sortowania Quick Sort z wyborem losowego pivota

Ilość danych w 10^3	czas w ms				
	dane losowe	33% posortowana	66% posortowana	posortowana	odwrotnie posortowana
10	1,26	1,33	1,17	10,01	10,80
20	2,71	2,63	2,43	19,19	21,00
40	5,76	5,42	5,07	38,23	40,17
80	12,20	11,92	10,42	81,78	83,08
160	25,28	25,16	21,74	161,16	207,45
320	55,90	53,59	46,07	347,66	368,33
640	113,48	108,34	99,52	682,59	724,95

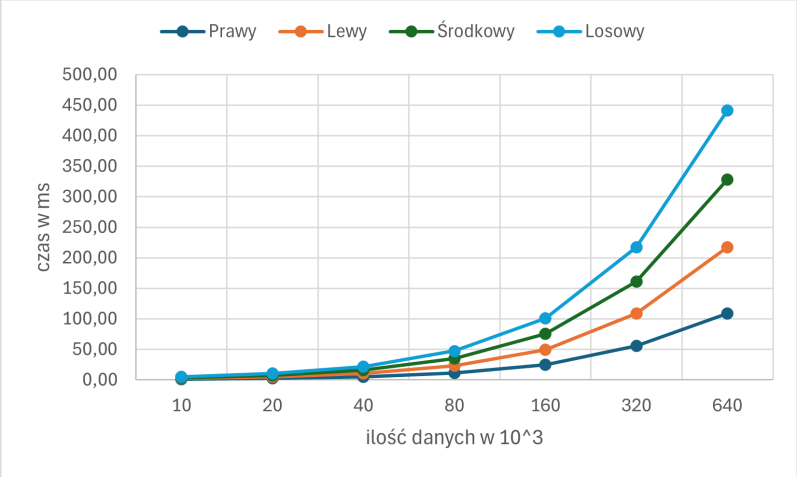
W tym przypadku wyniki są bardzo zbliżone do pivota środkowego



Rysunek 11: Wykres pokazujący szybkość quick sort z pivotem losowym

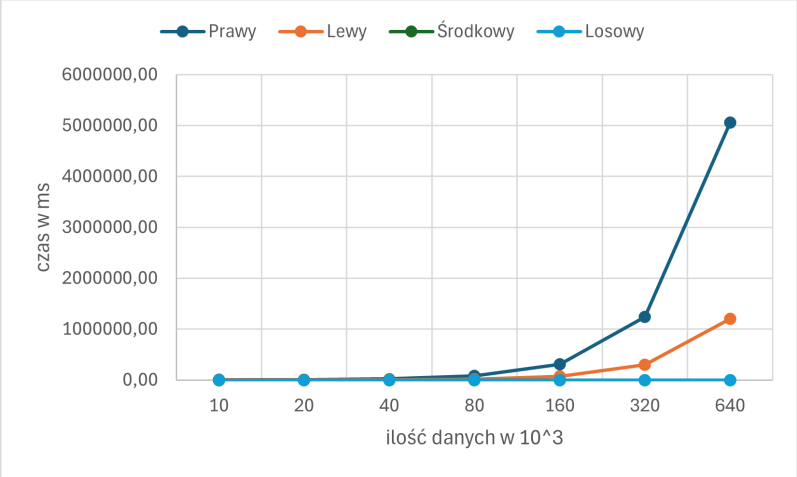
3.4.5 Porównanie algorytmów z różnymi pivotami

Jak widać na poniższym wykresie najwolniejszy jest wybór losowego pivota jednak przy takiej ilości danych różnice nie są zauważalne w wielkim stopniu dla danych losowych.



Rysunek 12: Wykres skumulowany porównujący wybór pivota do prędkości dla danych losowych

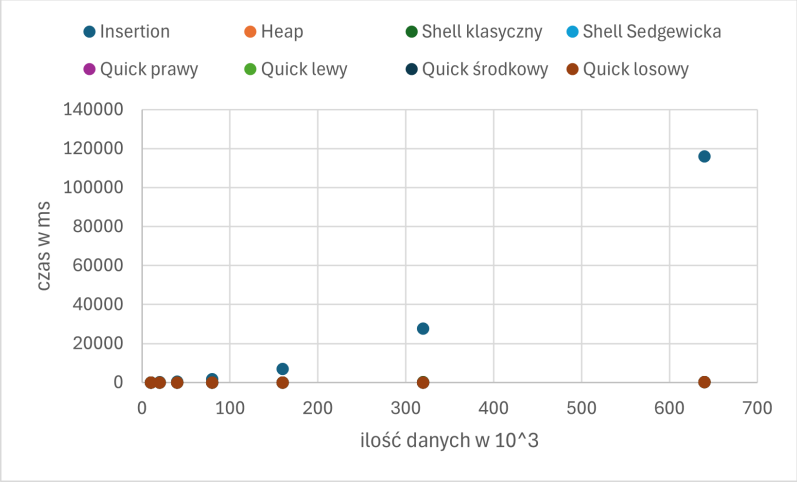
Dużą różnice możemy zaobserwować, gdy przejdziemy do tablic posortowanych gdzie dobrze widać zmianę złożoności na $O(n^2)$ dla pivota prawego i lewego. Jak widać na poniższym wykresie czasy dla środkowego i losowego są nieporównywalnie krótsze od prawego i lewego.



Rysunek 13: Wykres pokazujący czas obliczeń dla różnych wyborów pivota i tablicy posrtowanej

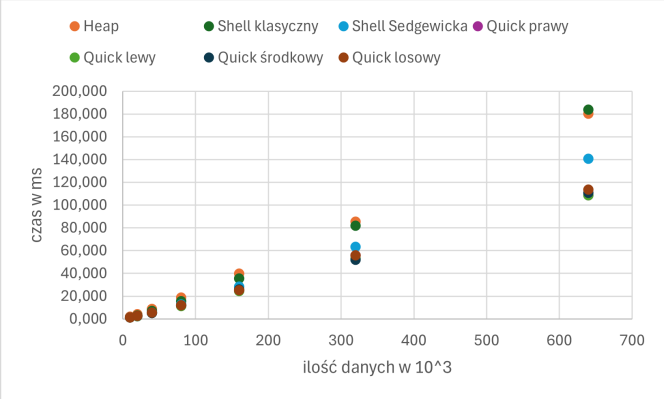
3.5 Porównanie algorytmów

Przy porównaniu sortowania przez wstawianie z innymi szybszymi algorytmami sortowania dobrze widać różnicę pomiędzy złożonością $O(n^2)$, a $O(nlogn)$



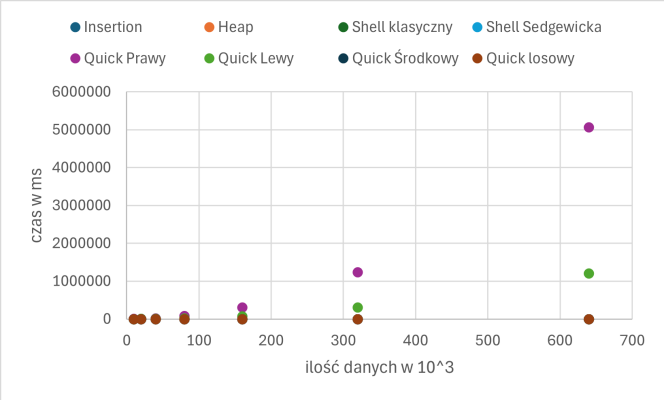
Rysunek 14: Wykres pokazujący czas obliczeń dla różnych algorytmów i losowych danych

Przyglądając się bliżej szybszym algorytmom możemy zauważyć, że w prędkości przoduje quick sort szczególnie dla dużych danych



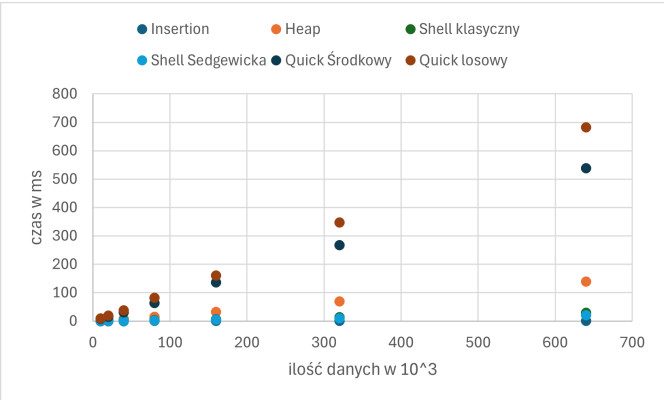
Rysunek 15: Wykres pokazujący czas obliczeń dla różnych algorytmów i losowych danych

Sytuacja wygląda inaczej, gdy patrzymy na tablice wcześniej posortowane.



Rysunek 16: Wykres pokazujący czas obliczeń dla różnych algorytmów i posortowanej tablicy

Przyglądając się bliżej algorytmom mniejszej złożoności widać przewagę sortowania przez wstawianie, który osiąga złożoność $O(n)$



Rysunek 17: Wykres pokazujący czas obliczeń dla różnych algorytmów i posortowanej tablicy

4 Badanie wpływu typu danych na czas sortowania

Tabela 8: Tabela porównująca czas sortowanie z typami danych dla tabeli 160 000 elementową

Algorytmy	Losowa tablica		33% posortowana		66% posortowana		Posortowana		Odwrotnie sortowana	
	int	float	int	float	int	float	int	float	int	float
insertion	6855,01	7316,21	6110,49	6445,83	3817,27	4022,93	0,26	0,31	13611,30	14395,50
heap	39,95	34,65	38,19	33,27	35,43	30,06	32,47	26,23	30,75	25,47
shell klasyczny	35,62	42,66	33,09	40,06	29,99	36,28	6,41	6,58	10,12	9,94
shell Sedgewick	28,40	33,12	27,40	32,11	25,19	29,51	4,64	4,67	7,54	7,12
quick prawy	24,92	26,15	24,34	26,99	23,97	26,61	308789	485627	219160	281061
quick lewy	24,45	26,72	26,96	28,37	29,95	34,25	75153	75311	221110	281377
quick środkowy	25,87	26,87	24,65	25,55	26,06	24,61	135,75	112,28	142,50	132,95
quick losowy	25,28	28,23	25,16	26,74	21,74	23,97	161,16	236,30	207,45	360,84

Po przeanalizowaniu tabeli możemy zauważyć, że w większości przypadków obliczenia na liczbach stałoprzecinkowych są szybsze od tych na zmiennoprzecinkowych mimo, że zajmują one tyle samo miejsca w pamięci. Wynika to prawdopodobnie ze zwiększonej złożoności obliczeń na liczbach typu float. Ciekawym przypadkiem jest Heap sort, gdzie czasy wykonywania dla float jest krótszy niż int, może wynikać to ze specyfiki algorytmu, gdzie każda iteracja wymaga odbudowy kopca.

5 Podsumowanie

Podsumowując, wybór algorytmu sortowania zależy od specyfiki problemu i charakterystyki danych. Jeśli najważniejszy jest czas wykonania, a początkowe ułożenie danych jest nieznane lub dane są losowe, najlepszym wyborem będzie Quick Sort z odpowiednim wyborem pivotu. Działa on średnio w czasie $O(nlogn)$, ale w najgorszym przypadku (z nieoptymalnym pivotem) może osiągnąć złożoność $O(n^2)$. W sytuacjach, gdzie dane są częściowo posortowane lub zawierają tylko kilka elementów do posortowania, lepszym rozwiązaniem może okazać się Insertion Sort. Ten algorytm ma złożoność $O(n^2)$ w najgorszym przypadku, ale dla prawie posortowanych danych jego czas działania jest liniowy, $O(n)$, co czyni go bardzo efektywnym w takich przypadkach. Insertion Sort jest również idealnym wyborem w przypadku małych zestawów danych, ponieważ jego implementacja jest prosta i efektywna. Jeśli chodzi o typ danych, Heap Sort może okazać się skuteczny w przypadku liczb zmiennoprzecinkowych (float), ponieważ działa na strukturze drzewa, a operacje na liczbach zmiennoprzecinkowych (w tym porównania) mogą być zoptymalizowane przez architekturę procesora. Heap Sort ma gwarantowaną złożoność $O(nlogn)$, co sprawia, że jest bardziej stabilny niż Quick Sort.

6 Literatura

1. `https://www.baeldung.com/java-shell-sort`
2. `https://www.programiz.com/dsa/heap-data-structure`
3. `https://www.home.umk.pl/~abak/wdimat/s/HeapSort.html`
4. `https://eduinf.waw.pl/inf/alg/003_sort/0012.php`
5. Grzegorz Jagiella, Wykład 5 (cz. 2) - algorytmy sortujące II, Uniwersytet Wrocławski, 2020, online: `https://math.uni.wroc.pl/~jagiella/p2python/skrypt_html/wyklad5-2.html`, dostęp: 13.04.2024