Projekt 1 - Algorytmy sortujące

Nazwa kursu	
Projektowanie Algorytmów i Metody Sztucznej Inteligencji	
Autor	**Data**
Dominik Koperkiewicz 254023	13.04.2021
Prowadząca	**Termin zajęć**
Mgr inż. Marta Emirsajłow	Wt 15:15

Wprowadzenie

Na potrzeby tego projektu zaimplementowałem trzy algorytmy sortujące (QuickSort, MergeSort i HeapSort) wykorzystując język C++, a następnie wykonałem testy prędkości dla tych algorytmów z użyciem różnych danych.

Algorytmy

- *QuickSort (Sortowanie szybkie)*

Algorytm QuickSort stosuje metodę „dziel i zwyciężaj”. Jeden z elementów zostaje wybrany jako tzw. *pivot*, a następnie elementy przestawiane są w taki sposób aby wszystkie elementy mniejsze od pivotu znalazły się po jednej stronie, a reszta po drugiej. Następnie ten sam algorytm jest powtarzany dla obu stron tablicy, aż do momentu kiedy uzyskamy pojedyncze elementy. Wybór pivotu jest kluczowy, ponieważ od tego zależy które przypadki są optymalne, a które nie.

- *MergeSort (Sortowanie przez scalanie)*

Algorytm ten polega na rekursywnym podzieleniu tablicy na coraz mniejsze części, a następnie scalaniu tych części jednocześnie układając elementy w odpowiedniej kolejności.

- *HeapSort (Sortowanie przez kopcowanie)*

Sortowanie przez kopcowanie polega na ułożeniu z tablicy kopca maksymalnego (drzewo binarne w którym każdy rodzic przechowuje większą/równą wartość do swoich dzieci). Następnie pierwszy element kopca odrzuca się na koniec tablicy, a ostatni element wchodzi na miejsce odrzuconego. Nowo powstały kopiec maksymalizujemy i cała procedura zostaje powtarzana, aż do momentu w którym zostaną odrzucone wszystkie elementy kopca.

Złożoność czasowa

*	Przypadek	
	Pesymistyczny	Średni
QuickSort	$O(n^2)$	$O(n \log n)$
MergeSort	$O(n \log n)$	$O(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$

Algorytm QuickSort jest najszybszy z tych trzech algorytmów, ale w pesymistycznym przypadku jest mało wydajny. Gdy jako pivot przyjmujemy pierwszy lub ostatni element to przypadek pesymistyczny występuje dla tablic posortowanych. Jeśli odpowiednio dobierzemy pivot, to najgorszy przypadek będzie dużo bardziej nietypowy, dlatego też zmniejszy się prawdopodobieństwo jego wystąpienia.

Lepszymi metodami doboru pivotu zamiast pierwszego lub ostatniego elementu są na przykład:

- Wybór środkowego elementu (Taką metodę zastosowałem w mojej implementacji).
- Wybór mediany z pierwszego, środkowego i ostatniego elementu.
- Wybór losowego elementu.

Alternatywnie zamiast algorytmu QuickSort można użyć np. algorytm hybrydowy IntroSort (sortowanie introspektywne), który łączy QuickSort i HeapSort tak, aby zapobiec występowaniu najgorszego przypadku.

MergeSort jest bardziej stabilnym algorytmem, niż QuickSort, ale z reguły jest wolniejszy. Jedną z jego największy wad jest jego złożoność pamięciowa która wynosi $O(n)$ ze względu na fakt, że MergeSort musi tworzyć kopie tablic podczas scalania.

HeapSort natomiast działa bezpośrednio na sortowanej tablicy dlatego nie zużywa tyle pamięci. Jest on trochę wolniejszy od MergeSort (chociaż w przypadku mojej implementacji jest na odwrót).

Przebieg testów

Podczas każdego testu w sortowane było 100 tablic składających się z n elementów z czego początkowe p procent elementów było posortowanych (elementy przyjmowały wartość numeru indeksu tablicy), a kolejne elementy były losowymi liczbami większymi od ostatniej posortowanej.

Ostatnie testy sprawdzały przypadek w którym tablica była posortowana odwrotnie.

Każdy algorytm sortował taki sam zestaw danych.

W poniżej zamieszczonych tabelach przedstawione są czasy (w milisekundach) w zależności od ilości elementów n i ilości posortowanych elementów p (wyrażone w procentach).

QuickSort

*	10 000	50 000	100 000	500 000	1 000 000
losowe	164	969	2071	11680	25371
25%	143	872	1810	10286	22059
50%	117	734	1547	8554	18349
75%	97	627	1330	7286	15283
95%	86	550	1104	6195	13063
99%	83	499	1097	5985	12930
99,70%	81	496	1072	5944	12813
odwrotne	93	520	1122	6481	13483

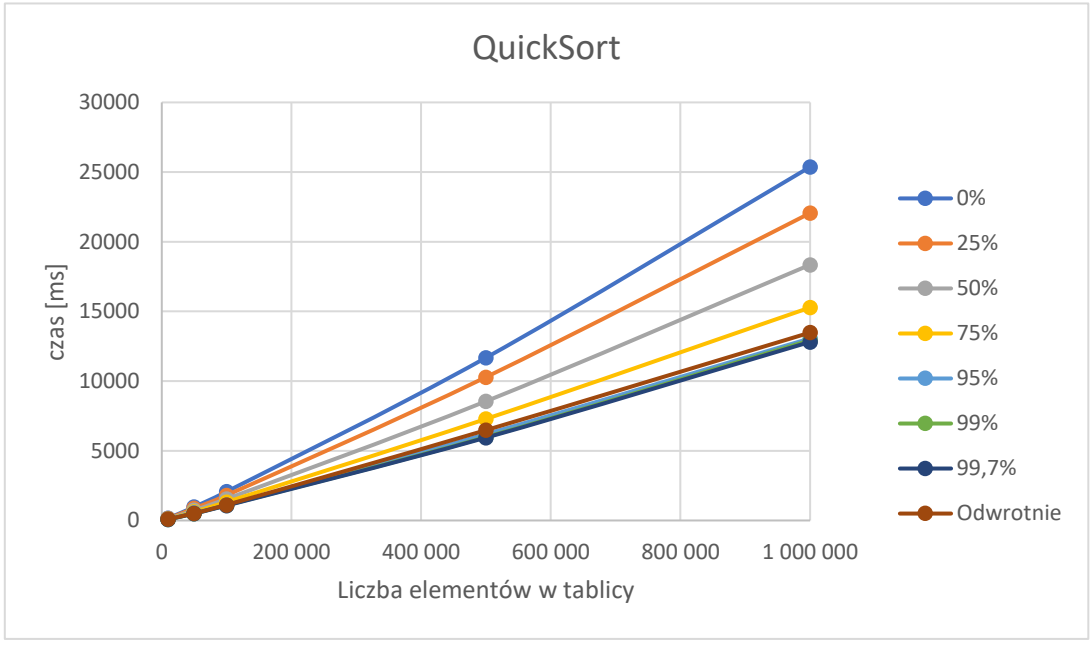
MergeSort

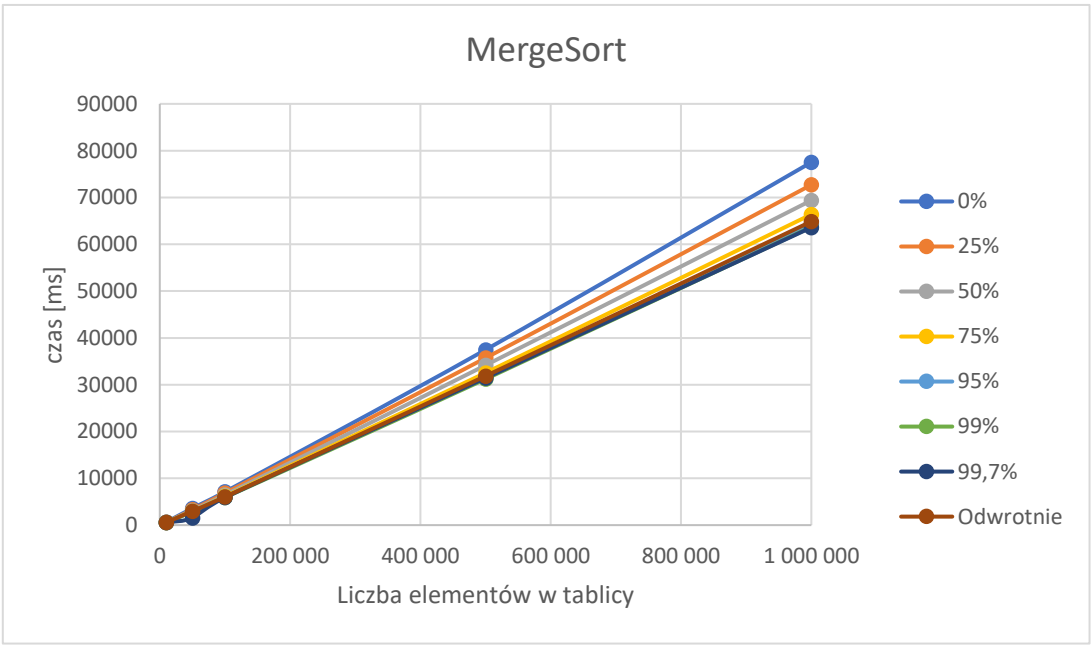
*	10 000	50 000	100 000	500 000	1 000 000
losowe	646	3587	7106	37457	77521
25%	614	3285	6838	35700	72702
50%	586	3126	6456	34153	69386
75%	566	2999	6150	32541	66386
95%	549	2912	5995	31610	64296
99%	560	2900	5908	31211	63711
99,70%	553	1538	5908	31437	63596
odwrotne	562	2996	6025	31856	64873

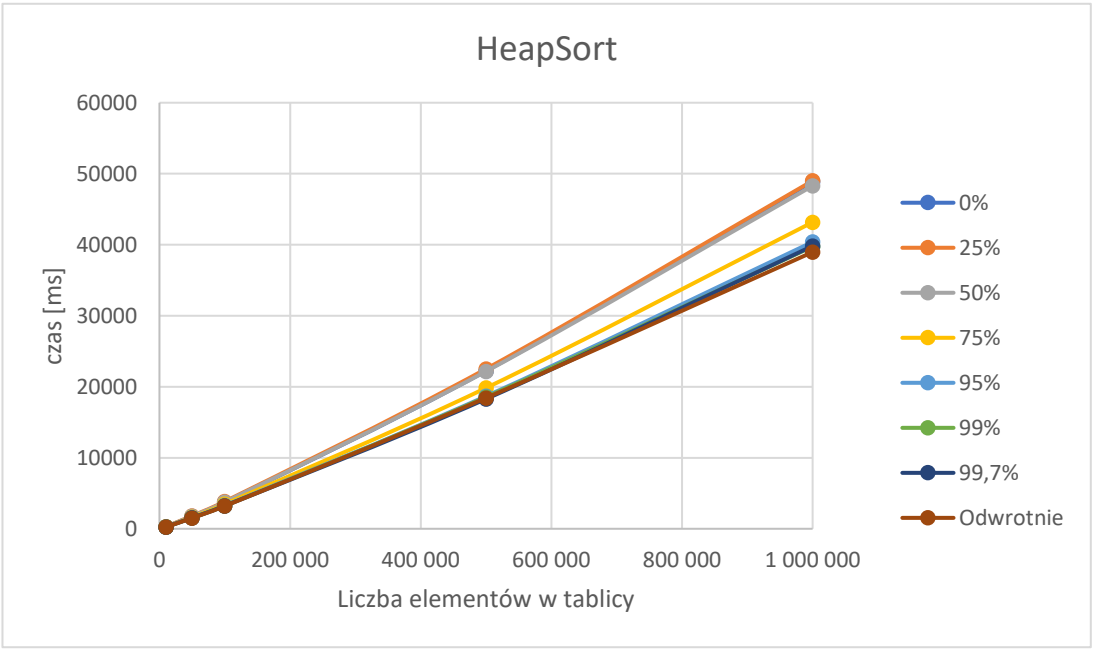
HeapSort

*	10 000	50 000	100 000	500 000	1 000 000
losowe	290	1732	3728	22224	48883
25%	292	1831	3842	22514	49004
50%	288	1768	3752	22170	48293
75%	273	1634	3495	19833	43168
95%	263	1549	3288	18673	40391
99%	262	1538	3240	18554	39737
99,70%	262	1536	3231	18279	39831
odwrotne	251	1515	3202	18423	38966

Wykresy na podstawie danych z tabel:







Podsumowanie i wnioski

Można zauważyć że wykresy faktycznie reprezentuje krzywa $n \log n$

W przypadku wszystkich algorytmów im bardziej posortowane były dane tym krótszy był czas sortowania. Wariant z odwróconą tablicą był nieco wolniejszy od wariantu z posortowaną tablicą w 99,7% lecz nadal był szybki.

Wykonując kilka testów dla Algorytmu QuickSort z wybieraniem ostatniego elementu jako pivotu otrzymałem czasy nawet kilkaset razy większe (jak pokazuje poniższa tabela).

*	10 000
losowe	174ms
25%	3468ms
50%	13333ms
75%	29907ms
95%	477772ms
99%	52266ms
99,70%	52826ms
odwrotne	32312ms

Jak można zauważyć, algorytm HeapSort osiągnął wyniki około dwa razy lepsze od MergeSort, mimo że to MergeSort powinien być szybszy. Powodem może być fakt że w algorytmie MergeSort na okrągło wywoływany jest konstruktor dla typu std::vector dla dodatkowych tablic na potrzeby scalania, natomiast HeapSort nie wykorzystuje dodatkowych tablic. W przypadku wykorzystania list MergeSort mógłby zyskać na szybkości.

Literatura

- https://www.youtube.com/watch?v=qBo_9cgBVpk
- https://www.youtube.com/watch?v=mB5HXBb_HY8
- https://pl.wikipedia.org/wiki/Sortowanie_przez_scalanie
- https://www.youtube.com/watch?v=M2bKENbdnI4
- https://www.youtube.com/watch?v=MtQL_ll5KhQ
- https://en.wikipedia.org/wiki/Merge_sort