Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową

Algorytmy i złożoność obliczeniowa
Sprawozdanie z projektu 1

Autor: Dominik Kaczmarek 281007
Prowadzący: dr inż. Zbigniew Buchalski
Data oddania: 16.05.2025

Spis treści

Wstęp

Celem zadania projektowego była implementacja określonych algorytmów sortowania oraz analiza ich efektywności w zależności od liczby elementów w tablicy, typu elementów oraz wstępnego ułożenia danych.

W ramach projektu zaimplementowane zostały następujące algorytmy sortowania:

- sortowanie przez wstawianie (*insertion sort*),
- sortowanie przez kopcowanie (*heap sort*),
- sortowanie Shella (*Shell sort*),
- sortowanie szybkie (*quick sort*).

Do analizy złożoności obliczeniowej poszczególnych algorytmów wykorzystano notację dużego O, która pozwala określić górną granicę czasochłonności lub zapotrzebowania na pamięć danej metody. Jest ona powszechnie stosowana w analizie algorytmów i pomaga w ich porównywaniu i ocenie efektywności w zależności od wielkości danych wejściowych.

Sortowanie przez wstawianie

Sortowanie przez wstawianie to prosty algorytm działający na zasadzie porządkowania elementów tablicy jeden po drugim. Algorytm ten wybiera kolejny element z nieposortowanej części tablicy i porównuje go z elementami już posortowanymi, przesuwając większe elementy w prawo, aby zrobić miejsce dla wstawianego elementu. Proces ten kontynuuje się aż do momentu, gdy wszystkie elementy zostaną posortowane.

Algorytm ten jest efektywny w przypadku niewielkich zbiorów danych i tablic częściowo posortowanych, ale może być o wiele mniej wydajny dla dużych, nieuporządkowanych zbiorów danych.

Sortowanie przez kopcowanie

Sortowanie przez kopcowanie opiera się na strukturze danych zwanej kopcem (ang. *heap*). Algorytm tworzy kopiec maksymalny, w którego korzeń, czyli pierwszy element tablicy jest największy. Następnie korzeń jest zamieniany z ostatnim liściem (ostatni nieposortowany element), jego rozmiar jest pomniejszany i przywracana jest struktura kopca. Proces ten powtarza się aż do momentu, gdy rozmiar kopca zostaje zredukowany do jednego elementu, co oznacza zakończenie sortowania.

Sortowanie Shella

Sortowanie Shella to ulepszona wersja sortowania przez wstawianie, która umożliwia porównywanie i zamienianie elementów oddalonych od siebie o określony krok. W kolejnych iteracjach krok ten jest zmniejszany, aż do osiągnięcia wartości 1, kiedy to algorytm działa jak zwykłe sortowanie przez wstawianie.

Wydajność sortowania Shella zależy od zastosowanej sekwencji kroków (ang. *gap sequence*). W oryginalnej wersji zaproponowanej przez Donalda Shella w 1959 roku, kroki ustalane są według wzoru:

$$h_k = \left\lfloor \frac{n}{2^k} \right\rfloor$$

Do bardziej efektywnych sekwencji należy sekwencja Tokudy, dla której kroki wyznaczane są ze wzoru:

$$h_k = \left\lceil \frac{1}{5}\left(9 * \left(\frac{9}{4}\right)^{\mathrm{k}-1} - 4\right) \right\rceil$$

Sekwencja Tokudy zapewnia lepszą równowagę między liczbą porównań i przestawień, przez co osiąga lepsze czasy działania w praktyce.

Sortowanie szybkie

Sortowanie szybkie jest algorytmem wykorzystującym technikę "dziel i zwyciężaj". Polega na wybraniu elementu zwanego *pivotem* i podzieleniu tablicy na dwie części: elementy mniejsze od pivota oraz elementy większe lub równe. Następnie procedura jest rekurencyjnie stosowana do obu podtablic, do momentu kiedy tablice nie osiągną rozmiaru jednego elementu.

Na działanie tego algorytmu wpływa sposób rozłożenia danych, oraz sposób doboru pivota. Zaimplementowane zostały metody wyboru elementu skrajnego (lewego lub prawego), elementu środkowego lub elementu losowego.

Porównanie złożoności obliczeniowych algorytmów

Nazwa algorytmu	Średnia złożoność obliczeniowa	Pesymistyczna złożoność obliczeniowa
Przez wstawianie	$O(n^2)$	$O(n^2)$
Przez kopcowanie	$O(n \log n)$	$O(n \log n)$
Shella	*zależy od sekwencji kroków*	$O(n^2)$
Szybkie	$O(n \log n)$	$O(n^2)$

W przypadku sekwencji Tokudy w sortowaniu Shella złożoność obliczeniowa nie jest znana. Udowodniono jednak, że sekwencja ta jest bardziej wydajna niż wiele innych sekwencji.

Plan eksperymentu

Wymagane algorytmy zostały zaimplementowane w języku C++ z użyciem szablonów (*templates*), aby można było je wykorzystać do sortowania danych różnych typów. Wszystkie algorytmy sortują elementy w porządku rosnącym.

W ramach eksperymentu zbadano wpływ trzech czynników na czas działania algorytmów:

- typu danych (liczby całkowite `int` i zmiennoprzecinkowe `float`),
- rozmiaru sortowanej tablicy,
- początkowego ułożenia elementów.

Dla każdego badanego przypadku przeprowadzono 100 niezależnych pomiarów, a ich wyniki zostały uśrednione. W każdej iteracji generowano nowy zestaw danych wejściowych.

Tablice generowane były losowo z wykorzystaniem silnika `mt19937`, rozkładów `uniform_int_distribution` i `uniform_real_distribution` oraz obiektu `random_device` z biblioteki `<random>`. Do uzyskania określonych początkowych ułożeń elementów użyto też funkcji `std::ranges::sort`, `std::ranges::reverse` oraz `std::shuffle`.

Czas wykonania sortowania mierzono za pomocą zegara `steady_clock` z biblioteki `<chrono>`.

Rozmiary testowanych tablic wynosiły zazwyczaj 10 000, 20 000, … do 100 000 elementów. W niektórych przypadkach zakres ten został ograniczony ze względu na zbyt długi czas działania lub przekroczenie dozwolonej głębokości rekurencji.

Uwzględnione warianty początkowego ułożenia elementów to:

- tablica losowa,
- tablica posortowana,
- tablica posortowana odwrotnie,
- tablica posortowana w 33%,
- tablica posortowana w 66%.

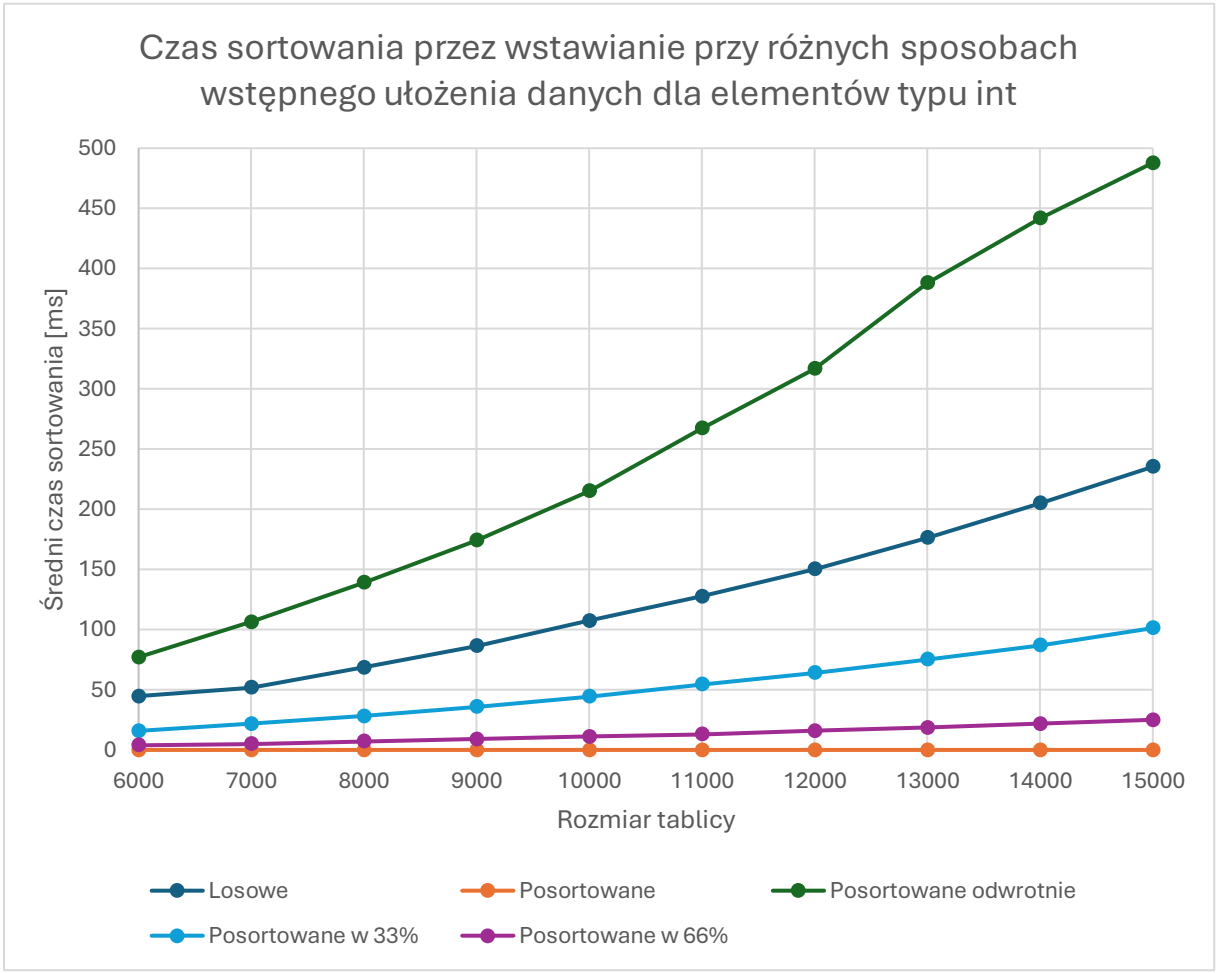
Przebieg eksperymentu

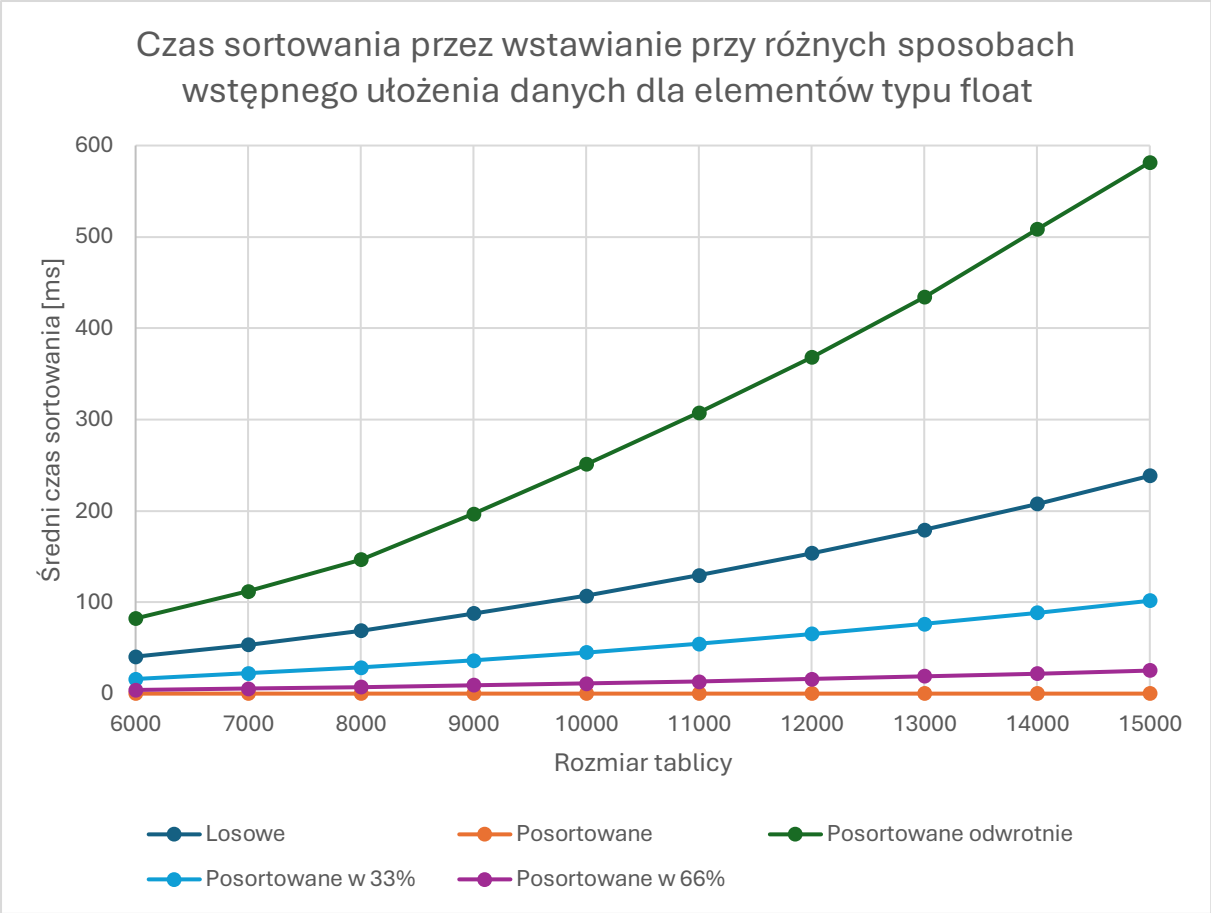
Sortowanie przez wstawianie

Ze względu na długi czas sortowania dla dużych rozmiarów tablic, mierzono czas dla rozmiarów od 6000 do 15000. Pod tabelą zamieszczono wykresy porównujące szybkość algorytmu przy różnych sposobach ułożenia danych dla dwóch typów elementów.

Sortowanie przez wstawianie			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	6000	44,70	40,52
	7000	51,77	53,39
	8000	68,42	68,86
	9000	86,37	87,85
	10000	107,29	106,99
	11000	127,55	129,52
	12000	150,17	153,56
	13000	176,24	179,17
	14000	204,93	207,68
	15000	235,26	238,4
Posortowane	6000	0	0
	7000	0	0
	8000	0	0
	9000	0	0
	10000	0	0
	11000	0	0
	12000	0	0
	13000	0	0
	14000	0	0
	15000	0	0
Posortowane odwrotnie	6000	77,07	82,27
	7000	106,24	112,03
	8000	138,93	146,74
	9000	174,16	196,82
	10000	215,15	250,98
	11000	267,20	307,65
	12000	316,64	368,08
	13000	388,01	434,13
	14000	441,88	508,43
	15000	487,86	581,61

Posortowane w 33%	6000	15,80	16,02
	7000	21,62	22,12
	8000	28,17	28,57
	9000	35,88	36,11
	10000	44,28	45,17
	11000	54,45	54,61
	12000	64,03	65,27
	13000	75,13	76,38
	14000	86,87	88,35
	15000	101,21	101,64
Posortowane w 66%	6000	3,81	3,98
	7000	5,00	5,44
	8000	7,02	7,25
	9000	8,83	9,15
	10000	10,99	11,15
	11000	13,02	13,23
	12000	15,86	16,05
	13000	18,51	19,03
	14000	21,69	21,96
	15000	25,03	25,23





Wnioski

Średnie czasy sortowania w przypadku sortowania przez wstawianie są stosunkowo długie. Wstępne ułożenie danych ma znaczący wpływ na czas sortowania – w przypadku, gdy dane są posortowane, algorytm nie zamienia miejscami żadnych elementów. Jednak wraz z poziomem nieuporządkowania danych rośnie również czas sortowania – szczególnie niekorzystny przypadek występuje, gdy elementy są posortowane odwrotnie. W takich sytuacjach czas sortowania jest wyraźnie dłuższy niż w innych przypadkach, gdzie przynamniej część elementów jest poprawnie posortowana lub kompletnie losowa.

W przypadku tablicy liczb całkowitych czasy sortowania są zwykle krótsze niż przy tablicy liczb zmiennoprzecinkowych.

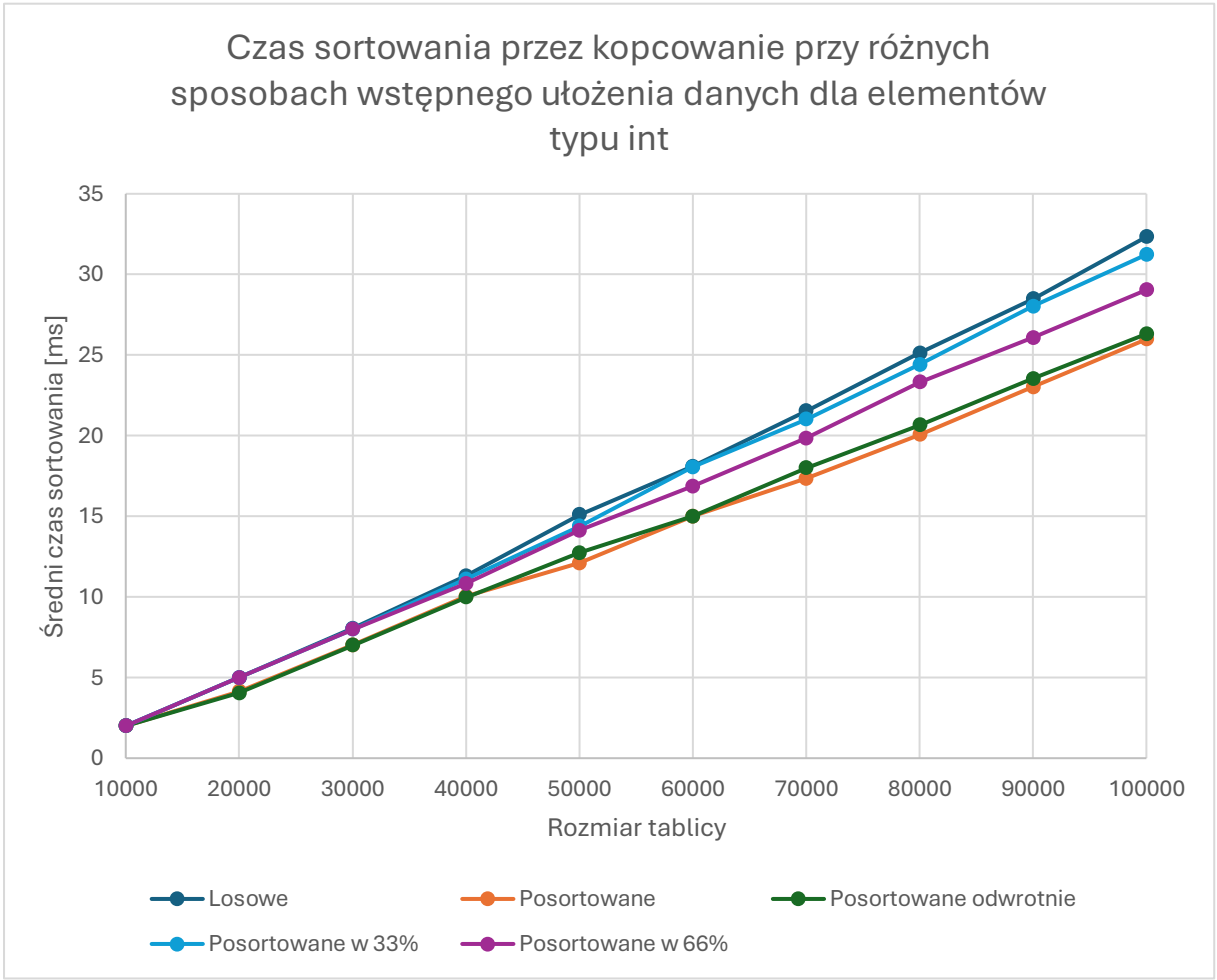
Sortowanie przez wstawianie nie jest odpowiednie do sortowania tablic dużych rozmiarów, ponieważ czas jego wykonywania rośnie bardzo szybko wraz z liczebnością danych. Najlepiej sprawdzi się dla bardzo małych zbiorów danych, gdzie prostota implementacji jest ważniejsza niż wydajność.

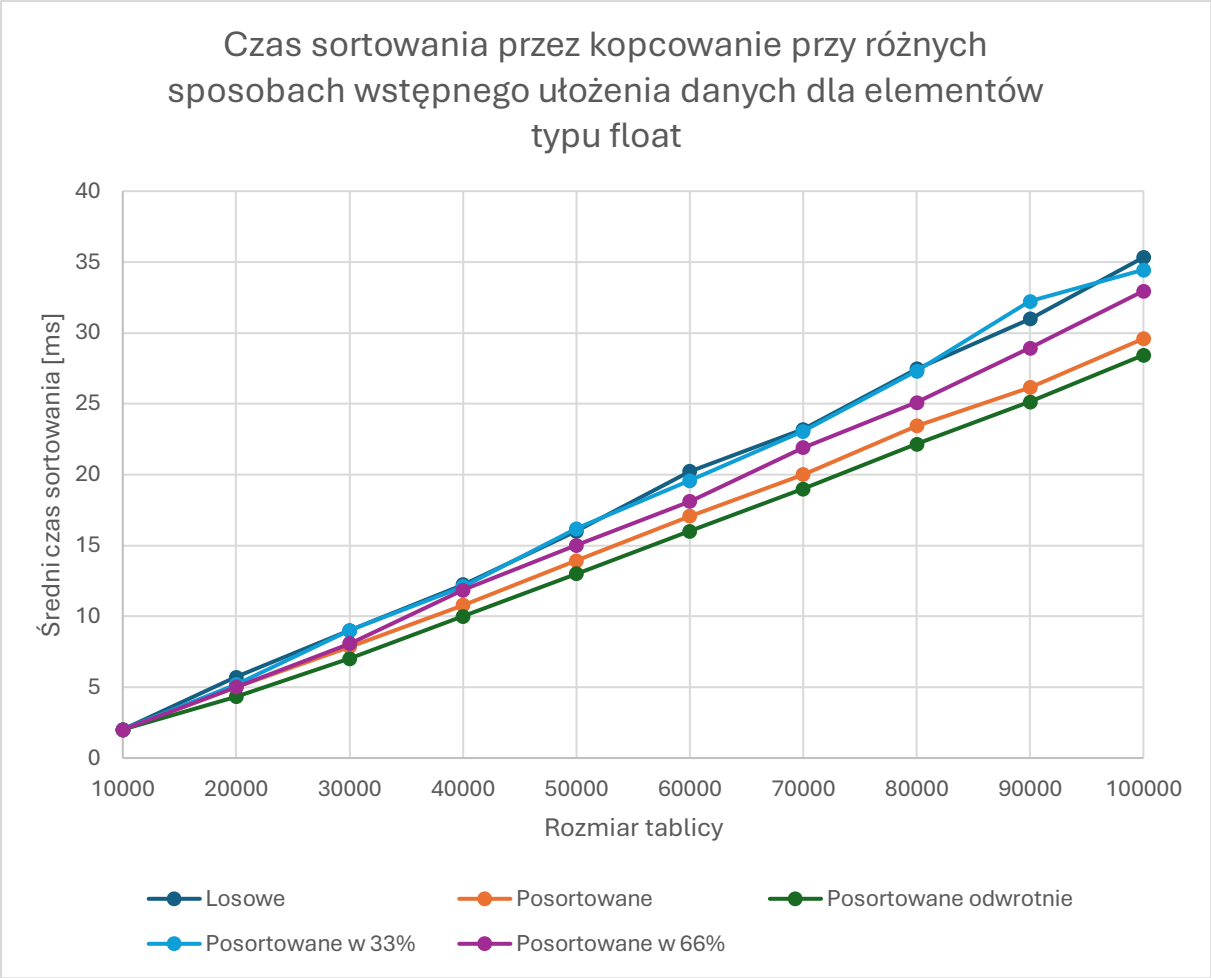
Sortowanie przez kopcowanie

Sortowanie przez kopcowanie jest szybkie i nie wymagało dostosowania rozmiaru tablicy w żadnym przypadku. Czas mierzony był dla rozmiarów od 10000 do 100000. Wyniki pomiarów przedstawione zostały w tabeli, a pod nią zamieszczone zostały wykresy porównujące szybkość algorytmu przy różnych wstępnych sposobach ułożenia danych dla dwóch typów elementów.

Sortowanie przez kopcowanie			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	2,00	2,01
	20000	5,00	5,71
	30000	8,05	9,00
	40000	11,30	12,23
	50000	15,09	16,01
	60000	18,10	20,24
	70000	21,52	23,20
	80000	25,12	27,47
	90000	28,47	30,98
	100000	32,33	35,32
Posortowane	10000	2,00	2,00
	20000	4,14	5,00
	30000	7,01	7,86
	40000	10,04	10,78
	50000	12,09	13,92
	60000	15,00	17,06
	70000	17,35	20,01
	80000	20,06	23,45
	90000	23,01	26,14
	100000	25,98	29,59
Posortowane odwrotnie	10000	2,00	2,01
	20000	4,03	4,32
	30000	7,00	7,01
	40000	9,98	10,00
	50000	12,73	13,00
	60000	15,00	16,01
	70000	18,00	19,00
	80000	20,65	22,16
	90000	23,53	25,13
	100000	26,31	28,43

Posortowane w 33%	10000	2	2,01
	20000	5	5,18
	30000	8	9
	40000	11,1	12,1
	50000	14,39	16,18
	60000	18,05	19,58
	70000	21,01	23,05
	80000	24,41	27,31
	90000	28,02	32,23
	100000	31,23	34,45
Posortowane w 66%	10000	2	2
	20000	5	5
	30000	8	8,07
	40000	10,83	11,85
	50000	14,13	15,01
	60000	16,86	18,12
	70000	19,84	21,91
	80000	23,31	25,09
	90000	26,08	28,93
	100000	29,04	32,95





Wnioski

Średni czas sortowania przy użyciu algorytmu sortowania przez kopcowanie jest bardzo zbliżony we wszystkich przypadkach wstępnego ułożenia danych – zarówno dla danych posortowanych, odwrotnie, jak i losowych, niezależnie od tego, czy są to liczby całkowite, czy zmiennoprzecinkowe. Wynika to z faktu, że algorytm ten zawsze wykonuje podobną liczbę operacji, niezależnie od struktury danych wejściowych

Sortowanie przez kopcowanie jest dobrym wyborem, gdy potrzeba niezawodnej metody sortowania odpornej na przypadki skrajne, z którymi inne algorytmy (np. sortowanie szybkie) mogą mieć problemy. Jego złożoność czasowa O(n log n) jest gwarantowana we wszystkich przypadkach. Dodatkową zaletą sortowania przez kopcowanie jest to, że działa ono w miejscu, co oznacza, że nie wymaga dodatkowej pamięci na przechowywanie danych pomocniczych.

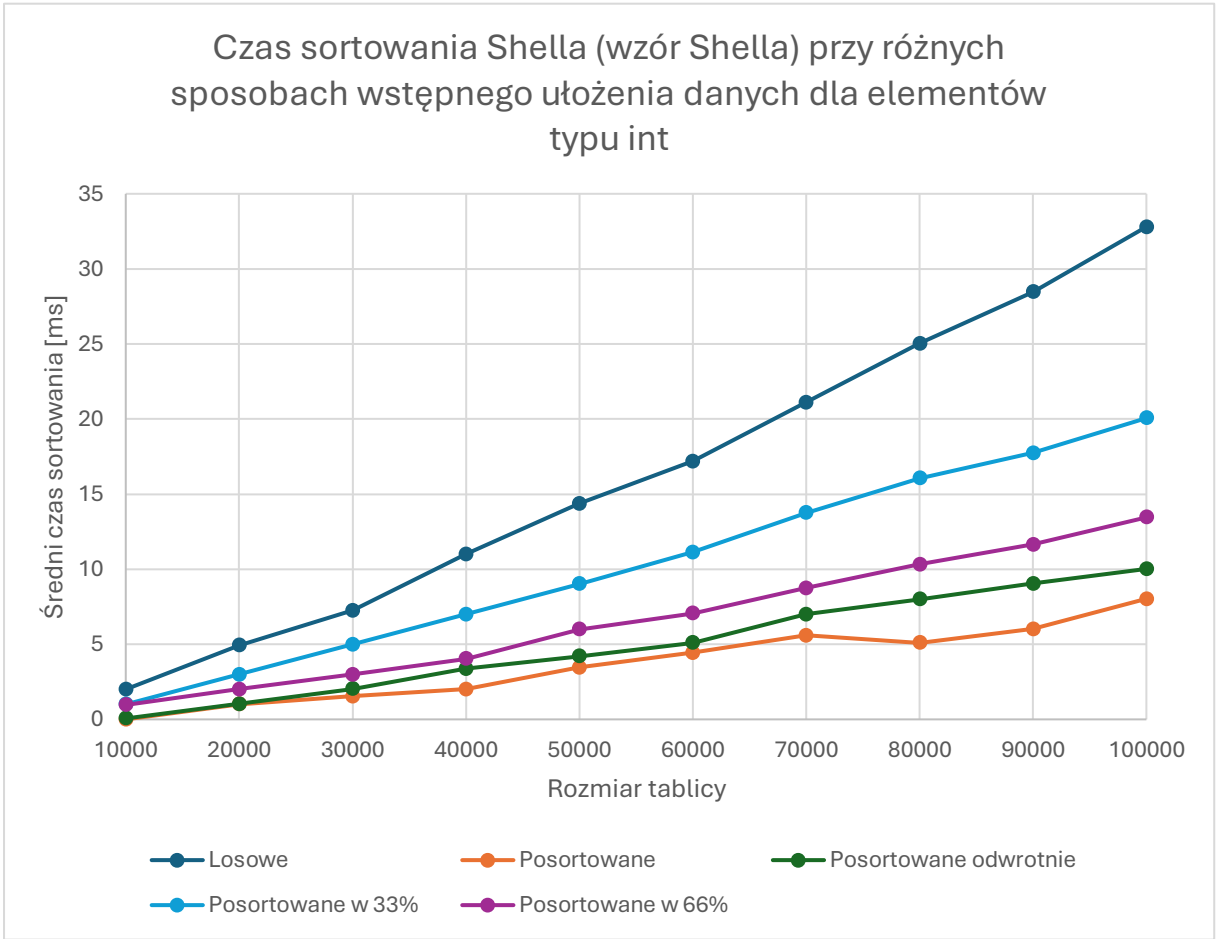
Sortowanie Shella

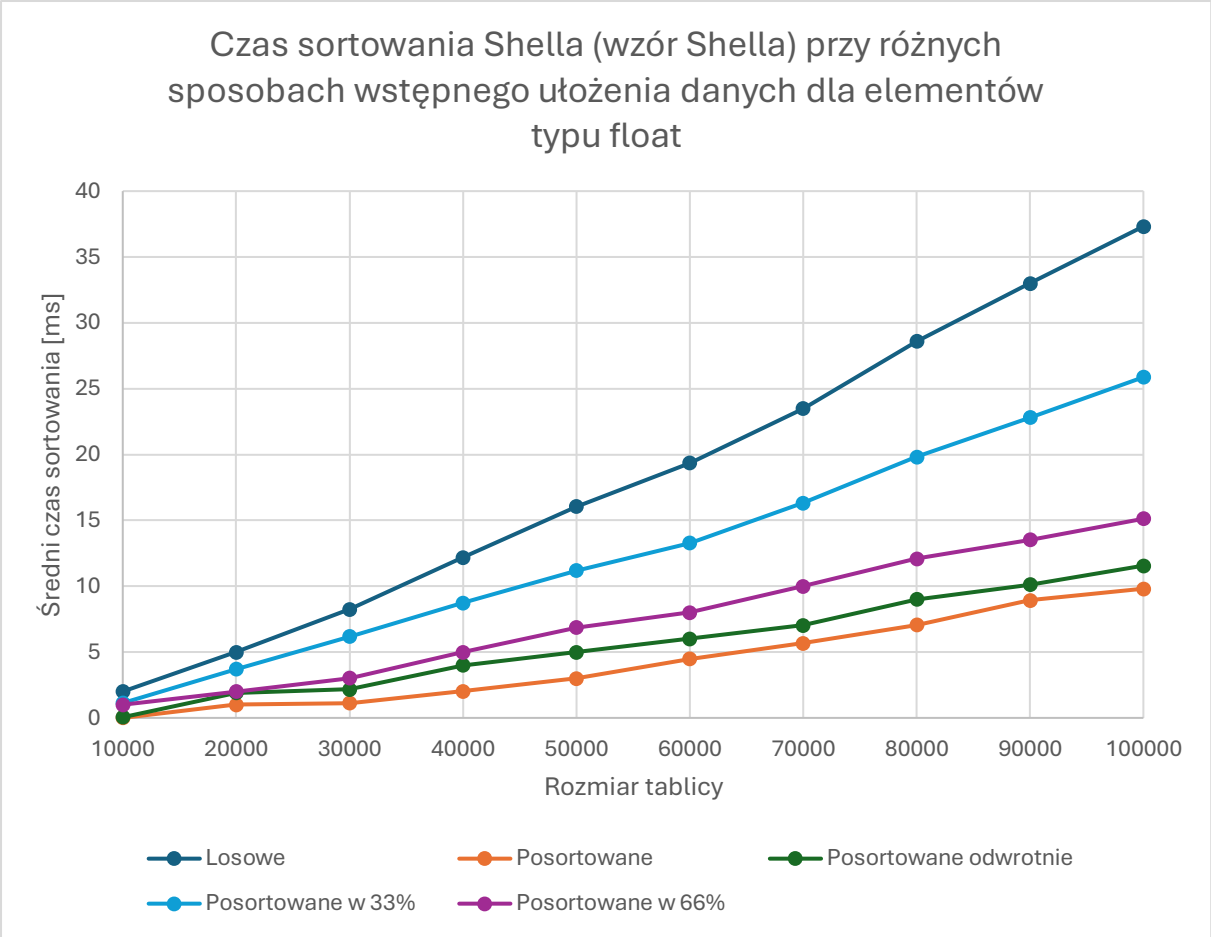
a) Algorytm Shella

Sortowanie Shella wykorzystujące wzór Shella również jest bardzo szybkim algorytmem sortowania. Nie wymagało dostosowania rozmiaru tablicy w żadnym przypadku – pomiary przeprowadzone zostały dla rozmiarów od 10000 do 100000.

Sortowanie Shella (wzór Shella)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	2,00	2,00
	20000	4,94	5,00
	30000	7,25	8,26
	40000	11,00	12,19
	50000	14,38	16,04
	60000	17,18	19,35
	70000	21,11	23,50
	80000	25,03	28,59
	90000	28,46	32,99
	100000	32,80	37,32
Posortowane	10000	0,00	0,00
	20000	1,00	1,00
	30000	1,53	1,13
	40000	2,01	2,02
	50000	3,46	3,00
	60000	4,44	4,48
	70000	5,58	5,66
	80000	5,10	7,05
	90000	6,01	8,94
	100000	8,03	9,81
Posortowane odwrotnie	10000	0,06	0,05
	20000	1,02	1,91
	30000	2,03	2,16
	40000	3,38	4,00
	50000	4,20	5,00
	60000	5,10	6,00
	70000	7,00	7,03
	80000	8,00	9,01
	90000	9,04	10,12
	100000	10,02	11,56

Posortowane w 33%	10000	1	1,13
	20000	3	3,71
	30000	5	6,17
	40000	7	8,74
	50000	9,02	11,2
	60000	11,13	13,27
	70000	13,75	16,33
	80000	16,06	19,82
	90000	17,74	22,8
	100000	20,06	25,87
Posortowane w 66%	10000	0,96	1
	20000	2	2
	30000	3	3,03
	40000	4,03	5
	50000	5,99	6,86
	60000	7,07	8
	70000	8,74	10
	80000	10,33	12,09
	90000	11,65	13,53
	100000	13,45	15,12



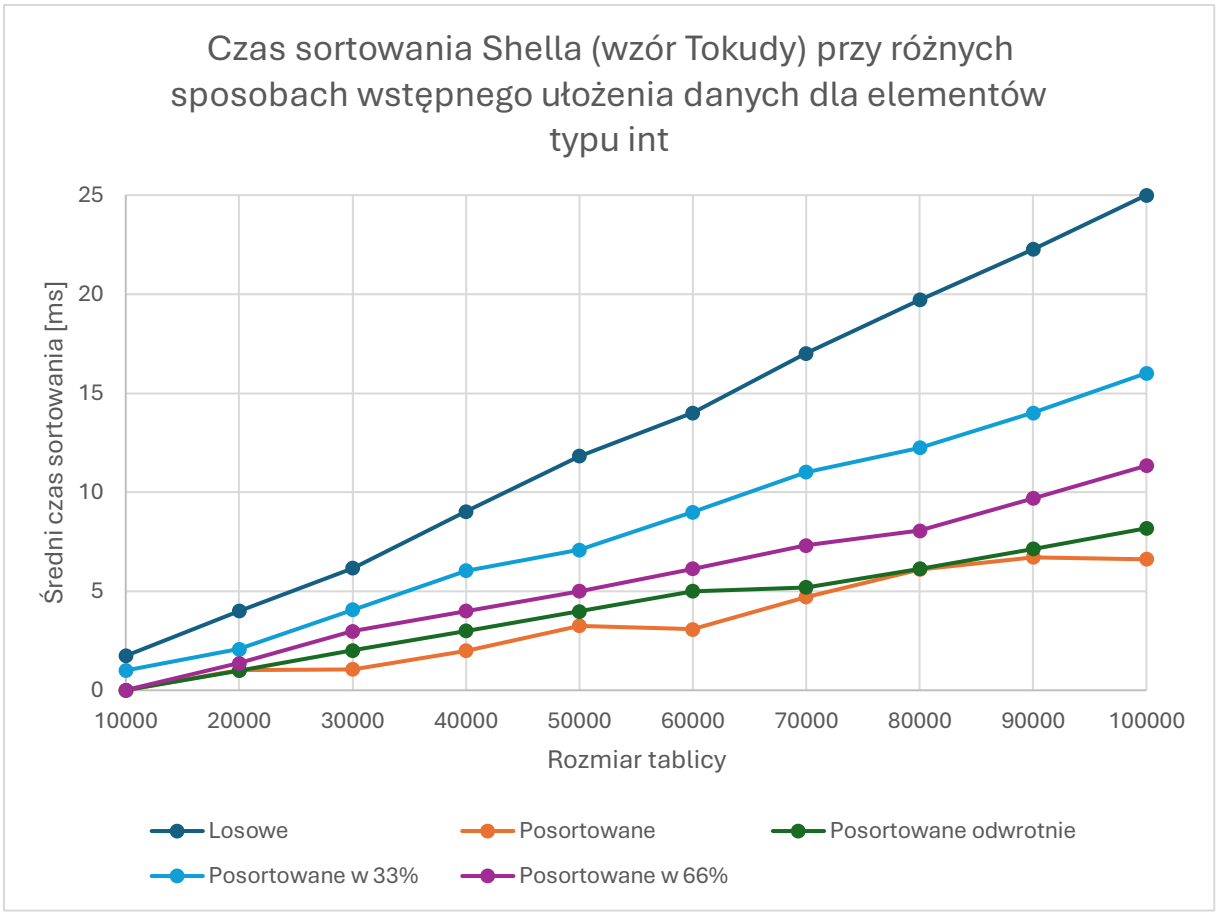


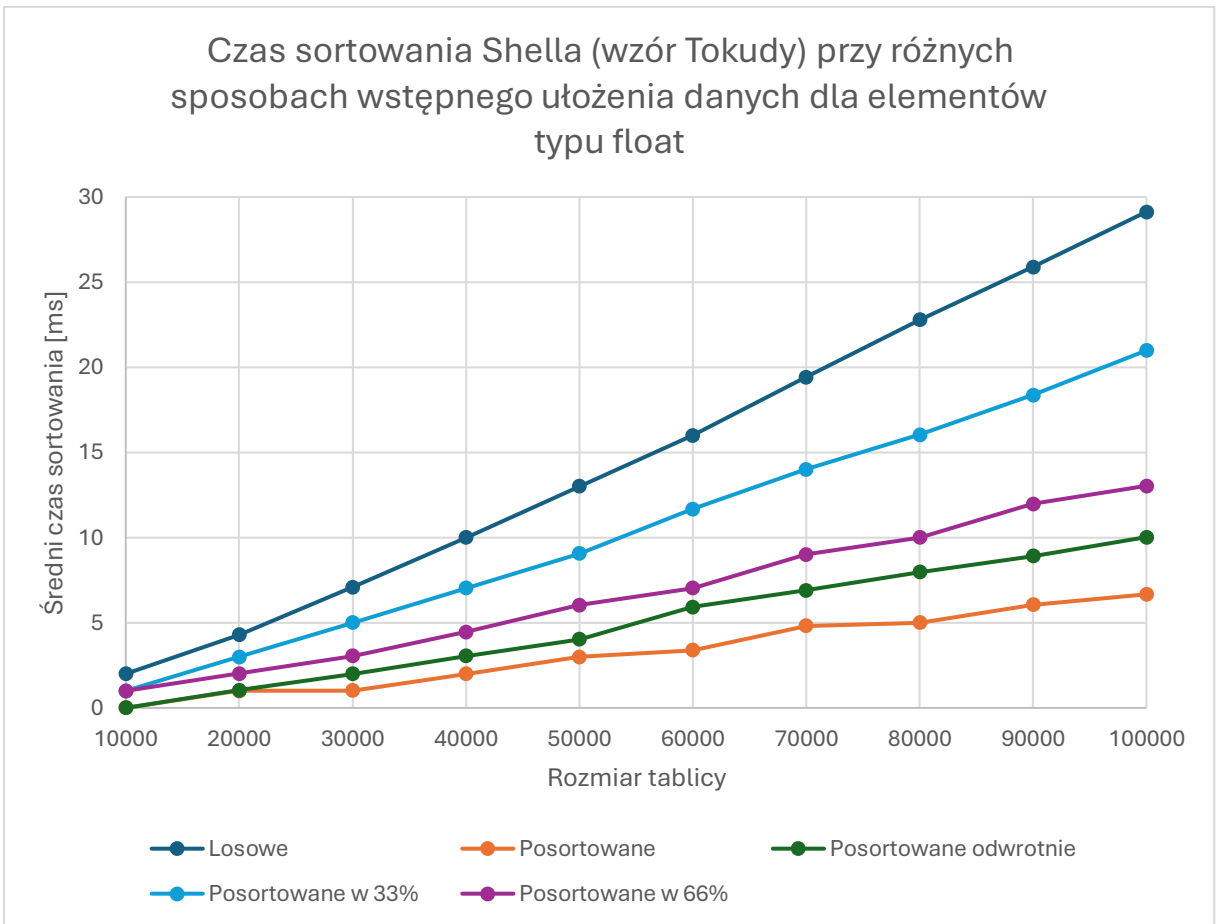
b) Algorytm Tokudy

Tak jak w przypadku algorytmu Shella wykorzystującego oryginalny wzór Shella, algorytm wykorzystujący wzór Tokudy był bardzo szybki oraz nie wymagał dostosowania liczby elementów w żadnym przypadku.

Sortowanie Shella (wzór Tokudy)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	1,74	2,00
	20000	4,00	4,29
	30000	6,17	7,10
	40000	9,02	10,00
	50000	11,83	13,01
	60000	14,01	16,00
	70000	17,02	19,42
	80000	19,71	22,78
	90000	22,26	25,88
	100000	25,00	29,13

Posortowane	10000	0,00	0,00
	20000	1,01	1,00
	30000	1,06	1,01
	40000	2,00	2,00
	50000	3,25	3,00
	60000	3,08	3,38
	70000	4,72	4,81
	80000	6,10	5,00
	90000	6,71	6,05
	100000	6,62	6,67
Posortowane odwrotnie	10000	0,01	0,00
	20000	1,00	1,03
	30000	2,01	2,00
	40000	3,00	3,04
	50000	3,99	4,03
	60000	5,01	5,93
	70000	5,20	6,90
	80000	6,14	7,98
	90000	7,13	8,92
	100000	8,18	10,03
Posortowane w 33%	10000	1	1
	20000	2,07	3
	30000	4,06	5
	40000	6,04	7,03
	50000	7,09	9,06
	60000	9	11,68
	70000	11,02	14
	80000	12,24	16,03
	90000	14,01	18,37
	100000	16,01	20,99
Posortowane w 66%	10000	0,01	1
	20000	1,36	2,01
	30000	2,98	3,05
	40000	4	4,46
	50000	5	6,04
	60000	6,14	7,04
	70000	7,32	9
	80000	8,06	10,01
	90000	9,69	11,97
	100000	11,34	13,04


Czas sortowania Shella (wzór Tokudy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu int
Średni czas sortowania [ms]
0
5
10
15
20
25
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%


Czas sortowania Shella (wzór Tokudy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu float
Średni czas sortowania [ms]
0
5
10
15
20
25
30
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%

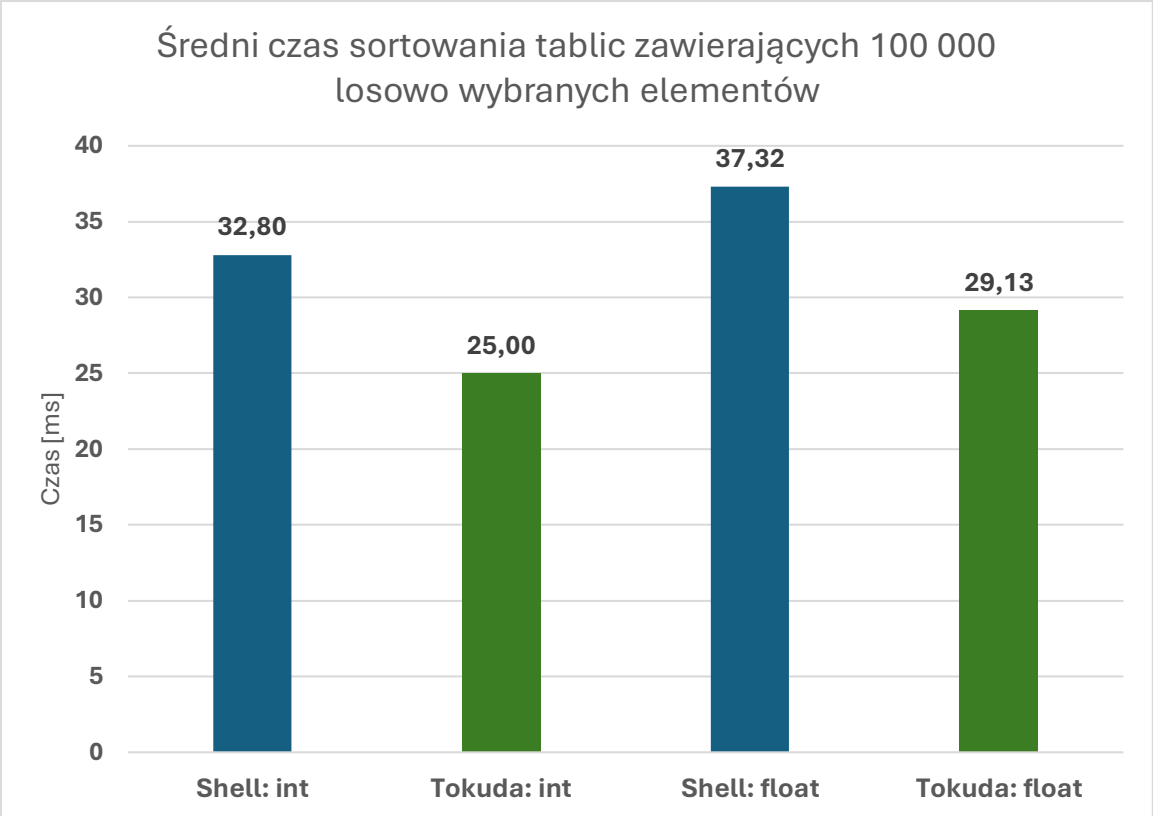
Wnioski

Czas działania sortowania Shella w dużym stopniu zależy od przyjętej sekwencji odstępów. W przeprowadzonych testach różnica w czasie sortowania między klasyczną sekwencją Shella a sekwencją Tokudy była zauważalna.

Sekwencja Tokudy okazała się bardziej efektywna – średni czas sortowania dla różnych przypadków początkowego ułożenia danych był krótszy w porównaniu do klasycznej sekwencji Shella. Wynika to z lepszego rozmieszczenia odstępów, które szybciej redukują nieuporządkowanie w tablicy.

Dla obu sekwencji zauważalny jest wpływ wstępnego ułożenia danych – posortowane dane były sortowane szybciej niż dane odwrotnie posortowane, choć różnice te nie były aż tak wyraźne jak w przypadku prostszego algorytmu sortowania (przez wstawianie).

Sortowanie Shella z sekwencją Tokudy jest sensownym kompromisem pomiędzy prostotą implementacji a efektywnością, zwłaszcza dla średnich rozmiarów danych.



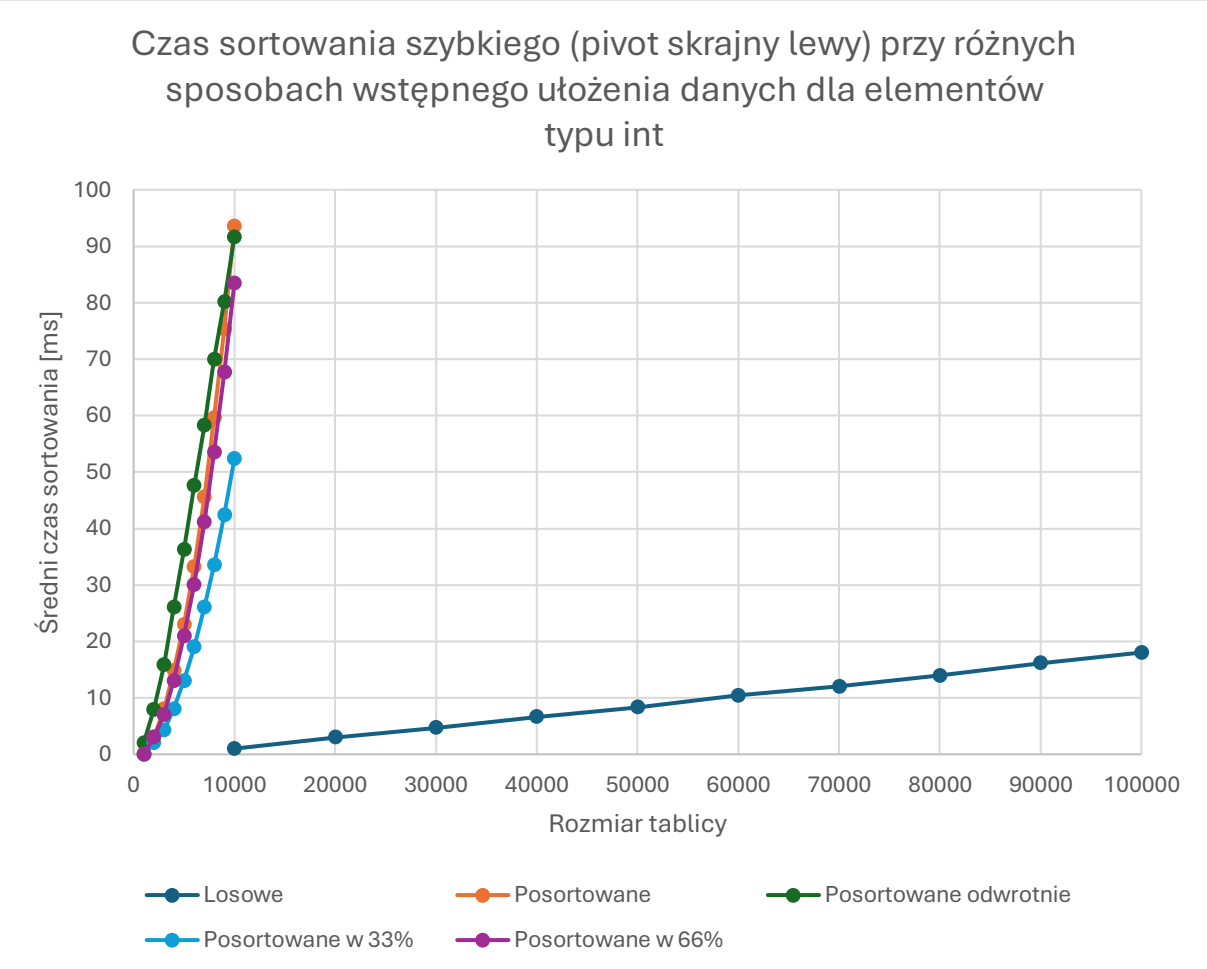
Sortowanie szybkie

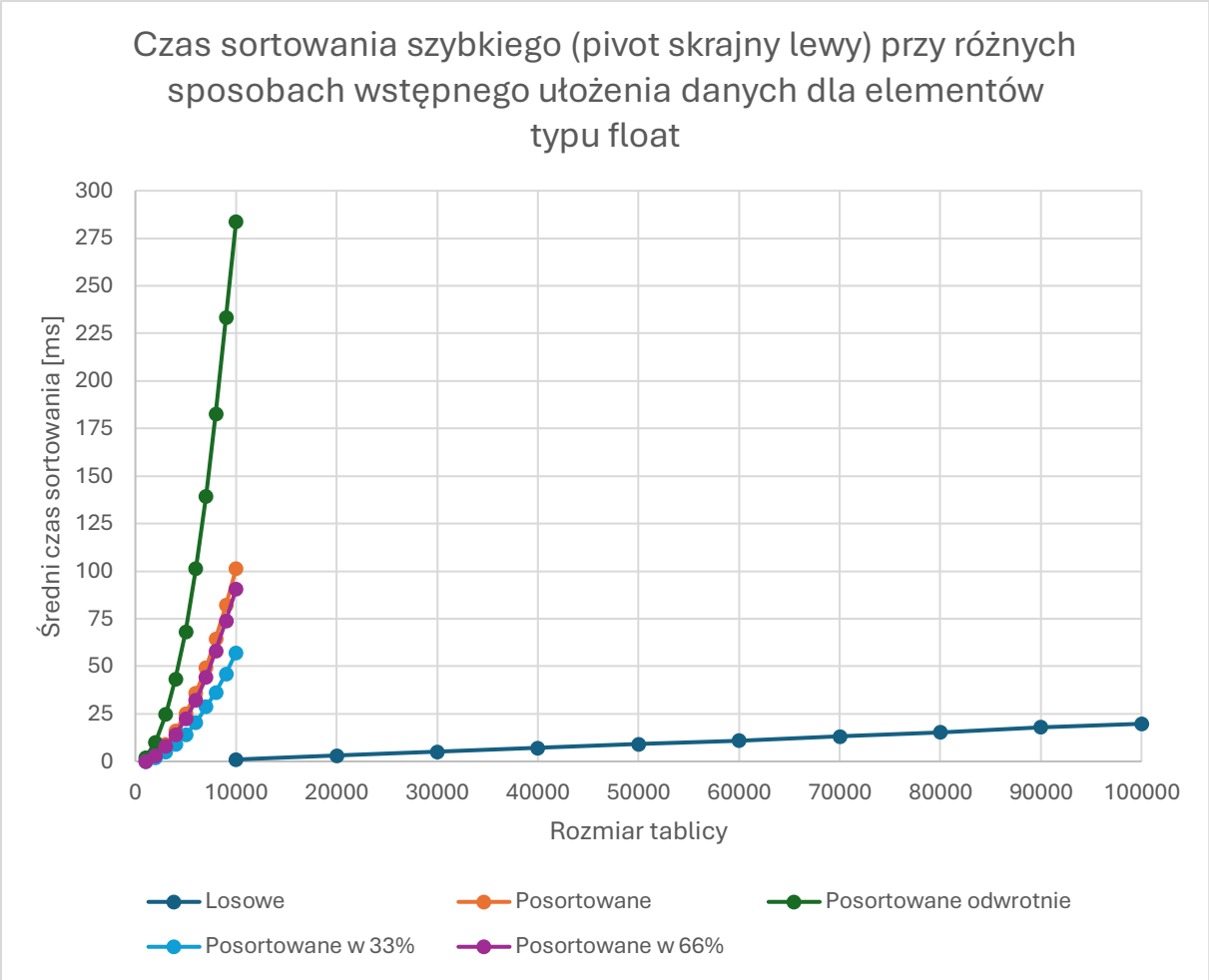
a) Pivot skrajny lewy

W przypadku sortowania szybkiego z wyborem skrajnego lewego pivota konieczne było zmniejszenie rozmiaru tablicy w przypadku danych posortowanych w całości lub częściowo, ze względu na zbyt głęboką rekurencję i długi czas sortowania.

Sortowanie szybkie (skrajny lewy)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	1,00	1,00
	20000	3,00	3,00
	30000	4,72	5,00
	40000	6,64	7,01
	50000	8,34	9,01
	60000	10,41	10,99
	70000	12,04	13,20
	80000	13,97	15,25
	90000	16,13	17,93
	100000	18,03	19,79
Posortowane	1000	0,03	1,00
	2000	3,03	4,00
	3000	8,01	9,00
	4000	14,82	16,00
	5000	23,00	25,01
	6000	33,18	36,01
	7000	45,58	49,24
	8000	59,58	64,52
	9000	75,32	82,08
	10000	93,55	101,22
Posortowane odwrotnie	1000	2,00	2,00
	2000	7,91	10,15
	3000	15,84	24,65
	4000	26,04	43,36
	5000	36,25	68,15
	6000	47,55	101,30
	7000	58,23	139,38
	8000	69,95	182,68
	9000	80,20	233,29
	10000	91,65	283,61

Posortowane w 33%	1000	0	0
	2000	2	2,01
	3000	4,26	5
	4000	8,02	9,01
	5000	13,04	14,02
	6000	19,01	20,41
	7000	26,07	28,7
	8000	33,49	36,29
	9000	42,41	46,05
	10000	52,34	57,17
Posortowane w 66%	1000	0,02	0,06
	2000	3	3,03
	3000	7,02	8,06
	4000	13,04	14,08
	5000	20,98	22,32
	6000	30,06	32,17
	7000	41,11	44,12
	8000	53,45	58,06
	9000	67,67	73,96
	10000	83,45	90,6



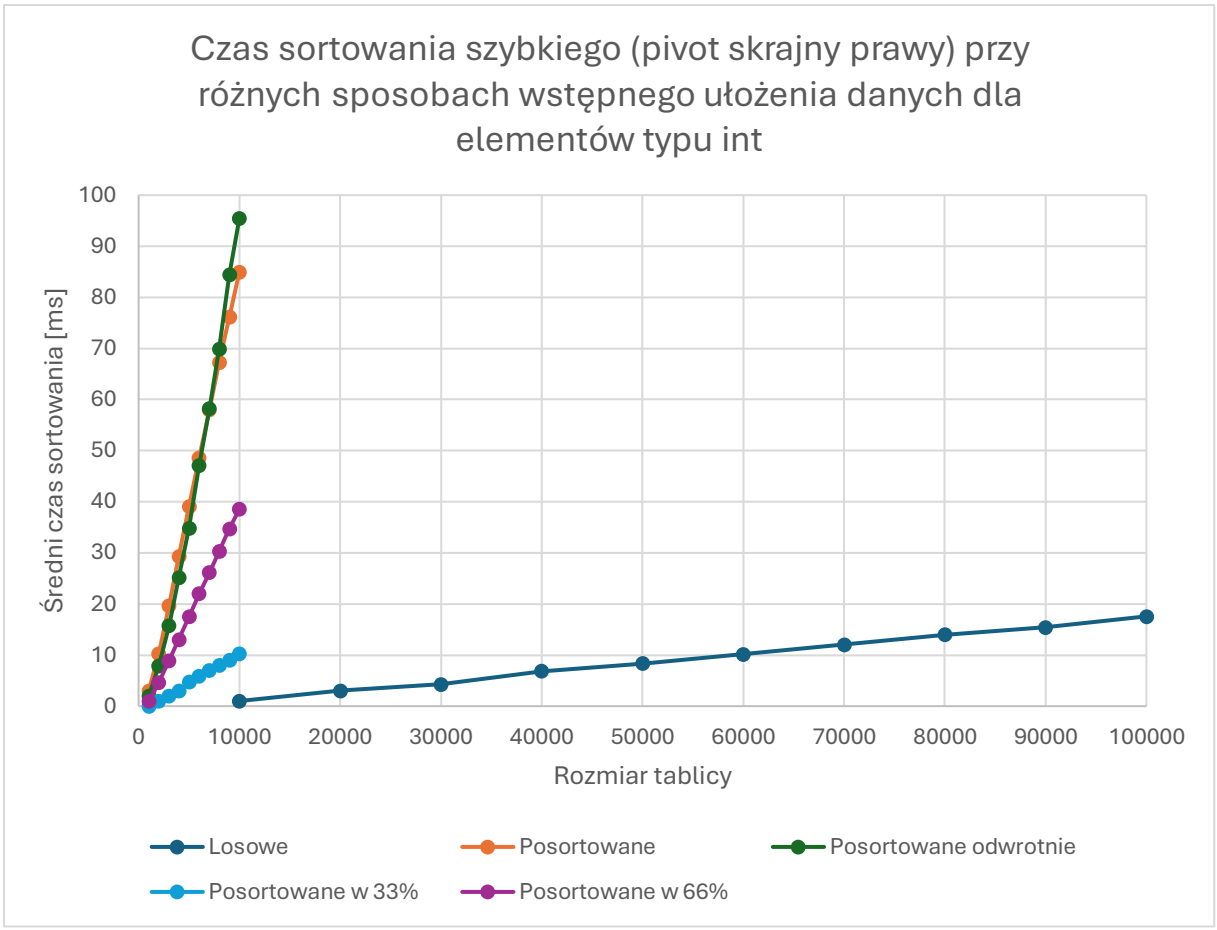


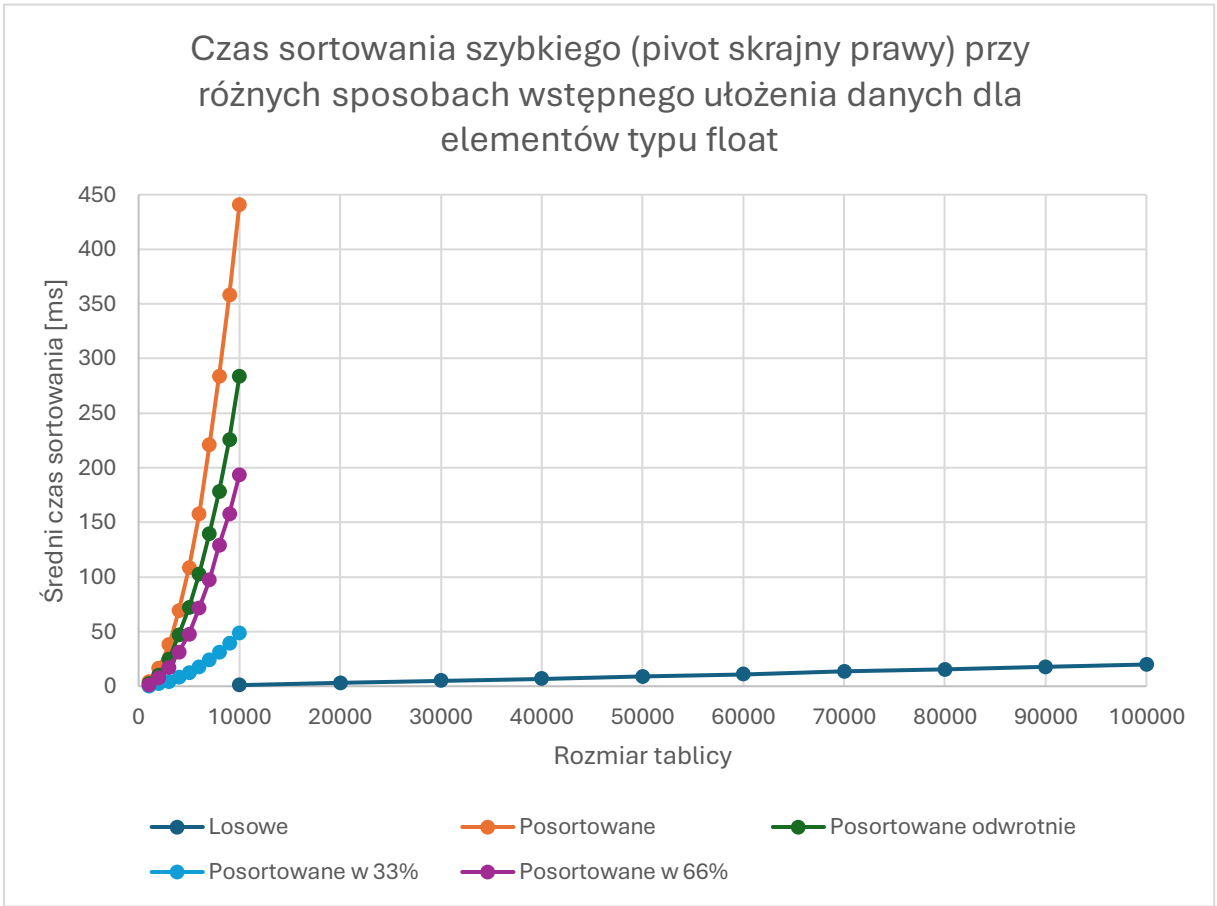
b) Pivot skrajny prawy

W tym przypadku również konieczne było zmniejszenie rozmiaru tablicy, ze względu na zbyt głęboką rekurencję i długi czas sortowania.

Sortowanie szybkie (skrajny prawy)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	1,00	1,00
	20000	2,99	3,00
	30000	4,29	4,98
	40000	6,85	6,91
	50000	8,35	8,94
	60000	10,15	10,89
	70000	12,03	13,43
	80000	13,98	15,39
	90000	15,43	17,70
	100000	17,57	19,89

Posortowane	1000	3,00	4,00
	2000	10,26	16,20
	3000	19,69	37,93
	4000	29,25	68,84
	5000	39,02	108,22
	6000	48,64	157,38
	7000	57,93	220,76
	8000	67,20	283,65
	9000	76,10	357,84
	10000	84,98	440,80
Posortowane odwrotnie	1000	2,00	2,05
	2000	7,90	10,13
	3000	15,74	24,78
	4000	25,10	46,80
	5000	34,85	71,86
	6000	47,14	102,49
	7000	58,20	139,43
	8000	69,86	177,88
	9000	84,37	225,57
	10000	95,47	283,69
Posortowane w 33%	1000	0	0
	2000	1	2
	3000	2	4
	4000	3,03	7,99
	5000	4,68	12,02
	6000	5,89	17,25
	7000	6,96	23,98
	8000	8,01	31,03
	9000	9,03	39,08
	10000	10,19	48,27
Posortowane w 66%	1000	1	1,13
	2000	4,63	7,32
	3000	8,9	16,99
	4000	13,05	31,01
	5000	17,51	47,57
	6000	22,01	71,63
	7000	26,11	97,11
	8000	30,29	128,92
	9000	34,69	157,72
	10000	38,51	193,3


Czas sortowania szybkiego (pivot skrajny prawy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu int
Średni czas sortowania [ms]
0
10
20
30
40
50
60
70
80
90
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%

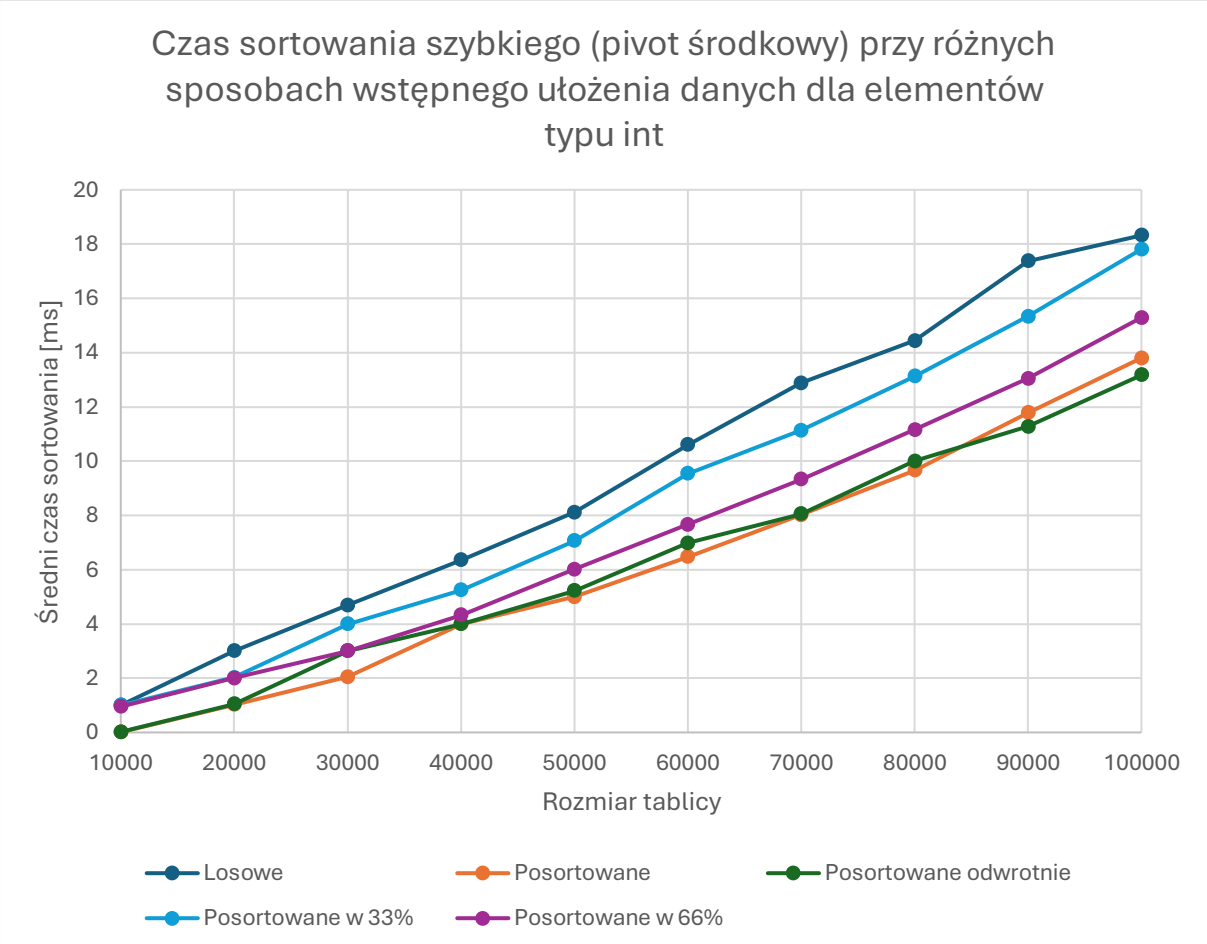

Czas sortowania szybkiego (pivot skrajny prawy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu float
Średni czas sortowania [ms]
0
50
100
150
200
250
300
350
400
450
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%

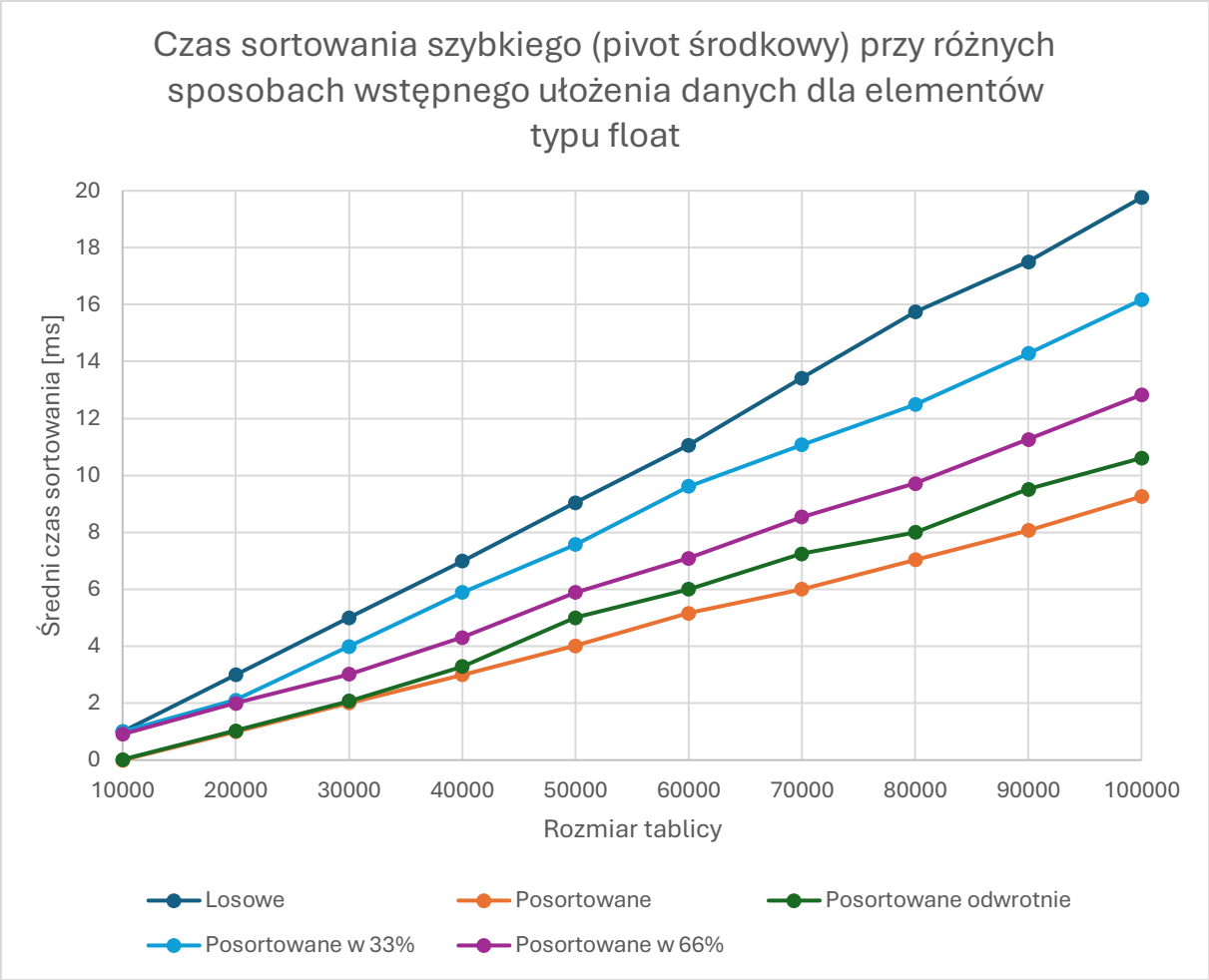
c) Pivot środkowy

W przypadku wyboru pivota środkowego algorytm działał bardzo szybko – nie wystąpiła potrzeba zmniejszenia rozmiaru tablicy. Czas mierzony był dla rozmiarów od 10000 do 100000 elementów. Wyniki przedstawiono w poniższej tabeli oraz następujących po niej wykresach.

Sortowanie szybkie (środkowy)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	1,00	1,00
	20000	3,00	3,00
	30000	4,69	5,00
	40000	6,35	6,99
	50000	8,11	9,04
	60000	10,60	11,06
	70000	12,88	13,42
	80000	14,44	15,74
	90000	17,37	17,51
	100000	18,32	19,76
Posortowane	10000	0,01	0,00
	20000	1,02	1,00
	30000	2,05	2,01
	40000	4,00	3,00
	50000	5,00	4,02
	60000	6,47	5,16
	70000	8,02	6,00
	80000	9,66	7,04
	90000	11,78	8,07
	100000	13,80	9,26
Posortowane odwrotnie	10000	0,02	0,02
	20000	1,04	1,03
	30000	3,00	2,08
	40000	4,00	3,29
	50000	5,22	5,00
	60000	6,98	6,00
	70000	8,05	7,25
	80000	10,00	8,00
	90000	11,27	9,52
	100000	13,17	10,61

Posortowane w 33%	10000	1	1
	20000	2,03	2,11
	30000	3,99	4
	40000	5,25	5,89
	50000	7,06	7,58
	60000	9,54	9,62
	70000	11,13	11,08
	80000	13,13	12,49
	90000	15,33	14,29
	100000	17,81	16,17
Posortowane w 66%	10000	0,95	0,91
	20000	2	2
	30000	3	3,02
	40000	4,33	4,31
	50000	6,01	5,89
	60000	7,66	7,09
	70000	9,33	8,54
	80000	11,16	9,72
	90000	13,05	11,27
	100000	15,29	12,83



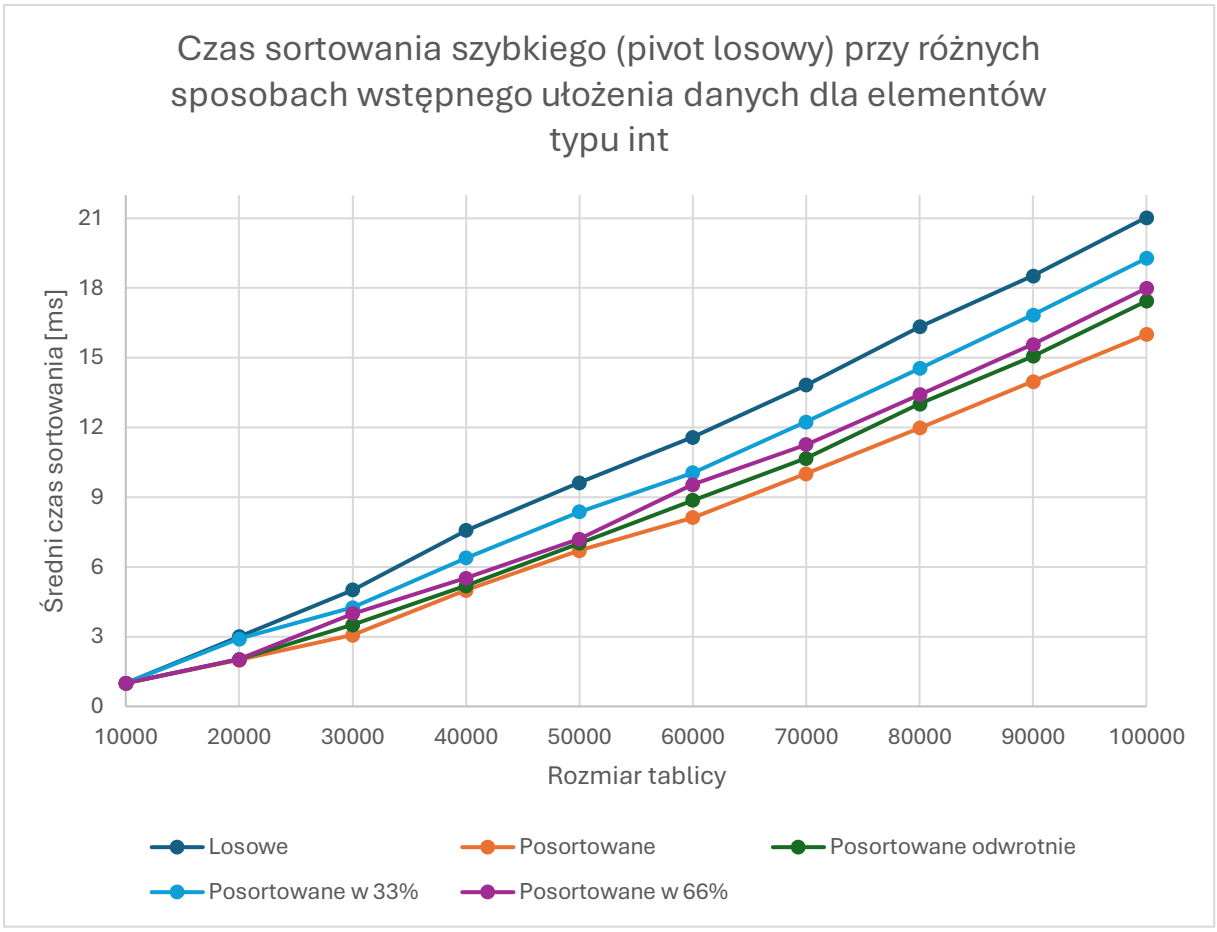


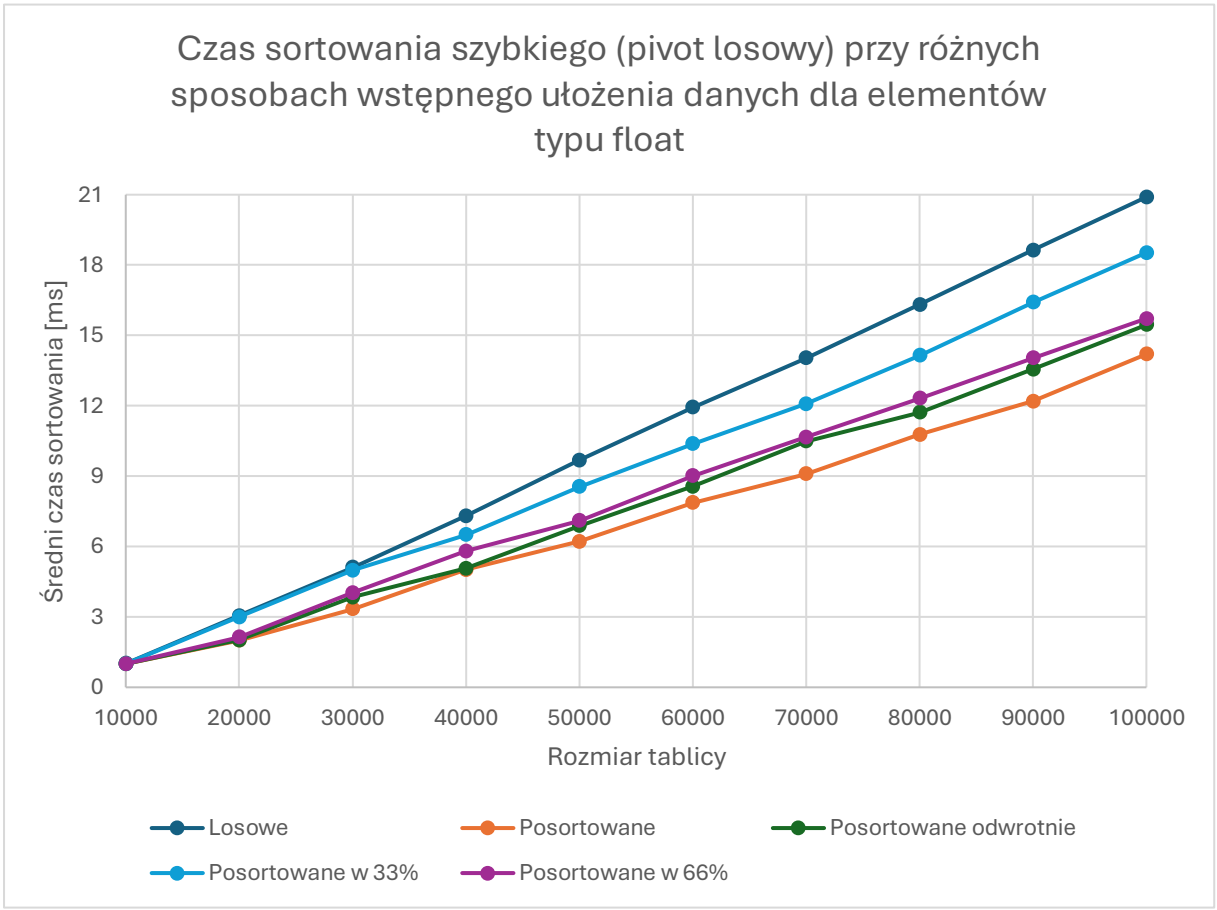
d) Pivot losowy

W przypadku pivota losowego również nie wystąpiła potrzeba zmniejszenia rozmiaru tablicy, a algorytm działał bardzo szybko dla wszystkich ułożeń danych.

Sortowanie szybkie (losowy)			
Wstępne ułożenie danych	**Rozmiar tablicy**	**Średni czas sortowania**	
		int	**float**
Losowe	10000	1,00	1,01
	20000	3,00	3,05
	30000	5,01	5,11
	40000	7,57	7,31
	50000	9,63	9,68
	60000	11,58	11,93
	70000	13,83	14,03
	80000	16,33	16,32
	90000	18,52	18,63
	100000	21,04	20,90

Posortowane	10000	1,00	1,00
	20000	2,00	2,00
	30000	3,07	3,33
	40000	5,00	5,02
	50000	6,71	6,21
	60000	8,13	7,87
	70000	10,01	9,09
	80000	11,98	10,77
	90000	13,98	12,19
	100000	16,00	14,20
Posortowane odwrotnie	10000	1,00	1,00
	20000	2,03	2,01
	30000	3,51	3,84
	40000	5,19	5,07
	50000	7,03	6,88
	60000	8,87	8,56
	70000	10,68	10,49
	80000	13,02	11,72
	90000	15,07	13,56
	100000	17,44	15,45
Posortowane w 33%	10000	1	1
	20000	2,91	3
	30000	4,25	4,99
	40000	6,39	6,5
	50000	8,37	8,55
	60000	10,06	10,38
	70000	12,24	12,09
	80000	14,54	14,15
	90000	16,84	16,41
	100000	19,28	18,53
Posortowane w 66%	10000	1	1
	20000	2,03	2,14
	30000	4	4,03
	40000	5,52	5,8
	50000	7,2	7,1
	60000	9,54	9,02
	70000	11,27	10,66
	80000	13,42	12,31
	90000	15,58	14,04
	100000	17,99	15,72


Czas sortowania szybkiego (pivot losowy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu int
Średni czas sortowania [ms]
0
3
6
9
12
15
18
21
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%


Czas sortowania szybkiego (pivot losowy) przy różnych sposobach wstępnego ułożenia danych dla elementów typu float
Średni czas sortowania [ms]
0
3
6
9
12
15
18
21
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Rozmiar tablicy
Losowe
Posortowane
Posortowane odwrotnie
Posortowane w 33%
Posortowane w 66%

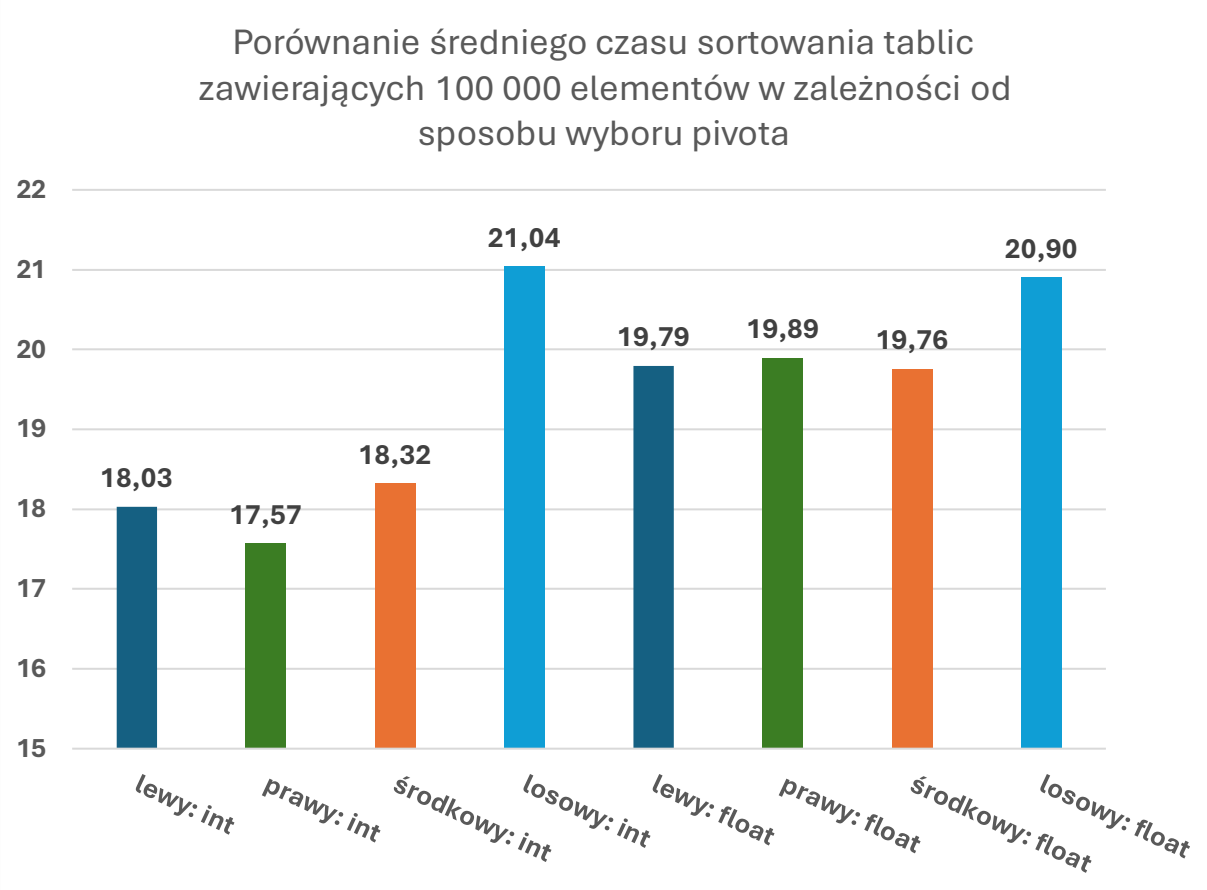
Wnioski

Wydajność algorytmu sortowania szybkiego jest silnie uzależniona od strategii wyboru pivota oraz od początkowego ułożenia danych. Największe różnice w czasie działania występowały w przypadku pivotów skrajnych (lewego i prawego) dla danych posortowanych oraz odwrotnie posortowanych. W tych przypadkach czas działania algorytmu był szczególnie długi, ponieważ dzielenie tablicy było bardzo niesymetryczne.

Wybór pivota środkowego lub losowego dawał o wiele lepsze rezultaty niż pivoty skrajne, szczególnie w przypadku danych uporządkowanych. Oba zapewniały względnie stabilny i niski czas sortowania, niezależnie od początkowego ułożenia danych. Pivoty średni i losowy prowadzą do bardziej symetrycznego dzielenia tablicy, dzięki czemu minimalizują ryzyko trafienia na najgorszy przypadek i osiągniecie złożoności $O(n^2)$.

Dla danych kompletnie losowych wszystkie strategie działały dobrze, osiągając zbliżone czasy. Najdłuższy czas sortowania wystąpił w przypadku pivota losowego, co może wynikać z niewielkiego narzutu czasowego związanego z generowaniem losowej liczby przy każdym podziale tablicy.

Na podstawie przeprowadzonych testów można stwierdzić, że stosowanie środkowego lub losowego pivota jest najbezpieczniejszym i najbardziej uniwersalnym wyborem w implementacjach sortowania szybkiego, szczególnie w sytuacjach, gdy nie można przewidzieć struktury danych wejściowych.



Podsumowanie

W ramach przeprowadzonych testów porównana została wydajność różnych algorytmów sortowania w zależności od typu danych wejściowych, ich początkowego uporządkowania oraz wariantów w ich implementacjach.

Algorytmy proste, takie jak sortowanie przez wstawianie, wykazują bardzo długi czas działania dla większych tablic i są wrażliwe na stopień uporządkowania danych. Ich zastosowanie ogranicza się do bardzo małych zbiorów danych lub jako część bardziej złożonych hybrydowych metod sortowania.

Sortowanie przez kopcowanie charakteryzuje się dużą stabilnością czasów niezależnie od ułożenia danych. Sprawdza się dobrze w przypadku danych trudnych do przewidzenia i skrajnych przypadków.

Dla sortowania Shella wykazano, że wybór sekwencji podziałów ma znaczący wpływ na wydajność – sekwencja Tokudy daje lepsze rezultaty niż klasyczna sekwencja Shella, szczególnie przy większych tablicach.

W przypadku sortowania szybkiego kluczowe znaczenie ma wybór pivota. Pivoty skrajne (lewy, prawy) prowadzą do bardzo nieefektywnego działania w przypadku danych uporządkowanych lub odwrotnie uporządkowanych. Natomiast pivoty środkowy i losowy zapewniają stabilne i szybkie działanie niezależnie od danych, co czyni je najbardziej uniwersalnymi w praktyce.

Podsumowując, nie istnieje jeden najlepszy algorytm sortowania – wybór odpowiedniej metody zależy od charakterystyki danych wejściowych, wymaganej szybkości działania i dostępnych zasobów. W praktyce często stosuje się algorytmy hybrydowe, łączące zalety kilku metod.

Literatura

[1] Cormen T., Leiserson C.E., Rivest R.L., Stein C., Wprowadzenie do algorytmów, WNT
[2] Wisnu A., C++. Struktury danych i algorytmy, Helion S.A.
[3] Rytter W., Diks K., Banachowski L., Algorytmy i struktury danych, PWN