Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji (W4N)
Informatyka Techniczna

Badanie efektywności wybranych algorytmów sortowania

Przedmiot: Algorytmy i Złożoność Obliczeniowa

Autor: Mikołaj Winiecki

Semestr: Letni 2024/2025
Data: 26 kwietnia 2024

1. Algorytmy Sortowania

1.1. Insertion Sort

Insertion Sort to algorytm sortowania, który przetwarza dane element po elemencie, wstawiając każdy nowy element w odpowiednie miejsce w już posortowanej części tablicy. Na początku pierwszy element uznawany jest za posortowany. Następnie kolejny element jest porównywany z wcześniejszymi i wstawiany w odpowiednie miejsce, przesuwając większe elementy w prawo.

1.2. Heap Sort

Heap Sort wykorzystuje strukturę danych zwaną kopcem do sortowania elementów. Algorytm najpierw przekształca tablicę w kopiec maksymalny, gdzie każdy element nadrzędny jest większy od swoich potomków. Następnie największy element (korzeń kopca) jest zamieniany z ostatnim elementem tablicy, a rozmiar kopca jest zmniejszany. Proces budowy kopca i zamiany największego elementu jest powtarzany aż do całkowitego uporządkowania danych.

1.3. Quick Sort

Quick Sort to algorytm oparty na metodzie "dziel i zwyciężaj". Polega na wyborze elementu zwanego pivotem, wokół którego dzielona jest tablica. Elementy mniejsze od pivota są przesuwane na lewo, a większe na prawo. Następnie procedura jest powtarzana rekurencyjnie dla powstałych podtablic, aż do momentu, gdy wszystkie części składowe będą miały jeden lub zero elementów.

1.4. Shell Sort

Shell Sort jest ulepszoną wersją Insertion Sorta, w której elementy są najpierw porównywane i zamieniane na większych odległościach, a następnie na coraz mniejszych. Pozwala to szybciej przesuwać elementy na odpowiednie miejsca i zmniejszyć liczbę przestawień, które są konieczne do pełnego uporządkowania danych.

1.5. Drunk Shell Sort

Drunk Shell Sort to wariant klasycznego Shell Sorta, który imituje sortowanie danych przez pijanego studenta. Podstawowy mechanizm sortowania pozostaje bez zmian. Unikalną cechą tego algorytmu jest to, że po każdym pełnym przejściu przez tablicę dla danego odstępu wprowadzane jest krótkie uśpienie programu (sleep), którego długość zależy od zadanego poziomu pijaństwa. To opóźnienie, które nie wpływa na wynik sortowania, ale wydłuża czas działania algorytmu, jest spowodowane przysypianiem studenta zmęczonego po długim maratonie alkoholowym.

2. Badanie 0 – Wybór parametrów

W tym badaniu sprawdzone zostaną parametry poszczególnych algorytmów takie jak różne pivoty Quick Sorta i wybrane wzory odstępów dla Shell Sorta w celu wybrania najoptymalniejszych do kolejnych badań. Pomiary zostaną przeprowadzone na zbiorach danych typu int. Zarówno w tym jak i kolejnych badaniach każdy pomiar będzie przeprowadzony 100 razy a wyniki uśrednione.

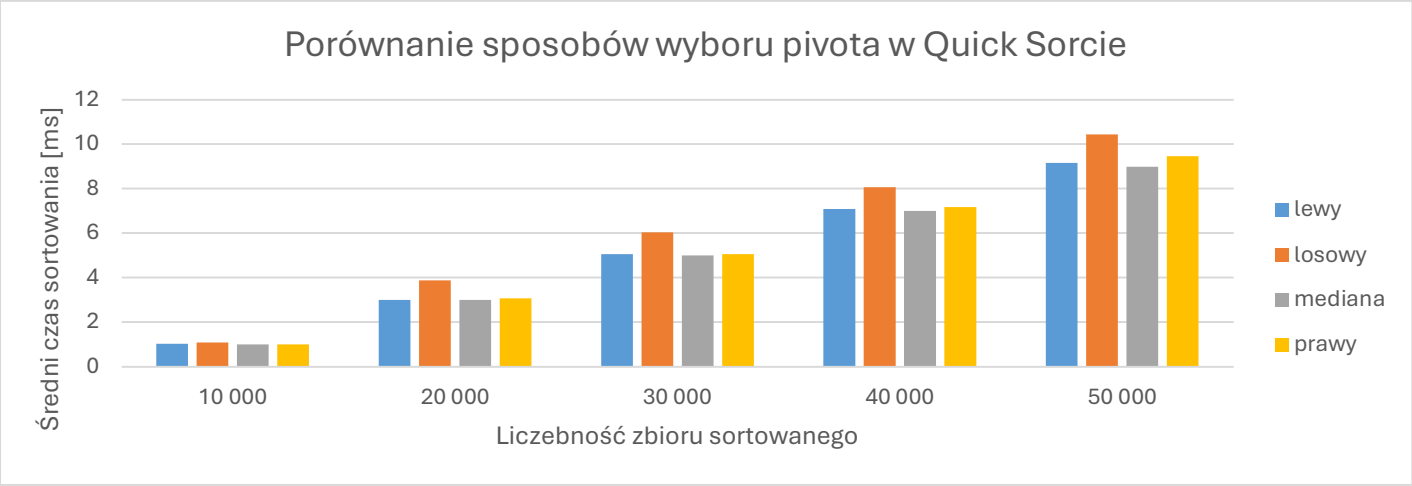
2.1. Wybór pivota w Quick Sorcie

Jeśli chodzi o wybór pivota to zdefiniować można cztery główne strategie:

- **Lewy** element – najprostsza strategia, jako pivot wybierany jest pierwszy element tablicy,
- **Prawy** element – pivotem staje się ostatni element tablicy,
- **Losowy** element – pivot wybierany jest losowo spośród dostępnych elementów tablicy,
- **Mediana** z trzech – pivot wybierany jako mediana z trzech elementów: pierwszego, środkowego i ostatniego.

Pierwsze badanie zostało przeprowadzone pod kątem średniego czasu wykonywania dla różnych liczebności zbiorów sortowanych, a wyniki prezentują się następująco:

parametry badania					wyniki				
algorytm	**dystrybucja**	**typ danych**	**pivot**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
quick sort	losowa	int	**lewy**	**10 000**	1,03	1	4	1	0,30
quick sort	losowa	int	**lewy**	**20 000**	3,00	3	3	3	0,00
quick sort	losowa	int	**lewy**	**30 000**	5,05	5	6	5	0,22
quick sort	losowa	int	**lewy**	**40 000**	7,08	7	8	7	0,27
quick sort	losowa	int	**lewy**	**50 000**	9,16	9	10	9	0,37
quick sort	losowa	int	**prawy**	**10 000**	1,00	1	1	1	0,00
quick sort	losowa	int	**prawy**	**20 000**	3,07	3	7	3	0,43
quick sort	losowa	int	**prawy**	**30 000**	5,06	5	6	5	0,24
quick sort	losowa	int	**prawy**	**40 000**	7,17	7	8	7	0,38
quick sort	losowa	int	**prawy**	**50 000**	9,46	9	10	9	0,50
quick sort	losowa	int	**losowy**	**10 000**	1,08	1	2	1	0,27
quick sort	losowa	int	**losowy**	**20 000**	3,88	3	4	4	0,33
quick sort	losowa	int	**losowy**	**30 000**	6,03	6	7	6	0,17
quick sort	losowa	int	**losowy**	**40 000**	8,06	8	9	8	0,24
quick sort	losowa	int	**losowy**	**50 000**	10,44	10	12	10	0,52
quick sort	losowa	int	**mediana**	**10 000**	1,00	1	1	1	0,00
quick sort	losowa	int	**mediana**	**20 000**	3,00	3	3	3	0,00
quick sort	losowa	int	**mediana**	**30 000**	5,00	5	5	5	0,00
quick sort	losowa	int	**mediana**	**40 000**	7,00	7	7	7	0,00
quick sort	losowa	int	**mediana**	**50 000**	8,98	8	10	9	0,20



Na podstawie przeprowadzonych pomiarów można zauważyć, że wybór pivota jako mediany trzech elementów zapewniał najlepsze rezultaty. Sortowanie przy takiej strategii było nie tylko najszybsze, ale także najbardziej stabilne odchylenie standardowe wynosiło zero lub było bardzo bliskie zeru.

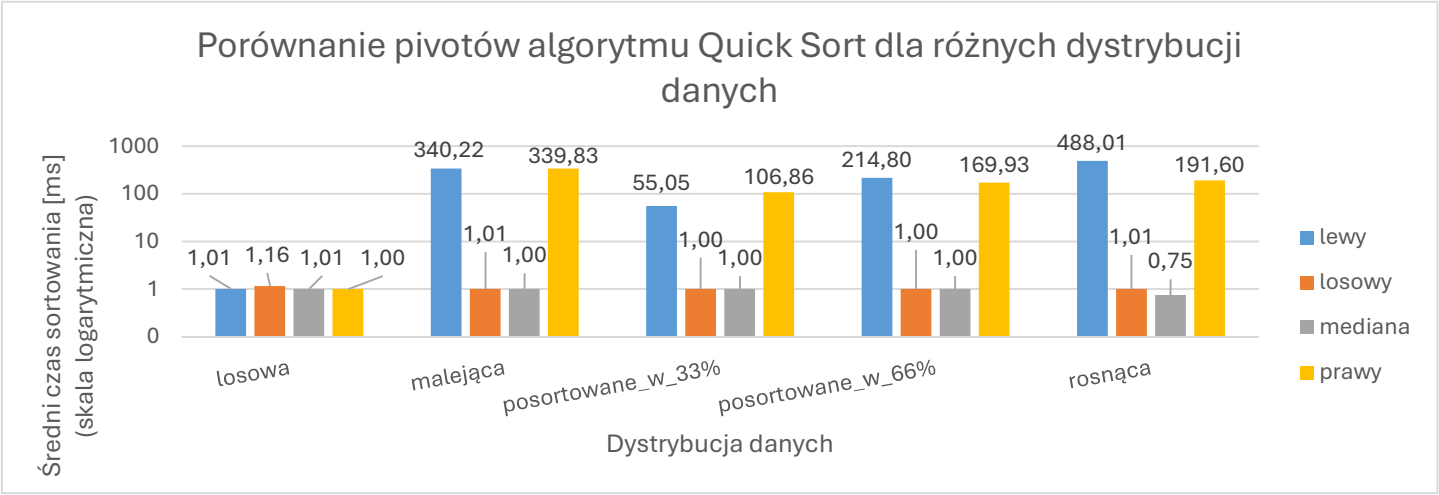
W przypadku wyboru lewego lub prawego elementu jako pivota, czasy sortowania były nieco wyższe niż przy medianie, ale nadal dobre. Odchylenie standardowe w tych przypadkach było nieznaczne, co oznacza, że wyniki były dość powtarzalne, choć nie tak idealnie równe jak w przypadku mediany. W praktyce oznacza to, że przy losowym rozkładzie danych wybór lewego lub prawego pivota jest akceptowalny, ponieważ prostota implementacji rekompensuje minimalnie gorszą wydajność.

Wybór pivota w sposób losowy okazał się najmniej korzystny. Przy mniejszych zbiorach wyniki były jeszcze zbliżone do pozostałych, jednak wraz ze wzrostem liczebności danych średni czas sortowania rósł szybciej niż w pozostałych metodach. Dodatkowo zauważalnie wzrastało odchylenie standardowe, co wskazuje na dużą zmienność wyników i mniejszą przewidywalność wydajności algorytmu.

Podsumowując, na losowo rozłożonych danych najlepsze rezultaty osiąga się wybierając pivot jako medianę trzech elementów. Wybór lewego lub prawego elementu daje zadowalające, nieznacznie gorsze wyniki.

Drugie badanie dotyczyło rozkładu danych i dopiero tutaj widać jak wielkie znaczenie ma odpowiednia strategia wyboru pivota:

parametry badania					wyniki				
algorytm	**pivot**	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
quick sort	lewy	10 000	int	**losowa**	1,01	1	2	1	0,10
quick sort	lewy	10 000	int	**rosnąca**	488,01	481	495	488	1,98
quick sort	lewy	10 000	int	**malejąca**	340,22	336	364	340	2,73
quick sort	lewy	10 000	int	**posortowane_w_33%**	55,05	55	56	55	0,22
quick sort	lewy	10 000	int	**posortowane_w_66%**	214,80	214	221	215	1,09
quick sort	prawy	10 000	int	**losowa**	1,00	1	1	1	0,00
quick sort	prawy	10 000	int	**rosnąca**	191,60	190	194	191	0,86
quick sort	prawy	10 000	int	**malejąca**	339,83	337	345	340	1,25
quick sort	prawy	10 000	int	**posortowane_w_33%**	106,86	105	109	107	0,70
quick sort	prawy	10 000	int	**posortowane_w_66%**	169,93	169	174	170	0,87
quick sort	losowy	10 000	int	**losowa**	1,16	1	2	1	0,37
quick sort	losowy	10 000	int	**rosnąca**	1,01	1	2	1	0,10
quick sort	losowy	10 000	int	**malejąca**	1,01	1	2	1	0,10
quick sort	losowy	10 000	int	**posortowane_w_33%**	1,00	1	1	1	0,00
quick sort	losowy	10 000	int	**posortowane_w_66%**	1,00	1	1	1	0,00
quick sort	mediana	10 000	int	**losowa**	1,01	1	2	1	0,10
quick sort	mediana	10 000	int	**rosnąca**	0,75	0	1	1	0,44
quick sort	mediana	10 000	int	**malejąca**	1,00	1	1	1	0,00
quick sort	mediana	10 000	int	**posortowane_w_33%**	1,00	1	1	1	0,00
quick sort	mediana	10 000	int	**posortowane_w_66%**	1,00	1	1	1	0,00



W przypadku losowego pivota i mediany trzech elementów czasy sortowania były bardzo niskie i stabilne niezależnie od rodzaju danych. Natomiast wybór lewego lub prawego pivota powodował drastyczny wzrost czasu działania dla danych uporządkowanych, szczególnie rosnąco lub malejąco. Dzieje się tak, ponieważ skrajne pivoty prowadzą do bardzo nierównych podziałów danych. Zamiast dzielić zbiór na dwie podobne części, większość elementów trafia do jednej strony. Wybór mediany lub elementu losowego pomaga zachować dobre podziały i wysoką wydajność, nawet gdy dane są uporządkowane lub częściowo posortowane.

Podsumowując oba testy, jeśli chodzi o liczbę danych do sortowania to wybór pivota nie ma więszego wpływu na wydajność, ale gdy dane mogą być nierównomiernie rozłożone to mediana jest niewiele lepsza od wyboru losowego ale zdecydowanie lepsza od lewego i prawego i dlatego to ona zostanie użyta w dalszych badaniach.

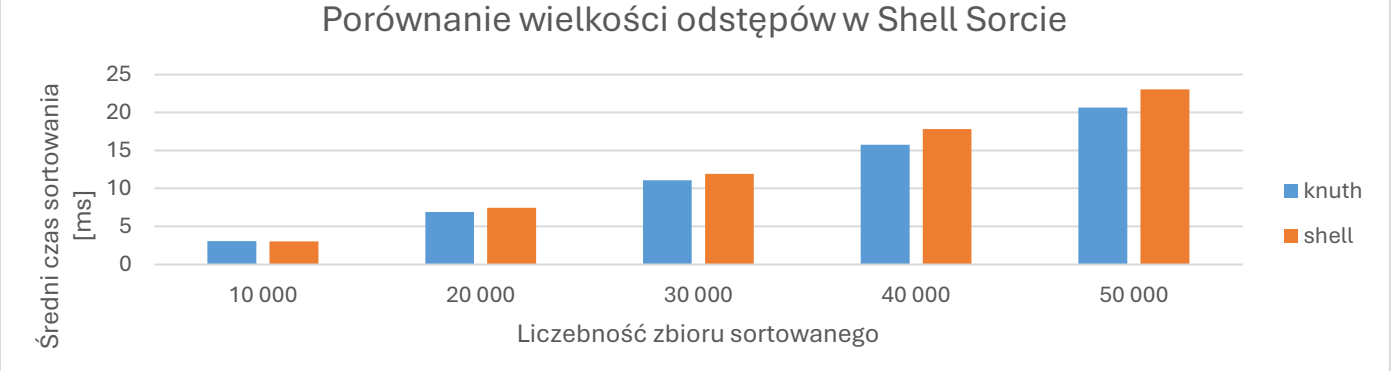
2.2. Ciągi odstępów w Shell Sorcie

W Shell Sorcie wykorzystuje się tzw. ciągi odstępów, czyli odległości między porównywanymi elementami. Pomysłów na obliczanie kolejnych odstępów może być nieskończenie wiele. W tym badaniu przetestujemy dwa:

- Klasyczny ciąg **Shella** – podziały rozmiaru tablicy przez 2: n/2, n/4, n/8, ..., 1. Każdy kolejny przebieg sortuje elementy oddalone o dany odstęp, aż na końcu sortuje się sąsiednie elementy.
- Ciąg **Knutha** – Odstępy wyznacza się wzorem h=3h+1, (1, 4, 13, 40, 121...). Przy sortowaniu zaczyna się od największego odstępu mniejszego od rozmiaru tablicy i stopniowo zmniejsza do 1.

Badanie zostało przeprowadzone dla różnych liczebności zbioru sortowanego i dało takie wyniki:

parametry badania					wyniki				
algorytm	**dystrybucja**	**typ danych**	**odstęp**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
shell sort	losowa	int	**shell**	**10 000**	3,00	3	3	3	0,00
shell sort	losowa	int	**shell**	**20 000**	7,42	7	8	7	0,50
shell sort	losowa	int	**shell**	**30 000**	11,90	11	13	12	0,33
shell sort	losowa	int	**shell**	**40 000**	17,83	17	32	18	1,54
shell sort	losowa	int	**shell**	**50 000**	23,06	22	26	23	0,63
shell sort	losowa	int	**knuth**	**10 000**	3,04	2	7	3	0,42
shell sort	losowa	int	**knuth**	**20 000**	6,88	6	8	7	0,36
shell sort	losowa	int	**knuth**	**30 000**	11,08	10	12	11	0,34
shell sort	losowa	int	**knuth**	**40 000**	15,74	15	17	16	0,58
shell sort	losowa	int	**knuth**	**50 000**	20,64	19	23	21	0,70



Na podstawie wyników i wykresu można wyciągnąć wniosek, że zastosowanie ciągu Knutha pozwala uzyskać lepsze czasy sortowania niż klasyczny ciąg Shella, zwłaszcza przy większych rozmiarach danych. W każdej testowanej liczebności zbioru sortowanego algorytm używający odstępów Knutha działał szybciej. Wynika to z tego, że ciąg Knutha szybciej zmniejsza odstępy i szybciej przechodzi do dokładniejszego sortowania końcowego, dzięki czemu szybciej uporządkowuje dane. Klasyczny ciąg Shella zmniejsza odstępy wolniej, przez co wykonuje więcej niepotrzebnych przestawień przy dużych odstępach, co wydłuża czas działania.

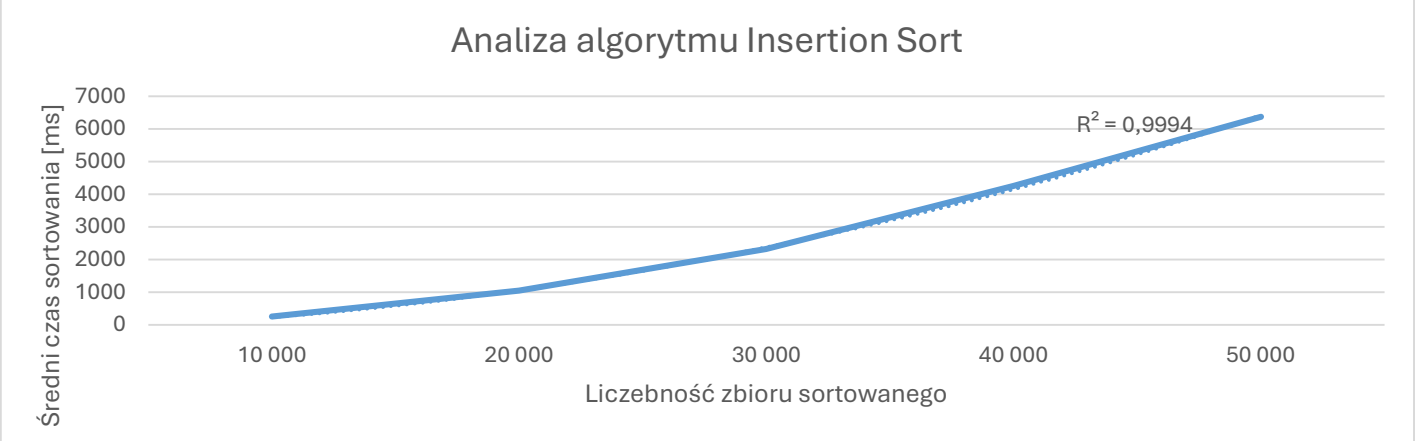
Podsumowując, ciąg Knutha jest lepszym wyborem i to on zostanie użyty w dalszych badaniach.

3. Badanie 1 – Wpływ liczebności zbioru na czas sortowania

Celem badania jest sprawdzenie, jak liczebność zbioru danych wpływa na czas działania algorytmów sortowania. Dla każdego algorytmu wykonano pomiary na pięciu różnych wielkościach zbiorów (10, 20, 30, 40, 50-tysięcy) danych typu int o losowym rozkładzie. Pozwoli to zobaczyć, jak rośnie czas sortowania wraz ze wzrostem liczby elementów.

3.1. Insertion Sort

parametry badania				wyniki				
algorytm	**typ danych**	**dystrybucja**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
insertion sort	int	losowa	**10 000**	259,20	253	311	258	5,96
insertion sort	int	losowa	**20 000**	1040,78	1010	1228	1036,5	28,67
insertion sort	int	losowa	**30 000**	2315,52	2268	2729	2306	49,89
insertion sort	int	losowa	**40 000**	4255,50	4040	4896	4254	137,29
insertion sort	int	losowa	**50 000**	6374,80	6289	6488	6376,5	32,84



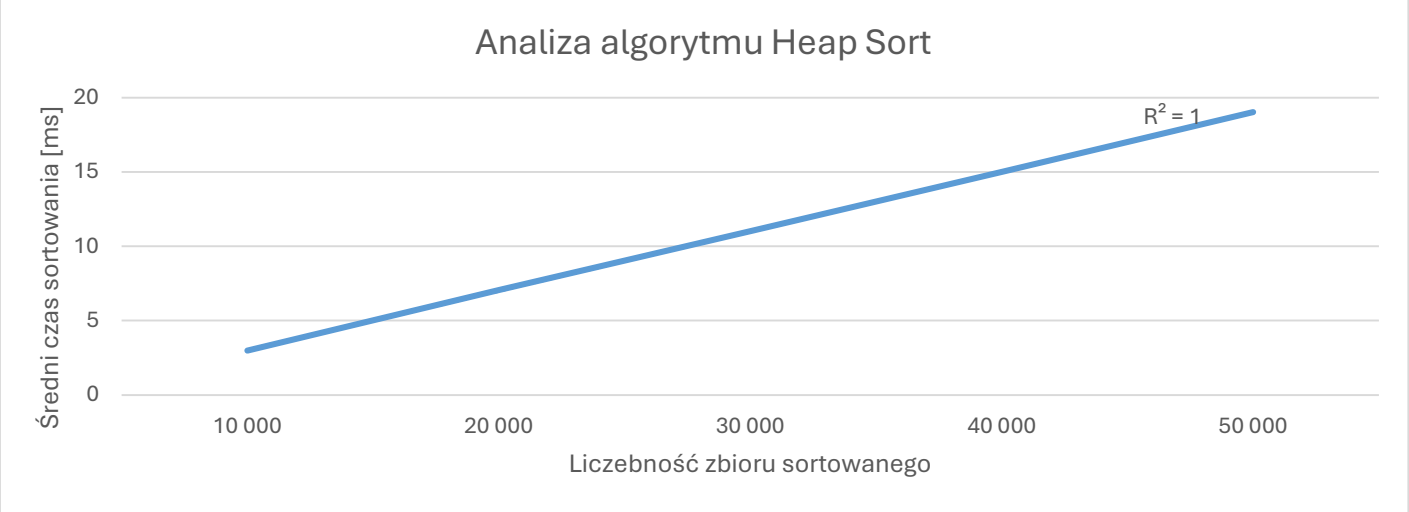
Wyniki pokazują wyraźnie, że średni czas sortowania rośnie szybciej niż liniowo, zgodnie z teoretyczną złożonością czasową Insertion Sort wynoszącą O(n^2). Dla 10 000 elementów średni czas wyniósł około 259 ms, a dla 50 000 elementów przekroczył 6300 ms. Wysoka wartość współczynnika determinacji R^2 = 0,9994 potwierdza bardzo dobrą zgodność przyrostu czasu z trendem kwadratowym.

Wzrost odchylenia standardowego dla większych zbiorów sugeruje, że Insertion Sort staje się coraz bardziej wrażliwy na różnice w początkowej kolejności danych wraz ze wzrostem ich liczby. Mimo że dla niewielkich zbiorów algorytm działa sprawnie, dla większych liczności staje się zdecydowanie nieefektywny.

Podsumowując, Insertion Sort przez swoją kwadratową złożonosc czasową nadaje się wyłącznie do małych zbiorów.

3.2. Heap Sort

parametry badania				wyniki				
algorytm	**typ danych**	**dystrybucja**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
heap sort	int	losowa	**10 000**	3,00	3	3	3	0,00
heap sort	int	losowa	**20 000**	7,06	7	13	7	0,60
heap sort	int	losowa	**30 000**	11,02	10	18	11	0,74
heap sort	int	losowa	**40 000**	15,00	15	15	15	0,00
heap sort	int	losowa	**50 000**	19,03	19	20	19	0,17

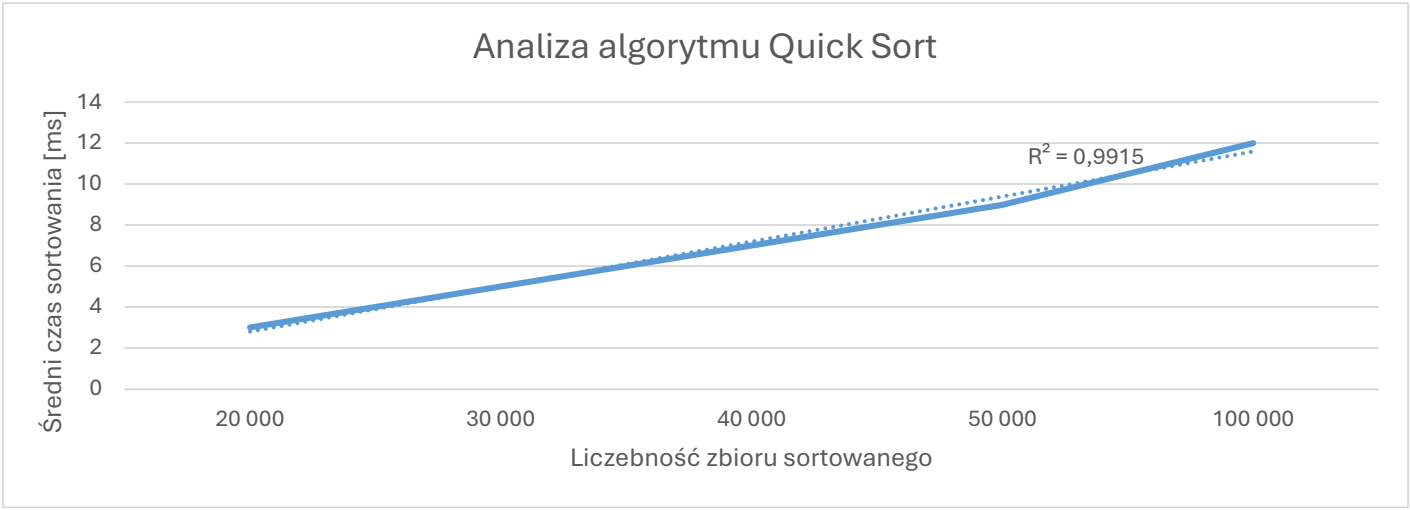


Średni czas sortowania rósł w sposób bardzo regularny: od 3 ms przy 10 000 elementach do około 19 ms przy 50 000 elementach. Odchylenie standardowe było minimalne, co świadczy o wysokiej powtarzalności pomiarów. Dodatkowo współczynnik dopasowania trendu R^2 = 1 wskazuje na niemal idealne dopasowanie z trendem liniowym.

Heap Sort charakteryzuje się złożonością czasową O(nlogn) w każdym przypadku, co znajduje odzwierciedlenie w obserwowanym umiarkowanym wzroście czasu wraz ze wzrostem liczby elementów. Algorytm zapewnia stabilną wydajność przy różnych rozmiarach danych.

3.3. Quick Sort

parametry badania					wyniki				
algorytm	**pivot**	**typ danych**	**dystrybucja**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
quick sort	mediana	int	losowa	**10 000**	1,00	1	1	1	0,00
quick sort	mediana	int	losowa	**20 000**	3,00	3	3	3	0,00
quick sort	mediana	int	losowa	**30 000**	5,00	5	5	5	0,00
quick sort	mediana	int	losowa	**40 000**	7,00	7	7	7	0,00
quick sort	mediana	int	losowa	**50 000**	8,98	8	10	9	0,20



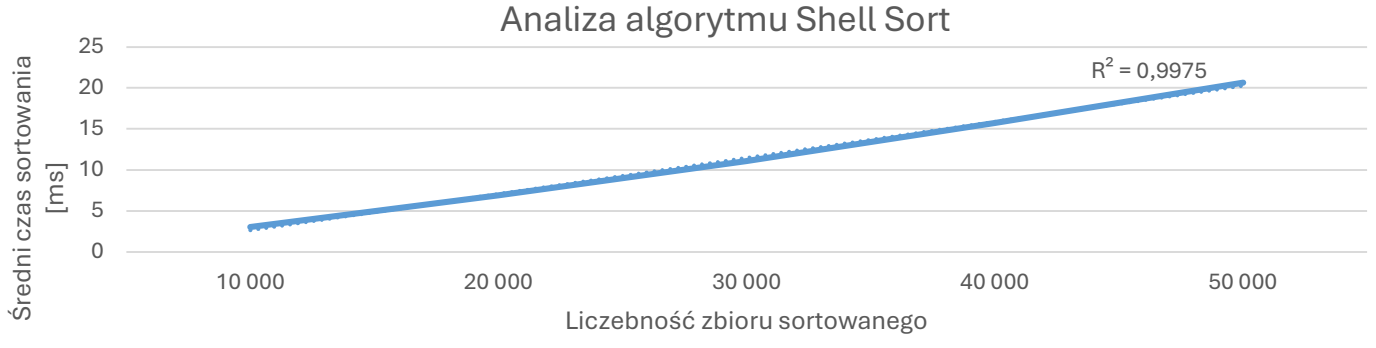
Uzyskane wyniki pokazują, że średni czas sortowania rośnie w przybliżeniu liniowo wraz ze wzrostem rozmiaru danych. Przykładowo, dla 10 000 elementów średni czas wyniósł 1 ms, a dla 50 000 elementów, około 9 ms. Odchylenie standardowe było minimalne lub zerowe, co wskazuje na bardzo powtarzalne i stabilne działanie algorytmu w przypadku danych losowych i przy wyborze dobrego pivota.

Wyniki są zgodne z teoretyczną złożonością czasową algorytmu Quick Sort, która dla dobrze dobranego pivota wynosi O(nlogn). Wybieranie pivota jako mediany trzech elementów zmniejsza ryzyko wystąpienia przypadków, w których Quick Sort działałby w czasie kwadratowym O(n^2)), i zapewnia zrównoważone dzielenie zbioru.

Podsumowując, Quick Sort z pivotem jako medianą trzech sprawdza się bardzo dobrze dla danych losowych i większych zbiorów, zapewniając szybkie i stabilne sortowanie przy korzystnej złożoności obliczeniowej.

3.4. Shell Sort

parametry badania					wyniki				
algorytm	**odstęp**	**typ danych**	**dystrybucja**	**liczebność zbioru**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
shell sort	knuth	int	losowa	**10 000**	3,04	2	7	3	0,42
shell sort	knuth	int	losowa	**20 000**	6,88	6	8	7	0,36
shell sort	knuth	int	losowa	**30 000**	11,08	10	12	11	0,34
shell sort	knuth	int	losowa	**40 000**	15,74	15	17	16	0,58
shell sort	knuth	int	losowa	**50 000**	20,64	19	23	21	0,70

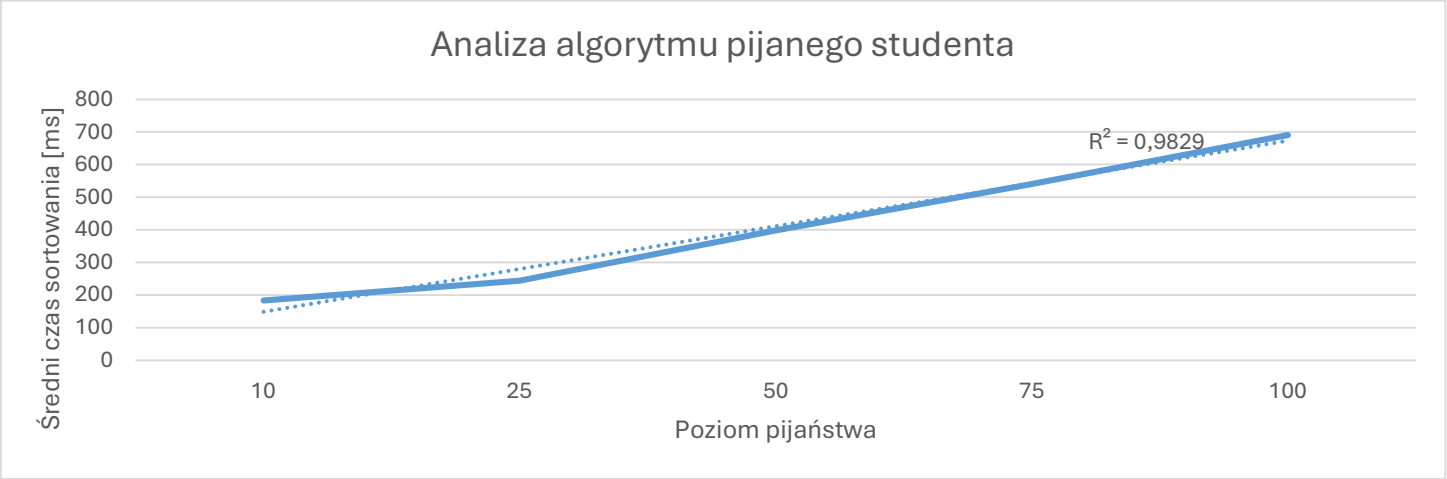


Czas wykonania algorytmu rósł stopniowo wraz ze wzrostem liczby elementów: od około 3 ms dla 10 000 elementów do około 21 ms dla 50 000 elementów. Wartości odchylenia standardowego były niewielkie, co świadczy o powtarzalności wyników i stabilności działania algorytmu. Na podstawie regresji liniowej uzyskano współczynnik determinacji R^2 = 0,9975, co oznacza bardzo wysoką zgodność wzrostu czasu z przyrostem rozmiaru danych, choć nie idealną.

Złożoność czasowa Shell Sorta w wersji z ciągiem Knutha wynosi średnio O(n^3/2), co przekłada się na lepszą efektywność w porównaniu do klasycznej wersji Shell Sorta. Wyniki pomiarów potwierdzają, że algorytm radzi sobie znacznie lepiej niż sortowania kwadratowe (np. Insertion Sort), a wzrost czasu działania jest umiarkowany nawet przy zwiększaniu rozmiaru danych.

3.5. Drunk Shell Sort

parametry badania					wyniki				
algorytm	**liczebność zbioru**	**typ danych**	**dystrybucja**	**parametr pijaństwa**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
drunk shell sort	50 000	int	losowa	**10**	183,70	143	283	169	39,70
drunk shell sort	50 000	int	losowa	**25**	243,97	173	301	248,5	21,15
drunk shell sort	50 000	int	losowa	**50**	398,43	313	467	393,5	30,13
drunk shell sort	50 000	int	losowa	**75**	540,34	466	639	539,5	34,72
drunk shell sort	50 000	int	losowa	**100**	691,30	623	779	683,5	41,55



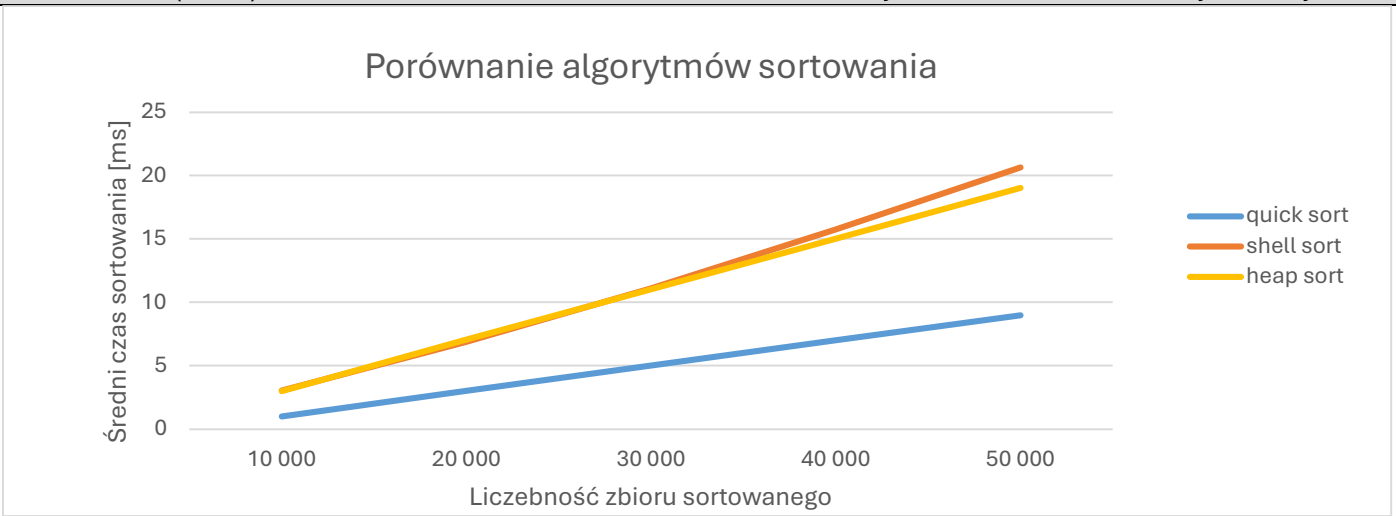
To badanie było trochę inne bo dotyczyło wpływu parametru pijaństwa na czas działania algorytmu Drunk Shell Sort.

Wyniki pokazały wyraźny wzrost średniego czasu sortowania wraz ze wzrostem parametru pijaństwa: od około 184 ms przy pijaństwie 10, do ponad 691 ms przy pijaństwie 100. Wzrost ten jest niemal liniowy, co wynika bezpośrednio ze sposobu implementacji – opóźnienie dodawane w trakcie działania algorytmu jest liniowo zależne od parametru pijaństwa. Sama logika sortowania pozostała niezmieniona i przebiega tak samo jak w zwykłym Shell Sorcie z ciągiem Knutha.

Porównując normalny Shell Sort (ok. 21 ms) do Drunk Shell Sort (powyżej 180 ms nawet przy najniższym pijaństwie), widać, że wprowadzenie błędów radykalnie zmniejsza efektywność algorytmu.

3.6. Podsumowanie

algorytm	średni czas dla 50 000 elementów [ms]	wzrost czasów w zależności od liczebności zbiorów	podsumowanie
Insertion Sort	6374,80	Kwadratowy	Wolny, nadający się tylko do niewielkich zbiorów danych
Heap Sort	19,03	Liniowy	Solidny, stabilny
Quick Sort (mediana)	8,98	Liniowy	Stabilny i piekielnie szybki
Shell Sort (knuth)	20,64	Liniowy	Solidny, stabilny



Wyniki pokazały, że Quick Sort uzyskuje najlepsze rezultaty. Jego średni czas działania rośnie liniowo wraz ze wzrostem liczby elementów w zbiorze. Heap Sort również pokazał liniowy wzrost czasu wykonania, czasy były około dwukrotnie wyższe niż dla Quick Sorta. Shell Sort z odstępami wyliczanymi według sekwencji Knutha także rośnie liniowo, jednak był minimalnie wolniejszy od Heap Sorta, zwłaszcza przy większych zbiorach.

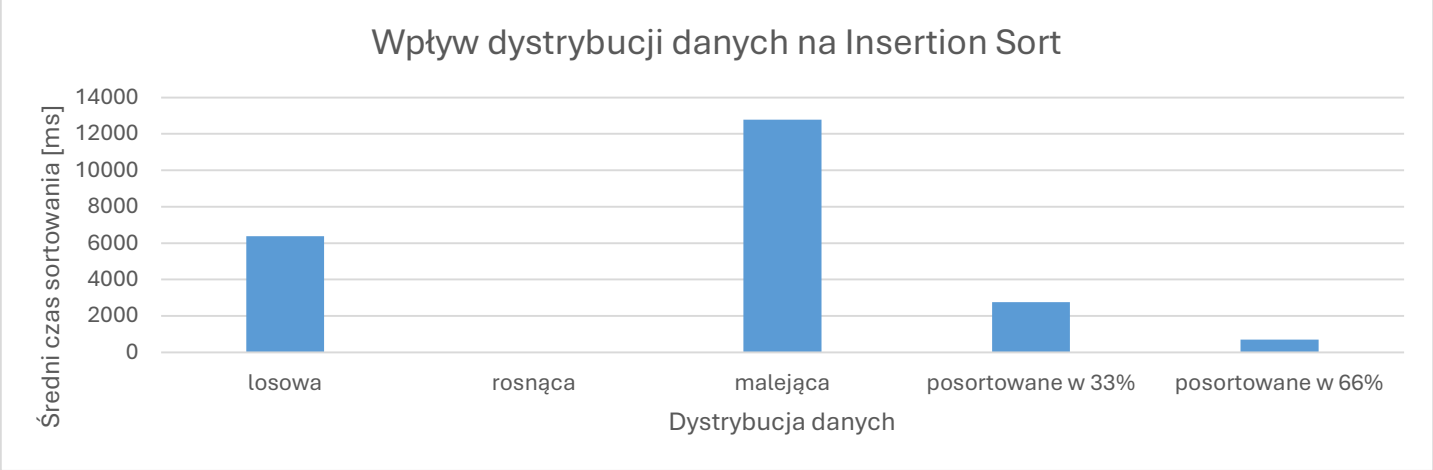
Insertion Sort wypadł wyraźnie najgorzej, czas sortowania 50 000 elementów wyniósł ponad 6300 milisekund, co było kilka rzędów wielkości gorszym wynikiem od innych algorytmów. Z tego względu nie został umieszczony na wspólnym wykresie, aby nie zniekształcać skali i nie odbierać czytelności pozostałym wynikom.

4. Badanie 2 – Wpływ początkowego rozkładu danych na czas sortowania

W tym badaniu analizowany jest wpływ początkowego ułożenia danych na efektywność sortowania. Testowano dane całkowicie losowe, rosnące, malejące oraz częściowo posortowane w 33% i 66%. Badanie ma na celu pokazanie, które algorytmy są wrażliwe na wstępną strukturę danych.

4.1. Insertion Sort

parametry badania				wyniki				
algorytm	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
insertion sort	50 000	int	**losowa**	6381,30	6298	6587	6379	41,13
insertion sort	50 000	int	**rosnąca**	0,00	0	0	0	0,00
insertion sort	50 000	int	**malejąca**	12767,44	12635	13187	12768	76,64
insertion sort	50 000	int	**posortowane w 33%**	2755,11	2715	2847	2755	19,59
insertion sort	50 000	int	**posortowane w 66%**	710,41	698	721	711	5,24



Najszybsze sortowanie miało miejsce dla zbioru wstępnie posortowanego rosnąco, czas wykonania wyniósł dokładnie 0 ms. Stało się tak dlatego, że w implementacji algorytmu nie dochodziło wtedy praktycznie do żadnych przesunięć ani zamian elementów, każdy kolejny element znajdował się już na właściwym miejscu, więc wewnętrzna pętla wykonywała się minimalną liczbę razy.

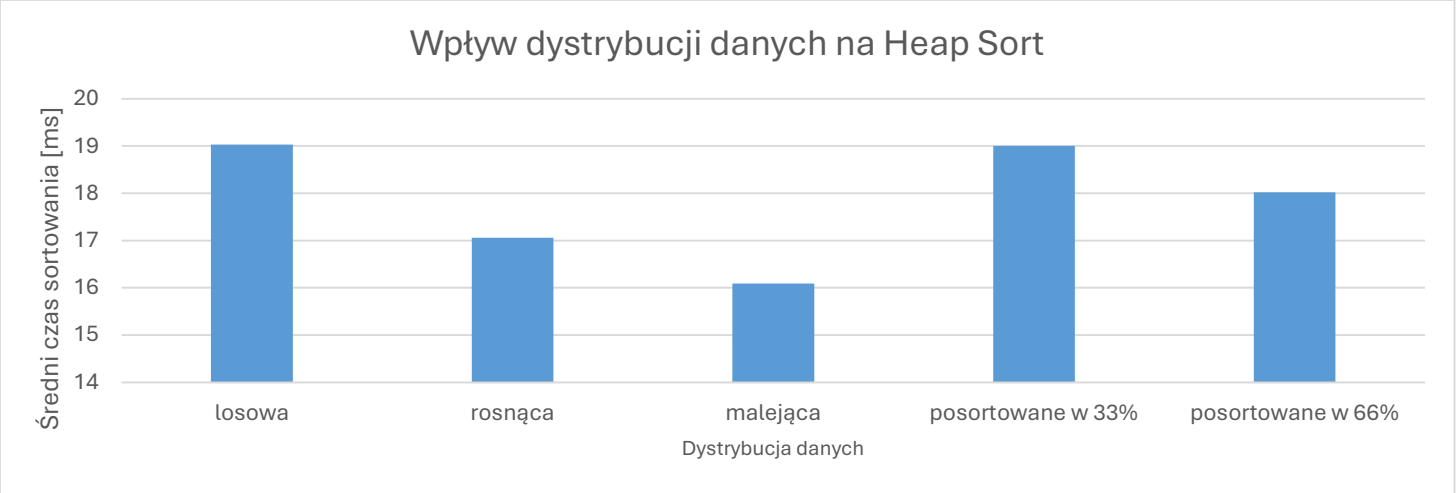
Po przeciwnej stronie znalazł się przypadek zbioru posortowanego malejąco. Tutaj Insertion Sort osiągnął dramatycznie wysoki czas działania, średnio 12767 ms. Każdy kolejny element musiał zostać przesunięty na sam początek listy, co powodowało maksymalną liczbę przesunięć i porównań na każdym etapie.

Dla zbioru całkowicie losowego czas sortowania wyniósł około 6381 ms, co było wynikiem pośrednim między dwoma skrajnościami. Jeszcze ciekawiej wyglądały wyniki dla danych częściowo uporządkowanych: przy 33% elementów ustawionych rosnąco średni czas spadł do około 2755 ms, a przy 66%, do zaledwie 710 ms. Widać tu wyraźnie, że nawet umiarkowany stopień uporządkowania danych znacząco wpływa na przyspieszenie działania Insertion Sorta.

Podsumowując, badanie bardzo dobitnie pokazuje, że efektywność Insertion Sort jest silnie uzależniona od stopnia uporządkowania danych wejściowych. Algorytm radzi sobie świetnie w przypadku danych już uporządkowanych, natomiast dramatycznie zwalnia, gdy dane są ułożone odwrotnie.

4.2. Heap Sort

parametry badania				wyniki				
algorytm	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
heap sort	50 000	int	**losowa**	19,03	19	20	19	0,17
heap sort	50 000	int	**rosnąca**	17,06	17	18	17	0,24
heap sort	50 000	int	**malejąca**	16,09	16	17	16	0,29
heap sort	50 000	int	**posortowane w 33%**	19,00	19	19	19	0,00
heap sort	50 000	int	**posortowane w 66%**	18,02	18	19	18	0,14



W przypadku Heap Sorta, analiza wpływu początkowego rozkładu danych na czas działania przyniosła zupełnie inny obraz niż w przypadku Insertion Sorta. Ten algorytm wykazał bardzo niewielką wrażliwość na uporządkowanie danych wejściowych.

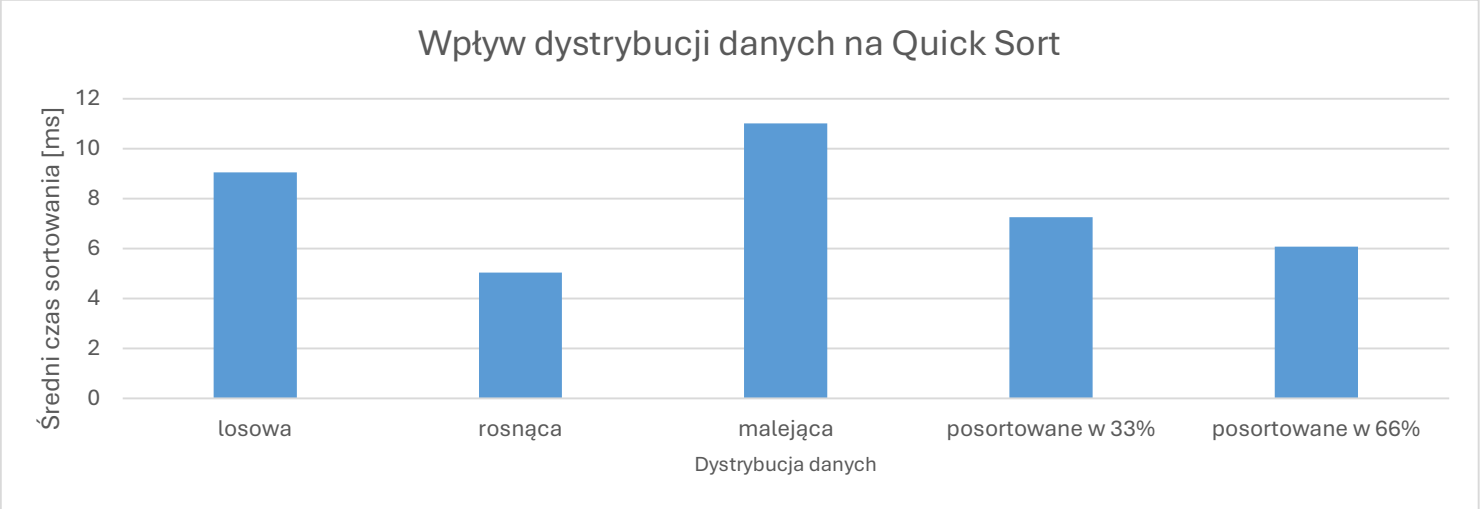
Dla danych całkowicie losowych średni czas sortowania wyniósł 19,03 ms. Dane wstępnie uporządkowane rosnąco lub malejąco pozwoliły na uzyskanie nawet nieco krótszych czasów, odpowiednio 17,06 ms oraz 16,09 ms. Różnice te są jednak bardzo niewielkie i w praktyce mogłyby być wręcz pomijalne.

Podobne rezultaty uzyskano dla danych częściowo uporządkowanych: 19,00 ms przy 33% oraz 18,02 ms przy 66% posortowanych elementów. Można więc stwierdzić, że Heap Sort działa niezależnie od początkowego ułożenia danych.

Wszystko to wynika bezpośrednio z natury algorytmu Heap Sort. W procesie sortowania algorytm i tak tworzy kopiec, co wymusza reorganizację całego zbioru niezależnie od jego początkowego stanu. Zarówno w przypadku danych losowych, jak i już uporządkowanych, kluczowe operacje, budowanie kopca i rekonstrukcja po każdym usunięciu elementu, zachodzą w ten sam sposób. W przeciwieństwie do Insertion Sorta, Heap Sort nie zyskuje więc praktycznie nic na tym, że dane są uporządkowane, ale też nie traci wydajności w przypadku danych nieuporządkowanych czy nawet odwrotnie posortowanych.

4.3. Quick Sort

parametry badania					wyniki				
algorytm	**pivot**	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
quick sort	mediana	50 000	int	**losowa**	9,05	8	17	9	0,82
quick sort	mediana	50 000	int	**rosnąca**	5,04	5	6	5	0,20
quick sort	mediana	50 000	int	**malejąca**	11,01	11	12	11	0,10
quick sort	mediana	50 000	int	**posortowane w 33%**	7,26	7	8	7	0,44
quick sort	mediana	50 000	int	**posortowane w 66%**	6,08	6	7	6	0,27



W przypadku Quick Sorta wyniki były ciekawe i pokazały wyraźną wrażliwość algorytmu na początkowe uporządkowanie danych, choć nie tak wielką jak w przypadku Insertion Sorta.

Dla danych całkowicie losowych średni czas sortowania wyniósł 9,05 ms. Co jednak interesujące, dla danych już posortowanych rosnąco, Quick Sort osiągnął dużo lepszy wynik, tylko 5,04 ms. Dane uporządkowane w 66% także zostały posortowane szybciej (6,08 ms), a częściowe uporządkowanie w 33% dało wynik pośredni równy 7,26 ms.

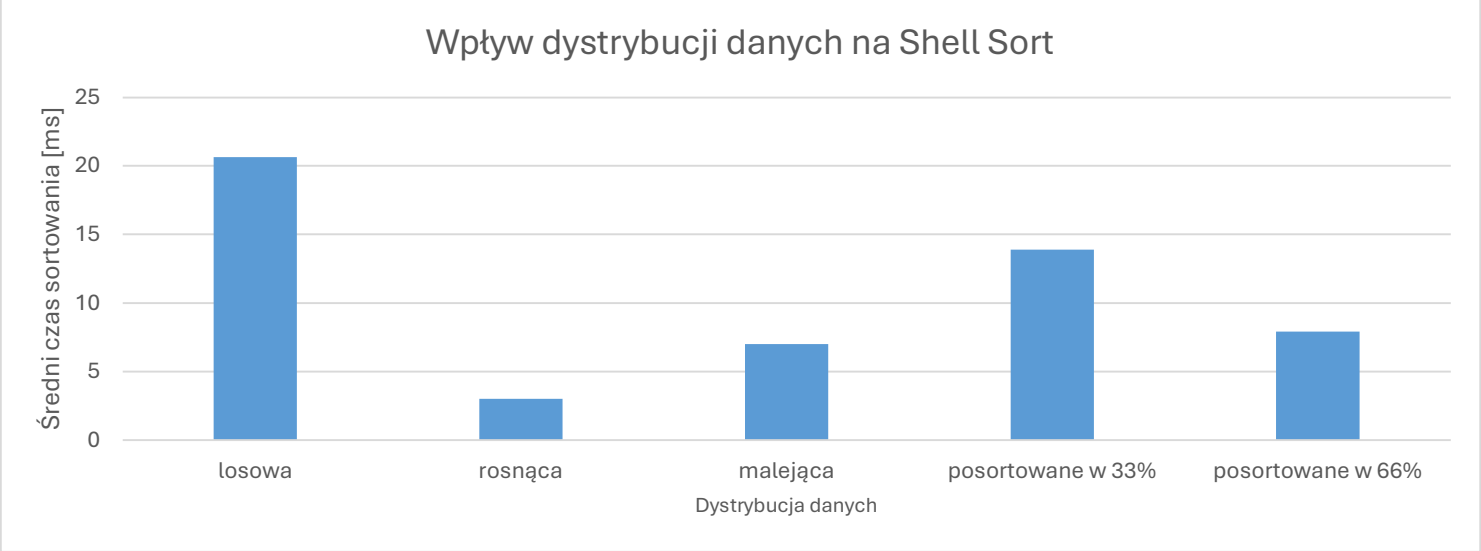
Najdłuższe czasy pojawiły się przy danych uporządkowanych malejąco, średnio 11,01 ms. Choć wzrost względem danych losowych nie był gigantyczny, to jednak zauważalny.

Takie zachowanie wynika z natury Quick Sorta i sposobu wyboru pivota. W przypadku danych uporządkowanych rosnąco, wybieranie środkowego elementu prowadzi do bardzo dobrych, zrównoważonych podziałów, dzięki czemu drzewo rekurencji jest płytkie i sortowanie bardzo szybkie. Gdy dane są uporządkowane malejąco, podziały są mniej optymalne, a głębokość rekurencji rośnie, co wpływa na wydłużenie całkowitego czasu działania. Przy danych losowych sytuacja jest bardziej zrównoważona, dlatego czas plasuje się pośrodku.

Warto przypomnieć, że wybór pivota jako mediana z trzech i tak znacznie minimalizuje te różnice między czasami dla poszczególnych dystrybucji, jak zostało to pokazane w badaniu 0.

4.4. Shell Sort

parametry badania					wyniki				
algorytm	**odstęp**	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
shell sort	knuth	50 000	int	**losowa**	20,65	19	23	21	0,66
shell sort	knuth	50 000	int	**rosnąca**	3,00	3	3	3	0,00
shell sort	knuth	50 000	int	**malejąca**	7,00	7	7	7	0,00
shell sort	knuth	50 000	int	**posortowane w 33%**	13,88	13	15	14	0,48
shell sort	knuth	50 000	int	**posortowane w 66%**	7,91	7	8	8	0,29



W przypadku Shell Sorta z odstępami według sekwencji Knutha wyniki były bardzo ciekawe i pokazały, że algorytm ten dość dobrze radzi sobie z danymi o różnym stopniu uporządkowania. Dla całkowicie losowych danych średni czas sortowania wyniósł 20,65 ms, co jest wynikiem umiarkowanym na tle innych algorytmów.

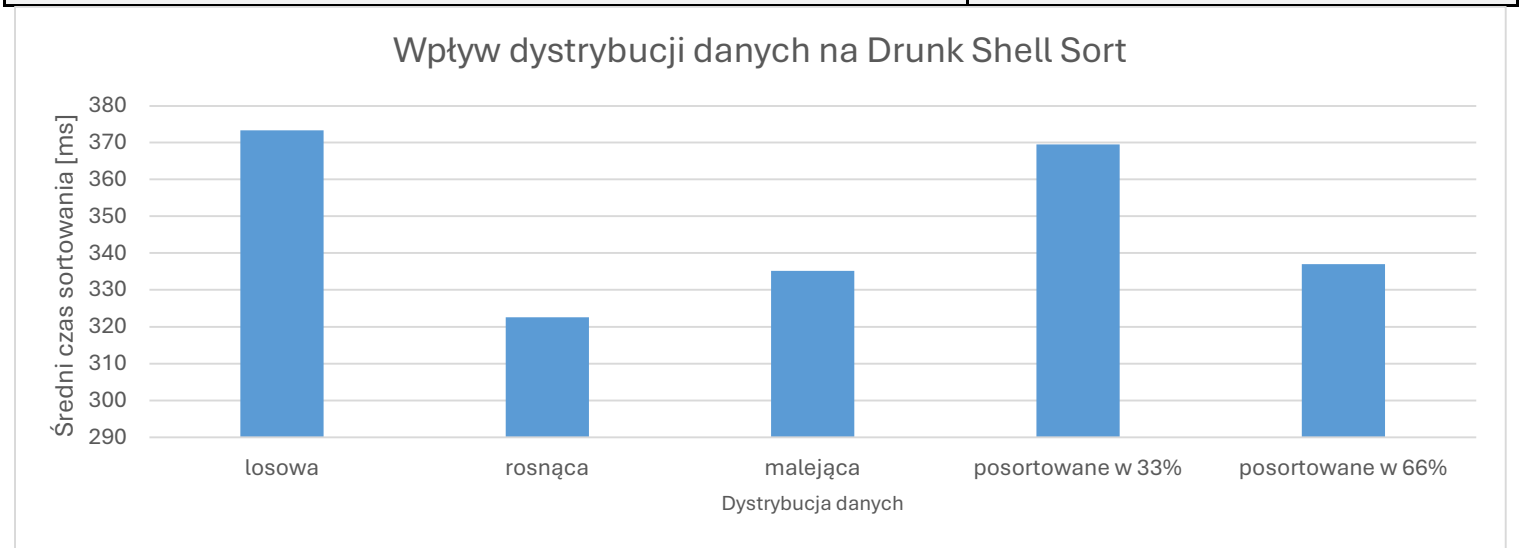
Gdy dane były już posortowane rosnąco, Shell Sort osiągał niemal natychmiastowy czas działania, średnio 3,00 ms. To normalne, bo w przypadku danych niemal idealnych, jego struktura działania pozwala bardzo szybko potwierdzić, że elementy są już na swoich miejscach. Dla danych malejących wynik był nieco wyższy, 7,00 ms, jednak nadal pozostawał niski w porównaniu do sytuacji z całkowicie losowym ułożeniem.

W przypadku częściowo uporządkowanych danych widoczne było także stopniowe skrócenie czasu sortowania. Przy 33% posortowaniu średni czas wyniósł 13,88 ms, natomiast przy 66%, tylko 7,91 ms, zbliżając się już do wyników dla danych całkowicie uporządkowanych.

Taka tendencja wynika bezpośrednio z charakterystyki działania Shell Sorta. W miarę jak dane są coraz bardziej uporządkowane, kolejne przejścia z coraz mniejszymi odstępami wykonują coraz mniej realnych przestawień. Algorytm potrafi więc bardzo szybko "dokończyć" sortowanie zbioru, który jest już częściowo lub prawie całkowicie poprawny.

4.5. Drunk Shell Sort

parametry badania						wyniki				
algorytm	**parametr pijaństwa**	**odstęp**	**liczebność zbioru**	**typ danych**	**dystrybucja**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
drunk shell sort	50	knuth	50 000	int	**losowa**	373,28	309	467	368,5	37,18
drunk shell sort	50	knuth	50 000	int	**rosnąca**	322,55	301	410	313	21,89
drunk shell sort	50	knuth	50 000	int	**malejąca**	335,15	304	442	316	35,33
drunk shell sort	50	knuth	50 000	int	**posortowane w 33%**	369,49	315	449	362,5	32,28
drunk shell sort	50	knuth	50 000	int	**posortowane w 66%**	336,95	303	450	325	33,38



Drunk Shell Sort, wykorzystujący odstępy według sekwencji Knutha oraz parametr pijaństwa ustawiony na 50, wykazał wyraźnie większe czasy działania w porównaniu do klasycznego Shell Sorta. Średni czas sortowania dla całkowicie losowego zbioru wyniósł 373,28 ms, a rozrzut wyników był zauważalny, od 309 do 467 ms.

Dla danych posortowanych rosnąco oraz malejąco czasy były nieco krótsze, odpowiednio 322,55 ms oraz 335,15 ms, ale nadal wielokrotnie wyższe niż w przypadku klasycznego odpowiednika. Nawet dla zbiorów częściowo uporządkowanych czasy pozostawały w okolicach 330–370 ms, co pokazuje, że poziom uporządkowania danych nie wpływał tak znacząco na przyspieszenie działania, jak w klasycznym Shell Sorcie.

Wszystko to wynika bezpośrednio z charakterystyki implementacji, po każdym przejściu z danym odstępem algorytm wprowadza opóźnienie zależne od ustawionego poziomu pijaństwa. Dlatego nawet jeśli algorytm szybko posortuje dane, to na koniec i tak musi odczekać swój narzucony czas, a jako że czas czekania jest dłuższy niż sortowania to różnice między dystrybucjami się trochę zacierają.

4.6. Podsumowanie

Wyniki jednoznacznie pokazały, że algorytmy takie jak Insertion Sort oraz Shell Sort znacząco przyspieszają w przypadku danych wstępnie uporządkowanych. Insertion Sort wręcz natychmiast kończył działanie na danych rosnących (czas 0 ms), co wynika z jego mechanizmu – brak konieczności przesuwania elementów. Będzie on więc idealnym wyborem do sortowania danych posortowanych (XD). Z kolei dane malejące były dla Insertion Sorta scenariuszem najgorszym możliwym, prowadząc do czasu prawie dwukrotnie dłuższego niż przy losowym układzie.

Quick Sort, chociaż w teorii działa optymalnie na losowych danych, również zauważalnie przyspieszał, gdy dane były częściowo lub całkowicie uporządkowane rosnąco. Jednak dla danych malejących wydłużał czas działania, co jest naturalne, biorąc pod uwagę wybór pivota metodą mediany.

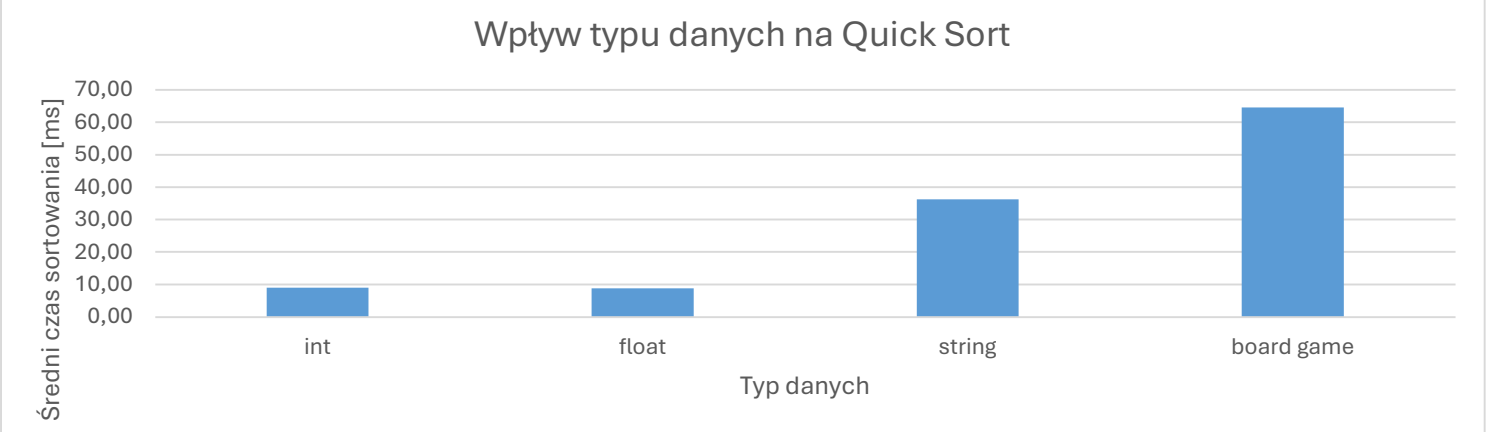
Heap Sort pokazał dużą stabilność, czas sortowania zmieniał się minimalnie w zależności od rozkładu danych, co jest zgodne z teorią, bo kopcowanie działa podobnie niezależnie od początkowej kolejności.

Drunk Shell Sort, ze względu na wbudowane opóźnienia niezależne od danych, miał czasy znacznie dłuższe i mniej wrażliwe na rzeczywiste trudności sortowania. Nawet przy idealnie posortowanych danych rosła tylko minimalnie wydajność.

5. Badanie 3 – Wpływ typu danych na czas sortowania

Celem badania jest sprawdzenie, jak typ danych wpływa na czas sortowania w wybranym algorytmie. To badanie przeprowadzone jest tylko dla Quick Sorta a danymi do sortowania jest 50 tysięcy elementów typu int, float, string oraz obiektów klasy BoardGame, wszystkie o rozkładzie losowym.

parametry badania					wyniki				
algorytm	**pivot**	**liczebność zbioru**	**dystrybucja**	**typ danych**	**średni czas [ms]**	**min [ms]**	**max [ms]**	**mediana [ms]**	**odchylenie std.**
quick sort	mediana	50 000	losowa	**int**	8,97	8	9	9	0,17
quick sort	mediana	50 000	losowa	**float**	8,79	8	10	9	0,46
quick sort	mediana	50 000	losowa	**string**	36,19	35	39	36	0,84
quick sort	mediana	50 000	losowa	**board game**	64,60	62	71	64	1,74



Wyniki wykazały, że dla typów prostych, takich jak int i float, czas sortowania był niemal identyczny, około 9 ms. Oznacza to, że dla prymitywnych typów danych koszt operacji porównania oraz kopiowania jest bardzo niski, co przekłada się na wysoką szybkość działania algorytmu.

W przypadku typu string czas sortowania wzrósł kilkukrotnie, średnio do około 36 ms. Wynika to głównie z większego kosztu operacji porównania (porównywanie ciągów znaków jest bardziej złożone niż porównywanie liczb) oraz z większego kosztu kopiowania danych podczas zamian elementów.

Najdłuższy czas sortowania zaobserwowano dla obiektów klasy BoardGame, średnio 64 ms. Przyczyną jest jeszcze wyższy koszt kopiowania i porównywania złożonych obiektów, w których może występować wiele pól (np. stringi, liczby, itp.), oraz fakt, że podczas sortowania wykonywane są dodatkowe operacje na konstruktorach kopiujących lub operatorach porównania.

Podsumowując, im bardziej złożony jest typ danych, tym dłużej trwa sortowanie, głównie ze względu na większe obciążenie operacji porównywania i przestawiania elementów.