Sprawozdanie z laboratorium 6

Filip Malinowski

14 maja 2015

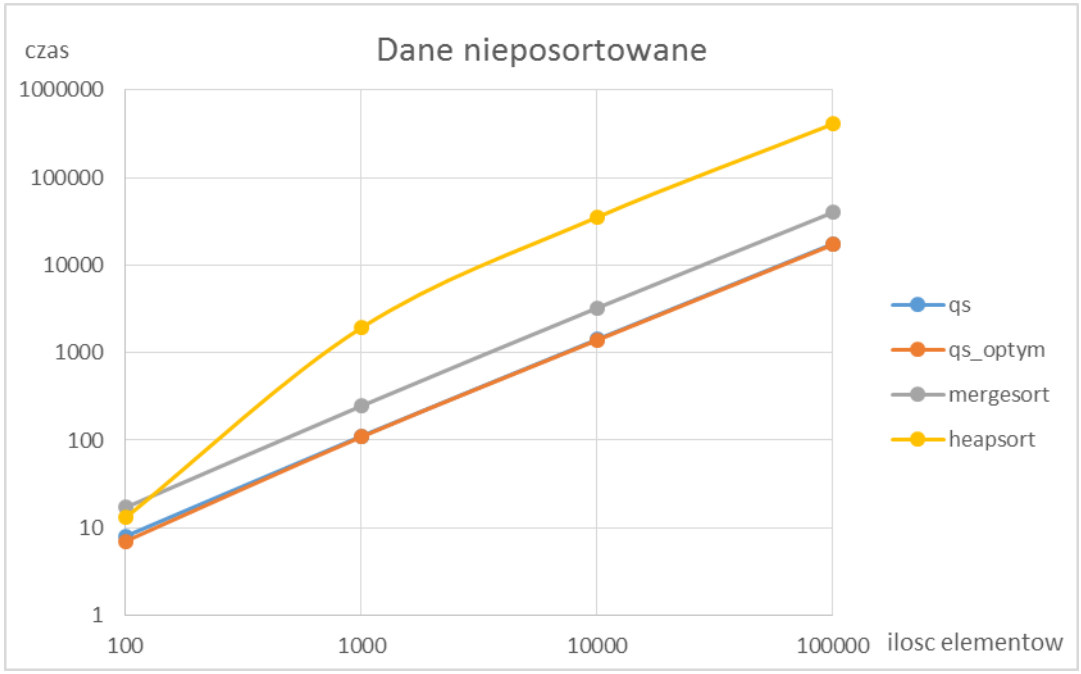
Implementacje sortowania przeniosłem z programu Szymona Furmańczyka do swojego z racji tego, że łatwiej mi było zautomatyzować badanie czasu sortowania.

W klasie Stos są teraz sortowania szybkie, przez scalanie i przez kopcowanie. Quicksort jest napisany w dwóch wersjach. Pierwsza, w której pivot pobierany jest z początku listy tablicowej. Druga w której pivot to średnia trzech elementów: z początku, środka i końca stosu.

Z otrzymanych danych można wywnioskować, że moja implementacja quicksort działa o wiele szybciej dla dużej ilości danych niż sortowanie przez kopcowanie. Natomiast po optymalizacji quicksort sortował wolniej dla małej ilości danych niż przed optymalizacją. Po optymalizacji pivota czas sortowania jest podobny do czasu sortowania bez optymalizacji. Dla drugiego zestawu danych wejściowych quicksort bez optymalizacji działał już 6 razy wolniej dla 100 000 elementów niż po optymalizacji.

Sortowanie przez kopcowanie ma dość wolny czas działania. Najwolniejszy z trzech napisanych przeze mnie sortowań. Implementację napisałem na nowo starając się napisać szybsze sortowanie.

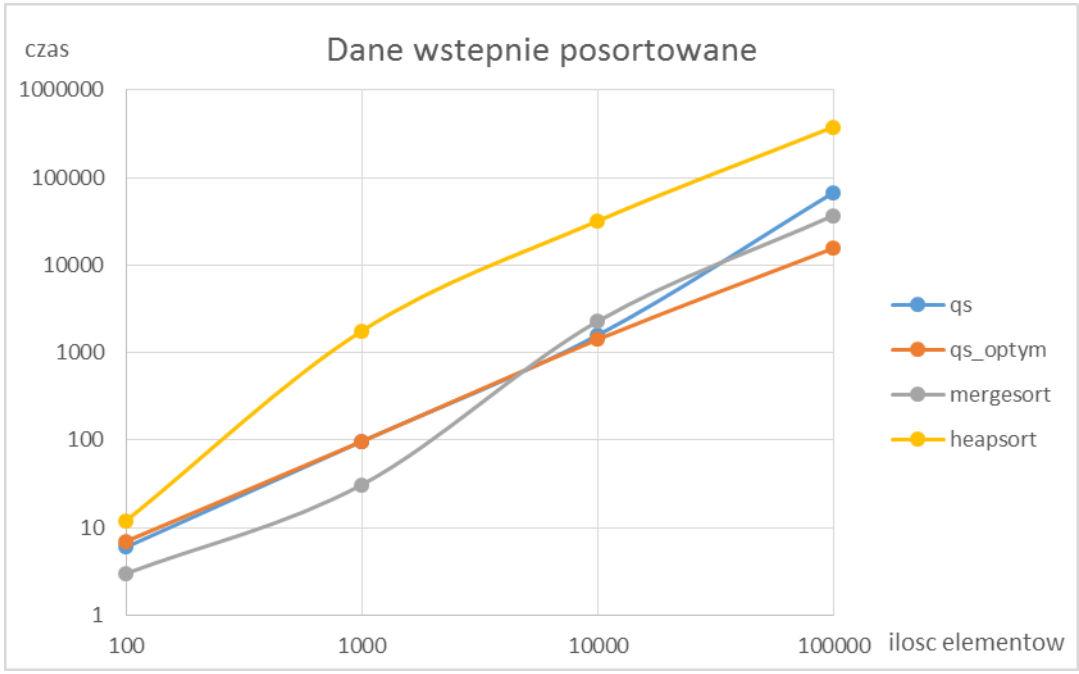
Sortowanie przez scalanie ma stałą złożoność obliczeniową w każdym przypadku danych wejściowych (posortowanych lub nie). Wynika to z działania algorytmu i zostało sprawdzone doświadczalnie podczas pomiarów. Podczas sortowania danych wstępnie posortowanych mergesort sortował wyraźnie szybciej niż quicksort po optymalizacji dla 100 i 1000 elementów Dla 10 000 i 100 000 sortował już wolniej. Można więc napisać sortowanie hybrydowe, które uruchamiałoby mergesort dla małej ilość danych wejściowych do sortowania, a powyżej pewnego progu (w tym przypadku 10 000 elemtów) uruchamiałoby quicksort. Można w ten sposób skorzystać z dobrych stron obu sortowań w jednym.



Rysunek 1: Czas sortowania elementów dla danych nieposortowanych

dane nieposortowane							
qs		qs optymalizacja		mergesort		heapsort	
elem	time	elem	time	elem	time	elem	time
100	8	100	7	100	17	100	13
1000	111	1000	109	1000	246	1000	1921
10000	1417	10000	1392	10000	3241	10000	35277
100000	17536	100000	17166	100000	40066	100000	409756

Rysunek 2: Tabela z wartościami pomiarowymi



Rysunek 3: Czas sortowania elementów dla danych wstępnie posortowanych

dane wstepnie posortowane							
qs		qs optymalizacja		mergesort		heapsort	
elem	time	elem	time	elem	time	elem	time
100	6	100	7	100	3	100	12
1000	97	1000	97	1000	31	1000	1774
10000	1585	10000	1419	10000	2283	10000	31981
100000	67559	100000	15610	100000	37021	100000	380004

Rysunek 4: Tabela z wartościami pomiarowymi