PAMSI - projekt 2
Algorytmy sortowania

Aleksandra Rzeszowska (234780)
środa, 18:55-20:35

25 kwietnia 2018

1 Wstęp teoretyczny

Podczas realizacji projektu zaimplementowano trzy rodzaje sortowań - sortowanie przez scalanie, sortowanie szybkie oraz introspektywne, a następnie przeprowadzono na nich testy efektywności.

1.1 Sortowanie przez scalanie

Sortowanie przez scalanie jest przykładem sortowania rekurencyjnego z grupy algorytmów szybkich, który wykorzystuje metodę *dziel i zwyciężaj*. Tablicę do posortowania dzielimy na dwie podtablice, na których rekurencyjnie wywołuje się tę samą funkcję do momentu, gdy podtablice będą zawierały tylko jeden element. Całe właściwe sortowanie przerzucone jest na funkcję scalającą.

1.2 Sortowanie szybkie

Sortowanie szybkie, podobnie jak sortowanie przez scalanie, wykorzystuje metodę *dziel i zwyciężaj*. W tym rodzaju sortowania praca nad sortowaniem tablicy wykonana jest podczas dzielenia problemu na dwa mniejsze podproblemy. Do wyznaczenia lewego i prawego podproblemu niezbędna jest wartość, względem której zostaną stworzone dwie podtablice - z wartościami mniejszymi od porównania oraz druga z większymi.

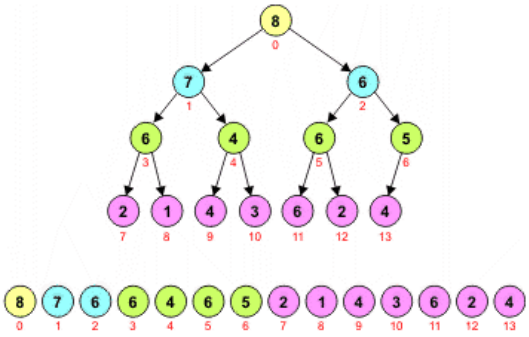
1.3 Sortowanie introspektywne

Sortowanie introspektywne to sortowanie, którego celem jest wyeliminowanie kwadratowej złożoności obliczeniowej algorytmu sortowania szybkiego. W tym sortowaniu określana jest wartość maksymalnej ilości rekurencyjnych wywołań ($max = 2 \cdot log_2 N$, gdzie $N -$rozmiar tablicy). Gdy $max = 0$ wywoływane jest sortowanie przez kopcowanie. W czasie sortowania, gdy $max > 0$, wywoływana jest rekurencyjne funkcja $SortujIntro$ z parametrem max o 1 mniejszym. $SortujIntro$, podobnie jak sortowanie szybkie, również dzieli problem na dwa mniejsze podproblemy.

1.3.1 Sortowanie przez kopcowanie

Sortowanie przez kopcowanie to przykład sortowania z grupy algorytmów szybkich. Sortowanie to wykorzystuje kopiec (binarne drzewo) reprezentowane przez tablicę. Przykład takiej reprezentacji przedstawia rysunek [1] na kolejnej stronie:

[1] Rysunek zaczerpnięto ze strony internetowej $http://eduinf.waw.pl/inf/alg/001_search/0113.php$



Rysunek 1: Reprezentacja kopca jako tablicy

Synowie $k - tego$ węzła mają indeksy $2 \cdot k + 1$ oraz $2 \cdot k + 2$. Sortowanie rozpoczyna się od ostatniego rodzica i, w razie potrzeby, zamieniane jest dziecko z rodzicem, by zachować strukturę kopca - każdy z rodziców ma wartość większą niż każde z jego dzieci.

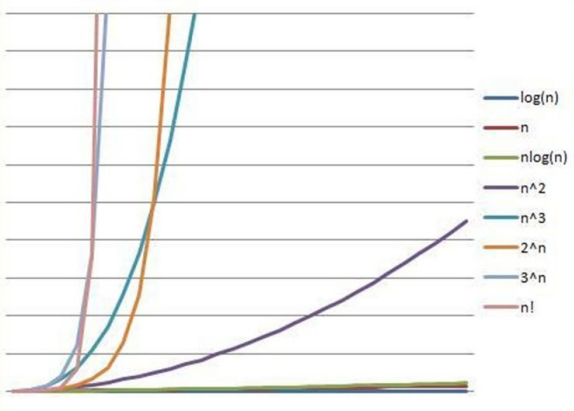
2 Złożoność obliczeniowa

Złożoność obliczeniową każdego z zaimplementowanych sortowań dla przypadku średniego i najgorszego przedstawia tabela poniżej:

sortowanie	czasowa		pamięciowa	
	średni	najgorszy	średni	najgorszy
przez scalanie	$O(n \cdot log_2 n)$		$O(n)$	
szybkie	$O(n \cdot log_2 n)$	$O(n^2)$	$O(1)$	$O(log_2 n)$
przez kopcowanie	$O(n \cdot log_2 n)$		$O(1)$	
introspektywne	$O(n \cdot log_2 n)$		$O(1)$	

Tabela 1: Złożoność obliczeniowa zaimplementowanych algorytmów

Złożoność obliczeniową w notacji dużego O przedstawia rysunek [2] poniżej:



Rysunek 2: Złożoność obliczeniowa w notacji dużego O

[2] http://slideplayer.pl/slide/8941861/

3 Sposób implementacji

Wszystkie funkcje zostały zaimplementowane przy użyciu szablonów (*template<typedef typ>*), by mogły zostać wykorzystane do sortowania różnego typu danych.

- **void Wyswietl(typ tablica[], int ilosc)** - wyświetla zawartość tablicy od elementu o indeksie 0 do elementu o indeksie $ilosc - 1$
- **void SortujScalanie(typ tablica[], int poczatek, int koniec)** - rekurencyjne funkcja realizująca sortowanie przez scalanie. Sortuje tablicę dowolnego typu od indeksu o numerze *poczatek* do indeksu o numerze *koniec*
- **void Scal(typ tablica[], int poczatek, int srodek, int koniec)** - funkcja realizująca scalanie podzielonej wcześniej tablicy. Na tę funkcję przerzucone jest całe właściwe sortowanie.
- **int WybierzPorownanie(typ tablica[], int lewy, int prawy)** - funkcja zwraca indeks elementu, który ma być porównaniem podczas sortowania szybkiego
- **void SortujSzybkie(typ tablica[], int lewy, int prawy)** - funkcja realizująca sortowanie szybkie. Działa rekurencyjnie, dzieląc tablicę na dwie mniejsze podtablice, dla których wywołuje się rekurencyjnie
- **void SortujKopcowanie(typ tablica[], int dlugosc)** - funkcja realizująca sortowanie przez kopcowanie
- **void MaxKopiec(typ tablica[], int dlugosc, int indeks_rodzica)** - pomocnicza funkcja potrzebna przy sortowaniu przez kopcowanie. Funkcja ta tworzy maksymalny kopiec, czyli taki, w którym każdy z rodziców ma większą wartość niż każde z jego dzieci
- **void SortujWstawianie(typ tablica[], int dlugosc)** - funkcja realizuje działanie sortowania przez wstawianie
- **void SortujIntro(typ tablica[], int dlugosc, int max)** - funkcja sortująca o implementacji podobnej do implementacji sortowania szybkiego
- **int Podziel(typ tablica[], int lewy, int prawy)** - funkcja pomocnicza sortowania *SortujIntro*, partycjonująca tablicę na dwie części, dla których wywoływana jest funkcja sortująca
- **void SortujIntrospektywne(typ tablica[], int dlugosc)** - funkcja właściwa sortowania introspektywnego
- **bool SprawdzSortowanie(typ tablica[], int dlugosc)** - funkcja do sprawdzenia poprawności sortowania
- **void TestujWstawianie(typ tablica[], int dlugosc, double &wstawianie)** - funkcja testująca działanie sortowania przez wstawianie
- **void TestujScalanie(typ tablica[], int dlugosc, double &scalanie)** - funkcja testująca działanie sortowania przez scalanie
- **void TestujSzybkie(typ tablica[], int dlugosc, double &szybkie)** - funkcja testująca działanie sortowania szybkiego
- **void TestujKopcowanie(typ tablica[], int dlugosc, double &kopcowanie)** - funkcja testująca działanie sortowania przez kopcowanie
- **void TestujIntrospektywne(typ tablica[], int dlugosc, double &introspektywne)** - funkcja testująca działanie sortowania introspektywnego
- **void Test(typ tablica[], int dlugosc, double &scalanie, double &szybkie, double &kopcowanie, double &introspektywne)** - funkcja testująca działanie każdego z sortowań (przez scalanie, szybkie, przez kopcowanie, introspektywne)
- **void LosowaTablica(typ tablica[], const int rozmiar)** - funkcja generująca tablicę wartości od 0 do dlugosc w losowej kolejności

- **void OdwrotnaTablica(typ tablica[], const int rozmiar** - funkcja generująca tablicę wartości od 0 do dlugosc uporządkowanych w kolejności od największej do najmniejszej
- **void Tablica(typ tablica[], const int rozmiar, double procent)** - funkcja generująca tablice o zadanej długości, której procent% początkowych wartości jest już posortowanych w kolejności od najmniejszej do największej, pozostałe są ustawione w kolejności losowej
- **void PosortowanaTablica(typ tablica[], const int rozmiar)** - funkcja generująca tablicę o zadanej długości, której wartości sa posortowane od najmniejszej do największej

4 Testy efektywności

Tak zaimplementowane sposoby sortowania zostały poddane testom efektywności. Dla każdej z rodzajów tablic:

- elementy losowe
- elementy posortowane w odwrotnej kolejności
- 25%, 50%, 75%, 95%, 99%, 99,7% początkowych elementów już posortowanych
- elementy posortowane

oraz dla każdej z ilości elementów w tablicy (10 000, 50 000, 100 000, 500 000, 1 000 000, 2 000 000). Ich opis i rezultaty przedstawiono poniżej.

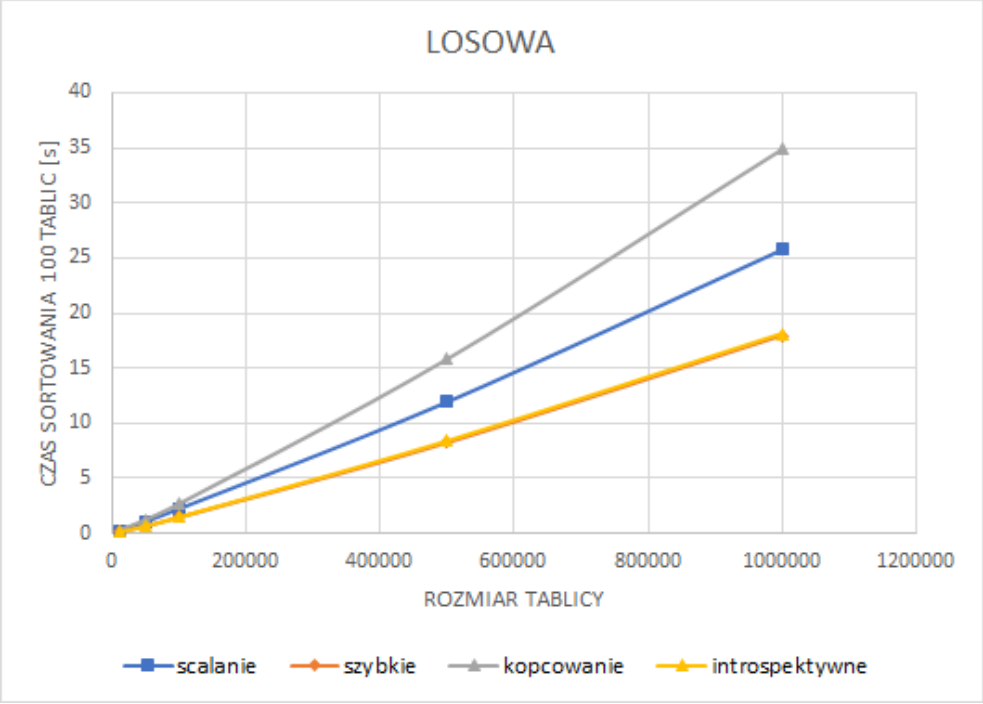
4.1 Elementy tablicy losowe

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

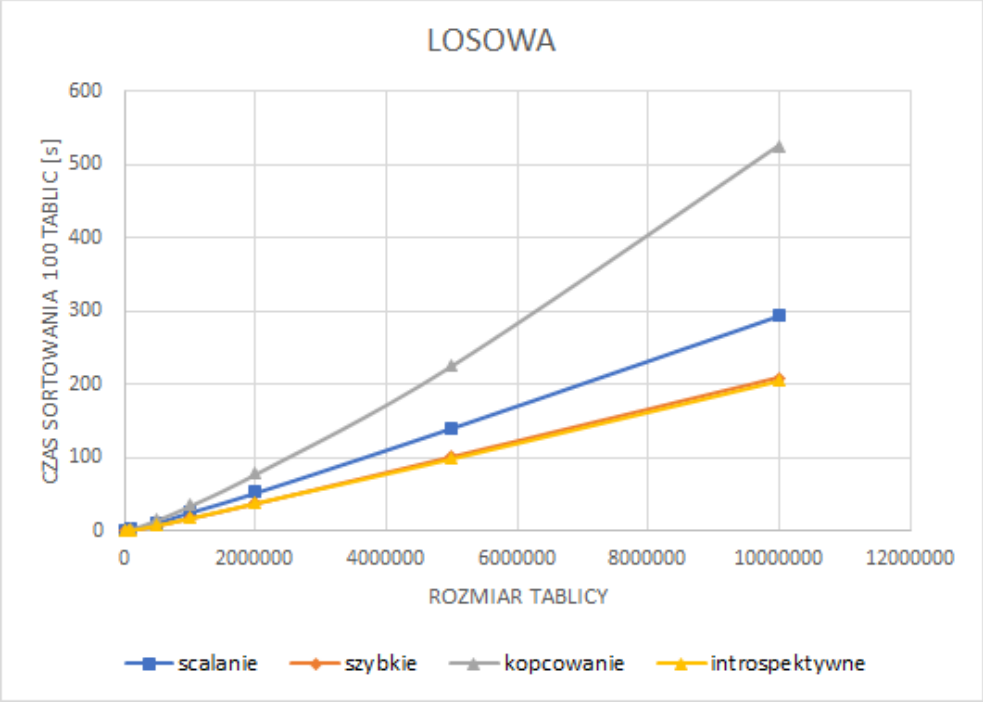
Tabela 2: Czas sortowania dla tablicy w 25% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,194525	0,123372	0,215742	0,125154
50000	1,0141	0,693824	1,209761	0,686327
100000	2,178324	1,479793	2,655989	1,500892
500000	11,922449	8,253421	15,822621	8,41483
1000000	25,813485	17,950627	34,907151	18,139577
2000000	52,714486	38,03902	78,007458	38,557792
5000000	140,626387	101,725065	225,275789	99,347967
10000000	293,256713	209,309972	525,102268	205,393223

Dane pomiarowe umieszczono na wykresach poniżej:



Rysunek 3: Czas sortowania tablicy uporządkowanej w losowy sposób w funkcji ilości elementów tablicy



Rysunek 4: Czas sortowania tablicy uporządkowanej w losowy sposób w funkcji ilości elementów tablicy

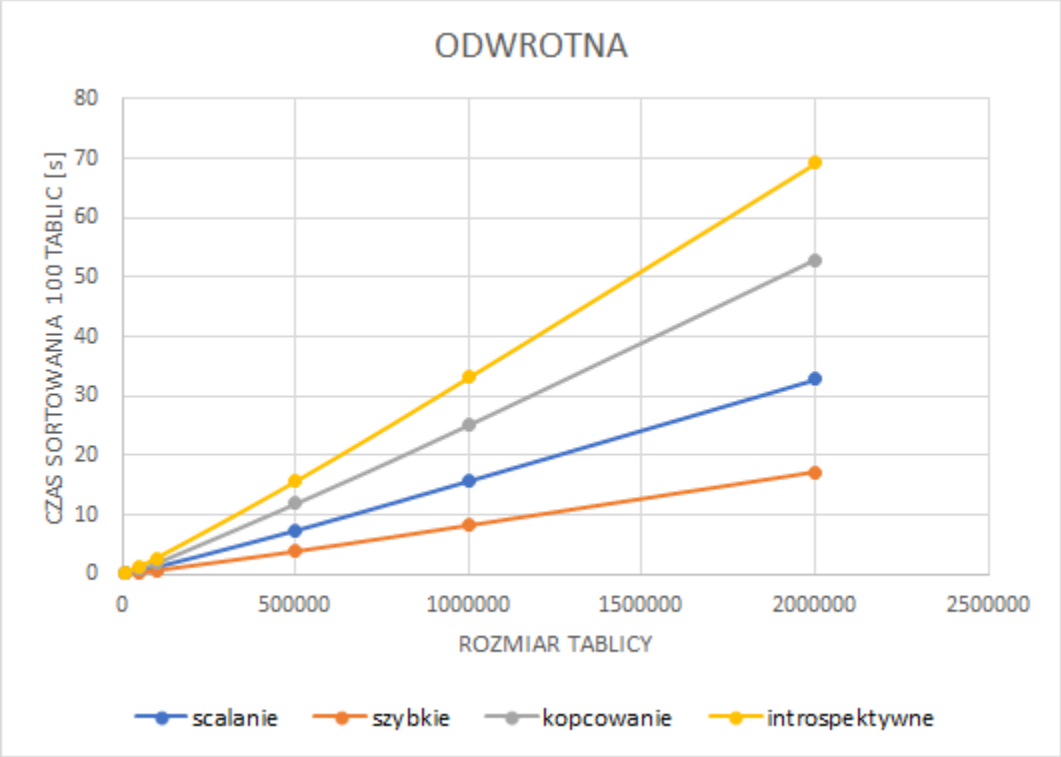
4.2 Wszystkie elementy tablicy już posortowane, ale w odwrotnej kolejności

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 3: Czas sortowania dla tablicy uporządkowanej w odwrotnym porządku

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,118207	0,058541	0,168886	0,220455
50000	0,638763	0,322615	0,961159	1,258827
100000	1,307941	0,67289	2,00561	2,64749
500000	7,452097	3,873703	11,976912	15,568203
1000000	15,800441	8,238916	25,20245	33,116007
2000000	32,901076	17,201392	52,991479	69,29152

Dane pomiarowe umieszczono na wykresie:



Rysunek 5: Czas sortowania tablicy uporządkowanej w odwrotnej kolejności w funkcji ilości elementów tablicy

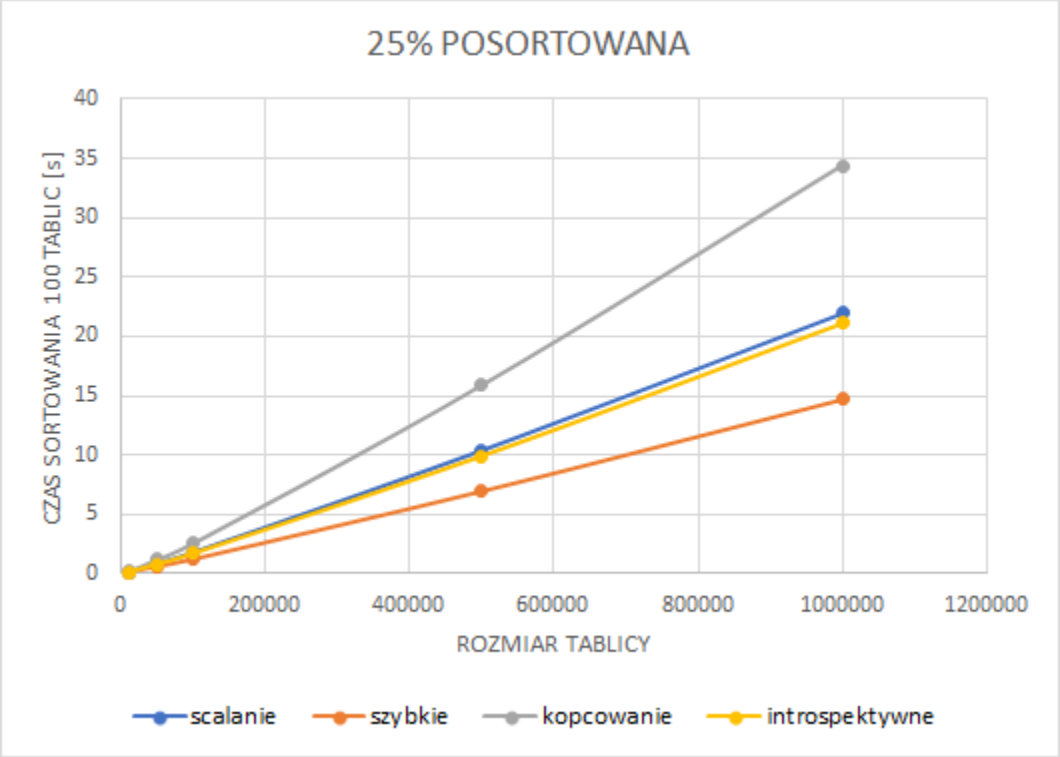
4.3 25% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 4: Czas sortowania dla tablicy w 25% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,156331	0,095782	0,202211	0,136872
50000	0,869112	0,565966	1,211511	0,806611
100000	1,807863	1,182445	2,588044	1,717915
500000	10,414407	6,945054	15,902818	9,929029
1000000	21,965925	14,720458	34,400336	21,10642
2000000	45,749263	31,068679	76,852092	44,625722

Dane pomiarowe umieszczono na wykresie:



Rysunek 6: Czas sortowania tablicy w 25% uporządkowanej w funkcji ilości elementów tablicy

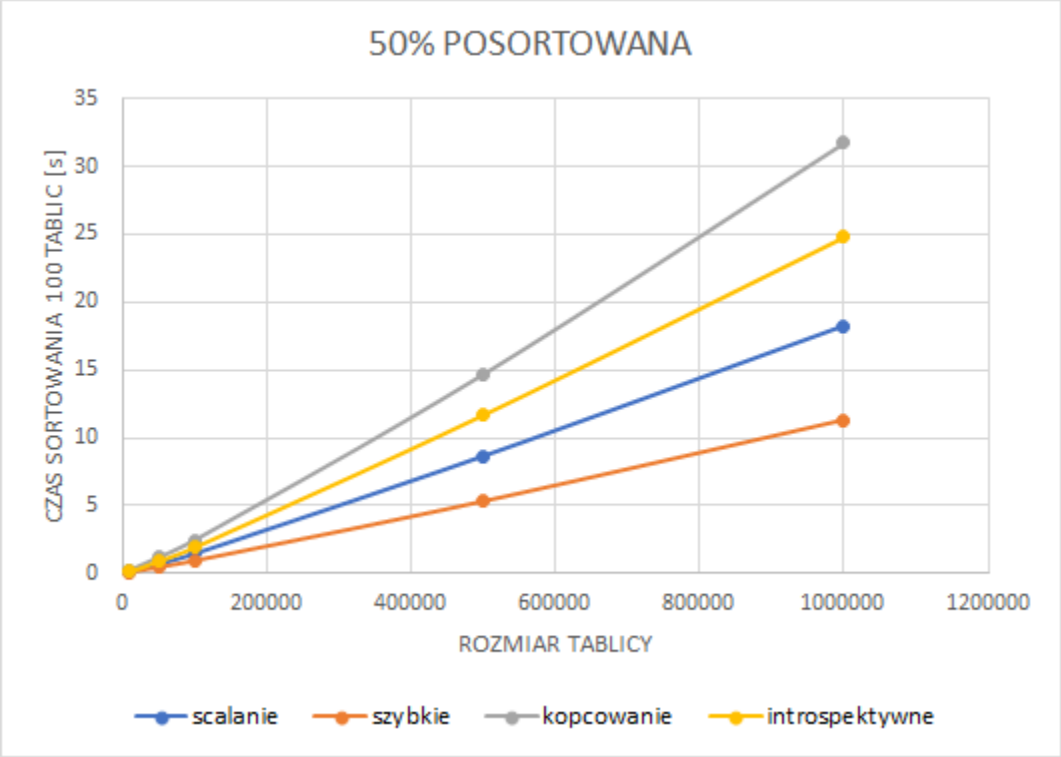
4.4 50% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 5: Czas sortowania dla tablicy w 50% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,136815	0,079368	0,202972	0,166696
50000	0,730319	0,439722	1,15777	0,947532
100000	1,493631	0,923141	2,425796	1,976559
500000	8,649051	5,318977	14,654435	11,723687
1000000	18,197809	11,328671	31,710878	24,809353
2000000	37,595717	23,678944	68,975288	51,873822

Dane pomiarowe umieszczono na wykresie:



Rysunek 7: Czas sortowania tablicy w 50% uporządkowanej w funkcji ilości elementów tablicy

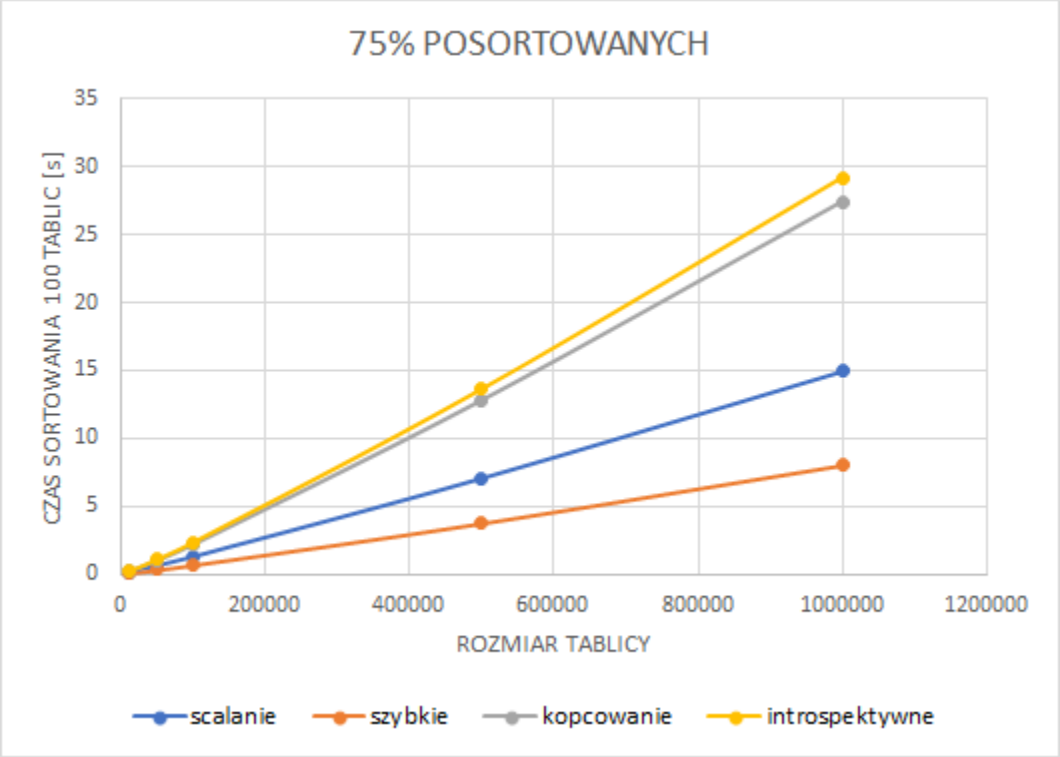
4.5 75% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 6: Czas sortowania dla tablicy w 75% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,116064	0,055861	0,191114	0,201449
50000	0,598974	0,311475	1,03383	1,109906
100000	1,253231	0,663649	2,208718	2,366085
500000	7,053238	3,729385	12,821081	13,6407
1000000	14,964038	7,984818	27,434333	29,196206
2000000	30,797544	16,698633	57,651759	60,824713

Dane pomiarowe umieszczono na wykresie:



Rysunek 8: Czas sortowania tablicy w 75% uporządkowanej w funkcji ilości elementów tablicy

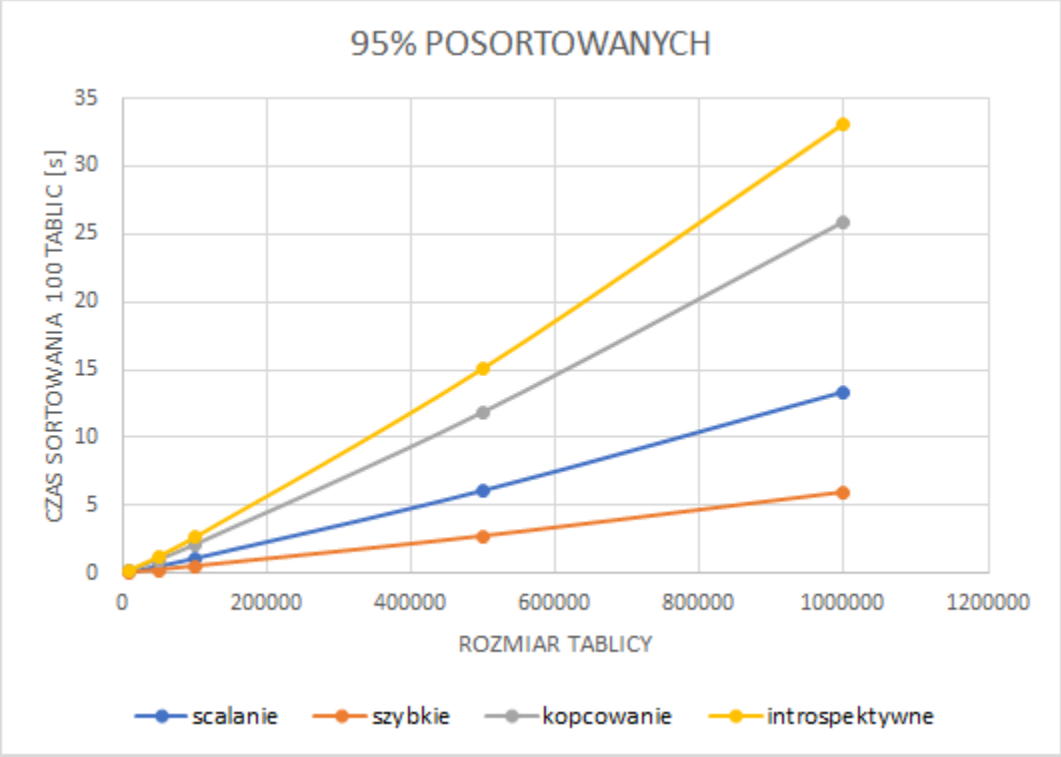
4.6 95% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 7: Czas sortowania dla tablicy w 95% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,10396	0,042678	0,181024	0,226698
50000	0,529058	0,234921	0,976443	1,238005
100000	1,109134	0,499271	2,08025	2,651178
500000	6,096833	2,724097	11,8714	15,107781
1000000	13,337679	5,958733	25,91996	33,125752
2000000	27,94076	12,603395	54,712027	70,755903

Dane pomiarowe umieszczono na wykresie:



Rysunek 9: Czas sortowania tablicy w 95% uporządkowanej w funkcji ilości elementów tablicy

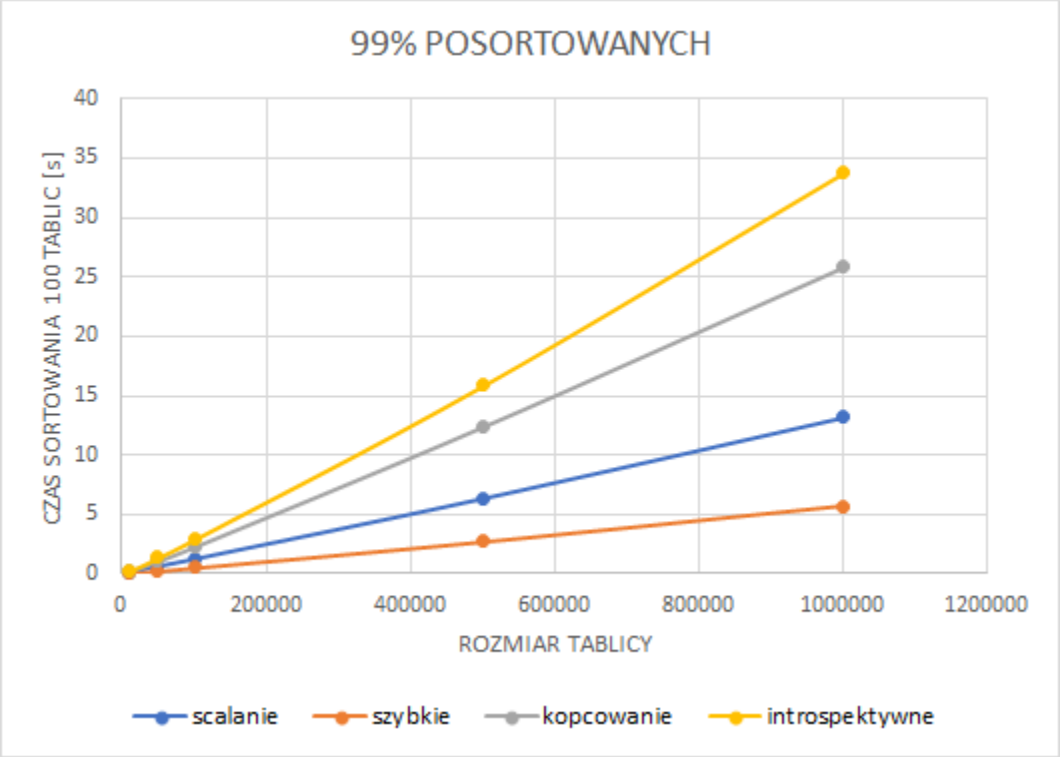
4.7 99% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 8: Czas sortowania dla tablicy w 99% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,099658	0,038951	0,173514	0,223515
50000	0,595107	0,244186	1,063985	1,37681
100000	1,210548	0,518549	2,261138	2,922421
500000	6,328053	2,706129	12,394986	15,865968
1000000	13,192227	5,726093	25,856293	33,801001
2000000	27,390419	12,011761	53,931417	70,652839

Dane pomiarowe umieszczono na wykresie:



Rysunek 10: Czas sortowania tablicy w 99% uporządkowanej w funkcji ilości elementów tablicy

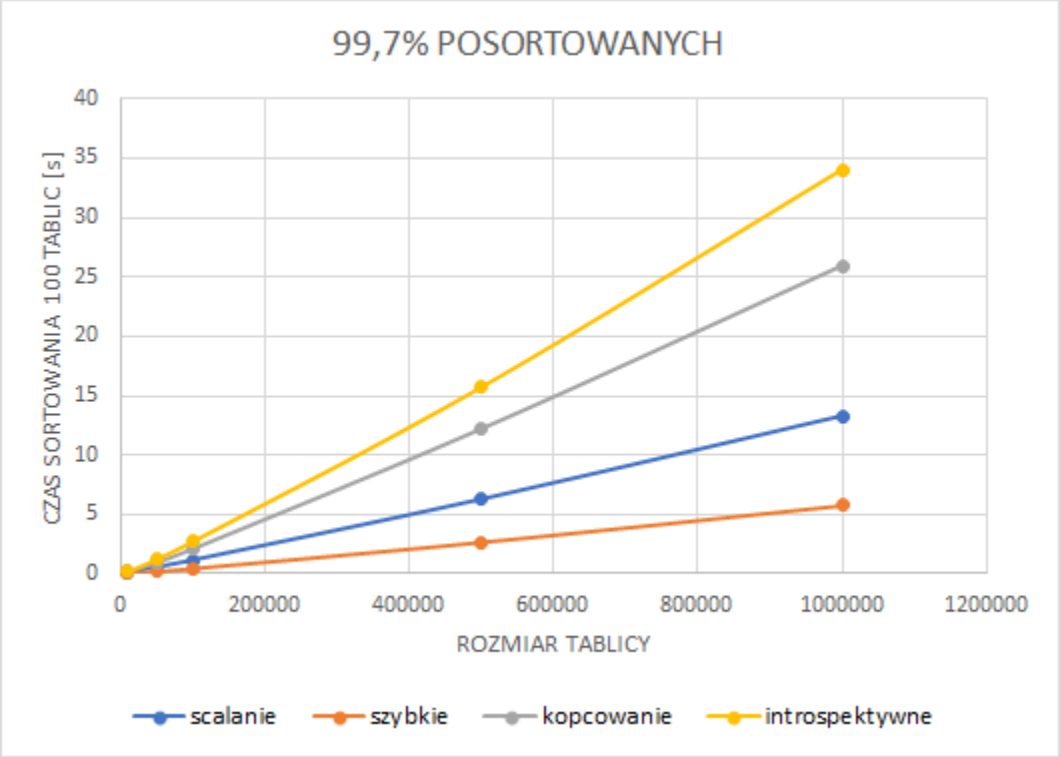
4.8 99,7% początkowych elementów tablicy już posortowanych

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 9: Czas sortowania dla tablicy w 99,7% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,099677	0,039933	0,17423	0,22395
50000	0,535507	0,227868	0,98174	1,275393
100000	1,13092	0,490541	2,130691	2,782036
500000	6,274795	2,685838	12,227825	15,766442
1000000	13,298416	5,751297	25,942735	34,045147
2000000	27,772995	11,903315	54,324998	70,681326

Dane pomiarowe umieszczono na wykresie:



Rysunek 11: Czas sortowania tablicy w 99,7% uporządkowanej w funkcji ilości elementów tablicy

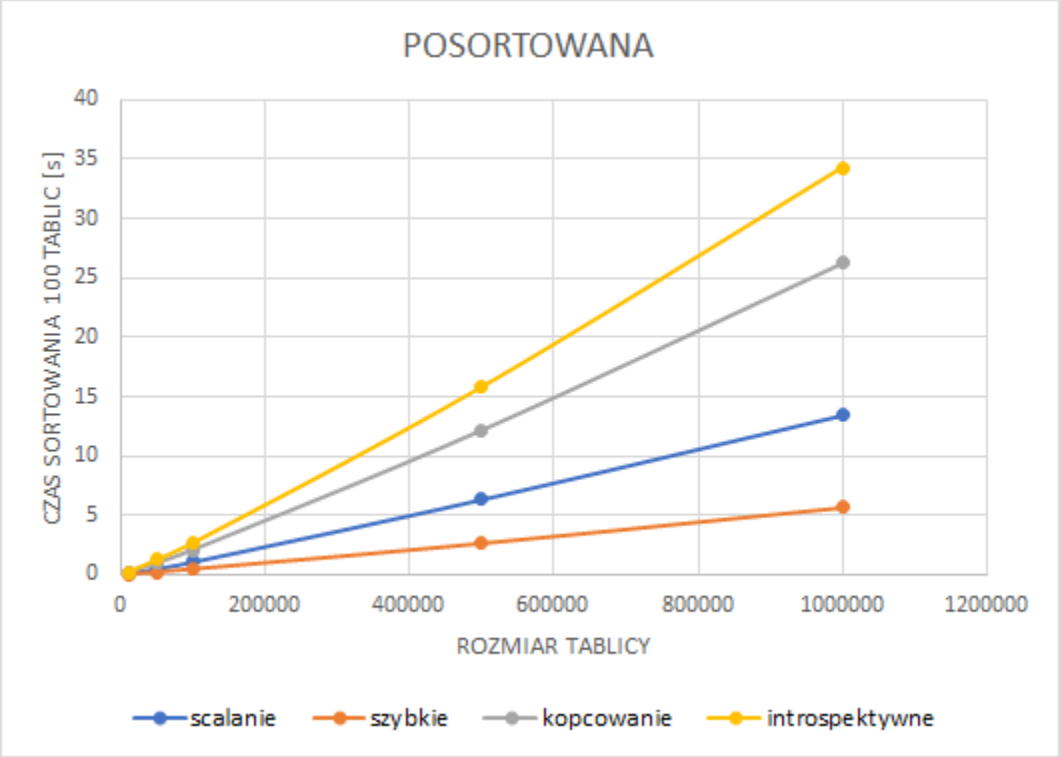
4.9 Wszystkie elementy tablicy już posortowane

Testom poddano 100 różnych tablic. Czas sortowania zbiera tabela poniżej.

Tabela 10: Czas sortowania dla tablicy już uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,105523	0,041461	0,185309	0,236436
50000	0,533644	0,226676	0,98627	1,272711
100000	1,101077	0,475565	2,074698	2,698508
500000	6,357649	2,646082	12,156076	15,798507
1000000	13,453473	5,675297	26,238	34,252998
2000000	29,024933	12,240112	56,913179	74,284274

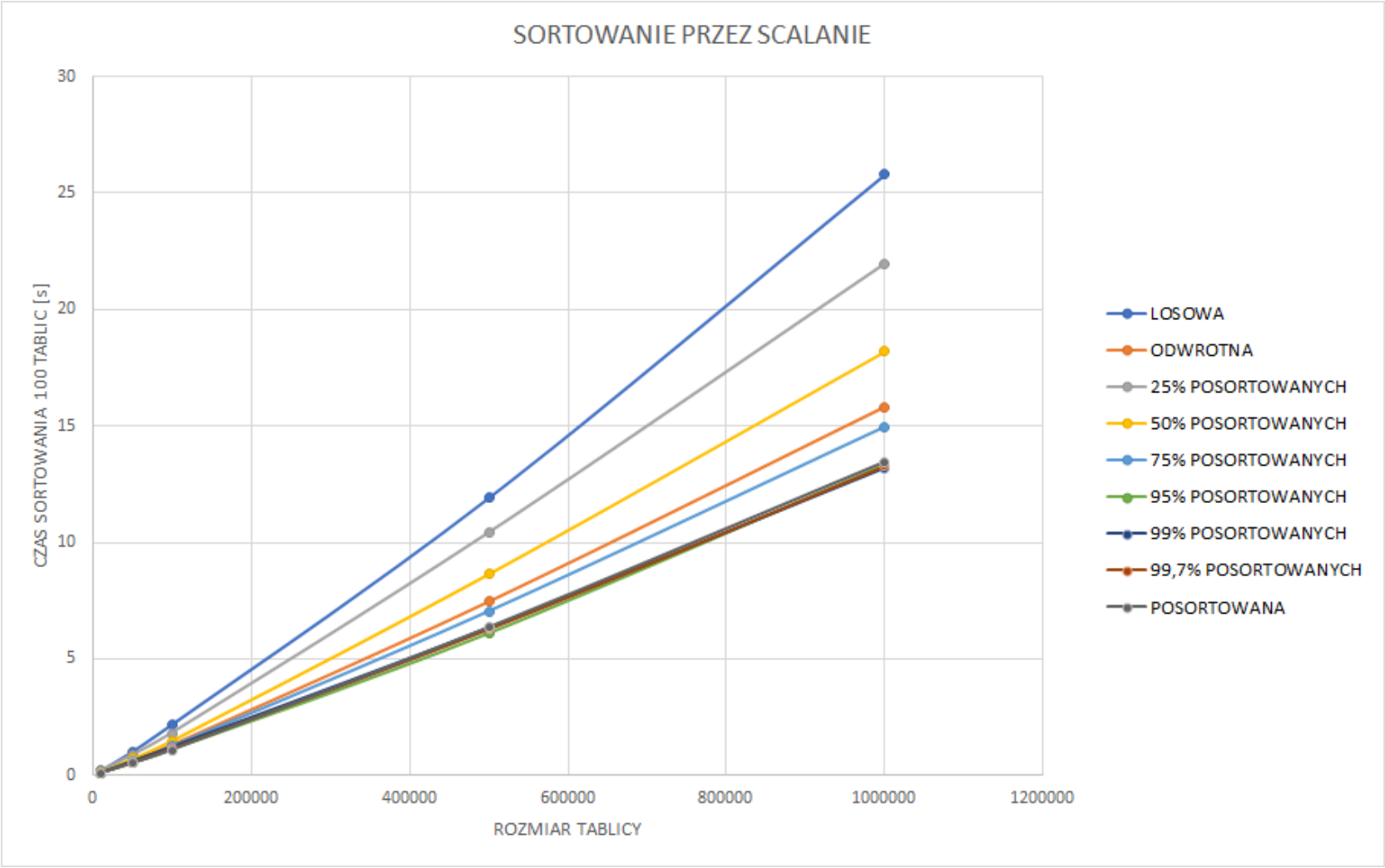
Dane pomiarowe umieszczono na wykresie:



Rysunek 12: Czas sortowania tablicy już uporządkowanej w funkcji ilości elementów tablicy

4.10 Sortowanie przez scalanie

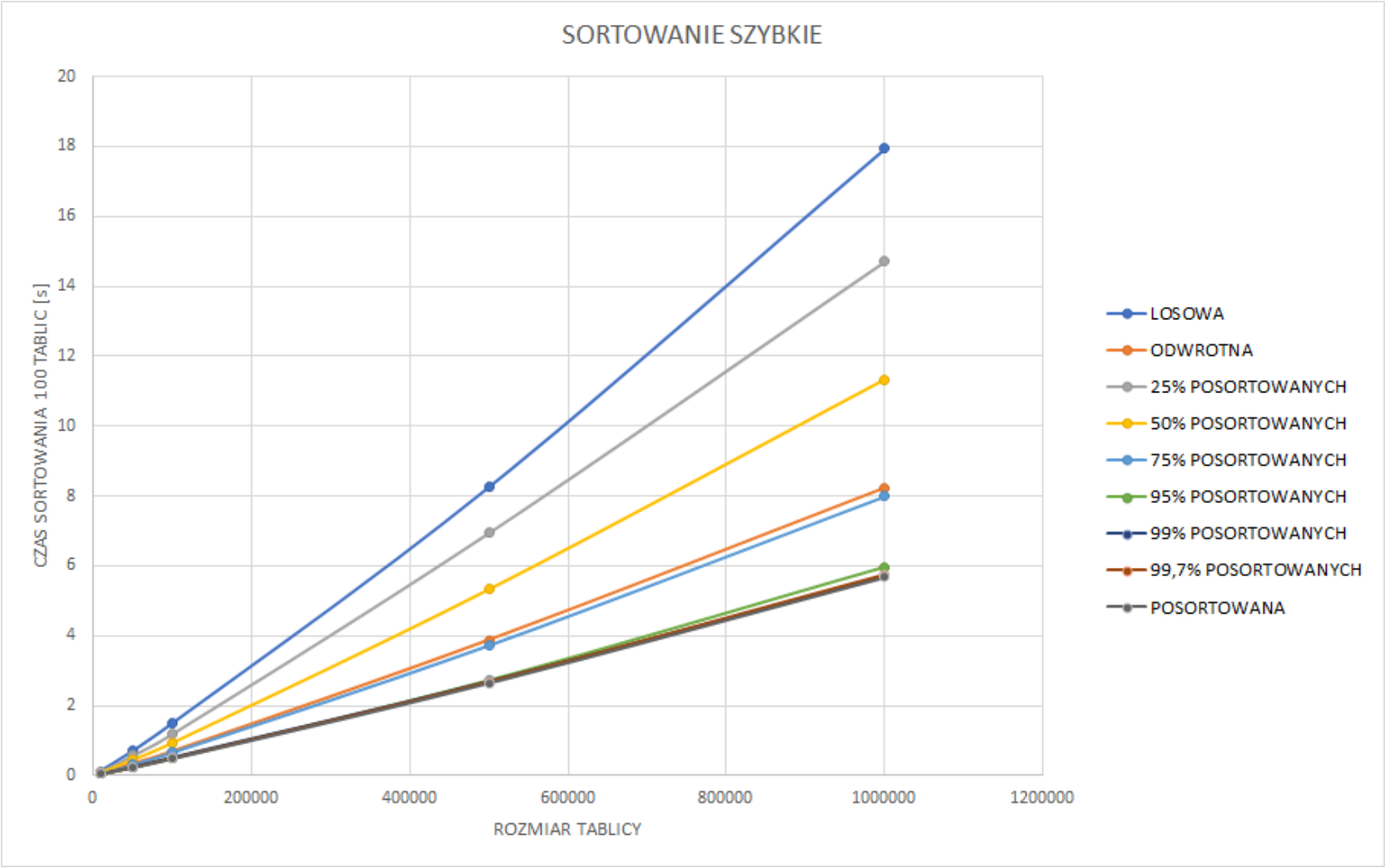
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 13: Sortowanie przez scalanie

4.11 Sortowanie szybkie

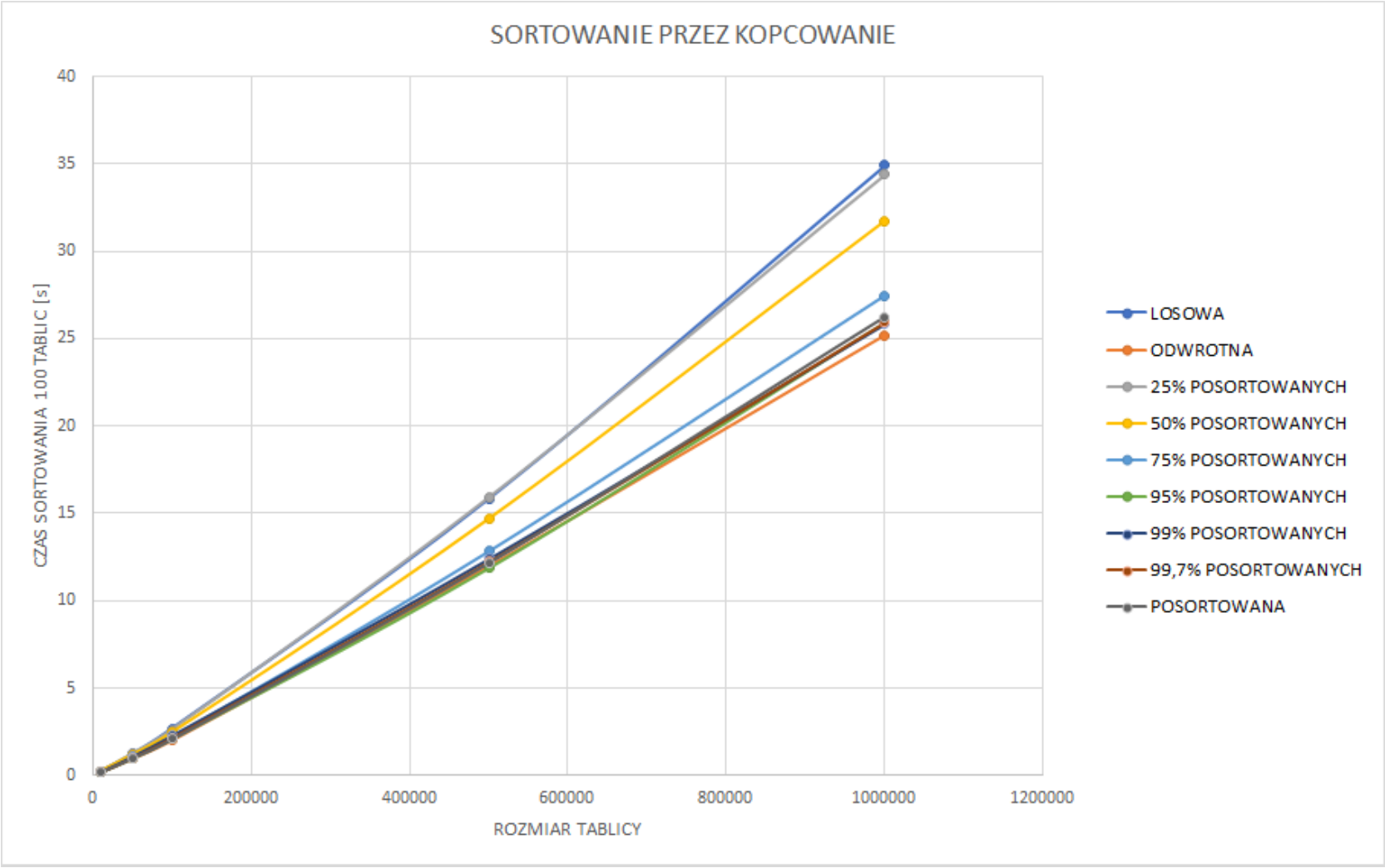
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 14: Sortowanie szybkie

4.12 Sortowanie przez kopcowanie

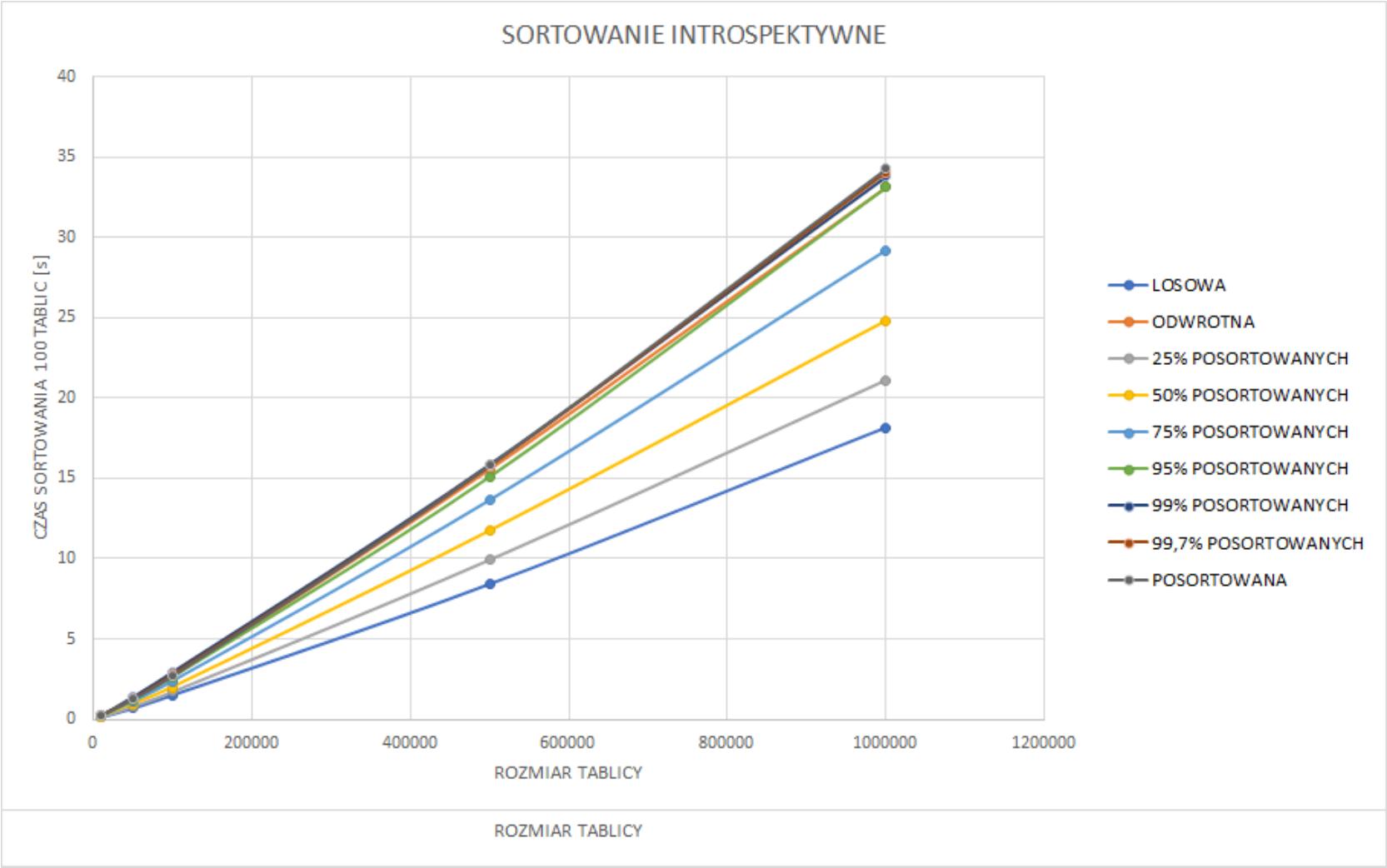
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 15: Sortowanie przez kopcowanie

4.13 Sortowanie introspektywne

Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 16: Sortowanie introspektywne

5 Uwagi i wnioski

Zaimplementowane algorytmy sortowania zostały poddane testom efektywności. Każdy z algorytmów sortowania, dla każdego z 6 rozmiarów tablic oraz w każdym z 9 przypadków ułożenia elementów w tablicy, przetestowano 100 razy. Czas działania tych algorytmów w funkcji ilości elementów umieszczony na kolejnych rysunkach (3 - 12) pozwala wysnuć następujące wnioski:

- W tablicy o elementach ustawionych w losowy sposób najwolniejszy jest algorytm sortowania przez kopcowanie, następnie sortowania przez scalanie, a sortowanie introspektywne i szybkie działają w podobnym czasie.
- Czas działania algorytmów wzrasta szybciej niż funkcja liniowa, ale wolniej niż potęgowa czy wykładnicza, co pozwala przypuszczać, że algorytmy sortowania, zgodnie z tym, co uwzględniono we wstępie teoretycznym, mają złożoność obliczeniową $O(n \cdot logn)$. Ten nieliniowy wzrost czasu w funkcji rozmiaru tablicy do posortowania przedstawia rysunek 4, na którym umieszczono dane pomiarowe zdjęte dla znacznie większych rozmiarów tablic, niż przyjęto w założeniach do badań.
- Gdy elementy posortowane są w odwrotnej kolejności, najwolniejszym okazuje się algorytm sortowania introspektywnego, zaś najszybszym sortowania szybkiego.
- Gdy elementy tablicy są posortowane w 25%, najwięcej czasu potrzebuje algorytm sortowania przez kopcowanie, zaś najmniej - szybkiego. Sortowanie introspektywne i przez scalanie potrzebuje podobnej ilości czasu.
- Dla elementów tablicy posortowanych w 50%, sytuacja szybkości działania algorytmów jest podobna, jak w przypadku elementów posortowanych w 25%, ale czas działania sortowania introspektywnsego i przez scalanie nie są już tak zbliżone jak poprzednio. Gdy tablica jest posortowana w 75% najdłużej sortuje algorytm sortowania introspektywnego, a niewiele krócej sortowania przez kopcowanie. Im bardziej tablica jest uporządkowana, tym różnica w czasie sortowania przez kopcowanie i sortowania introspektywnego jest większa. Dla tablicy już posortowanej (Rysunek 12) różnice w czasie każdego z sortowań są dobrze widoczne. Najszybszym okazuje się algorytm sortowania szybkiego, zaś najwolniejszym algorytm sortowania introspektywnego.
- Wykonane testy pozwalają również zauważyć, że najlepszym algorytmem do sortowania danych jest algorytm sortowania szybkiego. Jeśli przypuszczamy, że dane są ustawione w sposób losowy, dobrym wyborem może okazać się również algorytm sortowania introspektywnego. Z pewnością należy odrzucić sortowanie przez kopcowanie, zaś sortowanie przez wstawianie okazuje się nienajgorszym wyborem.

Dla każdego z 4 sortowań sprawdzono też dla jakich warunków początkowych (sposobu uporządkowania tablicy) radzą sobie najlepiej, a dla jakich najgorzej. I tak:

- **Sortowanie przez scalanie** najgorzej sortuje zestaw danych losowych, a z posortowanymi lub prawie posortowanymi danymi radzi sobie najlepiej (Rysunek 13).
- **Sortowanie szybkie** z danymi radzi sobie podobnie jak sortowanie przez scalanie - losowe dane potrzebują najwięcej czasu, by zostać uporządkowane, zaś dane już uporządkowane zajmują najmniej czasu, jednak sortowanie szybkie działa szybciej niż sortowanie przez scalanie (dla 1 000 000 elementów uporządkowanych losowo sortowanie przez scalanie potrzebuje prawie 22s, zaś sortowanie szybkie niespełna 18s).
- **Sortowanie przez kopcowanie** jest typem sortowania, które działa najdłużej ze wszystkich testowanych w tym projekcie. Sortowanie przez kopcowanie ma podobny czas działania dla tablicy o elementach uporządkowanych losowo, w 25% posortowanych lub w 50% posortowanych, oraz podobny dla tablic o elementach prawie uporządkowanych. Czas ten znacznie odbiega od czasu działania sortowania przez scalania czy sortowania szybkiego, dlatego można uznać, że ten rodzaj sortowania jest najmniej efektywny, choć łatwy do zaimplementowania.
- **Sortowanie introspektywne** najlepiej radzi sobie z danymi uporządkowanymi losowo (w przybliżeniu tak samo szybko, jak sortowanie szybkie), a najgorzej z danymi uporządkowanymi w kolejności odwrotnej. Dla dużej ilości danych posortowanie już przesortowanej tablicy zajmuje mu najwięcej czasu.

Podczas przeprowadzenia każdego z testów została wykorzystana funkcja sprawdzająca poprawność posortowania tablicy. W każdym przypadku funkcja ta nie zakończyła działania programu, co pozwala zauważyć, że sortowania działają prawidłowo.

Przy tworzeniu algorytmów sortowań korzystano z zasobów dostępnych w Internecie:

$$http://slideplayer.pl/slide/8941861/$$

$$http://eduinf.waw.pl/inf/alg/003_sort/index.php$$

$$http://www.algorytm.org/algorytmy-sortowania/$$

$$http://miroslawzelent.pl/kurs-c++/sortowanie-zlozonosc-algorytmow/$$

$$https://www.youtube.com/user/kolboch$$

$$https://pl.wikipedia.org/wiki/Sortowanie_introspektywne$$

$$https://www.szkolnictwo.pl/szukaj,Sortowanie_introspektywne$$

$$https://en.wikipedia.org/wiki/Introsort$$

$$https://programmingpraxis.com/2016/11/11/introspective-sort/$$

$$https://secweb.cs.odu.edu/\ zeil/cs361/web/website/Lectures/heapsort/pages/introspect.html$$

Kod programu udostępny pod adresem:

$$https://github.com/234780/PAMSI2$$