PAMSI - projekt 2
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.

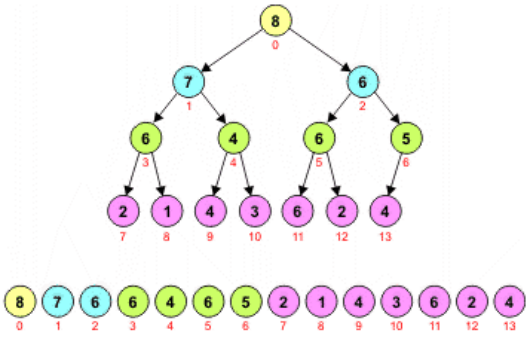
1.3.1 Sortowanie przez wstawianie

Sortowanie przez wstawianie bazuje na podziale tablicy na część uporządkowaną i nieporządkowaną. W kolejnych iteracjach pobiera pierwszy element z części nieuporządkowanej i wstawia go w odpowiednie miejsce w części uporządkowanej.

1.3.2 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.

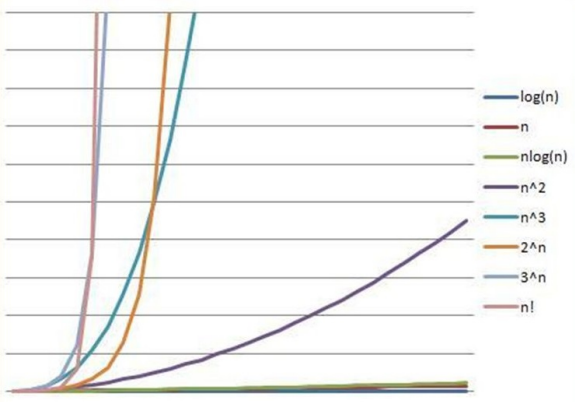
2 Złożoność obliczeniowa

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	$O(n \cdot log_2 n)$		$O(1)$	
przez wstawianie	$O(n^2)$		$O(1)$	
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). Ich opis i rezultaty przedstawiono poniżej.

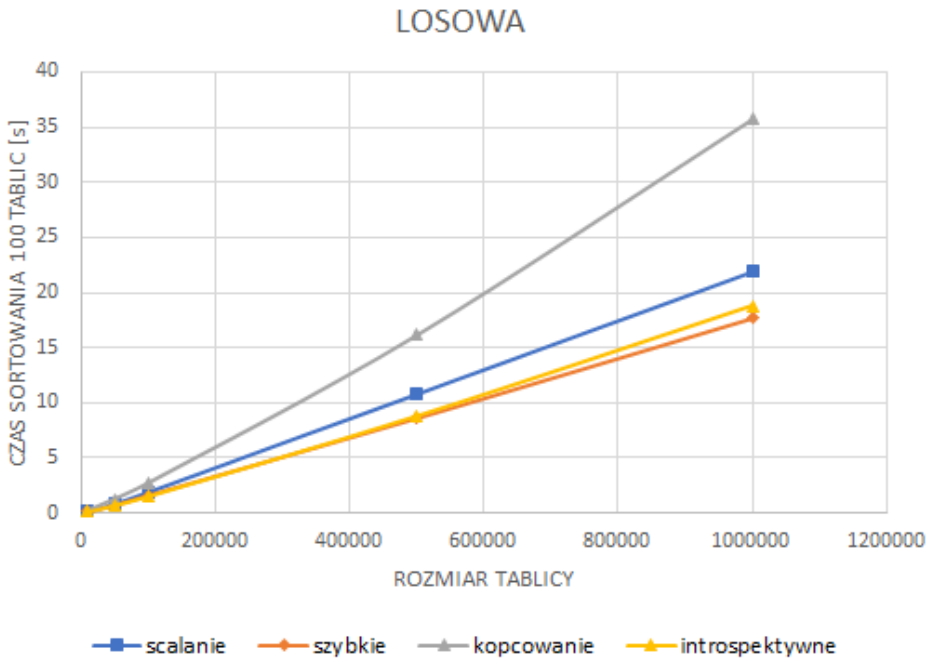
4.1 Elementy tablicy losowe

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	0,160109	0,116477	0,206111	0,117118
50000	0,890092	0,740222	1,260473	0,701058
100000	1,960839	1,623890	2,704142	1,527547
500000	10,846934	8,643381	16,135495	8,789630
1000000	21,940353	17,725443	35,759193	18,757495
3000000	89,083506	68,520545	166,247622	68,299367
5000000	167,175873	112,413806	283,811054	111,517907
7000000	234,538773	160,659968	412,563296	167,736351
10000000	313,232333	225,800502	584,440469	258,044859
15000000	495,900040	343,499286	1067,477198	337,478329
20000000	610,576420	479,908068	1393,137623	496,459270

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

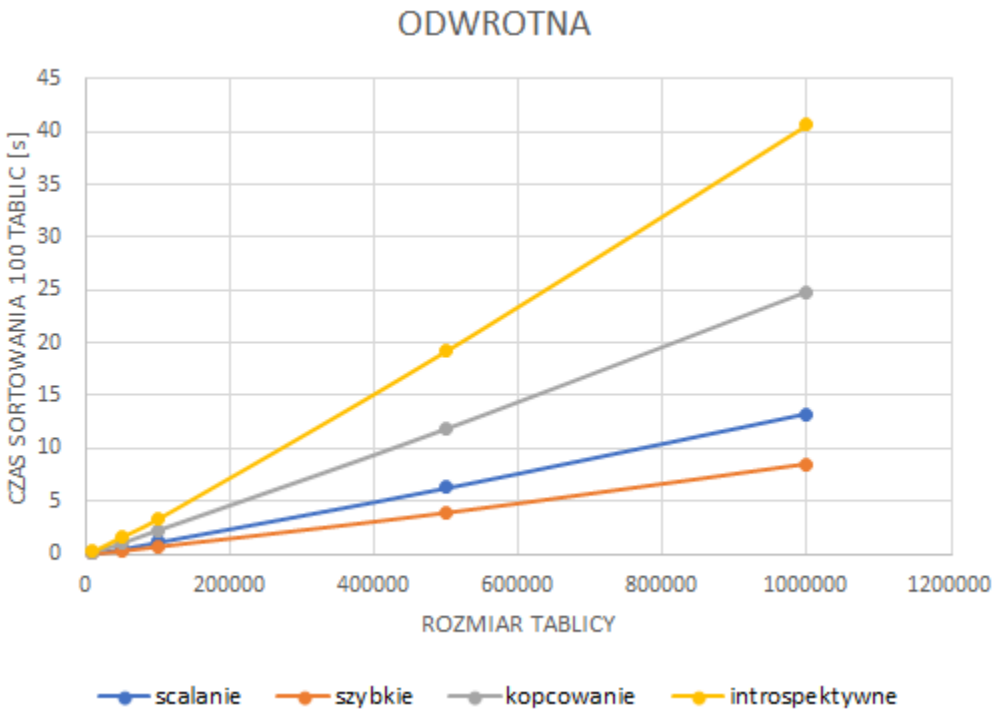
4.2 Wszystkie elementy tablicy już posortowane, ale w odwrotnej kolejności

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

Tabela 3: Czas sortowania dla tablicy w 25% uporządkowanej

	Czas sortowania 100 tablic [s]			
ilość elementów	przez scalanie	szybkie	przez kopcowanie	introspektywne
10000	0,099274	0,057484	0,179632	0,278989
50000	0,531506	0,330582	0,992377	1,685091
100000	1,170564	0,727962	2,216784	3,375426
500000	6,282431	3,903441	11,834984	19,274481
1000000	13,268344	8,470276	24,814866	40,621725

Dane pomiarowe umieszczono na wykresie:



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

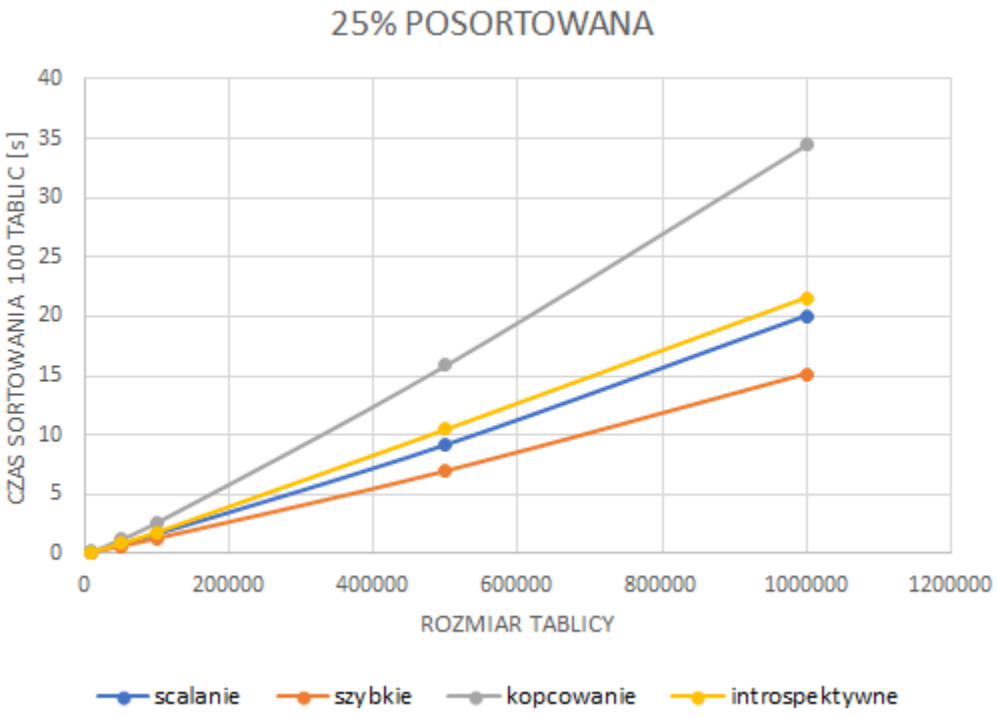
4.3 25% początkowych elementów tablicy już posortowanych

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	0,140971	0,097112	0,204581	0,142370
50000	0,762387	0,584171	1,243933	0,848392
100000	1,661725	1,249574	2,659417	1,817497
500000	9,147196	6,931092	15,873583	10,470132
1000000	19,994742	15,098572	34,426922	21,566960

Dane pomiarowe umieszczono na wykresie:



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

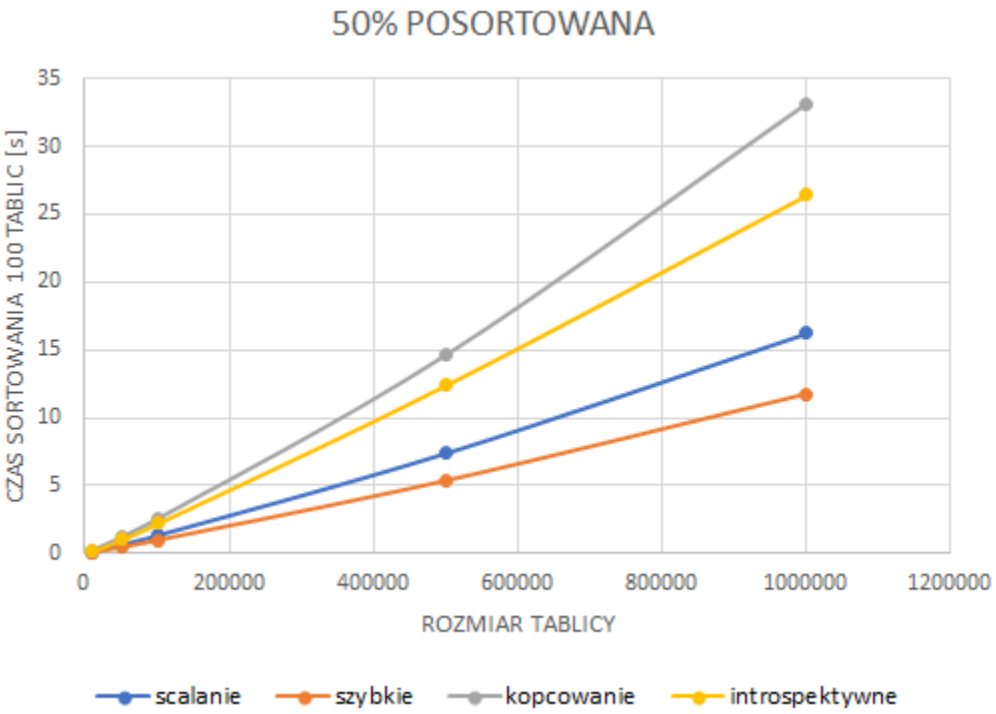
4.4 50% początkowych elementów tablicy już posortowanych

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	0,113840	0,072629	0,194691	0,165267
50000	0,643427	0,437765	1,160623	1,027943
100000	1,340807	0,951089	2,508672	2,224565
500000	7,420024	5,355796	14,616284	12,439745
1000000	16,265737	11,737463	33,158205	26,445331

Dane pomiarowe umieszczono na wykresie:



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

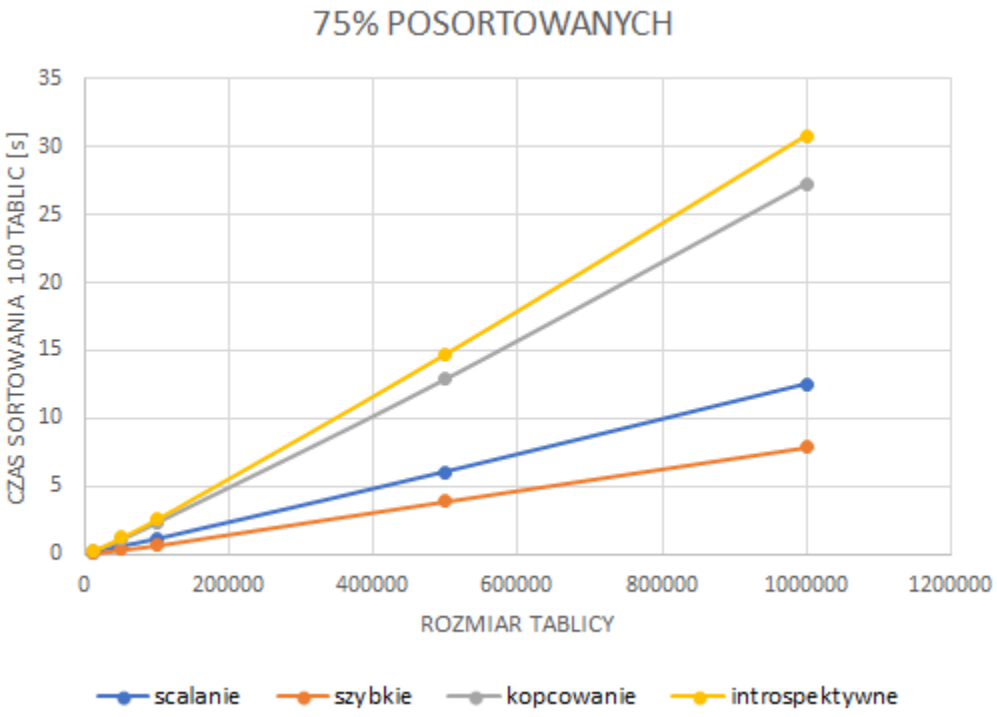
4.5 75% początkowych elementów tablicy już posortowanych

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	0,092453	0,051764	0,181627	0,201422
50000	0,534918	0,326709	1,065819	1,195859
100000	1,078979	0,655333	2,330272	2,583899
500000	6,044690	3,879468	12,879121	14,747678
1000000	12,533946	7,843819	27,239008	30,825509

Dane pomiarowe umieszczono na wykresie:



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

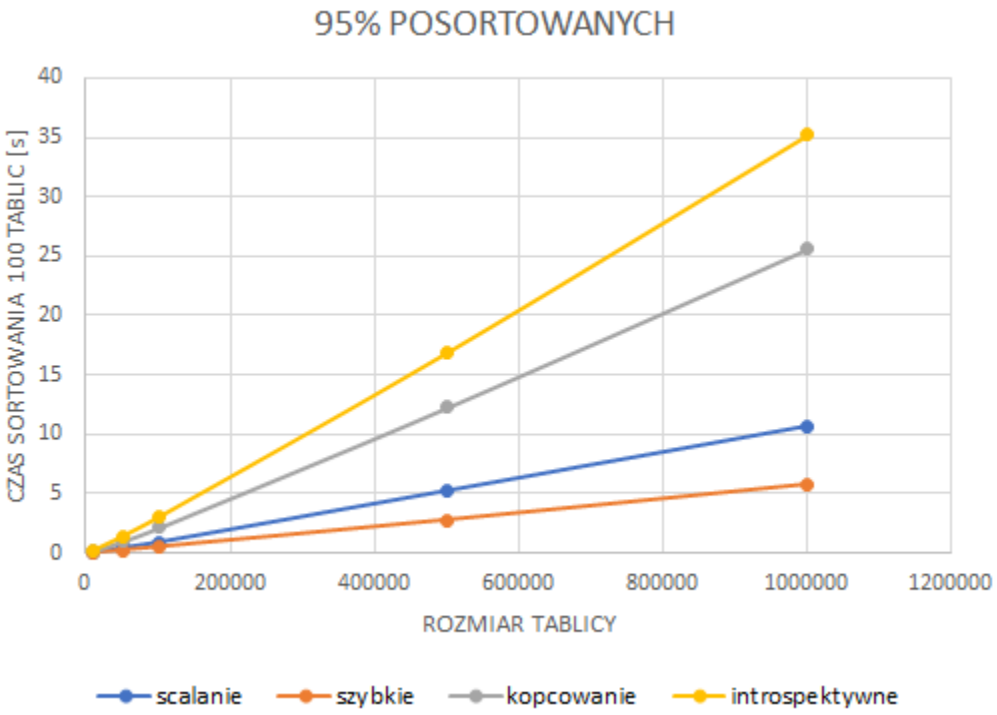
4.6 95% początkowych elementów tablicy już posortowanych

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	0,081162	0,039447	0,175631	0,239206
50000	0,468506	0,244602	1,031795	1,419647
100000	0,935499	0,514403	2,154280	3,023487
500000	5,273486	2,778816	12,269531	16,891187
1000000	10,714940	5,756867	25,597844	35,237640

Dane pomiarowe umieszczono na wykresie:



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

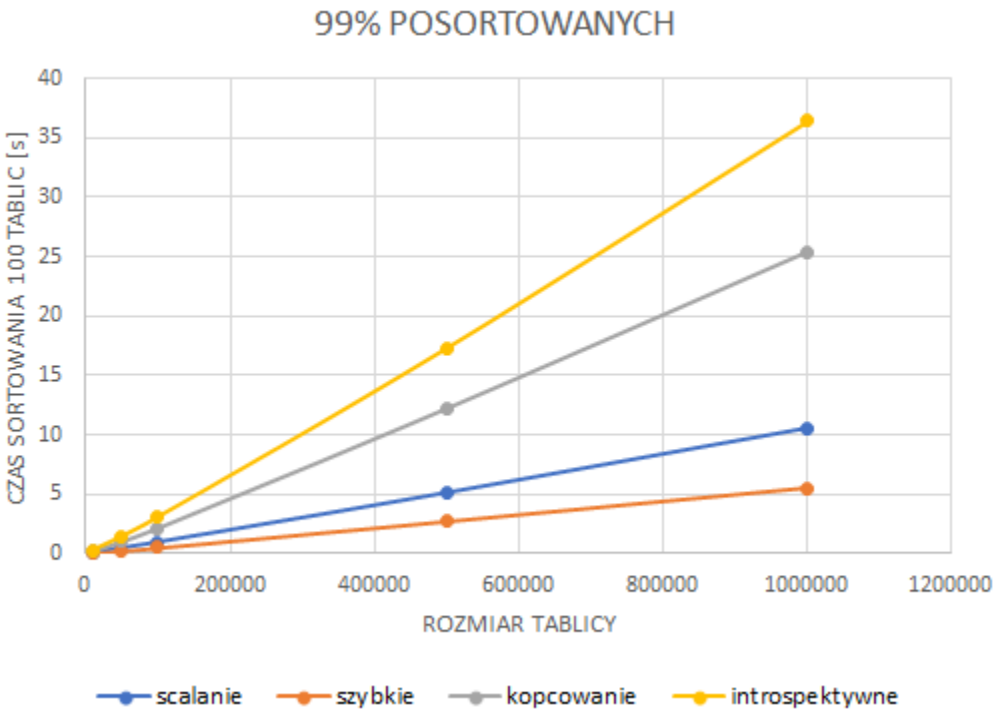
4.7 99% początkowych elementów tablicy już posortowanych

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	0,079537	0,038925	0,182962	0,261305
50000	0,451121	0,247698	1,014335	1,403895
100000	0,931138	0,506071	2,144270	3,020985
500000	5,117239	2,720055	12,218841	17,290181
1000000	10,589665	5,511627	25,396966	36,440969

Dane pomiarowe umieszczono na wykresie:



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

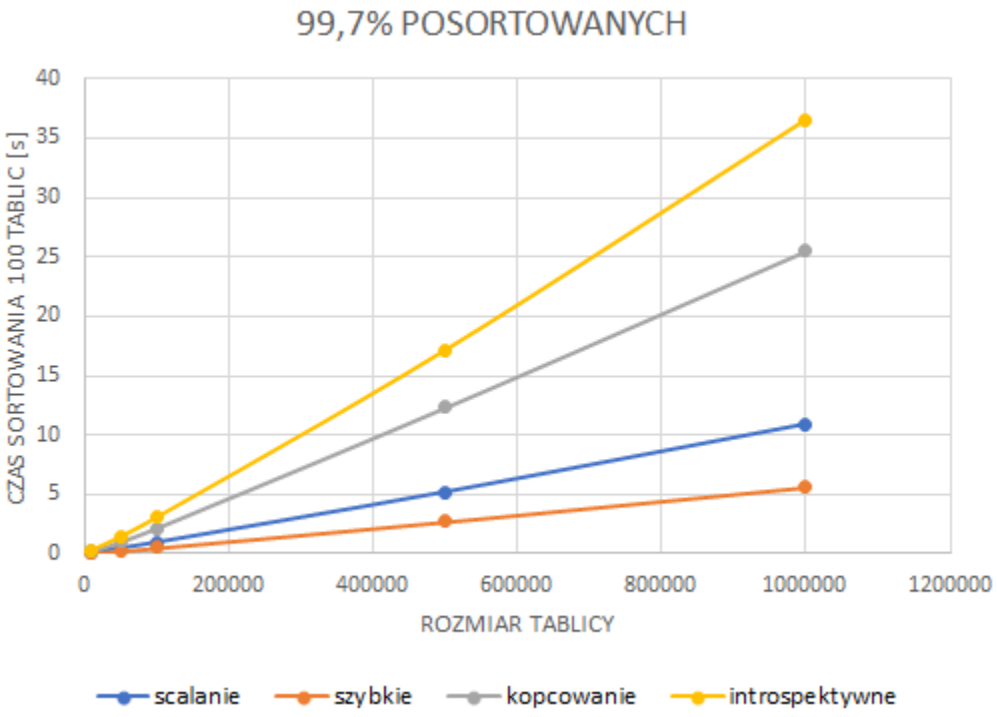
4.8 99,7% początkowych elementów tablicy już posortowanych

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	0,089647	0,041428	0,188034	0,258224
50000	0,450564	0,237690	1,017690	1,406239
100000	0,942336	0,500516	2,171213	3,035942
500000	5,175528	2,712100	12,311614	17,131137
1000000	10,875508	5,620394	25,484338	36,544896

Dane pomiarowe umieszczono na wykresie:



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

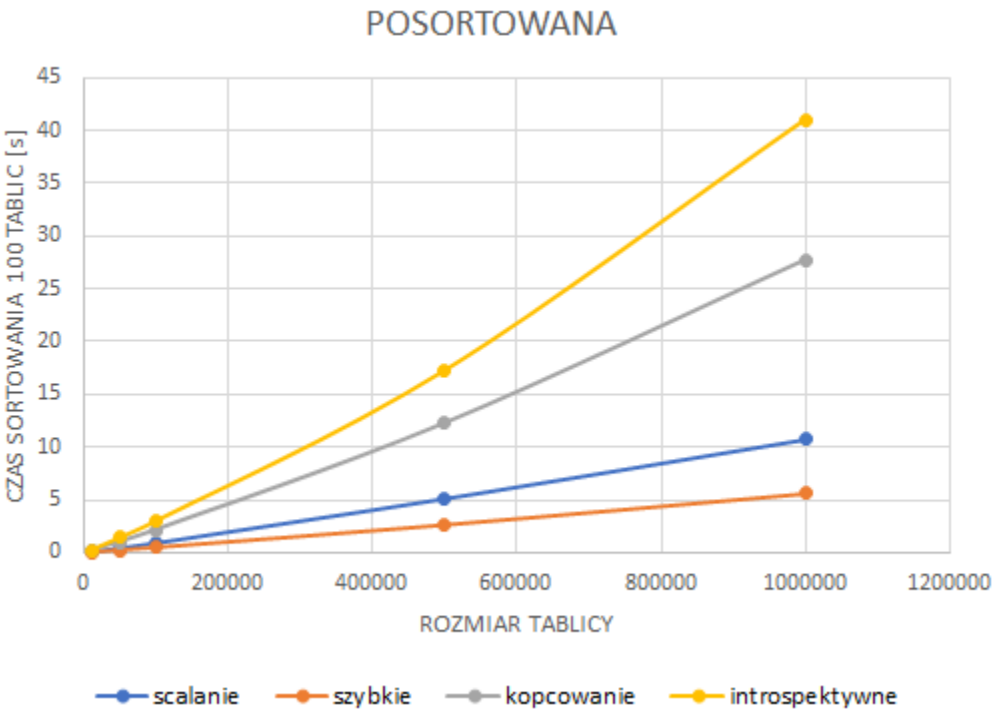
4.9 Wszystkie elementy tablicy już posortowane

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	0,079627	0,038154	0,177017	0,243080
50000	0,440682	0,248869	1,022846	1,442521
100000	0,935974	0,485194	2,152986	3,004170
500000	5,141887	2,632315	12,258966	17,234531
1000000	10,759708	5,606563	27,761565	40,969791

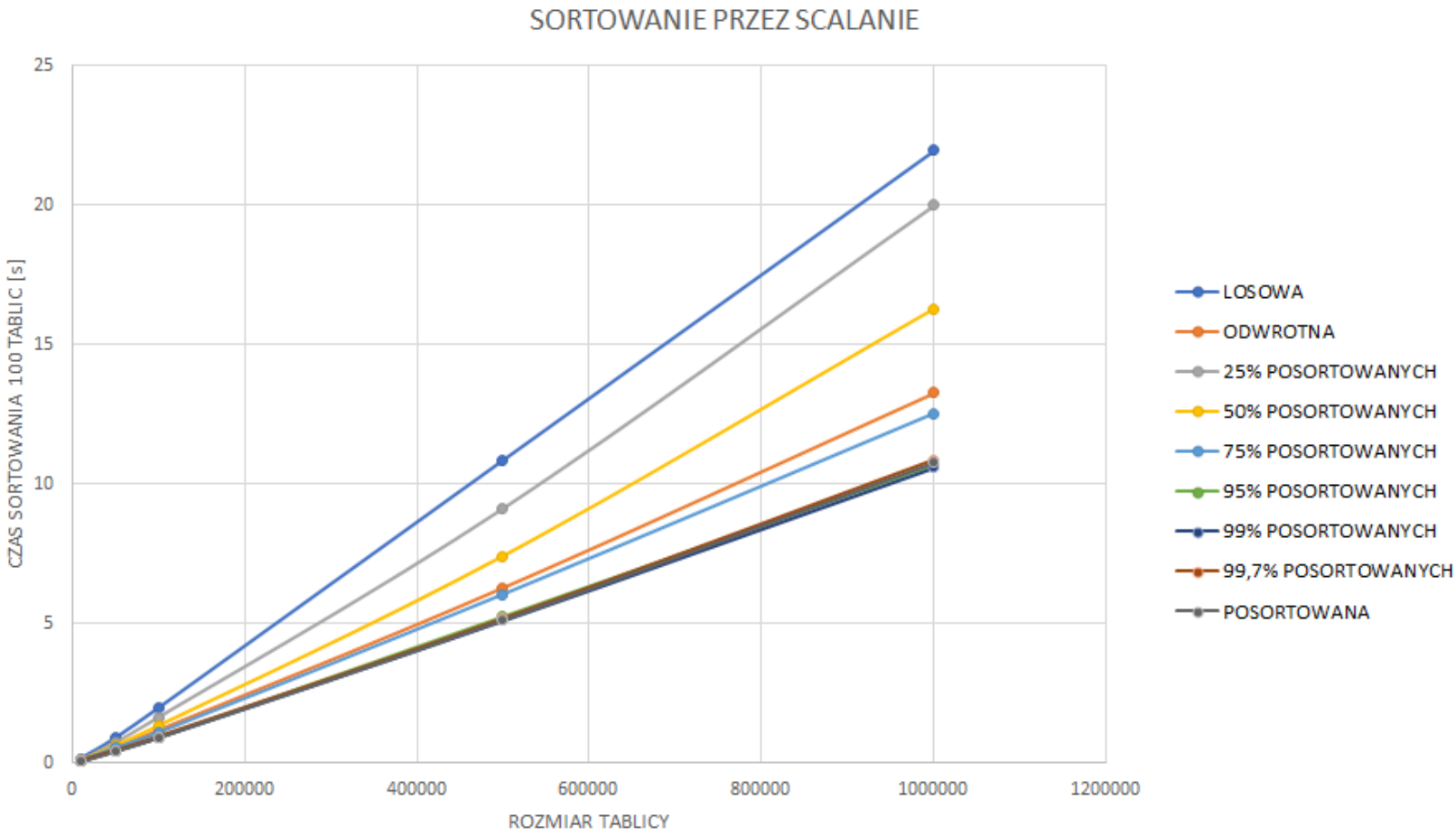
Dane pomiarowe umieszczono na wykresie:



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

4.10 Sortowanie przez scalanie

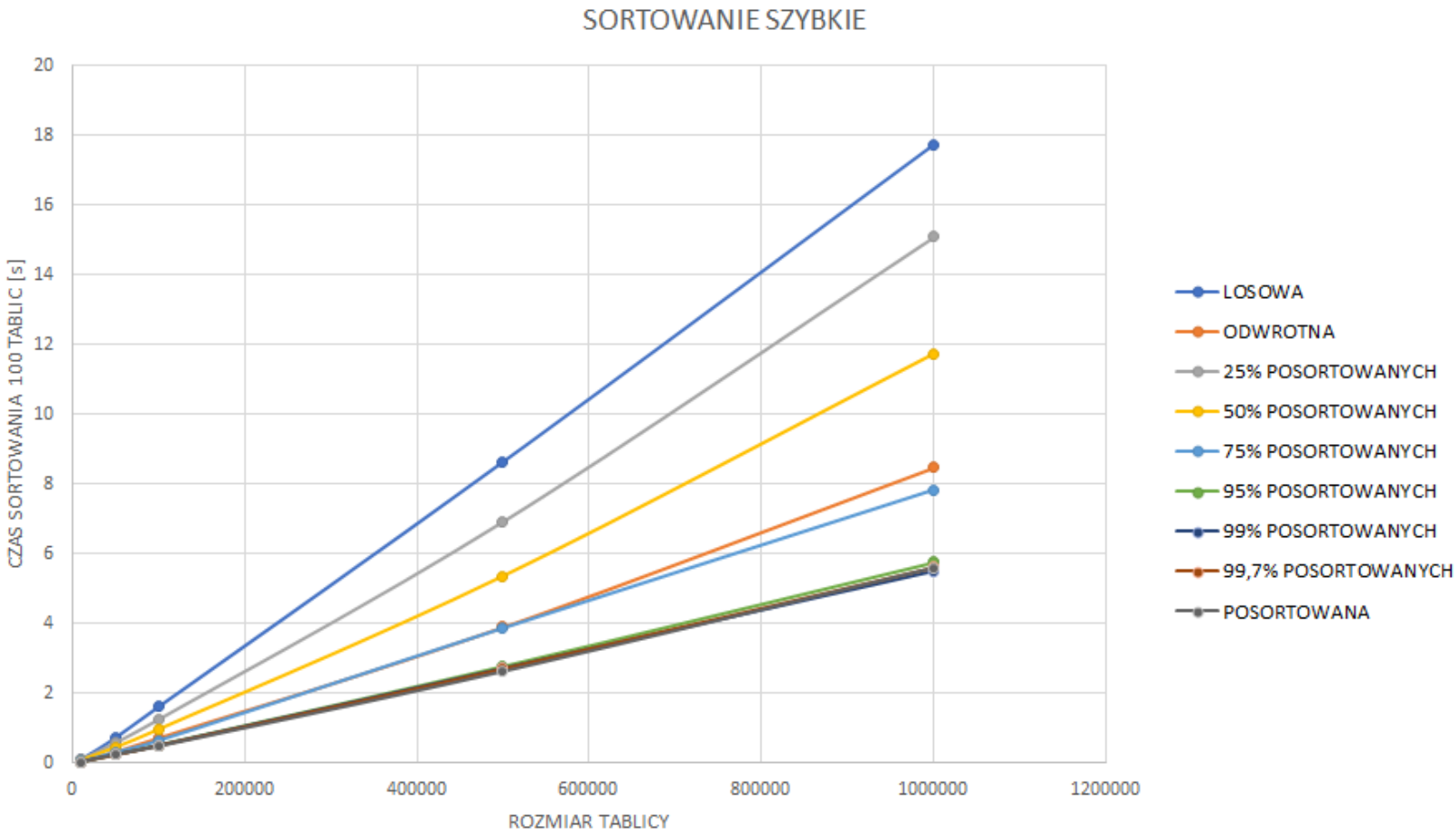
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 13: Sortowanie przez scalanie

4.11 Sortowanie szybkie

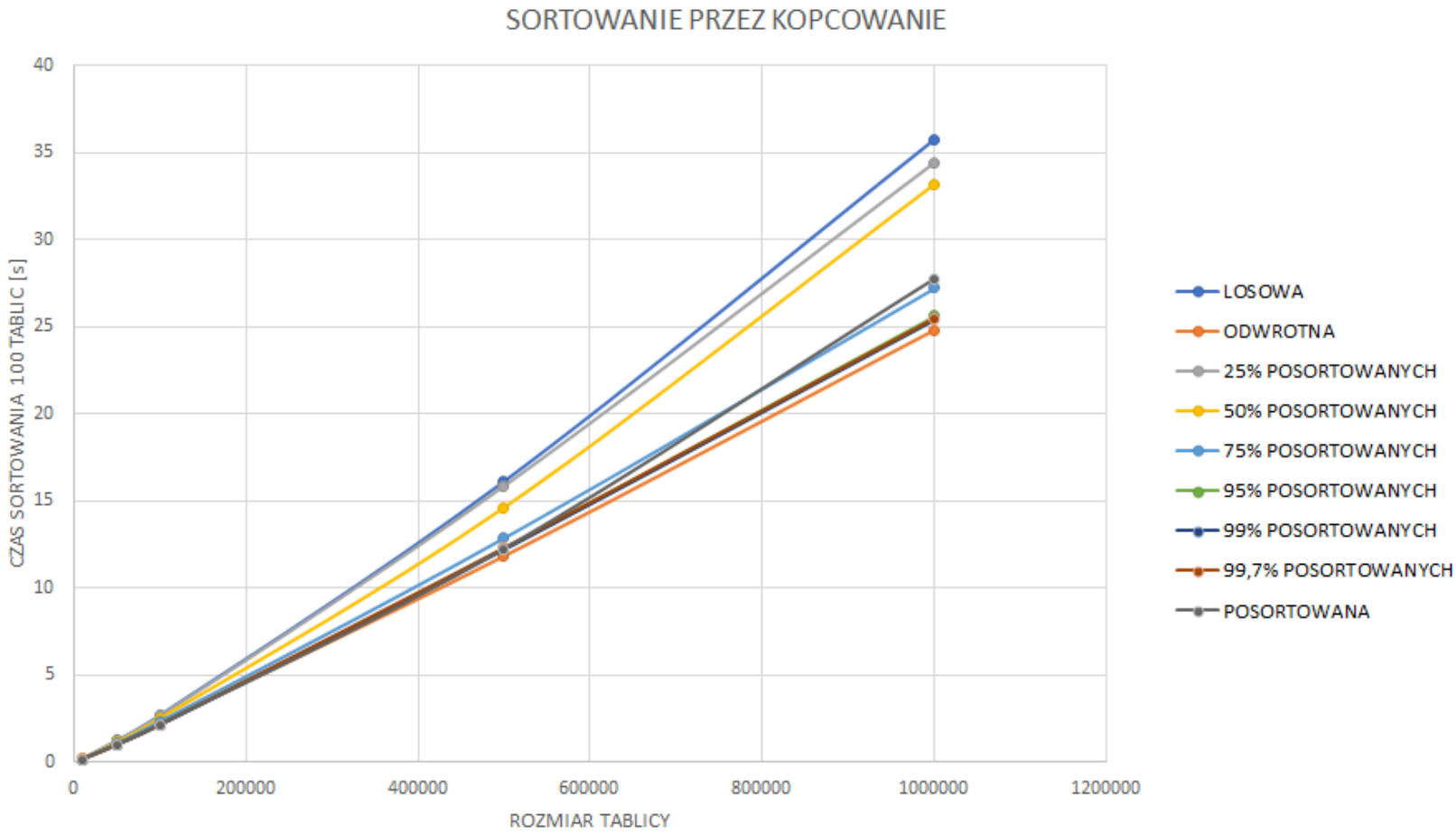
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



Rysunek 14: Sortowanie szybkie

4.12 Sortowanie przez kopcowanie

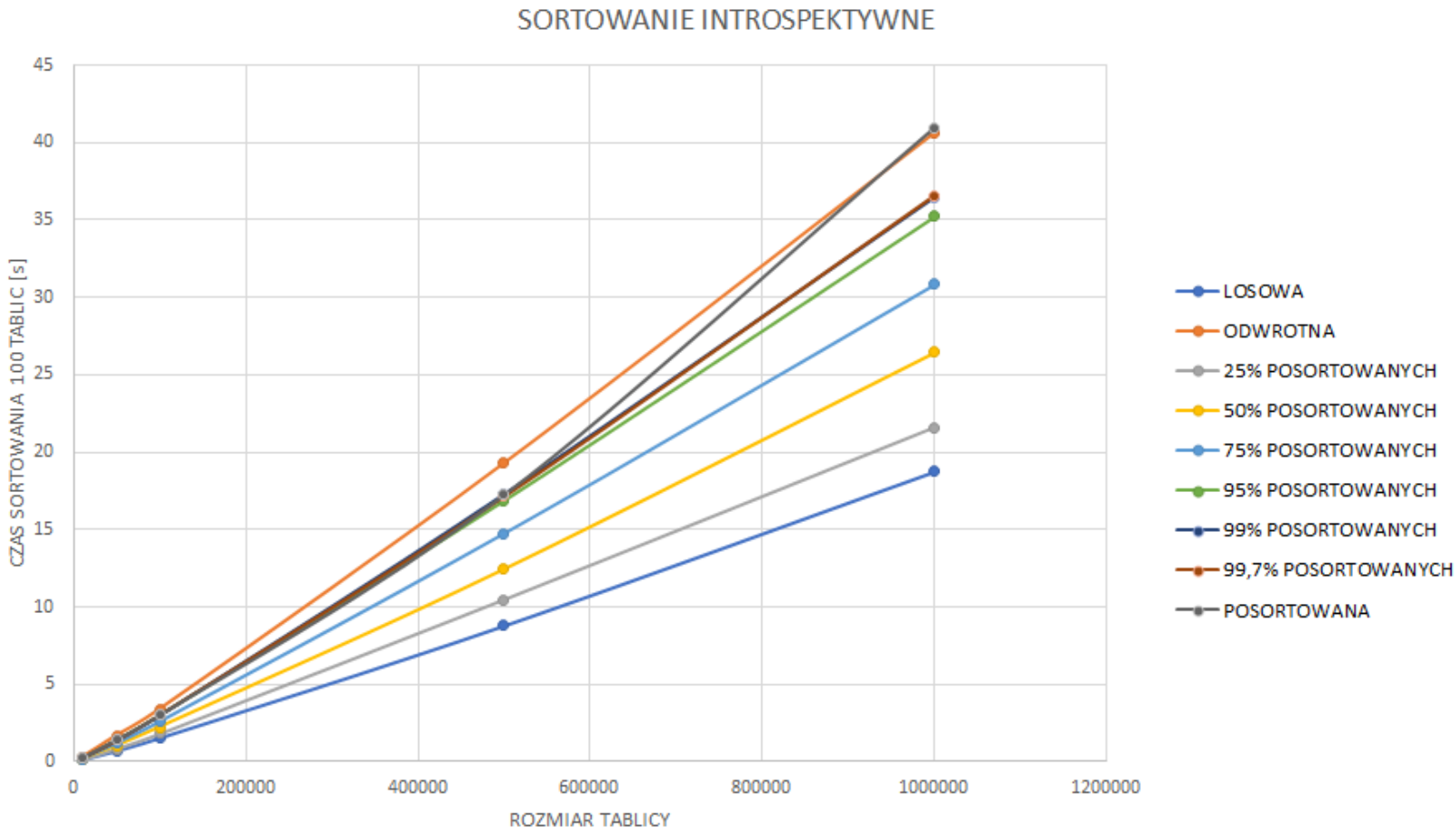
Na poniższym wykresie przedstawiono czas sortowania odpowiedniej tablicy w funkcji ilości elementów do posortowania



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 5 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 badaniach.
- Gdy elementy posortowane są w odwrotnej kolejności, najwolniejszym okazuje się algorytm sortowania introspektywnego, zaś najszybszym sortowania szybkiego.
- Gdy elementy tablicy są posortowane w 25 lub 50%, sytuacja szybkości działania algorytmów jest podobna, jak w przypadku elementów posortowanych w kolejności odwrotnej. 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:

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

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

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

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

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

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

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

Kod programu udostępny pod adresem:

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