Politechnika Wrocławska
Wydział Elektroniki
Kierunek: AiR

Rok akademicki
2019/2020
semestr letni

Sprawozdanie z projektu 1

Kurs:	Projektowanie algorytmów i metody sztucznej inteligencji
Prowadzący:	Dr inż. Łukasz Jeleń
Wykonał:	Taras Radchenko (nr indeksu 248021)
Termin zajęć:	czwartek 9:15

Wrocław
29.03.2020

Wprowadzenie

W dzisiejszych czasach człowieczeństwo generuje i zużywa ogromne ilości informacji, więc dla projektowania oprogramowania istotna jest umiejętność badania złożoności obliczeniowej stosowanych algorytmów. W tym projekcie badamy złożoność obliczeniową trzech algorytmów sortowania.

Badane algorytmy

Sortowanie przez scalanie

To sortowanie bazuje się na fakcie, że tablica o rozmiarze 1 jest tablicą posortowaną. Tak więc potrzebny jest algorytm, który potrafi scalić dwie tablice posortowane tak, aby otrzymać tablicę posortowaną. Mając taki algorytm, stosujemy go najpierw na tablicach jednoelementowych powstałych z „rozbicia" tablicy wejściowej. Tablicy wynikające ze scalań scalamy ponownie aż nie dostaniemy jedną tablicę, która będzie wynikiem sortowania. Zaletą algorytmu jest łatwość zapisania w postaci iteracyjnej.

Zakładając, że rozmiar tablicy jest potęgą dwójki ($n = 2^k$), jest łatwo policzyć złożoność obliczeniową tego algorytmu. Każde scalanie potrzebuje maksymalnie $2x - 1$ porównań, gdzie x jest liczbą elementów w każdej ze scalanych tablic (np. do scalenia tablic jednoelementowych potrzebujemy $2 \cdot 1 - 1 = 1$ porównań, dwuelementowych $- 2 \cdot 2 - 1 = 3$ porównań). Całkowita liczba potrzebnych operacji jest sumą iloczynów liczby scalań tablic x-elementowych i liczby potrzebnych do tego operacji:

$$\frac{2^k}{2}(2 \cdot 1 - 1) + \frac{2^k}{4}(2 \cdot 2 - 1) + \ldots + \frac{2^k}{2^k}(2 \cdot 2^{k-1} - 1) =$$
$$2^{k-1}(2^1 - 1) + 2^{k-2}(2^2 - 1) + \ldots + 2^0(2^k - 1) =$$
$$2^k - 2^{k-1} + 2^k - 2^{k-2} + \ldots + 2^k - 2^0 = k2^k - (2^{k-1} + 2^{k-2} + \ldots + 2^0) = k2^k - (2^k - 1) =$$
$$(k-1)2^k + 1 = (log_2 n - 1)n + 1 = nlog_2 n - n + 1$$

$$O(g(n)) = O(nlog_2 n)$$

Wynik wyprowadzenia zgadza się ze złożonościami obliczeniowymi podanymi w Wikipedii: w przypadku średnim zarówno jak i w najgorszym złożoność ta wynosi $O(nlog_2 n)$.

Sortowanie szybkie

Polega na rekurencyjnym stosowaniu algorytmu, który wybiera element rozdzielający (dalej ER) i przenosi elementy tablicy większe od wybranego do pierwszego fragmentu i mniejsze do drugiego, po czym przenosi ER do środka między fragmentami. Elementy równe mogą być w dowolnym fragmencie.

Dla tablicy n-elementowej na poziomie rekurencji 1 jest $n - 1$ elementów do porównania, bo jeden z elementów został wybrany jako ER. Na poziomie 2 jest $n - 3$ elementów do porównania, bo jeden z elementów jest ER na poziomie 1 oraz jeszcze 2 są ER na poziomie 2. Na kolejnych poziomach będzie $n - 7$, $n - 15$, … Ale można od razu pominąć odjęcie stałej gdyż w notacji dużego O bierzemy pod uwagę człon największego rzędu. Więc zakładamy, że na każdym poziomie jest n elementów do porównania. Maksymalna głębokość rekurencji zależy od metody wyboru ER i w najlepszym przypadku wynosi $log_2 n$ (wtedy czas obliczeń jest w $O(nlog_2 n)$), w najgorszym $- n$ (wtedy czas obliczeń jest w $O(n^2)$). Te wyniki zgadzają się ze złożonościami czasowymi podanymi w Wikipedii.

Sortowanie introspektywne

Sortowanie introspektywne jest sortowaniem hybrydowym. Ten algorytm stosuje sortowanie szybkie dopóki głębokość rekurencji nie osiągnie $2log_2n$, po czym stosuje sortowanie przez kopcowanie. W ten sposób nie trafiamy w przypadek najgorszy sortowania szybkiego.

O ile złożoność obliczeniowa sortowania szybkiego jest w $O(nx)$, gdzie x jest maksymalną głębokością rekurencji, ograniczając x do $2log_2n$ zostajemy w czasie $O(nlog_2n)$. Złożoność obliczeniowa sortowania przez kopcowanie również wynosi $O(nlog_2n)$ w przypadku średnim zarówno jak i w najgorszym, więc złożoność sortowania introspektywnego w obu przypadkach wynosi $O(nlog_2n)$.

Przebieg eksperymentu

Z grubsza sposób prowadzenia testu jest przedstawiony w następującym pseudokodzie:

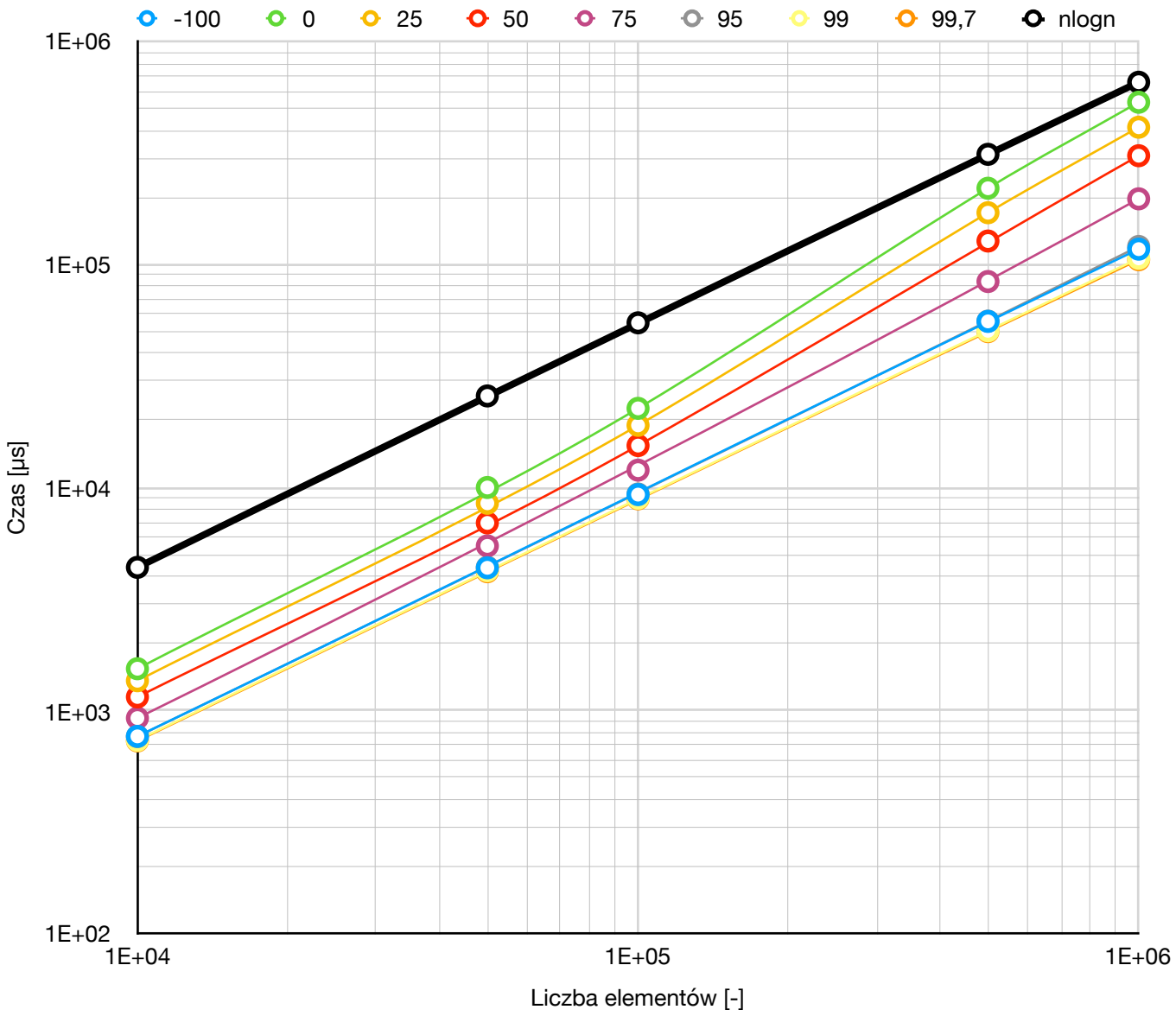
```
Algorytm testujAlgorytm(algorytm, tablica)
    Opis: mierzy czas sortowania algorytmem algorytm na tablicy tablica
    Wyjście: zmierzony czas w nanosekundach

    początek ← zegar_systemowy.czas()
    algorytm(tablica)
    koniec ← zegar_systemowy.czas()

    return (koniec - początek).nanosekundy()


Algorytm testuj(plikCSV, katalogTablic)
    Opis: mierzy czas działania każdego algorytmu na tablicach z katalogu
          katalogTablic i zapisuje wyniki do pustego pliku plikCSV
    Wyjście: nic

    algorytmy ← [
        sortowaniePrzezScalanie, sortowanieSzybkie, sortowanieIntrospektywne]

    plikCSV.zapiszLinię(
        "Liczba elementów, Stopień sortowania, Czas SPS, Czas SS, Czas SI")

    for tablica: katalogTablic do
        plikCSV.zapiszWPole(tablica.rozmiar())
        plikCSV.zapiszWPole(tablica.stopieńSortowania())

        for algorytm: algorytmy do
            plikCSV.zapiszWPole(testujAlgorytm(algorytm, tablica.kopia()))

        plikCSV.zapiszLinię()
```

Do pomiarów czasu jest wykorzystywany `std::chrono::steady_clock`, który zapewnia największą dokładność.

Wyniki

Sortowanie przez scalanie

Tu i dalej „-100” oznacza wynik testu na tablicy posortowanej w odwrotnej kolejności, „nlogn” oznacza funkcję $k \cdot nlog_2n$, pozostałe oznaczają liczbę elementów na początku już posortowanych (w procentach).

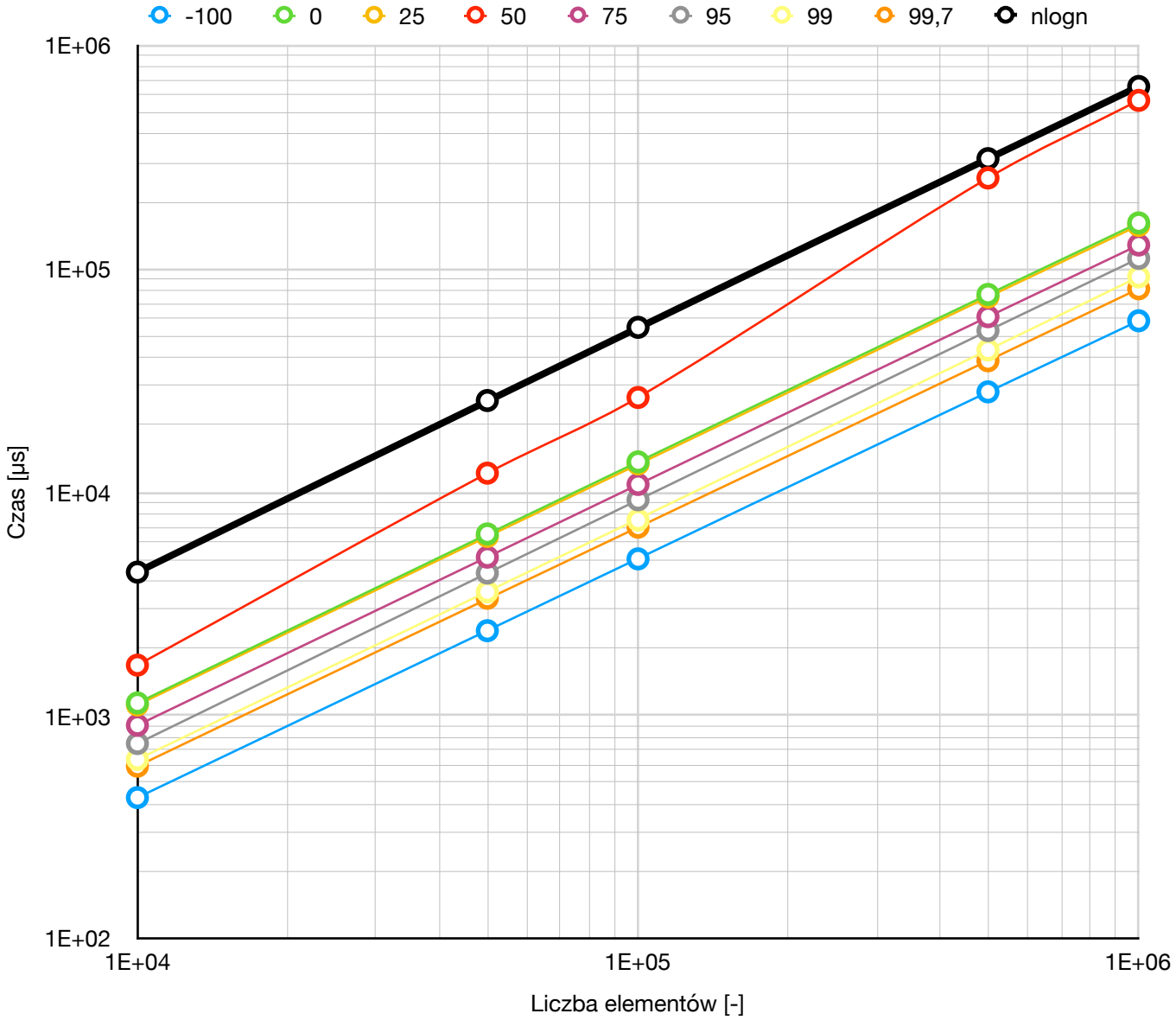


Jak widać z wykresów, rzeczywisty czas sortowania przez scalanie dla $n \leq 10^6$ jest w $O(nlog_2n)$. W tej akurat implementacji najszybciej algorytm zadziałał na tablicy prawie posortowanej, ponieważ implementacja ta działa na liście jednokierunkowej i nie dokonuje żadnych czynności w przypadku wystąpienia elementów w żądanej kolejności. Również dobrze działa na tablicy posortowanej w odwrotnej kolejności gdyż tylko przepisuje wskaźniki, odwracając kolejność listy. Najwolniej działa na tablicy losowej.

Wyniki testu sortowania przez scalanie

Liczba elem.	Średni czas sortowania tablicy [µs]								
	-100	0	25	50	75	95	99	99,7	nlogn
10000	765,494	1539,417	1359,808	1150,053	924,679	764,198	734,953	730,106	4384,945
50000	4365,031	10006,396	8501,481	6940,099	5473,624	4392,408	4238,173	4192,449	25755,907
100000	9295,735	22655,859	19012,978	15426,297	11967,712	9354,233	8961,475	8897,376	54811,814
500000	55636,570	220031,314	170897,452	127576,934	83968,347	55775,643	50487,251	50041,423	312370,881
1000000	117324,466	535000,425	413306,520	307916,168	197273,428	120635,886	106678,621	105181,188	657741,763

Sortowanie szybkie

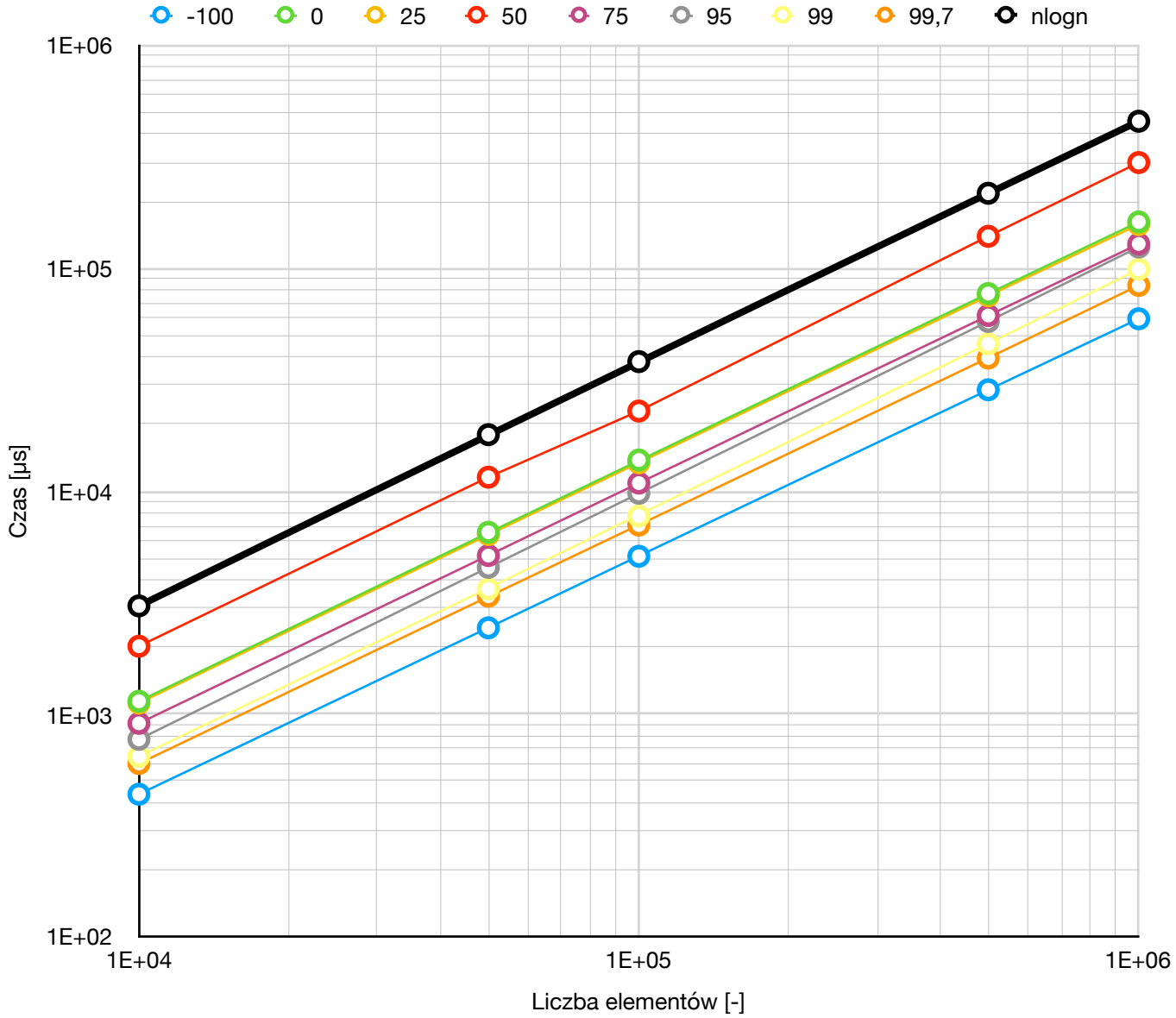


Jak widać z wykresów, rzeczywisty czas sortowania szybkiego dla $n \leq 10^6$ jest w $O(nlog_2n)$. Ta akurat implementacja najgorzej zadziałała na tablicy z posortowaną pierwszą połową, trudno jest mi zrozumieć dlaczego. Najszybciej natomiast posortował tablicę początkowo posortowaną w odwrotnej kolejności, gdyż już po pierwszym poziomie rekurencji cała tablica jest posortowana.

Wyniki testu sortowania szybkiego

Liczba elem.	Średni czas sortowania tablicy [µs]								
	-100	0	25	50	75	95	99	99,7	nlogn
10000	428,103	1136,757	1113,692	1679,068	903,101	748,006	635,000	592,075	4384,945
50000	2396,326	6502,484	6340,787	12173,716	5116,415	4343,562	3574,307	3348,078	25755,907
100000	5026,682	13676,721	13395,415	26577,550	10834,661	9247,963	7497,637	6971,564	54811,814
500000	28159,916	76995,793	74988,879	255870,852	61171,879	52991,552	43330,328	38803,740	312370,881
1000000	58645,841	161006,552	156872,267	569850,807	128140,961	111596,101	92263,166	81597,925	657741,763

Sortowanie introspektywne



Jak widać z wykresów, rzeczywisty czas sortowania introspektywnego dla $n \leq 10^6$ jest w $O(nlog_2n)$. Kolejność wykresów zgadza się z kolejnością wykresów w przypadku sortowania szybkiego. Widać ewidentnie, że znacznie polepszył się czas sortowania tablicy z pierwszą połową posortowaną.

Wyniki testu sortowania introspektywnego

Liczba elem.	Średni czas sortowania tablicy [µs]								
	-100	0	25	50	75	95	99	99,7	nlogn
10000	436,239	1140,826	1119,547	2014,806	907,853	769,439	644,477	599,331	3056,174
50000	2436,482	6543,352	6392,322	11563,219	5144,065	4551,586	3641,913	3379,242	17951,087
100000	5107,211	13812,879	13423,717	22915,072	10880,153	9791,737	7743,940	7047,047	38202,173
500000	28553,326	77208,286	75455,638	139540,303	61533,768	57853,419	45881,516	39625,503	217713,039
1000000	59490,434	161836,245	157534,173	299082,739	128922,932	124848,318	99307,702	83994,069	458426,077

Wnioski

Wykonując projekt, zapoznałem się lepiej z niektórymi algorytmami sortowania oraz implementowałem je. Zbadałem ich teoretyczną oraz rzeczywistą złożoność obliczeniową. Nauczyłem się interpretować notację dużego O: nie pozwala to policzyć czas wykonywania się algorytmów, zaś tylko je porównać, ale nawet algorytmy o tej samej złożoności w notacji dużego O mogą znacząco różnić się czasem działania.

Literatura

1. Merge sort, Wikipedia (*en.wikipedia.org/wiki/Merge sort*)
2. Quicksort, Wikipedia (*en.wikipedia.org/wiki/Quicksort*)
3. Heapsort, Wikipedia (*en.wikipedia.org/wiki/Heapsort*)
4. David R. Musser, Introspective Sorting and Selection Algorithms, Rensselaer Polytechnic Institute (*cs.rpi.edu/~musser/gp/introsort.ps*)